МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Л. І. Кублій

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

ОСНОВИ АЛГОРИТМІЗАЦІЇ

Підручник

Затверджено Вченою радою КПІ ім. Ігоря Сікорського
як підручник для здобувачів ступеня бакалавра
за спеціальністю 121 «Інженерія програмного забезпечення»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензенти: *Нестеренко К. С.*, доктор технічних наук, професор, факультет кібербезпеки, комп'ютерної та програмної інженерії Національного авіаційного університету (м. Київ)

Гнатієнко Г. М.., кандидат технічних наук, факультет інформаційних технологій Київського національного університету імені Тараса Шевченка (м. Київ)

Ющук О. І., провідний програміст Atg Luther & Maelzer GmbH (м. Вертхайм, Німеччина)

Відповідальний редактор *Сегеда І. В.*, канд. екон. наук, доцент

Гриф надано Вченою радою КПІ ім. Ігоря Сікорського
(протокол № 4 від 27.06.2022 р.)

У підручнику викладено матеріал дисципліни «Алгоритми та структури даних» з основ алгоритмізації. Підручник містить теоретичний і практичний матеріал, який подається на лекціях і пропонується студентам для самостійного опрацювання, містить запитання і завдання для самоконтролю, практичні завдання і рекомендації щодо їхнього виконання, список використаної і рекомендованої літератури, предметний покажчик, додатки. Особливу увагу приділено не тільки поданню алгоритмів, а й оцінці їхньої складності, прийомам оптимізації алгоритмів, врахуванню обмежень при програмуванні. Наведено багато алгоритмів і їхню програмну реалізацію мовою С. У багатьох випадках до розгляду запропоновано по кілька алгоритмів розв'язання тієї самої задачі, які відрізняються складністю логіки, швидкістю роботи і можливостями виконання на комп'ютері.

Підручник призначений для здобувачів ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення». Буде також корисним для тих, хто вивчає основи алгоритмізації за іншою спеціальністю галузі знань 12 «Інформаційні технології» чи самостійно.

Реєстр. № П 21/22-032. Обсяг 18,15 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© Л. І. Кублій
© КПІ ім. Ігоря Сікорського, 2022

Зміст

Вступ . . . 6
1. Поняття алгоритму. Подання алгоритмів . . . 9
   1.1. Поняття алгоритму . . . 9
   1.2. Властивості алгоритмів . . . 12
   1.3. Складність алгоритмів . . . 13
   1.4. Способи подання алгоритмів . . . 19
   1.5. Приклади алгоритмів і їхнього подання . . . 30
   1.6. Найпоширеніші стратегії алгоритмізації . . . 34
   Запитання та завдання для самоконтролю . . . 36
   Практичне завдання № 1 «Основи роботи в інтегрованому середовищі Visual Studio» . . . 37
2. Типи обчислювальних процесів. Різноманітність алгоритмів . . . 39
   2.1. Типи обчислювальних процесів: лінійні, розгалужені, циклічні . . . 30
   2.2. Приклад розробки простого алгоритму . . . 43
   2.3. Різноманітність алгоритмів розв’язування задачі . . . 47
   Запитання та завдання для самоконтролю . . . 59
   Практичне завдання № 2 «Оформлення звітів засобами текстового редактора Word» . . . 59
3. Позиційні і непозиційні системи числення . . . 61
   3.1. Системи числення . . . 61
   3.2. Подання чисел у позиційних системах . . . 63
   3.3. Бінарні й небінарні коди Грея . . . 74
   3.4. Подання числових даних у комп’ютері: цілі й дійсні числа . . . 78
   3.5. Алгоритми виконання арифметичних операцій . . . 94
   Запитання та завдання для самоконтролю . . . 108
   Практичне завдання № 3 «Позиційні системи числення» . . . 110
   Практичне завдання № 4 «Робота з дійсними числами на комп’ютері» . . . 111
4. Лінійні алгоритми . . . 113
   4.1. Поняття лінійного алгоритму . . . 113
   4.2. Програмування виразів . . . 113
   4.3. Структура лінійного алгоритму . . . 125
   4.4. Приклади лінійних алгоритмів . . . 130
   4.5. Складність виразу. Оптимізація обчислень . . . 133
   Запитання та завдання для самоконтролю . . . 139
   Практичне завдання № 5 «Програмування обчислювальних алгоритмів лінійної структури» . . . 140
5. Алгоритми з розгалуженнями . . . 143
   5.1. Поняття алгоритму з розгалуженнями . . . 143
   5.2. Реалізація алгоритму з розгалуженнями . . . 143
   5.3. Цикломатична складність алгоритму . . . 147
   5.4. Приклади алгоритмів з розгалуженнями . . . 150
   5.5. Складність алгоритмів з розгалуженнями. Оптимізація виконання . . . 157
   Запитання та завдання для самоконтролю . . . 163

Практичне завдання № 6 «Розробка і реалізація алгоритмів розгалужених процесів з послідовною перевіркою умов» 163
Практичне завдання № 7 «Розробка і реалізація алгоритмів розгалужених процесів з вкладеною перевіркою умов» 165
6. Алгоритми з циклами 167
6.1. Поняття алгоритму з циклами 167
6.2. Реалізація і використання алгоритмів з циклами 168
6.3. Приклади алгоритмів з циклами 184
6.4. Складність і оптимізація циклічних алгоритмів 218
Запитання та завдання для самоконтролю 222
Практичне завдання № 8 «Проектування алгоритмів циклічної структури. Цикли з параметрами» 223
Практичне завдання № 9 «Проектування циклічних ітераційних алгоритмів» 224
Практичне завдання № 10 «Проектування алгоритмів з вкладеними циклами. Робота з дійсними числами» 226
7. Алгоритми модульної структури 229
7.1. Поняття модуля і модульного програмування 229
7.2. Проектування й програмування згори вниз і знизу вгору 232
7.3. Організація міжмодульних зв’язків 234
7.4. Функції в мові програмування С 236
7.5. Рекурсивні алгоритми 247
7.6. Складність і оптимізація алгоритмів модульної структури 266
Запитання та завдання для самоконтролю 270
Практичне завдання № 11 «Алгоритми модульної структури» 272
Практичне завдання № 12 «Рекурсивні алгоритми» 272
8. Обробка одновимірних масивів 273
8.1. Поняття масиву. Одновимірний масив 273
8.2. Основні типові алгоритми роботи з одновимірними масивами 278
8.3. Пошук елемента в масиві 290
8.4. Сортування масивів 297
8.5. Приклади алгоритмів з використанням одновимірних масивів 321
8.6. Складність і оптимізація алгоритмів обробки одновимірних масивів . . . 335
Запитання та завдання для самоконтролю 341
Практичне завдання № 13 «Алгоритми обробки одновимірних масивів» . . 342
Практичне завдання № 14 «Алгоритми сортування масивів структур» 346
9. Обробка двовимірних масивів 352
9.1. Поняття двовимірного масиву 352
9.2. Основні алгоритми роботи з двовимірними масивами 355
9.3. Сортування елементів матриці і рядків символів 369
9.4. Розріджені матриці 373
9.5. Приклади алгоритмів з використанням двовимірних масивів 377
9.6. Складність і оптимізація алгоритмів обробки двовимірних масивів . . 385
Запитання та завдання для самоконтролю 387
Практичне завдання № 15 «Алгоритми обробки двовимірних масивів» 388

10. Алгоритмізація розв'язування задач 392
10.1. Вибір алгоритму розв'язування задачі (на прикладі обчислення кількості сполук) 392
10.2. Швидке піднесення чисел до цілих степенів 407
10.3. Генерація випадкових чисел 418
10.4. Генерація кодів Грея 443
10.5. Генерація комбінаторних множин 453
10.6. Розкладання натуральних чисел на доданки і прості множники 489
Запитання та завдання для самоконтролю 501
Практичне завдання № 16 «Виконання додаткового завдання» 503
Список використаної літератури 505
Посилання на список використаної літератури за розділами 508
Список рекомендованої літератури 509
Предметний покажчик 510
Додаток А. Обхід секторів квадратної матриці (мови C і Pascal) 517
Додаток Б. Орієнтовні завдання до модульної контрольної роботи 522
Додаток В. Запитання і орієнтовні завдання до залікової контрольної роботи . . . 524

Вступ

Алгоритми використовують у всіх сферах, пов'язаних з інформаційними технологіями. Вони є основою програмування. Знання існуючих алгоритмів і їхнє застосування дає програмістам можливість швидше створювати надійні й ефективні програми для розв'язування практичних задач. Невід'ємною складовою роботи кожного програміста також є розробка нових алгоритмів.

Підготовка програмістів спрямована не тільки на надання їм певних знань і вироблення відповідних навичок застосовувати ці знання, а й на формування в них алгоритмічного мислення. Для цього, насамперед, треба проаналізувати досить велику кількість різноманітних алгоритмів, зрозуміти принципи їхньої роботи, з'ясувати умови і обмеження щодо їхнього використання, застосовувати в практичних навчальних завданнях існуючі алгоритми, навчитися адаптувати їх, модифікувати і розробляти свої.

У підручнику розглянуто поняття алгоритму, властивості алгоритмів, способи їхнього подання, найпоширеніші стратегії алгоритмізації, типи обчислювальних процесів, системи числення, подання числових даних у комп'ютері, алгоритми лінійні, з розгалуженнями і циклами, рекурсивні. Прикладами алгоритмів є як класичні, так і сучасні алгоритми, зокрема, алгоритм Евкліда знаходження найбільшого спільного дільника, решето Ератосфена пошуку простих чисел, схема Ньютона для обчислення коренів, алгоритми пошуку і сортування, генерації випадкових чисел і комбінаторних множин, алгоритми, які базуються на використанні побітових операцій, кодів Грея і «магічних» чисел. Для розв'язання деяких задач запропоновано по кілька алгоритмів, проаналізовано їхні переваги й недоліки. Акцентовано увагу на точності обчислень при роботі з дійсними числами і на наслідках арифметичного переповнення при роботі з цілими числами. Значну увагу приділено дослідженню ефективності алгоритмів і шляхам оптимізації їхньої роботи. Подані в підручнику алгоритми в основному працюють з простими структурами даних.

Матеріал підручника орієнтовано на зацікавлення студентів, активізацію їхньої самостійної роботи, вироблення практичних навичок алгоритмізації, формування професійних умінь. Велика кількість прикладів у підручнику сприяє кращому засвоєнню матеріалу, при цьому більшість алгоритмів студенти мають опрацювати самостійно.

Наведені в підручнику алгоритми належать до дуже невеликої кількості сфер застосування. Реально ж коло задач, для розв'язання яких треба розробляти і застосовувати комп'ютерні алгоритми, по суті, нескінченне. Проте розглянуті підходи, які стосуються основ алгоритмізації, можуть бути використані при розробці навіть найскладніших алгоритмів, хоч у них, швидше за все, будуть застосовані якісь свої, значно складніші алгоритми і програмістські трюки, будуть використовуватися не тільки базові й досить прості структури даних (змінні, масиви, структури-записи), а й такі структури даних, як списки, стеки, черги, хеш-таблиці, дерева, словники тощо, які потребують відповідних алгоритмів обробки.

Базовою мовою програмування при вивченні основ алгоритмізації вибрано мову С. Майже всі алгоритми, розглянуті в підручнику, програмно реалізовано і налагоджено в інтегрованому середовищі програмування Visual Studio. Кілька алгоритмів подано мовою програмування Turbo Pascal. Як правило, поряд з текстами програм розміщено скриншоти результатів їхньої роботи.

Хоч у середній школі є предмет інформатики, але, як свідчить опитування, в останні 3 роки (2019-2021) серед студентів 1 курсу спеціальності 121 «Інженерія програмного забезпечення» кафедри АПЕПС великий відсоток таких, що взагалі не вивчали тем, пов'язаних з програмуванням (відповідно за роками — 38,3 %, 19,2 %, 21,3 %), або тільки ознайомилися з мовами програмування, але самі нічого не програмували (9,9 %, 25,6 %, 6,7 %), і лише незначна кількість студентів відповіли, що вільно програмують (–, 7,7 %, –). У зв'язку з цим у підручнику виклад матеріалу починається від розгляду й аналізу алгоритмів з найпростішою логікою і поступово переходить до складніших і досить складних.

Основи алгоритмізації студенти спеціальності 121 «Інженерія програмного забезпечення» вивчають на першому курсі в першому семестрі. Паралельно з циклу професійної підготовки вони також вивчають «Основи програмування»; у зв'язку з цим при поданні матеріалу увага не акцентується на синтаксисі мови С, яку в підручнику використано як базову. Проте в розділах 4-9 як додатковий матеріал подано дуже стисло дрібнішим шрифтом найсуттєвіші відомості з мови С.

Підручник містить теоретичний і практичний матеріал, який подається на лекціях і пропонується студентам для самостійного опрацювання, містить запитання і завдання для самоконтролю, практичні завдання і рекомендації до їхнього виконання, список використаних джерел, список рекомендованої літератури, предметний покажчик, додатки (зокрема, орієнтовні завдання до модульної контрольної роботи, теоретичні запитання й завдання до заліку). Підручник призначений для здобувачів ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення», проте він буде також корисним для тих, хто вивчає програмування за іншою IT-спеціальністю чи самостійно.

Підручник пройшов апробацію — матеріал викладався студентам протягом двох років.

Особливу вдячність за слушні поради й зауваження під час роботи над підручником висловлюю Ющуку Олегу Івановичу, провідному програмістові американської фірми Atg Luther & Maelzer GmbH (м. Вертхайм, Німеччина), авторові відомої в широких колах програмістів системи налагодження програм OllyDbg (http://www.ollydbg.de/, https://uk.wikipedia.org/wiki/OllyDbg).

Лариса Іванівна Кублій, кандидат технічних наук, доцент
e-mail: kublii_l_i@ukr.net
тел.: 063-71-91-231, 097-558-27-17
Telegram: 0637191231

1. Поняття алгоритму. Подання алгоритмів

Поняття алгоритму. Властивості алгоритмів. Складність алгоритмів. Способи подання алгоритмів. Приклади алгоритмів і їхнього подання. Найпоширеніші стратегії алгоритмізації.

1.1. Поняття алгоритму

У повсякденному житті люди постійно користуються алгоритмами[1], навіть не задумуючись над цим. Це різноманітні правила, інструкції, рецепти тощо. Наприклад, щоб заварити чай, ми кип'ятимо воду, в чашку насипаємо заварку, заливаємо її кип'ятком, настоюємо 5 хвилин. Хоч можна спочатку в чашку насипати заварки, а потім починати кип'ятити воду — але так буде довше. Отже, послідовність дій має значення. Звичайно, кожна з вказаних дій, залежно від виконавця і умов виконання алгоритму, може потребува́ти певної деталізації. Так, щоб закип'ятити воду, треба взяти чайник чи металеву миску (а перед тим треба взяти миску з тумбочки, відчинивши дверцята) треба налити води з крана чи з пляшки (а перед тим відкрити кран чи відкрутити корок пляшки), увімкнути газ чи електроплиту і т. д. Як видно з наведеного прикладу, хтось кип'ятить воду в чайнику, а хтось у мисці. Але хтось може кип'ятити воду й кип'ятильником зразу в чашці — тоді поданий вище алгоритм трохи зміниться (не треба наливати кип'яток у чашку). Тобто, щоб заварити чай можна скористатися дещо різними алгоритмами — все залежить від певних умов.

Наявність алгоритму дає можливість розв'язувати задачу формально, механічно виконуючи команди алгоритму у вказаній послідовності. Тому виконувати алгоритм може не тільки людина, а й технічний засіб, зокрема — комп'ютер.

Процес розв'язування будь-якої задачі за допомогою комп'ютера має кілька етапів: постановка задачі, формалізація постановки задачі (побудова моделі), побудова працюючого алгоритму (розробка алгоритму і перевірка його правильності), оцінка ефективності алгоритму, реалізація алгоритму у

[1] Слово алгоритм походить від імені перського математика Аль-Хорезмі (9 століття), який розробив правила виконання чотирьох арифметичних дій (додавання, віднімання, множення й ділення) над багатозначними десятковими числами — спочатку тільки ці правила називали алгоритмами.

вигляді програми, перевірка програми, розробка документації, супровід програми.

Постановка задачі виникає, коли в матеріальному світі чи в світі ідей виникає певна ситуація. Часом при цьому доводиться виробити цілу систему нових понять, але частіше ситуація описується в термінах існуючих понять.

Якщо задача має бути розв'язана математично, то постановка задачі має бути формалізована, тобто треба побудувати *математичну модель*, виражену в термінах понять, які мають строго певні властивості і які перебувають у строго певних відношеннях. Часом може бути, що задача з самого початку виникла як математична, тобто вже в формальній постановці. Коли кажуть, що задача поставлена, то вважають, що відомі не тільки характеристики існуючої ситуації (вхідні дані), а й вимоги до кінцевого бажаного стану (вихідні дані).

Порівняно рідко шлях, який веде від постановки задачі до її розв'язання, буває відразу зрозумілим. Як правило, цей шлях намічається саме внаслідок аналізу формальної постановки задачі і виявлення нових властивостей і відношень між поняттями і об'єктами, про які в ній ідеться. Шлях розв'язання задачі, тобто послідовність дій, які треба виконати, щоб від початкової постановки задачі прийти до бажаного нового стану, називають *алгоритмом* розв'язання задачі. Створення навіть найпростішого алгоритму є творчим процесом.

Існують різні визначення алгоритму.

Алгоритм — точний набір інструкцій (правил, процедур, команд), які описують порядок дій певного виконавця для розв'язання певної задачі чи певного класу задач за скінченну кількість кроків.

Алгоритм — точний і зрозумілий опис послідовності дій над заданими об'єктами, який після скінченної кількості кроків приводить виконавця до досягнення вказаної мети чи розв'язання поставленої задачі.

Алгоритм (за визначенням А. Н. Колмогорова) — будь-яка система обчислень, що виконуються за строго певними правилами, яка після деякої кількості кроків приводить до розв'язання поставленої задачі.

Алгоритм (за визначенням А. А. Маркова) — точне розпорядження, що визначає обчислювальний процес, який іде від варійованих початкових даних до шуканого результату.

При використанні комп'ютерів основною метою є обробка даних різних типів (числових, символьних, графічних, звукових, відеоданих, різноманітних сигналів тощо). Тому тут можна дати таке визначення: *алгоритм* — це сукупність приписів, які визначають процес перетворення вхідних даних у вихідні.

Програма теж є алгоритмом з певним рівнем деталізації, який розробляється для певної архітектури комп'ютера за допомогою алгоритмічної мови або машинних команд.

Одним із найважчих етапів створення алгоритму є *перевірка* його *правильності* — при застосуванні алгоритму до різноманітних вхідних даних має бути отримано потрібні результати. Найрозповсюдженіша процедура перевірки правильності алгоритму (як і програми) — обґрунтування правомірності й перевірка правильності виконання кожного з кроків на наборі тестів, які охоплюють якомога більшу ймовірну множину вхідних і вихідних даних. Тобто, після розробки алгоритму його треба протестувати на різних вхідних даних.

Аналіз *ефективності алгоритму* полягає в оцінці *часової складності* (ще кажуть: трудомісткості, обчислювальної складності) — часових витрат на його виконання, і *ємнісної складності* (ще кажуть: просторової складності) — обсягу використовуваної оперативної пам'яті залежно від обсягу вхідних даних. Фактично, оцінка часової складності алгоритму зводиться до кількості основних операцій, які виконуються в алгоритмі. При цьому різні операції потребують різного часу виконання і тому часто оцінюють складність алгоритмів щодо певних операцій (наприклад, порівнянь, перестановок, проходів циклу тощо).

Від правильності і міри складності алгоритму залежить ефективність наступних етапів розв'язання задачі за допомогою комп'ютера — програмування алгоритму, перевірка правильності програми, розробка документації і

особливо супровід програми, тобто її покращення, оптимізація і виправлення помилок під час використання.

Таким чином, при підготовці задач до розв'язування на комп'ютері найбільш відповідальним, трудомістким і тривалим є етап алгоритмізації.

1.2. Властивості алгоритмів

Характерними властивостями алгоритмів є такі:

1) *дискретність* (роздільність, перервчастість) — алгоритм складається з окремих кроків (послідовності закінчених дій) і перехід до наступного кроку можливий лише після закінчення виконання попереднього;

2) *визначеність* (однозначність) — кожен алгоритм будують з врахуванням певного виконавця — наприклад, комп'ютера, програміста чи звичайного користувача. Для розв'язання задачі з використанням заданого алгоритму треба, щоб кожна команда завжди однозначно сприймалася і точно виконувалася, наступний крок роботи має однозначно визначатися результатами попередніх кроків; завдяки цій властивості виконання алгоритму має механічний характер і не потребує ніяких додаткових вказівок;

3) *результативність* (скінченність) — виконання алгоритму має приводити до одержання конкретного результату (цей результат може бути й негативним: задача не має розв'язку, алгоритм не можна застосувати тощо) при будь-яких вхідних даних за скінченну кількість кроків (і не просто скінченну, а досить обмежену, розумну кількість кроків);

4) *масовість* — за допомогою алгоритму можна розв'язувати не одну конкретну задачу, а багато однотипних задач певного класу і робити це неодноразово, що дає можливість збільшити практичну цінність алгоритму.

Дональд Кнут у книзі «Мистецтво програмування. Основні алгоритми» [19] також виділяє такі властивості, як наявність вхідних (можливо й нульових; беруться з певного набору і визначаються до початку роботи алгоритму чи визначаються динамічно під час його роботи) і вихідних (результуючих, яких може бути одне чи кілька) даних, а також ефективність (алгоритм вважають ефектив-

ним, якщо його можна точно виконати протягом скінченного часу за допомогою олівця й паперу).

Алгоритми можуть бути детермінованими й недетермінованими. Детермінований алгоритм для тих самих вхідних даних видає ті ж самі результати і передбачає єдиний шлях їхньої обробки; у такому алгоритмі не використовується жодна випадковість. Недетермінований алгоритм залежно від деяких випадкових чинників передбачає кілька шляхів обробки тих самих вхідних даних і може привести до різних результатів. Проте, якщо в недермінованому алгоритмі випадковість вважати частиною вхідних даних, то алгоритм стає детермінованим. У даному курсі будемо розглядати детерміновані алгоритми.

Алгоритм (детермінований) є однозначною процедурою розв'язування задачі, але ту саму задачу можна розв'язати, використовуючи різні алгоритми. Одні з них будуть ефективнішими чи простішими, а інші — ні. Серед багатьох алгоритмів треба вибрати найкращий, враховуючи умови його застосування. Таким чином, виникає таке поняття, як аналіз алгоритмів.

1.3. Складність алгоритмів

Складність алгоритмів оцінюють за часом виконання або за обсягом використовуваної пам'яті. У більшості випадків складність залежить від обсягу вхідних даних. Так, масив з 10 елементів буде оброблятися швидше, ніж із 10000. При цьому точний час залежить від процесора, типу даних, мови програмування та інших параметрів.

Для оцінки складності алгоритмів використовують таке поняття, як *асимптотична складність*, і його пов'язують з обробкою дуже великих наборів даних.

Нехай n — кількість елементів вхідних даних, функція $f(n)$ описує верхню межу максимальної кількості основних операцій (додавання, множення, порівняння, присвоювання, перестановок тощо), які виконуються в алгоритмі. Якщо функція $f(n)$ зростає не швидше, ніж деякий багаточлен (поліном) $P(n)$, то кажуть, що алгоритм має поліноміальну складність, або *алгоритм є*

поліноміальним, інакше алгоритм є *неполіноміальним*. Поліноміальні алгоритми для великих обсягів даних, характерних для практичних задач, можуть виконуватися на сучасних комп'ютерах, а неполіноміальні (це, як правило, алгоритми повного перебору) через великі потреби часу не можуть виконатися повністю.

Оскільки в більшості випадків точний вигляд функції $f(n)$ записати складно (навіть, коли алгоритм деталізовано і добре досліджено, одержати точний вигляд функції досить важко), то використовують її асимптотичну оцінку $O(g(n))$.

Функцію $O(g(n))$ називають *асимптотичною складністю алгоритму*, якщо:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = const.$$

Наприклад, якщо певний алгоритм виконує $5n^3 + 7n$ операцій, то асимптотична складність цього алгоритму дорівнює $O(n^3)$ (відповідно алгоритм називають кубічним), оскільки $\lim\limits_{n\to\infty}\frac{5n^3+7n}{n^3} = 5$ є константою.

Формально $O(g(n))$ означає, що час роботи алгоритму або обсяг займаної пам'яті росте залежно від обсягу вхідних даних не швидше, ніж деяка константа, помножена на $g(n)$.

На практиці найчастіше розглядають такі складності алгоритмів:

$O(1)$ — константна складність алгоритму (сталий час виконання алгоритму) не залежить від обсягу вхідних даних (наприклад, при вибіркових обстеженнях успішності студентів треба одержати інформацію про студентів з номерами 5, 12, 28, … у загальному списку факультету; визначення парності числа, поданого у двійковій системі числення); одиниця у формулі не означає, що алгоритм виконується за одну операцію або потребує дуже мало часу — він може виконуватися і мілісекунду, і день, і рік чи навіть більше;

$O(n)$ — лінійна складність (наприклад, пошук максимального елемента в масиві; додавання чи віднімання двох *n*-розрядних чисел); подвоєння обсягу даних збільшує вдвічі час обробки;

$O(n^2)$ — квадратична складність (наприклад, сортування масиву методом бульбашки; множення чисел у стовпчик чи за алгоритмом Аль-Хорезмі; ділення чисел; алгоритм Дейкстри знаходження найкоротшого шляху між вершинами зваженого графа); подвоєння обсягу даних збільшує час обробки в 4 рази; якщо один елемент обробляється одну мілісекунду (10^{-3} с), то одна тисяча елементів буде оброблятися майже 17 хвилин, а один мільйон — майже 32 роки;

$O(n^3)$ — кубічна складність (наприклад, звичайне множення матриць; алгоритм Уоршалла знаходження транзитивного замикання відношення); подвоєння обсягу даних збільшує час обробки у 8 разів;

$O(n^d)$ при $0 < d < 1$ — сублінійна складність — менша від лінійної (наприклад, пошук у *k*-вимірному дереві);

$O(\log n)$ — логарифмічна складність — час виконання алгоритму зростає логарифмічно зі збільшенням обсягу даних (алгоритм працює, скорочуючи обсяг оброблюваних даних на певний постійний коефіцієнт на кожній ітерації; наприклад, бінарний пошук — діленням впорядкованого масиву елементів навпіл; алгоритм швидкого піднесення числа до цілого степеня); подвоєння обсягу даних збільшує час обробки на величину $\log 2$; в аналізі алгоритмів за замовчуванням використовують логарифм за основою 2;

$O(\log\log n)$ — дуже повільне зростання складності (алгоритм на кожній ітерації обробляє масив обсягу, рівного кореню квадратному з обсягу даних попередньої ітерації, наприклад, пошук в дереві Ван Емде Боаса[2], алгоритм Хопкрофта-Фортуни знаходження найближчої пари точок в масиві двовимірних точок); подвоєння обсягу даних майже не відображається на збільшенні часу обробки;

---

[2] Кореневий вузол дерева має $\sqrt{n}$ вказівників, розділяючи діапазон 0..*n* на $\sqrt{n}$ сегментів, кожен з яких містить діапазон приблизно $\sqrt{n}$ елементів. Потім кожне з цих піддерев внутрішньо поділяється на $\sqrt{\sqrt{n}}$ піддерев, кожне з яких містить приблизно $\sqrt{\sqrt{n}}$ елементів. Обхід дерева починається з кореня, визначається належність елемента певному піддереву і далі у відповідному поддереві алгоритм працює рекурсивно.

$O(n\log n)$ — лінійно-логарифмічна (лінеаритмічна) складність (наприклад, сортування масиву злиттям; швидке перетворення Фур'є); подвоєння обсягу даних збільшує час обробки трохи більше ніж вдвічі;

$O(k^n)$ — експоненційна складність (наприклад, розв'язання задачі комівояжера за допомогою динамічного програмування; семантико-залежні задачі обробки природномовного тексту); збільшення обсягу даних на 1 призводить до *k*-кратного збільшення необхідного часу; подвоєння обсягу даних призводить до квадрату часу; прикладами експоненційної складності є $O(2^n)$, $O(n^{\log n})$, $O(n^{\sqrt{n}})$ тощо;

$O(n!)$ — факторіальна складність (наприклад, розв'язання задачі комівояжера повним перебором); збільшення обсягу даних на 1 призводить до $(n+1)$-кратного збільшення необхідного часу.

Задачі, для яких розроблено алгоритми з константною чи логарифмічною часовою складністю, розв'язуються досить швидко.

Більшість використовуваних алгоритмів розв'язання задач мають поліноміальну складність $O(n^\alpha)$, де $\alpha \geq 1$ (α може бути і дробовим). Задачі, які можна розв'язати алгоритмами з поліноміальною часовою складністю належать до класу P — класу задач поліноміальної складності. Їхній розв'язок для великих обсягів вхідних даних можна знайти за прийнятний час, тобто задачі є практично розв'язними. Поліноміальними також є алгоритми з лінійно-логарифмічною $O(n\log n)$ і меншими (наприклад, $O(\log n)$) складностями.

Алгоритми з експоненційною і факторіальною складністю можуть розв'язувати задачі при дуже незначних обсягах вхідних даних.

Коли говорять про асимптотичну складність алгоритму, то мають на увазі найгірший випадок. Так, часова складність $O(n)$ означає, що в алгоритмі виконується не більше ніж $c \cdot n$ дій, де c — константний множник. Часом вказують найкращу і середню складність. Наприклад: алгоритм сортування бульбашкою з індикатором перестановок у найкращому випадку має часову складність $\Omega(n)$, у середньому — $\theta(n^2)$, у найгіршому — $O(n^2)$; швидке сортування — $\Omega(n\log n)$, $\theta(n\log n)$, $O(n^2)$.

Незважаючи на те, що характеристикою асимптотичної складності алгоритму є порядок росту $O(g(n))$, великий порядок для одного алгоритму може мати менший константний множник *c*, ніж малий порядок складності іншого алгоритму. У зв'язку з цим алгоритм з великим порядком складності може бути для задач з невеликим *n* більш швидким, ніж алгоритм з невеликим порядком складності. У книзі [1, с. 14] подано приклад: якщо для розв'язування тої самої задачі існує кілька алгоритмів і алгоритм A_1 потребує виконання $1000n$ операцій, A_2 — $100n\log_2 n$, A_3 — $10n^2$, A_4 — 2^n операцій, то найкращим буде A_4 для $2 \le n \le 9$, A_3 для $10 \le n \le 58$, A_2 для $59 \le n \le 1024$ і A_1 для $n \ge 1024$. При розв'язуванні практичних задач константний множник *c* має велике значення. Асимптотична оцінка є, так би мовити, першим наближенням до визначення складності алгоритму.

У таблиці 1.1 подано залежність часу виконання алгоритму від обсягу даних і складності алгоритму, якщо одна операція виконується одну наносекунду (тобто виконується один мільярд операцій за секунду):

Таблиця 1.1. Час виконання алгоритмів

Обсяг даних / Складність алгоритму	10	20	30	40	50
n	0,00001 с	0,00002 с	0,00003 с	0,00004 с	0,00005 с
n^2	0,0001 с	0,0004 с	0,0009 с	0,001 с	0,0025 с
n^3	0,001 с	0,008 с	0,027 с	0,064 с	0,125 с
$\log_2 n$	$3{,}32\cdot10^{-6}$ с	$4{,}32\cdot10^{-6}$ с	$4{,}91\cdot10^{-6}$ с	$5{,}32\cdot10^{-6}$ с	$5{,}64\cdot10^{-6}$ с
$n\cdot\log_2 n$	$3{,}32\cdot10^{-5}$ с	$8{,}64\cdot10^{-5}$ с	$1{,}47\cdot10^{-4}$ с	$2{,}13\cdot10^{-4}$ с	$2{,}82\cdot10^{-4}$ с
$\log_2\log_2 n$	$1{,}73\cdot10^{-6}$ с	$2{,}11\cdot10^{-6}$ с	$2{,}29\cdot10^{-6}$ с	$2{,}41\cdot10^{-6}$ с	$2{,}5\cdot10^{-6}$ с
3^n	0,059 с	58,11 хв	6,53 років	386,59 століть	$2{,}28\cdot10^{8}$ століть

На рисунку 1.1 відображено залежності кількості операцій від обсягу даних, які обробляються, при різних часових складностях алгоритмів:

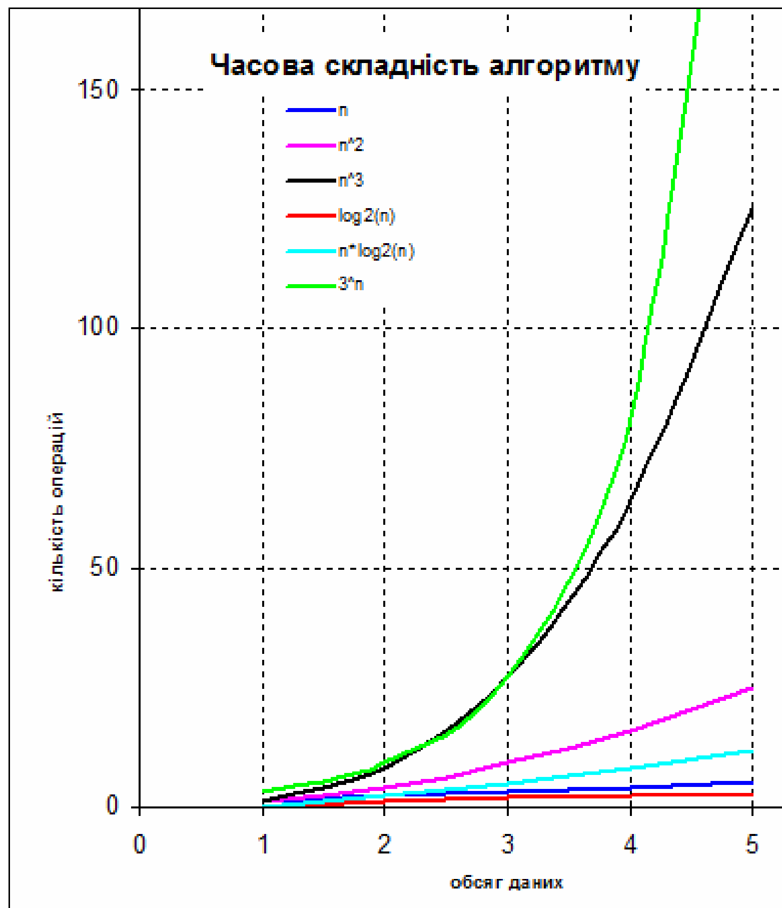


Рисунок 1.1. Залежність кількості операцій від обсягу даних

Аналогічно до часової складності можна оцінювати ємнісну складність алгоритмів. При цьому може бути, що зі збільшенням обсягу даних потрібний обсяг пам'яті значно збільшується, але алгоритм виконується швидко або навпаки — обсяг пам'яті збільшується несуттєво, але час виконання значно збільшується. Тому при виборі конкретного алгоритму треба враховувати вимоги й умови щодо його використання. Наприклад, алгоритм, який виконується вдесятеро швидше, але використовує вдесятеро більше пам'яті може працювати з великою кількістю даних на серверному комп'ютері з великим обсягом пам'яті, але на звичайних комп'ютерах з невеликим обсягом пам'яті він виконуватися не буде. Відомі алгоритми сортування щодо додаткової пам'яті мають такі складності: сортування вибіркою (з пошуком мінімуму чи максимуму), метод бульбашки, сортування простими вставками — $O(1)$; сортування злиттям — $O(n)$; швидке сортування — $O(\log n)$, але в алгоритмі виконується до $O(n)$ рекурсивних викликів, що може призвести до переповнення стека.

1.4. Способи подання алгоритмів

Існують різні способи подання алгоритмів. Алгоритм, який розробляється для подальшого виконання на комп'ютері можна, наприклад, описати словесно, подати графічно або запрограмувати, використовуючи якусь алгоритмічну мову чи машинні коди. Можуть бути й інші способи подання алгоритмів. Наприклад, в пристрої Блеза Паскаля[3], арифмометрі Готфріда Вільгельма Лейбніца[4] алгоритми виконання арифметичних дій були закладені в самому механізмі; в машині Еніак[5] алгоритм задавали за допомогою спеціального під'єднання проводів, що займало багато годин або навіть днів.

Словесне подання алгоритму. При словесному поданні алгоритму використовують по суті звичайну мову, але з ретельно підібраними словами так, щоб у записові алгоритму не було зайвих слів, синонімів (щоб не виникало неоднозначностей) тощо, доповнену математичними позначеннями і деякими спеціальними домовленостями (про знак присвоювання, про нумерацію кроків алгоритму тощо). Таке подання легко сприймати, від нього легко перейти до кодової форми мовою програмування. Але деякі алгоритми можуть бути дуже складними, щоб зразу перейти від словесного опису до програми, тому може бути потрібною подальша деталізація.

Розглянемо приклади словесного подання алгоритмів.

---

[3] У 1641 році француз Блез Паскаль сконструював на основі зубчастих коліс пристрій для механічного виконання додавання і віднімання 6-розрядних (пізніше й 8-розрядних) десяткових чисел. Машини Паскаля були першими у світі калькуляторами, які випускалися серійно. Pascal Blaise (1623-1662) — французький математик, механік, фізик і філософ. Мова програмування Pascal, розроблена в 1970 році швейцарцем Ніклаусом Віртом (Niklaus Wirth), названа в його честь.

[4] У 1673 році німецький математик Лейбніц сконструював арифмометр, який виконував 4 арифметичні дії (додавання, віднімання, множення й ділення) над 12-розрядними десятковими числами (до зубчастих коліс він додав ступінчатий валик, який давав можливість множити й ділити). Особливо широке застосування арифмометри мали в 19 столітті, використовувалися вони й у 20 столітті, але всі дії на арифмометрі й запис проміжних результатів виконували люди. Лейбніц також досліджував двійкову систему числення, яка стала основою при створенні комп'ютерів уже в 20 столітті. Leibniz Gottfried Wilhelm (1646-1716) — німецький математик, фізик, філософ, винахідник, юрист, історик, мовознавець.

[5] У 1943-1946 роках в США в Пенсильванському університеті було створено найбільшу електронну цифрову обчислювальну машину ENIAC (Electronic Numerical Integrator and Automatic Calculator) на основі електронних ламп (близько 18 тисяч; довжина її — 26 м, висота — 6 м, вага — 30 тонн), яка працювала в десятковій системі числення. Цю машину використовували для розрахунку балістичних ракет, розрахунків для атомної енергетики, вивчення космічних променів, конструювання аеродинамічних труб, конструювання водневої бомби.

Приклад 1.4.1. Знайти абсолютне значення числа і вивести його.

К1. Ввести число a.

К2. Якщо a<0, то a= –a.

К3. Вивести a.

Приклад 1.4.2. Знайти більше серед двох чисел.

К1. Ввести числа a і b.

К2. Якщо a>b, то вивести a, інакше вивести b.

Приклад 1.4.3. Розробити алгоритм для обчислення площі трикутника за формулою Герона $s = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$, де $p = \frac{a+b+c}{2}$, якщо: а) значення *a, b, c* задані коректно; б) треба перевірити, чи існує трикутник.

а) *К1.* Ввести числа a, b, c.

К2. Обчислити p=(a+b+c)/2.

К3. Обчислити $\mathrm{s} = \sqrt{\mathrm{p} \cdot (\mathrm{p}-\mathrm{a}) \cdot (\mathrm{p}-\mathrm{b}) \cdot (\mathrm{p}-\mathrm{c})}$.

К4. Вивести s.

б) *К1*. Ввести числа a, b, c.

К2. Якщо a<=0, або b <=0, або c<=0, то видати повідомлення, що є недодатне число. Перейти на К7.

К3. Якщо a + b <= c, або a + c <= b, або b + c <= a, то видати повідомлення, що трикутник не можна побудувати. Перейти на К7.

К4. Обчислити p=(a+b+c)/2.

К5. Обчислити $\mathrm{s} = \sqrt{\mathrm{p} \cdot (\mathrm{p}-\mathrm{a}) \cdot (\mathrm{p}-\mathrm{b}) \cdot (\mathrm{p}-\mathrm{c})}$.

К6. Вивести s.

К7. Кінець.

Прості алгоритми можна описати словесно і без покрокової деталізації.

Графічне подання алгоритму. Алгоритм можна подати за допомогою блок-схеми — схеми зв’язаних між собою лініями і стрілками блоків. Лінії і стрілки позначають послідовність обчислень, потоки даних або керування. Кожен блок відповідає одному крокові алгоритму і містить опис відповідної дії. Така схема може бути деталізованою або досить загальною за-

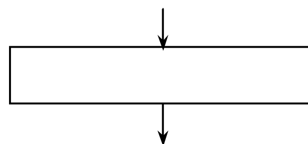
лежно від складності алгоритму. Блоки узагальненої схеми можуть у подальшому деталізуватися.

Графічне подання алгоритму дає можливість відслідковувати всі шляхи переходів і завдяки наочності полегшує процес написання програми. Також блок-схеми використовують для пояснення роботи вже готових алгоритмів, при цьому блоками беруть блоки алгоритмів, робота яких не потребує пояснень.

Блок-схеми будують відповідно до міждержавного стандарту ISO 5807-85 (ГОСТ 19.701-90) «Єдина система програмної документації. Схеми алгоритмів, програм, даних і систем».

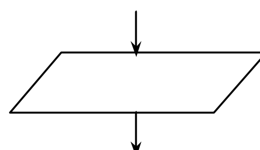
Блок-схема має містити по одному блоку початку (входу) й кінця (виходу) алгоритму:

Обчислювальні (або арифметичні) блоки зображають прямокутниками, від яких виходить тільки одна стрілка, що показує, до якого блоку треба перейти після закінчення обчислень, вказаних у даному блоці.



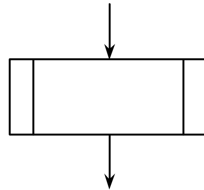
Кожен такий блок вказує на виконання певної операції чи групи операцій.

Блоки введення-виведення даних, якщо носій даних не визначений чи не має значення, позначають паралелограмами:

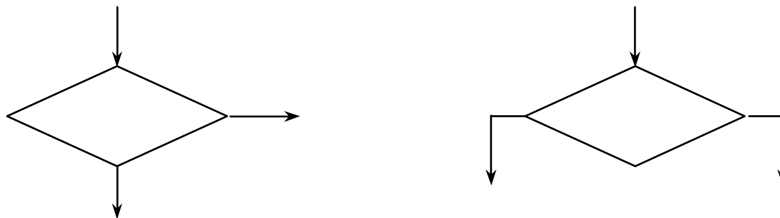


Якщо треба конкретизувати джерело введення чи виведення інформації, то існує ряд спеціальних блоків. Наприклад, для позначення клавіатури і дисплею відповідно використовують такі блоки:

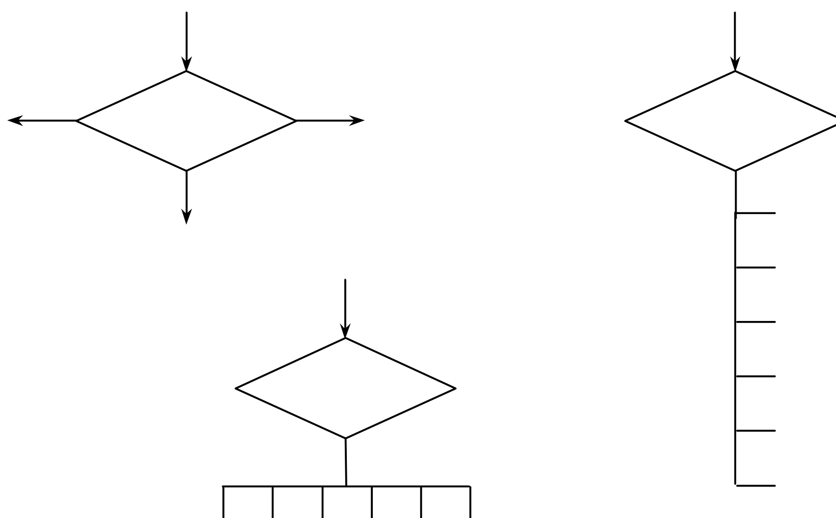
Блок підпрограми (модуля) має вигляд:



Логічні блоки (блоки перевірки, прийняття рішення) зображають ромбами, в яких записано деяку умову, яка може або виконатися, або не виконатися при поточних значеннях змінних. Із логічного блока виходять дві стрілки, помічені словами «так» і «ні» (або відповідно знаками + і –). Одна стрілка показує перехід у випадку виконання умови, інша — коли умова не виконується:

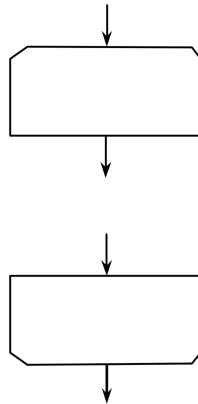


Блоки вибору, залежно від кількості можливих варіантів (3 чи більше), позначають так:



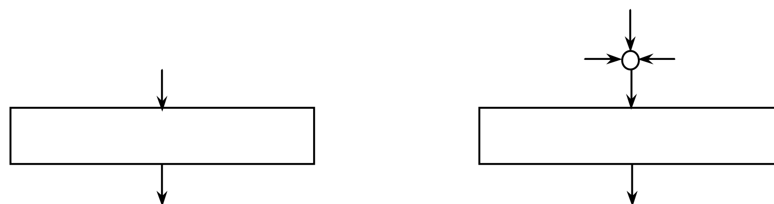
Кожен вихід із такого блоку має бути підписаний відповідним значенням умови.

Початок і кінець циклу позначають двома симетричними блоками (між якими можуть розміщуватися будь-які інші блоки):



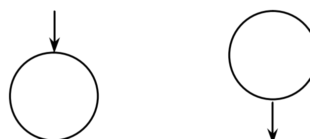
У цих блоках записують ім'я змінної циклу, передумову чи післяумову.

До кожного блоку, крім блоку початку, має вести принаймні одна стрілка, але їх може бути й кілька. Також із кожного блоку, крім блоку кінця, має виходити принаймні одна стрілка. Круг позначає злиття двох і більше шляхів переходу:



Лінії і стрілки відображають потоки даних чи керування. Напрямок потоку згори вниз і зліва направо є стандартним і в цьому випадку стрілки не обов'язкові. Якщо потік має напрямок, відмінний від стандартного, то стрілки обов'язкові.

Блок з'єднання (круг) використовують для обриву лінії і продовження її в іншому місці. При цьому в круги вписують той самий символ, наприклад, число чи букву.



Символ коментаря має вигляд квадратної дужки, біля якої записують коментар. Символ коментаря з’єднується з блоком пунктирною лінією або пунктирна лінія обводить групу блоків і з’єднує цю групу з символом коментаря.

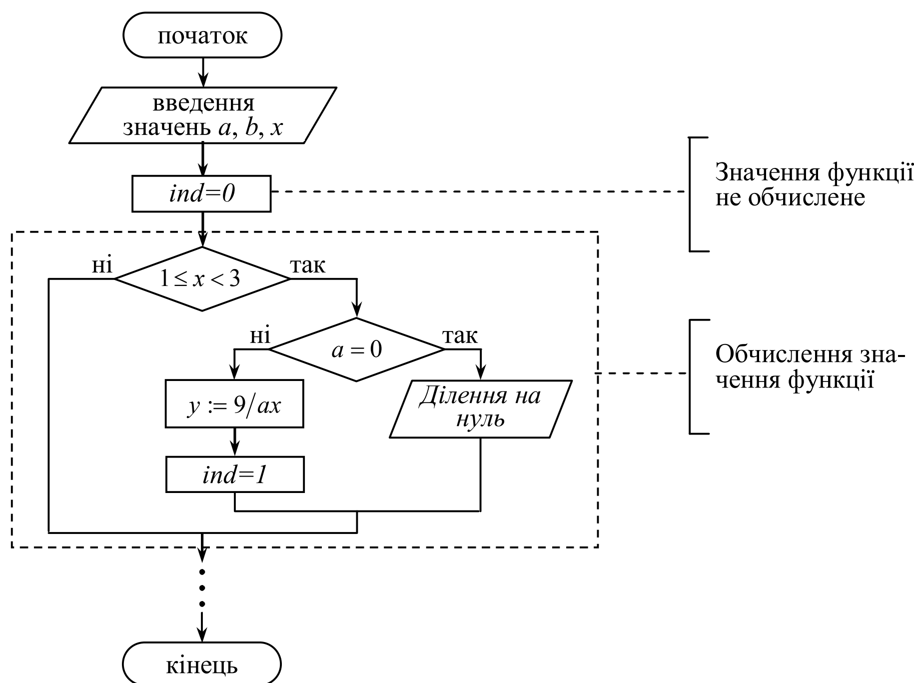
Для позначення пропуску повторів поміщають три крапки, розриваючи з’єднувальні лінії.

У схемах блоки мають розміщуватися рівномірно, має бути розумною довжина ліній, мінімальною кількість довгих ліній. Не повинні змінюватися кути й інші параметри, які впливають на форму блоків. За можливості блоки мають бути одного розміру. Можна, але не бажано, блоки відображати дзеркально. Бажаною є горизонтальна орієнтація блоків. Усі лінії і границі блоків повинні мати одну товщину, шрифти — один розмір і гарнітуру.

Приклад 1.4.4. Для обчислення значення функції, заданої логічною залежністю

$$f(x) = \begin{cases} \dfrac{9}{ax} & \text{при } x \in [1,3), \\ \left|ax^2 + x + b\right| & \text{при } x = 3, \end{cases}$$

фрагмент блок-схеми має вигляд:



Програмне подання алгоритму. Щоб певний алгоритм виконувався комп’ютером, його треба запрограмувати, тобто перетворити в програму.

Програма — це логічно впорядкована послідовність команд, потрібних для керування комп'ютером, тобто виконання ним конкретних операцій.

Текст програми містить повний, закінчений і детальний опис алгоритму мовою програмування. Програма не псується від часу й повторного використання. Запрограмовані знання і вміння можна накопичувати, зберігати в бібліотеках і застосовувати в нових програмах.

Опис алгоритму на етапі алгоритмізації розкриває загальну схему розв'язання задачі. На етапі програмування алгоритми записують за допомогою спеціальних алгоритмічних мов, які розробляються для цієї мети. Тексти програм транслюються в машинний код. *Алгоритмічна мова* — це набір символів із заданими правилами утворення конструкцій, за допомогою яких описується порядок виконання алгоритму. *Машинний код* (машинна мова) — набір команд, поданих двійковими числами, які виконуються безпосередньо центральним процесором. Кожен процесор має власний машинний код.

Спосіб подання програми залежить від використовуваної мови програмування. Крім того, той самий алгоритм різні програмісти програмно реалізують по-різному.

Приклад 1.4.5. Подамо мовою C програму обчислення значення функції, заданої логічною залежністю (фрагмент блок-схеми подано в прикладі 1.4.4):

$$f(x) = \begin{cases} \dfrac{9}{ax} & \text{при } x \in [1;3), \\ \left|ax^2 + x + b\right| & \text{при } x = 3. \end{cases}$$

У програмі на екран виводиться вигляд функції, з клавіатури вводяться значення змінної x і параметрів a і b. Залежно від значення змінної x обчислюється значення функції $f(x)$ на першому чи на другому (в точці) проміжках, а також враховано, що значення x може потрапити поза заданий проміжок $[1;3]$. Вважається, що функція визначена на заданому проміжку, а поза ним не визначена. На першому проміжку $[1;3)$ функція може мати особливість (ділення на нуль при $a = 0$), тоді програма видає про це повідомлення. Якщо $a \neq 0$, то обчислюється значення функції. Якщо значення x

потрапляє в другий проміжок (точка $x=3$), то особливостей функція не має, і тому зразу обчислюється її значення.

Видача повідомлень про помилки здійснюється при їхньому виявленні. Результат обчислення значення функції $f(x)$ виводиться в кінці програми. Для цього використано змінну-індикатор, яка вказує на те, чи значення функції було обчислене, і яка набуває значення 0 (значення функції не обчислено і виводити його не треба) чи 1 (значення функції обчислено і його треба вивести).

Крім того, щоб вікно видачі результатів залишалося на екрані, треба якимось чином його затримати — у даній програмі це зроблено за допомогою функції system звернення до операційної системи і передачі їй відповідного параметра pause.

Програма має вигляд:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main() {
/* Обчислення значення функції, заданої логічною залежністю */
  double x,    // значення аргументу
         y,    // значення функції
         a,b; // параметри функції
  int ind;    // індикатор друку значення функції
  system("chcp 1251 & cls");   /* українізація інтерфейсу в
                                  MS Windows */
  /* Виведення вигляду функції на екран */
  printf("f(x)=9/(ax)          при x [1,3)\n");
  printf("     abs(ax^2+x+b)   при x=3\n");
  /* Введення вхідних даних */
  printf ("\nВведіть значення a ");
  scanf("%lf",&a);    // printf (" %f",a);
  printf ("Введіть значення b ");
  scanf("%lf",&b);
  printf ("Введіть значення x ");
```

```
  scanf("%lf",&x);
  /* Обчислення значення функції */
  ind=0;
  /* Перший проміжок. Є особливість */
  if (x>=1 && x<3)
    if (a==0)  // ОДЗ порушено
      printf("Ділення на нуль");
    else {  // ОДЗ не порушено
      y=9/(a*x);
      ind=1;
    }
  /* Другий проміжок. Особливостей нема */
  if (x==3) {
    y=fabs(a*x*x+x+b);
    ind=1;
  }
  /* Значення x задано поза проміжком */
  if ((x<1) || (x>3))
    printf("Функція не визначена");
  /* Друк результату обчислення */
  if (ind)
    printf("f(%4.2f)=%6.3f", x, y);
  printf("\n\n");
  /* Затримка вікна виведення */
  system("pause");
  return 0;
}
```

Для подання алгоритму у вигляді програми певною мовою програмування треба знати синтаксис цієї мови, тобто правила запису команд, і правила використання лексичних одиниць мови. У мовах програмування лексичними одиницями є ключові слова, ідентифікатори, константи, знаки операцій, коментарі; також існують розділювачі (кома, дужки, крапка з комою тощо).

У програмі мовою C ключові слова (наприклад, int, double, if, else, for тощо) можуть бути використані тільки як ключові слова, тобто ніколи не допускається їхнє використання як імен змінних, констант, функцій чи міток. Але, наприклад, мова Fortran зарезервованих слів не має.

Ідентифікатор — це ім'я, яке програміст надає об'єктові — змінній, константі, функції, мітці. Для зручності читання програмного коду бажано, щоб ідентифікатор відображав призначення відповідного об'єкта, був осмисленим. Так, якщо змінна зберігає значення кута, то її можна назвати alfa, beta тощо; якщо значення кількості, то n, m; якщо вказує номери об'єктів, то i, j, k; якщо матриці, то A, B, C, як це роблять в математиці. Для лічильників циклів, як правило, використовують імена i, j, для вказівників — p, q, для рядків символів — s, t. Проте для компілятора це не має ніякого значення. Це важливо для читабельності програми. Глобальним змінним доцільно надавати осмислені інформативні імена, оскільки вони можуть використовуватися в будь-якому місці програми, а для локальних — за можливості, короткі.

Якщо в програмі є не більше 20 ідентифікаторів, то питання вибору їхніх імен не надто принципове, але, коли їх є кілька сотень, а то й тисяч, то це питання стає досить важливим.

При написанні програми треба дотримуватися якогось певного стилю щодо вибору імен. Існують різні корпоративні домовленості і традиції.

Так, серед програмістів існують писані й неписані правила щодо застосування знака підкреслення і великих букв у ідентифікаторах. Якщо поняття, якому відповідає змінна, зручно описати не одним, а кількома словами, то можна скористатися підкресленням (caps_lock — класичний стиль), або кожне слово починати з великої букви (CapsLock — сучасний стиль), або кожне слово, крім першого, починати з великої букви (capsLock). Існує навіть угода про правила створення імен — так звана угорська нотація (запропонував її співробітник Microsoft угорець за національністю; у 1990-х роках була внутрішнім стандартом для програмістів Microsoft; її перевага — полегшує читання програм і зменшує ймовірність

неправильного використання змінної; її недоліком є те, що при зміні типу змінної назва втрачає зміст і не вказує на тип змінної), відповідно до якої кожне слово, яке входить в ідентифікатор, починається з великої літери, а спочатку ставиться префікс, який відповідає типу величини, наприклад, iMaxLength. Крім того, не рекомендується починати ідентифікатори з символу підкреслення, оскільки вони можуть збігтися з іменами системних функцій, констант або змінних; це також знижує мобільність програми. Макровизначення (імена препроцесорних констант) прийнято подавати великими буквами (#define MAX 25, #define BEGIN { , #define END }). Також прийнято позначати змінні — малими буквами, константи — великими буквами. Вибравши певний стиль, треба його дотримуватися, щоб не заплутувати себе.

У програмах треба надавати перевагу іменованим константам, значення яких у подальшому легко можна змінити, порівняно зі звичайними числовими константами.

Стандарт C99 мови програмування C дозволяє описувати змінні в будь-якому місці програми, але правила хорошого стилю програмування передбачають опис всіх змінних на початку програми — у так званій декларативній частині.

Ознакою хорошого стилю програмування є використання коментарів, оскільки коментарі полегшують розуміння програми будь-яким програмістом. Коментарі можна писати кирилицею (українською мовою). З метою подальшого легкого розуміння програми, її модифікації і зменшення в ній помилок треба опис кожної глобальної змінної, функції, типу даних, константи, а також локальної змінної супроводжувати коротким коментарем. Коментованість програми є одним із показників зрозумілості коду. Нормальним мінімальним показником вважається 10 % від загальної кількості рядків тексту програми, а найкращим — 17-25 %; за іншою шкалою мінімальній коментованості відповідає 10-15 %, хорошій — 16-25 %, найкращій — 26-35 %.

Стиль подання коду програми має бути таким, щоб на екрані розмістити максимальну кількість коду і щоб код був читабельним. Читабельності кодові надає структуризація. Відступи структуризації рекомендується брати від 2 до 8 позицій, але, щоб код поміщався на екрані. Відступ бажано робити пробілами, а не табулюванням, оскільки в іншому програмному середовищі табулювання може давати інші відступи, а пробіли не зміняться.

1.5. Приклади алгоритмів і їхнього подання

Розглянемо кілька прикладів з використанням різного подання алгоритмів.

Приклад 1.5.1. Подати алгоритм впорядкування двох чисел за зростанням, використовуючи словесний опис, блок-схему і програмну реалізацію.

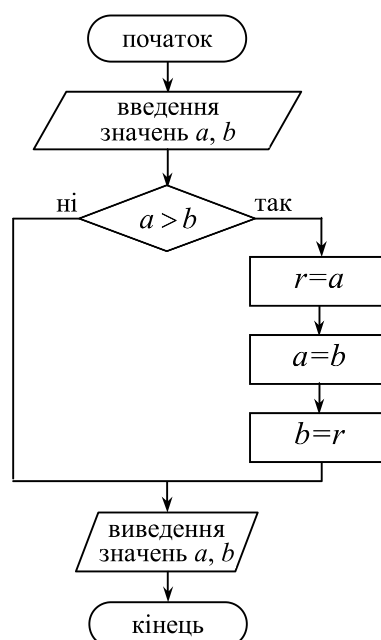
Розв'язання. Словесне подання алгоритму:

К1. Ввести числа a, b.

К2. Якщо a>b, то r=a, a=b, b=r.

К3. Вивести a, b.

Блок-схема алгоритму:



Програмна реалізація мовою С:

```
#include <stdio.h>
```

```
#include <stdlib.h>
/* Впорядкування двох чисел за зростанням */
int main() {
  float a, b, r;
  system("chcp 1251 & cls");
  /* Введення вхідних даних */
  printf ("Введіть значення a і b  ");
  scanf("%f%f", &a, &b);
  /* впорядкування даних */
  if (a>b) {
    r=a;
    a=b;
    b=r;
  }
  /* Друк результату обчислення */
  printf("%f  %f", a, b);
  printf("\n\n");
  /* Затримка вікна виведення */
  system("pause");
  return 0;
}
```

```
Введіть значення a і b  3 8
3.000000  8.000000
```

```
Введіть значення a і b  7 4
4.000000  7.000000
```

```
Введіть значення a і b  5 5
5.000000  5.000000
```

Програма мовою Turbo Pascal:

```
program vporiadkuvannia;
{ Впорядкування двох чисел за зростанням }
uses crt;
  var a, b, r: real;
begin
  clrscr;
  { Введення вхідних даних }
  writeln('Vvedit znachennia a i b  ');
  readln(a, b);
  { Впорядкування даних }
  if (a>b) then
```

```
    begin
      r:=a;
      a:=b;
      b:=r;
    end;
  { Друк результату обчислення }
  writeln(a, b);
  { Затримка вікна виведення }
  readln;
end.
```

Приклад 1.5.2. Дія, яка розглядається в деяких алгоритмах як елементарна, може бути сама описана у вигляді алгоритму. Так, знаходження остачі від ділення числа a ($a \geq 0$) на число b ($b > 0$), яка є, наприклад, операцією мови програмування C (a%b) чи Pascal (mod), можна подати таким алгоритмом:

K1. Ввести цілі числа a, b.

K2. Якщо a<b, перейти на К4.

K3. a=a–b. Перейти на К2.

K4. Вивести остачу від ділення a.

Приклад 1.5.3. Розробити алгоритм обчислення значень квадратичного багаточлена з довільними цілими коефіцієнтами *a*, *b*, *c* при цілих значеннях аргументу *x* на заданому проміжку $[x_0, x_n]$ з заданим цілим кроком *h*. Подати алгоритм різними способами.

Розв'язання. Відповідно до завдання треба розробити алгоритм табулювання функції $y = ax^2 + bx + c$ на проміжку $[x_0, x_n]$ з кроком *h*.

Словесне подання алгоритму:

K1. Ввести цілі числа a, b, c.

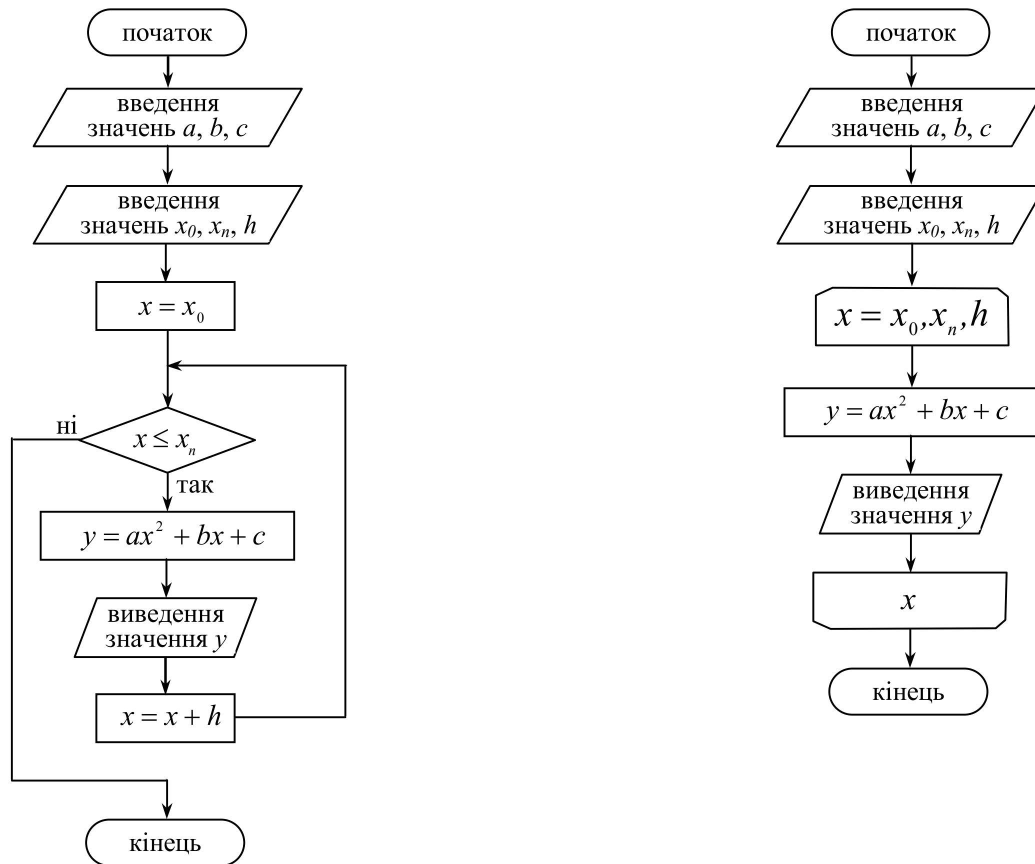
K2. Ввести цілі числа x_0, x_n, h.

K3. $x=x_0$.

K4. Якщо $x \leq x_n$, то $y = ax^2 + bx + c$, вивести значення y, збільшити значення x на h. Перейти на К4.

K5. Кінець.

Два варіанти блок-схеми алгоритму — з використанням умовного блоку й переходу (відповідає циклу з передумовою) і з використанням блоків початку й кінця циклу:



Реалізація алгоритму мовою програмування С з використанням оператора циклу з параметром:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  /* Обчислення значення квадратичного
     багаточлена ax2+bx+c */
  int x, y, a, b, c;
  int x0, xn, h;
  system("chcp 1251 & cls");
  /* Введення вхідних даних */
  printf ("   Обчислення значення ax2+bx+c\n");
  printf ("Введіть значення a, b, c  ");
  scanf("%d%d%d", &a, &b, &c);
```

```
  printf ("Введіть значення x0, xn, h  ");
  scanf("%d%d%d", &x0, &xn, &h);
  /* обчислення й виведення результату */
  for (x=x0; x<=xn; x+=h) {
    y=a*x*x+b*x+c;   // оптимальніше: y=(a*x+b)*x+c;
    printf("x=%d, y=%d;    ", x, y);
  }
  printf("\n\n");
  /* Затримка вікна виведення */
  system("pause");
  return 0;
}
```

```
   Обчислення значення y=ax2+bx+c
Введіть значення a, b, c  1 2 3
Введіть значення x0, xn, h  -1 3 1
x=-1, y=2;   x=0, y=3;   x=1, y=6;   x=2, y=11;   x=3, y=18;
```

Подані в прикладах словесні описи алгоритмів і блок-схеми є досить деталізованими. Це майже ніяк не погіршує сприйняття алгоритму, але, якщо кількість кроків чи блоків вимірюється сотнями, то роль кожного окремого кроку чи блоку стає мало зрозумілою, а їхня взаємодія — не відслідковується. Частково це є наслідком складності алгоритму. У такому випадку алгоритм треба подавати узагальненими блоками, а потім при потребі певні блоки деталізувати.

1.6. Найпоширеніші стратегії алгоритмізації

В алгоритмізації використовують різні підходи. Найпоширенішими є жадібна стратегія, стратегія розділяй і володарюй, динамічне програмування.

Жадібна стратегія розв'язує завдання розміру *n* покроково. На кожному кроці жадібний алгоритм знаходить найкращий локальний розв'язок, який можна одержати, враховуючи наявну інформацію. Як правило, розмір завдання при цьому зменшується на 1. Після того, як буде виконано всі *n* кроків, алгоритм повертає загальний розв'язок.

Наприклад, для сортування масиву a[n] жадібне сортування вибором знаходить найбільше значення серед усіх *n* елементів масиву і переставляє

його з останнім елементом. Потім знаходить найбільше значення серед решти $(n-1)$ елемента і переставляє його з передостаннім елементом і т.д., доки не буде відсортовано весь масив.

Алгоритми, в яких використовується жадібна стратегія, як правило, характеризуються повільним зменшенням розміру розв'язуваних підзадач в міру обробки алгоритмом вхідних даних. Наприклад, якщо підзадачу можна розв'язати за час $O(\log n)$, то жадібна стратегія матиме продуктивність $O(n\log n)$; якщо підзадача має складність $O(n)$ (як в розглянутому алгоритмі сортування вибором), то загальна складність буде $O(n^2)$.

Жадібні алгоритми найкраще підходять для розв'язування простих задач.

Стратегія *розділяй і володарюй* (ділення навпіл) розв'язує задачу розміру *n*, рекурсивно розділяючи її на дві незалежні підзадачі того ж типу, кожна з яких має близько половини розміру початкової задачі. Задача розбивається доти, поки всі підзадачі не стануть елементарними. У загальному випадку задачу можна розбити на більшу ніж дві кількість підзадач і підзадачі можуть мати різний розмір.

Прикладом застосування методу розділяй і володарюй є бінарний пошук елемента у відсортованому масиві — масив ділиться навпіл, доки не буде знайдено певного елемента або доки не буде встановлено, що елемента в масиві нема. Такий алгоритм має середню й найгіршу складність $O(\log n)$, найкраща складність $O(1)$.

Динамічне програмування є варіацією підходу розділяй і володарюй, яка розв'язує задачу шляхом її поділу на кілька простіших підзадач, які розв'язуються в певному порядку. Вона розв'язує кожну з простіших підзадач тільки один раз і зберігає результати для подальшого використання, щоб уникнути повторного перерахунку. Потім цей метод розв'язує задачу більшого розміру і об'єднує в єдиний розв'язок розв'язки менших підзадач. У багатьох випадках знайдений розв'язок є доказово оптимальним. Динамічне програмування часто використовують для задач оптимізації, в яких мінімізується або максимізується значення певної функції, яку називають цільовою функцією.

Одним з прийомів методу динамічного програмування є *мемоїзація*, яку використовують для прискорення роботи програм, виключення повторного обчислення результатів попередніх викликів функцій шляхом зберігання у спеціальних таблицях результатів їхніх викликів, виключення повторного обчислення результатів попередніх проходів циклів, тобто використання результатів попередніх ітерацій циклу для обчислення значень на наступних ітераціях.

Запитання та завдання для самоконтролю

1. Дайте визначення алгоритму. Наведіть приклади алгоритмів. Чи є алгоритмом програма, подана певною мовою програмування або машинними кодами?

2. Вкажіть і схарактеризуйте основні властивості алгоритмів.

3. У чому полягає аналіз ефективності алгоритму? Що таке часова складність алгоритму? Що таке ємнісна складність алгоритму? Що таке асимптотична складність алгоритму? Поясніть суть асимптотичної оцінки $O(g(n))$. Схарактеризуйте константну, лінійну, квадратичну, логарифмічну, лінійно-логарифмічну і експоненційну складності алгоритму.

4. Які задачі належать до класу задач поліноміальної складності? Чи завжди велика асимптотична складність свідчить про те, що алгоритм буде виконуватися довше для будь-яких обсягів даних?

5. Вкажіть способи подання алгоритмів. Схарактеризуйте словесне подання алгоритму. Які блоки використовують для графічного подання алгоритму? Що таке програма? Які відмінності алгоритмічної мови від машинного коду? Що таке синтаксис мови програмування? Що є лексичними одиницями мови програмування? Чим ключові слова мови програмування відрізняються від ідентифікаторів? Вкажіть існуючі правила вибору імен ідентифікаторів. Для чого використовують коментарі? У чому перевага використання іменованих констант?

6. Вкажіть і схарактеризуйте найпоширеніші стратегії алгоритмізації.

Практичне завдання № 1

«Основи роботи в інтегрованому середовищі Visual Studio»

Мета виконання завдання: засвоїти основи роботи в інтегрованому середовищі Microsoft Visual Studio (консольний режим С++); виконати в цьому середовищі просту програму.

Завдання 1. Відпрацювати можливості роботи в інтегрованому середовищі Visual Studio (консольний режим С++):

Ctrl+Shift+N — створення нового проекту;

Ctrl+Shift+O — відкривання наявного проекту;

Ctrl+Shift+A — створення пункту проекту, у полі Name треба вказати ім'я файла з розширенням .c;

Ctrl+S — збереження файла програми;

F5 — компіляція й виконання програми в режимі налагодження або продовження виконання програми після точки зупинки;

Ctrl+F5 — компіляція й виконання програми без налагодження;

Ctrl+F7 — компіляції проекту;

Ctrl+U — усі букви виділеного фрагменту зробити малими;

Ctrl+Shift+U — усі букви виділеного фрагменту зробити великими;

Ctrl+K, C — закоментувати виділені рядки програми;

Ctrl+K, U — розкоментовувати виділені рядки програми;

Ctrl+K, F — структуризація виділеного фрагменту тексту;

нумерація рядків тексту — команда Tools – Options – розкрити список Text Editor – розкрити список C/C++ – General – Line numbers – OK;

Alt+F4 — закінчення роботи.

Завдання 2. Набрати текст програми, яка видає привітання.

```
/* Завдання № 1. Основи роботи в середовищі Visual Studio */
#include <stdio.h>    // директива підмикання заголовочного
                      // файла бібліотеки stdio.h */
#include <stdlib.h>    // підмикання бібліотеки stdlib.h
#define N 255   // директива означення макросу -- визначає
                // константу з іменем N і значенням 255
```

```
int main() { // тіло програми
  char st[N];   // оголошення рядкової змінної st
  printf("\n\n\n\t\t\tPryvit! Tse -- C\n\n");   // символ
                  // керування \n переводить курсор на новий
                  // рядок; \t задає табулювання на 8 позицій
  printf("Yak tebe zvaty?\n");
  gets(st);   // введення рядка тексту
  system("color 0B");   // параметр зміни кольору: 0 - колір
                      // фону (від 0 до F), B – колір шрифту
  printf("Pryvit, %s!\a",st);   // виведення рядкової змінної st,
                      // символ \a задає звуковий сигнал
  getchar();   // введення символу чи затримка екрану — чекає
              // натискання клавіші Enter
  system("color DE");
  printf("Bazhaiu uspikhiv!\n");
  getchar();
  return 0;   // повертає код завершення функції main
}
```

Виконати програму. Подати повідомлення різними кольорами.

2. Типи обчислювальних процесів. Різноманітність алгоритмів

Типи обчислювальних процесів: лінійні, розгалужені, циклічні. Приклад розробки простого алгоритму. Різноманітність алгоритмів розв'язування задачі.

2.1. Типи обчислювальних процесів: лінійні, розгалужені, циклічні

Відповідно до методології програмування, яку називають *структурним програмуванням* і яка була запропонована в 1970-х роках нідерландським науковцем у галузі комп'ютерних наук Едсгером Дейкстрою[6], розроблена й доповнена швейцарським програмістом Ніклаусом Віртом[7], логічну структуру будь-якого алгоритму можна подати (будь-яку програму можна створити), за допомогою комбінації трьох *базових алгоритмічних структур*: лінійної, розгалуження і циклу. Кожна структура має один вхід і один вихід.

Одна із цілей структурного програмування — позбутися поганої структури програми, коли програми заплутані, використовується багато переходів (програми нагадували блюдо спагеті «bowl of spaghetti» — їх називали BS-програмами). Інша ціль — створення програм, які можна розуміти, супроводжувати і модифікувати без участі автора (як правило, вартість супроводу й модифікації програм у 3-5 разів більша від вартості їхнього створення).

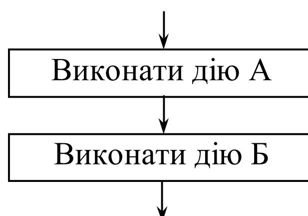
Структурне програмування ще називали програмуванням без goto. Щодо доцільності вживання цього оператора довгий час тривала дискусія. Так, Е. В. Дейкстра в листі редактору журналу Communications of the ACM (том 11, № 3, 1968) писав, що кваліфікація програмістів — спадна функція від щільності операторів goto в програмах, які вони створюють, і що він переконаний, що цей оператор треба вилучити з усіх мов програмування високого рівня (крім, можливо, машинного коду). Проте Д. Кнут у статті «Структурне програмування з операторами goto», опублікованій у журналі Computing Surveys (том 6, № 4, 1974), писав, що можна писати структурова-

[6] Едсгер Вібе Дейкстра (Edsger Wybe Dijkstra, 1930-2002) — нідерландський науковець у галузі комп'ютерних наук, один з розробників концепції структурного програмування.
[7] Ніклаус Вірт (Niklaus Emil Wirth, 1936 р.н.) — швейцарський програміст і теоретик програмування, автор мови програмування Pascal.

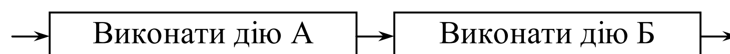
ні програми, які містять оператори goto, як і неструктуровані програми, які не містять жодного goto [39].

Лінійна структура передбачає, що дії мають виконуватися одна за одною по одному разу в порядку їхнього запису в алгоритмі чи тексті програми.

За допомогою блок-схеми лінійну структуру можна подати так (розміщуючи блоки виконання згори вниз):



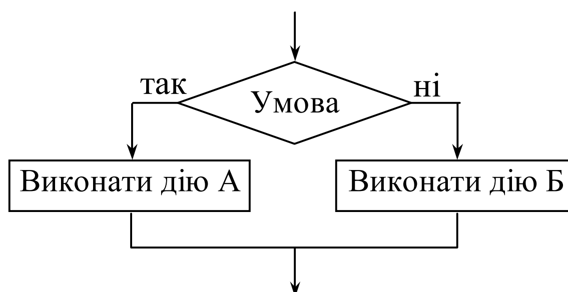
чи так (розміщуючи блоки виконання зліва направо):



Прямокутники у схемі можуть позначати як одну дію (оператор), так і багато дій, які відповідають складним обчисленням і які можна деталізувати. Лінійні структури є найважливішими серед алгоритмічних структур.

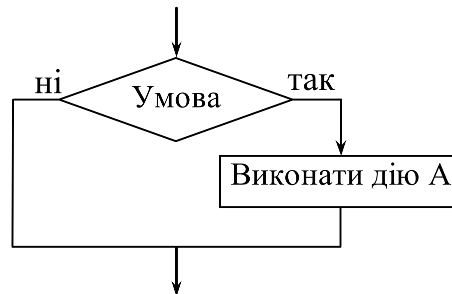
Використання лінійного підходу в програмуванні полегшує сприйняття і супровід програм. Відповідно до цього підходу в алгоритмах і програмах нема (крім циклів) повернень назад, виконання відбувається згори вниз.

Структура розгалуження (її також називають «якщо-то-інакше») забезпечує вибір між двома альтернативами залежно від значення наперед заданої умови — вибирається один із шляхів, кожен з яких передбачає виконання певної операції або кількох операцій:

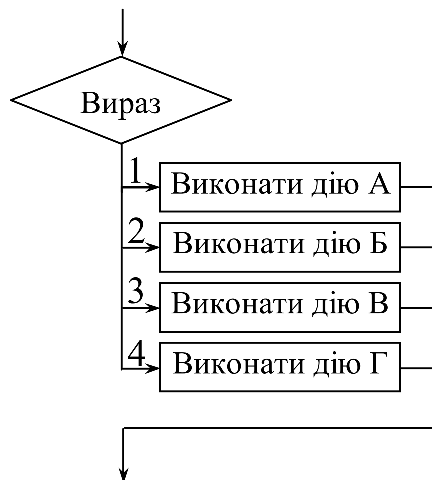


Кожен із шляхів (альтернатива «то», альтернатива «інакше») веде до спільної точки злиття; тому обробка продовжується незалежно від того, який шлях було вибрано.

Може бути так, що для одного з результатів перевірки (виконання чи невиконання умови) нічого виконувати не треба. У цьому випадку можна використати тільки один блок виконання, наприклад:



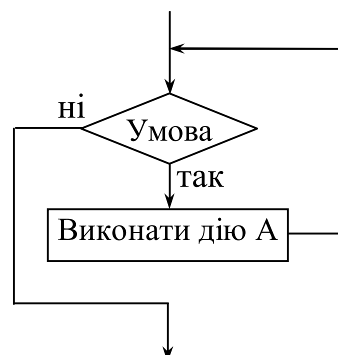
Узагальненням структури розгалуження є структура множинного вибору, коли залежно від значення умови-виразу виконується одна з кількох можливих дій:



Циклічна структура (структура повторення) передбачає багаторазове виконання дій при виконанні деякої умови.

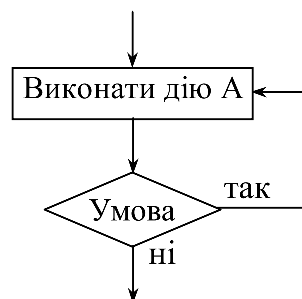
У циклічній структурі умова може перевірятися перед або після виконання дій.

Наприклад, якщо умова перевіряється перед виконанням дій (цикл з передумовою; у мові С це цикли while і for), то блок-схема має вигляд:



При вході в цикл обчислюється умова. Якщо умова набуває істинного значення, то виконуються певні дії, після чого знову перевіряється умова. Ці дії виконуються, доки умова набуває істинного значення. Як тільки умова набуває хибного значення, робота циклу закінчується. Оскільки умова перевіряється на самому початку, то умова може зразу бути хибною і в цьому випадку цикл зовсім не буде виконуватися. Оператори в циклі (їх може бути досить багато) повинні змінювати умову, бо інакше програма буде працювати до нескінченності — зациклиться.

Якщо умова перевіряється після виконання дій (цикл з післяумовою; у мові C це цикл do-while), то блок-схема буде, наприклад, такою:



При вході в цикл виконуються певні задані дії. Після цього обчислюється умова. Якщо умова набуває істинного значення, то робота циклу продовжується. Як тільки умова набуде хибного значення, робота циклу закінчиться[8]. Оскільки умова перевіряється в кінці циклу, то дії виконуються принаймні один раз. Щоб не було зациклювання, як і у випадку циклу з передумовою, треба, щоб оператори в циклі змінювали умову.

Циклічні структури використовують у більшості обчислювальних алгоритмів.

Наведені вище три базові алгоритмічні структури можна комбінувати. Так, будь-який прямокутник на схемі можна замінити будь-якою базовою структурою чи кількома структурами.

Особливістю базових алгоритмічних структур є їхня повнота, тобто за допомогою цих структур можна створити найскладніший алгоритм.

[8] У мові програмування Pascal все відбувається навпаки: цикл з післяумовою repeat-until працює, доки умова набуває хибного значення, і закінчує роботу, коли умова набуває істинного значення.

2.2. Приклад розробки простого алгоритму

Розглянемо процес розробки алгоритму — послідовність наближень від алгоритму розв'язання часткової задачі до алгоритму розв'язання задачі в загальному вигляді.

Нехай треба обчислити значення 5^3. При цьому вважатимемо, що операція піднесення до степеня не реалізована, але комп'ютер може виконувати множення.

Найпростіше (надто не задумуючись) обчислення цього значення можна реалізувати так:

①*К1.* X3=5*5*5.

К2. Вивести X3.

або так:

②*К1.* X1=5.

К2. X2=X1*5.

К3. X3=X2*5.

К4. Вивести X3.

Якщо ж задачу трохи ускладнити — треба знайти не 5^3, а, наприклад, 5^{20} (результатом обчислення є число 95367431640625), то наведений алгоритм буде дуже довгим. Тому, якщо трохи подумати, для знаходження 5^{20} можна алгоритм зробити універсальнішим:

③*К1.* Встановити початкове значення X20=1.

К2. Встановити початкове значення I=1.

К3. Якщо I>20, то перейти на К5;

К4. X20=X20*5, I=I+1, перейти на К3.

К5. Вивести X20.

К6. Кінець.

Цей алгоритм (але ще не повну програму, а лише її фрагмент) мовою програмування С можна подати так:

```
x20=1;
i=1;
```

```
K3:
  if (i>20) goto K5;
  else {
    x20*=5;
    i++;
    goto K3;
  }
    printf("  %lld ", x20);
```

Можна початкове значення зразу встановити рівним 5 (цим зекономимо один прохід циклу) і на кроці К3 змінимо умову:

④*К1.* Встановити початкове значення X20=5. //початкове значення не 1, а 5

К2. Встановити початкове значення I=1.

К3. Якщо I=20, то вивести X20 і перейти на К4.

інакше X20=X20*5, I=I+1 і перейти на К3.

К4. Кінець.

При поданні мовою програмування С алгоритм матиме вигляд:

```
  x20=5 ;
  i=1;
K3:
  if (i==20)
    printf("  %lld ", x20);
  else {
    x20*=5;
    i++;
    goto K3;
  }
```

У попередніх алгоритмах на кроці К3 істинне значення умови вказувало, що результат одержано, його треба вивести і закінчити роботу алгоритму. Переробимо умову так, щоб її істинне значення вказувало на необхідність продовження обчислень:

⑤*К1.* Встановити початкове значення X20=5.

К2. Встановити початкове значення I=1.

К3. Якщо I<20, то X20=X20*5; I=+1; перейти на К3.

К4. Вивести X20.

Мовою програмування C цей алгоритм буде подано так:

```
  x20=5;
  i=1;
K3:
  if (i<20) {
    x20*=5;
    i++;
    goto K3;
  }
  else
    printf("  %lld ", x20);
```

На основі розробленого алгоритму для обчислення конкретного значення 5^{20} можна легко записати алгоритм для обчислення будь-якого цілого додатного степеня N (N>0)[9] будь-якого числа X (замінимо число 20 на N, а 5 — на X):

⑥*К1.* Ввести X і N.

К2. Встановити початкове значення XN=X.

К3. Встановити початкове значення I=1.

К4. Якщо I<N, то XN=XN*X; I=I+1; перейти на К4.

К5. Вивести XN.

Подамо алгоритм мовою програмування C:

```
  printf(" введіть x i n  ");
  scanf("%d%d", &x, &n);
  xn=x;
  i=1;
K3:
  if (i<n) {
    xn*=x;
```

[9] Якщо брати N>=0, то треба повернутися на кроці К1 до початкового значення XN=1 (див. ③) і умову на кроці К3 записати як I<=N.

```
    i++;
    goto K3;
  }
  else
    printf("  %lld ", xn);
```

```
введіть x і n  5 20
 95367431640625
```

Проте принципи структурного програмування не дозволяють використовувати оператор безумовного переходу goto. Подамо повну програму, яка реалізує алгоритм, використовуючи оператор циклу (в операторі циклу оператор goto використовується неявно, але логіка програми не заплутується):

```
#include <stdio.h>
#include <stdlib.h>
int main() {
// тіло програми
  int x, n, i;
  long long int xn;
  system("chcp 1251 & cls");
  printf(" введіть x і n (n>0) ");
  scanf("%d%d", &x, &n);
  xn=x;
  i=1;
  while (i<n) {
    xn*=x;
    i++;
  }
  printf("  %lld ", xn);
  printf("\n\n");
  system("pause");
  return 0;
}
```

Або можна використати цикл for:

```
  xn=x;
  for (i=1;i<n; i++)
    xn*=x;
  printf("  %lld ", xn);
```

```
введіть x і n (n>0) 5 20
 95367431640625
```

Як видно з поданого повного тексту програми, при записові алгоритму мовою програмування C ще треба врахувати те, що значення результату буде дуже великим і може не поміститися у виділену ділянку пам'яті, тобто

виникне *арифметичне переповнення*. Якщо змінна xn матиме тип int (займає 4 байти пам'яті; максимальне значення величини типу int — десятизначне число $2^{31}-1=2147483647$), то при обчисленні значення 5^{20} «результат» буде таким 1977800241 — за рахунок арифметичного переповнення, хоч правильна відповідь $5^{20}=95367431640625$. Тому змінна, в яку буде накопичуватися результат, повинна мати тип long long int (займає 8 байтів пам'яті; максимальне значення величини типу long long int — дев'ятнадцятизначне число $2^{63}-1=9223372036854775807$). Якщо і в цьому випадку можливе переповнення, то можна скористатися змінною типу double, але результат вже може бути наближеним за рахунок втрати молодших розрядів. Також для роботи з великими цілими числами без втрати точності їх подають як послідовність (масив) цифр в системі числення з досить великою основою[10] (див. приклад 8.5.3).

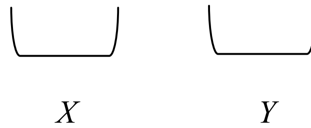
Сказане вище вказує на те, що на етапі програмування треба враховувати особливості виконання алгоритмів на комп'ютері.

2.3. Різноманітність алгоритмів розв'язування задачі

Хоч алгоритм є однозначною процедурою розв'язання задачі, але ту саму задачу можна розв'язати, використовуючи різні алгоритми. Розглянемо кілька можливих алгоритмів виконання переприсвоєння значень двох змінних, визначення парності цілого числа, однаковості парності двох чисел, однаковості знаків двох чисел.

Приклад 2.3.1. ***Переприсвоєння значень двох змінних.*** При виконанні програм на комп'ютерах будь-які дані (наприклад, числа, символи) зберігаються в пам'яті машини. Залежно від того, якого типу ці дані, вони можуть розміщуватися в різних за обсягом ділянках пам'яті. Ділянкам пам'яті при розробці алгоритму чи написанні програми дають імена:

[10] Для роботи з великими числами існує досить багато оптимізованих бібліотек. Наприклад, мова програмування Python (випущена в 1991 році), підтримує роботу з великими цілими числами довільної довжини. Слід зазначити, що машина МИР, розроблена в Інституті кібернетики АН УРСР і впроваджена в серійне виробництво в 1965 році, могла працювати з числами з довільною розрядністю і діапазоном. Довга десяткова арифметика була реалізована в мові програмування Аналітик для машини МИР-2 (1969 рік).



У багатьох алгоритмах виникає необхідність обміну значень змінних, тобто, якщо є дві змінні X і Y, то треба переприсвоїти їхні значення, щоб змінна X набула значення Y, а змінна Y набула значення X (переприсвоєння значень змінних використано в алгоритмі впорядкування двох чисел — приклад 1.5.1).

Спосіб 1. У найчастіше застосовуваному на практиці алгоритмі переприсвоювання значень двох змінних використовують додаткову ділянку пам'яті, а в алгоритмі додаткову робочу змінну (часто її називають буфером).

Нехай X і Y — змінні, значення яких треба обміняти, R — робоча змінна. Алгоритм має вигляд:

K1. Ввести X і Y.

K2. R=X.

K3. X=Y.

K4. Y=R.

K5. Вивести X і Y.

Перевіримо, чи правильно працює цей алгоритм. Нехай X=5, Y=3.

Ділянки пам'яті	X	Y	R
Початковий стан пам'яті	∞	∞	∞
K1	**5**	**3**	∞
K2	5	3	5
K3	3	3	5
K4	**3**	**5**	5
K5	**3**	**5**	

Цей алгоритм потребує виконання трьох операцій присвоювання і його можна застосовувати до даних будь-яких типів — числових, символьних, рядкових (якщо мова програмування підтримує присвоєння рядків; у мові C для копіювання рядків можна скористатися функцією strcpy) і навіть структур.

Спосіб 2. Алгоритм з використанням операцій додавання й віднімання без використання додаткової ділянки пам'яті (таке завдання часом формулюють на олімпіадах з програмування):

К1. Ввести X і Y.

К2. X=X+Y.

К3. Y=X–Y.

К4. X=X–Y.

К5. Вивести X і Y.

Перевіримо, чи алгоритм працює правильно. Нехай X=5, Y=3.

Ділянки пам'яті		X	Y
Початковий стан пам'яті		∞	∞
	К1	**5**	**3**
	К2	5+3=8	3
	К3	8	8–3=5
	К4	8–5=**3**	**5**
	К5	**3**	**5**

Цей алгоритм потребує виконання трьох операцій присвоювання і трьох адитивних (додавання і віднімання) операцій, які виконуються довше, ніж присвоювання. Його можна застосувати тільки до числових даних.

Крім того, цей спосіб не завжди дасть правильну відповідь. Так, при роботі з дійсними числами можуть *втратитися молодші розряди*. Наприклад, якщо в якомусь віртуальному комп'ютері виділяється два розряди для зберігання мантиси (значущих цифр) дійсного десяткового числа, то при додаванні 2,5+0,43 (у пам'яті доданки зберігаються як $2{,}5\ 10^{0}$ і $4{,}3\ 10^{-1}$) результат буде не 2,93, а 2,9 (молодший розряд втратиться).

Проте для цілих чисел, не зважаючи на можливість при обчисленнях в комп'ютері виникнення арифметичного переповнення, результат буде правильним за рахунок двох переповнень (при додаванні, а потім при відніманні) у двійкових поданнях чисел у пам'яті (див. приклади 3.4.2, 3.4.3).

Аналогічний алгоритм можна записати, використовуючи операції множення й ділення:

К1. Ввести X і Y.

К2. X=X*Y.

К3. Y=X/Y.

К4. X=X/Y.

К5. Вивести X і Y.

При цьому операції множення й ділення виконуються значно довше, ніж додавання й віднімання, і проблеми з втратою розрядів існують ще гостріші.

Подібними є алгоритми, в яких використані раніше арифметичні дії міняються місцями:

К2. X=X–Y.

К3. Y=X+Y.

К4. X= –X+Y (тобто X=Y–X).

або

К2. X=X/Y.

К3. Y=X*Y.

К4. X=Y/X.

У цих алгоритмах, крім операції присвоювання (по три присвоювання в кожному алгоритмі), використовується також по три арифметичні операції. Останній алгоритм для цілих чисел буде працювати правильно тільки тоді, коли X/Y є цілим числом.

Алгоритми з використанням арифметичних операцій можна застосувати тільки для переприсвоювання числових значень, але і з числовими значеннями вони не завжди будуть коректно працювати.

Спосіб 3. Алгоритм з використанням побітової (порозрядної) операції додавання за модулем 2 ($0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$) без використання додаткової ділянки пам'яті:

К1. Ввести X і Y.

К2. X=X⊕Y.

К3. Y=X⊕Y.

К4. X=X⊕Y.

К5. Вивести X і Y.

Перевіримо, чи правильно працює алгоритм. Нехай X=1001, Y=0011:

Ділянки пам'яті	X	Y
Початковий стан пам'яті	∞	∞
К1	**1001**	**0011**

К2	$\oplus\begin{array}{r}1001\\ \underline{0011}\\ 1010\end{array}$	0011
К3	1010	$\oplus\begin{array}{r}1010\\ \underline{0011}\\ 1001\end{array}$
К4	$\oplus\begin{array}{r}1010\\ \underline{1001}\\ 0011\end{array}$	1001
К5	**0011**	**1001**

З наведеного прикладу видно, що побітові операції здійснюються над кожним бітом окремо, тобто сусідні біти незалежні (а, наприклад, при додаванні чисел здійснюється перенесення значень у старші розряди). Таким чином, при безпосередній роботі з пам’яттю алгоритм дає можливість перепривсоювати значення окремих бітів і тому може застосовуватися до даних будь-яких типів.

Мовою С (у мові С побітові операції можна застосовувати тільки до даних цілих типів; операцію побітового додавання за модулем 2 в мові С позначають ^) кроки К2-К4 можна подати так:

```
x=x^y;
y=x^y;
x=x^y;
```

або за допомогою скороченого запису (тут присвоювання виконуються справа наліво):

```
x^=y^=x^=y;    // або x=x^(y=(y^(x=x^y)));.
```

Алгоритм з операцією додавання за модулем 2 не використовує додаткової ділянки пам’яті, потребує виконання трьох операцій присвоювання і тричі виконання операції побітового додавання за модулем 2. Побітові операції виконуються приблизно з тією ж швидкістю, що й операція додавання.

Але цей алгоритм при реалізації його мовою С щодо типів змінних має ще жорсткіші обмеження, ніж алгоритми з використанням арифметичних операцій — його можна застосовувати тільки до даних цілих типів.

Таким чином, алгоритм з використанням додаткової пам'яті є найкращим щодо швидкості (що може бути суттєвим при переприсвоєнні значень великої кількості змінних) і можливості застосування до даних різних типів, навіть складних типів (наприклад, структур — struct у мові C, record у мові Pascal). При роботі мовою C з даними цілих типів можна застосувати економніший щодо пам'яті алгоритм з досить швидкими побітовими операціями. Можна зауважити, що процесори Intel мають спеціальну команду XCHG — обміняти.

Приклад 2.3.2. ***Визначення парності цілого числа.*** Парність числа можна визначити, знайшовши остачу від ділення на 2 і за допомогою побітової операції кон'юнкції (логічного «і») чи побітових операцій зсуву. Перевірка на парність здійснюється тільки для цілих чисел.

Спосіб 1. Якщо число непарне, то остача від ділення на 2 дорівнює 1, якщо парне — 0. Операцію знаходження остачі від ділення в математиці позначають словом mod.

К1. Ввести ціле число X.

К2. Якщо X mod 2 =1, то вивести «непарне»;

інакше вивести «парне»;

Відповідний фрагмент програми мовою C буде таким:

```
printf("Введіть x ");
scanf("%d", &x);
printf("Число %d - ", x);
if (x%2)
  printf("непарне");
else
  printf("парне");
```

Спосіб 2. Якщо у двійковому поданні число закінчується одиницею, то воно непарне, якщо закінчується нулем, то парне. Наприклад, $5_{10} = 101_2$ — непарне число, $6_{10} = 110_2$ — парне. Визначити останню цифру двійкового числа можна за допомогою побітової кон'юнкції ($0 \& 0 = 0$, $0 \& 1 = 0$, $1 \& 0 = 0$, $1 \& 1 = 1$) з числом 1 ($1_{10} = 1_2$). Наприклад:

$$\& \frac{\begin{array}{r}101\\001\end{array}}{001_2} = 1_{10} \qquad \& \frac{\begin{array}{r}110\\001\end{array}}{000_2} = 0_{10}$$

К1. Ввести ціле число X.

К2. Якщо X and 1 =1, то вивести «непарне»;

інакше вивести «парне»;

Відповідний фрагмент програми мовою C буде таким:

```
printf("Введіть x ");
scanf("%d", &x);
printf("Число %d - ", x);
if (x&1)
  printf("непарне");
else
  printf("парне");
```

Спосіб 3. До двійкового подання даних цілого типу можна застосувати операції побітових зсувів — спочатку виконати зсув вправо на один розряд >>1 (при цьому наймолодший розряд числа вийде за розрядну сітку і вилучиться з двійкового подання), а потім виконати зсув вліво на один розряд <<1 (при цьому всі розряди зсунуться вліво, а наймолодший розряд числа стане рівним 0). Якщо число парне, то у двійковому поданні воно закінчується нулем, і після виконання зсувів число не зміниться. Якщо ж число непарне, то у двійковому поданні воно закінчується одиницею, і після виконання зсувів ця одиниця пропаде і буде одержано інше число (на 1 менше від початкового). Наприклад:

Число	$5_{10} = 101_2$	$6_{10} = 110_2$
Початковий стан пам'яті	1 0 1	1 1 0
Здійснено зсув на один розряд вправо	0 1 0	0 1 1
Здійснено зсув на один розряд вліво	1 0 0	1 1 0

Таким чином, можна записати алгоритм мовою C для перевірки парності числа:

```
printf("Введіть x ");
```

```
scanf("%d", &x);
printf("Число %d - ", x);
if (((x>>1)<<1)==x)
  printf("парне");
else
  printf("непарне");
```

Серед розглянутих алгоритмів визначення парності цілого числа найефективнішим є алгоритм з використанням побітової кон'юнкції.

Приклад 2.3.3. ***Визначення парності двох цілих чисел.*** Визначити, чи мають два числа однакову парність, можна за допомогою порівняння остач від ділення на 2, обчислення суми чи різниці цих чисел, а також за допомогою побітових операцій додавання за модулем 2 і кон'юнкції.

Спосіб 1. Якщо обидва числа *a* і *b* парні чи обидва непарні, то вони мають однакову парність. Цю перевірку можна здійснити за допомогою логічного виразу. Перше, що спадає на думку, це записати такий вираз:

$$((a \bmod 2 = 0) \text{ and } (b \bmod 2 = 0)) \text{ or } ((a \bmod 2 = 1) \text{ and } (b \bmod 2 = 1)).$$

При обчисленні значення цього виразу виконується чотири рази ділення з остачею, чотири рази порівняння і три логічні операції. Проте нам потрібно визначити тільки те, чи однакові остачі, тому досить порівняти тільки їх:

$$(a \bmod 2) = (b \bmod 2).$$

Цей вираз дає той самий результат, що й попередній, але він значно простіший і в ньому виконується два ділення з остачею і одне порівняння.

Маємо алгоритм:

K1. Ввести a і b.

K2. Якщо (a mod 2) = (b mod 2), то вивести «однакова парність»; інакше вивести «різна парність».

Мовою C алгоритм запишеться так:

```
printf("Введіть a і b ");
scanf("%d%d", &a, &b);
printf("Числа a=%d і b=%d мають ", a, b);
if (a%2 == b%2)
```

```
    printf("однакову парність");
  else
    printf("різну парність");
```

Спосіб 2. Сума (різниця) двох чисел з однаковою парністю є парним числом. Наприклад, сума двох непарних чисел $5+7=12$ (різниця $5-7=-2$) є парним числом і сума двох парних чисел $6+2=8$ (різниця $6-2=4$) є парним числом. При цьому навіть, якщо виникне арифметичне переповнення, то втратяться старші розряди результату, а в молодшому розряді двійкового подання результату буде зберігатися потрібна одиниця чи нуль.

Алгоритм визначення, чи однакову парність мають два задані числа, з використанням їхньої суми і остачі від ділення цієї суми на 2 мовою C можна подати так:

```
  printf("Введіть a i b ");
  scanf("%d%d", &a, &b);
  printf("Числа a=%d i b=%d мають ", a, b);
  if ((a+b)%2)
    printf("різну парність");
  else
    printf("однакову парність");
```

У цьому алгоритмі при перевірці умови виконується одна операція додавання й одне ділення з остачею. Тому він швидший від попереднього.

Спосіб 3. Якщо обидва числа мають однакову парність, то двійкові подання обох чисел закінчуються або 0, або 1. Тоді застосування побітового додавання за модулем 2 дасть в обох випадках 0 ($0\oplus 0=0$, $1\oplus 1=0$). Якщо два числа мають різну парність, то двійкове подання одного з них закінчуватиметься 0, а іншого 1, і побітове додавання за модулем 2 дасть результат 1 ($0\oplus 1=1$, $1\oplus 0=1$). Побітова кон’юнкція одержаного результату з числом 1 дасть 0, якщо обидва числа мають однакову парність, і дасть 1, якщо числа мають різну парність. Наприклад:

Числа	$5_{10}=101_2$	$6_{10}=110_2$	$5_{10}=101_2$
	$7_{10}=111_2$	$2_{10}=010_2$	$6_{10}=110_2$

Побітове додавання за модулем 2	$\oplus\begin{array}{r}101\\ \underline{111}\\ 010\end{array}$	$\oplus\begin{array}{r}110\\ \underline{010}\\ 100\end{array}$	$\oplus\begin{array}{r}101\\ \underline{110}\\ 011\end{array}$
Побітова кон'юнкція результату з 1	$\&\begin{array}{r}010\\ \underline{001}\\ 000\end{array}$	$\&\begin{array}{r}100\\ \underline{001}\\ 000\end{array}$	$\&\begin{array}{r}011\\ \underline{001}\\ 001\end{array}$

Алгоритм визначення, чи однакову парність мають два задані числа, з використанням побітової суми за модулем 2 і побітової кон'юнкції можна мовою C подати так:

```
printf("Введіть a i b ");
scanf("%d%d", &a, &b);
printf("Числа a=%d i b=%d мають ", a, b);
if ((a^b)&1)
  printf("різну парність");
else
  printf("однакову парність");
```

У цьому алгоритмі при перевірці умови виконується одна операція побітового додавання за модулем 2 і одна побітова кон'юнкція.

З використанням побітових операцій розглянуті вище в способі 1 і способі 2 умови `a%2==b%2` і `(a+b)%2` можна відповідно подати так: `(a&1)==(b&1)` і `(a+b)&1`.

Алгоритми з використанням побітових операцій будуть працювати швидко, оскільки побітові операції виконуються швидше від арифметичних.

Приклад 2.3.4. ***Визначення того, чи мають два числа однакові знаки.*** Визначити, чи мають два числа однакові знаки, можна за допомогою логічного виразу, а також шляхом обчислення добутку цих чисел. Вважатимемо, що нуль може бути як додатним, так і від'ємним. При цьому числа можуть бути як цілими, так і дійсними. Якщо числа цілі зі знаком, то ще можна скористатися побітовою операцією зсуву.

Спосіб 1. Якщо обидва числа *a* і *b* від'ємні чи обидва додатні, то вони мають однакові знаки. Якщо одне з чисел додатне, а інше від'ємне, то вони

вони мають різні знаки. Цю перевірку можна здійснити за допомогою логічного виразу. Алгоритм мовою С матиме вигляд:

```
printf("Введіть a і b ");
scanf("%d%d", &a, &b);
printf("Числа a=%d і b=%d мають ", a, b);
if (a>=0 && b>=0 || a<=0 && b<=0)
  printf("однакові знаки");
else
  printf("різні знаки");
```

У цьому алгоритмі при перевірці умови виконуються чотири порівняння і три логічні операції. Це швидкі операції.

Спосіб 2. Добуток двох чисел з однаковими знаками дасть додатне число. Тому в алгоритмі можна скористатися цією властивістю. Код алгоритму мовою С матиме вигляд:

```
printf("Введіть a і b ");
scanf("%d%d", &a, &b);
printf("Числа a=%d і b=%d мають ", a, b);
if (a*b>=0)
  printf("однакові знаки");
else
  printf("різні знаки");
```

При перевірці умови виконується одна операція множення і одне порівняння. Операція множення виконується довше, ніж порівняння і логічні операції. При множенні досить великих цілих чисел може виникнути арифметичне переповнення і знак результату буде непередбачуваним. При множенні дійсних чисел знак має зберегтися навіть при великих числах. Але, якщо числа невеликі, то умова з множенням досить наочна.

Спосіб 3. Щоб визначити, чи одного знаку два цілі числа однієї розрядності зі знаком, досить порівняти найстарші розряди їхнього машинного подання, які містять знаки чисел — якщо число додатне чи 0, то 0, якщо від’ємне, то 1. Якщо числа одного знаку, то побітове виключне «або» дасть

у старшому розряді 0 ($0 \oplus 0 = 0$, $1 \oplus 1 = 0$), якщо числа мають різні знаки, то дасть 1 ($0 \oplus 1 = 1$, $1 \oplus 0 = 1$). Код мовою С такий:

```
printf("Введіть a i b ");
scanf("%d%d", &a, &b);
k=sizeof(a)*8-1;
printf("Числа a=%d i b=%d мають ", a, b);
if ((a^b)>>k)
  printf("різні знаки");
else
  printf("однакові знаки");
```

У цьому алгоритмі при перевірці умови виконується одна операція побітового виключного «або» і одна операція зсуву, які є швидкими. Недоліком цього алгоритму є те, що його можна використати тільки для цілих чисел однієї розрядності.

Для розв’язання кожної з розглянутих задач запропоновано по кілька алгоритмів. Деякі з них базуються на знаннях з елементарної математики, але виконуються довше. Інші враховують особливості кодування даних у комп’ютері, але потребують менше часу на виконання. Проте, якщо алгоритм має виконуватися невелику кількість разів, то можна надати перевагу простішому алгоритмові і не звертати уваги на його швидкість. Швидкість має значення в алгоритмах, які будуть виконуватися дуже велику кількість разів або на великих масивах даних.

Запитання та завдання для самоконтролю

1. Схарактеризуйте структурне програмування як методологію програмування. Вкажіть цілі структурного програмування. Вкажіть базові алгоритмічні структури. Що передбачає лінійна алгоритмічна структура? Як можна подати лінійну структуру схематично? Схарактеризуйте структуру розгалуження. Вкажіть види структур розгалуження. Як можна подати структури розгалуження схематично? Що таке структура множинного вибору? Як можна подати схематично структуру множинного вибору? Схарактеризуйте цик-

лічну структуру. Вкажіть види циклічних структур. Як циклічні структури можна подати схематично?

2. Що треба враховувати при реалізації алгоритмів на комп'ютері?

3. Чи можна для розв'язування певної задачі розробити кілька алгоритмів? Використання яких операцій при роботі з цілими числами пришвидшує роботу алгоритму?

Практичне завдання № 2 «Оформлення звітів засобами текстового редактора Word»

Мета виконання завдання: навчитися оформляти звіти виконання завдань засобами текстового редактора Word.

Завдання: Набрати текст поданої програми, коментарі доповнити, де треба (вказано в тексті програми), інформацією.

```
/* Прізвище Ім'я По батькові, група ТХ-ХХ, варіант № XX
 * Завдання № 2. Оформлення звітів засобами текстового
 * редактора Word */
#include <stdio.h>   / підмикання бібліотеки stdio.h; у
        // програмі використано функції XXX,… (вкажіть, які)
#include <stdlib.h>   /* підмикання бібліотеки stdlib.h; у
       // програмі використано функції XXX,… XXX (вкажіть, які)
/* За введеним користувачем числовим кодом від 32 до 255
   (кодування ASCII) виводить символ */
int main() {
  int kod;   // код символа; символи з кодами 0-31 є
             // службовими; пробіл має код 32 */
  system("chcp 1251 & cls");   // для відображення кирилиці
                               // в MS Windows
  printf("Введіть ASCII-код символа від 32 до 255\n");
  scanf("%d",&kod);
  if (kod<32 || kod>255)
    printf("неправильний код\n");
  else
    printf("символ   %c\n",kod);
```

```
  system("pause");  // затримка екрана
  return 0;
}
```

Розібратися з логікою програми і діями, які виконуються в програмі, з інформацією з коментарів.

Виконати цю програму з кількома різними значеннями кодів символів.

Оформити звіт: вказати мету виконання завдання, зміст завдання, подати текст програми, намалювати блок-схему алгоритму програми (дотримуватися ГОСТ 19.701–90), подати не менше трьох зображень результату роботи, сформулювати висновки, у верхніх колонтитулах подати номер завдання, його назву, своє прізвище, номер своєї групи і номер варіанту, в нижні колонтитули вставити нумерацію сторінок.

3. Позиційні і непозиційні системи числення

Системи числення. Подання чисел у позиційних системах. Бінарні й небінарні коди Грея. Подання числових даних у комп'ютері: цілі числа, дійсні числа. Алгоритми виконання арифметичних операцій

3.1. Системи числення

Система числення (ще кажуть: нумерація) — це сукупність правил і знаків, за допомогою яких можна подавати (кодувати) будь-яке невід'ємне число.

Протягом історії людства існувало досить багато різних систем числення. Системи числення є непозиційні, позиційні і змішані.

У *непозиційних системах числення* для кодування значень величин використовують замість цифр, як правило, знаки-числа. При такому кодуванні величина, яку позначає цифра, не залежить від її позиції в числі.

Прикладом непозиційної системи числення є римська система, яка має цифри: I — 1, V — 5, X — 10, L — 50, C — 100, D — 500, M — 1000. При цьому, наприклад, число 9 кодується як IX, 11 — XI, 12 — XII, 13 — XIII, 14 — XIV, 15 — XV, 16 — XVI і т. д., 134 — CXXXIV.

Іншим прикладом є алфавітна система, наприклад, адитивна грецька система (виникла близько 500 року до н.е. у давньогрецькому місті-державі Мілеті), в якій для подання чисел використовували букви грецького алфавіту (α — 1, β — 2, γ — 3, …, θ — 9, i — 10, k — 20, λ — 30, …, θ — 9, …, A — 1000, B — 2000, Г — 3000, …), число є сумою цифр і при цьому порядок цифр значення не має, наприклад: $39 = \lambda\theta = \theta\lambda$; якщо число більше або рівне 10000, то використовували букву M, яка означає тисячу і виконує роль сучасного експоненційного формату подання дійсних чисел. Подібна до грецької системи кирилична система числення.

З числами в непозиційних системах важко виконувати арифметичні й логічні операції.

При передачі й перетворенні сигналів, а також у багатьох алгоритмах генерації різноманітних послідовностей використовують непозиційну систему запи-

су цілих натуральних чисел, у якій два послідовні коди відрізняються лише значенням одного розряду, — це бінарні й небінарні коди Грея[11] (див. п. 3.3).

У *позиційних системах числення* та сама цифра в записі числа набуває різних значень залежно від своєї позиції. Вага кожної позиції кратна натуральному числу $b > 1$, яке є *основою системи числення*. Число a, яке записують як $a_n a_{n-1} a_{n-2} ... a_1 a_0$, де $0 \le a_k < b$, $k = \overline{0,n}$, $b = 2,3,...$, можна подати так: $a = \sum_{k=0}^{n} a_k b^k$. Десяткова система числення має основу $b = 10$ (це найвідоміша система числення; у ній для подання чисел використовують цифри від 0 до 9; найдавніший відомий запис позиційної десяткової системи зроблено в Індії в 595 році; індійські математики в десятковій системі числення використовували цифру і число нуль; десяткові цифри були принесені в Європу арабськими купцями, хоч з того часу написання цифр значно змінилося). Двійкова система числення має основу $b = 2$ (для подання чисел використовують цифри 0 і 1). У програмуванні також використовують вісімкову (тепер досить рідко; цифри від 0 до 7) і шістнадцяткову (цифри від 0 до F) системи числення. Слід зауважити, що при будь-якому $b > 1$, його значення у відповідній системі подається як 10_b.

Вавилоняни (2,5 тис. р. до н.е.) і шумери (4-3 ст. до н.е.) використовували шістдесяткову систему числення.

Племена майя ще в останні сторіччя до н.е. використовували двадцяткову (з основою $b = 20$) позиційну систему числення; у цій системі були цифра і число нуль; також були цифри нескінченності; існували поняття частин і дробів.

Дослідженням двійкової системи числення, яка стала основою при створенні комп’ютерів уже в 20 столітті, займався у XVII-XVIII ст. німецький вчений Лейбніц.

[11] Френк Грей (Frank Gray, 1887-1969) — американський фізик і дослідник. Працював у лабораторії Bell Labs (дослідницький центр у сфері телекомунікацій, електронних і комп’ютерних систем), розробив багато механічних і електронних інновацій в телебаченні. Френк Грей вперше використав розроблений ним код у власній імпульсній системі зв’язку.

У перших комп'ютерах були спроби використовувати десяткову (наприклад, машина ENIAC — США, 1943-46 роки) і навіть трійкову системи числення (єдина в світі машина, в якій використовували трійкову систему числення (двобітові цифри 00 — 0, 11 — 0, 01 — –1, 10 — +1) — радянська машина Сетунь, створена в 1959 році в обчислювальному центрі Московського університету; у 1962-64 роках випускалася серійно). Нейманівські[12] принципи побудови цифрових електронних обчислювальних машин передбачають, що машина працює в двійковій системі числення.

Узагальненням позиційних систем є *змішані системи числення*. Найвідомішим прикладом змішаної системи числення є подання часу — кількість діб, годин, хвилин і секунд. Змішаною є система числення Фібоначчі: $a = \sum_{k=1}^{n} f_k F_k$, де F_k — число Фібоначчі, яке задається рекурентним співвідношенням другого порядку $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$, $n = 3,4,5,...$, $f_k \in \{0,1\}$, причому в записі $f_n f_{n-1} ... f_1$ ніколи дві одиниці не розміщуються поряд (бо інакше $F_{n-1} + F_{n-2} = F_n$). У факторіальній системі числення число подають у вигляді: $a = \sum_{k=1}^{n} f_k k!$, де $0 \le f_k \le n$.

Системи числення мають пряме відношення до математичної теорії чисел.

3.2. Подання чисел у позиційних системах

Розглянемо подання чисел у різних позиційних системах числення, зокрема: десятковій, двійковій, вісімковій і шістнадцятковій.

Перехід від двійкового подання числа до десяткового. Спочатку в десятковій системі числення (основа 10) дійсне число $a = a_n a_{n-1} a_{n-2} ... a_1 a_0 , a_{-1} a_{-2} ... a_{-m}$ подамо як: $a = \sum_{k=-m}^{n} a_k 10^k$.

Приклад 3.2.1. Нехай дано десяткове число $142{,}35_{10}$. Це означає, що:

[12] Джон фон Не́йман (Neumann John (Janos) von, 1903-1957) — американський математик, угорець за походженням.

$$142{,}35_{10} = 100 + 40 + 2 + 0{,}3 + 0{,}05 = 1 \cdot 100 + 4 \cdot 10 + 2 \cdot 1 + 3 \cdot \frac{1}{10} + 5 \cdot \frac{1}{100} =$$

$$= 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 5 \cdot 10^{-2}.$$

У відповідність цифрам (розрядам) десяткового числа $142{,}35_{10}$ можна поставити показники степенів основи 10_{10}:

$$\overset{2}{1}\ \overset{1}{4}\ \overset{0}{2}\ ,\ \overset{-1}{3}\ \overset{-2}{5}\ .$$

Аналогічний підхід можна застосувати й до подання чисел у двійковій системі числення.

Приклад 3.2.2. Нехай число подано у двійковій системі числення (основа 2): $10011{,}1001_2$. Знайдемо його подання в десятковій системі. Скористаємося підходом, аналогічним до того, як це зроблено для числа в десятковій системі числення. У відповідність цифрам заданого числа поставимо показники степенів основи 2:

$$\overset{4}{1}\ \overset{3}{0}\ \overset{2}{0}\ \overset{1}{1}\ \overset{0}{1}\ ,\ \overset{-1}{1}\ \overset{-2}{0}\ \overset{-3}{0}\ \overset{-4}{1}\ .$$

Двійкове число подамо, використовуючи степені основи 2, і виконаємо обчислення в десятковій системі:

$$10011{,}1001_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$$

$$= 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} =$$

$$= 16 + 0 + 0 + 2 + 1 + 0{,}5 + 0 + 0 + 0{,}0625 = 19{,}5625_{10}\ .$$

Одержано десяткове подання заданого числа.

Алгоритм переходу від двійкового подання числа до десяткового:

К1. Поставити у відповідність цифрам цілої частини двійкового числа, ідучи справа наліво, числа 0, 1, 2, 3…, а цифрам дробової частини, ідучи зліва направо, — числа -1, -2, -3… Ці числа є показниками степенів основи 2.

К2. Знайти суму добутків цифр (0 чи 1) двійкового числа на відповідні їм степені числа 2. Одержане значення суми є десятковим поданням двійкового числа.

Аналогічним є алгоритм переходу від подання чисел в системі числення з будь-якою основою b_1 до подання в системі з іншою будь-якою основою b_2.

При переході від двійкового подання цілого числа до десяткового можна скористатися схемою Горнера:

$$\sum_{k=0}^{n} a_k b^k = a_0 + a_1 \cdot b + a_2 \cdot b^2 + \ldots + a_{n-2} \cdot b^{n-2} + a_{n-1} \cdot b^{n-1} + a_n \cdot b^n =$$

$$= (((\ldots((a_n \cdot b + a_{n-1}) \cdot b + a_{n-2}) \cdot b + \ldots) \cdot b + a_2) \cdot b + a_1) \cdot b + a_0 .$$

Для дробової частини аналогічно:

$$\sum_{k=-m}^{-1} a_k b^k = a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + \ldots + a_{-m+2} \cdot b^{-m+2} + a_{-m+1} \cdot b^{-m+1} + a_{-m} \cdot b^{-m} =$$

$$= (((\ldots((a_{-m} \cdot b^{-1} + a_{-m+1}) \cdot b^{-1} + a_{-m+2}) \cdot b^{-1} + \ldots) \cdot b^{-1} + a_{-2}) \cdot b^{-1} + a_{-1}) \cdot b^{-1} .$$

Алгоритм з використанням схеми Горнера має меншу часову складність. Він не потребує знаходження степенів основи і в ньому використовуються лише операції додавання і множення, за рахунок чого прискорюються обчислення.

Приклад 3.2.3. Для числа $10011{,}1001_2$ знайдемо подання в десятковій системі, скориставшись схемою Горнера:

$$10011_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$$= (((2 + 0) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1 =$$

$$= (4 \cdot 2 + 1) \cdot 2 + 1 = 9 \cdot 2 + 1 = 18 + 1 = 19;$$

$$0{,}1001_2 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$$

$$= (((1 \cdot 2^{-1} + 0) \cdot 2^{-1} + 0) \cdot 2^{-1} + 1) \cdot 2^{-1} =$$

$$= (0{,}5 \cdot 0{,}5 \cdot 0{,}5 + 1) \cdot 0{,}5 = 1{,}125 \cdot 0{,}5 = 0{,}5625 .$$

Отже, $10011{,}1001_2 = 19{,}5625_{10}$.

При роботі з числами у двійковій системі числення часто використовують такі рівності:

$$1\underbrace{00\ldots00}_{n \text{ разів}}{}_2 = 2^n_{10} ,$$

$$\underbrace{11\ldots11}_{n \text{ разів}}{}_2 = 2^n_{10} - 1 .$$

Перехід від десяткового подання числа до двійкового. Спочатку розглянемо приклад одержання цифр десяткового числа, а потім за аналогією одержимо цифри двійкового числа, тобто перейдемо до подання числа в двійковій системі числення.

Приклад 3.2.4. Одержимо десяткові цифри цілого десяткового числа 142_{10}, виконавши ділення на 10:

$$\begin{array}{lll} 142:10= & 14 & \text{і остача } \mathbf{2}; \\ 14:10= & 1 & \text{і остача } \mathbf{4}; \quad \uparrow \\ 1:10= & 0 & \text{і остача } \mathbf{1}. \end{array}$$

Якщо остачі від ділення на 10 записати у зворотному порядку, то матимемо початкове число.

Тепер одержимо десяткові цифри дробового десяткового числа (з цілою частиною 0) $0{,}35_{10}$, виконавши множення на 10:

$$\begin{array}{lll} 0{,}35\cdot 10= & 3{,}5 & \text{ціла частина } \mathbf{3}\text{, дробова } 0{,}5; \quad \downarrow \\ 0{,}5\cdot 10= & 5 & \text{ціла частина } \mathbf{5}\text{, дробової нема.} \end{array}$$

Якщо після нуля з комою записати значення цілих частин у прямому порядку, то матимемо початкове число.

Аналогічно можна одержати двійкові цифри десяткового числа. При цьому цілу і дробову частини треба перетворювати окремо.

Приклад 3.2.5. Розглянемо число $142{,}35_{10}$. Одержимо двійкові цифри цілої частини, виконавши ділення на 2:

$$\begin{array}{lll} 142:2= & 71 & \text{і остача } \mathbf{0}; \\ 71:2= & 35 & \text{і остача } \mathbf{1}; \\ 35:2= & 17 & \text{і остача } \mathbf{1}; \\ 17:2= & 8 & \text{і остача } \mathbf{1}; \quad \uparrow \\ 8:2= & 4 & \text{і остача } \mathbf{0}; \\ 4:2= & 2 & \text{і остача } \mathbf{0}; \\ 2:2= & 1 & \text{і остача } \mathbf{0}; \\ 1:2= & 0 & \text{і остача } \mathbf{1}. \end{array}$$

Остачі від ділення запишемо у зворотному порядку: 10001110_2. Перевіримо правильність одержаного результату:

$$10001110_2 = 1\cdot 2^7 + 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 = 128 + 8 + 4 + 2 = 142_{10}.$$

Тепер одержимо двійкові цифри дробової частини, виконавши множення на 2:

$$
\begin{array}{lll}
0{,}35 \cdot 2 = & 0{,}7 & \text{ціла частина } \mathbf{0}, \text{ дробова } 0{,}7; \\
0{,}7 \cdot 2 = & 1{,}4 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0{,}4; \\
\mathbf{0{,}4} \cdot 2 = & 0{,}8 & \text{ціла частина } \mathbf{0}, \text{ дробова } 0{,}8; \\
0{,}8 \cdot 2 = & 1{,}6 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0{,}6; \quad \downarrow \\
0{,}6 \cdot 2 = & 1{,}2 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0{,}2; \\
0{,}2 \cdot 2 = & 0{,}4 & \text{ціла частина } \mathbf{0}, \text{ дробова } 0{,}4; \\
\mathbf{0{,}4} \cdot 2 = & 0{,}8 & \text{ціла частина } \mathbf{0}, \text{ дробова } 0{,}8...
\end{array}
$$

У даному випадку при перетворенні скінченного десяткового дробового числа $0{,}35_{10}$ у двійкове подання одержано періодичний (нескінченний) двійковий дріб $0{,}01(0110)_2$.

Отже:

$$142{,}35_{10} = 10001110{,}01(0110)_2 .$$

Приклад 3.2.6. Розглянемо десятковий дріб $0{,}6875_{10}$ (цей дріб можна подати звичайним дробом $\frac{11}{16} = \frac{11}{2^4}$, кратним степеневі двійки 2^{-4}) і для нього одержимо подання в двійковій системі числення:

$$
\begin{array}{lll}
0{,}6875 \cdot 2 = & 1{,}375 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0{,}375; \\
0{,}375 \cdot 2 = & 0{,}75 & \text{ціла частина } \mathbf{0}, \text{ дробова } 0{,}75; \quad \downarrow \\
0{,}75 \cdot 2 = & 1{,}5 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0{,}5; \\
0{,}5 \cdot 2 = & 1{,}0 & \text{ціла частина } \mathbf{1}, \text{ дробова } 0.
\end{array}
$$

У даному випадку при перетворенні скінченного десяткового дробового числа $0{,}6875_{10}$, кратного степеневі двійки, у двійкове подання одержано неперіодичний (скінченний) двійковий дріб $0{,}1011_2$.

Отже:

$$0{,}6875_{10} = 0{,}1011_2 .$$

Аналогічно, наприклад, $\frac{3}{4} = 0{,}75_{10} = 0{,}11_2$, $\frac{13}{32} = 0{,}40625_{10} = 0{,}01101_2$ тощо.

Алгоритм переходу від десяткового подання числа до двійкового:

К1. Виділити в десятковому числі цілу і дробову частини.

К2. Якщо ціла частина дорівнює нулю, то перейти на К3. Якщо ціла частина відмінна від нуля, то поділити її на 2 — буде одержано цілу частину і остачу (0 чи 1); запам'ятати, що виконувалося ділення; перейти на К2.

К3. Якщо ділення виконувалося, то одержані остачі від ділення записати у зворотному порядку — буде одержано двійкове подання цілої частини числа; інакше ціла частина залишається нульовою.

К4. Якщо дробова частина дорівнює нулю, то перейти на К5. Якщо дробова частина відмінна від нуля, то помножити її на 2 — буде одержано цілу частину (0 чи 1) і дробову; запам'ятати, що виконувалося множення. Якщо дробова частина повторюється (двійковий дріб — періодичний), то перейти на К5 чи передбачити іншу дію; інакше перейти на К4.

К5. Якщо множення виконувалося, то одержані після множення цілі частини записати у прямому порядку — буде одержано двійкове подання дробової частини числа; інакше дробова частина залишається нульовою.

К6. Сформувати з двійкових подань цілої і дробової частин двійкове подання числа.

Зауваження 1. Подання цілого числа в двійковому вигляді потребує приблизно в три рази більше розрядів, ніж у десятковому (наприклад, $3_{10} = 11_2$, $4_{10} = 100_2$. $9_{10} = 1001_{10}$, $37_{10} = 100101_2$, $99_{10} = 1100011_2$, $100_{10} = 1100100_2$, $586_{10} = 1001001010_2$, $1000_{10} = 1111101000_2$). У загальному випадку для запису числа $N > 0$ в системі числення з основою b потрібно $\lceil \log_b (N+1) \rceil$ (заокруглення до більшого), тобто приблизно $\log_b N \pm 1$ розрядів. При переході від основи b_1 до основи b_2 кількість цифр у поданні числа змінюється приблизно в $\log_{b_2} b_1$ разів (отже, при переході від десяткового до двійкового подання кількість цифр збільшується приблизно в $\log_2 10 \approx 3$ рази).

Зауваження 2. При переведенні десяткових дробів у двійкові, як правило, буде одержано періодичний двійковий дріб (наприклад, для чисел 0,1, 0,2, 0,3, 0,4, 0,6, 0,7, 0,8, 0,9). Періодичний дріб — нескінченний, а комп'ютер оперує лише зі скінченними числами, поданими скінченними послідовностя-

ми 0 і 1. Це призводить до помилок заокруглення. Наприклад, після виконання оператора мови Turbo Pascal `if 0.1+0.6=0.7 then write('true') else write('false')` буде одержано результат `false` (у мові Turbo Pascal для зберігання значень типу real виділяється 6 байтів пам'яті); у мові С (для зберігання значень типу double виділяється 8 байтів пам'яті, що дає більшу точність обчислень, але помилка заокруглення залишається) — `if (0.1+0.1+0.1==0.3) printf("true"); else printf("false");` `if (0.1+0.2==0.3) printf("true"); else printf("false");` — обидва оператори дають результат `false`; а також оператор `if (0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1==0.8) printf("true"); else printf("false");` — дає `false`; оператор `if (0.4+0.4==0.8) printf("true"); else printf("false");` — дає `true`). Щоб зменшити ці помилки, в комп'ютерах реалізовано спеціальні правила заокруглення [43, с. 16]. Але повністю позбутися помилок закруглення при проведенні комп'ютерних обчислень не можливо; можна тільки збільшити точність обчислень, збільшивши кількість розрядів для подання дробів (наприклад, якщо в якомусь віртуальному комп'ютері під дробову частину відвести чотири двійкові розряди, то значення числа $0{,}2_{10}$ буде зберігатися як $0{,}0011_2$ і якщо його перевести назад в десяткове подання, то одержимо $0{,}0011_2 = \frac{1}{8} + \frac{1}{16} = 0{,}1875_{10}$, якщо ж відвести вісім розрядів, то $0{,}00110011_2 = \frac{1}{8} + \frac{1}{16} + \frac{1}{128} + \frac{1}{256} = 0{,}19841875_{10}$ — точність збільшилася).

Зауваження 3. Десяткові дроби, кратні степеневі числа $\frac{1}{2}$ (наприклад, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$, $\frac{1}{8}$, $\frac{3}{8}$, $\frac{5}{8}$ тощо), переводяться в скінченні двійкові дроби. Тому робота з ними, як і з цілими числами, до помилок заокруглення не призводить.

Для деяких десяткових звичайних дробів виконуються такі відповідності:

$\frac{1}{2^n} = 0,\underbrace{00...00}_{(n-1)\text{ разів}}1$ (наприклад, $\frac{1}{2} = \frac{1}{2^1} = 0{,}1_2$, $\frac{1}{32} = \frac{1}{2^5} = 0{,}00001_2$);

$$1-\frac{1}{2^n}=0,\underbrace{11...11}_{n\ \text{разів}}\ (\text{наприклад, } 0,111_2=\frac{1}{2}+\frac{1}{4}+\frac{1}{8}=\frac{7}{8}=1-\frac{1}{8}=1-\frac{1}{2^3});$$

$$\frac{XX_{10}}{2^n}=0,\underbrace{XX...XX_2}_{n\ \text{цифр}},\ \text{де } X_2\in\{0,1\}\ (\text{наприклад, } \frac{13}{32}=0,01101_2);$$

$$\frac{1}{2^n-1}=0,(\underbrace{00...00}_{(n-1)\ \text{разів}}1)_2\ (\text{наприклад, } \frac{1}{31}=\frac{1}{32-1}=\frac{1}{2^5-1}=0,(00001)_2);$$

$$\frac{1}{2^n+1}=0,(\underbrace{00...00}_{n\ \text{разів}}\underbrace{11...11}_{n\ \text{разів}})_2\quad(\text{наприклад, }\quad \frac{1}{9}=\frac{1}{8+1}=\frac{1}{2^3+1}=0,(000111)_2).$$

Вісімкова й шістнадцяткова системи числення. Записувати числа, а також програмувати мовою машинних кодів, використовуючи 1 і 0, досить складно і треба бути дуже уважним. Тому для спрощення запису машинних кодів застосовують вісімкову (тепер досить рідко) й шістнадцяткову системи числення.

Для переходу від двійкового до вісімкового подання числа треба в двійковому поданні цілої частини виділити, ідучи справа наліво, по три розряди (якщо зліва розрядів не вистачає, то дописати потрібну кількість нулів), у двійковому поданні дробової частини іти зліва направо і теж виділити по три розряди (якщо справа розрядів не вистачає, то дописати потрібну кількість нулів), а потім скористатися відповідностями (таблиця 3.1):

Таблиця 3.1. Перехід від двійкового до вісімкового подання числа

Десяткове число	Двійкове число	Вісімкова цифра
0	000	0
1	001	1
2	010	2
3	011	3

Десяткове число	Двійкове число	Вісімкова цифра
4	100	4
5	101	5
6	110	6
7	111	7

Приклад 3.2.7. Подамо число $1111011000,11101_2$ у вісімковій системі:

$$\underbrace{\overset{\frown}{00}1}_{1}\underbrace{111}_{7}\underbrace{011}_{3}\underbrace{000}_{0},\underbrace{111}_{7}\underbrace{01\overset{\frown}{0}}_{2}.$$

Отже, $1111011000,11101_2=1730,72_8$.

Аналогічно здійснюється перехід від двійкового до шістнадцяткового подання, тільки замість трьох треба виділяти по чотири розряди і скористатися такою таблицею (таблиця 3.2) відповідностей:

Таблиця 3.2. Перехід від двійкового до шістнадцяткового подання числа

Десяткове число	Двійкове число	Шістнадцяткова цифра	Десяткове число	Двійкове число	Шістнадцяткова цифра
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Приклад 3.2.8. Подамо число $1111011000{,}11101_2$ у шістнадцятковій системі:

$$\underbrace{\overbrace{00}11}_{3}\underbrace{1101}_{D}\underbrace{1000}_{8},\underbrace{1110}_{E}\underbrace{1\overbrace{000}}_{8}.$$

Отже, $1111011000{,}11101_2 = 3D8{,}E8_{16}$.

Зворотний перехід від вісімкового і шістнадцяткового подання до двійкового виконується аналогічно на основі таблиць відповідностей.

Прямий перехід від десяткового подання до вісімкового і шістнадцяткового виконується, як і для двійкового, тільки ділити чи множити треба відповідно на 8 і 16.

Приклад 3.2.9. Щоб подати десяткове число $984{,}90625_{10}$ у шістнадцятковій системі, виконаємо для цілої частини ділення, а для дробової множення на 16:

$$\begin{array}{lll} 984:16 = & 61 & \text{і остача } \mathbf{8}; \\ 61:16 = & 3 & \text{і остача } \mathbf{13} \sim \mathbf{D}; \uparrow \\ 3:16 = & 0 & \text{і остача } \mathbf{3}. \end{array}$$

$$\begin{array}{lll} 0{,}90625 \cdot 16 = & 14{,}5 & \text{ціла частина } \mathbf{14} \sim \mathbf{E}, \text{ дробова } 0{,}5; \downarrow \\ 0{,}5 \cdot 16 = & 8 & \text{ціла частина } \mathbf{8}, \text{ дробова } 0. \end{array}$$

Отже, $984{,}90625_{10} = 3D8{,}E8_{16}$.

Виконання операцій над числами у двійковій системі числення. Арифметичні дії у позиційних системах числення з різними основами, як і в десятковій, виконують порозрядно.

Додавання в десятковій системі числення виконують за модулем 10, а в двійковій — за модулем 2:

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	10

При цьому $1_2 + 1_2 + 1_2 = 11_2$. Як і в десятковій системі числення, у двійковій теж застосовують перенесення значень у старші розряди.

Приклад 3.2.10. Знайдемо суму чисел $217_{10} = 11011001_2$ і $93_{10} = 1011101_2$:

$$\begin{array}{l} \text{перенесення} \\ \\ \\ \\ \end{array} \begin{array}{r} {\scriptstyle 1\;\;1\;\;\;\;} \\ +\begin{array}{r} 2\;1\;7 \\ 9\;3 \\ \hline 3\;1\;0 \end{array} \end{array} \qquad \begin{array}{r} {\scriptstyle 1\;\;1\;\;\;\;\;1\;\;1\;\;\;\;\;\;\;\;\;1\;\;\;\;} \\ +\begin{array}{r} 1\;1\;0\;1\;1\;0\;0\;1 \\ 1\;0\;1\;1\;1\;0\;1 \\ \hline 1\;0\;0\;1\;1\;0\;1\;1\;0 \end{array} \end{array}.$$

Дію віднімання можна виконувати аналогічно. Операція віднімання у двійковій системі числення дає результати:

x	y	$x-y$
0	0	0
0	1	-1
1	0	1
1	1	0

Як і в десятковій системі числення, у двійковій теж застосовують позичання одиниць у старших розрядів. При цьому $10_2 - 1_2 = 1_2$, $100_2 - 1_2 = 11_2$, $11_2 - 1_2 = 10_2$.

Приклад 3.2.11. Знайдемо різницю чисел $217_{10} = 11011001_2$ і $93_{10} = 1011101_2$:

$$\begin{array}{l} \text{позичання} \\ \\ \\ \\ \end{array} \begin{array}{r} {\scriptstyle -1\;\;\;\;\;\;\;\;} \\ -\begin{array}{r} 2\;1\;7 \\ 9\;3 \\ \hline 1\;2\;4 \end{array} \end{array} \qquad \begin{array}{r} {\scriptstyle -1\;\;-1\;\;-1\;\;-1\;\;-1\;\;-1\;\;\;\;\;\;\;\;} \\ -\begin{array}{r} 1\;1\;0\;1\;1\;0\;0\;1 \\ 1\;0\;1\;1\;1\;0\;1 \\ \hline 0\;1\;1\;1\;1\;1\;0\;0 \end{array} \end{array}.$$

Операція множення у двійковій системі числення дає результати:

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Множення й ділення у двійковій арифметиці можна виконати, як і в десятковій арифметиці, — у стовпчик.

Приклад 3.2.12. Знайдемо добуток і частку чисел $321_{10} = 101000001_2$ і $14_{10} = 1110_2$:

$$\begin{array}{r} 321 \\ \times \quad 14 \\ \hline 1284 \\ +\,321 \\ \hline 4494_{10} \end{array} \qquad \begin{array}{l} 321 \mid 14 \\ -22 \mid 22 \\ \hline 41 \\ -28 \\ \hline 13 \text{ – остача} \end{array}$$

$$\begin{array}{r} 101000001 \\ \times \qquad 1110 \\ \hline 000000000 \\ 101000001 \\ +\;101000001 \\ 101000001 \\ \hline 1000110001110 \end{array} \qquad \begin{array}{l} 101000001 \mid 1110 \\ -1110 \qquad\; \mid 10110 \\ \hline 11000 \\ -1110 \\ \hline 10100 \\ -1110 \\ \hline 1101 \text{ – остача} \end{array}$$

У комп’ютері при роботі з дійсними числами для прискорення обчислень ділення x / y реалізовано через множення x на $1/y$. При цьому в процесорі[13] є таблиця початкових наближень і методом Ньютона (див. приклад 6.3.10) за 2-3 ітерації обчислюється досить точне значення $1/y$. До використання такого алгоритму операція ділення реалізувалася операціями зсуву і віднімання.

Вище для переходу від двійкового подання цілої частини числа до десяткового було використано множення на степені двійки. Але, використовуючи ділення у двійковій системі числення, можна здійснити перехід від двійкового подання числа до десяткового (аналогічно до того, як вище здійснювали перехід від десяткового подання до двійкового).

[13] У 1994 році в процесорі Pentium 1 при застосуванні цього алгоритму в модулі операцій з плаваючою арифметикою в таблиці початкових наближень було зроблено помилку, що в деяких обчисленнях призводило до неправильних результатів. Фірма Intel безплатно міняла процесори (це їй обійшлося майже в 0,5 млрд доларів).

Приклад 3.2.13. Перейдемо від подання числа 10001110_2 у двійковій системі числення до його подання в десятковій системі, застосовуючи ділення на $10_{10} = 1010_2$:

$$\begin{array}{r|l}
-\,10001110 & 1010 \\ \cline{2-2}
1010 & 1110 \\ \cline{1-1}
-\,1111 & \\
1010 & \\ \cline{1-1}
-\,1011 & \\
1010 & \\ \cline{1-1}
-\,10 & \\
0 & \\ \cline{1-1}
10_2 = 2_{10} & -\text{ остача}
\end{array}
\qquad
\begin{array}{r|l}
-\,1110 & 1010 \\ \cline{2-2}
1010 & 1 \\ \cline{1-1}
100_2 = 4_{10} & -\text{ остача}
\end{array}
\qquad
\begin{array}{r|l}
-\,1 & 1010 \\ \cline{2-2}
0 & 0 \\ \cline{1-1}
1_2 = 1_{10} & -\text{ остача}
\end{array}$$

Записавши остачі у зворотному порядку, матимемо число 142_{10}.

Аналогічний результат одержимо, скориставшись множенням на степені двійки:

$$10001110_2 = 1 \cdot 2^7 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 = 128 + 8 + 4 + 2 = 142_{10}.$$

3.3. Бінарні й небінарні коди Грея

Серед двійкових кодів, крім звичайного двійкового коду, важливе значення мають коди Грея. Коди Грея є непозиційною системою запису цілих натуральних чисел. У кодуванні Грея два послідовні коди відрізняються лише значенням одного розряду (перший і останній код вважаються сусідніми; таке кодування називають циклічним). Наприклад, якщо розглянути трирозрядні стандартні коди Грея для чисел від 0 до 7, то послідовність кодів буде такою: 000, 001, 011, 010, 110, 111, 101, 100. Існує досить багато різних двійкових кодів Грея: стандартний, збалансований, комплементарний, довгосерійний, нелокальний, монотонний, безтрендовий та ін. Ці коди особливо важливі в тих випадках, коли аналогові сигнали перетворюються в цифрові чи навпаки, оскільки вони при цьому мінімізують помилки перетворення. Їх використовують у кодово-імпульсній модуляції, методі аналогової передачі цифрових сигналів, у датчиках-енкодерах, при нумерації доріжок на жорстких дисках (для збільшення надійності роботи системи оптичних фотодіодів при встановленні кута

повороту дисків-носіїв інформації), для спрощення виявлення й виправлення помилок у системах зв'язку, при кодуванні генетичних ознак у теорії генетичних алгоритмів та ін. Використання кодів Грея дає можливість зменшити величину помилок від завад, які виникають при передачі інформації.

Послідовність *стандартних* (їх ще називають *віддзеркаленими*) *кодів Грея* можна визначити різними способами. Одним із них є рекурентне визначення. Якщо послідовність *n*-розрядних кодів Грея позначити G_n, то:

$$G_0 = \varnothing,$$

$$G_{n+1} = \{0G_n, 1G_n^R\},$$

де $n = 1, 2, 3, \ldots$, G_n^R — зворотна послідовність кодів. Наприклад:

$G_0 = \varnothing$,

$G_1 = \{0\varnothing, 1\varnothing\} = \{0,1\}$,

$G_2 = \{0\{0,1\}, 1\{1,0\}\} = \{00, 01, 11, 10\}$,

$G_3 = \{0\{00,01,11,10\}, 1\{10,11,01,00\}\} = \{000,001,011,010,110,111,101,100\}$

і т.д.

Щоб при програмній реалізації алгоритму не виконувати перестановку елементів масиву у зворотному порядку, зворотну послідовність кодів G_n^R можна одержати за формулою:

$$G_n^R = G_n \oplus 1\underbrace{00\ldots0}_{(n-1)\,\text{раз}},$$

де знак $\oplus$ позначає порозрядне додавання за модулем 2 (без перенесення одиниць у старші розряди). Наприклад, знайдемо G_2^R, якщо $G_2 = \{00, 01, 11, 10\}$:

$$\oplus\begin{array}{cc}0&0\\ 1&0\\ \hline 1&0\end{array}\qquad \oplus\begin{array}{cc}0&1\\ 1&0\\ \hline 1&1\end{array}\qquad \oplus\begin{array}{cc}1&1\\ 1&0\\ \hline 0&1\end{array}\qquad \oplus\begin{array}{cc}1&0\\ 1&0\\ \hline 0&0\end{array}.$$

Маємо: $G_2^R = \{10, 11, 01, 00\}$.

Якщо число подано в двійковій системі числення $b = b_n b_{n-1} b_{n-2} \ldots b_1 b_0$, то значення розрядів коду Грея $g = g_n g_{n-1} g_{n-2} \ldots g_1 g_0$ можна знайти за формулою:

$$g_i = b_i \oplus b_{i+1},$$

де $0 \le i \le n$, $b_{n+1} = 0$, знак $\oplus$ позначає додавання за модулем 2, або в термінах порозрядної арифметики це можна записати так:

$$g(b) = b \oplus \lfloor b/2 \rfloor,$$

де $\lfloor * \rfloor$ означає заокруглення до меншого (щоб поділити число на 2 і заокруглити до меншого, треба в двійковому поданні числа здійснити зсув на один розряд вправо, тобто, відкинути наймолодший розряд; так, якщо є число 101_2, то матимемо 10_2). Одержимо код Грея для числа $5_{10} = 101_2$

$$\begin{array}{r} \oplus\begin{array}{ccc} 1 & 0 & 1 \\ & 1 & 0 \end{array} \\ \hline \begin{array}{ccc} 1 & 1 & 1 \end{array} \end{array}$$

Програмну реалізацію алгоритму мовою C подано в прикладі 10.4.1.

Будь-яке ціле число у десятковій системі числення можна подати $a = a_n a_{n-1} a_{n-2} \ldots a_1 a_0 = \sum_{k=-m}^{n} a_k 10^k$, тобто як суму добутків значень цифр числа на відповідний степінь основи 10; у двійковій системі числення — $b = b_n b_{n-1} b_{n-2} \ldots b_1 b_0 = \sum_{k=0}^{n} b_k 2^k$, тобто відповідне десяткове число можна одержати як суму добутків значень цифр числа на відповідний степінь основи 2. При знаходженні відповідного двійковому кодові Грея $g = g_n g_{n-1} g_{n-2} \ldots g_1 g_0$ (який є непозиційним поданням числа) десяткового числа такої можливості нема. За рахунок цього коди Грея важко декодувати. Тому для їхньої обробки треба або мати таблицю відповідності, або спочатку код треба перевести у звичайне двійкове число, а потім одержати відповідне десяткове. При переведенні двійкового коду Грея у звичайне двійкове число треба виконати такі перетворення:

$$b_n = g_n,$$

$$b_i = b_{i+1} \oplus g_i,$$

де $i = n-1, n-2, \ldots, 0$.

Оскільки

$$b_i = b_{i+1} \oplus g_i = b_{i+2} \oplus g_{i+1} \oplus g_i = \ldots = b_{n+1} \oplus g_n \oplus g_{n-1} \oplus \ldots \oplus g_{i+1} \oplus g_i,$$

$$b = b_n b_{n-1} b_{n-2} \ldots b_1 b_0 = 0 b_n b_{n-1} b_{n-2} \ldots b_1 b_0, \text{ тобто } b_{n+1} = 0,$$

то одержимо для обчислень формулу:

$$b_i = \bigoplus_{j=i}^{n} g_j ,$$

де $i = n-1, n-2, ..., 0$. Наприклад, якщо $g = 10011$, то:

$$\begin{array}{cccccc} & b_4 & b_3 & b_2 & b_1 & b_0 \\ & 1 & 0 & 0 & 1 & 1 \\ & & 1 & 0 & 0 & 1 \\ \oplus & & & 1 & 0 & 0 \\ & & & & 1 & 0 \\ & & & & & 1 \\ \hline & 1 & 1 & 1 & 0 & 1 \end{array}$$

тобто маємо відповідне двійкове число $b = 11101_2$, якому відповідає десяткове число 29_{10}. Програмну реалізацію розглянутого алгоритму мовою C подано в прикладі 10.4.2.

Крім досить добре досліджених двійкових кодів Грея, широкі застосування також мають недвійкові (небінарні) коди, пов'язані з генерацією векторів з цілими компонентами. Слід зауважити, що в таких кодах кожна координата g_i $(n+1)$-вимірного вектора $(g_n, g_{n-1}, ..., g_1, g_0)$ може набувати значень у діапазоні з іншою верхньою межею (від 0 до d_i, $i = \overline{0,n}$), тобто кожен розряд має свою основу (див. приклад 10.4.3). Кількість усіх таких кодів становитиме $\prod_{i=0}^{n}(d_i + 1)$.

Прикладом небінарного коду Грея є *рефлексивний десятковий код*, у якому кожна координата почергово змінюється спочатку від 0 до 9, а потім від 9 до 0: 000, 001, …, 009, 019, 018, …, 011, 010, 020, 021, …, 091, 090, 190, 191, …, 900 (тут закодовано числа від 0 до 999, тобто всіх кодів є $(9+1)\cdot(9+1)\cdot(9+1) = 1000$).

Незалежно, які коди Грея розглядати — бінарні чи небінарні, кодові відстані між двома сусідніми кодами завжди дорівнюють 1. Наприклад, для звичайних бінарних 5-розрядних кодів чисел $15_{10} = 01000_G$ і $16_{10} = 11000_G$ $d(01000, 11000) = |0-1| + |1-1| + |0-0| + |0-0| + |0-0| = 1$, для небінарних 3(1,5,1)-кодів чисел $15_{10} = 140_G$ і $16_{10} = 130_G$ $d(140, 130) = |1-1| + |4-3| + |0-0| = 1$, а для звичайних 5-розрядних двійкових чисел $15_{10} = 01111_2$ і $16_{10} = 10000_2$

$d(01111, 10000) = |0-1| + |1-0| + |1-0| + |1-0| + |1-0| = 5$. З наведеного прикладу видно, що для бінарних кодів кодова відстань визначається кількістю одноіменних розрядів з різними значеннями. Таким чином, при зміні коду Грея на сусідній (15 на 16) треба виконати тільки одну дію, а при зміні на сусідній звичайного двійкового коду — 5 дій, що потребує значно більше часу. Також завдяки змінам тільки в сусідніх розрядах коди Грея дають змогу зменшити кількість помилок від перешкод, які виникають при передачі інформації каналами зв'язку.

3.4. Подання числових даних у комп'ютері: цілі й дійсні числа

Цілі числа і дробові в комп'ютері кодуються по-різному, але як і вся інформація в сучасних комп'ютерах, вони подаються у двійковому вигляді — використовується двійкова система числення.

У комп'ютерах для зберігання значень констант і змінних різних типів відводиться різна за довжиною, але фіксована для даного типу, ділянка пам'яті. Наприклад, у більшості реалізацій мови програмування С для даних типу char і типу unsigned char виділяється 1 байт пам'яті (8 бітів або ще кажуть: 8 розрядів), для даних типу int і float — 4 байти (32 біти), long long int і double — 8 байтів (64 біти); у мові Turbo Pascal для даних типу byte і shortint виділяється 1 байт (8 бітів) пам'яті, integer і word — 2 байти (16 бітів) , longint — 4 байти (32 біти), comp — 8 байтів (64 біти), real — 6 байтів (24 біти).

Додатковий код числа. Формат цілого числа в комп'ютері. У комп'ютері віднімання реалізується тією ж мікросхемою (суматором), що й додавання. Нехай для зберігання цілого числа відводиться 2 байти, тобто 16 бітів (розрядів). Тоді, наприклад, число 5_{10} кодується як $0000\,0000\,0000\,0101_2$ або для полегшення запису можна використати шістнадцятковий код 0005_{16}.

Для подання від'ємних чисел у комп'ютері використовується *додатковий код*:

$$\text{додатковий код} = \text{зворотний код} + 1,$$

де зворотний код є логічним доповненням (інверсією: 1 треба замінити на 0, а 0 — на 1) коду додатного числа.

Для переходу від додаткового коду до звичайного застосовується той самий алгоритм — спочатку треба інвертувати додатковий код, а потім додати 1. Можна також здійснити зворотний хід — від додаткового коду відняти 1 і одержаний результат інвертувати.

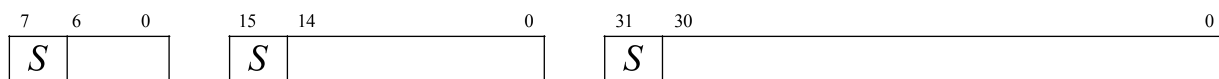
Приклад 3.4.1. Нехай для зберігання коду цілого числа відведено 2 байти (16 бітів) пам'яті. Одержимо код числа -5_{10}:

Код числа 5_{10}	0000000000000101_2
Зворотний код числа 5_{10}	1111111111111010
Додатковий код	1111111111111011,

тобто від'ємне число -5_{10} у комп'ютері кодується так: 1111111111111011.

Якщо ціле число без знака (unsigned), то всі розряди виділеної ділянки пам'яті містять значення цього числа.

Цілі числа зі знаком (signed) у пам'яті зберігаються в спеціальному форматі. Перший зліва розряд коду вказує на знак числа: 0 — число додатне, 1 — число від'ємне; решта розрядів містять значення числа. Розряди, які розміщуються справа, називають *молодшими розрядами*, зліва — *старшими розрядами числа*. Якщо для подання цілого числа виділяється 8, 16, 32 тощо розряди, то відповідно пам'ять розподіляється так:



де *S* — *знаковий розряд* числа: $S(+) = 0$, $S(-) = 1$.

При виконанні арифметичних операцій може відбуватися вихід за межі розрядної сітки і за рахунок використання такого формату для подання цілих чисел можуть з'являтися непередбачувані результати. Наприклад, при обчисленні значення факторіалу *n!*, якщо *n* набуває значення, більшого від певного числа (у мові С для даних типу char і unsigned char більшого від 5, для short int — 7, unsigned short int — 8, int і unsigned int — 12, long long int і unsigned long long int — 20; у мові Turbo Pascal для byte і shortint — 5, integer — 7,

word — 8, longint — 12, comp — 20), результат буде неправильним і навіть можуть з'явитися від'ємні числа (якщо в розряд знака числа потрапить 1). При виконанні звичайного додавання навіть невеликих чисел (типу char) теж може відбутися вихід за межі розрядної сітки.

Приклад 3.4.2. Нехай дві змінні типу char (для їхнього зберігання виділяється по 1 байту пам'яті, тобто по 8 розрядів; вони можуть набувати значень від –128 до 127) набули значень 125 і 10. Знайдемо їхню суму:

$$
\begin{array}{r} + \begin{array}{ccc} 1 & 2 & 5 \\ & 1 & 0 \\ \hline 1 & 3 & 5 \end{array} \end{array}
\qquad\qquad
+ \begin{array}{cccccccc} \mathit{0} & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ \mathit{0} & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ \hline \mathit{1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}.
$$

У знаковому розряді з'явилася 1, яка вказує на від'ємний результат і на додатковий код. Тобто це не число -7_{10}, а число -121_{10} (перейдемо від додаткового коду до звичайного: інвертуємо додатковий код 10000111 => 01111000 і додамо 1 до результату $01111000+1=01111001_2= 121_{10}$; або віднімемо 1 від вмісту пам'яті 10000111–1=10000110 і одержаний результат інвертуємо $01111001_2= 121_{10}$). Такий самий результат одержано при виконанні програми мовою С (фрагмент програми подано нижче):

```
...
char n=125, m=10, k;
  k=n+m;
  printf("k=%d",k);
...
```

```
k=-121
```

Розглянемо приклади роботи з від'ємними числами в комп'ютері.

Приклад 3.4.3. Нехай змінні типу char набули значень 10, 27 і 37. Треба обчислити значення $10-37$, $37-10$, $-10-27$, використовуючи додатковий код. У комп'ютері ці значення (з врахуванням знака +, код 0 якого зберігається в найстаршому розряді) подаються так:

$$
\begin{array}{lccccccccc}
+10 & : & \mathit{0} & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
+27 & : & \mathit{0} & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
+37 & : & \mathit{0} & 0 & 1 & 0 & 0 & 1 & 0 & 1
\end{array}.
$$

Відповідні від’ємні значення (інверсія плюс один) будуть такими:

$$
\begin{array}{lcl}
-10 & : & 11110101+1= \mathit{1}\;1\;1\;1\;0\;1\;1\;0 \\
-27 & : & 11100100+1= \mathit{1}\;1\;1\;0\;0\;1\;0\;1 \\
-37 & : & 11011010+1= \mathit{1}\;1\;0\;1\;1\;0\;1\;1 .
\end{array}
$$

Тоді:

$$
10-37 \;:\; \begin{array}{r} \mathit{0}\;0\;0\;0\;1\;0\;1\;0 \\ +\;\mathit{1}\;1\;0\;1\;1\;0\;1\;1 \\ \hline \mathit{1}\;1\;1\;0\;0\;1\;0\;1_2 \end{array} = -27_{10} \;;
$$

$$
37-10 \;:\; \begin{array}{r} \mathit{0}\;0\;1\;0\;0\;1\;0\;1 \\ +\;\mathit{1}\;1\;1\;1\;0\;1\;1\;0 \\ \hline \mathbf{1}\;\mathit{0}\;0\;0\;1\;1\;0\;1\;1_2 \end{array} = 27_{10} \;;
$$

$$
-10-27 \;:\; \begin{array}{r} \mathit{1}\;1\;1\;1\;0\;1\;1\;0 \\ +\;\mathit{1}\;1\;1\;0\;0\;1\;0\;1 \\ \hline \mathbf{1}\;\mathit{1}\;1\;0\;1\;1\;0\;1\;1_2 \end{array} = -37_{10} \;.
$$

У другому й третьому випадках при виконанні операції додавання з’явився перед знаковим розрядом ще один розряд. Значення цього розряду ніяк не впливає на результат — воно не враховується і просто відкидається.

Зауваження 1. В арифметиці в додаткових кодах 1, яка виноситься за розрядну сітку, не враховується.

Зауваження 2. Якщо для подання числа використовувати, наприклад, 4 байти пам’яті (32 розряди), то додатковий код числа a визначають як $2^{32}-a$. Це пов’язано з тим, що число 32 кодується 33 розрядами (одна одиниця й тридцять два нулі):

значення розряду	1	0	0	...	0
степінь двійки	32	31	30	...	0

і при цьому тридцять третій розряд виходить за межі розрядної сітки і відкидається. Отже, маємо: $0-a=-a$. Аналогічно, якщо для подання числа використовувати 8 байтів, то додатковий код числа a визначається як $2^{64}-a$.

Якщо працювати з числами, які займають 4 чи 8 байтів пам’яті, то користувачеві легше проводити обробку не двійкового коду, а шістнадцяткового. При цьому інверсії для шістнадцяткових цифр будуть такими:

цифра	0	1	2	3	4	5	6	7	8	9	*A*	*B*	*C*	*D*	*E*	*F*
інверсія	*F*	*E*	*D*	*C*	*B*	*A*	9	8	7	6	5	4	3	2	1	0

(наприклад, $6_{16} = 0110_2$, тоді інверсія $1001_2 = 9_{16}$).

Приклад 3.4.4. Нехай для зберігання числа 5 виділено 4 байти пам'яті; тоді:

$$5_{10} = \underbrace{0000}\underbrace{0000}\underbrace{0000}\underbrace{0000}\underbrace{0000}\underbrace{0000}\underbrace{0000}\underbrace{0101}_2 = 00000005_{16}\ .$$

Отже:

Код числа 5_{10}	$0000\,0005_{16}$
Зворотний код числа 5_{10}	*FFFF FFFA*
Додатковий код	*FFFF FFFB*

Приклад 3.4.5. Обчислимо значення $13_{10} - 13_{10}$ в шістнадцятковій системі числення, використовуючи додатковий код. Виконаємо перетворення:

$$13_{10} = 0000000D_{16},$$

$$-13_{10} = FFFFFFF2_{16} + 1_{16} = FFFFFFF3_{16},$$

$$13 - 13\ :\quad \begin{array}{r} \ 0\ 0\ 0\ 0\ 0\ 0\ 0\ D \\ +\ F\ F\ F\ F\ F\ F\ F\ 3 \\ \hline \mathbf{1}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0_{16} \end{array} = 0_{16}\ .$$

Подання дійсних чисел у комп'ютері. Стандарт IEEE 754. Для виконання обчислень з дійсними числами (ще кажуть: числами з плаваючою точкою, рухомою комою) у двійковому коді розроблено стандарт IEEE (Institute of Electrical and Electronics Engineers) 754-1985 [42] (стандарт в 2008 і 2019 роках доповнено новими форматами), який використовується в більшості мікропроцесорів і логічних пристроїв, а також програмних засобів.

Відповідно до стандарту IEEE при роботі з дійсними числами одинарної і подвійної точності, для яких відповідно виділяється 4 і 8 байтів пам'яті (у мові C це дані типів float і double), 23 і 52 біти зберігають *мантису* — значущі двійкові цифри числа (відповідно це 7-8 і 15-16 десяткових розрядів), а також 8 і 11 бітів зберігають значення порядку — *експоненту*. Такі формати дають діапазони значень 1,4E-45…3,4E+38 і 4,9E-324…1,7E+308.

Розглянемо, наприклад, число $123{,}375_{10}$. В експоненційному вигляді його можна подати по-різному: $123{,}375 \cdot 10^{0}$, $12{,}3375 \cdot 10^{+1}$, $123375 \cdot 10^{-3}$ тощо; у цих поданнях мантиси відповідно рівні $123{,}375$, $12{,}3375$, 123375, а експоненти (показники степенів основи числення 10) — 0, 1, –3. Якщо мантиса M десяткового числа набуває значення з діапазону $1 \le M < 10$, то число подано в *нормалізованому експоненційному вигляді* (наприклад, $1{,}23375 \cdot 10^{+2}$).

Якщо мантиса M десяткового числа набуває значення з діапазону $0 \le M < 1$, то число подано в *денормалізованому експоненційному вигляді* (наприклад $0{,}123375 \cdot 10^{+3}$).

У двійковій системі числення десяткове число $123{,}375_{10} = 1111011{,}011_{2}$. Тоді $1{,}111011011 \cdot 10_{2}^{+110}$ (тут $6_{10} = 110_{2}$) — це нормалізований експоненційний вигляд числа ($1 \le M < 2$); $0{,}1111011011 \cdot 10_{2}^{+111}$ — денормалізований експоненційний вигляд ($0 \le M < 1$). Таким чином, у двійковій системі числення при нормалізованому поданні числа ціла частина мантиси дорівнює 1, а при денормалізованому — 0.

Дійсні числа можуть бути як додатними, так і від'ємними — в пам'яті в знаковому розряді, як і для цілих чисел, зберігається інформація про знак числа: $S(+) = 0$, $S(-) = 1$. Особливістю двійкового нормалізованого вигляду є те, що запис мантиси завжди починається з 1 (в інших системах числення мантиса такої властивості не має, бо $1 \le M < b$, де $b > 2$ — основа системи числення). Тому перша 1 мантиси в пам'яті не зберігається — вона є неявною (прихованою); зберігається *залишок мантиси*. Крім того, оскільки експонента може бути як додатною, так і від'ємною, то, щоб не зберігати інформацію про її знак, до значення експоненти у випадку зберігання значення числа в 4 байтах пам'яті додається число 127 (у нашому прикладі для числа $1{,}111011011 \cdot 10_{2}^{+110}$ *зміщена експонента* буде дорівнювати $6_{10} + 127_{10} = 133_{10} = 10000101_{2}$), а у випадку зберігання значення числа у 8 байтах — 1023.

Формат для зберігання значень дійсних чисел одинарної точності, які займають 4 байти пам'яті, має вигляд:

Більшість дійсних чисел у комп'ютері подається в нормалізованому вигляді (з неявним бітом мантиси 1 і ненульовим порядком), а числа, близькі до 0, — у денормалізованому (з порядком 0 і неявним старшим бітом мантиси 0). Також є спеціальні формати для подання нуля, нескінченностей і NaN (Not a Number):

x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx	— $XXXXXXXX_{16}$, де $x \in \{0,1\}$ і в полі експоненти не всі x одночасно рівні 0 чи 1 — нормалізовані числа;
x	00000000	xxxxxxxxxx xxxxxxxxxx xxx	— $x00000000xxxXXXXX_{16}$, де $x \in \{0,1\}$ і не всі x рівні 0 — денормалізовані числа;
0	00000000	00000000000000000000000	— 00000000_{16} — число +0;
1	00000000	00000000000000000000000	— 80000000_{16} — число –0;
0	11111111	00000000000000000000000	— $7F800000_{16}$ — число $+\infty$;
1	11111111	00000000000000000000000	— $FF800000_{16}$ — число $-\infty$;
x	11111111	xxxxxxxxxx xxxxxxxxxx xxx	— $\left.\begin{matrix}7F\\ FF\end{matrix}\right\}1xxxXXXXX_{16}$, де $x \in \{0,1\}$ і не всі x рівні 0 — не число, тобто NaN (результат недопустимої операції).

Хоч множина дійсних чисел (навіть лише з деякого проміжку) нескінченна, дійсні числа, які можна подати в форматах IEEE 754, утворюють скінченну множину. Тобто неперервна *множина дійсних чисел дискретизується* і подається в комп'ютері своєю дискретною підмножиною. За рахунок цього

більшість дійсних чисел подаються з помилкою. При цьому абсолютна максимальна помилка дорівнює половині кроку дискретизації чисел, який подвоюється зі збільшенням експоненти двійкового числа на одиницю — чим далі від нуля, тим крок чисел буде більшим.

Крок дискретизації нормалізованих чисел одинарної точності (у мові C тип float) дорівнює $2^{E-127-23} = 2^{E-150}$, а подвійної точності (у мові C тип double) — $2^{E-1023-52} = 2^{E-1075}$, де E — значення зміщеної експоненти (максимальна абсолютна помилка дорівнює половині кроку числа — 2^{E-151} і 2^{E-1076}; відносна помилка — $(2^{E-151} / F) \cdot 100\%$ і $(2^{E-1076} / F) \cdot 100\%$, де F — десяткове значення числа), крок денормалізованих чисел відповідно дорівнює 2^{E-149} і 2^{E-1074} (максимальна абсолютна помилка — 2^{E-150} і 2^{E-1075}; відносна помилка — $(2^{E-150} / F) \cdot 100\%$ і $(2^{E-1075} / F) \cdot 100\%$). Максимально можлива відносна помилка для чисел одинарної точності приблизно становить $11{,}92 \cdot 10^{-6}\%$, для подвійної точності — $2{,}22 \cdot 10^{-14}\%$.

Якщо значущі цифри числа не поміщаються у відведену форматом ділянку пам'яті, то числа заокруглюються. Стандартом IEEE 754 передбачено *п'ять правил заокруглення*:

— до ближчого числа з прив'язкою до парного (стосується чисел з дробовою частиною 0,5 при заокругленні до цілого: $12{,}5 \approx 12$, $13{,}5 \approx 14$, $-8{,}5 \approx -8$; аналогічно, якщо число, яке закінчується цифрою 5, заокруглювати до десятків, то: $125 \approx 120$, $135 \approx 140$, $-85 \approx -80$; при заокругленні до десятих, тобто до одного знака після коми: $1{,}25 \approx 1{,}2$, $1{,}35 \approx 1{,}4$, $-0{,}85 \approx -0{,}8$ тощо), яке використовується за замовчуванням відповідно до стандарту 2008 року. Використання такого заокруглення сприяє тому, що в середньому похибка не накопичується;

— до ближчого з прив'язкою до нескінченності ($12{,}5 \approx 13$, $-7{,}5 \approx -8$);

— до нуля ($12{,}5 \approx 12$, $-7{,}5 \approx -7$);

— до плюс нескінченності ($12{,}5 \approx 13$, $-7{,}5 \approx -7$);

— до мінус нескінченності ($12{,}5 \approx 12$, $-7{,}5 \approx -8$).

Розглянемо приклади, які демонструють особливості роботи з дійсними числами в комп’ютері.

Приклад 3.4.6. Помилка, пов’язана з точністю подання чисел з плаваючою точкою в комп’ютері:

```
...
  float a, b, c;
  a=123456789;
  b=123456788;
  c=a-b;
  printf(" a=%f  b=%f  c=%f\n", a, b, c);
...
```

```
a=123456792.000000  b=123456784.000000  c=8.000000
```

Якщо порахувати результат на папері, то c=1, але програма видає значення c=8. Такий результат одержано за рахунок того, що двійкове подання десяткових чисел $a = 123456789_{10} = 111010110111100110100010101_2$ і $b = 123456788_{10} = 111010110111100110100010100_2$ потребує для свого розміщення в пам’яті по 27 розрядів, але для зберігання мантиси числа типу float виділяється тільки 23+1(неявний розряд) =24 розряди, тобто тут 3 молодші розряди числа не поміщаються в розрядну сітку. Тому числа заокруглюються і при цьому, якщо за розрядну сітку виходить число 0,5, то використовується заокруглення до парного. У числі

$$a = 123456789_{10} = 1\underbrace{1101011011110011010001\mathbf{0}}_{\text{23 розряди}}\mathbf{101}_2$$

у розрядах, які виходять за розрядну сітку, міститься число $0{,}101_2 = 0{,}625_{10} > 0{,}5_{10}$, тому число a заокруглюється до більшого:

$$1\underbrace{1101011011110011010001\mathbf{1}}_{\text{23 розряди}}\mathbf{000}_2 = 123456792_{10}.$$

У числі

$$b = 123456788_{10} = 1\underbrace{1101011011110011010001\mathbf{0}}_{\text{23 розряди}}\mathbf{100}_2$$

у розрядах, які виходять за розрядну сітку, міститься число $0{,}100_2 = 0{,}5_{10}$, тому число b заокруглюється до парного, тобто до 0:

$$\underbrace{11101011011110011010001}_{23\ \text{розряди}}\mathbf{0000}_2 = 123456784_{10}.$$

Оскільки в розрядну сітку не поміщалися 3 молодші розряди цілої частини числа, то крок дискретизації дорівнює $2^3 = 8$.

Приклад 3.4.7. Помилка, пов'язана з різною точністю зберігання в пам'яті значень типу float і типу double:

```
...
  float a;
  double b, c;
  a=123456789.123456;
  b=123456789.123456;
  c=a-b;
  printf(" a=%f  b=%f  c=%f\n", a, b, c);
...
```

```
a=123456792.000000  b=123456789.123456  c=2.876544
```

Якщо порахувати результат на папері, то c=0, але програма видає результат c=2,876544. Це пов'язано з тим, що для зберігання значення мантиси змінної a типу float виділяється 23+1(неявний розряд) = 24 розряди, тому (як і в попередньому прикладі) заокруглюється навіть значення цілої частини числа. Змінна b має тип double і для зберігання її мантиси виділяється 52+1(неявний розряд) = 53 розряди, в які поміщається більша кількість розрядів значення мантиси і при заокругленні втрачається менша кількість значень розрядів.

Приклад 3.4.8. Помилка, пов'язана з різною точністю зберігання в пам'яті значень типу float і типу double (у мові C, якщо тип константи не вказано явно, то всі дійсні константи мають тип double):

```
...
  float a=0.1;
  if (a==0.1) printf(" true");
  else printf(" false");
...
```

```
false
```

Десятковому дробовому числу $0{,}1_{10}$ відповідає періодичний двійковий дріб $0{,}0(0011)_2$. Змінна a має значення 0,1 і має тип float, а константа 0,1 має

тип double. Тому періодичний двійковий дріб $0{,}0(0011)_2$ у пам’яті зберігається з різною точністю, отже, порівнюються різні значення.

Якщо задати тип константи float, то результатом порівняння буде true:

```
...
  float a=0.1;
  if (a==0.1f) printf(" true");
  else printf(" false");
...
```

```
true
```

Таким чином, при порівнянні дані треба приводити до одного типу.

Приклад 3.4.9. Десяткові скінченні дроби перетворюються в періодичні (нескінченні) двійкові дроби (програму було виконано двічі — масив a[9] було спочатку оголошено як float, а потім як double):

```
...
  int i;
  float a[9]={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};
  for (i=0; i<9; i++)
    printf(" float a=%3.1f -- %22.19f\n",a[i],a[i]);
...
```

```
float a=0.1 --  0.1000000014901161200
float a=0.2 --  0.2000000029802322400
float a=0.3 --  0.3000000119209289600
float a=0.4 --  0.4000000059604644800
float a=0.5 --  0.5000000000000000000
float a=0.6 --  0.6000000238418579100
float a=0.7 --  0.6999999880790710400
float a=0.8 --  0.8000000119209289600
float a=0.9 --  0.8999999761581420900
```

```
double a=0.1 --  0.1000000000000000100
double a=0.2 --  0.2000000000000000100
double a=0.3 --  0.2999999999999999900
double a=0.4 --  0.4000000000000000200
double a=0.5 --  0.5000000000000000000
double a=0.6 --  0.5999999999999999800
double a=0.7 --  0.6999999999999999600
double a=0.8 --  0.8000000000000000400
double a=0.9 --  0.9000000000000000200
```

У пам’яті комп’ютера значення залежно від типу зберігаються і з надлишком (більшість), і з недостачею (float — тільки 0,7 і 0,9; double — 0,3, 0,6, 0,7), причому надлишки і недостачі у кожного числа індивідуальні; для числа 0,5 зберігається точне значення. Розглянемо двійкове подання десяткових дробів від 0,1 до 0,9 для типу float, запишемо їхні мантиси, заокруглимо їх, щоб поміщалися в 23+1=24 розряди, визначимо величини експонент:

$0{,}1_{10} = 0{,}0(0011)_2$,

$1\underbrace{1001100110011001100110\mathbf{0}}_{\text{23 розряди}}\mathbf{110...}_2$, експонента -4_{10},

$1\underbrace{1001100110011001100110\mathbf{1}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}2_{10} = 0{,}(0011)_2$,

$1\underbrace{1001100110011001100110\mathbf{0}}_{\text{23 розряди}}\mathbf{110...}_2$, експонента -3_{10},

$1\underbrace{1001100110011001100110\mathbf{1}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}3_{10} = 0{,}0(1001)_2$,

$1\underbrace{0011001100110011001100\mathbf{1}}_{\text{23 розряди}}\mathbf{100...}_2$, експонента -2_{10},

$1\underbrace{001100110011001100110\mathbf{10}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}4_{10} = 0{,}(0110)_2$,

$1\underbrace{1001100110011001100110\mathbf{0}}_{\text{23 розряди}}\mathbf{110...}_2$, експонента -2_{10},

$1\underbrace{1001100110011001100110\mathbf{1}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}5_{10} = 0{,}1_2$ — точне подання

$1\underbrace{00000000000000000000000}_{\text{23 розряди}}{}_2$, експонента -1_{10}

$0{,}6_{10} = 0{,}(1001)_2$,

$1\underbrace{0011001100110011001101}_{\text{23 розряди}}\mathbf{1100...}_2$, експонента -1_{10},

$1\underbrace{001100110011001100110\mathbf{10}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}7_{10} = 0{,}1(0110)_2$,

$1\underbrace{0110011001100110011001}_{\text{23 розряди}}\mathbf{1001...}_2$, експонента -1_{10},

$1\underbrace{0110011001100110011001\mathbf{1}}_{\text{23 розряди}}{}_2$ — значення з недостачею;

$0{,}8_{10} = 0{,}(1100)_2$,

$1\underbrace{1001100110011001100110\mathbf{0}}_{\text{23 розряди}}\mathbf{110...}_2$, експонента -1_{10},

$1\underbrace{1001100110011001100110\mathbf{1}}_{\text{23 розряди}}{}_2$ — значення з надлишком;

$0{,}9_{10} = 0{,}1(1100)_2$,

$1\underbrace{1100110011001100110011\mathbf{0}}_{\text{23 розряди}}\mathbf{011...}_2$, експонента -1_{10},

$1\underbrace{1100110011001100110011\mathbf{0}}_{\text{23 розряди}}{}_2$ — значення з недостачею.

Приклад 3.4.10. Пояснити, чому при збереженні в пам’яті комп’ютера значень 0,3 і 1,3 типу float дробові частини чисел відрізняються, причому дробова частина першого числа зберігається з надлишком, а другого — з недостачею:

```
...
  float a, b;
  a=0.3f;
  b=1.3f;
  printf(" float a=%22.19f\n float b=%22.19f\n",a,b);
...
```

```
float a= 0.3000000119209289600
float b= 1.2999999523162842000
```

Розглянемо подання цих чисел у пам’яті комп’ютера:

$0{,}3_{10} = 0{,}0(1001)_2$,

$1\underbrace{0011001100110011001100\mathbf{1}}_{\text{23 розряди}}\mathbf{100...}_2$ з експонентою -2_{10},

$1\underbrace{001100110011001100110\mathbf{10}}_{\text{23 розряди}}{}_2$ — при заокругленні одержано значення з надлишком;

$1{,}3_{10} = 1{,}0(1001)_2$,

$1\underbrace{0100110011001100110011\mathbf{0}}_{\text{23 розряди}}\mathbf{011...}_2$ з експонентою 0_{10},

$1\underbrace{0100110011001100110011\mathbf{0}}_{\text{23 розряди}}{}_2$ — при заокругленні одержано значення з недостачею.

Таким чином, при збереженні в пам’яті комп’ютера значень типу float 0,3 і 1,3 двійкові подання їхніх дробових частин, які є двійковими періодичними дробами, відрізняються, оскільки ці числа мають різні мантиси, але для зберігання цих мантис виділяються однакові ділянки пам’яті і, крім того, мантиси заокруглюються.

Приклад 3.4.11. Якщо до будь-якого цілого чи дійсного числа додати дробове число, не кратне степеневі двійки, потім до одержаного результату знову додати це число і т.д., то в результуючій послідовності чисел похибка матиме хвилеподібний характер, тобто може спочатку збільшуватися, потім зменшуватися, потім знову збільшуватися і т.д., але без певної сталої амплітуди, бути як додатною, так і від’ємною. Одержимо, наприклад, послідовність чисел типу float з проміжку від 1 до 9,4 з кроком 0,6:

```
...
  float a;
  for (a=1.0f; a<=9.5f; a+=0.6f)
    printf(" %5.1f -- %12.10f\n", a, a);
...
```

```
Значення: точне -- float
  1.0 -- 1.0000000000   1.6 -- 1.6000000238   2.2 -- 2.2000000477
  2.8 -- 2.8000001907   3.4 -- 3.4000000954   4.0 -- 4.0000000000
  4.6 -- 4.5999999046   5.2 -- 5.1999998093   5.8 -- 5.7999997139
  6.4 -- 6.3999996185   7.0 -- 6.9999995232   7.6 -- 7.5999994278
  8.2 -- 8.1999998093   8.8 -- 8.8000001907   9.4 -- 9.4000005722
```

Як і в попередніх прикладах, наявність похибки пояснюється тим, що дробове число подається періодичним двійковим дробом і при зберіганні в комп’ютері заокруглюється. Те, що похибка постійно не накопичується (не зростає) при обчисленні нових значень, а має хвилеподібний характер пояснюється тим, що одержувані значення заокруглюються, причому, як з надлишком, так і з недостачею.

Приклад 3.4.12. Застосування операції множення, а також додавання (див. приклад 3.4.2) й віднімання до цілих чисел може призвести до втрати результату за рахунок арифметичного переповнення:

Неправильний результат

```
...
  int a=54321;
  double b;
  b=100*a*a;
  printf(" %f", b);
...          -1275639324.000000
```

Правильний результат

```
...
  int a=54321;
  double b;
  b=100*(double)a*a;
  //або b=100.*a*a;
  printf(" %f", b);
...          295077104100.000000
```

У випадку, коли одержано неправильний результат (–1 275 639 324), спочатку обчислилося ціле значення виразу 100*a*a (всі операнди мають тип int), його старші розряди вийшли за розрядну сітку (обчислене значення виразу 295 077 104 100 більше від максимально допустимого значення для типу int — 2 147 483 647) і вміст відведеної ділянки пам'яті інтерпретувався як від'ємне число, а після цього результат було присвоєно змінній b типу double. Неправильний результат (–134419625500) також дають вирази a*a*100. і 100.*(a*a), хоч у них константа 100. має тип double; вираз 1.+100*a*a дає результат –1 275 639 323, який теж є неправильним. Слід зауважити, що послідовність обчислення виразу може залежати і від компілятора.

У випадку правильного результату весь вираз у правій частині оператора присвоювання перетворився до типу double (оскільки у виразі другий операнд, а в закоментованому виразі перший операнд мають тип double) і одержаний результат було присвоєно змінній типу double.

Таким чином, програмуючи арифметичні вирази, щоб уникнути помилок при обчисленнях, обов'язково треба всі проміжні результати привести до одного типу.

Приклад 3.4.13. При роботі з плаваючою арифметикою зміна порядку доданків може змінити результат. Нехай у віртуальному комп'ютері, який працює з десятковими дійсними числами, для зберігання мантиси відведено чотири розряди, заокруглення здійснюється до ближчого, а заокруглення чисел з дробовою частиною 0,5 — до парного. Нехай дано числа

$1{,}000 = 1{,}000 \cdot 10^0$, $0{,}0003 = 3{,}000 \cdot 10^{-4}$, $0{,}0004 = 4{,}000 \cdot 10^{-4}$, $0{,}0005 = 5{,}000 \cdot 10^{-4}$.

Виконаємо їхнє додавання, змінюючи порядок доданків:

① $0{,}0003 + 0{,}0004 + 0{,}0005 + 1{,}000$

$$+\begin{array}{l}0{,}0003\\ \underline{0{,}0004}\\ 0{,}0007\end{array} \quad +\begin{array}{l}0{,}0007\\ \underline{0{,}0005}\\ 0{,}0012\end{array} \quad +\begin{array}{l}0{,}0012\\ \underline{1{,}000}\\ \underbrace{1{,}001}_{\text{4 розряди}}\mathbf{2} \approx 1{,}001\end{array}$$

результат 1,001.

② $1{,}000 + 0{,}0003 + 0{,}0004 + 0{,}0005$

$$+\begin{array}{l}1{,}000\\ \underline{0{,}0003}\\ \underbrace{1{,}000}_{\text{4 розряди}}\mathbf{3} \approx 1{,}000\end{array} \quad +\begin{array}{l}1{,}000\\ \underline{0{,}0004}\\ \underbrace{1{,}000}_{\text{4 розряди}}\mathbf{4} \approx 1{,}000\end{array} \quad +\begin{array}{l}1{,}000\\ \underline{0{,}0005}\\ \underbrace{1{,}000}_{\text{4 розряди}}\mathbf{5} \approx 1{,}000\end{array}$$

результат 1,000.

③ $0{,}0003 + 0{,}0004 + 1{,}000 + 0{,}0005$

$$+\begin{array}{l}0{,}0003\\ \underline{0{,}0004}\\ 0{,}0007\end{array} \quad +\begin{array}{l}0{,}0007\\ \underline{1{,}000}\\ \underbrace{1{,}000}_{\text{4 розряди}}\mathbf{7} \approx 1{,}001\end{array} \quad +\begin{array}{l}1{,}001\\ \underline{0{,}0005}\\ \underbrace{1{,}001}_{\text{4 розряди}}\mathbf{5} \approx 1{,}002\end{array}$$

результат 1,002.

Залежно від порядку доданків одержано різні результати: 1,001, 1,000, 1,002.

Останній приклад показує, що при знаходженні сум чисел з плаваючою точкою, діапазон значень яких досить широкий, треба ці числа групувати — окремо малі, окремо середні й окремо великі. Крім того, якщо числа типу float відрізняються більше, ніж у 2^{23} разів, а числа типу double більше, ніж в 2^{52} разів, то операції додавання й віднімання між ними не мають смислу.

Із наведених прикладів видно, що при роботі в комп'ютері з дійсними числами помилка заокруглення може бути пов'язана як з помилкою, яка виникає при переведенні скінченних десяткових дробів у нескінченні періодичні двійкові дроби і їхньому заокругленні, так і з помилкою заокруглення при виході молодших значущих розрядів за розрядну сітку. У деяких випадках помилка обчислень може бути настільки великою, що кінцевий результат може втратити будь-який зміст.

3.5. Алгоритми виконання арифметичних операцій

При розв'язуванні задач обчислювального характеру арифметичні операції виконуються досить багато разів. Розуміння алгоритмів їхнього виконання може сприяти створенню оптимальніших програмних кодів.

Додавання чисел. Сума будь-яких двох десяткових однозначних чисел буде однозначним чи двозначним числом, яке не перевищує числа 18_{10} (3+2=5; 9+9=18), тобто в розряд десятків перенесеться тільки 0 чи 1. Якщо додавати два однозначні двійкові числа, то їхня сума не буде перевищувати 10_2 ($1_2+1_2=10_2$), і теж буде однозначним чи двозначним числом. Аналогічно сума будь-яких трьох однозначних чисел буде однозначним чи двозначним числом і не буде перевищувати числа 27_{10} (9+9+9=27). Якщо додавати три однозначні двійкові числа, то їхня сума не буде перевищувати 11_2 ($1_2+1_2+1_2=11_2$) — теж буде однозначним чи двозначним числом.

Якщо в стовпчик додавати два числа і в розряді одиниць результатом буде двозначне число ($5_{10}+7_{10}=12_{10}$ або $1_2+1_2=10_2$), то тоді в розряді десятків вже треба додавати три числа, які теж можуть дати двозначне число і буде перенос в розряд сотень і т. д.

Із сказаного вище випливає, що якщо двійкові числа a і b мають у своєму записові по n розрядів (чи менше), то у двійковому записові суми цих чисел $a+b$ буде не більше ніж $n+1$ розряд. На обчислення кожного розряду суми використовується обмежений час і загальний час буде лінійним $c_0 + c_1 n$, тобто час роботи можна оцінити як $O(n)$. Слід зауважити, що суттєво швидшого алгоритму, ніж додавання в стовпчик, не існує, оскільки навіть для зчитування вхідних бітів і запису відповіді потрібно $O(n)$ операцій. Таким чином, алгоритм додавання оптимальний з точністю до мультиплікативної константи.

Від реалізації додавання цілих чисел у комп'ютері залежить його продуктивність, оскільки цілочисельне додавання використовується при генерації адрес для доступу до пам'яті, воно лежить в основі всіх операції з плаваючою точкою.

Сучасні комп'ютери за один крок однією командою додають два числа, які поміщаються в машинне слово (32 чи 64 розряди). Проте всередині про-

цесора ця команда реалізується за допомогою багатьох операцій з окремими бітами і кількість таких бітових операцій суттєва, оскільки від неї залежить кількість функціональних елементів при реалізації алгоритму в металі.

Операцію додавання двох чисел $a+b$ можна реалізувати через побітове додавання за модулем 2, побітове логічне «і» і побітовий зсув.

Приклад 3.5.1. Алгоритм додавання цілих чисел $a+b$ (з будь-якими знаками) з використанням побітових операцій, записаний мовою С, такий:

```
...
  int a, b,    // доданки з будь-якими знаками
      r;   // перенесення значень у старші розряди
  scanf("%d%d", &a, &b)
  while (b){    // результат міститься в змінній a
    r=a&b;   // побітове логічне "і"
    a=a^b;   // побітове виключне "або"
    b=r<<1;   // зсув на 1 біт вліво
  }
  printf("Сума %d\n", a);
...
```

Знайдемо суму, наприклад, чисел 12 (1100_2) і 7 (0111_2), використовуючи поданий алгоритм.

І прохід циклу:

$$r = \begin{array}{r} \& \begin{array}{r}1100\\ \underline{0111}\end{array} \\ \hline 0100 \end{array} \qquad a = \begin{array}{r} \oplus \begin{array}{r}1100\\ \underline{0111}\end{array} \\ \hline 1011 \end{array} \qquad b = 0100 << 1 = 1000\,;$$

ІІ прохід циклу:

$$r = \begin{array}{r} \& \begin{array}{r}1011\\ \underline{1000}\end{array} \\ \hline 1000 \end{array} \qquad a = \begin{array}{r} \oplus \begin{array}{r}1011\\ \underline{1000}\end{array} \\ \hline 0011 \end{array} \qquad b = 1000 << 1 = 10000\,;$$

ІІІ прохід циклу:

$$r = \begin{array}{r} \& \begin{array}{r}00011\\ \underline{10000}\end{array} \\ \hline 00000 \end{array} \qquad a = \begin{array}{r} \oplus \begin{array}{r}00011\\ \underline{10000}\end{array} \\ \hline 10011 \end{array} \qquad b = 00000 << 1 = 00000\,.$$

Оскільки b=0, то алгоритм закінчує роботу і результатом обчислень є значення $a=10011_2=19_{10}$, рівне сумі 12+7=19.

З'ясуємо, як працює цей алгоритм. По суті він реалізує додавання в стовпчик. За один прохід циклу виконуються такі дії: у змінну r за-

пам'ятовуються перенесення значень у старші розряди, які виникають при додаванні двох 1, розміщених у відповідних розрядах чисел a і b; у змінну а поміщається (майже) сума двох чисел a і b, але ще без перенесень; при виконанні зсуву бітів числа r на один біт вліво виконується зміщення бітів перенесень в бік старшого розряду і результат запам'ятовується в змінну b. Цикл виконується доти, поки є перенесення значень у старші розряди, тобто $b \neq 0$.

Якщо при розв'язуванні реальних задач треба працювати з довгими числами, то їх розбивають на частини, які зберігають у масивах. При додаванні таких чисел перенесення значень здійснюються до сусідніх елементів масиву.

Множення чисел. При множенні двох десяткових чисел у стовпчик треба помножити перше число на кожну цифру другого числа з відповідними зсувами і потім додати результати. Аналогічно виконується множення двійкових чисел, але тут цифрами є тільки 0 і 1. Тому при множенні першого числа на 0 буде одержано нуль, а при множенні на 1 число залишатиметься без змін — воно тільки зсувається вліво.

Якщо множити n-розрядні числа в стовпчик, то буде не більш ніж n проміжних чисел-результатів довжиною не більше від $2n$ (враховуючи нулі справа після зсувів). При цьому буде послідовно виконано $(n-1)$ додавання цих чисел (першого з другим, результату з третім, результату з четвертим і т. д.). Таким чином, оскільки операція додавання має складність $O(n)$, то множення має складність $O(n^2)$. При множенні двох двійкових чисел з розрядністю n_1 і n_2 їхній добуток буде не більш ніж $(n_1 + n_2)$-розрядним числом.

У комп'ютері, який працює з двійковими поданнями чисел, операцію множення в стовпчик можна реалізувати за допомогою операцій зсуву і додавання.

Приклад 3.5.2. Нехай треба знайти добуток $a \cdot b$ двох цілих беззнакових чисел, жодне з яких не є степенем числа 2 (якщо одне з чисел дорівнює 2^k, то досить виконати побітовий зсув іншого числа на k розрядів вліво). Алгоритм множення такий:

K1. Ввести множники a і b.

K2. d=0.

К3. Доки $b \neq 0$,

якщо (b&1)=1, то d=d+a;

a=(a<<1), b=(b>>1).

К4. Вивести добуток d.

Мовою C алгоритм запишеться так:

```
...
  scanf("%d%d", &a, &b);
  d=0;
  while (b) {
    if (b&1)
      d+=a;
    a<<=1;
    b>>=1;
  }
  printf("Добуток %d", d);
...
```

Знайдемо, наприклад, добуток чисел $a=12_{10}=1100_2$ і $b=11_{10}=1011_2$, використовуючи поданий алгоритм.

Початкові значення: *a*=1100; *b*=1011; *d*=0.

І прохід циклу:

Оскільки молодший розряд числа *b* дорівнює 1, то

$$d = \begin{array}{r} 0000 \\ +\,1100 \\ \hline 1100 \end{array}$$

$a = 1100 << 1 = 11000$;

$b = 1011 >> 1 = 0101$;

ІІ прохід циклу:

Оскільки молодший розряд числа *b* дорівнює 1, то

$$d = \begin{array}{r} 1100 \\ +\,11000 \\ \hline 100100 \end{array}$$

$a = 11000 << 1 = 110000$;

$b = 0101 >> 1 = 0010$;

ІІІ прохід циклу:

Оскільки молодший розряд числа *b* дорівнює 0, то значення *d* не змінюється;

$a = 110000 << 1 = 1100000$;

$b = 0010 >> 1 = 0001$;

IV прохід циклу:

Оскільки молодший розряд числа *b* дорівнює 1, то

$$
\begin{array}{r}
+\begin{array}{r} 100100 \\ 1100000 \\ \hline \end{array} \\
d = \ 10000100
\end{array}
$$

$a = \ 1100000 << 1 = 11000000$;

$b = \ 0001 >> 1 = 0000$;

Оскільки b=0, то алгоритм закінчує роботу і результатом обчислень є значення d=10000100_2=132_{10}, рівне добуткові 12*11=132.

У розглянутому алгоритмі перший множник зсувається вліво, а біти другого множника аналізуються, починаючи з молодшого розряду. Проте можна значення добутку знайти й іншим шляхом, — аналіз другого множника починати зі старшого розряду, а одержувані проміжні значення d (суми часткових добутків) зсувати вліво.

Для виконання множення чисел зі знаком з використанням поданого алгоритму треба: визначити знак добутку (для цього можна скористатися операцією виключного «або» і проаналізувати старший розряд); знайти абсолютні значення множників і перемножити їх; якщо знак добутку «мінус», то одержати додатковий код результату.

Ще розглянемо *алгоритм множення*, яким користувався *Аль-Хорезмі*. Відповідно до цього алгоритму треба записати два числа поряд, перше число поділити на 2, відкидаючи, якщо є, дробову частину, а друге число помножити на 2. Таку послідовність дій виконувати, доки перше число не набуде значення 1. Після цього треба викреслити всі рядки, в яких перше число парне. Числа, які залишаться у другому стовпчику додати і буде одержано результат множення двох чисел.

Приклад 3.5.3. Використовуючи алгоритм Аль-Хорезмі, помножимо 12 на 7. Подамо виконання алгоритму для десяткових і для двійкових чисел; також розглянемо множення двійкових чисел у стовпчик (для наочності — другого на перше):

~~12~~	~~7~~
~~6~~	~~14~~
3	28
1	56
	84

~~1100~~	~~111~~
~~110~~	~~1110~~
11	11100
1	111000
	1010100

$$
\begin{array}{r}
\times\begin{array}{r} 111 \\ 1100 \\ \hline \end{array} \\
+\begin{array}{r} 000 \\ 000 \\ 111 \\ 111 \\ \hline \end{array} \\
1010100
\end{array}
$$

Як видно з прикладу, алгоритм Аль-Хорезмі неявно використовує двійкову систему числення. Складність цього алгоритму теж $O(n^2)$.

Проте існує рекурсивний алгоритм (*метод Карацуби*), який має меншу складність $O(n^{\log_2 3}) = O(n^{1,59})$ — він базується на методі «розділяй і володарюй». Нехай треба помножити два *n*-розрядних числа *x* і *y*. Розіб'ємо кожне число навпіл, одержавши верхнє й нижнє півслова. Тоді ці числа у випадку десяткового подання можна записати так: $x_{10} = 10^{n/2} a_{10} + b_{10}$, $y_{10} = 10^{n/2} c_{10} + d_{10}$ (аналогічно можна подати двійкові числа: $x_2 = 2^{n/2} a_2 + b_2$, $y_2 = 2^{n/2} c_2 + d_2$). Тоді, наприклад, для двійкових чисел, матимемо:

$$x \cdot y = (2^{n/2} a + b) \cdot (2^{n/2} c + d) = 2^n ac + 2^{n/2} (ad + bc) + bd .$$

При цьому обчислення добутку двох *n*-розрядних чисел зведено до обчислення чотирьох добутків $n/2$-розрядних чисел. Кожен добуток має складність $const \cdot (n/2)^2$, а чотири добутки мають складність $const \cdot n^2 = O(n^2)$, тобто це нічого не змінило. Але суму $(ad + bc)$ можна ще подати так (подання Гаусса[14]):

$$ad + bc = (a + b) \cdot (c + d) - ac - bd .$$

Тоді:

$$x \cdot y = 2^n ac + 2^{n/2} ((a + b) \cdot (c + d) - ac - bd) + bd ,$$

при цьому треба обчислити тільки три добутки: *ac*, *bd* і $(a + b) \cdot (c + d)$, що дає складність $O(n^{\log_2 3}) = O(n^{1,59})$.

Ділення цілих чисел. У комп'ютері виконання команд мікропроцесора, пов'язаних з діленням цілих чисел a/b, дає два результати — частку і остачу від ділення. При діленні двійкових чисел з розрядністю n_1 і n_2 їхня частка буде не менш ніж ($n_1 - n_2$)-розрядним числом. Існують алгоритми ділення зі зсувом діленого вліво і зі зсувом дільника вправо. Відповідно до цих алгоритмів ділене повинно мати подвійну довжину порівняно з дільником; якщо ж розрядності діленого і дільника однакові, то довжину діленого треба подвої-

[14] Йоганн Карл Фрідрих Гаусс (Johann Carl Friedrich Gauß, 1777-1855) — німецький математик.

ти. Крім того, якщо дільник дорівнює нулю, чи, коли старша половина діленого не менша від дільника, то виникає переповнення частки.

При діленні двійкових чисел зі знаком, як і при виконанні множення, перед застосуванням алгоритму ділення треба визначити знак результату (частки), знайти модулі діленого й дільника, а після одержання результату, якщо треба, то подати його в додатковому коді.

Розглянемо алгоритм зі зсувом дільника вправо.

Приклад 3.5.3. Нехай треба поділити беззнакові числа a/b і нехай ділене a для свого зберігання потребує не більше 16 розрядів, а дільник b — не більше 8 розрядів. На початку алгоритму дільник зсувається максимально вліво (на 8 розрядів), а при виконанні алгоритму на кожному проході циклу зсувається на 1 розряд вправо. Ділене на кожному проході циклу, якщо воно не менше від поточного значення дільника, зменшується на величину дільника і при цьому останній біт частки c набуває значення 1; якщо ж ділене менше від дільника, то останній біт частки набуває значення 0. Циклічний алгоритм закінчить роботу, коли дільник набуде меншого значення, ніж початкове. Після виконання алгоритму ділене буде містити остачу від ділення.

Мовою C алгоритм запишеться так:

```
...
  unsigned short int a,   // ділене, остача від ділення
                     b,   // дільник
                     c;   // частка
  char n, i;
  scanf("%u%u", &a, &b);
  c=0;   // значення частки
  n=8;   // кількість розрядів дільника
  i=0;   // кількість проходів циклу
  b=b<<n;   // зсув дільника максимально вліво
  do {
    c<<=1;   // зсув частки на 1 розряд вліво
    if (a>=b) {
```

```
        a=a-b;
        c=c|1;    // запис 1 в останній розряд частки
      }
      b>>=1;
      i++;
    } while (i<=n);
    printf("Частка %u, остача %u\n", c, a);
...
```

Використовуючи поданий алгоритм, знайдемо, наприклад, частку і остачу від ділення чисел $a=123_{10}=1111011_2$ і $b=6_{10}=110_2$.

Початкові значення: *a*=0000 0000 0111 1011; *b*=0000 0000 0000 0110; *c*=0; *n*=8; *i*=0.
Значення *b* зміщується на 8 розрядів вліво — *b*=0000 0110 0000 0000

I прохід циклу:

(*c*=0) <<1=0;
Оскільки (*a*=0000 0000 0111 1011) < (*b*=0000 0110 0000 0000), то
a і *c* не змінюються;
(*b* = 0000 0110 0000 0000) >> 1 = 0000 0011 0000 0000 ;
(*i*=1) <= (*n*=8);

II прохід циклу:

(*c*=0) <<1=0;
Оскільки (*a*=0000 0000 0111 1011) < (*b*= 0000 0011 0000 0000), то
a і *c* не змінюються;
(*b* = 0000 0011 0000 0000) >> 1 = 0000 0001 1000 0000 ;
(*i*=2) <= (*n*=8);

III прохід циклу:

(*c*=0) <<1=0;
Оскільки (*a*=0000 0000 0111 1011) < (*b*= 0000 0001 1000 0000), то
a і *c* не змінюються;
(*b* = 0000 0001 1000 0000) >> 1 = 0000 0000 1100 0000 ;
(*i*=3) <= (*n*=8);

IV прохід циклу:

(*c*=0) <<1=0;
Оскільки (*a*=0000 0000 0111 1011) < (*b*= 0000 0000 1100 0000), то
a і *c* не змінюються;
(*b* = 0000 0000 1100 0000) >> 1 = 0000 0000 0110 0000 ;
(*i*=4) <= (*n*=8);

V прохід циклу:

(*c*=0) <<1=0;
Оскільки (*a*=0000 0000 0111 1011) >=(*b*= 0000 0000 0110 0000), то

$$
\begin{array}{r}
-\begin{array}{r} 0000000001111011 \\ 0000000001100000 \end{array} \\
\hline
a = \quad 0000000000011011
\end{array}
$$

$(c = 0)\,|\,1 = 1$;
(*b* = 0000 0000 0110 0000)>> 1 = 0000 0000 0011 0000 ;
(*i*=5) <= (*n*=8);

VI прохід циклу:

$(c=1) << 1=10$;

Оскільки $(a=0000\,0000\,0001\,1011) < (b=0000\,0000\,0011\,0000)$, то

a і c не змінюються;

$(b = 0000\,0000\,0011\,0000) >> 1 = 0000\,0000\,0001\,1000$;

$(i=6) <= (n=8)$;

VII прохід циклу:

$(c=100) << 1=100$;

Оскільки $(a=0000\,0000\,0001\,1011) >= (b=0000\,0000\,0001\,1000)$, то

$$a = \begin{array}{r} -\,0000000000011011 \\ 0000000000011000 \\ \hline 0000000000000011 \end{array}$$

$(c = 100)\,|\,1 = 101$;

$(b = 0000\,0000\,00011000) >> 1 = 0000\,0000\,0000\,1100$;

$(i=7) <= (n=8)$;

VIII прохід циклу:

$(c=101) << 1=1010$;

Оскільки $(a=0000\,0000\,0000\,0011) < (b=0000\,0000\,0000\,1100)$, то

a і c не змінюються;

$(b = 0000\,0000\,0000\,1100) >> 1 = 0000\,0000\,0000\,0110$;

$(i=8) <= (n=8)$;

IX прохід циклу:

$(c=1010) << 1=10100$;

Оскільки $(a=0000\,0000\,0000\,0011) < (b=0000\,0000\,0000\,0110)$, то

a і c не змінюються;

$(b = 0000\,0000\,0000\,0110) >> 1 = 0000\,0000\,0000\,0011$;

$(i=9) > (n=8)$ — умова продовження роботи циклу не виконується (при цьому значення b стало меншим від початкового).

Виконання циклу закінчується, результатом обчислень є значення частки $c=10100_2=20_{10}$ і остача від ділення $a=11_2=3_{10}$.

Ділення дійсних чисел. Ділення дійсних чисел x/y у сучасних мікропроцесорах реалізується через множення x на $1/y$ (значення $1/y$ досить точно обчислюється на основі початкового значення, яке береться з таблиці мікропроцесора, і його уточнення за допомогою 2-3 ітерацій алгоритму Ньютона). Хоч алгоритм ділення чисел виконується трохи довше, ніж множення, але він теж має складність $O(n^2)$.

Для одержання досить точного значення величини $1/y$ можна застосувати лінійний *алгоритм з «магічним» числом*. Слід зауважити, що «магічне» число насправді не є магічним. Воно залежить від використовуваного типу даних і від вигляду функції (у даному випадку функції $1/y$). Його розраховують, використовуючи параметри формату подання даних відповідно до

стандарту IEEE 754, а також підбираючи константу для кусково-лінійного наближення функції. Так, якщо y має тип float, то для функції $1/y$ «магічним» є число $7ef311c3_{16}$=2129859011_{10}.

Відповідно до алгоритму з використанням «магічного» числа виконується така послідовність дій. Біти числа y в форматі з плаваючою точкою типу float (стандарт IEEE 754), які містяться в пам'яті комп'ютера, інтерпретуються як ціле число типу int. Одержане ціле число віднімається від «магічного», що дає ціле число, біти якого відповідають початковому наближенню результату. Після цього виконується зворотне перетворення — біти цілого наближення типу int перетворюються на біти числа з плаваючою точкою типу float. Для уточнення результату до одержаного числа застосовуються дві ітерації методу Ньютона (див. приклад 6.3.10; для випадку функції $1/y = y^{-1}$ значення $k = -1$ і рекурентна формула набуває вигляду $b_{n+1} = b_n(2 - yb_n)$, де b_n — наближене значення величини $1/y$).

Приклад 3.5.3. Нехай y має тип float і треба обчислити значення $1/y$. Мовою С алгоритм обчислення значення $1/y$ з використанням «магічного» числа можна записати так:

```
...
  float y,   // задане число
        b;   // наближене значення функції 1/y
  int i;   // ціле початкове наближення результату
  printf("Обчислення значення 1/y\n  Введіть y ");
  scanf("%f", &y);
  memcpy(&i, &y, sizeof(i));  // подання бітів float цілим числом
  i=0x7ef311c3-i;   // ціле початкове наближення функції
  memcpy(&b, &i, sizeof(b));  // подання бітів int дійсним числом
  b=b*(2.0f-y*b);   // перша ітерація методу Ньютона
  b=b*(2.0f-y*b);   // друга ітерація методу Ньютона
  printf("1/y= %f\n", b);
...
```

```
Обчислення значення 1/y
  Введіть y
5
1/y= 0.200000
2
1/y= 0.499997
3
1/y= 0.333331
8
1/y= 0.124999
11
1/y= 0.090909
кінець
```

У розглянутому записі алгоритму мовою С для подання бітів числа з плаваючою точкою типу float цілим числом типу int і навпаки використано функцію memcpy з бібліотеки string.h, яка виконує копіювання однієї ділянки пам’яті в іншу. Звернення до функції `memcpy(&i, &y, sizeof(i))` і `memcpy(&b, &i, sizeof(b))` можна відповідно замінити операторами `i=*(int*)&y;` і `b=*(float*)&i;`.

Максимальна відносна похибка результату, одержаного за поданим алгоритмом, становить приблизно $6{,}51\times10^{-4}$ %, що відповідає 17,2 коректним бітам результату [29] для всього діапазону чисел з плаваючою точкою, поданих у форматі одинарної точності стандарту IEEE-754.

У [29] також подано вдосконалений алгоритм, який має максимальну відносну похибку $1{,}18\times10^{-5}$ %, що відповідає 23,1 коректним бітам результату. В алгоритмі використано інше, отимальніше, «магічне» число $7eb504f3_{16}=$ $=2125792499_{10}$ і модифіковано формулу першої ітерації методу Ньютона:

```
...
  float y, b;
  int i;
  printf("Обчислення значення 1/y\n  Введіть y \n");
  scanf("%f", &y);
  i=*(int*)&y;    // подання бітів float цілим числом
  i=0x7eb504f3-i;    // ціле початкове наближення функції
  b=*(float*)&i;    // подання бітів int дійсним числом
  b=1.96875f*b*(1.4255685f-y*b);    // перша модифікована ітерація
```

```
  b=b*(2.0f-y*b);    // друга ітерація методу Ньютона
  printf("1/y= %f\n", b);
...
```

```
Обчислення значення 1/y
  Введіть y
5
1/y= 0.200000
2
1/y= 0.500000
3
1/y= 0.333333
8
1/y= 0.125000
11
1/y= 0.090909
```

Використовуючи подані алгоритми знаходження значення $1/y$, можна знайти значення x/y, помноживши одержаний результат на x.

Алгоритми знаходження часток x/y з використанням «магічних» чисел ще називають алгоритмами швидкого ділення.

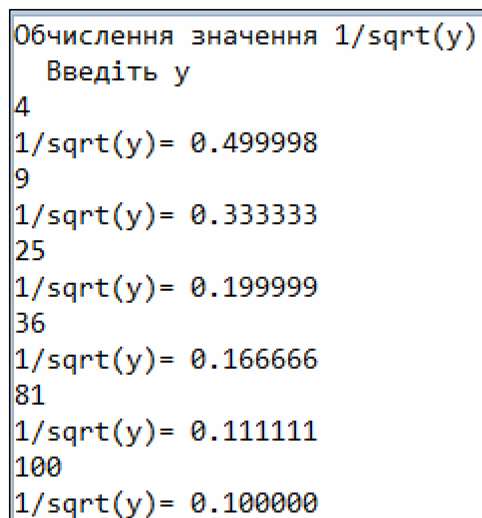
Швидке обчислення оберненого квадратного кореня. Хоч обчислення оберненого квадратного кореня не є звичайною арифметичною операцією, проте це одна з основних операцій у 3D-програмуванні. При роботі з 3D-графікою для затемнення або просвітлення окремих ділянок зображення, обчислення кутів падіння і відбивання світла використовують нормалі до поверхонь — тривимірні вектори одиничної довжини. При нормалізації векторів (x, y, z) треба кожну координату поділити на довжину вектора $\sqrt{x^2 + y^2 + z^2}$. Таких векторів потрібно знайти дуже багато. Тому швидкість обчислення оберненого квадратного кореня впливає на швидкість обробки 3D-зображень і є досить актуальною задачею.

Для швидкого обчислення оберненого квадратного кореня можна скористатися алгоритмом з «магічним» числом і кількома ітераціями методу Ньютона (див. приклад 6.3.10; для випадку функції $1/\sqrt{y} = 1/\sqrt[2]{y}$ значення $k = -2$ і рекурентна формула набуває вигляду $b_{n+1} = b_n(1{,}5 - 0{,}5 y b_n^2)$, де b_n — наближене значення величини $1/\sqrt{y}$).

Приклад 3.5.4. Нехай y має тип float і треба обчислити значення $1/\sqrt{y}$.

Мовою С алгоритм обчислення значення $1/\sqrt{y}$ з використанням «магічного» числа $5f3759df_{16}=1597463007_{10}$ можна записати так:

```
...
  float y,    // задане число; половина числа
        b;    // наближене значення функції 1/sqrt(y)
  int i;      // ціле початкове наближення результату
  printf("Обчислення значення 1/sqrt(y)\n  Введіть y ");
  scanf("%f", &y);
  i=*(int*)&y;    // подання бітів float цілим числом
  i=0x5f3759df-(i>>1);   // ціле початкове наближення функції
  b=*(float*)&i;    // подання бітів int дійсним числом
  y=0.5f*y;
  b=b*(1.5f-y*b*b);    // перша ітерація методу Ньютона
  b=b*(1.5f-y*b*b); // друга ітерація методу Ньютона
  printf("1/sqrt(y)= %f\n", b);
...
```



Цей алгоритм приблизно у 3-4 рази швидше обчислює значення оберненого квадратного кореня, ніж алгоритм з використанням бібліотечних функцій. Максимальна відносна похибка алгоритму становить $4{,}65\times10^{-4}$ %, а точність обчислень — 17,7 коректних [30] бітів результату після двох ітерацій методу Ньютона для всього діапазону чисел з плаваючою точкою (без врахування помилок заокруглення), поданих у форматі одинарної точності стандарту IEEE-754.

У [30] також подано вдосконалений алгоритм, який має максимальну відносну похибку $6{,}52\times10^{-5}$ % і дає 20,5 коректних бітів результату. Відносна похибка порівняно з попереднім алгоритмом зменшилася більш ніж у 7 разів (700 %) для всього діапазону значень чисел типу float.

У вдосконаленому алгоритмі використано інше, оптимальне, «магічне» число $5F375A86_{16}=1597463174_{10}$ і модифіковано формули обох ітерацій методу Ньютона (застосування тільки нового оптимального «магічного» числа без модифікації ітераційних формул методу Ньютона зменшує максимальну відносну похибку всього на 0,12 % порівняно з 700 % при модифікації ітераційних формул) . Мовою С алгоритм записується так:

```
...
  float y,    // задане число; половина числа
        b;    // наближене значення функції 1/sqrt(y)
  int i;      // ціле початкове наближення результату
  printf("Обчислення значення 1/sqrt(y)\n  Введіть y ");
  scanf("%f", &y);
  i=*(int*)&y;    // подання бітів float цілим числом
  i=0x5f375a86-(i>>1);   // ціле початкове наближення функції
  b=*(float*)&i;    // подання бітів int дійсним числом
  y=0.5f*y;   // половина заданого числа
  b=b*(1.50089090f-y*b*b);   // модифікація першої ітерації
  b=b*(1.50000060f-y*b*b);   // модифікація другої ітерації
  printf("1/sqrt(y)= %f\n", b);
...
```

```
Обчислення значення 1/sqrt(y)
  Введіть y
4
1/sqrt(y)= 0.500000
9
1/sqrt(y)= 0.333333
25
1/sqrt(y)= 0.200000
36
1/sqrt(y)= 0.166667
81
1/sqrt(y)= 0.111111
100
1/sqrt(y)= 0.100000
```

У розглянутих прикладах 3.5.3 і 3.5.4 обчислення виконувалися для дійсних чисел одинарної точності (32-розрядних). Проте можна розрахувати «магічні» числа і для 64- і 128-розрядних чисел.

Крім оберненого числа (степінь –1; константа $7ef311c3_{16}$=2129859011_{10}), а також оберненого квадратного кореня (степінь –1/2; константа $5f3759df_{16}$=1597463007_{10}), з використанням «магічних» чисел і кількох ітерацій методу Ньютона можна знайти досить точні значення інших функцій y^p, де $p \in [-1; 1]$. Наприклад, для звичайного квадратного кореня ($p = 1/2$) з числа типу float «магічне» число дорівнює $1fbd1df5_{16}$=532487669_{10}, відповідний оператор для знаходження початкового наближення має вигляд `i=0x1fbd1df5+(i>>1);`; для кубічного кореня ($p = 1/3$) — $2f517d3c_{16}$=709983548_{10}, `i=(int)(0x2a517d3c+(0.333f*i));`.

Будь-яке число в степені 0 ($p = 0$) дорівнює 1, для нього «магічним» є число $3f7a3bea_{16}$=1064975338_{10}, відповідний оператор має вигляд `i=0x3f7a3bea;`. Число $3f7a3bea_{16}$ є не випадковим значенням — це одиниця (точніше — майже одиниця) у форматі чисел с плаваючою точкою.

У загальному випадку для обчислення початкового наближення при $p \in [-1;1]$ можна використати оператор `i=(1-p)*0x3f7a3bea+(p*i);` або, виконавши перетворення, одержимо `i=0x3f7a3bea+p*(i-0x3f7a3bea);` (при цьому економиться одна дія множення).

Слід зауважити, що пóшукові найкращих (у певному сенсі) «магічних» чисел свої роботи присвятили ряд дослідників. У зв’язку з цим в різних роботах можна знайти різні «магічні» числа для обчислення значень тієї самої функції.

Запитання та завдання для самоконтролю

1. Що таке система числення? Наведіть приклади непозиційних систем числення. Наведіть приклади позиційних систем числення. Які позиційні системи числення використовують найчастіше? Як можна подати число в позиційній системі числення? Наведіть приклади змішаних систем числення.

2. Як перейти від двійкового подання числа до десяткового і навпаки? Яка особливість переведення десяткових дробів у двійкові? Як перейти від подання числа в одній системі числення до подання в іншій? Як перейти від двійкового подання числа до вісімкового чи шістнадцяткового і навпаки. Як виконати додавання, віднімання, множення й ділення у двійковій системі числення?

3. До якого типу систем числення належать коди Грея? Яка особливість кодів Грея? Де застосовують коди Грея?

4. Вкажіть формати IEEE 754 подання цілих і дійсних чисел у комп'ютері. Що таке додатковий код цілого від'ємного числа і як його знайти? Як кодуються в комп'ютері числа з плаваючою точкою? Що таке мантиса і що таке експонента числа? Що таке залишок мантиси і що таке зміщена експонента в поданні дійсного числа в комп'ютері? Яке подання числа називають нормалізованим, а яке денормалізованим? Чому при зберіганні в комп'ютері дійсні числа дискретизуються? Які правила заокруглення передбачено стандартом IEEE 754, яке правило використовується за замовчуванням? Вкажіть джерела помилок при роботі з дійсними числами. Чи може змінитися результат, якщо змінити порядок кількох дійсних доданків?

5. Скільки двійкових розрядів має сума двох двійкових n-розрядних чисел? Як у суматорах реалізується операція додавання цілих чисел? Скільки двійкових розрядів має добуток двох двійкових чисел з розрядністю n_1 і n_2? Скільки двійкових розрядів має частка двох двійкових чисел з розрядністю n_1 і n_2? Які обчислювальні складності операцій додавання, множення, ділення; яка складність алгоритму множення Аль-Хорезмі; яка складність алгоритму множення Карацуби? Що є результатом ділення цілих чисел? Які особливості множення й ділення цілих чисел зі знаком? Як виконується в комп'ютері ділення дійсних чисел? Що таке «магічне» число, від чого залежить його значення? Для чого використовують «магічні» числа?

Практичне завдання № 3 «Позиційні системи числення»

Мета виконання завдання: навчитися подавати числа в різних системах числення, дослідити і зрозуміти наслідки арифметичного переповнення при роботі з цілими числами.

Завдання 1. Застосовуючи розглянуті алгоритми переходу між різними поданнями чисел, а також алгоритми виконання операцій над числами у двійковій системі числення (див. п. 3.2), виконати подані нижче завдання:

1) яке найбільше число можна показати на пальцях обох рук, якщо скористатися двійковою системою числення? (1023_{10});

2) здійснити перехід від двійкового подання числа до десяткового: $10110{,}1101_2$ ($22{,}8125_{10}$);

3) подати в двійковій системі числення десяткові числа: 587_{10}; $0{,}3_{10}$; $426{,}35_{10}$; $323{,}375_{10}$; $87{,}2_{10}$ (1001001011_2; $0{,}0(1001)_2$; $110101010{,}01(0110)_2$; $101000011{,}011_2$; $1010111{,}(0011)_2$);

4) подати двійкові числа $1001111010{,}10111_2$; $0{,}0(1001)_2$ у вісімковій системі числення ($1172{,}56_8$; $0{,}2(3146)_8$);

5) подати двійкові числа $1001111010{,}10111_2$; $0{,}0(1001)_2$ у шістнадцятковій системі числення ($27A{,}B8_{16}$; $0{,}4(6)_{16}$);

6) здійснити прямий перехід від десяткового подання числа $325{,}53125_{10}$ до вісімкового. ($505{,}42_8$);

7) здійснити прямий перехід від десяткового подання числа $984{,}90625_{10}$ до шістнадцяткового. ($3D8{,}E8_{16}$);

8) подати числа $101110{,}0111_2$, $56{,}34_8$ і $2E{,}7_{16}$ у десятковій системі числення ($46{,}4375_{10}$);

9) знайти суму чисел у двійковій системі числення 100011111_2 і 1011110_2 (101111101_2).

Завдання 2. Враховуючи особливості подання цілих чисел у комп'ютері (див. п. 3.4), виконати завдання:

1) змінні типу char мають значення 9, 15 і 24. Використовуючи додатковий код, знайти подання у пам’яті комп’ютера результатів виконання операцій $9-24$, $24-9$, $-9-15$ (*1*1110001, **1***0*0001111, **1***1*1101000 — що треба зробити з 1, яка вийшла за розрядну сітку?)

2) виконати програму і пояснити, чому одержано різні результати:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char n=123, m=9, k;
  k=n+m;
  printf("k=%d m+n=%d ", k, n+m);
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
k=-124
n+m=132
```

3) у пам’яті комп’ютера містяться 8-бітові послідовності 00000000, 10000000 і 11111111. Які десяткові числа їм відповідають, якщо відповідна змінна має тип char; тип unsigned char? (0_{10}, -128_{10} і -1_{10}; 0_{10}, 128_{10} і 255_{10}).

Практичне завдання № 4
«Робота з дійсними числами на комп’ютері»

Мета виконання завдання: ознайомитися зі стандартом IEEE подання дійсних чисел у комп’ютері, дослідити причини похибок при роботі з дійсними числами.

Завдання 1. Виконати програму і пояснити, чому одержано такі результати.

```
...
  float a, b, c;
  a=987654350;
  b=987654310;
  c=a-b;
  printf(" a=%f  b=%f  c=%f\n", a, b, c);
...
```

```
a=987654336.000000  b=987654336.000000  c=0.000000
```

Завдання 2. Обґрунтувати одержання поданого неправильного результату, виконавши розрахунки на папері.

```
...
  float a;
  double b, c;
  a=987654321;
  b=987654321;
  c=a-b;
  printf(" c=%f\n", c);
...
```

```
c=15.000000
```

Завдання 3. Пояснити, чому в комп'ютері дробова частина 0,9 порізному подається при різних значеннях цілої частини.

```
...
  double a=0.9, b=2.9, c=9.9;
  printf(" a=%22.19f\n b=%22.19f\n c=%22.19f\n", a, b, c);
...
```

```
a= 0.9000000000000000200
b= 2.8999999999999999000
c= 9.9000000000000004000
```

Завдання 4. З'ясувати, за рахунок чого при одержанні значень від 3 до 4,4 з кроком 0,2 похибка постійно не збільшується, а має хвилеподібний характер.

```
...
 float a;
  for (a=3.0f;a<=5.0f; a+=0.2f)
    printf(" %22.19f\n", a);
...
```

```
3.0000000000000000000
3.2000000476837158000
3.4000000953674316000
3.6000001430511475000
3.8000001907348633000
4.0000000000000000000
4.1999998092651367000
4.3999996185302734000
```

Завдання 5. З'ясувати, за рахунок чого одержано неправильну відповідь.

```
...
  double a;
  int i=2000;
  a=i*i*i;   // правильна відповідь 8000000000
  printf(" a=%f", a);
...
```

```
a=-589934592.000000
```

4. Лінійні алгоритми

Поняття лінійного алгоритму. Програмування виразів. Структура лінійного алгоритму. Приклади лінійних алгоритмів. Складність виразу; оптимізація обчислень.

4.1. Поняття лінійного алгоритму

Алгоритм, у якому дії виконуються послідовно одна за одною, називають *лінійним*. Кожна дія в лінійному алгоритмі є обов'язковою для виконання і кожна дія виконується тільки один раз незалежно від початкових даних чи проміжних результатів. У лінійному алгоритмі дії описуються в тому порядку, в якому вони мають бути виконані.

Лінійний алгоритм є найпростішим. Він не містить розгалужень і циклів, тобто в ньому відсутні обчислення, пов'язані з перевіркою деякої умови і вибором тієї чи іншої дії залежно від значення цієї умови. Проте будь-який нелінійний алгоритм містить лінійні фрагменти.

Лінійні алгоритми містять тільки послідовності простих команд — обчислювальні інструкції (оператори присвоювання, виклики функцій) і інструкції введення-виведення. Ці алгоритми використовують, наприклад, для обчислення значень функцій з необмеженою областю визначення, для задач з наперед визначеною послідовністю їхнього розв'язування.

4.2. Програмування виразів

Вираз у широкому розумінні — це константа, або змінна, або звернення до функції, або вираз, який містить константи, змінні, звернення до функцій, з'єднані знаками операцій.

Константи, змінні, функції мають певний тип. Значення констант повинні бути визначені, а змінні оголошені до використання в програмі. Функції беруться з бібліотек, заголовочні файли яких вказуються директивою препроцесора (наприклад, `#include <math.h>`).

У мові C є 5 базових типів даних:

- char — символьні;
- int — цілі;
- float — дійсні;
- double — дійсні подвійної довжини;
- void — порожні, які не набувають ніякого значення.

Типи char і int є цілими типами і призначені для зберігання цілих чисел. Хоч тип char за назвою є символьним, будь-який символ у комп'ютері зв'язаний з цілим числом — кодом цього символа в таблиці ASCII. Сам символ потрібен, коли інформація виводиться на екран чи принтер, або, коли вводиться з клавіатури. Перетворення з коду в символ і навпаки виконуються автоматично. Символьний тип є знаковим і змінні набувають значень від –128 до 127 (але в деяких реалізаціях може бути за замовчуванням беззнаковим і набувати значень від 0 до 255). Розмір об'єкта int, як правило, збігається з розміром слова в конкретному середовищі програмування. У більшості випадків в 16-розрядному середовищі (DOS або Windows 4.1) int займає 16 бітів, а в 32-розрядному (Windows 95 і вище) — 32 біти. Тому при перенесенні програми в інше середовище це треба враховувати.

Змінні типу float і double є дійсними числами зі знаком, їх називають *числами з плаваючою точкою* (ще кажуть: рухомою комою). Конкретний формат числа с плаваючою точкою залежить від його реалізації в компіляторі.

Ключове слово void запозичене із стандарту ANSI для мови C++; змінну типу void створити не можливо; цей тип використовують при оголошенні функцій, які не повертають значень, або для створення універсального вказівника.

На основі цих типів будуються інші типи даних за допомогою використання модифікаторів (ще кажуть: специфікаторів, кваліфікаторів), які записують перед відповідним типом. Такими модифікаторами є:

- signed — знаковий (застосовується до char і int, хоч тип int за замовчуванням є знаковим);
- unsigned — беззнаковий (застосовується до char і int);
- short — короткий (застосовується до int);
- long — довгий (застосовується до int і double; до int (в C99) може застосовуватися двічі).

Модифікатори signed і unsigned можна комбінувати з модифікаторами short і long для типу int (таблиця 4.1).

Таблиця 4.1. Типи даних

Тип	Пам'ять (байтів) у Visual Studio	Діапазон значень у Visual Studio
char / signed char	1	–128 … 127 ($-2^7 \dots 2^7-1$)
unsigned char	1	0 … 255 (2^8-1)
short int / signed short int	2	–32768 … 32767 ($-2^{15} \dots 2^{15}-1$)
unsigned short int	2	0 … 65535 ($2^{16}-1$)
int / signed int / long int / signed long int	4 int за стандартом — мінімум 2	–2147483648 … 2147483647 ($-2^{31} \dots 2^{31}-1$)
unsigned int / unsigned long int	4	0 ... 4294967295 ($2^{32}-1$)
long long int	8	–9223372036854775808 … 9223372036854775807 ($-2^{63} \dots 2^{63}-1$); *стандарт C99*
unsigned long long int	8	0 ... 18446744073709551615 ($0 \dots 2^{64}-1$); *стандарт C99*
float	4	9.9e–45 … 3.4e+38; точність — 6-7 десяткових цифр; мантиса — 24 біти
double / long double	8 long double за стандартом — 10	4.94e–324 ... 1.7e+308; точність — 15-16 десяткових цифр; мантиса — 53 біти

Цілі числа зі знаком і без знака відрізняються інтерпретацією найстаршого біта числа. Якщо ціле зі знаком, то компілятор вважає, що цей біт містить знак числа: 0 — плюс, 1 — мінус. Якщо ціле без знака, що цей біт інтерпретується як двійкова цифра числа.

У мові програмування C немає спеціального ключового слова для позначення такого окремого типу даних, як рядки (у мові Pascal — string), хоч формат виведення рядка у функцій scanf і printf є — %s). Рядки в мові C подають масивами символів.

У стандарті C99 додано ще 3 типи даних: _Bool (логічний), _Complex (комплексний), _Imaginary (уявний), але, наприклад, у Visual Studio 2010 вони не реалізовані. Стандарт C11 нових типів не додавав.

У мові C є оператор typedef, який дає можливість призначати синоніми для будь-яких типів. Наприклад:

```
typedef unsigned char ascii; // призначення синоніма ascii для типу
                             // unsigned char
ascii c; // оголошення змінної c типу ascii
```

Усі *змінні* перед використанням повинні бути оголошені (перед головною функцією — глобальні змінні, у тілі функції чи блоку у функції — локальні змінні):

тип список_змінних;

Тут тип означає один із стандартних або оголошених програмістом типів, а список змінних складається з одного або більше ідентифікаторів, розділених комами:

```
int k, l, m;
unsigned int ui;
double length, suma;
```

Оголошення змінних може бути розташоване в трьох місцях коду програми: всередині функції (у декларативній частині і в блоці — локальні змінні), у списку параметрів функції при її описові (формальні параметри), поза всіма функціями (глобальні змінні). Стандарт C99 передбачає можливість оголошення змінних не тільки в декларативній частині функції, а й будь-де в її тілі. Проте для забезпечення читабельності коду програми оголошення повинні розміщуватися до виконуваних операторів.

Змінна при оголошенні може бути ініціалізована:

тип ім'я_змінної = константа;

Наприклад:

```
char ch='a';
int i=0, j, n=20; // змінні i й n — ініціалізуються, j — ні.
```

Неініціалізовані локальні змінні до першого присвоєння мають довільне значення (сміття з пам'яті).

Константи можуть бути цілими, дійсними, символьними.

Цілі константи визначаються як числа (зі знаком чи без) без дробової частини. Цілі константи в C можуть бути:

— десятковими (`135`, `-10`);

— вісімковими (записана цифрами 0-7 і перша цифра 0: `0207`, `-012`). *Увага*: оскільки цифра 0 вказує на вісімкове подання, то 0123 ($0123=123_8=83_{10}$) і 123 — це різні числа;

— шістнадцятковими — записана цифрами 0-9, буквами A-F або a-f і починається з 0X чи 0x: `0X87`, `0x87`, `-0xA`.

Цілі константи мають той мінімальний тип int чи long long int (4 чи 8 байтів), в діапазон значень якого їхні значення поміщаються (це також може залежати від архітектури комп'ютера).

Тип цілої константи може визначити програміст — можна явно вказати компілятору, який тип має ціла константа, використовуючи відповідні модифікатори-суфікси (регістр не має значення), приписувані після молодшої цифри константи:

H чи h — тип short (у Visual Studio не працює),
L чи l — тип long,
LL чи ll (стандарт C99) тип long long,
U чи u — unsigned.

Наприклад, `120h`, `3H`, `823L`, `042l`, `0x7D3Al`, `846U`, `054U`, `0x7A4Fu`, `235lu`, `0578ul`, `0x5ul`.

Дійсні константи — це десяткові дроби. Усі дійсні константи мають тип double. Вони можуть бути записані в двох форматах:

— з фіксовано точкою (<знак><ціла частина>.<дробова частина>): `-123.4567`, `561.9`, `.78` (відсутня ціла частина), `2.` (відсутня дробова частина);

— з порядком (з плаваючою точкою — <мантиса>E<порядок> або <мантиса>е<порядок>): `0.1345e3` (=134,5), `-1.34e-1` (=-0,134), `22E-2` (=0,22), `.45e2` (=45,0), `-45E3` (=-45000,0), `23.75e-3` ($=23{,}75\cdot10^{-3}$), `0.28e5` ($=0{,}28\cdot10^{5}$).

Усі дійсні константи, навіть дуже малі, мають тип double і вони займають 8 байтів пам'яті, якщо їхні значення виходять за діапазон даних типу double, то вони мають тип long double і займають 10 байтів пам'яті (у Visual Studio цей тип не відрізняється від типу double).

Можна явно вказати компілятору, який тип має дійсна константа, використовуючи відповідні модифікатори-суфікси (регістр не має значення): F чи f — тип float і їй виділяється 4 байти пам'яті; L чи l — тип long double і виділяється 10 байтів пам'яті незалежно від значення дійсної константи. Наприклад: `5.6F`, `2.7e-10f`, `3.85L`, `8.074e+32L`.

Дійсні числа треба подавати тільки десятковими цифрами (у випадку вісімкового чи шіснадцяткового подання результат буде непередбачуваним).

Символьні константи можна подати як символи (`'%'`, `'Є'`), десятковим ASCII-кодом (`37`, `242`), використати вісімкове (`'\45'`, `'\362'`) чи шіснадцяткове подання (`'\x25'`, `'\xF2'`).

Якщо при оголошенні змінної, значення якої ініціалізується, додати кваліфікатор const, то її значення в програмі змінювати не можна — вона стає іменованою константою вказаного типу:

const тип ім'я_змінної = константа;

Наприклад:
```
const int n=100;
const double PI=3.14159265358979, a=-10., g=9.80665, eps=1e-10;
const char c1='\45', c2='\xF2', c3='Ї';
```
Також константу можна визначити директивою препроцесора:
```
#define N 5.
```
Константи, визначені директивами препроцесора, як і константи, оголошені в функції, можна використовувати у виразах.

Кожен тип даних передбачає набір операцій, виконання яких допускається над даними цього типу. У мові C порівняно з іншими мовами є досить велика кількість операцій (більше 35). Операції за спаданням їхніх пріоритетів подано в таблиці 4.2.

У мові C допускається багато неявних перетворень типів. Наприклад, у виразах можна вільно змішувати змінні дійсного і цілого типів. При обчисленні виразів, у яких змішано дані різних типів, відбувається перетворення типів за спеціальними правилами, поданими в таблиці 4.3.

Із наведених у таблиці 4.3 правил перетворення видно, що якщо операнди мають різні типи, то відбувається приведення операнда молодшого типу до старшого типу; при цьому для даних цілих типів старшим вважається тип unsigned порівняно з типом signed. Проте, хоч стандарт мови C визначає неявне перетворення до типу unsigned, реалізація може залежати від компілятора.

Поєднання у виразах цілих чисел типів signed і unsigned може призвести до непередбачуваних результатів. Тому у виразах краще використовувати явне перетворення типів.

Щоб явно вказати перетворення типів, треба до окремих операндів застосовувати відповідну операцію:

(тип)операнд

Наприклад: 7/4 дає 1, а (double)7/4 дає 1,75; s=s+1/(double)i;

Якщо значення виразу присвоюється змінній, то результат обчислення перетворюється до типу цієї змінної. При цьому при перетвореннях цілих int (4 байти) в короткі цілі short int (2 байти) втрачаються 16 старших бітів, а при перетворенні коротких цілих зі зна-

ком signed short int (2 байти) у символи зі знаком signed char (1байт) втрачаються 8 бітів (таблиця 4.4). Наприклад, якщо виконується фрагмент програми з присвоєнням:

```
int a=283;
char c;
c=a;
```

то змінна c набуде значення 27 (оскільки $283_{10} = 1\ 0001\ 1011_2$ і старший розряд при присвоєнні втрачається, а залишиться число $0001\ 1011_2$, яке інтерпретується як 27).

Таблиця 4.2. Операції мови C (операції подано за спаданням пріоритетів; у кожній групі, відокремленій суцільною лінією, пріоритети однакові)

Група операцій	Знак операції	Опис
первинні	( )	підвищення пріоритету (дужки)
	[ ]	виділення елемента масиву
	.	виділення елемента запису
	->	виділення елемента запису
унарні	!	логічне заперечення
	~	побітове заперечення
	-	зміна знака
	++	збільшення на одиницю
	--	зменшення на одиницю
	&	взяття адреси
	*	звернення за адресою
	(тип)	перетворення типу
	sizeof( )	визначення розміру в байтах
мультиплікативні	*	множення
	/	ділення
	%	визначення остачі від ділення
адитивні	+	додавання
	-	віднімання
зрушення	<<	зсув вліво
	>>	зсув вправо
відношень (порівняння)	<	менше ніж
	<=	менше або дорівнює
	>	більше ніж
	>=	більше або дорівнює
	==	дорівнює
	!=	не дорівнює
побітові (порозрядні)	&	побітове логічне «і» (логічне множення)
	^	побітове виключне «або»
	|	побітове логічне «або»
логічні	&&	логічне «і»
	||	логічне «або»
умови	?:	умовна (тернарна) операція
присвоювання	=	присвоювання
	+=, - =, *=, /=, %=, <<=, >>=, &=, |=, ^=	складені операції присвоювання (a*=b означає a=a*b)
кома	,	операція кома

Таблиця 4.3. Перетворення типів операндів у виразах мови С (перетворення виконуються відповідно до таблиці при перегляді її згори вниз)

№	Початкові типи операндів	Типи операндів після перетворення	Пояснення
1.	long double α ∀ β	long double α **long double β**	якщо один операнд має тип long double, то і другий операнд перетвориться на long double
2.	double α ∀ β	double α **double β**	якщо один операнд має тип double, то і другий операнд перетвориться на double
3.	float α ∀ β	float α **float α**	якщо один операнд має тип float, то і другий операнд перетвориться на float
4.	long long int α ∀ ціле зі знаком β	long long int α **long long int β**	якщо один операнд має тип long long int і другий операнд є беззнаковим цілим, то і другий перетвориться на long long int
5.	unsigned long int α ∀ β	unsigned long int α **unsigned long int β**	якщо один операнд має тип unsigned long int, то і другий перетвориться на unsigned long int
6.	long int α unsigned int β	long int α **long int β**	якщо long int покриває всі значення unsigned int, то другий операнд перетвориться на long int
		unsigned long int α **unsigned long int β**	якщо long int не покриває всі значення unsigned int, то обидва операнди перетворяться на unsigned long int
7.	long int α ∀ β	long int α **long int β**	якщо один операнд має тип long int, то і другий перетвориться на long int
8.	unsigned int α ∀ β	unsigned int α **unsigned int β**	якщо один операнд має тип unsigned int, то і другий перетвориться на unsigned int
9. 10.	char α unsigned char α	**int α**	усі операнди типу char перетворюються на int
11.	short int α	**int α**	усі операнди типу short int перетворюються на int
12.	unsigned short int α	**unsigned int α**	усі операнди типу unsigned short int перетворюються на unsigned int

Таблиця 4.4. Результати перетворення типів у мові С при присвоюванні

Тип змінної	Тип виразу	Втрата інформації
signed char (8 бітів)	char (8 бітів)	якщо значення >127, то результат від'ємний
char (8 бітів)	short int (16 бітів)	старші 8 бітів
char (8 бітів)	int / long int (32 біти)	старші 24 біти
short int (16 бітів)	int (32 біти)	старші 16 бітів
int (32 біти)	long int (32 біти)	нема
long int (32 біти)	long long int (64 біти)	старші 32 біти (стандарт С99)
int	float	дробова частина
float	double	результат заокруглюється
double	long double	результат заокруглюється

Деякі компілятори при перетворенні char в int вважають змінну char позитивною незалежно від її значення. Інші компілятори вважають змінну char негативною, якщо вона має значення, більше від 127. Тому для забезпечення мобільності програм треба використовувати змінні типу unsigned char для зберігання символів, а змінні типу signed char і int для зберігання чисел.

Якщо якесь перетворення не подано в таблиці, то, щоб визначити, що саме втрачається в результаті цього перетворення, потрібно подати його у вигляді суперпозиції зазначених у таблиці перетворень і потім провести послідовні перетворення. Наприклад, перетворення double в int еквівалентне послідовному виконанню двох перетворень: спочатку double у float, а потім float в int.

Таким чином, якщо розмір значення більший від розміру змінної, якій воно присвоюється, то старші розряди цього значення будуть втрачені. Тому при присвоюванні треба ретельно слідкувати за типами, щоб не було спотворення інформації.

Основними групами операцій є: арифметичні, порівняння, логічні й побітові.

Арифметичні вирази. Як було сказано раніше, арифметичний вираз у широкому розумінні — це константа, або змінна, або звернення до функції, або вираз, який містить константи, змінні, звернення до функцій і операції.

В арифметичних виразах використовують такі операції (подано за спаданням пріоритетів):

++ (інкремент — збільшення), -- (декремент — зменшення);
– (унарний мінус);
* (множення), / (ділення), % (ділення за модулем — остача від ділення);
+ (додавання), – (віднімання).

Якщо операцію ділення / застосувати до цілих чисел чи символів, то дробова частина відкидається (7/4 дає 1; але 7./4, або 7/4., або 7./4. дає 1,75). Операція ділення за модулем повертає остачу від цілочисельного ділення (7%4 дає 3); цю операцію не можна застосовувати до чисел з плаваючою точкою.

Операція множення цілих чисел може призвести до втрати результату, тому тут у виразах обов’язково треба застосовувати перетворення типів:

```
double f;
int a=56789;
f=(double)a*a; // максимальне значення типу int 2147483647
```

Унарний мінус, як і в математиці, змінює знак операнда на протилежний.

Операції інкрементації (збільшення) і декрементації (зменшення) унарні. Їх застосовують тільки до даних цілого типу і вони відповідно додають і віднімають 1 від свого операнда; можуть бути префіксні (++b, --b) й постфіксні (d++, d--). Замість оператора a=a+1; можна записати оператор a++; чи ++a;. Відповідно замість оператора a=a-1; можна записати a--; чи --a;.

Проте між префіксною і постфіксною формами існує відмінність при їхньому використанні у виразах: якщо форма префіксна, то спочатку значення змінюється, а потім використовується у виразі, якщо постфіксна — то спочатку використовується у виразі, а потім змінюється:

```
x=5; y=++x; // ікс збільшується на 1 і присвоюється ігреку; отже,
            // x=6 і y=6
x=5; y=x++; // ікс присвоюється ігреку і ікс збільшується на 1;
            // отже, y=5 і x=6
```

Операторам інкрементації і декрементації треба надавати перевагу, оскільки більшість компіляторів створюють дуже швидкий і ефективний код, який виконується значно швидше, ніж код для операції присвоювання.

Операції з однаковим пріоритетом виконуються зліва направо. Використовуючи круглі дужки (це теж операція, причому з найвищим пріоритетом), можна змінити порядок обчислень. У мові C круглі дужки інтерпретуються компілятором так само, як і в будь-якій іншій мові програмування: вони надають операції (або послідовності операцій) найвищий пріоритет.

При програмуванні виразів передусім треба слідкувати, щоб кількість відкриваючих і закриваючих дужок була однаковою, і, крім того, жодна закриваюча дужка не повинна передувати відкриваючій.

Як приклад подамо мовою С такі арифметичні вирази: $\frac{\ln(a+3x)+8}{\cos(bx-2)}-2e^{-0,1|x+7|}$, $\frac{1-x^3}{1+\sqrt{x-2}}-7\log_3|x-2|$ (`double x;`); присвоїмо змінній k третю з кінця цифру числа n (`int k, n;`); присвоїмо змінній m першу після крапки цифру числа a (`int m; float a;`):

```
(log(a+3*x)+8)/cos(b*x-2)-2*exp(-0.1*fabs(x+7)) ;
(1-x*x*x)/(1+sqrt(x-2))-7*log(fabs(x-2))/log(3.) ;
k=(n/100)%10; ;
m =(int)(a*10)%10; .
```

Логічні вирази. Логічний вираз — це вираз, результатом обчислення якого є логічне значення «істина» або «хибність». Значення «істина» або «хибність» є результатом операцій порівняння або логічних операцій. У мові С значення «істина» подається числом 1, значення «хибність» — 0. Але, якщо замість логічного виразу умовою є число (включаючи додатні й від'ємні, цілі й дійсні), відмінне від нуля, то воно інтерпретується як «істина», і тільки число 0 інтерпретується як «хибність».

У стандарті С99 є тип даних — _Bool, у якому можна зберігати значення 1 і 0 («істина» (true) і «хибність» (false)) (у мові С ++ визначається ключове слово bool і вбудовані логічні константи true і false; отже, в написанні цього типу С99 і С++ несумісні). _Bool є цілим типом даних. У С99 є заголовок <stdbool.h>, в якому визначені імена макросів bool, true і false, завдяки чому можна легко створювати код, сумісний з С / С ++.

Логічні вирази можуть бути простими й складеними.

Простий логічний вираз записується за допомогою операцій відношень (порівняння):

< <= > >= == (дорівнює), != (не дорівнює)

Наприклад: **`a+b>c-d`** (спочатку виконуються арифметичні дії — вони мають вищий пріоритет, а потім порівняння отриманих результатів, тобто заданий вираз еквівалентний виразові **`(a+b)>(c-d)`**), **`3*x<=y+2`** (еквівалентно **`(3*x)<=(y+2)`**) , **`i==n+2`** (еквівалентно **`i==(n+2)`**).

Треба мати на увазі, що в загальному випадку до дійсних чисел не можна застосовувати операцію порівняння дорівнює ==, оскільки дійсні числа в пам'яті комп'ютера подаються неточно. Для дійсних чисел порівняння виду `x==y`, як правило, замінюють на порівняння `fabs(x-y)<=eps`, де eps — невелике число, яке вказує на допустиму похибку, наприклад $eps = 10^{-5}$ чи $eps = 10^{-8}$, fabs — функція знаходження модуля дійсного числа з бібліотеки <math.h>.

Складеним логічним виразом є вираз, у якому використовуються логічні операції (подано за спаданням пріоритетів):

! (не — унарна операція);
&& (і — бінарна операція);
|| (або— бінарна операція).

Таблиці істинностей логічних операцій такі:

x	$!x$
0	1
1	0

x	y	$x\&\&y$	$x\|\|y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Логічні операції мають нижчий пріоритет порівняно з операціями відношень (у мові Pascal — навпаки). Таким чином, у логічних виразах операції мають такі пріоритети (від вищого до нижчого):

унарні операції (математична зміна знака – , ++, - -, і логічна заперечення !)
математичні (*, /, %; +, -)
відношень (<, <=, >, >=, ==, !=)
логічні (&&; ||) — операція && має вищий пріоритет, ніж ||

Наприклад, вираз `5<10 || 10>5+2 && !(20<1)` еквівалентний виразові `(5<10) || ((10>(5+2)) && (!(20<1)))`, результатом буде 1 («істина»).

У мові C не визначена операція «виключного або» (exclusive or, або xor, або додавання за модулем 2; у мові Pascal є операція xor; у мові C є тільки побітове виключне «або»):

x	y	$x\ xor\ y$
0	0	0
0	1	1
1	0	1
1	1	0

Проте за допомогою логічних операцій можна написати вираз, який реалізує цю операцію:

```
(a&&!b) || (!a&&b)
```

або

```
(a||b) && !(a&&b)
```

Другий вираз містить на одну операцію заперечення менше, ніж перший, тому йому варто надати перевагу (порівняйте з симетричною різницею множин, яку ще називають об'єднанням за модулем 2: $A\Delta B = (A \cap \overline{B}) \cup (\overline{A} \cap B)$ або $A\Delta B = (A \cup B) \setminus (A \cap B) = (A \cup B) \cap \overline{(A \cap B)}$).

Як і в арифметичних виразах, для зміни порядку виконання операцій порівняння і логічних операцій можна використовувати круглі дужки. Наприклад, вираз `!0&&0||0`,

еквівалентний виразові `((!0)&&0) || 0`, дає 0. Але, якщо дужки розставити по-іншому `!(0&&0) || 0`, то результатом буде 1.

Для подання складних умов, наприклад, $x \in [-5;8)$ треба скористатися логічним виразом `(x>=-5) && (x<8)`. Якщо умову записати з використанням подвійної нерівності `-5<=x<8`, то компілятор не вкаже на помилку — цей вираз буде обчислюватися, як `(-5<=x)<8`: спочатку обчислиться вираз `(-5<=x)` і результат залежно від значення x буде 0 чи 1, а потім виконається порівняння 0<8 чи 1<8, кожне з яких дасть значення 1 («істина»). Якщо ж, наприклад, розглянути проміжок $x \in [-5;-2)$ і записати умову `-5<=x<-2`, то при будь-якому значенні x результат порівняння буде 0 («хибність»).

Нехай в пам'яті комп'ютера значення x подається точно. Тоді, якщо, наприклад, для функції $f(x) = \dfrac{1-x}{1-\sqrt{x+2}}$ перевіряти її ОДЗ (область допустимих значень) і записати логічний вираз `x+2>=0 && 1-sqrt(x+2)!=0`, то спочатку виконається перше порівняння, і якщо воно дасть результат «хибність», то наступне порівняння не перевірятиметься, оскільки кон'юнкція все одно матиме результат «хибність» (тобто корінь з від'ємного числа обчислюватися не буде і вираз `sqrt(x+2)` не дасть значення `NaN`). Аналогічно, якщо записати логічний вираз `x+2<0 || 1-sqrt(x+2)==0`, то спочатку виконається перше порівняння, і якщо воно матиме значення «істина», то наступне порівняння не перевірятиметься, оскільки диз'юнкція все одно дасть результат «істина».

Ще один приклад. Нехай задано функцію $\log_2 \dfrac{ax+3}{bx^2+cx-7}$. Тоді перевірка її ОДЗ буде описуватися виразом `b*x*x+c*x-7!=0 && (a*x+3)/(b*x*x+c*x-7)>0`; при цьому, якщо $bx^2+cx-7=0$, то буде обчислено тільки значення першої простої умови, що дасть «хибність», а друга умова, тобто знак виразу $\dfrac{ax+3}{bx^2+cx-7}$, перевірятися не буде і весь вираз набуде значення «хибність». Порушення ОДЗ даної функції можна описати виразом `b*x*x+c*x-7==0 || (a*x+3)/(b*x*x+c*x-7)<0;` при цьому, якщо

$bx^2 + cx - 7 = 0$, то буде обчислено тільки значення першої умови, що дасть значення «істина», а друга умова, тобто знак виразу $\frac{ax+3}{bx^2+cx-7}$, перевірятися не буде і весь вираз набуде значення «істина». Якщо, наприклад, у виразі `b*x*x+c*x-7!=0 && (a*x+3)/(b*x*x+c*x-7)>0` прості умови переставити місцями, то по-перше, вираз $\frac{ax+3}{bx^2+cx-7}$ буде довше обчислюватися, ніж $bx^2 + cx - 7$, а, по-друге, якщо знаменник набуває значення 0, то ділення на 0 не допустиме (хоч мова С на це не відреагує; мова Pascal видасть повідомлення про ділення на нуль і програма закінчить роботу).

Таким чином, виходячи зі сказаного вище, логічні вирази, в яких використовується кілька послідовних операцій кон'юнкції чи кілька послідовних операцій диз'юнкції, треба будувати так, щоб спочатку обчислювалися простіші вирази, а потім вирази, які потребують виконання більшої кількості операцій. Наприклад:

```
i==1 && (x<=5 || x>8) && (x+2<0 || 1-sqrt(x+2)==0);
i!=1 || (x>5 && x<=8) || (x+2>=0 && 1-sqrt(x+2)!=0).
```

Побітові (*порозрядні*) *операції*. На відміну від багатьох мов програмування, в мові С, яка розроблялася як мова системного програмування, визначено повний набір побітових операцій.

Побітовими є такі операції, як:
- побітове заперечення ~,
- зсув вліво <<,
- зсув вправо >>,
- побітове логічне «і» &,
- побітове виключне «або» (додавання за модулем 2) ^,
- побітове логічне «або» |.

Таблиці істинностей побітових операцій такі самі, як і відповідних логічних операцій.

Їх можна застосовувати тільки до даних типу char або int (до даних типу float, double, void або інших складніших типів побітові операції застосовувати не можна).

При написанні програм не можна сплутувати логічні й побітові операції. Наприклад, вираз з логічною операцією 2 && 5 (будь-яке ненульове число інтерпретується як «істина») дає результат «істина», якому в мові С відповідає 1; вираз з побітовою операцією 2 & 5 (логічний добуток виконується позиційно

над окремими бітами двійкового подання чисел: 010_2 & $101_2 = 000_2$) дає результат 0; вираз 5 && 6 дає результат 1, а 5 & 6 (101_2 & $110_2 = 100_2$) дає результат 4.

При використанні операції зсуву вліво на один розряд виконується множення числа на 2; аналогічно при зсувові вправо виконується ділення числа на 2. Зсув вправо чи вліво на *k* розрядів відповідає діленню чи множенню на 2^k.

Зсув може бути логічним чи арифметичним.

Логічний зсув застосовують до беззнакових чисел. При логічному зсувові всі біти зсуваються в певному напрямкові, крайній зсунутий біт зникає, а у звільнений розряд з іншого краю записується 0. Наприклад, якщо для зберігання значення числа виділено 8 бітів пам'яті і є число $141_{10}=10001101_2$, тоді:

— зсув вправо 10001101 >> 1 = 01000110 (відповідає цілочисельному діленню, тобто дробова частина відкидається, числа на 2 — 141/2=70);

— зсув вліво 10001101 << 1 = 00011010 (відповідає множенню числа на 2 — 141*2=282, але в даному випадку відбулося арифметичне переповнення і одержано неправильний результат 26).

Арифметичний зсув застосовують до чисел зі знаком. При арифметичному зсувові вправо у звільнений лівий розряд дублюється знаковий розряд. Арифметичний зсув вліво не відрізняється від логічного зсуву — зсуваються всі розряди і при цьому знаковий розряд виходить за розрядку сітку. Наприклад, візьмемо число –115, додатковий двійковий код якого 10001101. Тоді:

— зсув вправо 1 0001101 >> 1 = 1 1000110 (є додатковим кодом числа –58; використання в програмі звичайного ділення дає результат –115/2= –57, тобто спочатку абсолютне значення $115_{10}=01110011_2$ ділиться на 2 і буде одержано $00111001_2=57_{10}$, а потім змінюється знак). Цю особливість застосування зсувів до непарних від'ємних чисел треба мати на увазі (як правило, дії виконують з абсолютною величиною числа);

— зсув вліво 1 0001101 << 1 = 0 0011010 (відповідає множенню числа на 2 — –115*2= –230, але за рахунок виходу знакового розряду за розрядну сітку одержано число 26).

Операції зсувів схемотехнічно реалізуються дуже просто. Тому зсуви рекомендують використовувати для множення й ділення цілих чисел на числа 2, 4, 8, 16…

Перевірка парності цілого числа з застосуванням кон'юнкції з 1 (див. приклад 2.3.2) , теж виконується набагато швидше, ніж знаходження остачі від ділення на 2 із застосуванням операції %. Таким чином, використання побітових операцій, де це можливо, значно прискорює виконання алгоритмів на комп'ютерах. Код стає більш швидким і ефективним.

4.3. Структура лінійного алгоритму

Лінійний алгоритм містить тільки послідовності операторів присвоювання і звернення до різноманітних функцій, які, як правило, стосуються введення і виведення інформації. Перетворення інформації відбувається послідовно за певними формулами в тому порядку, в якому вони записані в алгоритмі. За таким алгоритмом виконується вся вказана в ньому послідовність дій без пропусків або повторень.

Оператор присвоювання мови C в найпростішій формі має вигляд:

ім'я змінної = вираз; .

Виразом тут може бути константа, змінна, звернення до функції і вираз, який містить, константи, змінні, звернення до функцій, з'єднані знаками операцій. Наприклад:

```
y=3+x-sin(2*x*x+1); .
```

При використанні функцій треба уважно слідкувати за їхнім призначенням і за типом їхнього результату. Наприклад:

```
double d;
d=abs(-3.1415); printf("%f\n", d);    // дає 3.000000
d=fabs(-3.1415); printf("%f\n", d);    // дає 3.141500
```

Виконання оператора присвоювання полягає в тому, що спочатку обчислюється значення виразу, а потім це значення присвоюється змінній, ім'я якої стоїть у лівій частині.

У випадку, якщо та сама змінна входить і в ліву, і в праву частини оператора присвоювання (наприклад, `i=i+2;`), то береться значення змінної, яке вона має в даний момент, це значення використовується при обчисленні виразу, а потім результат поміщається в комірку цієї змінної.

Якщо у виразі змішуються значення різних типів, то виконуються перетворення типів (див. табл. 4.3). Після цього значення правої частини перетворюється до типу змінної лівої частини (див. табл. 4.4).

Таким чином, при програмній реалізації алгоритмів, щоб не було спотворення інформації, треба слідкувати за типами даних як при записові виразів, так і при присвоюванні.

У мові C операція присвоювання є виразом, значенням якого є присвоєне значення. На відміну від більшості інших мов програмування (Pascal, BASIC, Fortran), в яких при-

своювання можливе тільки в окремому операторі, у мові С оператор присвоювання може використовуватися в будь-якому виразі. Будь-який вираз з операцією присвоювання, взятий у круглі дужки, має значення, рівне присвоюваному. Наприклад:

оператор `z=(x=a+2)*(y=a-7);` еквівалентний трьом операторам: `x=a+2; y=a-7; z=x*y;` ;

оператор `y=((x=2+3)<7);` еквівалентний двом операторам `x=2+3; y=x<7;` (у другому операторі порівняння x<7 брати в дужки не обов'язково, бо присвоювання має нижчий пріоритет, ніж порівняння);

конструкція `((c=getchar())!='\n')` дає можливість присвоїти змінній c введене значення, а потім введене значення порівняти і одержати істинний результат, якщо це значення не є символом '\n' .

У дужках можна записати кілька операторів, розділених комою, які утворюють складний вираз. Такі оператори виконуються послідовно зліва направо; весь вираз набуде значення останнього обчисленого виразу. Оператор `c=(a=b, b=getchar());` (дужки після першого знака присвоювання потрібні, оскільки операція кома має нижчий пріоритет, ніж операція присвоювання) еквівалентний таким трьом операторам: `a=b; b=getchar(); c=b;`.

Оператори збільшення (інкремент) і зменшення (декремент) неявно містять операцію присвоювання. Наприклад, `++a;` і `a++;` , `--b;` і `b--;` еквівалентні операторам `a=a+1;` і `b=b+1`. Оператор `c=(a++)*(--b);` еквівалентний послідовності трьох операторів: `b=b-1; c=a*b; a=a+1;`.

Мова С дає можливість в одному операторі присвоїти те саме значення кільком змінним (але при оголошенні змінних таке присвоювання початкових значень недопустиме). Таке присвоювання називають множинним присвоюванням. Наприклад, `a=b=c=d=1;` еквівалентно виконанню чотирьох присвоювань `d=1;` , `c=1;` , `b=1;` , `a=1;` .

Різновидом оператора присвоювання є складене присвоювання, в якому запис скорочується. Наприклад, замість оператора `x=x+5;` можна записати `x+=5;` (тут операція += вказує компілятору, що до x треба додати 5). Складені операції існують для всіх бінарних арифметичних операцій +=, –=, *=, /=, %= і побітових операцій <<=, >>=, &=, |=, ^=. Складене присвоювання значно компактніше, ніж відповідне просте присвоювання, тому його іноді називають стенографічним.

Виклик функцій. До функцій можна звертатися не тільки у виразах, а й окремим оператором. Оператор виклику функції має вигляд (зверніть увагу, навіть, якщо функція не має параметрів, круглі дужки все одно треба записувати):

ім'я_функції();

або

ім'я_функції(параметри);

Наприклад, `scanf("%f%f", &a, &x);` , `printf("y=%0.3f z=%0.3f t=%0.3f", y, z, t);` , `printf("%f\n", sin((double)n));` , `abc();` .

Порожній оператор не виконує ніяких дій. Він вставляється в тих місцях програми, де необхідне використання оператора, але не потрібна за логікою алгоритму ніяка дія. Записується як `;` або `{}` .

Блок операторів (складний оператор) — це послідовність операторів, взятих у фігурні дужки. Після закриваючої фігурної дужки крапка з комою не ставиться (якщо поставити, то це не буде помилкою, а буде сприйнято, як порожній оператор). У блоці після відкриваючої фігурної дужки можна оголошувати змінні, але вони використовуються лише в даному блоці, а поза блоком вони недоступні. Такі змінні існують тільки під час виконання програмного блоку, в якому вони оголошені, створюються вони при вході в блок, а при виході з нього стають недоступними. Крім того, змінна, оголошена в одному блоці, не має ніякого відношення до змінної з тим самим іменем, оголошеної в іншому блоці.

Хоч лінійні алгоритми є найпростішими і мають просту логіку, проте обчислення виразів, які містяться в операторах присвоювання, може бути до-

сить складним процесом, який потребує багато часу і додаткової пам'яті для зберігання проміжних результатів. Вирази можуть бути настільки складними, що можуть викликати збої в роботі програми. Тому складні вирази доцільно розбивати на кілька простіших, при цьому запис виразу будь-якої складності має бути лінійним.

Якщо в обчисленнях є підвирази, які використовуються кілька разів, то доцільно обчислити їх один раз і запам'ятати в робочих змінних. Наприклад, нормалізація вектора (x, y, z) полягає в побудові відповідного вектора одиничної довжини $\left(\frac{x}{\sqrt{x^2 + y^2 + z^2}}, \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \frac{z}{\sqrt{x^2 + y^2 + z^2}} \right)$. У цьому випадку доцільно в робочу змінну запам'ятати значення $1 \big/ \sqrt{x^2 + y^2 + z^2}$, щоб тричі не виконувати ті самі обчислення, які потребують досить багато часу. Можуть бути і набагато складніші вирази, в яких деякі обчислення повторюються (див. також п. 4.5).

Крім того, якщо, наприклад, a — дійсне число, а *m*, *n*, і *k* — цілі числа, то при обчисленні значення вирази виду $\frac{a}{m \cdot n \cdot k}$ для запобігання арифметичному переповненню у знаменнику при виконанні проміжних обчислень, особливо при можливих великих значеннях змінних *m*, *n* і *k*, вираз доцільніше програмувати з використанням операції ділення `a/m/n/k`, ніж з використанням множення `a/(m*n*k)`. Але, якщо мова програмування надає можливість застосувати в знаменнику явне перетворення типів `a/((double)m*(double)n*(double)k)`, то точність обчислень підвищиться за рахунок меншої кількості заокруглень і, крім того, множення виконується швидше, ніж ділення.

Також для підвищення точності і прискорення обчислень, наприклад, вираз $a^2 - b^2$ треба програмувати `(a-b)*(a+b)`.

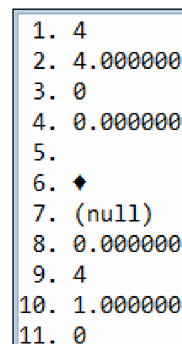
При роботі з цілими числами треба пам'ятати, що операція ділення двох цілих чисел дає ціле число (тобто дробова частина просто відкидається, наприклад, `7/8` дає результат 0, а `7./8` дає результат 0,875). Також треба звернути увагу, що для цілих чисел, наприклад, вирази $\frac{a}{b} + \frac{c}{d}$ і $\frac{ad + bc}{bd}$

можуть дати різні результати ($\frac{5}{2}+\frac{7}{4}$ дає результат 2+1=3, а $\frac{5\cdot 4+2\cdot 7}{2\cdot 4}$ дає результат 4).

Мова C не має операторів для введення і виведення даних. Для форматованого введення даних найчастіше використовують функцію scanf, а для форматованого виведення — функцію printf зі стандартної бібліотеки stdio.h. Правильне використання форматів введення й виведення при зверненні до функцій scanf і printf є дуже важливим. Від цього залежить і правильність обчислень, у яких беруть участь вхідні дані, і правильність візуалізації результатів — алгоритм може бути ідеальним, але не правильно введене значення вплине на результат обчислень і, наприклад, тестове виконання алгоритму буде давати неправильний результат; також програма може порахувати все правильно, а виведення за невідповідним до типу одержаного значення форматом відобразить зовсім інший результат. Крім того, оскільки формати введення чи виведення вказуються «на відстані» (у першому параметрі — рядку формату) від аргументу (змінної чи виразу у списку параметрів), то не завжди вистачає уважності швидко виявити помилку.

Поданий нижче приклад ілюструє результати правильного і неправильного застосування форматів при виведенні даних (тут тільки у випадках 1, 2, 9 і 10 виведено правильний результат):

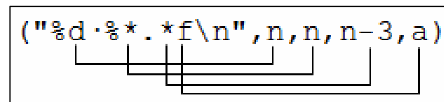
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
int main() {
  int k=4;
  printf(" 1. %d \n",4);
  printf(" 2. %f \n",4.);
  printf(" 3. %d \n",4.);
  printf(" 4. %f \n",4);
  printf(" 5. %c \n",4.);
  printf(" 6. %c \n",4);
  printf(" 7. %s \n",4.);
  printf(" 8. %f \n",k);
  printf(" 9. %d \n",k);
  printf("10. %f \n",cos((double)0));
  printf("11. %d \n",cos((double)0));
  system("pause");
  return 0;
}
```



У функції printf для виведення даних типу float у формі з фіксованою точкою застосовують формат %f, для виведення даних типу double у формі з фіксованою точкою можна застосовувати формати %f і %lf.

При виведенні даних модифікатором формату функції printf, який вказує мінімальну ширину поля виведення, може бути ціле число чи зірочка * (вказує на відповідний за порядком параметр зі списку виведення). Використання модифікатора зірочки дає можливість регулювати ширину поля виведення під час виконання програми. Розглянемо застосування модифікатора * у функції printf:

```
int n=5;
double a=7.;
printf("%d %*.*f\n",n,n,n-3,a);
```



Цей оператор еквівалентний такому:

```
printf("%d %5.2f\n",n,a);
```

У функції scanf для введення даних типу float застосовують формат %f, для введення даних типу double треба застосовувати формат %lf (застосування формату %f призводить до введення неправильного значення).

При введенні рядкових даних, щоб не зчитувалося більше символів, ніж відведено пам'яті під рядок символів, треба у форматі задати модифікатор — кількість символів. Наприклад, якщо є рядок символів `char st[10]`, то, щоб не зчитувалося більше, ніж 9 символів у st, треба вказати формат %9s (буде зчитано або менше дев'яти символів — до першого пробільного, або рівно дев'ять, якщо пробільний символ розміщується далі).

У функції scanf модифікатор *, який ставиться між знаком % і кодом формату, вказує на пропуск введеної з клавіатури інформації. Наприклад:

```
int n;
printf("Введіть символ і ціле число:\n");
scanf("%*c%d",&n);
printf("Зчитана інформація:\n\t%d\n",n);
```

```
Введіть символ і ціле число:
a    1234
Зчитана інформація:
        1234
```

```
Введіть символ і ціле число:
1234
Зчитана інформація:
        234
```

```
Введіть символ і ціле число:
 1234
Зчитана інформація:
        1234
```

У мові програмування C часом виникає проблема з введенням даних функцією scanf: коли натискати клавішу Enter, то символ переходу на новий рядок зберігається в буфері вводу і присвоюється при наступному виклику функції scanf символьній змінній. Тому зайвого символу можна позбутися, використовуючи в форматі модифікатор *.

Якщо при використанні функції scanf з клавіатури вводити «зайву інформацію», то вона також має бути подана в рядку формату. Наприклад, якщо при введенні дати використовувати формат з крапкою день.місяць.рік, то в рядку форматів теж треба подати крапки: `scanf("%d.%d.%d",&d,&m,&r);`; якщо з клавіатури вводити інформацію виду `i=3, j=5`, то треба використати такий формат: `scanf("i=%d, j=%d",&a,&b);`.

Проектування введення і виведення при програмній реалізації алгоритму є однією з найбільш вважливих проблем. Повідомлення програми повинні бути зрозумілими будь-кому, хто з нею буде працювати, не потребувати додаткових пояснень. Проте найголовнішим є захист програми від помилок користувача при введенні даних, що може значно збільшити код програми, і такий код обов'язково вже буде містити і розгалуження, і цикли.

Наявність навіть у найпростішій програмі коментарів робить її зрозумілішою, пояснює алгоритм. Добре прокоментована програма є ознакою кваліфікованості і сумлінності програміста.

4.4. Приклади лінійних алгоритмів

Розглянемо кілька прикладів лінійних алгоритмів.

Приклад 4.4.1. Записати мовою C, не використовуючи ні логічного оператора, ні оператора вибору, вирази для обчислення значення максимуму з двох чисел і для обчислення мінімуму з двох чисел.

Розв'язання. Враховуючи те, що результатом порівняння буде число 1 (відповідає значенню «істина») чи 0 (відповідає значенню «хибність»), то для обчислення значення максимуму з двох чисел і мінімуму з двох чисел можна записати такі вирази:

```
max=(a>=b)*a+(a<b)*b; ,
min=(a<=b)*a+(a>b)*b; .
```

Недоліком такого запису є те, що по суті те саме порівняння (тільки з іншим знаком, наприклад, при обчисленні максимуму це знаки >= і <) виконується двічі. Якщо використати робочу змінну, то ці вирази можна записати так:

```
max=(ind=a>=b)*a+!ind*b; ,
min=(ind=a<=b)*a+!ind*b; .
```

Проте порядок обчислення значень доданків може залежати від компілятора, тому тут краще записати таку послідовність операторів:

```
ind=a>=b;
max=ind*a+!ind*b; ,
min=!ind*a+ind*b; .
```

Приклад 4.4.2. Подати мовою програмування C алгоритм обчислення довжини кола і площі круга (величина радіуса вводиться з клавіатури, на екран виводяться обчислені значення у форматі з фіксованою точкою).

Код програми має вигляд:

```
#include <stdlib.h>
```

```
#include <stdio.h>
#define _USE_MATH_DEFINES   // для константи PI - M_PI
#include <math.h>
/* Обчислення довжини кола і площі круга */
int main() {
  double r, l, s;   // радіус, довжина кола, площа круга
  system("chcp 1251 & cls");
  printf("Введіть r\n");
  scanf("%lf", &r);
  l=2*M_PI*r;
  s=M_PI*r*r;
  printf("Результат:\n\tl=%0.3f  s=%0.3f\n", l, s);
  printf("\n\n"); system("pause");
  return 0;
}
```

```
Введіть r
5
Результат:
        l=31.416  s=78.540
```

```
Введіть r
1
Результат:
        l=6.283  s=3.142
```

Приклад 4.4.3. Запрограмувати алгоритм (див. приклад 1.4.3) обчислення площі трикутника за формулою Герона:

$$s = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}, \text{ де } p = \frac{a+b+c}{2},$$

вважаючи, що довжини сторін трикутника a, b, c задано коректно (тобто, трикутник існує).

Код програми має вигляд:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
/* Обчислення площі трикутника за формулою Герона
   (без перевірки умов існування трикутника) */
int main() {
  double a, b, c, p, s;
  system("chcp 1251 & cls");
  printf("Введіть a, b, c\n");
  scanf("%lf%lf%lf", &a, &b, &c);
  p=(a+b+c)/2;
```

```
  s=sqrt(p*(p-a)*(p-b)*(p-c));
  printf("s=%0.3f\n",s);
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Введіть a, b, c
3 4 5
s=6.000
```

Приклад 4.4.4. Подати мовою програмування С алгоритм розв'язування квадратного рівняння $ax^2 + bx + c = 0$. Скористатися формулами $D = b^2 - 4ac$, $x_{1,2} = \dfrac{-b \pm \sqrt{D}}{2a}$ (вважати, що рівняння має розв'язок, тобто значення дискримінанта $D \geq 0$).

Код програми має вигляд:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
/* Обчислення коренів квадратного рівняння
   (вважається, що дискримінант не від'ємний) */
int main() {
  double a, b, c, sD, x_1, x_2;
  system("chcp 1251 & cls");
  printf("Розв'язування квадратного рівняння\n");
  printf("  Введіть a, b, c\n");
  scanf("%lf%lf%lf", &a, &b, &c);
  sD=sqrt(b*b-4*a*c);
  x_1=(-b-sD)/(2*a);
  x_2=(-b+sD)/(2*a);
  printf("x1=%0.3f,  x2=%0.3f\n",x_1,x_2);
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Розв'язування квадратного рівняння
  Введіть a, b, c
1 -1 -12
x1=-3.000,  x2=4.000
```

```
Розв'язування квадратного рівняння
  Введіть a, b, c
1 -2 1
x1=1.000,  x2=1.000
```

В останніх двох прикладах лінійних алгоритмів вважається, що користувач задає коректні вхідні дані. Якщо ж таких припущень не робити, то треба

перевіряти, чи при певних вхідних даних задачі мають розв'язки. Але це вже будуть алгоритми з розгалуженнями, які розглядаються в наступному розділі.

4.5. Складність виразу. Оптимізація обчислень

Швидкість виконання програми, яка не містить розгалужень і циклів, визначається кількістю обчислень при отриманні значень виразів, які містяться в операторах присвоювання і операторах виведення.

Кількість обчислень, потрібних для отримання значення виразу, є його складністю. Звичайно ж, швидкість обчислення значень функцій, мультиплікативних операцій (множення, ділення, знаходження остачі від ділення), адитивних операцій (додавання й віднімання) і логічних операцій значно відрізняються і залежать від реалізації мови програмування і мікропроцесора.

Проте можна припустити, що швидкості обчислення, наприклад, усіх трансцендентних функцій (тригонометричні, гіперболічні, показникова, логарифмічна) приблизно рівні, швидкості виконання мультиплікативних операцій приблизно рівні, швидкості виконання адитивних і побітових логічних операцій теж приблизно рівні.

Часову складність виконання будь-яких дій, зокрема арифметичних і логічних операцій, викликів функцій, можна вимірювати кількістю тактів мікропроцесора. Кількість потрібних тактів для виконання команд мікропроцесора залежить від їхньої складності, від того, де розміщуються операнди команд — у регістрах, регістрі-пам'яті, пам'яті-регістрі чи пам'яті. Найменше тактів виконується, коли операнди розміщуються в регістрах. Кількість тактів також залежить від довжини операнда; проте часом обробка операндів більшої довжини виконується за меншу кількість тактів, ніж меншої довжини. Крім того, при обчисленнях перед чи після виконання арифметичних команд можуть виконуватися команди корекції даних, що теж потребує певної кількості тактів.

З метою прискорення обчислень операції з плаваючою арифметикою оптимально реалізує спеціальний математичний співпроцесор. Математичний співпроцесор, крім операцій над числами з плаваючою точкою, також реалі-

зує найчастіше використовувані математичні функції: знаходження абсолютної величини числа, квадратний корінь, трансцендентні функції (звичайні і зворотні тригонометричні, звичайні і зворотні гіперболічні, логарифмічні і показникові), які потребують багато тактів на виконання.

Нижче в таблиці 4.5 вказано кількості тактів при виконанні арифметичних і логічних команд різними x86-сумісними мікропроцесорами при обробці даних довжини до 16 чи 32 бітів.

Таблиця 4.5. Кількості тактів при виконанні арифметичних і логічних команд x86-сумісними мікропроцесорами

Команда	**Кількість тактів**		
	Процесори i8086 [28, с. 107- 118; 33, с. 879-881] (до 16 бітів)	Процесор Intel Core 2 [41, с. 177-182] (до 32 бітів)	64-бітний x86-сумісний процесор VIA Nano 3000 [41, с. 415-417] (до 32 бітів)
Пересилання даних	2…22	2…4	2…3
Додавання двох цілих операндів	3…29	2…11	2…8
Віднімання двох цілих операндів	3…29	2…11	2…8
Інкремент, декремент	2…27	2…9	2…8
Зміна знака операнда	3…28	2…9	2…8
Порівняння двох операндів за результатом віднімання	3…21	2	2
Цілочисельне множення на число без знака	70…172	4…8	3…6
Цілочисельне множення на число зі знаком	128…154	4…8	3…6
Цілочисельне ділення на число без знака (результат і остача)	80…180	13…23	22…30
Цілочисельне ділення на число зі знаком (результат і остача)	101…202	13…23	18…28
Додавання двох операндів з плаваючою точкою	—	4	3
Віднімання двох операндів з плаваючою точкою	—	4	3
Множення на число з плаваючою точкою	—	6	5
Ділення на число з плаваючою точкою	—	7…22	14…23
Інверсія всіх бітів операнда	3…28	2…9	2…8
Побітові логічні «і», «або», виключне «або» двох операндів	3…29	2…8	2…8
Побітовий зсув вліво, вправо на 1 розряд	2…27	2…9	2…4

Таким чином, треба враховувати, що різні процесори виконують однакові команди за різну кількість тактів, причому кількість тактів, витрачених на команду, може змінюватися від одного такту до десятків або навіть сотень. На сучасних мікропроцесорах прості команди (пересилання, додавання, віднімання, логічні тощо) за рахунок розпаралелювання мікрооперацій можуть виконуватися швидше, ніж за один такт.

Перемикання між командами роботи з цілими і плаваючими числами теж потребує часу на «затримку переходу» до 3 тактів залежно від мікропроцесора. Тому, якщо швидкість виконання є суттєвим показником роботи програми, треба уникати змішування обчислень з плаваючою точкою і цілими числами.

Виклик функції, наприклад у С-програмі, залежно від кількості параметрів і від мікропроцесора потребує 25-250 (відповідно до давнішої літератури) чи для функцій з невеликою кількістю параметрів 15-30 (новіші дані) тактів мікропроцесора.

Нехай t_a, t_m, t_f — відповідно час виконання адитивної операції, мультиплікативної операції, обчислення значення функції. Щодо часових складностей t_a, t_m, t_f виконуються такі співвідношення:

$$t_a < t_m \ll t_f .$$

Звідси випливає, що особливу увагу треба приділяти мультиплікативній (особливо використанню операції ділення) і функціональній складності за рахунок адитивної.

Для зменшення складності виразів у програмах використовують тотожні перетворення і попередні обчислення загальних підвиразів.

Розглянемо приклади розрахунку обчислювальних складностей різних виразів і підходи щодо зменшення їхніх складностей.

Приклад 4.5.1. При повороті системи координат на кут α застосовують формули переходу від старих координат до нових:

$$x_1 = x\cos\alpha + y\sin\alpha ,$$

$$y_1 = -x\sin\alpha + y\cos\alpha .$$

Для знаходження значення виразу $x\cos\alpha + y\sin\alpha$ треба виконати два обчислення значень тригонометричних функцій, два множення й одне додавання. Це є його складністю. Обчислення значення виразу $-x\sin\alpha + y\cos\alpha$ має ту саму складність.

Нехай t_x — час обчислення виразу $x\cos\alpha + y\sin\alpha$; t_y — час обчислення виразу $-x\sin\alpha + y\cos\alpha$. Тоді:

$$t_x = 2t_f + 2t_m + t_a,$$

$$t_y = t_x.$$

Одержано часові міри складності для виразів $x\cos\alpha + y\sin\alpha$ і $-x\sin\alpha + y\cos\alpha$.

У розглянутих виразах тотожні перетворення, направлені на зменшення складності, виконати не можливо (все і так дуже просто), а значення тригонометричних функцій можна обчислити по одному разу і зберегти в робочих змінних. Тобто в програмі, замість операторів:

```
x1=x*cos(a)+y*sin(a);
y1=-x*sin(a)+y*cos(a); ,
```

сумарна складність виконання яких становить $t_{xy} = 4t_f + 4t_m + 2t_a$, можна записати таку послідовність операторів:

```
rcos=cos(a);
rsin=sin(a);
x1=x*rcos+y*rsin;
y1=-x*rsin+y*rcos;
```

У результаті в цьому фрагменті програми зекономлено два обчислення тригонометричних функцій і сумарна складність становить $t_{xy} = 2t_f + 4t_m + 2t_a$. Враховуючи те, що операції множення й додавання порівняно з обчисленням значення тригорометричної функції виконуються набагато швидше, то часова складність зменшилася майже в 2 рази.

Можна ще зменшити мультиплікативну складність обчислень виразів, якщо скористатися таким алгоритмом:

```
rcos=cos(a);
rsin=sin(a);
u=(rcos+rsin)*(x+y);
v=x*rsin;
w=x*rcos;
x1=u-v-w;
y1=w-v;
```

часова складність якого становить $t_{xy} = 2t_f + 3t_m + 5t_a$. Те, який з двох останніх алгоритмів кращий щодо швидкодії, залежить від реалізації операції множення в комп'ютері.

Приклад 4.5.2. При обчисленні значень багаточленів можна скористатися схемою Горнера. Нехай треба обчислити значення квадратного багаточлена $y = ax^2 + bx + c$. Оскільки операція піднесення до квадрату не реалізована, то вираз запишеться так:

```
y=a*x*x+b*x+c;
```

Вираз має складність $t_y = 3t_m + 2t_a$. Скориставшись схемою Горнера, вираз можна записати так: $y = (ax + b)x + c$. Тоді:

```
y=(a*x+b)*x+c;
```

Тут зекономлено одну операцію множення — складність виразу $t_y = 2t_m + 2t_a$.

Крім того, якщо є, наприклад, вираз виду $y = ax^4 + bx^2 + c$, то, застосувавши схему Горнера, отримаємо $y = (ax^2 + b)x^2 + c$. Але тут квадрат змінної x обчислюється двічі. Для зменшення складності обчислень можна використати тимчасову змінну $r = x^2$ і порахувати результат $y = (ar + b)xr + c$. При цьому буде виконано на одну операцію множення менше — замість `y=(a*x*x+b)*x*x+c;` одержимо: `r=x*x; y=(a*r+b)*r+c;`, а порівняно з початковим виразом `y=a*x*x*x*x+b*x*x+c;` виконується на три операції множення менше, тобто замість складності $t_y = 6t_m + 2t_a$ одержано складність $t_y = 3t_m + 2t_a$.

Приклад 4.5.3. Знайти корені рівняння $y = x^2 - px + q$ (вважаючи, що $D > 0$). Алгоритм знаходження коренів можна записати так:

```
D=p*p-4*q;
x1=(p-sqrt(D))/2;
x2=(p+sqrt(D))/2;
```

Обчислювальна складність поданого алгоритму — $t = 2t_f + 4t_m + 3t_a$.

Проте, значення кореня з дискримінанта можна порахувати один раз:

```
sD=sqrt(p*p-4*q);
x1=(p-sD)/2;
x2=(p+sD)/2;
```

Обчислювальна складність алгоритму зменшилася — $t = t_f + 4t_m + 3t_a$.

Крім того, враховуючи, що за теоремою Вієта $p = x_1 + x_2$, можна записати такий алгоритм:

```
d=p*p-4*q;
x1=(p-sqrt(d))/2;
x2=p-x1;
```

складність якого становить $t = t_f + 3t_m + 3t_a$ (на одне обчислення функції і на одну операцію ділення менше).

При роботі з тригонометричними функціями для спрощення виразів не треба забувати про тригонометричні тотожності. Наприклад, $\sin^2 x + \cos^2 x = 1$, $\cos^2 x - \sin^2 x = \cos 2x$, $\mathrm{ctg}x - \mathrm{tg}x = 2\mathrm{ctg}2x$, $3\sin x - 4\sin^3 x = \sin 3x$ тощо.

Спрощуючи вирази, також треба враховувати, що при діленні цілих чисел від порядку виконання операцій може залежати результат. Так, якщо у формулі `28+(n+n/8)%2+2%n+1/n*2` для обчислення кількості днів невисокосного року в місяці з номером *n* «вдосконалити» останній доданок `28+(n+n/8)%2+2%n+2/n`, то у лютому замість 28 буде нараховано 29 днів, як у високосному році.

Звичайно ж, якщо програма виконує небагато обчислень, то оптимізувати вирази на противагу їхній наочності навряд чи доцільно — незалежно від того, чи код оптимізований, чи ні, відмінність щодо швидкості виконання

програми буде незначною і непомітною. Але, якщо обчислення повторюються в програмі досить часто (наприклад, циклічно сотні чи тисячі разів) і час виконання програми є суттєвим параметром, то оптимізація потрібна.

Запитання та завдання для самоконтролю

1. Схарактеризуйте лінійні алгоритми. У яких випадках використовують лінійні алгоритми? Чи можуть бути в лінійному алгоритмі пропуски або повторення кроків при виконанні?

2. Що таке вираз у широкому розумінні? Як будують арифметичні вирази? Що може бути операндом в арифметичному виразі? Які операції використовують в арифметичних виразах?

3. Що таке логічний вираз? Як у мові C подаються значення «істина» і «хибність»? Які числа в мові C інтерпретуються як «істина», а які як «хибність»? Як треба будувати логічні вирази для оптимізації обчислень? Як перевірити рівність двох дійсних чисел?

4. Які побітові операції є в мові C, до даних якого типу їх можна застосовувати? Як швидко виконати множення чи ділення цілого числа на 2? Чим відрізняється логічний зсув від арифметичного?

5. Які прийоми можна застосовувати при обчисленні арифметичних виразів для підвищення точності і для підвищення швидкості обчислень?

6. Як визначити складність виразу? Чим вимірюють часову складність виконання арифметичних і логічних операцій, викликів функцій? Від чого залежить кількість потрібних тактів для виконання команд мікропроцесора? Чому треба уникати змішування обчислень з плаваючою точкою і цілими числами?

7. Чи однакова складність виконання логічних, адитивних і мультиплікативних операцій, обчислення значення функції? Як співвідносяться швидкості обчислення значень функцій, виконання мультиплікативних, адитивних і логічних операцій?

8. Які прийоми використовують для зменшення складності виразів і чи завжди це доцільно робити?

Практичне завдання № 5

«Програмування обчислювальних алгоритмів лінійної структури»

Мета виконання завдання: навчитися складати й програмувати найпростіші обчислювальні алгоритми лінійної структури, а також використовувати функції стандартної математичної бібліотеки math.h.

Завдання. Послідовно виконати такі дії: ввести значення параметра a і змінної x; трьом змінним y, z, t дійсного типу послідовно присвоїти значення трьох заданих функцій — при цьому константи для першої функції f_1 визначити в декларативній частині, функцію f_2 записати з мінімумом операцій, функцію f_3 запрограмувати без оптимізації; вивести обчислені значення на екран; виконати переприсвоєння значень змінних $y \leftarrow z$, $z \leftarrow t$, $t \leftarrow y$; знову вивести на екран значення змінних.

Подати блок-схему алгоритму і запрограмувати його. Текст програми структурувати. Коментарі обов'язкові — 17-25 %.

Вказівки. При програмуванні арифметичних виразів ОДЗ не перевіряйте (при виконанні програми підберіть вхідні значення так, щоб не виникало особливостей), враховуйте пріоритети операцій (див. п. 4.2), зверніть увагу на типи операндів (див. п. 4.2, 4.3), правильно застосовуйте формати для введення й виведення значень змінних (див. п. 4.3). Щоб функцію f_2 записати з мінімумом операцій, треба, де можливо, обчислити значення дробів, застосувати схему Горнера (див. п. 4.5). Для переприсвоювання значень змінних використайте робочу змінну. Для контролю обчислення виконайте в програмі Excel.

Варіанти завдань

1. $\sqrt{\dfrac{\ln x+1}{5x+a}}+|x+5tgx|$, $\quad \dfrac{1}{8}x-\dfrac{3}{10}x^2-e^{ctgx+1}$, $\quad \dfrac{5}{8}x^{-4,1}$;

2. $\dfrac{7x-a}{3+\log_2 x}+\sqrt{|x+3|}$, $\quad \dfrac{1}{4}x^2-\dfrac{4}{10}x^3-e^{3x+5}$, $\quad \dfrac{4}{5}x^{-0,3}$;

3. $\dfrac{2x-a}{3-\sqrt{x-1}}-3\cos|x|$, $\quad \dfrac{3}{2}x^3-\dfrac{3}{10}x+7e^{-2tgx}$, $\quad \dfrac{4}{5}x^{6,8}$;

4. $\sqrt{\dfrac{x+2}{2-ax}}+4\left|\log_2 x-1\right|,\quad \dfrac{5}{4}x^2+\dfrac{3}{10}x-2e^{3tgx+5},\quad \dfrac{2}{5}x^{2,6};$

5. $\sqrt{\dfrac{5x-a}{x^2-1}}+\cos\left|x-1\right|,\quad \dfrac{5}{4}x^2-\dfrac{4}{20}x-2e^{\sin x},\quad \dfrac{4}{5}x^{5,4};$

6. $\dfrac{\sqrt{\ln x+3}}{3x+a-4}+ctg\left|x-4\right|,\quad \dfrac{7}{4}x^3-\dfrac{4}{5}x+xe^{-3tgx},\quad \dfrac{7}{8}x^{0,7};$

7. $\dfrac{\sqrt{x+3}}{2a-x}+\log_3\left|\cos x+3\right|,\quad \dfrac{8}{10}x+\dfrac{7}{4}x^3-e^{1+x},\quad \dfrac{3}{8}x^{-4,4};$

8. $\sqrt{\dfrac{x+a}{2x+5}}+\left|\sin x-2\right|,\quad \dfrac{9}{2}x+\dfrac{3}{10}x^2+e^{ctg\,2x},\quad \dfrac{1}{2}x^{6,3};$

9. $\dfrac{2x+a}{\sqrt{x}-3}-\log_3\left|x-4\right|,\quad \dfrac{1}{2}x+\dfrac{3}{10}x^2-e^{-tgx+2},\quad \dfrac{3}{4}x^{3,1};$

10. $\dfrac{\sqrt{\ln x+2}}{4x-a}+\sin\left|x+4\right|,\quad \dfrac{3}{4}x^2-\dfrac{4}{5}x^3+e^{ctg(x+2)},\quad \dfrac{3}{5}x^{-0,2};$

11. $\sqrt{\dfrac{x+1}{2-ax}}-\log_5\left|x-1\right|,\quad \dfrac{1}{4}x^2+\dfrac{7}{5}x+4e^{tg(x-6)},\quad \dfrac{3}{5}x^{3,4};$

12. $\dfrac{5x-a}{1-\sqrt{x+1}}-ctg\left|x-5\right|,\quad \dfrac{3}{4}x+\dfrac{6}{5}x^3-7e^{x-8},\quad \dfrac{1}{4}x^{4,2};$

13. $\dfrac{\sqrt{\ln(x+1)}}{5x-a}+\sin\left|x+5\right|,\quad \dfrac{7}{2}x^2-\dfrac{2}{5}x^3+2e^{tgx},\quad \dfrac{3}{8}x^{-3,8};$

14. $\sqrt{\dfrac{4x-a}{x-2}}+\log\left|x-2\right|,\quad \dfrac{5}{2}x+\dfrac{7}{5}x^2-e^{4tgx},\quad \dfrac{3}{4}x^{1,9};$

15. $\dfrac{\sqrt{x-2}}{3-ax}+\left|\cos x-4\right|,\quad \dfrac{1}{4}x^3+\dfrac{3}{10}x^2+7e^{1-tgx},\quad \dfrac{7}{8}x^{-0,12};$

16. $\dfrac{x-a}{1-7\sqrt{x}}-\log_5\left|x-1\right|,\quad \dfrac{3}{4}x^2-\dfrac{7}{10}x+2e^{-tgx},\quad \dfrac{3}{5}x^{-5,3};$

17. $\sqrt{\dfrac{x+a}{5x-3}}+\left|\sin x\right|,\quad \dfrac{3}{2}x^3-\dfrac{7}{10}x+e^{ctgx},\quad \dfrac{5}{8}x^{4,8};$

18. $\dfrac{\sqrt{\ln x+5}}{3x+a}+\log_3\left|x+5\right|,\quad \dfrac{1}{8}x^2-\dfrac{8}{10}x^3+e^{tg3x},\quad \dfrac{1}{8}x^{-4,4};$

19. $\dfrac{3x-2a}{\sqrt{x}+5}-\ln|x+2|$, $\quad \dfrac{5}{2}x^2-\dfrac{7}{10}x-e^{1-ctgx}$, $\quad \dfrac{4}{5}x^{-2,1}$;

20. $\sqrt{\dfrac{x-a}{3x+1}}+|\sin x-2|$, $\quad \dfrac{1}{4}x+\dfrac{7}{5}x^3-2e^{x-3}$, $\quad \dfrac{7}{10}x^{4,2}$;

21. $\sqrt{\dfrac{x-1}{7x+a}}+|x-tgx|$, $\quad \dfrac{1}{2}x+\dfrac{7}{5}x^3+2e^{-x+5}$, $\quad \dfrac{1}{2}x^{-0,4}$;

22. $\dfrac{\sqrt{3x+5}}{x-a}+\cos|x+5|$, $\quad \dfrac{1}{10}x^2+\dfrac{7}{4}x+7e^{ctg(x-3)}$, $\quad \dfrac{3}{4}x^{-3,11}$;

23. $\dfrac{x-3a}{2+\sqrt{x-3}}-tg|x-2|$, $\quad \dfrac{7}{2}x^3-\dfrac{3}{10}x+e^{ctgx}$, $\quad \dfrac{2}{5}x^{3,8}$;

24. $\dfrac{\sqrt{x+2}}{2-ax}+|\cos x+5|$, $\quad \dfrac{3}{4}x^3+\dfrac{7}{10}x^2+7e^{2-tgx}$, $\quad \dfrac{7}{8}x^{-0,26}$;

25. $\sqrt{\dfrac{x-a}{2x+1}}+|\sin x-3|$, $\quad \dfrac{1}{4}x+\dfrac{11}{5}x^2+e^{tgx-5}$, $\quad \dfrac{1}{5}x^{3,9}$;

26. $\sqrt{\dfrac{5x+a}{x-3}}+\sin|x-5|$, $\quad \dfrac{3}{8}x^3-\dfrac{7}{5}x-6e^{ctgx-1}$, $\quad \dfrac{2}{5}x^{4,9}$;

27. $\dfrac{x-3}{1-\sqrt{x+a}}-5\sin|x|$, $\quad \dfrac{1}{4}x-\dfrac{1}{2}x^3-e^{tgx+7}$, $\quad \dfrac{5}{8}x^{6,2}$;

28. $\sqrt{\dfrac{x-3}{3x+a}}-5\sin|x|$, $\quad \dfrac{1}{2}x^2+\dfrac{8}{5}x-5e^{ctg(x+1)}$, $\quad \dfrac{1}{4}x^{4,6}$;

29. $\sqrt{\dfrac{2x+a}{4x-3}}+|\cos x|$, $\quad \dfrac{3}{4}x^3+\dfrac{9}{5}x+4e^{tg(x-6)}$, $\quad \dfrac{3}{5}x^{3,3}$;

30. $\dfrac{\sqrt{x+2}}{a-3x}+tg|\cos x+3|$, $\quad \dfrac{1}{10}x+\dfrac{5}{4}x^2-e^{2+x}$, $\quad \dfrac{3}{8}x^{-5,4}$.

5. Алгоритми з розгалуженнями

Поняття алгоритму з розгалуженнями. Реалізація алгоритму з розгалуженнями. Цикломатична складність алгоритму. Приклади алгоритмів з розгалуженнями. Складність алгоритмів з розгалуженнями; оптимізація виконання.

5.1. Поняття алгоритму з розгалуженнями

Алгоритм, який містить хоча б одну умову, в результаті перевірки якої здійснюється перехід до одного з можливих кроків, називають *розгалуженим алгоритмом*. На противагу алгоритмам лінійної структури, які на практиці трапляються досить рідко, обчислення, які залежать від деякої умови і проходять по тій чи іншій гілці алгоритму, є досить поширеними. Кожна гілка алгоритму у свою чергу може містити лінійні фрагменти, цикли і нові розгалуження. Завдяки використанню розгалужень реалізується складна логіка алгоритму розв'язання задачі.

Характерною рисою алгоритму з розгалуженнями є те, що при кожному його виконанні деякі команди будуть виконуватися, причому кожна по одному разу, а деякі виконуватися не будуть. Це залежить від результату виконання команди перевірки умови.

5.2. Реалізація алгоритму з розгалуженнями

Для програмування розгалужень у мовах програмування реалізовано умовні оператори (оператори розгалуження) if і оператори множинного вибору (багатоваріантного розгалуження), зокрема, у мові C оператор switch, у мові Pascal — case. Ці оператори є операторами керування. Оператори керування визначають, в якій послідовності виконується програма. Якби їх не було, оператори програми завжди виконувалися б послідовно, в тому порядку, в якому вони записані.

Умовний оператор дає можливість вибрати і виконати одну дію з двох залежно від значення умови. Зразу можна зауважити, що «Оператор if — це стиль мислення програміста» (таке моє переконання). Оператор вибору є уза-

гальненням умовного оператора, він реалізує багатоваріантне розгалуження — дає можливість вибрати і виконати один варіант розв'язання задачі з кількох залежно від значення селектора вибору. Завдяки його використанню логіка програми при багатоваріантному розгалуженні стає більш зрозумілою. Оператор вибору найчастіше використовують для обробки команд з клавіатури, наприклад, при виборі пунктів меню.

Умовний оператор. Умовний оператор мови С має вигляд:

```
if (умова)
  оператор1;
else
  оператор2;
```

структуризацію запису оператора можна подати й по-іншому, якщо оператори короткі:

```
if (умова) оператор1;
else оператор2;
```

якщо оператори дуже короткі, то можна в один рядок записати весь оператор if:

```
if (умова) оператор1; else оператор2;
```

Роблячи відступи структуризації, треба розміщувати оператори так, щоб не вийти далеко вправо, бо може не вистачити місця на екрані чи папері. Головне, щоб програма легко читалася. Але, вибравши якийсь стиль, треба його дотримуватися.

Тут оператор може бути одним оператором, чи кількома операторами (блок операторів), чи взагалі бути відсутнім (порожній оператор). Умова може бути будь-яким виразом (значення нуль інтерпретується як «хибність», а відмінне від нуля значення — як «істина»), але найчастіше вираз є логічним — містить порівняння й логічні операції. Якщо значення умови «істина», то виконується оператор, розміщений після умови, якщо «хибність», то виконується оператор, розміщений після слова else; обидва оператори одночасно ніколи не виконуються.

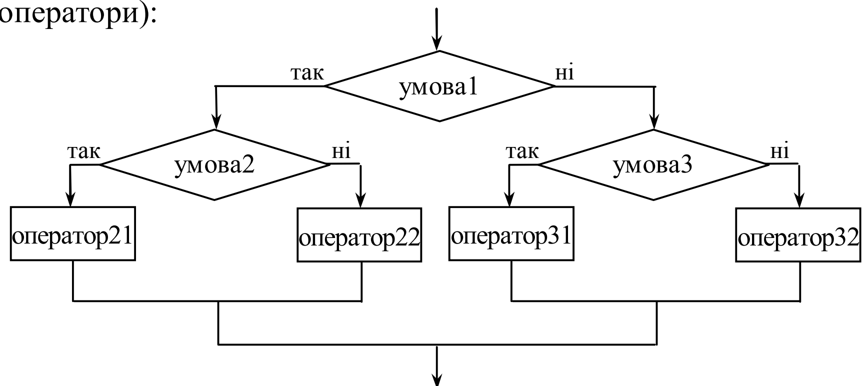
Гілка else в операторі не обов'язкова. Такий вигляд умовного оператора називають неповною формою:

```
if (умова)
  оператор;
```

при цьому, якщо умова набуває значення «хибність», то виконується наступний оператор програми.

Оператор, розміщений після умови, і оператор, розміщений після слова else, також можуть бути умовними операторами (які, в свою чергу, теж можуть містити умовні оператори):

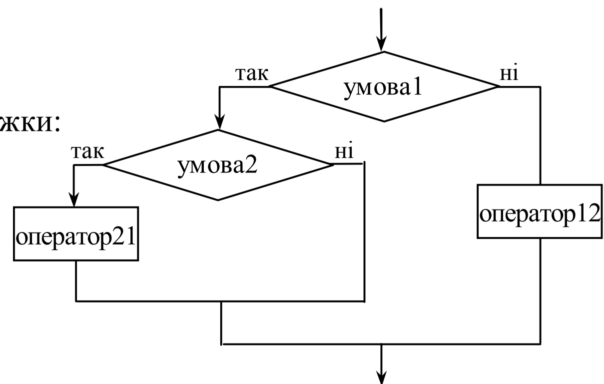
```
if (умова1)
  if (умова2)
    оператор21;
  else
    оператор22;
else
  if (умова3)
    оператор31;
  else
    оператор32;
```



Такі оператори називають вкладеними. У випадку вкладених умовних операторів гілка else завжди відноситься до найближчого if у відповідному блоці.

Якщо зовнішній оператор if (умова1) має гілку else, а після умови містить вкладений оператор if, то цей вкладений оператор обов'язково повинен мати гілку else (якщо при цьому ніякої дії виконувати не треба, то тут потрібен порожній оператор):

```
if (умова1)
  if (умова2)
    оператор21;
  else;
else
  оператор12;
```



або треба відповідно розставити операторні дужки:

```
if (умова1) {
  if (умова2)
    оператор21;
}
else
  оператор12;
```

Приклади умовних операторів:

1)
```
if (b!=0)
    a=a/b;
  else {
    printf("Ділення на нуль");
    ind=0;
  }
```
2)
```
if (a>=b) {a=1; b=0};
```
3)
```
if (a==b)
    if (a>=5)
      b=1;
    else b=0;
  else a=12;
```
4)
```
if (a==b) {
    if (a>=5)
      b=1;
  }
  else b=0;
```

Стандарт C89 допускає 15 рівнів вкладеності умовних операторів, C99 — 127 рівнів. У даний час більшість компіляторів допускають значно більшу кількість рівнів вкладеності. Проте на практиці необхідність у глибині вкладеності, більшій ніж кілька рівнів, виникає досить рідко, оскільки збільшення глибини вкладення швидко заплутує програму і робить її важкою для розуміння.

Тернарна операція. Замість оператора if-else у мові C можна використовувати тернарну операцію ?: . Її загальний вигляд:

вираз1 ? вираз2 : вираз3;

Операція працює так: спочатку обчислюється вираз1, якщо він має значення «істина», то обчислюється вираз2 і його значення присвоюється всьому виразу; якщо вираз1 набуває значення «хибність», то обчислюється вираз3 і всьому виразу присвоюється його значення. У прикладі

```
a = 2;
b = a<=10 ? 0 : 1;
```

змінній b буде присвоєно значення 0. Якби *a* було більшим від 10, то змінній *b* було б присвоєно значення 1. Це можна написати, використовуючи оператор if-else, так:

```
a=2;
if(a<=10) b=0;
  else b=1;
```

Тернарну операцію можна також подати як окремий оператор (при цьому обчислене значення пропадає):

```
int n=15;
n<10 ? printf("менше"): printf("більше");
```

Після виконання даного оператора (тернарної операції) на екран буде виведено результат `більше`. Тут після першого виклику функції `printf` стоїть двокрапка і нема крапки з комою — виконується не оператор функції (який записується окремим оператором), а вираз, який містить тільки звернення до функції. Ця операція еквівалентна операторові:

```
if (n<=10) printf("менше");
else printf("більше");
```

І в операторі if, і в тернарній операції умовою не обов'язково має бути логічний вираз — може бути навіть дійсне число: якщо воно 0, то інтерпретується як «хибність», якщо відмінне від 0, то як «істина»:

```
double r=0;
r ? printf("істина"): printf("хибність");
```

оператор виводить на екран результат «`хибність`». Він еквівалентний умовному операторові:

```
if (r) printf("істина");
else printf("хибність");
```

Зауважимо, що в цьому випадку запис `r!=0` (чи `r==0`) є надлишковим. Такий запис може знизити ефективність обчислень і тому вважається ознакою поганого стилю програмування.

Оператор вибору. У мові С узагальненням умовного оператора, який дає можливість вибрати одну з двох дій залежно від значення виразу, є оператор багатоваріантного розгалуження switch (він є більш ефективним, ніж послідовності умовних операторів). Оператор вибору має вигляд:

switch (вираз) {
case константа1: послідовність операторів1 break;
case константа2: послідовність операторів2 break;
...
case константаN: послідовність операторівN break;
default: послідовність операторів / не обов'язкова гілка */*
}

Значенням керуючого виразу (селектора вибору) має бути значення цілого чи символьного типу, константа — теж ціле число чи символ. Це означає, що значенням виразу не може бути значення з плаваючою точкою (в операторі if може).

Оператор switch послідовно порівнює значення виразу зі списком цілих чисел чи символьних констант. Якщо значення виразу збігається з якоюсь із констант із певної гілки case, то керування передається на відповідну мітку case і виконується послідовність операторів до оператора break, після чого керування передається на оператор, розміщений за оператором switch (після фігурної дужки). Якщо в гілці оператор break відсутній, то виконання послідовності операторів продовжується доти, поки не дійде до оператора break (після іншої мітки) або не закінчиться тіло оператора switch. Те, що при відсутності в гілці case оператора break виконуються наступні гілки послідовно одна за одною, дає можливість уникнути зайвого дублювання операторів і підвищити ефективність програми. Гілка default не є обов'язковою, вона виконується в тому випадку, коли значення керуючого виразу не збіглося з жодною константою; якщо гілка відсутня, то при відсутності збігів не виконується жоден оператор в операторі switch. У тому самому операторі switch ніякі дві гілки case не можуть мати однакових констант. Гілки case і default можуть розміщуватися в будь-якому порядку.

Якщо у виразі-селекторі оператора switch є символьні константи, вони автоматично перетворюються до цілого типу за прийнятими в мові С правилам перетворення типів.

Згідно зі стандартом C89, оператор switch може мати як мінімум 257 гілок case, C99 — як мінімум 1023 гілки case.

Оператор switch відрізняється від оператора if тим, що в ньому вираз-селектор перевіряється тільки на рівність з константами в гілках case, тоді як в if перевіряється будь-який вираз на рівність нулю чи не нулю.

Оператори switch можуть бути вкладені один в оден. Гілки case зовнішнього і внутрішнього операторів можуть мати однакові константи, наприклад:

```
switch(x) {
  case 1:
    switch(y) {
      case 0: printf("Ділення на нуль.\n");
              break;
      case 1: abc(x,y);
              break;
    }
    break;
  case 2:
...
```

У мові програмування Pascal є аналогічний оператор case, де виразом-селектором і константами вибору можуть відповідно бути вираз і константи порядкового типу — integer, byte, char, boolean, перераховуваного.

Використання умовних операторів і операторів вибору дає можливість створювати програми, які аналізують різні ситуації і відповідно до цього виконують певну обробку даних. Використання вкладених умовних операторів дає можливість компактно програмувати складну логіку обчислень, а використання оператора вибору — реалізувати вибір однієї з кількох альтернатив.

Не зважаючи на те, що умовний оператор може виконувати багато перевірок, при великій кількості таких операторів код програми може стати досить важким для сприйняття і в ньому через деякий час може заплутатися навіть автор. Тому, де це можливо, прийнятнішим буде використання оператора вибору, який робить логіку програми більш прозорою.

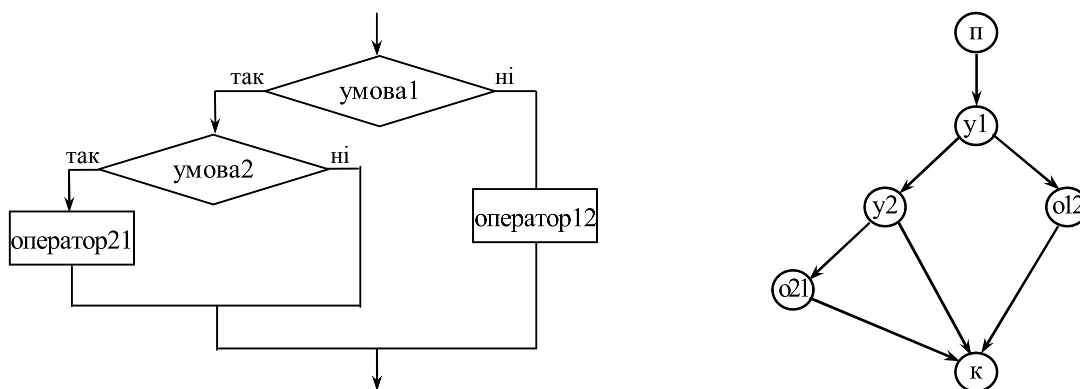
5.3. Цикломатична складність алгоритму

З операторами умови а також умовами циклів пов'язують метрику програмного забезпечення — *цикломатичну складність алгоритму* (ще кажуть: метрика Мак-Кейба; метрику розробив Томас Дж. Мак-Кейб, 1976). Цю метрику можна визначити за поданням алгоритму без його виконання.

Цикломатична складність вказує кількість лінійних незалежних шляхів через програмний код. Якщо код не містить умовних операторів і операторів

циклу, то складність дорівнює 1 (існує один шлях через послідовність команд), якщо є один if, то існує 2 шляхи через код (один через гілку then, інший — через гілку else).

Для того, щоб зрозуміти суть метрики цикломатичної складності можна скористатися *графом керування алгоритму*. Вершинами графа є вершини початку й кінця, виконувані оператори, умовні оператори, оператори циклу (див. розд. 6). При цьому лінійні ділянки алгоритму можна замінити однією вершиною графа. Оператори циклу бажано перетворити в еквівалентні умовні оператори, додавши лічильники повторення циклів. Дуги графа вказують шляхи передачі керування. Наприклад, фрагменту алгоритму, поданому блок-схемою, відповідає граф:



Тут використано позначення вершин графа: п — початок, к — кінець, у1 і у2 — умови, о12 і о21 — виконувані оператори.

Цикломатичне число графа дорівнює найбільшій кількості незалежних (один цикл не є частиною іншого циклу) циклів у графі і обчислюється так. Якщо незв'язний граф має n вершин, m ребер і k компонент зв'язності, то число $m-n+k$ є *цикломатичним числом графа*. Якщо граф зв'язний, тобто має одну компоненту зв'язності, то його цикломатичне число дорівнює $m-n+1$.

У загальному випадку цикломатична складність алгоритму, яка вказує кількість незалежних шляхів через програмний код, обчислюється за формулою $m-n+2k$, де m — кількість ребер, n — кількість вершин, k — кількість компонент зв'язності (модулів, підпрограм). Для одного модуля цикломатична складність дорівнює $m-n+2$, тобто вона на 1 більша від цикломатичного

числа однозв'язного графа керування модуля. Як приклад обчислимо цикломатичне число поданого вище графа і цикломатичну складність поданого цим графом алгоритму. Оскільки $m=7$, $n=6$ і $k=1$, то цикломатичне число має значення $7-6+1=2$ (тобто граф має два незалежних цикли: у1-у2-к-о12-у1, у2-о21-к-у2), а цикломатична складність алгоритму дорівнює $7-6+2=3$ (тобто в алгоритмі існує три лінійні незалежні шляхи: п-у1-у2-о21-к, п-у1-у2-к, п-у1-о12-к).

Цикломатичну складність будь-якого структурованого модуля з однією точкою входу і однією точкою виходу легко порахувати — вона дорівнює кількості точок розгалуження (умовних операторів і умовних циклів) плюс 1. Наприклад, у розглянутому вище фрагменті алгоритму є дві точки розгалуження (умова1 і умова2), отже, цикломатична складність дорівнює $2+1=3$.

Цикломатична складність вказує кількість тестів, потрібних для повного покриття модуля тестуванням (для тестування кожного лінійного незалежного шляху через алгоритм). Вважається, що модуль застосовний при складності 1-10 і нормальне значення показника для модуля становить 5-7; модуль буде тестовним при складності, яка не перевищує 10-15 (Мак-Кейб вважав, що якщо показник перевищить 10, то модулі треба розбивати на дрібніші, але в деяких випадках доцільно це обмеження послабити і дозволити модулі зі складністю до 15); при значенні 11-20 складність є помірною; при 21-50 — ризикованою; складність більше 50 означає дуже високий ризик і практично нетестований код, а отже, нестабільне застосування.

Високе значенням показника свідчить про складність логіки алгоритму і, відповідно, про високі трудовитрати на розробку, тестування і супровід програми. Чим більше шляхів виконання має код, тим більше є варіантів, які треба перевірити, і тому ймовірність помилки є більшою.

Цикломатична складність алгортитму (програми) є мірою складності і використовується для вимірювання якості алгоритмів і програмного забезпечення. За її допомогою можна оцінити той момент, коли алгоритм стає досить складним і код програми потребує розбиття на окремі модулі.

5.4. Приклади алгоритмів з розгалуженнями

Розглянемо кілька прикладів алгоритмів з розгалуженнями.

Приклад 5.4.1. Умовний оператор і тернарна операція дають можливість компактно записувати деякі функції.

а) Знайдемо абсолютну величину числа. Абсолютну величину числа можна знайти або за допомогою умовного оператора:

```
scanf("%lf", &x);
if (x<0) x=-x;
printf("%lf ", x);
```

або за допомогою тернарної операції:

```
scanf("%lf", &x);
x=x<0 ? -x : x;
printf("%lf ", x);
```

б) Знайдемо максимум з двох чисел за допомогою умовного оператора:

```
scanf("%lf%lf", &x, &y);
if (x<y) max=y;
else max=x;
printf("max=%lf", max);
```

і за допомогою тернарної операції:

```
scanf("%lf%lf", &x, &y);
max= (x<y) ? y : x;
printf("max=%lf", max);
```

в) Знайдемо максимум і мінімум з двох чисел за допомогою умовного оператора:

```
scanf("%lf%lf", &x, &y);
if (x<y) {min=x; max=y;}
else {min=y; max=x;}
printf("min=%lf, max=%lf", min, max);
```

і за допомогою тернарної операції:

```
scanf("%lf%lf", &x, &y);
(x<y) ? (min=x, max=y) : (min=y, max=x);
printf("min=%lf, max=%lf", min, max);
```

Приклад 5.4.2. Складну логіку обчислень можна компактно запрограмувати, використовуючи вкладені умовні оператори.

а) Запрограмуємо функцію знака

$$sign(x) = \begin{cases} -1, & \text{якщо } x < 0; \\ 0, & \text{якщо } x = 0; \\ 1, & \text{якщо } x > 0, \end{cases}$$

використовуючи вкладені умови:

```
  scanf("%lf", &x);
  if (x<0) sign=-1;
  else
    if (x) sign=1;
    else sign=0;
  printf("sign=%d", sign);
```

б) За трьома заданими числами дослідимо, чи може існувати трикутник з такими сторонами, і якщо так, то визначимо, який він: різнобічний, рівнобедрений чи рівнобічний.

Подамо не деталізований словесний опис алгоритму розв'язування задачі:

К1. З'ясувати, чи існує трикутник.

К2. Перевірити, чи жодна зі сторін не дорівнює іншій. Якщо це так, то трикутник різнобічний і програма закінчить роботу. Якщо це не так, то програма перейде до наступної перевірки.

К3. Перевірити, чи всі сторони рівні. Якщо це так, то трикутник рівнобічний. Якщо це не так, то залишається тільки один варіант — трикутник рівнобедрений.

Програмно алгоритм можна реалізувати так:

```
#include <stdio.h>
#include <stdlib.h>
int main() { // Визначення типу трикутника
  int a, b, c;   // довжини сторін трикутника
  system("chcp 1251 & cls");
  printf("Визначення типу трикутника\n");
  printf("Введіть a, b, c \n\t");
```

```
  scanf("%d %d %d",&a,&b,&c);
  if (a+b <= c || a+c <= b || b+c <= a)
    printf("Трикутник не існує");
  else
    if (a != b && a != c && b != c)
      printf("Різнобічний");
    else
      if (a == b && b == c)
        printf("Рівнобічний");
      else
        printf("Рівнобедрений");
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Визначення типу трикутника
Введіть a, b, c
        1 2 1
Трикутник не існує
```

```
Визначення типу трикутника
Введіть a, b, c
        3 4 5
Різнобічний
```

```
Визначення типу трикутника
Введіть a, b, c
        2 2 2
Рівнобічний
```

```
Визначення типу трикутника
Введіть a, b, c
        2 3 2
Рівнобедрений
```

в) Розв'яжемо квадратне рівняння $ax^2 + bx + c = 0$, врахувавши, всі можливі випадки — програма при будь-яких значеннях *a*, *b*, *c* повинна видати результат, тобто коректно працювати.

Проаналізуємо всі можливі випадки:

1) $a = 0$ і $b = 0$ — рівняння вироджене;

2) $a = 0$ і $b \neq 0$ — лінійне рівняння має розв'язок $x = -c/b$;

3) $a \neq 0$ і $c = 0$ — рівняння має розв'язки $x_1 = -b/a$, $x_2 = 0$;

4) у решті випадків рівняння матиме вигляд $ax^2 + bx + c = 0$ або $ax^2 + c = 0$ і для обчислення коренів можна скористатися формулою:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}, \text{ де } D = b^2 - 4ac.$$

Введемо позначення $re = \frac{-b}{2a}$, $im = \frac{\sqrt{|D|}}{2a}$.

Розглянемо випадки:

а) якщо $D \geq 0$, то $x_1 = re - im$, $x_2 = re + im$;

б) якщо $D < 0$, то $x_1 = re - i \cdot im$, $x_2 = re + i \cdot im$, де $i = \sqrt{-1}$ (уявна 1).

Проаналізовані можливі випадки дають алгоритм розв'язання задачі. При написанні програми алгоритм можна дещо оптимізувати, виносячи з першої і другої умов спільне порівняння $a = 0$ (застосувати закон дистрибутивності). Беручи за основу розглянуті випадки і використовуючи вкладені умовні оператори, можна запрограмувати алгоритм:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/* Коректний алгоритм розв'язування квадратного рівняння ax^2+bx+c=0 */
int main() {
  double a, b, c;     // коефіцієнти рівняння
  double D, re, im;  // дискримінант, дійсна й уявна частини коренів
  double x1, x2;      // корені рівняння
  /* задання коефіцієнтів квадратного рівняння */
  system("chcp 1251 & cls");
  printf("Розв'язування квадратного рівняння ax^2+bx+c=0\n");
  printf("  Введіть коефіцієнти рівняння a, b, c:  ");
  scanf("%lf%lf%lf",&a,&b,&c);
  /* Розв'язування квадратного рівняння з врахуванням
     усіх можливих випадків */
  if (a==0)   //рівняння не квадратне
    if (b==0)   // вироджене рівняння c=0
      printf("Рівняння вироджене");
    else   // лінійне рівняння bx+c=0
      printf("Єдиний корінь рівняння: x=%0.3f", -c/b);
  else   // квадратне рівняння
    if (c==0)   // рівняння ax^2+bx=0 - один із коренів дорівнює 0
      printf("Дійсні корені рівняння: x1=%0.3f, x2=%0.3f", -b/a, 0.);
    else {   // повне рівняння ax^2+bx+c=0
      re=-b/(2*a);             //дійсна частина кореня
      D=b*b-4*a*c;             // дискримінант рівняння
      im=sqrt(fabs(D))/(2*a);  // уявна частина кореня
```

```
      if (D>=0)   // рівняння має дійсні корені
        printf("Дійсні корені рівняння:\n\
          x1=%0.3f, x2=%0.3f", re-im, re+im);
      else   // рівняння має комплексні корені
        printf("Комплексні корені рівняння:\n\
          x1=%0.3f-i*%0.3f, x2=%0.3f+i*%0.3f",re, im, re, im);
    }
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  0 0 3
Рівняння вироджене
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  0 3 5
Єдиний корінь рівняння: x=-1.667
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  3 5 0
Дійсні корені рівняння: x1=-1.667, x2=0.000
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  -1 0 0
Дійсні корені рівняння: x1=0.000, x2=0.000
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  2 -5 2
Дійсні корені рівняння:
          x1=0.500, x2=2.000
```

```
Розв'язування квадратного рівняння ax^2+bx+c=0
  Введіть коефіцієнти рівняння a, b, c:  5 4 1
Комплексні корені рівняння:
          x1=-0.400-i*0.200, x2=-0.400+i*0.200
```

Приклад 5.4.3. Використання оператора вибору дає можливість реалізувати вибір однієї з кількох альтернатив і робить логіку програми більш прозорою.

а) Програма за введеним числом від 0 до 99 виводить фразу «Користувачеві X років», замінюючи відповідно до контексту слово «років» на «рік» або «роки»:

```
#include <stdio.h>
#include <stdlib.h>
/* Виведення кількості років */
int main() {
  int k;    // кількість років
  system("chcp 1251 & cls");
  printf("Введіть кількість повних років ");
  scanf("%d",&k);
  printf("Користувачеві %d ", k);
  if (k>=11 && k<=19)
    printf("років");
  else
    switch (k%10) {
      case 1:
        printf("рік"); break;
      case 2: case 3: case 4:
         printf("роки"); break;
      default:
        printf("років");
  }
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Введіть кількість повних років 51
Користувачеві 51 рік
```

```
Введіть кількість повних років 32
Користувачеві 32 роки
```

```
Введіть кількість повних років 0
Користувачеві 0 років
```

б) Програма виводить на екран інформацію про те, яка дата раніша, а яка пізніша, якщо користувач вводить дві дати у форматі дд.мм.рррр. Дні, місяці і роки — цілочисельні дані.

```
#include <stdio.h>
#include <stdlib.h>
/* Порівняння двох дат */
int main() {
  int d1,d2,m1,m2,y1,y2;
  int ind;    // номер пізнішої дати
```

```
  system("chcp 1251 & cls");
  printf("Введіть першу дату: dd.mm.yyyy    ");
  // Крапки в форматі вказують, що вхідна інформація їх містить
  scanf("%d.%d.%d",&d1,&m1,&y1);
  printf("Введіть другу дату: dd.mm.yyyy    ");
  scanf("%d.%d.%d",&d2,&m2,&y2);
  if (y1 > y2)
    ind=1;
  else
    if (y1<y2) ind = 2;
    else   // рівні роки
      if (m1>m2) ind=1;
      else
        if (m1<m2) ind=2;
        else   // рівні роки і місяці
          if (d1>d2) ind=1;
          else
            if (d1<d2) ind=2;
            else
              ind=0;   // дати збігаються
  switch (ind) {
    case 0: printf("Дати однакові"); break;
    case 1: printf("Перша дата пізніша"); break;
    case 2: printf("Друга дата пізніша");
  }
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
Введіть першу дату: dd.mm.yyyy    30.05.1959
Введіть другу дату: dd.mm.yyyy    26.10.1963
Друга дата пізніша
```

```
Введіть першу дату: dd.mm.yyyy    21.11.2020
Введіть другу дату: dd.mm.yyyy    28.10.2020
Перша дата пізніша
```

```
Введіть першу дату: dd.mm.yyyy    30.10.2020
Введіть другу дату: dd.mm.yyyy    30.10.2020
Дати однакові
```

5.5. Складність алгоритмів з розгалуженнями. Оптимізація виконання

Складність алгоритмів з розгалуженнями визначають складністю обчислення умов і складністю обчислень, які виконуються на гілках. На відміну від лінійних алгоритмів час виконання при розгалуженні залежить від того, яка гілка буде виконуватися. Тому тут використовують поняття складності алгоритму в гіршому випадку і поняття середньої складності алгоритму.

При визначенні складності в найгіршому випадку треба взяти гілку розгалуження з найбільшим часом виконання. Для кожного окремого оператора розгалуження (має дві гілки) або вибору (має *n* гілок) складність за часом в гіршому випадку можна обчислити за формулою:

$$T = t_{\text{у}} + \max(t_{\text{г}1},...,t_{\text{г}n}),$$

а складність за часом у середньому залежить від ймовірностей $p_{\text{г}i}$ виконання певних гілок:

$$T = t_{\text{у}} + p_{\text{г}1}t_{\text{г}1} + ... + p_{\text{г}n}t_{\text{г}n}, \quad \text{де} \sum_{i=1}^{n} p_{\text{г}i} = 1.$$

Оскільки умовою, як правило, є логічний вираз, то загальні прийоми оптимізації виразів застосовують і до логічних виразів. При цьому треба враховувати, що, як правило (але все залежить від процесора — див. табл. 4.5), час t_l виконання логічних операцій &&, | |, !, ==, != менший від часу виконання адитивних операцій, а час виконання порівнянь >, >=, <, <= рівний часові виконання адитивних операцій. Таким чином:

$$t_l < t_a < t_m << t_f.$$

Приклад 5.5.1. Для перевірки того, чи хоч би одне із чисел *a* і *b* дорівнює нулю можна, скористатися одним з логічних виразів:

`a==0 || b==0` або `a*b==0`.

Перший вираз має часову складність $t_1 = 3t_l$, другий — $t_2 = t_m + t_l$. При цьому $t_1 << t_2$. Отже, для перевірки умови треба вибрати перший вираз `a==0 || b==0`.

Приклад 5.5.2. Замість використання оператора

```
if (a>0) ind=1; else ind=0;
```

можна скористатися оператором

```
ind=(a>0);
```

Приклад 5.5.3. Замість використання вкладених умовних операторів

```
if (x>0)
  if (y<0) nom=4;
```

можна записати оператор зі складним логічним виразом

```
if (x>0 && y<0) nom=4;
```

Приклад 5.5.4. При програмуванні функції, заданої логічно,

$$y = \begin{cases} \cos x & \text{при } x < 3; \\ ax + b & \text{при } x \in [3;5) \end{cases}$$

у фрагменті

```
...
  if (x<3) y=cos(x);
  else
    if (x>=3 && x<5) y=a*x+b;
    else
     if (x>=5) printf("Значення x задано поза проміжком");
...
```

можна порівняння в першому вкладеному умовному операторі спростити (оскільки значення x не менше від 3, то воно буде більшим або рівним 3), а перевірка у другому вкладеному умовному операторі взагалі зайва (якщо значення x не потрапило в перший проміжок $(-\infty;3)$ і не потрапило в другий проміжок $[3;5)$, то воно потрапило в проміжок $[5;+\infty)$; отже, перевірка x>=5 тут не потрібна).

Матимемо замість попереднього такий оптимізований фрагмент програми:

```
...
  if (x<3) y=cos(x);
  else
    if (x<5) y=a*x+b;
    else printf("Значення x задано поза проміжком");
...
```

При програмуванні логічних операторів, якщо це можливо, треба для спрощення обчислень застосовувати закони алгебри логіки, зокрема, закон дистрибутивності.

Приклад 5.5.4. При програмуванні заданої логічно функції

$$y=\begin{cases} \dots \\ \ln(ax+b) \quad \text{при } x\in[3;5) \end{cases}$$

замість послідовних логічних операторів

```
...
  if (x>=3 && x<5 && a*x+b>0) y=ln(a*x+b);
  if (x>=3 && x<5 && a*x+b<=0)
    printf("Підлогарифмічний вираз не додатний");
...
```

застосувавши закон дистрибутивності і використавши вкладені логічні оператори, можна записати:

```
...
  if (x>=3 && x<5)
    if (a*x+b>0) y=ln(a*x+b);
    else printf("Підлогарифмічний вираз не додатний");
...
```

У початковому фрагменті програми виконувалося 6 порівнянь і 4 логічні операції, а в оптимізованому — всього 3 порівняння і 1 логічна операція; при цьому ті самі умови повторно не перевіряються.

Крім того, в одержаному фрагменті програми значення `a*x+b` обчислюється двічі. Для оптимізації алгоритму можна використати робочу змінну, в яку запам'ятати це значення. Матимемо такий фрагмент програми:

```
...
  if (x>=3 && x<5) {
    r=a*x+b;
    if (r>0) y=ln(r);
    else printf("Підлогарифмічний вираз не додатний");
  }
...
```

Також, як було сказано раніше (див. п. 4.2), при записові логічних виразів треба враховувати, що якщо операціями в логічному виразі є кон'юнкції, то вираз обчислюється зліва направо до першого значення «хибність»; якщо опе-

раціями в логічному виразі є диз'юнкції, то вираз обчислюється до першого значення «істина». У зв'язку з цим для збільшення швидкості обчислень першими в логічному виразі треба записувати умови, які простіше обчислюються, або частіше трапляються, або від яких залежать наступні умови.

Крім того, порівняно з використанням для програмування множинного вибору вкладених умовних операторів if, оператор вибору switch працює набагато швидше за рахунок того, що компілятор генерує таблицю переходів. Таким чином, під час виконання алгоритму замість перевірки того, яка умова виконується, з таблиці переходів вибирається випадок, який відповідає значенню селектора вибору. Крім того, використання оператора множинного вибору робить текст програми читабельнішим, такий текст набагато легше модифікувати, ніж при використанні вкладених умовних операторів.

Слід також зауважити, що хоч при програмуванні складної логіки використання вкладених умовних операторів і зменшує часову складність алгоритмів, проте алгоритми з використанням послідовних умовних операторів мають просту структуру, вони легкі для сприйняття й реалізації.

Приклад 5.5.5. Для обчислення значення функції

$$f(x)=\begin{cases}\dfrac{9}{ax} & \text{при } x\in[1,3),\\ \left|ax^2+x+b\right| & \text{при } x=3,\end{cases}$$

заданої логічно, подамо два алгоритми — з послідовною і вкладеною перевіркою умов.

Програмна реалізація алгоритму з використанням послідовних умовних операторів (загальні вимоги до алгоритму подано в прикладі 1.4.4):

```
...
  /* Обчислення значення функції */
  ind=0;    // індикатор друку значення функції
  /* Перший проміжок. ОДЗ не порушено */
  if (x>=1 && x<3 && a!=0) {
    y=9/(a*x);
    ind=1;
  }
```

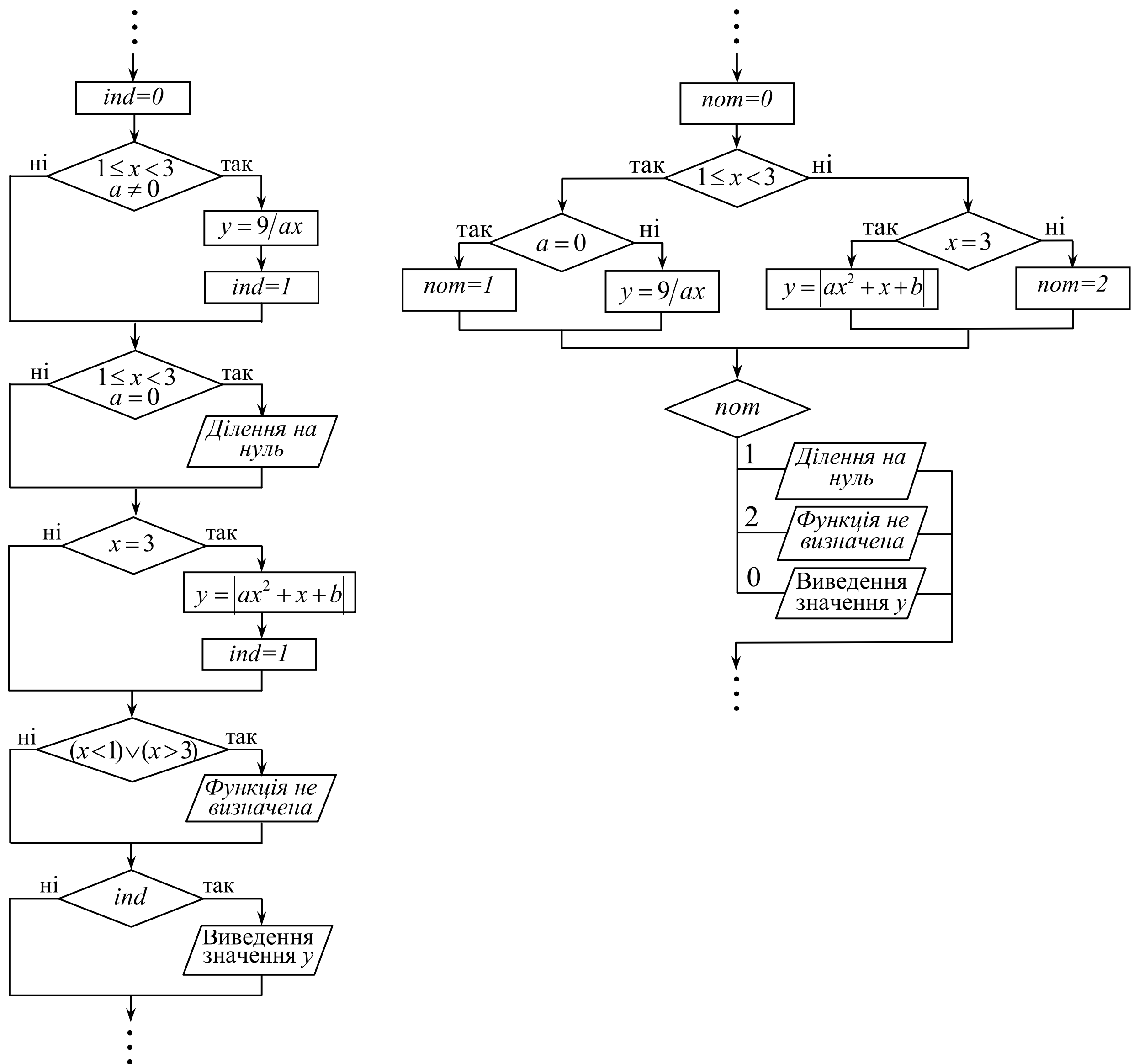
```
  /* Перший проміжок. ОДЗ порушено */
  if (x>=1 && x<3 && a==0)
    printf("Ділення на нуль");
  /* Другий проміжок. Особливостей нема */
  if (x==3) {
    y=fabs(a*x*x+x+b);
    ind=1;
  }
  /* Значення x задано поза проміжком */
  if (x<1 || x>3)
    printf("Функція не визначена");
  /* Друк результату обчислення */
  if (ind)
    printf("f(%4.2f)=%6.3f", x, y);
...
```

Програмна реалізація алгоритму з використанням вкладених умовних операторів, а також оператора вибору для виведення результату (повідомлення про помилку чи обчисленого значення) в кінці програми:

```
  ...
  /* Обчислення значення функції */
  nom=0;   // номер повідомлення; 0 – вивід значення функції
  /* Перший проміжок - може бути порушення ОДЗ */
  if (x>=1 && x<3)
    /* Перший проміжок - ОДЗ порушено */
    if (a==0) nom=1;   // ділення на 0
    /* Перший проміжок - ОДЗ не порушено */
    else
      y=9/(a*x);
  else
    /* Другий проміжок (точка x=3) - особливостей нема */
    if (x==3)
      y=fabs(a*x*x+x+b);
    /* Значення x задано поза проміжком */
```

```
    else
      nom=2;  // функція не визначена - x поза проміжком
  /* Друк результату обчислення */
  switch (nom) {
    case 1: printf("Ділення на 0"); break;
    case 2: printf("Функція не визначена"); break;
    default: printf("f(%4.2f)=%6.3f", x, y);
  }
...
```

Відповідні фрагменти блок-схем такі:



Алгоритм з використанням послідовних умовних операторів має простішу структуру і легший для сприйняття й реалізації, але він менш ефективний, ніж алгоритм з вкладеними умовами. При обчисленні значення з вико-

ристанням послідовних логічних операторів виконується 9 порівнянь і 5 логічних операцій, а при використанні вкладених логічних операторів виконується 4 порівняння і 1 логічна операція. Використання оператора вибору дає можливість відокремити блок обчислень від блоку видачі результатів, що структуризує логіку програми.

Запитання та завдання для самоконтролю

1. Схарактеризуйте алгоритми з розгалуженнями. Чим вони відрізняються від лінійних алгоритмів? У яких випадках використовують алгоритми з розгалуженнями?

2. Які оператори використовують для програмування розгалужень? Схарактеризуйте різні форми умовного оператора і їхнє використання. Схарактеризуйте оператор вибору. Коли його доцільно використовувати?

3. На що вказує цикломатична складність алгоритму? Як для алгоритму побудувати граф керування? Як цикломатична складність алгоритму пов'язана з цикломатичним числом графа керування? Які значення цикломатичної складності вважають нормальними, помірними, а які свідчать про практичну нетестованість алгоритму?

4. Від чого залежить складність алгоритму з розгалуженнями? Як визначити складність алгоритму в гіршому випадку і середню складність алгоритму? Вкажіть прийоми оптимізації алгоритмів з розгалуженнями.

Практичне завдання № 6
«Розробка і реалізація алгоритмів розгалужених процесів з послідовною перевіркою умов»

Мета виконання завдання: навчитися складати й програмувати алгоритми з послідовною перевіркою умов.

Завдання. Розробити алгоритм з послідовною перевіркою умов (неповна форма умовного оператора) для обчислення значень заданої логічною залежністю функції $f(x)$ при довільних значеннях параметрів a і b і незалежної змінної x. Видачу повідомлень про помилки здійснювати зразу ж при їхньому

виявленні, а видачу результату обчислення значення функції $f(x)$ — у кінці алгоритму. В алгоритмі передбачити перевірку, чи немає ділення на нуль, чи підкореневий вираз невід'ємний, чи аргумент функції логарифма набуває додатних значень тощо. Вважається, що функція визначена на заданому проміжку, а поза ним не визначена.

Алгоритм подати за допомогою блок-схеми і програмно. Текст програми структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з п'ятьма різними наборами вхідних даних, які передбачають виконання різних гілок алгоритму.

Вказівки. Як зразок використати алгоритм з прикладу 5.5.5, а також програму з прикладу 1.4.5.

Варіанти завдань

1. $f(x)=\begin{cases}\sin x & \text{при } x\in[-1,6),\\ 3x^2+bx & \text{при } x=6,\\ \ln(ax-3b) & \text{при } x\in(6,8).\end{cases}$

2. $f(x)=\begin{cases}\cos x & \text{при } x\in[0,5),\\ \left(\sqrt{ax-b}\right)^{-1} & \text{при } x=5,\\ bx^2+3x-2 & \text{при } x\in(5,8).\end{cases}$

3. $f(x)=\begin{cases}\sqrt{3x^2-a} & \text{при } x\in(-3,3),\\ -bx+3 & \text{при } x=3,\\ \cos(x-4) & \text{при } x\in(3,8).\end{cases}$

4. $f(x)=\begin{cases}\sqrt{bx+3} & \text{при } x\in(-2,10),\\ \left|x^2-ax+7\right| & \text{при } x=10,\\ \cos(x-1) & \text{при } x\in(10,+\infty).\end{cases}$

5. $f(x)=\begin{cases}\sqrt{x^3-a} & \text{при } x\in(-\infty,7),\\ 3x^2-x+b & \text{при } x=7,\\ \sin 2x & \text{при } x\in(7,8).\end{cases}$

6. $f(x)=\begin{cases}\ln(ax+3) & \text{при } x\in(-2,5),\\ x^2-bx+7 & \text{при } x=5,\\ \cos(x+1) & \text{при } x\in(5,+\infty).\end{cases}$

7. $f(x)=\begin{cases}\sin 3x & \text{при } x\in[0,5),\\ 2e^{ax-1}+1 & \text{при } x=5,\\ (bx-a)^{-1} & \text{при } x\in(5,12].\end{cases}$

8. $f(x)=\begin{cases}bx^2+x-6 & \text{при } x\in[0,1),\\ \left(\sqrt{ax+b}\right)^{-1} & \text{при } x=1,\\ \cos x & \text{при } x\in(1,9).\end{cases}$

9. $f(x)=\begin{cases}\sin(x+1) & \text{при } x\in[0,1),\\ ax^2+2bx-4 & \text{при } x\in[1,7),\\ (ax+b)^{-1} & \text{при } x=7.\end{cases}$

10. $f(x)=\begin{cases}\sin x & \text{при } x\in[-1,3),\\ \sqrt{ax-2} & \text{при } x=3,\\ bx+a & \text{при } x\in(3,5).\end{cases}$

11. $f(x)=\begin{cases}\cos x & \text{при } x\in(-\infty,4),\\ x^3-bx+1 & \text{при } x=4,\\ \sqrt{ax} & \text{при } x\in(4,7).\end{cases}$

12. $f(x)=\begin{cases}\cos x & \text{при } x\in[0,4),\\ 3x^2+bx & \text{при } x=4,\\ \ln(ax+b) & \text{при } x\in(4,9).\end{cases}$

13. $f(x)=\begin{cases}\sin x & \text{при } x\in[0,5),\\ 3e^{ax}+1 & \text{при } x=5,\\ (bx-10)^{-1} & \text{при } x\in(5,11).\end{cases}$

14. $f(x)=\begin{cases}|ax+b| & \text{при } x\in(-\infty,8)\\ 5\sqrt{bx+3} & \text{при } x=8,\\ \cos x & \text{при } x\in(8,10].\end{cases}$

15. $f(x)=\begin{cases}\cos x & \text{при } x\in(-2,4],\\ \sqrt{ax+b} & \text{при } x\in(4,9),\\ x^3-bx+3 & \text{при } x=9.\end{cases}$

16. $f(x)=\begin{cases}\sin x & \text{при } x\in(-\infty,-2),\\ 2e^{ax}-1 & \text{при } x=-2,\\ (bx+2a)^{-2} & \text{при } x\in(-2,8).\end{cases}$

17. $f(x)=\begin{cases}\sqrt{x^3+a} & \text{при } x\in(-\infty,7),\\ -bx-3 & \text{при } x=7,\\ \sin x & \text{при } x\in(7,10).\end{cases}$

18. $f(x)=\begin{cases}\sqrt{5x^2-a} & \text{при } x\in(-\infty,3),\\ bx+8 & \text{при } x=3,\\ \cos x & \text{при } x\in(3,11).\end{cases}$

19. $f(x)=\begin{cases}\sin x & \text{при } x\in[-1,6),\\ 3x^2+bx-3 & \text{при } x=6,\\ \ln(bx+a) & \text{при } x\in(6,+\infty),\end{cases}$

20. $f(x)=\begin{cases}|ax+x^2| & \text{при } x\in(-\infty,4),\\ \sqrt{bx-1}+3 & \text{при } x=4,\\ \cos x & \text{при } x\in(4,10].\end{cases}$

21. $f(x)=\begin{cases}\sin x & \text{при } x\in[0,2),\\ 2e^{ax} & \text{при } x=2,\\ (bx+a)^{-1} & \text{при } x\in(2,8).\end{cases}$

22. $f(x)=\begin{cases}2x+7 & \text{при } x\in[0,5),\\ 3e^{ax} & \text{при } x=5,\\ \ln(bx+a) & \text{при } x\in(5,8).\end{cases}$

23. $f(x)=\begin{cases}\sin x & \text{при } x\in[0,7),\\ 3e^{ax+1}-1 & \text{при } x=7,\\ (bx-a)^{-1} & \text{при } x\in(7,11).\end{cases}$

24. $f(x)=\begin{cases}\sqrt{x^2-a} & \text{при } x\in(-3,3),\\ 5x^3-bx+3 & \text{при } x=3,\\ \cos x & \text{при } x\in(3,8).\end{cases}$

25. $f(x)=\begin{cases}\sqrt{ax-3} & \text{при } x\in(-8,3),\\ |x^2-bx+7| & \text{при } x=3,\\ \cos(x+1) & \text{при } x\in(3,+\infty).\end{cases}$

26. $f(x)=\begin{cases}\sqrt{ax+2} & \text{при } x\in(-3,2),\\ |\cos 2x| & \text{при } x=2,\\ 3x^2+bx+1 & \text{при } x\in(2,+\infty).\end{cases}$

27. $f(x)=\begin{cases}\sqrt{x+b} & \text{при } x\in[0,11),\\ ax^2-3x+14 & \text{при } x\in[11,15),\\ e^{-ax} & \text{при } x=15.\end{cases}$

28. $f(x)=\begin{cases}\cos x & \text{при } x\in[0,9),\\ 3e^{ax}+1 & \text{при } x=9,\\ \ln(bx-1) & \text{при } x\in(9,11).\end{cases}$

29. $f(x)=\begin{cases}|ax+7| & \text{при } x\in(-\infty,9),\\ 5\sqrt{bx+1} & \text{при } x\in[9,15),\\ \sin x & \text{при } x=15.\end{cases}$

30. $f(x)=\begin{cases}\sin 5x & \text{при } x\in[-0,7),\\ x^2-bx & \text{при } x=7,\\ \ln(ax-8) & \text{при } x\in(7,10).\end{cases}$

Практичне завдання № 7
«Розробка і реалізація алгоритмів розгалужених процесів з вкладеною перевіркою умов»

Мета виконання завдання: навчитися складати й програмувати алгоритми з розгалуженнями, використовуючи вкладені умовні оператори і оператор вибору.

Завдання. Розробити два алгоритми з вкладеною перевіркою умов (порівняння в умовах не повторювати) для обчислення значень заданої логічною залежністю функції $f(x)$ при довільних значеннях параметрів a і b і незалежної змінної x. У першому алгоритмі передбачити видачу повідомлень про помилки зразу ж при їхньому виявленні, а видачу результату обчислення значення функції $f(x)$ — у кінці алгоритму. Відповідно до другого алгоритму видача повідомлень про помилки і видача значення функції $f(x)$ мають виконуватися після проведення обчислень. В алгоритмах передбачити перевірку, чи немає ділення на нуль, чи підкореневий вираз невід'ємний, чи аргумент функції логарифма набуває додатних значень тощо. Вважається, що функція визначена на заданому проміжку, а поза ним не визначена.

Алгоритми подати за допомогою блок-схем і програмно. Тексти програм структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з п'ятьма різними наборами вхідних даних, які передбачають виконання різних гілок алгоритму.

Знайти і порівняти часові складності в гіршому випадку алгоритму з практичного завдання № 6 і першого алгоритму з цього практичного завдання.

Вказівки. Як зразок використати алгоритм з прикладу 5.5.5, а також програму з прикладу 1.4.5. При обчисленні часової складності використати інформацію з пункту 5.5 і, зокрема, з прикладу 5.5.1.

Функція $f(x)$ та сама, що і в практичному завданні № 6.

6. Алгоритми з циклами

Поняття алгоритму з циклами. Реалізація і використання алгоритмів з циклами. Приклади алгоритмів з циклами. Складність і оптимізація циклічних алгоритмів.

6.1. Поняття алгоритму з циклами

Досить часто на практиці виникають задачі, при розв'язуванні яких одну дію або деяку послідовність дій треба повторити кілька разів. Кількість повторень може бути визначена наперед, а також може визначатися умовою, яка залежить від розрахованих у процесі обчислення значень або від введеної користувачем інформації. Щоб виконати певний оператор або групу операторів повторно, використовують цикли.

Фрагмент алгоритму з командами (командою), які можуть виконуватися більше одного разу залежно від виконання певної умови, називають *циклом* (повторенням, структурою повторення, циклічною структурою). Команду або групу команд, виконання яких повторюється при кожному проходженні циклу (кожній ітерації), називають *тілом циклу*. Алгоритм, який містить цикли, називають *алгоритмом з циклами* (алгоритмом з повтореннями, циклічним алгоритмом, ітераційним алгоритмом).

Розрізняють цикли з наперед відомою і наперед невідомою кількістю проходів (ітерацій). Цикл називають *детермінованим*, якщо кількість повторень тіла циклу наперед відома. Цикл називають *ітераційним*, якщо кількість повторень тіла циклу наперед невідома і залежить від значень параметрів (деяких змінних), які беруть участь в обчисленнях.

При логічних (семантичних) помилках в циклічних алгоритмах може виникнути ситуація, коли тіло циклу виконується раз за разом, а умова припинення циклу не настає. Таку ситуацію називають *зациклюванням*.

У циклічних алгоритмах, крім циклічних блоків, можуть міститися лінійні фрагменти і розгалуження, будь-який цикл також може містити лінійні фрагменти і розгалуження, а також вкладений у своє тіло інший цикл або навіть кілька циклів. Використання циклів дає можливість коротко записати

алгоритм, який здійснює велику кількість дій. Дії, які повторюються в циклі, є однотипними. Використання циклів суттєво скорочує обсяг алгоритму.

Таким чином, характерною рисою алгоритмів з циклами є те, що вони забезпечують виконання довгої послідовності дій, записаних порівняно короткою послідовністю команд.

6.2. Реалізація і використання алгоритмів з циклами

Найпоширенішими конструкціями, які дають можливість описати циклічні процеси, є:

— оператор циклу з передумовою — «доки виконується умова, виконувати дію»;

— оператор циклу з післяумовою — «виконувати дію, доки виконується умова»;

— оператор циклу з параметром (цикл з лічильником), який ще називають арифметичним циклом, — «для параметра циклу від початкового значення до кінцевого значення з кроком виконувати дію».

Цикл з параметром використовують, коли під час розв'язування задачі наперед відома кількість ітерацій (повторень команд). В ітераційному циклі кількість ітерацій наперед невідома, тут задається лише умова виконання циклу, відбувається перевірка цієї умови і залежно від результату перевірки здійснюється або вихід з циклу, або повторення виконання тіла циклу. При цьому в циклі з передумовою перевірка умови відбувається перед виконанням блоку операторів, а в циклі з післяумовою перевірка проводиться після виконання тіла циклу. Особливість цих циклів полягає в тому, що тіло циклу з післяумовою завжди виконується хоч би один раз, а тіло циклу з передумовою чи параметром може не виконатися жодного разу.

Цикл з параметром легко можна подати як цикл з передумовою, перед початком якого параметру присвоюється початкове значення, а умовою виходу з циклу є набуття лічильником кінцевого значення; в тіло циклу при цьому додається оператор зміни параметра на заданий крок:

Цикл з параметром

```
for (i=0; i<n; i++) {
  ... тіло циклу
}
```

Цикл з передумовою

```
i=0;
while (i<n) {
  ... тіло циклу
  i++;
}
```

Проте спеціальні оператори циклу з параметром можуть ефективніше компілюватися і використовувати спеціальні процесорні команди організації циклів.

Швейцарський програміст і теоретик програмування Ніклаус Вірт стверджував, що цикл з параметром є надлишковою конструкцією і його треба виключити з синтаксису мов програмування. У розробленій і реалізованій ним в 1986 році мові програмування Оберон (її основою була мова Pascal) циклу з параметром не було. Проте в мові Оберон-2 цикл for з параметром з'явився для практичної зручності використання.

Той чи інший вид циклів використовують залежно від умови розв'язуваної задачі. Крім того, це також залежить і від особистого вподобання програміста, але в деяких ситуаціях краще використовувати конкретну циклічну структуру. Так, при обробці повідомлень з клавіатури найкраще використовувати цикл з післяумовою. Але цей цикл використовують досить часто і при розв'язуванні інших задач. Цикл з параметром рекомендується використовувати тільки в тих випадках, коли точно відомо, що в процесі виконання параметр циклу має набувати саме певних значень. Така ситуація може виникнути, наприклад, при заповненні таблиць, побудові сітки системи координат тощо.

Цикл з передумовою використовують у тому випадку, коли треба, щоб на початку перевірялася деяка умова, а потім, якщо умова істинна, виконувалися деякі дії. Найчастіше програмісти використовують саме цей вид циклу.

Хоч, трохи подумавши, можна будь-яке завдання розв'язати з використанням будь-якого циклу, вказані три види циклу в мовах програмування введено передусім для зручності.

Організація будь-якого виду циклу потребує встановлення початкових значень тим змінним, які використовуються в циклі.

Приклад 6.2.1. Розглянемо алгоритми обчислення суми $S = \sum_{i=1}^{n} i^2$ з перевіркою умови до і після виконання дій.

а) Умова продовження накопичення суми (виконання циклу) перевіряється до виконання дії:

K1. Ввести n.

K2. S=0; i=1.

K3. Якщо i<=n, то S=S+i^2, i=i+1, перейти на К3.

K4. Вивести S.

Наведений алгоритм можна запрограмувати з використанням оператора циклу з параметром і з використанням оператора циклу з передумовою.

б) Умова продовження накопичення суми (виконання циклу) перевіряється після виконання дії:

K1. Ввести n.

K2. S=0, i=0.

K3. S=S+i^2, i=i+1.

K4. Якщо i<=n, то перейти на К3.

K5. Вивести S.

Цей алгоритм можна запрограмувати з використанням оператора циклу з післяумовою.

Як і розглянуті раніше умовні оператори, цикли можна вкладати один в оден. Це можуть бути цикли різних типів. Вкладені цикли можуть бути як залежними (вкладений цикл при обчисленні значень своїх параметрів використовує значення, змінювані в зовнішньому циклі), так і незалежними. Вкладення циклів застосовують, наприклад, при розв'язуванні таких задач, як задачі на перебір варіантів, алгоритми сортування, обробка двовимірних масивів тощо.

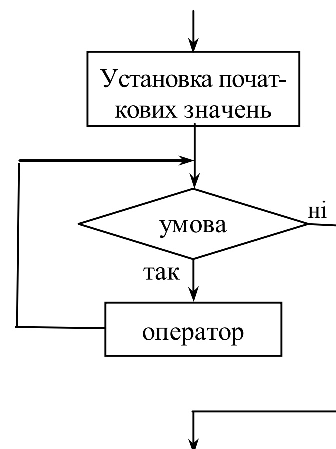
У мові C для програмування циклічних алгоритмів реалізовано всі розглянуті вище види циклів: з передумовою — while, післяумовою — do-while і параметром — for (у мові C оператор циклу з параметром надає програмісто-

ві значно більше можливостей, ніж однойменний оператор в інших мовах, зокрема, в мові Pascal, де крок може бути рівним тільки 1 чи –1, що дуже незручно).

Оператор циклу з передумовою while. Оператор циклу з передумовою має вигляд:

while (вираз) оператор;

Тут оператор утворює тіло циклу. Він може бути порожнім оператором, звичайним простим оператором або блоком операторів (після блоку крапка з комою не ставиться, хо це не є помилкою). Умова в операторі while перевіряється на початку циклу перед кожним його виконанням. Доки вираз набуває істинного (ненульового) значення, цикл виконується. Якщо вираз має значення хибність (нульове), то цикл while закінчує роботу і керування передається оператору, наступному за оператором циклу. Схему виконання циклу з передумовою подано на рисунку.



Оскільки умова в операторі циклу з передумовою перевіряється на початку циклу, то, як правило (хоча в мові С ініціалізація може відбутися і в самому виразі), вона має бути визначена вже перед циклом, тобто має відбутися ініціалізація циклу. Крім того, в середині циклу, як правило, має бути оператор, який впливає на умову, інакше оператор while зациклиться (буде виконуватися нескінченно). До зациклювання також може призвести неправильне задання умови.

Завдяки можливостям мови С щодо побудови виразів, ініціалізація циклу і зміна умови може відбуватися в самій умові. Наприклад, для пропуску символів-роздільників при введенні символьної інформації можна використати такий оператор з порожнім тілом циклу (тут немає ні ініціалізації змінних перед циклом, ні перерахунку значення параметра в тілі циклу):

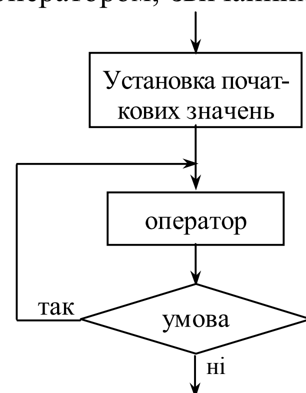
```
while ((c=getchar())==' '||c=='\n'||c=='\t');
```

Тут (а саме у виразі `(c=getchar())` використано правило мови С, що будь-який вираз з операцією присвоювання, взятий в круглі дужки, має значення, рівне тому, яке присвоюється.

Оператор циклу з післяумовою do-while. Оператор циклу з післяумовою має вигляд:

do оператор while (вираз);

Тут оператор утворює тіло циклу. Він може бути порожнім оператором, звичайним простим оператором або блоком операторів. Вираз є умовою виконання циклу.



При виконанні циклу do-while спочатку виконується тіло циклу, а потім перевіряється умова. Як і в операторі while, цикл виконується доти, поки вираз набуває істинного (ненульового) значення; якщо вираз має значення хибність (нульове), то цикл закінчує роботу. Схему виконання циклу з післяумовою подано на рисунку.

При використанні циклу з післяумовою, як і при використанні циклу з передумовою, як правило, треба встановити початкові значення тих змінних, які впливають на умову, а в тілі циклу їхні значення треба змінювати, щоб виконання не зациклилося.

Основною відмінністю між циклами while і do-while є те, що в циклі do-while тіло виконується принаймні один раз. Таким чином навіть, якщо при

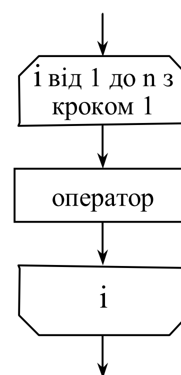
вході в цикл значення виразу хибне, то тіло циклу все одно один раз виконається. Тому при складанні алгоритмів з використанням оператора циклу з післяумовою треба це враховувати.

Цикл do-while аналогічний циклові repeat-until у мові Pascal з тією відмінністю, що в мові C вказується умова продовження циклу, а в мові Pascal — умова його закінчення. Крім того, у Pascal ключові слова repeat і until відіграють роль операторних дужок і тому, якщо тіло циклу утворюють кілька простих операторів, то їх не треба обмежувати словами begin-end. У мові C у випадку, якщо тіло циклу з післяумовою утворюють кілька операторів, то їх обов'язково треба брати у фігурні дужки.

Оператор циклу з параметром. Якщо треба виконати якийсь оператор чи групу операторів кілька разів і кількість повторень не залежить від результату роботи оператора, то найкраще скористатися оператором циклу з параметром. У всіх процедурних мовах програмування цикли for дуже схожі. Проте в мові C цей цикл особливо гнучкий і потужний.

Оператор циклу з параметром має вигляд:

for (вираз1; вираз2; вираз3) оператор;

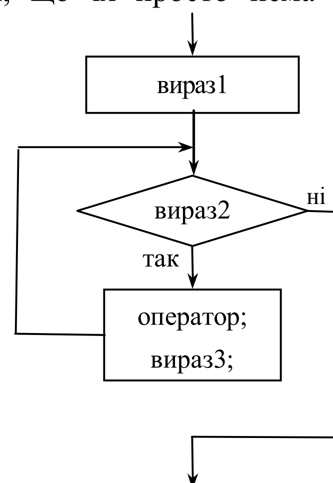


Оператор, який виконується в циклі, є тілом циклу; він може бути або звичайним окремим оператором, або порожнім оператором, або складним оператором (послідовністю операторів, взятих у фігурні дужки).

Усі три вирази, які містяться в дужках після слова for — будь-які вирази, але найчастіше вираз1 і вираз3 — це присвоювання чи виклики функцій, а вираз2 — відношення (порівняння). Кожен вираз може бути складним, тобто містити кілька виразів, розділених комами.

У виразі1, як правило, задаються початкові значення змінних; вираз 2, як правило, є логічним виразом, який задає умову виконання тіла циклу (доки умова істинна, цикл виконується); вираз3 виконується в кінці кожної ітерації циклу і, як правило, тут змінюється значення змінної, заданої в заголовку у виразі1.

Будь-який вираз у заголовку циклу може бути відсутнім, але крапку з комою опускати не можна. Якщо відсутній вираз1 чи вираз3, то вважається, що їх просто нема в конструкції циклу, якщо відсутній вораз2, то вважається, що його значення завжди істинне. Наприклад, for (;;) оператор; є нескінченним циклом, виконання якого може бути перерване оператором break чи ззовні.



Процедура виконання циклу така: спочатку обчислюється (чи виконується) вираз1, потім обчислюється вираз2 і, якщо його значення хибне (нульове), то цикл закінчує роботу і керування передається на наступний після циклу оператор, якщо значення істинне (ненульове), то виконується тіло циклу, потім обчислюється значення виразу3 і керування передається на обчислення виразу2 і т.д. Отже, вираз1 обчислюється всього один раз — перед початком циклу; вираз2 обчислюється перед кожною ітерацією (проходом циклу), а вираз3 — після кожної ітерації.

Оператор циклу з параметром еквівалентний такій послідовності операторів:

вираз1; while (вираз2) {оператор; вираз3;};

Тому оператор циклу з параметром можна подати блок-схемою, аналогічною до блок-схеми оператора циклу з передумовою.

Цикл for може мати велику кількість варіацій. Найбільш традиційне (як і в інших мовах програмування) використання оператора for полягає в тому, що на місці виразу1 стоїть присвоєння змінній-параметру циклу початкового значення, вираз2 є логічним виразом, у якому значення параметра порівнюється з граничним значенням, а вираз3 збільшує чи зменшує значення параметра на деяку величину (крок). Таким чином, найвживаніша форма його використання циклу for така:

for (ініціалізація; умова; приріст) оператор;

При традиційному використанні цикли for у мові C і в Pascal подібні. Але у мові Pascal крок зміни значення параметра може бути тільки 1 чи –1, границю і параметр циклу в тілі циклу змінювати не можна. У мові C крок циклу може бути довільним, він може змінюватися в тілі циклу, можуть у тілі циклу змінюватися значення параметра і граничне значення, значення параметра циклу після виходу з циклу завжди визначене. Оскільки в операторі for мови C усі три вирази можуть бути будь-якими, то організація for-циклів не обмежується тільки випадком арифметичної прогресії. Проте включати в заголовок циклу обчислення, які не мають відношення до ініціалізації і збільшення чи зменшення, вважається поганим стилем.

Алгоритми, подані в прикладі 6.2.1 словесно, подамо фрагментами програм з використанням операторів циклів мови C.

Приклад 6.2.2. Обчислення суми $S = \sum_{i=1}^{n} i^2$ з перевіркою умови продовження роботи циклу до і після виконання дій.

а) Умова продовження накопичення суми (виконання циклу) перевіряється до виконання дії

— з використанням циклу з передумовою:

```
...
  scanf("%d",&n);
  s=0;
  i=1;
  while(i<=n) {
    s=s+i*i;
    i++;
  }
  printf("Сума= %d",s);
...
```

— з використанням циклу з параметром:

```
...
  scanf("%d",&n);
```

```
  s=0;
  for (i=1; i<=n; i++)    // при n<1 алгоритм працює коректно –
                          // цикл жодного разу не виконується
    s=s+i*i;
  printf("Сума= %d",s);
...
```

б) Умова продовження накопичення суми (виконання циклу) перевіряється після виконання дії — з використанням циклу з післяумовою:

```
...
  scanf("%d",&n);
  s=0;
  i=0;    // i=0, щоб при n<1 алгоритм працював
          // коректно і видавав значення суми 0
  do {
    s=s+i*i;
    i++;
  } while(i<=n);
  printf("Suma= %d",s);
...
```

При обчисленні сум змінній, у якій накопичується значення суми, обов'язково треба надати початкове значення. Як правило, — це 0. Якщо обчислюється добуток, то початкове значення, як правило, встановлюють рівним 1.

Майже завжди можна обійтися без оператора циклу з післяумовою, але в деяких ситуаціях його використання спрощує код.

Приклад 6.2.3. Нехай треба вивести на екран окремі цифри цілого беззнакового числа у зворотному порядку. Будь-яке число має хоч би одну цифру, навіть число 0. Можна записати алгоритм розв'язання цієї задачі з використанням циклу з передумовою while:

```
...
  if (a==0) printf("%d\n", a);
  else
```

```
    while (a> 0) {
      printf("%d\n",a%10);
      a = a/10;
    }
...
```

У цьому алгоритмі, якщо *a* дорівнює 0, то цикл не виконається жодного разу. Тому перед циклом використано оператор if, в якому значення змінної порівнюється з 0.

Використання циклу з післяумовою do-while не потребує окремо розглядати випадок значення 0:

```
  do {
    printf("%d\n", a% 10);
    a =a/10;
  } while (a>0);
```

оскільки тут тіло циклу виконується хоч би один раз, а при нульовому значенні змінної *a* — рівно один раз.

Крім програмування циклічних алгоритмів обчислювального характеру, використання операторів циклу, дає можливість виконувати деякі корисні дії, пов'язані з введенням інформації. Це, зокрема, захист користувача від механічних помилок під час вводу, багаторазове виконання алгоритмів з різними вхідними даними, наприклад, при тестуванні їхньої роботи чи проведенні обчислювальних експериментів.

У поданих нижче циклічних алгоритмах враховується те, що якщо числові дані вводити за допомогою функції scanf і числа задавати без синтаксичних помилок, то функція повертає значення, рівне кількості правильно введених даних. Також враховується те, що після останнього введення даних функцією scanf у буфері залишається символ '\n' (Enter) або якась інша зайва інформація від будь-якого попереднього вводу.

Приклад 6.2.4. Для ***запобігання механічним помилкам під час вводу інформації*** доцільно замість фрагменту програми:

```
  printf("Введіть a, b, c  ");
```

```
  scanf("%lf %lf %lf",&a,&b,&c);
```
використовувати такий циклічний алгоритм:

```
  fseek(stdin,0,SEEK_END);   // очистка буфера(бібліотека stdio.h),
                             // працює не у всіх середовищах
  printf("Введіть a, b, c  ");
  while (scanf("%lf %lf %lf",&a,&b,&c)<3) {   // передбачення
                                     // механічних помилок вводу
    printf("Помилка в даних. Повторіть ввід a, b, c  ");
    fseek(stdin,0,SEEK_END);   // очистка буфера
  }
```

Приклад 6.2.5. Для ***перевірки правильності введення даних*** можна скористатися таким алгоритмом — цикл працює доти, доки не буде введено правильне значення:

```
  ind=1;
  do {
    printf("Введіть значення x в межах від 0 до 9:");
    scanf("%lf", &x);
    fseek(stdin,0,SEEK_END);    // очистка буфера
    if (x<0 || x>9) printf("\tнеправильне значення x\n");
    else ind=0;
  }  while (ind);
```

Приклад 6.2.6. Для ***зациклювання роботи програми***, точніше, виконання алгоритму програми невизначену кількість разів, можна скористатися такими підходами.

а) Якщо числові дані вводити за допомогою функції scanf і перше число задати з помилкою, то функція поверне значення 0. Таким чином, при використанні функції scanf у циклі для закінчення вводу чисел першим можна ввести будь-який символ, відмінний від цифри (а також відмінний від знаків плюс і мінус, крапки, з якої може починатися дійсне число), тобто ввести явно *помилкове щодо числового формату значення.* Це можна реалізувати так:

```
  printf("Вводьте цілі числа. Для закінчення - букву\n");
  ind=scanf("%lld",&n);
```

```
  while (ind) {
    ...   // оператори обробки даних
    ind=scanf("%lld",&n);
  }
```

або, використовуючи можливості оператора циклу і можливості мови С щодо побудови виразів, можна подати значно ефективніший фрагмент алгоритму:

```
  printf("Вводьте цілі числа. Для закінчення – букву\n");
  while (scanf("%lld",&n)){  // дає 0, якщо вводиться не число
    ...   // оператори обробки даних
  }
```

б) Продовжувати чи припиняти повторне виконання алгоритму також можна, використовуючи *діалог з користувачем*. Його можна реалізувати, наприклад, так:

```
...
  char vidp;   // відповідь користувача
  do {   // зациклювання роботи програми
    ...   // виконуваний блок програми
    fseek(stdin,0,SEEK_END);   // очистка буфера - у буфері
                               // залишається Enter або інша зайва
                               // інформація від попереднього вводу
    printf("Продовжувати роботу (Y - так)? ");
    while ((vidp=getchar())==' '||vidp=='\n'||vidp=='\t');   // пропуск
                 // пробільних символів і одержання відповіді
    fseek(stdin,0,SEEK_END);    // очистка буфера
  } while (vidp=='Y'||vidp=='y'||vidp=='T'||
      vidp=='т');   // з українськими буквами Т чи т працює,
       // якщо є system("chcp 1251") і char vidp описано без unsigned
...
```

Приклад 6.2.7. Для очищення буфера вводу, крім функції fseek, можна циклічно виконати функцію getchar, скориставшись циклом з передумовою з порожнім тілом:

```
while (getchar()!='\n');
```

Виконання цього циклу полягає в зчитуванні всіх символів із буфера вводу, доки не буде зчитано символ '\n' (Enter). Таким чином буфер буде очищено від непотрібної інформації. Але, якщо при зчитуванні інформації виникне помилка чи якщо користувач при роботі в Windows натисне комбінацію клавіш Ctrl+Z або при роботі в Linux натисне комбінацію клавіш Ctrl+D, то функція getchar поверне значення EOF (кінець файла) і після цього зчитування зупиниться, і тільки наступне натискання на клавішу Enter завершить виконання циклу. Тому, щоб значення EOF не зупиняло виконання програми, краще виконати також перевірку, чи не зчитано значення EOF:

```
char c;
while ((c=getchar())!='\n' && c!=EOF); .
```

Використання розглянутих циклічних алгоритмів значно спрощує роботу користувача при введенні інформації.

Приклад 6.2.8. Подані нижче приклади демонструють різні можливості використання циклу з параметром мови C, зокрема його параметра.

а) *Вирази, які задають границі циклу*, можуть бути будь-якої складності, зокрема, містити звернення до функцій.

Обчислимо суму квадратів усіх цілих чисел, які потрапляють в інтервал $(\ln x; e^x)$ при $x \ge 1$.

Щоб одержати всі цілі числа з інтервалу $(\ln x; e^x)$ при $x \ge 1$, початковим значенням параметра циклу має бути ціла частина виразу $\ln x + 1$, а верхньою границею — ціла частина e^x.

Мовою C алгоритм можна подати так:

```
/* Сума квадратів усіх цілих чисел з інтервалу
    від логарифма до експоненти */
...
  int s, i;
  double x;
  printf("Введіть значення x  ");
```

```
  scanf("%lf",&x);
  s=0;
  // параметр циклу набуде цілих значень - без дробової частини
  for (i=(int)log(x)+1; i<=(int)(exp(x)); i++)
    s=s+i*i;
  printf("%5d",s);
...
```

```
Введіть значення x  1.5
   30
```

```
Введіть значення x  2
  140
```

б) Параметром циклу може бути змінна не тільки цілого, а й символьного чи дійсного типу:

— *параметром циклу for є змінна символьного типу.* Розглянемо, як приклад, виведення таблиці кодів символів від ‘0’ до ‘9’.

Програмна реалізація має вигляд:

```
/* Таблиця кодів символів від '0' до '9' */
...
  unsigned char c;
  printf(" symvol   kod\n");
  for (c='0'; c<='9'; c++) {
    printf("%7c - %d\n",c,c);
  }
...
```

```
 symvol   kod
      0 - 48
      1 - 49
      2 - 50
      3 - 51
      4 - 52
      5 - 53
      6 - 54
      7 - 55
      8 - 56
      9 - 57
```

— *параметром циклу for є змінна дійсного типу.* Виведення таблиці синусів і косинусів для кутів від 0° до 90° з кроком 5°. Оскільки тригонометричні функції з бібліотеки math.h працюють з радіанами, то градусну міру кута треба перевести в радіанну за формулою $\varphi_r = \varphi_{\circ} \cdot \pi/180$. Значення числа π (константу M_PI) треба взяти з бібліотеки math.h, вказавши перед її підмиканням директиву #define _USE_MATH_DEFINES.

Код програми має вигляд:

```
/* Таблиця синусів і косинусів для кутів від 0° до 90° з кроком 15° */
#include <stdio.h>
#include <stdlib.h>
#define _USE_MATH_DEFINES
#include <math.h>
```

```
int main() {
  double g,    // величина кута в градусах
         r,    // величина кута в радіанах
         k;    // коефіцієнт
  printf("     x        sin(x)        cos(x)\n");
  k=M_PI/180.0;
  for (g=0.0; g<=90.0; g+=15.0) {
    r=g*k;
    printf("%5.0f%13.7lf%13.7lf\n",g,sin(r),cos(r));
  }
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
    x        sin(x)        cos(x)
    0     0.0000000     1.0000000
   15     0.2588190     0.9659258
   30     0.5000000     0.8660254
   45     0.7071068     0.7071068
   60     0.8660254     0.5000000
   75     0.9659258     0.2588190
   90     1.0000000     0.0000000
```

в) Цикли з параметрами можуть бути вкладеними, тобто в тілі циклу можуть розміщуватися інші оператори циклу:

— *вкладені*, але *незалежні цикли for*. Розглянемо формування таблиці Келі для операції множення.

Програмно алгоритм запишеться так:

```
/* Формування таблиці Келі для операції множення */
...
  int i,j;    // параметри циклів - множники
  printf("\tТаблиця множення\n");
  for (i=1; i<=10; i++) {
    for (j=1; j<=10; j++)
      printf("%4d",i*j);
    printf("\n");
  }
...
```

```
        Таблиця множення
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

— *вкладені,* але *залежні цикли for* — в умові використовується значення параметра зовнішнього циклу. Розглянемо побудову таблиці кубів натура-

льних чисел від 1 до n. Алгоритм знаходження значень кубів цілих чисел легко побудувати, проаналізувавши подану послідовність обчислень: $1^3 = 1$, $2^3 = 3 + 5$, $3^3 = 7 + 9 + 11$, $4^3 = 13 + 15 + 17 + 19$ і т.д.

Програмна реалізація алгоритму:

```
/* Знаходження кубів цілих чисел */
...
  long long int dod,cub;    //
  int i, j, n;
  printf("Вкажіть кількість перших членів натурального ряду,\
    \nдля яких треба знайти значення кубів\n");
  scanf("%d", &n);
  printf("\t n   n^3\n");
  dod=-1;
  for (i=1; i<=n; i++) {
    cub=0;
    for (j=1; j<=i; j++) {
      dod=dod+2;
      cub=cub+dod;
    }
    printf("%10d   %d\n", i, cub);
  }
...
```

```
Вкажіть кількість перших членів натурального ряду,
для яких треба знайти значення кубів
7
         n   n^3
         1   1
         2   8
         3   27
         4   64
         5   125
         6   216
         7   343
```

в) Цикли можуть мати не обов’язково один параметр:

— *цикл for з двома параметрами.* Розглянемо побудову таблиці множення на 9 без виконання операцій множення й додавання, а тільки виводячи окремі цифри.

Програмно алгоритм можна подати так:

```
/* Таблиця множення на 9 */
...
  for (i=1, j=9; i<=10; i++, j--)
    printf("9*%d=%d%d\n",i,i-1,j);
...
```

```
9*1=09
9*2=18
9*3=27
9*4=36
9*5=45
9*6=54
9*7=63
9*8=72
9*9=81
9*10=90
```

Використання циклу з двома параметрами також дає можливість оптимально реалізувати розміщення елементів масиву у зворотному порядкую (див. алгоритм 8.2.3).

г) По суті немає ніяких обмежень щодо використання параметрів:

— *ініціалізація параметра циклу перед циклом* — початкове значення параметра залежить від певної умови. Як приклад, розглянемо табулювання функцій $\ln x$, $\lg x$ і $\log_2 x$ на проміжку [*a*, *b*] з кроком *h*=0,25.

Програмна реалізація алгоритму:

```
/* Табулювання функцій ln(x), lg(x) i log2(x)
   на проміжку [a;b] з кроком h=0,25 */
...
  if (b<=0) printf("Проміжок не входить в ОДЗ");
  else {
    printf("    x   |     ln(x)     lg(x)  log2(x)\n");
    if (a<=0) {
      x=h;
      printf("Нижню границю проміжку відкореговано\n");
    }
    else
      x=a;
    for ( ; x<=b; x+=h)
      printf("%5.2f  |%9.3f%9.3f%9.3f\n", x, log(x),
        log10(x), log(x)/log(2.));
  }
...
```

```
 Табулювання функцій ln(x), lg(x) і log2(x)
на проміжку [a;b] з кроком h=0,25

Введіть значення a, b: 0 2
Нижню границю проміжку відкореговано
    x   |     ln(x)    lg(x)  log2(x)
 0.25   |    -1.386   -0.602   -2.000
 0.50   |    -0.693   -0.301   -1.000
 0.75   |    -0.288   -0.125   -0.415
 1.00   |     0.000    0.000    0.000
 1.25   |     0.223    0.097    0.322
 1.50   |     0.405    0.176    0.585
 1.75   |     0.560    0.243    0.807
 2.00   |     0.693    0.301    1.000
```

— *умова може бути зовсім не пов'язана з параметром циклу* — вона може бути будь-яким логічним оператором або порівнянням. Наведений нижче приклад демонструє застосування *складної умови* циклу для перевірки пароля, що вводиться користувачем. Користувачеві надаються три спроби введення пароля. Програма закінчує роботу циклу, коли використано всі три спроби або коли введено правильний пароль (функція strcmp з бібліотеки string.h порівнює два рядки і повертає 0, якщо вони рівні):

```
/* Вводити пароль можна до трьох разів */
...
  char str[20]=" ";
  for(i=0; i<3 && strcmp(str,"134"); i++) {
    printf("Введіть пароль: ");
    scanf("%s",str);
  }
  if(i==3)
    printf("Нема допуску");
  else
    printf("Працюйте");
...
```

```
Введіть пароль: 1234
Введіть пароль: 2908765
Введіть пароль: Так1234
Нема допуску
```

```
Введіть пароль: 234
Введіть пароль: 134
Працюйте
```

Якби не було обмеження щодо кількості спроб введення пароля і користувач мав можливість підбирати правильний пароль, то цикл for можна подати взагалі без використання параметра, а умова закінчення виконання циклу залежить тільки від введеної користувачем інформації:

```
/* Вводити пароль можна будь-яку кількість разів */
  char str[20]=" ";
...
  for(; strcmp(str,"134"); ) {
    printf("Введіть пароль: ");
    scanf("%s",str);
  }
    printf("Працюйте");
...
```

```
Введіть пароль: 1234
Введіть пароль: 2908765
Введіть пароль: Так1234
Введіть пароль: 234
Введіть пароль: 134
Працюйте
```

Розширені порівняно з деякими іншими мовами програмування (наприклад, Pascal, Оберон-2, Ада, Fortran) можливості мови програмування C щодо роботи з циклами значно полегшують програмування алгоритмів циклічної структури.

6.3. Приклади алгоритмів з циклами

Розглянемо приклади, які демонструють роботу і певні особливості використання операторів циклу (з параметром, передумовою, післяумовою) в алгоритмах розв'язування задач різного характеру — пов'язаних з роботою комп'ютера, класичних і просто цікавих.

Приклад 6.3.1. Програма-калькулятор для обчислення значень простих арифметичних виразів, які вводить користувач, працює в циклі до введення інформації, відмінної від числової.

Програмна реалізація простого калькулятора має вигляд:

```
/* Обчислення значень простих арифметичних виразів —
   простий калькулятор. Працює в циклі до введення букви */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main() {
  double a,b,   // числа
         r;     // результат
  char oper;    // символ операції +, -, *, /, ^
  int ind;      // індикатор друку
  system("chcp 1251 & cls");
  printf("Введіть вираз: число операція число\n\tоперація + - * / ^\
    \nВвід букви з нового рядка закінчує роботу\n");
  while (scanf("%lf%c%lf",&a,&oper,&b)) { // працює до
                                    // введення не числа
    ind=1;
    switch (oper) {
      case '+': r=a+b; break;
```

```
        case '-': r=a-b; break;
        case '*': r=a*b; break;
        case '/': if (b==0.) {
                        ind=0;
                        printf("\tДілення на нуль");
                    }
                    else r=a/b;
                    break;
        case '^': r=pow(a,b); break;
        default: ind=0; printf("\tНевідома операція");
      }
      if (ind) printf("\t%g%c%g= %g  ",a,oper,b,r);
      while (getchar()!='\n')
        printf("#");
      printf("\n");
    }
   printf("\n\n");
   system("pause");
   return 0;
}
```

```
Введіть вираз: число операція число
        операція + - * / ^
Ввід букви з нового рядка закінчує роботу
3+-7
        3+-7= -4
3*7
        3*7= 21
3/8
        3/8= 0.375
25^0.5
        25^0.5= 5
3-7*5
        3-7= -4  ##
3/0
        Ділення на нуль
6:2
        Невідома операція
кінець
```

У програмуванні існує таке поняття, як *«машинний епсилон»* — числове значення, менше від якого не можливо задати відносну точність для будь-якого алгоритму, який повертає дійсне число. Абсолютне значення «машинного епсилона» залежить від розрядності сітки певного комп’ютера, типу (розрядності) використовуваних при розрахунках чисел і від прийнятого в конкретному трансляторі формату подання дійсних чисел (кількості бітів, які відводяться на мантису й експоненту). «Машинний епсилон» також можна визначити як мінімальне, відмінне від 0 дійсне число, яке після додавання до 1,0 дає ще результат, відмінний від 1,0.

Значення «машинних епсилонів» зберігаються в бібліотеці float.h. Для типу float — $FLT_EPSILON=1.192092896\cdot10^{-7}$, а для типу double — $DBL_EPSILON = 2.2204460492503131\cdot10^{-16}$.

Практичне значення «машинного епсилона» пов'язане з тим, що два відмінних від нуля числа є однаковими з точки зору машинної арифметики, якщо їхня різниця за модулем менша від «машинного епсилона».

Приклад 6.3.2. Знаходження «машинного епсилона» для даних типу double.

Алгоритм знаходження «машинного епсилона» полягає в тому, що спочатку його значення береться рівним 1,0; на кожному кроці циклу воно зменшується вдвічі; цикл виконується доти, поки при додаванні цього значення до 1,0 одержується результат, відмінний від 1,0.

Програмна реалізація алгоритму для даних типу double така:

```
/* Обчислення і виведення на екран значення "машинного
   епсилона" - мінімального, відмінного від 0 дійсного
   числа, яке після додавання до 1.0 дає ще результат,
   відмінний від1.0. Стандартне бібліотечне значення
   DBL_EPSILON = 2.2204460492503131e-016 */
...
#include <float.h>
int main() {
  double eps=1;
  while (eps/2+1.0>1.0)
    eps=eps/2;
  printf("Обчислений машинний епсилон\n\
         типу double = %.18e",eps);
  printf("\nБібліотечний машинний епсилон\n\
         типу double\n\t DBL_EPSILON= %.18e", DBL_EPSILON);
...
```

```
Обчислений машинний епсилон
        типу double = 2.220446049250313100e-016
Бібліотечний машинний епсилон
        типу double
        DBL_EPSILON= 2.220446049250313100e-016
```

Оскільки треба знайти останнє значення eps, яке впливає на результат, то в умові закінчення циклу використати вираз `eps+1.0>1.0` не можна, бо це буде перший eps, який не впливає на результат. Тому умова має бути

`eps/2+1.0>1.0`. Хіба що можна подати такий фрагмент програми, використавши робочу змінну eps1:

```
...
  double eps=1,eps1;
  while (eps+1.0>1.0) {
    eps1=eps;
    eps=eps/2;
  }
  printf("Обчислений машинний епсилон\n\
          типу double = %.18e",eps1);
...
```

Поданий фрагмент буде працювати трохи швидше, оскільки в кожному проході циклу буде виконуватися одне ділення замість двох. Зауважимо, що тут ділення на 2 замінити побітовим зсувом вправо не можна, оскільки змінна eps має тип double, а не int.

Якщо в розглянутому в прикладі 6.3.2 алгоритмі виконати ще один прохід циклу, то буде одержано число, яке при додаванні до 1,0 дасть все одно 1,0. Це число, менше від «машинного епсилона», і воно сприймається машиною як нуль. Таких чисел може бути багато. Відносно числа 1,0 «машинним нулем» є максимальне, відмінне від 0 дійсне число, яке після додавання до 1,0 дає результат, рівний 1,0. Як і значення «машинного епсилона», значення «машинного нуля» залежить від розрядності сітки комп'ютера, типу використовуваних при розрахунках чисел і від прийнятого в конкретному трансляторі формату подання дійсних чисел. У будь-якому ітераційному алгоритмі, який повертає дійсне число, точність не може бути меншою або рівною значенню «машинного нуля».

Приклад 6.3.3. Обчислення з точністю до «машинного нуля» значення суми числового ряду:

$$\frac{1}{1\cdot 2\cdot 3}+\frac{1}{2\cdot 3\cdot 4}+\frac{1}{3\cdot 4\cdot 5}+\frac{1}{4\cdot 5\cdot 6}+...$$

Щоб досягнути точності «машинного нуля», треба обчислення виконувати доти, поки два послідовні значення суми ряду не будуть сприйматися машиною, як рівні.

Програма має вигляд:

```
/* Обчислення з точністю до "машинного нуля" значення
   суми числового ряду
   1/(1*2*3)+1/(2*3*4)+1/(3*4*5)+1/(4*5*6)+...(=0.25) */
...
  double sum1, sum2;   // попередня й наступна суми
  int n;   // кількість членів ряду, які ввійшли в суму
  n=1;
  sum2=0.;
  do {   // обчислення суми
    sum1=sum2;
    sum2 +=1.0/(n*(n+1)*(n+2));
    n++; }
  while (sum1<sum2);
  printf("Сума %d членів ряду\
    \n 1/(1*2*3)+1/(2*3*4)+... = %lg",n,sum2);
...
```

```
Сума 1291 членів ряду
 1/(1*2*3)+1/(2*3*4)+... = 0.25
```

У поданій програмі сума обчислюється як значення змінної sum2. Її попереднє значення зберігається в змінній sum1. Оскільки наближене значення з додаванням невід’ємних доданків не зменшується, умовою продовження циклу є відношення sum1<sum2. Коли при додаванні поточного доданка значення суми залишається незмінним (за рахунок скінченності розрядної сітки для подання дійсних чисел), порушується умова sum1<sum2 і цикл припиняє виконуватися. Скінченність розрядної сітки подання дійсних чисел у комп’ютері визначає «машинний нуль».

При обчисленні сум порядок доданків має значення. Раніше в прикладі 3.4.13 було розглянуто вплив порядку доданків на значення сум при обчисленнях на віртуальному комп’ютері. Розглянемо прості приклади знаходження сум на реальному комп’ютері з використанням оператора циклу з па-

раметром. Для накопичення сум скористаємося змінними типу float, які зберігають 7-8 значущих десяткових розрядів.

Приклад 6.3.4. Обчислення сум: а) $S=\sum_{i=1}^{n}\frac{1}{i}$, $S=\sum_{i=n}^{1}\frac{1}{i}$; б) $S=10000+\sum_{i=1}^{n}\frac{1}{i}$, $S=\sum_{i=1}^{n}\frac{1}{i}+10000$.

а) Обчислимо ту саму суму з великою кількістю доданків у прямому і зворотному порядках.

```
/* Знаходження суми в прямому й зворотному порядку */
/* УВАГА: програма може працювати кілька хвилин – не зависає */
...
  float s1,s2;
  long long int n; // максимальне значення: 9223372036854775807
  /* Знаходження першої суми - від більшого
     значення до меншого */
  s1=0.;
  for (n=1; n<=100000000; n++)
    s1+=1./(n*n);
  /* Знаходження другої суми - від меншого
     значення до більшого */
  s2=0.;
  for (n=100000000; n>=1; n--)
    s2+=1./(n*n);
  printf("s1=%.8lf\n",s1);
  printf("s2=%.8lf\n\n",s2);
...
```

```
s1=1.64472532
s2=1.64493406
```

Знайдені суми відрізняються, починаючи з 4 знака після коми.

б) Додамо до великого числа малі числа і додамо до суми малих чисел велике число.

```
/* Додавання великого і малих чисел */
...
  float s1,s2;
  long long int n;
```

```
  /* Знаходження першої суми - до великого
     числа додаємо малі числа */
  s1=10000.;
  for (n=1; n<=100000000; n++)
    s1+=1./(n*n);
  /* Знаходження другої суми - до суми малих
     чисел додаємо велике число */
  s2=0.;
  for (n=1; n<=100000000; n++)
    s2+=1./(n*n);
  s2=s2+10000.;
  printf("s1=%.8f\n",s1);
  printf("s2=%.8f\n\n",s2);
...
```

```
s1=10001.62695313
s2=10001.64453125
```

З наведених обчислень видно, що сума ста мільйонів малих чисел у випадку, коли малі числа додавати до великого, відрізняється вже другим знаком після коми від числа, одержаного додаванням великого числа до суми малих.

Таким чином, якщо в суму входять дуже малі значення і великі, то треба обчислення починати з малих значень.

Якщо лічильником циклу є беззнакова цілочисельна змінна і вона набуває свого максимально можливого значення, то наступним її значенням (при додаванні 1) за рахунок арифметичного переповнення буде число 0; якщо цілочисельна величина має знак, то внаслідок арифметичного переповнення (при додаванні 1) буде одержано від'ємне число — мінімальне, яке відповідає діапазону значень певного типу змінних. Обидва випадки без використання додаткових умов виходу з циклу призведуть до зациклювання роботи програми.

Приклад 6.3.5. Формування таблиці символів і відповідних кодів ASCII (якщо в програмі використати команду `system("chcp 1251")`, то формуватиметься таблиця кодів ANSI).

Для формування таблиці досить в циклі перебрати коди всіх друкованих символів — від 32 до 255. При цьому треба врахувати, що якщо при формуванні таблиці використати змінну типу unsigned char (набуває значень в діапазоні від

0 до 255) і значення змінної перевищить 255, то за рахунок арифметичного переповнення це значення перетвориться на 0 і робота програми зациклиться. При складанні алгоритму треба, щоб код при значенні 255 не збільшувався.

Мовою С алгоритм формування таблиці символів і кодів можна записати з використанням циклу з передумовою так:

```
/* Виведення таблиці символів і кодів ASCII, якщо не виконано
   команду  system("chcp 1251"); виведення таблиці символів і
   кодів ANSI, якщо виконано команду system("chcp 1251"); */
...
  unsigned char sym;    // символ
  /* Символи з кодами 0-31 є службовими; пробіл має код 32 */
  printf("\t\tTablytsia symvoliv ASCII\n");
  sym=31;
  while (sym<255) {
    sym++;    // якщо до sym=255 додати 1, то буде значення 0
              // за рахунок переповнення і програма зациклиться
    printf("\t%c - %d\t",sym,sym);   // по 16 символів на колонку
  }
...
```

```
            Tablytsia symvoliv ASCII
  - 32        ! - 33        " - 34        # - 35        $ - 36
% - 37        & - 38        ' - 39        ( - 40        ) - 41
* - 42        + - 43        , - 44        - - 45        . - 46
/ - 47        0 - 48        1 - 49        2 - 50        3 - 51
4 - 52        5 - 53        6 - 54        7 - 55        8 - 56
9 - 57        : - 58        ; - 59        < - 60        = - 61
> - 62        ? - 63        @ - 64        A - 65        B - 66
C - 67        D - 68        E - 69        F - 70        G - 71
H - 72        I - 73        J - 74        K - 75        L - 76
M - 77        N - 78        O - 79        P - 80        Q - 81
R - 82        S - 83        T - 84        U - 85        V - 86
W - 87        X - 88        Y - 89        Z - 90        [ - 91
\ - 92        ] - 93        ^ - 94        _ - 95        ` - 96
a - 97        b - 98        c - 99        d - 100       e - 101
f - 102       g - 103       h - 104       i - 105       j - 106
k - 107       l - 108       m - 109       n - 110       o - 111
p - 112       q - 113       r - 114       s - 115       t - 116
u - 117       v - 118       w - 119       x - 120       y - 121
z - 122       { - 123       | - 124       } - 125       ~ - 126
⌂ - 127       А - 128       Б - 129       В - 130       Г - 131
Д - 132       Е - 133       Ж - 134       З - 135       И - 136
Й - 137       К - 138       Л - 139       М - 140       Н - 141
О - 142       П - 143       Р - 144       С - 145       Т - 146
У - 147       Ф - 148       Х - 149       Ц - 150       Ч - 151
Ш - 152       Щ - 153       Ъ - 154       Ы - 155       Ь - 156
Э - 157       Ю - 158       Я - 159       а - 160       б - 161
в - 162       г - 163       д - 164       е - 165       ж - 166
з - 167       и - 168       й - 169       к - 170       л - 171
м - 172       н - 173       о - 174       п - 175         - 176
▒ - 177       ▓ - 178       │ - 179       ┤ - 180       ╡ - 181
╢ - 182       ╖ - 183       ╕ - 184       ╣ - 185       ║ - 186
╗ - 187       ╝ - 188       ╜ - 189       ╛ - 190       ┐ - 191
└ - 192       ┴ - 193       ┬ - 194       ├ - 195       ─ - 196
┼ - 197       ╞ - 198       ╟ - 199       ╚ - 200       ╔ - 201
╩ - 202       ╦ - 203       ╠ - 204       ═ - 205       ╬ - 206
╧ - 207       ╨ - 208       ╤ - 209       ╥ - 210       ╙ - 211
╘ - 212       ╒ - 213       ╓ - 214       ╫ - 215       ╪ - 216
┘ - 217       ┌ - 218       █ - 219       ▄ - 220       ▌ - 221
▐ - 222       ▀ - 223       р - 224       с - 225       т - 226
у - 227       ф - 228       х - 229       ц - 230       ч - 231
ш - 232       щ - 233       ъ - 234       ы - 235       ь - 236
э - 237       ю - 238       я - 239       Ё - 240       ё - 241
Є - 242       є - 243       Ї - 244       ї - 245       Ў - 246
ў - 247       ° - 248       ∙ - 249       · - 250       √ - 251
№ - 252       ¤ - 253       ■ - 254         - 255
```

```
            Таблиця символів ANSI
  - 32        ! - 33        " - 34        # - 35        $ - 36
% - 37        & - 38        ' - 39        ( - 40        ) - 41
* - 42        + - 43        , - 44        - - 45        . - 46
/ - 47        0 - 48        1 - 49        2 - 50        3 - 51
4 - 52        5 - 53        6 - 54        7 - 55        8 - 56
9 - 57        : - 58        ; - 59        < - 60        = - 61
> - 62        ? - 63        @ - 64        A - 65        B - 66
C - 67        D - 68        E - 69        F - 70        G - 71
H - 72        I - 73        J - 74        K - 75        L - 76
M - 77        N - 78        O - 79        P - 80        Q - 81
R - 82        S - 83        T - 84        U - 85        V - 86
W - 87        X - 88        Y - 89        Z - 90        [ - 91
\ - 92        ] - 93        ^ - 94        _ - 95        ` - 96
a - 97        b - 98        c - 99        d - 100       e - 101
f - 102       g - 103       h - 104       i - 105       j - 106
k - 107       l - 108       m - 109       n - 110       o - 111
p - 112       q - 113       r - 114       s - 115       t - 116
u - 117       v - 118       w - 119       x - 120       y - 121
z - 122       { - 123       | - 124       } - 125       ~ - 126
⍰ - 127       Ђ - 128       Ѓ - 129       ‚ - 130       ѓ - 131
„ - 132       … - 133       † - 134       ‡ - 135       € - 136
‰ - 137       Љ - 138       ‹ - 139       Њ - 140       Ќ - 141
Ћ - 142       Џ - 143       ђ - 144       ‘ - 145       ’ - 146
“ - 147       ” - 148       • - 149       – - 150       — - 151
⍰ - 152       ™ - 153       љ - 154       › - 155       њ - 156
ќ - 157       ћ - 158       џ - 159         - 160       Ў - 161
ў - 162       Ј - 163       ¤ - 164       Ґ - 165       ¦ - 166
§ - 167       Ё - 168       © - 169       Є - 170       « - 171
¬ - 172       - - 173       ® - 174       Ї - 175       ° - 176
± - 177       І - 178       і - 179       ґ - 180       µ - 181
¶ - 182       · - 183       ё - 184       № - 185       є - 186
» - 187       ј - 188       Ѕ - 189       ѕ - 190       ї - 191
А - 192       Б - 193       В - 194       Г - 195       Д - 196
Е - 197       Ж - 198       З - 199       И - 200       Й - 201
К - 202       Л - 203       М - 204       Н - 205       О - 206
П - 207       Р - 208       С - 209       Т - 210       У - 211
Ф - 212       Х - 213       Ц - 214       Ч - 215       Ш - 216
Щ - 217       Ъ - 218       Ы - 219       Ь - 220       Э - 221
Ю - 222       Я - 223       а - 224       б - 225       в - 226
г - 227       д - 228       е - 229       ж - 230       з - 231
и - 232       й - 233       к - 234       л - 235       м - 236
н - 237       о - 238       п - 239       р - 240       с - 241
т - 242       у - 243       ф - 244       х - 245       ц - 246
ч - 247       ш - 248       щ - 249       ъ - 250       ы - 251
ь - 252       э - 253       ю - 254       я - 255
```

У цьому випадку для формування таблиці скористатися класичним виглядом оператора циклу for досить важко. Так:

— якщо використати оператор

```
for (sym=32; sym<256; sym++) printf("\t%c - %d\t",sym,sym); ,
```

еквівалентний оператору циклу з передумовою

```
sym=32;
while (sym<256) {
  printf("\t%c - %d\t",sym,sym);
  sym++;
} ,
```

то змінна sym замість значення 256, яке в 1 байт (8 бітів) пам'яті не поміститься, набуде значення 0 і робота програми зациклиться;

— якщо використати оператор

```
for (sym=32; sym<255; sym++) printf("\t%c - %d\t",sym,sym); ,
```

то значення останнього символу таблиці не виведеться на екран;

— якщо використати оператор

```
for (sym=31; sym<255; sym++) printf("\t%c - %d\t",sym+1,sym+1); ,
```

то таблиця виведеться правильно, але тут для одержання коду символу на кожній ітерації циклу двічі обчислюється вираз sym+1;

— якщо використати оператор

```
for (sym=32; sym!=0; sym++) printf("\t%c - %d\t",sym,sym); ,
```

то таблиця виведеться правильно, але тут втрачається наочність умови `sym!=0` виконання циклу.

При використанні циклів треба уважно слідкувати, чи при певних значеннях вхідних даних програма буде коректно працювати.

Приклад 6.3.6. Значення цілої змінної *a* скоротити на множники, рівні 2.

Програмно алгоритм можна реалізувати так:

```
/* Скорочення цілого числа на множники, рівні 2 */
...
  int a,    // введене число – може бути додатне, від'ємне чи нуль
      k;    // лічильник кількості кроків циклу - степінь 2
```

```
  printf("Введіть ціле число a ");
  scanf("%d",&a);
  k=0;
  if (a!=0)
    while(a%2==0) {    // можна також задати умову(!(a%2))
      a=a/2;
      k++;
    }
  printf("Скорочене число= %d на 2^%d",a,k);
...
```

```
Введіть ціле число a 48
Скорочене число= 3 на 2^4
```

```
Введіть ціле число a 7
Скорочене число= 7 на 2^0
```

Якщо в цій програмі не врахувати, що *a* може бути рівне 0, то програма зациклиться навіть при тому, що виконується оператор зміни значення `a=a/2`.

Алгоритм скорочення на 2 також можна записати, використовуючи побітові операції:

```
/* Скорочення цілого числа на множники, рівні 2,
   із застосуванням побітових операцій */
...
  k=0;
  if (a!=0)
    while(!(a&1)) {
      a=a>>1;
      k++;
    }
  printf("Скорочене число= %d на 2^%d",a,k);
...
```

```
Введіть ціле число a 480
Скорочене число= 15 на 2^5
```

Завдяки використанню побітових операцій, які виконуються значно швидше, ніж операції знаходження остачі від ділення й ділення, цей циклічний алгоритм працює значно швидше, ніж попередній.

Використання схеми Горнера при знаходженні значень поліномів прискорює обчислення. Поданий нижче приклад демонструє, як ці обчислення можна реалізувати з використанням класичного вигляду оператора циклу з параметром.

Приклад 6.3.7. Обчислення в точці *x* за схемою Горнера значення полінома:

$$y = 11x^{10} + 10x^{9} + 9x^{8} + ... + 2x + 1.$$

Перетворимо поліном до вигляду:

$$y = (...(11x + 10)x + 9)x + ... + 2)x + 1.$$

Програмна реалізація схеми Горнера така:

```
/* Обчислення значення полінома в точці x за схемою Горнера */
...
  printf("Введіть значення x  ");
  scanf("%d",&x);
  y=11;
  for (i=10; i>=1; i--)
    y=y*x+i;
  printf("%5d",y);
...
```

```
Введіть значення x  1
   66
```

```
Введіть значення x  -2
 7737
```

```
Введіть значення x  3
930022
```

У поданому прикладі коефіцієнти полінома мають спеціальний вигляд — на 1 більші від степеня змінної. У реальних розрахунках коефіцієнтами можуть бути елементи масиву з відповідними індексами (див. розд. 8).

У програмуванні цикли з післяумовою використовують набагато рідше, ніж з передумовою чи з параметром. Але при програмуванні ітераційних процесів (наближених обчислень) їм надають перевагу.

Приклад 6.3.8. Існує досить багато формул для обчислення наближених значень числа π . Розглянемо дві з них:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...\right) = 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n-1},$$

$$\pi = 3 + 4\left(\frac{1}{2 \cdot 3 \cdot 4} - \frac{1}{4 \cdot 5 \cdot 6} + \frac{1}{6 \cdot 7 \cdot 8} - ...\right) = 3 + 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n(2n+1)(2n+2)} =$$

$$= 3 + \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n(2n+1)(n+1)}.$$

Одна з поданих формул має повільнішу, інша — швидшу збіжність.

Алгоритм обчислення значення числа π на основі першої формули може бути таким (обчислення закінчиться, як тільки значення загального члена ряду стане меншим від заданого досить малого числа *eps*):

```
/* Обчислення значення числа PI за простою формулою */
...
  double eps,    // точність
         s,      // сума (наближене значення PI)
         an;     // загальний член ряду
  long long int n;    // кількість членів ряду
  int one_n;    // знак доданка
  printf("Введіть значення eps: ");
  scanf("%lf",&eps);
  one_n=1;
  n=1;
  s=0;    // можна і s=1, але для точності обчислень краще s=0
  do {    // обчислення суми, починаючи з другого доданка
    one_n=-one_n;
    n++;
    an=1./(2*n-1);
    s=s+one_n*an; }
  while (an>=eps);
  s=4.*(s+1.);
  printf("Наближене значення PI: %.17f\n",s);
  printf("Кількість ітерацій: %d",n);
...
```

```
Введіть значення eps: 1e-7
Наближене значення PI: 3.14159285358967820
Кількість ітерацій: 5000001
```

При точності 10^{-7} цей алгоритм дав 6 правильних знаків після коми за 5000001 (5 млн.) кроків (у бібліотеці math.h зберігається значення $\pi \approx 3{,}14159265358979323846$ з 20 значущими цифрами після коми).

Алгоритм обчислення значення числа π на основі другої формули:

```
/* Обчислення значення числа PI за складнішою формулою */
...
  double eps,    // точність
         s,      // сума (наближене значення PI)
         an;     // загальний член ряду
  long long int n;    // кількість членів ряду
  int one_n;    // знак доданка
```

```
  printf("Введіть значення eps: ");
  scanf("%lf",&eps);
  one_n=-1; n=0;
  s=0;
  do {    // обчислення суми, починаючи з першого доданка
    one_n=-one_n;
    n++;
    an=1./(n*(2*n+1)*(n+1));
    s=s+one_n*an; }
  while (an>=eps);
  s=3.+s;
  printf("Наближене значення PI: %.17f\n",s);
  printf("Кількість ітерацій: %d\n",n);
...
```

```
Введіть значення eps: 1e-7
Наближене значення PI: 3.14159270271861150
Кількість ітерацій: 171
```

При точності 10^{-7} другий алгоритм дав 6 правильних знаків після коми за 171 крок. Тобто збіжність ряду дуже велика.

В алгоритмах виконання арифметичних операцій у прикладах 3.5.3 і 3.5.4 для дуже часто обчислюваних значень (знаходження $1/y$ і $1/\sqrt{y}$ для дійсного аргументу y одинарної точності (float), який зберігається в пам'яті комп'ютера в форматі стандарту IEEE-754) після вибору хорошого початкового наближення з використанням «магічних» чисел виконувалися дві ітерації методу Ньютона, щоб досягнути досить високої точності обчислень. Крім знаходження значень вказаних двох функцій (степені –1 і $-1/2$), також розроблено формулу обчислення «магічних» чисел для загального випадку функції y^p при $p \in [-1; 1]$. Проте недоліком такого підходу є те, що значення «магічного» числа залежить від використовуваної точності і формату зберігання дійсного числа. Використання «магічних» чисел значно прискорює обчислення, але якщо таке число не знайдено, то початковим наближенням також можна взяти і число 1, як це робиться в методі Ньютона (методі дотичних для знаходження нулів функції з заданим початковим наближенням). При цьому, звичайно ж, знадобиться значно більша кількість ітерацій, але метод збігається досить швидко.

Приклад 6.3.9. Обчислення квадратного кореня з будь-якого дійсного числа. Для обчислення квадратного кореня можна запропонувати такий алгоритм:

Якщо a<0, то квадратний корінь не існує;

якщо a=0, то корінь =0;

якщо a>0, то для обчислення квадратного кореня застосувати *метод Ньютона*:

$$b_0 = 1; \quad b_{n+1} = \frac{1}{2}\left(\frac{a}{b_n} + b_n\right),$$

де b_n — наближене значення $\sqrt{a}$. Подану рекурентну формулу називають ще *формулою Герона*.

Для виходу із циклу ітерацій (послідовних наближень) можна використати абсолютну похибку, наприклад:

$$\left|a - b_n^2\right| < 10^{-6}.$$

Але цей критерій не спрацює, якщо число буде меншим від 10^{-6}. Тому тут треба скористатися відносною похибкою:

$$\left|\frac{a}{b_n^2} - 1\right| < 10^{-6}.$$

Програмна реалізація алгоритму така:

```
/* Обчислення квадратного кореня з дійсного числа a
   за методом дотичних Ньютона (формула Герона) */
...
  const double eps=1.e-6;
  double a,   // введене число
         bn;  // наближене значення кореня
  int n;   // кількість ітерацій
  printf(" Обчислення квадратного кореня\n з точністю %lg\n", eps);
  printf("Вводьте дійсні числа.\nДля закінчення вводу - букву");
  while (printf("\n   a= "), scanf("%lg", &a)) {
    if (a<0) printf("\tкорінь не існує");
```

```
    else
      if (a==0) printf("\t0");
      else {    // число >0
         bn=1;
         n=0;
         do {    // обчислення за формулою Герона
           bn=(a/bn+bn)/2;
           n++;
         } while (fabs(a/(bn*bn)-1)>=eps);
         printf("\t%lg (ітерацій - %d)", bn, n);
      }
  }
...
```

```
 Обчислення квадратного кореня
 з точністю 1e-006
Вводьте дійсні числа.
Для закінчення вводу - букву
   a= -5
        корінь не існує
   a= 0
        0
   a= 1
        1 (ітерацій - 1)
   a= 1.44
        1.2 (ітерацій - 3)
   a= 2
        1.41421 (ітерацій - 4)
   a= 81
        9 (ітерацій - 7)
   a= 10000
        100 (ітерацій - 10)
   a= кінець
```

```
 Обчислення квадратного кореня
 з точністю 1e-012
Вводьте дійсні числа.
Для закінчення вводу - букву
   a= -3
        корінь не існує
   a= 0
        0
   a= 1
        1 (ітерацій - 1)
   a= 1.44
        1.2 (ітерацій - 4)
   a= 2
        1.41421 (ітерацій - 5)
   a= 81
        9 (ітерацій - 8)
   a= 10000
        100 (ітерацій - 11)
   a= кінець
```

Приклад 6.3.10. Метод дотичних Ньютона також дає можливість знайти корінь будь-якого цілого степеня k з дійсного числа a.

Пропонується такий алгоритм.

Якщо k=0, то корінь знайти неможливо;

інакше, якщо a<0 і k — парне або a=0 і k — від'ємне, то корінь не існує;

інакше, якщо k=1 або a=0, то a;

інакше, якщо k= –1, то $\frac{1}{a}$;

інакше, якщо k>0, то шукати $\sqrt[k]{a}$;

інакше (якщо k<0) шукати $1/\sqrt[k]{a}$.

Для обчислення кореня степеня *k* застосовуємо метод Ньютона:

$$b_0 = 1; \quad b_{n+1} = \frac{1}{k}\left(\frac{a}{b_n^{k-1}} + (k-1)b_n\right),$$

де b_n — наближене значення $\sqrt[k]{a}$.

Для виходу із циклу ітерацій треба скористатися відносною похибкою:

$$\left|\frac{a}{b_n^k} - 1\right| < 10^{-6}.$$

Програмна реалізація алгоритму така:

```
/* Обчислення кореня степеня k з дійсного числа a
   за методом дотичних Ньютона */
...
  const double eps=1.e-6;
  double a,   // введене число
         bn;  // наближене значення кореня
  int k,   // степінь кореня
      n,   // кількість ітерацій
      signk,   // знак степеня: 1 – додатний, -1 – від'ємний
      signa;   // знак числа: 1 – додатний, -1 – від'ємний
  printf(" Обчислення кореня степеня k \n з точністю %lg\n", eps);
  printf("Для закінчення вводу введіть букву\n");
  printf("Вводьте цілі степені і дійсні числа\n k  a\n");
  while (scanf("%d %lg",&k,&a)) {
    n=0;
    if (k==0) printf("\tзнайти неможливо");
    else
      if ((a<0 && k%2==0) || (a==0 && k<0))
        printf("\tкорінь не існує");
      else {   // корінь існує
        if (k==1 || a==0) bn=a;
        else
          if (k==-1) bn=1/a;
          else {   // знаходження кореня з числа a
```

```
          signk=1; signa=1;  // степінь додатний, число додатне
          if (k<0) { k=-k; signk=-1; }   // знак степеня
          if (a<0) { a=-a; signa=-1; }   // знак числа
          bn=1;
          do {   // ітерації методу Ньютона
            bn=(a/pow(bn,k-1)+(k-1)*bn)/k;
            n++;
          } while (fabs(a/pow(bn,k)-1)>=eps);
          if (signk==-1)
            bn=1/bn;   // знак степеня від'ємний
          bn=bn*signa;   // знак числа від'ємний
        }
      printf("\t%lg (ітерацій - %d)", bn, n);
    }
  printf("\n");
}
...
```

```
Обчислення кореня степеня k
 з точністю 1e-006
Для закінчення вводу введіть букву
Вводьте цілі степені і дійсні числа
 k  a
2 5
        2.23607 (ітерацій - 4)
2 -5
        корінь не існує
1 -5
        -5 (ітерацій - 0)
3 -27
        -3 (ітерацій - 7)
-3 8
        0.5 (ітерацій - 6)
0 5
        знайти неможливо
кінець
```

```
Обчислення кореня степеня k
 з точністю 1e-012
Для закінчення вводу введіть букву
Вводьте цілі степені і дійсні числа
 k  a
2 5
        2.23607 (ітерацій - 5)
6 -3
        корінь не існує
1 8
        8 (ітерацій - 0)
3 -27
        -3 (ітерацій - 8)
-3 8
        0.5 (ітерацій - 7)
2 1.44
        1.2 (ітерацій - 4)
кінець
```

Як видно з наведених прикладів, збільшення точності обчислень від 10^{-6} до 10^{-12} призвело до збільшення кількості ітерацій лише на одну, що свідчить про швидку збіжність методу Ньютона.

На практиці часто використовують циклічні алгоритми, пов’язані з задачами теорії чисел — зокрема, це обчислення чисел Фібоначчі, знаходження суми цифр цілого числа, визначення того, чи є число простим, це алгоритм Евкліда знаходження найбільшого спільного дільника тощо.

Приклад 6.3.11. Знаходження *n* чисел Фібоначчі. Числа Фібоначчі визначають за рекурентною формулою другого порядку (у формулу обчислення значення поточного члена послідовності входять значення двох попередніх членів):

$$f_0 = f_1 = 1;$$

$$f_n = f_{n-1} + f_{n-2};\ n = 2,3,...$$

При обчисленні значень членів послідовностей за рекурентними формулами другого і вищих порядків використовують змінні для зберігання кількох попередніх (їхня кількість збігається з порядком формули) і одержання поточного значень. На кожному проході циклу відбувається зсув попередніх значень на одне вниз — при цьому поточне значення поточного проходу стає попереднім щодо наступного проходу циклу (зауважимо, що якщо рекурентна формула має перший порядок, то додаткову змінну для зберігання попереднього значення не використовують). Оскільки числа Фібоначчі визначаються рекурентною формулою другого порядку, то при обчисленнях треба зберігати два попередні значення.

Фрагмент програми обчислення чисел Фібоначчі такий:

```
/* Обчислення значень n чисел Фібоначчі */
...
  unsigned long long int fn_2, fn_1, fn;   // числа Фібоначчі
  int n, i;   // кількість чисел, лічильник
  fn_2=fn_1=1;   // задання значень f0 і f1
  printf("f0= %llu\nf1= %llu\n",fn_2,fn_1);
  printf("Введіть значення n  ");
  scanf("%d",&n);
  /* Одержання значень fn */
  for (i=2; i<=n; i++) {
```

```
    fn=fn_1+fn_2;
    printf("f%-2d= %-15llu",i,fn);
    fn_2=fn_1;
    fn_1=fn;
  }
...
```

```
Введіть значення n  60
f0 = 1
f1 = 1
f2 = 2              f3 = 3              f4 = 5              f5 = 8
f6 = 13             f7 = 21             f8 = 34             f9 = 55
f10= 89             f11= 144            f12= 233            f13= 377
f14= 610            f15= 987            f16= 1597           f17= 2584
f18= 4181           f19= 6765           f20= 10946          f21= 17711
f22= 28657          f23= 46368          f24= 75025          f25= 121393
f26= 196418         f27= 317811         f28= 514229         f29= 832040
f30= 1346269        f31= 2178309        f32= 3524578        f33= 5702887
f34= 9227465        f35= 14930352       f36= 24157817       f37= 39088169
f38= 63245986       f39= 102334155      f40= 165580141      f41= 267914296
f42= 433494437      f43= 701408733      f44= 1134903170     f45= 1836311903
f46= 2971215073     f47= 4807526976     f48= 7778742049     f49= 12586269025
f50= 20365011074    f51= 32951280099    f52= 53316291173    f53= 86267571272
f54= 139583862445   f55= 225851433717   f56= 365435296162   f57= 591286729879
f58= 956722026041   f59= 1548008755920  f60= 2504730781961
```

Для типу даних long long int правильно обчислюються числа до f_{91} включно (19 знаків: f_{90}=4660046610375530309, f_{91}=7540113804746346429), а далі відбувається арифметичне переповнення; при unsigned long long int правильно обчислюються числа до f_{92} включно (20 знаків: f_{92}=12200160415121876738), а далі відбувається арифметичне переповнення.

Приклад 6.3.12. Знаходження суми цифр цілого числа.

Для одержання окремих цифр числа треба скористатися операцією знаходження остачі від ділення цілого числа на 10, а для відкидання від числа останньої цифри треба поділити число на 10 (при діленні цілого числа на ціле результат теж буде цілим числом — дробова частина відкидається, незалежно від того, яке вона має значення). Умовою припинення одержання цифр числа є рівність числа нулю після відкидання цифр.

Оскільки число може бути досить великим, то в програмній реалізації використаємо змінну типу long long int (19-значні десяткові числа; діапазон

значень –9223372036854775808 … 9223372036854775807) і відповідні формати і функції роботи з даними цього типу, щоб можна було вводити досить великі цілі числа. Для введення функцією scanf цілих чисел використаємо формат %19lld (зчитує до пробільного символа, але не більше 19 символів).

Щоб програмою було зручно користуватися, треба: для візуального контролю вивести на екран введене число; очистити буфер від зайвих символів які можуть залишатися в ньому після зчитування числа; передбачити можливість програмної обробки багатьох чисел.

При організації циклу введення даних треба врахувати, що в мові C функція scanf повертає кількість реально введених елементів даних, а результатом операції кома є значення останнього обчисленого виразу.

Програмно алгоритм можна подати так:

```
/* Обчислення суми цифр цілого числа. Числа вводяться
   до введення символу, відмінного від цифри */
...
  long long int n;   // введене число
  int i,   // лічильник кількості цифр у числі
      s;   // сума цифр
  printf("Вводьте цілі числа з діапазону\n\
         -9223372036854775808 … 9223372036854775807.\n\
         Для закінчення – букву.\n");
  while (printf("Введіть число N "), scanf("%19lld",&n)) {
    printf("  Введене число %lld.\n",n);
    n=llabs(n);   //функція бібліотеки stdlib.h
    s=0; i=0;
    /* Обчислення суми цифр */
    while (n>0) {
      s=s+(int)(n%10);
      n=n/10;
      i++;
    }
    printf("  Кількість цифр= %d,\n  сума цифр= %d.\n",i,s);
```

```
    while(getchar()!='\n');    // очистка буфера
  }
...
```

```
Вводьте цілі числа з діапазону
         -9223372036854775808 … 9223372036854775807.
         Для закінчення – букву.
Введіть число N 1234567890123456789012345
  Введене число 1234567890123456789.
  Кількість цифр= 19,
  сума цифр= 90.
Введіть число N -98765
  Введене число -98765.
  Кількість цифр= 5,
  сума цифр= 35.
Введіть число N кінець
```

Приклад 6.3.13. Визначення того, чи є число простим.

Прості числа діляться лише самі на себе і на 1. Якщо є число *n*, то, щоб перевірити, чи воно просте, ділити його на всі $(n-1)$ попередні числа не треба. За визначенням число *n* є простим, якщо в нього немає дільників в інтервалі [2, n/2]. Досить поділити на числа до n/2, бо далі результат буде меншим від 2. Проте і це забагато — досить поділити на числа до заокругленого до більшого $\sqrt{n}$ (наприклад, число 50=1·2·5·5 можна подати парами множників: 1 і 50, 2 і 25, 5 і 10, 10 і 5, 25 і 2, 50 і 1, які після половини послідовності повторюються (тільки в зворотному порядку) тому досить перевірити множники 2 і 5, що приблизно вкладається в діапазон чисел від 2 до $\sqrt{50} \approx 8$; число 36=1·2·2·3·3 можна подати добутками пар чисел: 1 і 36, 2 і 18, 3 і 12, 4 і 9, 6 і 6, 9 і 4, 12 і 3, 18 і 2, 36 і 1 — після пари 6 і 6 пари повторюються у зворотному порядку — тому досить перевіряти подільність на числа від 2 до $\sqrt{36} = 6$).

Отже, число *n* є *простим*, якщо в нього немає дільників в інтервалі $[2, \sqrt{n}\,]$.

Програмно алгоритм можна реалізувати так:

```
/* У циклі вводяться цілі числа, доки не буде введено будь-яку
   букву. Визначається, чи є число простим */
...
  long long int n;    // введене число
  int ind,    // індикатор простоти: 0 – не просте, 1 - просте
      i, s;   // лічильник і корінь з введеного числа
  printf("Вводьте цілі числа. Для закінчення – букву\n");
```

```
  while (scanf("%lld",&n)){
    ind=1;   // вважаємо, що введене число просте
    n=llabs(n);   // llabs - функція бібліотеки stdlib.h
    if (n>3) {
      s=ceil(sqrt((double)n)); i=2;   // sqrt вимагає дійсного
                   // аргументу; ceil - заокруглення до більшого
      while(i<=s && ind) {
        if (n%i==0) ind=0;   // число не просте
        i++;
      }
    }
    if (ind)
      printf("\tпросте\n");
    else
      printf("\tне просте\n");
  }
}
...
```

```
Вводьте цілі числа. Для закінчення - букву
17
        просте
111
        не просте
113
        просте
119
        не просте
1234567
        не просте
13
        просте
кінець
```

Приклад 6.3.14. Розкладання великих чисел на прості множники (див. приклади 10.6.2 і 10.6.3) для знаходження найбільшого спільного дільника (НСД) є досить трудомісткою задачею. Швидко обчислити НСД великих чисел без знаходження їхніх дільників можна за допомогою *алгоритму Евкліда.*

Алгоритм Евкліда базується на тому, що у випадку $a \ge b$, якщо число a подати, як $a = k \cdot b + r$, то $\text{НСД}(a,b) = \text{НСД}(b,r)$.

Оригінальний (описаний ще Евклідом) алгоритм для пошуку остачі r використовує (k разів) операцію віднімання. В іншій версії алгоритму для пошуку остачі r повторні віднімання замінено діленням. Дослідження алгоритму, який використовує ділення, показали, що для одержання НСД він виконує не більше кроків, ніж п’ятикратна кількість цифр n десяткового подання меншого числа. Обчислювальні витрати кожного кроку теж, як правило, мають порядок n, тому загальні обчислювальні витрати оцінюються порядком

$O(n^2)$. Якщо взяти до уваги те, що операція віднімання швидша за операцію ділення, особливо для великих чисел, то алгоритм на основі віднімання за ефективністю можна прирівняти до алгоритму на основі ділення. Для обчислення найбільшого спільного дільника двох невід'ємних цілих чисел також існує двійковий (бінарний) алгоритм Евкліда, який використовує простіші арифметичні операції, ніж звичайний алгоритм Евкліда — для одержання непарних чисел ділення замінено побітовим зсувом, а також використано порівняння і, як в оригінальному алгоритмі Евкліда, віднімання.

а) *Оригінальний варіант алгоритму Евкліда* (*на основі віднімання*): від більшого числа віднімають менше, доки числа не стануть рівними — це і є НСД.

Словесно алгоритм можна подати так:

К1. Ввести числа a, b.

К2. Якщо a=b, то перейти на К5.

К3. Якщо a>b, то a=a–b, інакше b=b–a.

К4. Перейти на К2.

К5. Вивести НСД=a.

Зауважимо, що якщо одне з чисел 0, а інше відмінне від 0, то алгоритм зациклиться. Якщо обидва числа рівні 0, то алгоритм видасть значення 0 (хоч НСД для такої пари не визначений; щоб підкреслити невизначеність результату, у програмній реалізації буде видано значення –1). При програмній реалізації алгоритмів треба звертати на це увагу й передбачати можливість таких ситуацій.

Продемонструємо виконання описаного алгоритму на різних парах чисел:

a=36, b=24
a≠b, a>b, a=36–24=12
=> a=12, b=24
a≠b, a<b, b=24–12=12
=> a=12, b=12
a=b, НСД=12

a=14, b=49
a≠b, a<b, b=49–14=35
=> a=14, b=35
a≠b, a<b, b=35–14=21
=> a=14, b=21
a≠b, a<b, b=21–14=7
=> a=14, b=7
a≠b, a>b, a=14–7=7
=> a=7, b=7
a=b, НСД=7

a=3, b=5
a≠b, a<b, b=5–3=2
=> a=3, b=2
a≠b, a>b, a=3–2=1
=> a=1, b=2
a≠b, a<b, b=2–1=1
=> a=1, b=1
a=b, НСД=1

Програмна реалізація алгоритму Евкліда на основі віднімання:

```
/* Алгоритм Евкліда на основі віднімання для знаходження
   найбільшого спільного дільника двох цілих чисел */
...
  int a,b;    // введені числа
  printf("   Алгоритм Евкліда\n(на основі віднімання)");
  while(printf("\nВведіть числа a i b \n"),scanf("%d %d",&a,&b)) {
    a=abs(a); b=abs(b);    //функція бібліотеки stdlib.h
    if (a==0||b==0) // якщо не перевірити,
                    // то алгоритм зациклиться
      printf("   НСД=%d",a<b?b:(a!=0?a:-1));
    else {  // алгоритм Евкліда
      while(a!=b)
        if (a>b)
          a=a-b;
      else
        b=b-a;
      printf("   НСДе=%d",a);
    }
  }
...
```

```
   Алгоритм Евкліда
(на основі віднімання)
Введіть числа a i b
36 24
   НСДе=12
Введіть числа a i b
14 49
   НСДе=7
Введіть числа a i b
3 5
   НСДе=1
Введіть числа a i b
0 0
   НСД=-1
Введіть числа a i b
3 0
   НСД=3
Введіть числа a i b
0 3
   НСД=3
Введіть числа a i b
кінець
```

б) *Алгоритм Евкліда* (*на основі ділення*): одне число ділиться на інше і одержується остача від ділення (a mod b → r1), друге число ділиться на одержану остачу і одержується нова остача (b mod r1 → r2), попередня остача ділиться на одержану остачу (r1 mod r2 → r3) і т.д., доки остача не стане рівною 0; останній знаменник — це НСД.

Словесно алгоритм можна подати так:

К1. Ввести числа a, b.

К2. Якщо b=0, то перейти на К5.

К3. r=a mod b, a=b, b=r.

К4. Перейти на К2.

К5. Вивести НСД=a.

Даний алгоритм нормально працює (не зациклюється, не виконує ділення на 0) на парах будь-яких чисел, наприклад, на парах 0 і 3 (результат 3), 3 і 0 (результат 3), 0 і 0 (хоч НСД для такої пари не визначений, дає результат 0).

Продемонструємо виконання алгоритму на різних парах чисел:

a=36, b=24
b ≠0, r=36 mod 24=12
=> a=24, b=12
b ≠0, r=24 mod 12=0
=> a=12, b=0
b=0, НСД=12

a=14, b=49
b ≠0, r=14 mod 49=14
=> a=49, b=14
b ≠0, r=49 mod 14=7
=> a=14, b=7
b ≠0, r=14 mod 7=0
=> a=7, b=0
b=0, НСД=7

a=3, b=5
b ≠0, r=3 mod 5=3
=> a=5, b=3
b ≠0, r=5 mod 3=2
=> a=3, b=2
b ≠0, r=3 mod 2=1
=> a=2, b=1
b ≠0, r=2 mod 1=0
=> a=1, b=0
b=0, НСД=1

Програмна реалізація алгоритму:

```
/* Алгоритм Евкліда на основі ділення для знаходження
   найбільшого спільного дільника двох цілих чисел */
...
  int a,b;    // введені числа
  int r;      // остача від ділення
  printf("   Алгоритм Евкліда\n(на основі ділення)");
  while(printf("\nВведіть числа a i b \n"),scanf("%d %d",&a,&b)) {
    a=abs(a); b=abs(b); //функція бібліотеки stdlib.h
    if (a==0||b==0) // без перевірки алгоритм працює
      printf("   НСД=%d",a<b?b:(a!=0?a:-1));
    else { // алгоритм Евкліда
      while (b) {
        r=a%b;
        a=b;
        b=r;
      }
      printf("   НСДе=%d",a);
    }
  }
...
```

```
   Алгоритм Евкліда
(на основі ділення)
Введіть числа a i b
194040 156000
   НСДе=120
Введіть числа a i b
123456 789
   НСДе=3
Введіть числа a i b
-54 51
   НСДе=3
Введіть числа a i b
кінець
```

в) *Двійковий алгоритм Евкліда* (бінарний алгоритм Евкліда, алгоритм Стайна[15]). У цьому алгоритмі враховується, що:

— якщо обидва числа a і b парні, то $НСД(a,b) = 2 \cdot НСД(a/2, b/2)$;

— якщо одне з чисел парне, а інше непарне, наприклад, при a — парне, b — непарне, то $НСД(a,b) = НСД(a/2, b)$, бо 2 не є спільним дільником;

— якщо a і b непарні, то різниця $a - b$ буде парним числом.

Словесно алгоритм описується так:

K1. Ввести числа a, b.

K2. k=0.

K3. Якщо a і b парні, то a=a/2 і b=b/2; k=k+1; перейти на К3.

K4. Якщо a парне, то a=a/2; перейти на К4. // далі a завжди непарне

K5. Якщо b парне, то b=b/2; перейти на К5.

K6. Якщо $a > b$, то виконати переприсвоєння a і b; b=b–a; перейти на К5.

K7. Вивести НСД=a*2^k.

Поданий алгоритм зациклюється на парах будь-яких чисел, серед яких є хоч би один 0, наприклад, на парах 0 і 3, 3 і 0, 0 і 0. У програмній реалізації це треба передбачити.

Програмна реалізація двійкового алгоритму Евкліда:

```
/* Двійковий алгоритм Евкліда на основі побітових зсувів
   і віднімання непарних чисел для знаходження найбільшого
   спільного дільника двох цілих чисел */
...
  int a,b;  // введені числа
  int k,    // степінь 2, на який скорочуються обидва числа
      r;    // робоча змінна для переприсвоєння
  printf("  Двійковий алгоритм Евкліда\n");
  printf("(на основі зсувів і віднімання)");
  while(printf("\nВведіть числа a i b \n"),scanf("%d %d",&a,&b)) {
```

[15] Джозеф Стайн (Josef Stein, Hebrew University of Jerusalem) — ізраїльський фізик і програміст. Опублікував двійковий алгоритм Евкліда в 1967 році (проте цей алгоритм міг бути відомим ще в першому столітті в Китаї).

```
a=abs(a); b=abs(b);   //функція бібліотеки stdlib.h
if (a==0||b==0) // якщо не перевірити, то алгоритм зациклиться
  printf("   НСД=%d",a<b?b:(a!=0?a:-1)); // якщо є один 0,
      // то видає ненульове число, якщо обидва нулі, то -1
else {
  /* Реалізація алгоритму Евкліда на основі зсувів
  (скорочення на степінь 2) і віднімання */
  k=0;
  /* Доки обидва числа парні, скорочуємо на 2 */
  while(!((a | b) & 1)) {
    a=a>>1;
    b=b>>1;
    k++;
  }
  /* Доки a парне, скорочуємо на 2 */
  while(!(a & 1))
    a=a>>1;
  /* Далі a буде завжди непарним. Доки a і b не рівні,
  виконуємо зсув b до непарного і віднімання */
  do {
    while(!(b & 1))
      b=b>>1;
    if (a>b) { // переприсвоєння значень a і b
      r=a;
      a=b;
      b=r;
    }
  } while (b=b-a);
  /* Одержання НСД з врахуванням
     степеня двійки, на який
     скорочувалися одночасно
     обидва парні числа */
  a=a<<k;
```

```
  Двійковий алгоритм Евкліда
(на основі зсувів і віднімання)
Введіть числа a і b
0 0
   НСД=-1
Введіть числа a і b
3 0
   НСД=3
Введіть числа a і b
583 264
   НСДе=11
Введіть числа a і b
17 21
   НСДе=1
Введіть числа a і b
кінець
```

```
      printf("   НСДе=%d",a);
    }
  }
...
```

Двійковий алгоритм Евкліда має часову складність $O(n^2)$, де n — кількість бітів у більшому з двох чисел. Операції побітового зсуву і віднімання виконуються за лінійний час $O(n)$ для дуже великих чисел. Хоч теоретично двійковий алгоритм обчислення НСД в середньому на 60 % ефективніший (в термінах кількості бітових операцій), ніж алгоритм Евкліда з використанням віднімання чи ділення, проте у всіх трьох алгоритмів Евкліда асимптотична складність становить $O(n^2)$.

Алгоритм Евкліда відіграє важливу роль у широковживаному криптографічному алгоритмі з відкритим ключем RSA (шифрування, цифровий підпис). Знаходження найбільшого спільного дільника потрібне при скороченні звичайних дробів, при розв’язуванні рівнянь у цілих числах (діофантових рівнянь), при побудові ланцюгових дробів, у методах факторизації (розкладанні цілих чисел на множники) та інших задач, пов’язаних з теорією чисел.

Приклад 6.3.15. Знаходження в тексті всіх цілих чисел і виведення їх на екран у стовпчик.

При виконанні завдання треба в тексті знаходити символи, які позначають цифри від '0' до '9' (їхні коди від 48 до 57). Для одержання числа, за відповідним символом треба виконати віднімання `sym-'0'`, а для формування числа n — циклічно виконувати обчислення `n*10+sym-'0'`.

Програмно алгоритм реалізується так:

```
/* Знаходить у тексті цілі числа і друкує їх у стовпчик.
   Кінець оброблюваного тексту позначається символом # */
...
  unsigned char sym=' ';   // символ з введеного тексту
  int n;   // ціле число
  printf("Введіть текст з числами (кінець тексту #)\n");
```

```
  while (sym!='#') {
    // пропуск символів, відмінних від цифри, доки символ не #
    while (((sym=getchar()) <'0' || sym>'9') && sym!='#');
    if (sym!='#') {
      // формування числа
      n=0;
      while (sym>='0' && sym<='9') {
        n=n*10+sym-'0';
        sym=getchar();
      }
      printf("\t%d\n",n);
    }
  }
...
```

```
Введіть текст з числами (кінець тексту #)
Абвг12 ґдеєжз    134и
        12
        134
іїйклм7896ноп45 рстуф67 45 221
        7896
        45
        67
        45
        221
хцчшщ55 юяь# далі не аналізує123
        55
```

Приклад 6.3.16. Визначення кількості одиниць у двійковому поданні цілого числа.

а) Найпростіше визначити кількість одиниць у двійковому поданні числа — це перевіряти, чи не є остання цифра подання одиницею, і зсувати двійковий код числа вправо на один біт. Ці дії треба виконувати, доки не буде одержано число нуль.

Мовою С алгоритм можна подати так:

```
/* Підрахунок кількості 1 у двійковому поданні числа */
...
  long long int x;   // задане число
  int n;   // кількість одиниць
  while (printf("  Ведіть x "), scanf("%lld",&x)) {
    fseek(stdin,0,SEEK_END);   // очистка буфера
    n=0;
    while (x) {
      if (x&1)
        n++;
      x=x>>1;
```

```
  Ведіть x 214
Кількість 1 у двійковому поданні x - 5
  Ведіть x 31
Кількість 1 у двійковому поданні x - 5
  Ведіть x 8
Кількість 1 у двійковому поданні x - 1
  Ведіть x
кінець
```

```
    }
    printf("Кількість 1 у двійковому поданні x - %d\n",n);
  }
...
```

Відповідно до цього алгоритму тіло циклу буде виконуватися стільки разів, скільки значущих цифр має його двійкове подання.

б) Існує інший алгоритм визначення кількості одиниць у двійковому поданні числа, який полягає в підрахункові кількості проходів циклу знаходження порозрядної кон’юнкції чисел x і $x-1$ (результат треба присвоїти x), доки x не стане рівним нулю.

Мовою С алгоритм можна подати так:

```
  /* Підрахунок кількості 1 у двійковому поданні числа */
...
    n=0;
    while (x) {
      x=x&(x-1);
      n++;
    }
...
```

```
  Ведіть x 1000
Кількість 1 у двійковому поданні x - 6
  Ведіть x 7
Кількість 1 у двійковому поданні x - 3
  Ведіть x 9
Кількість 1 у двійковому поданні x - 2
  Ведіть x
  кінець
```

Використовуючи поданий алгоритм, знайдемо, наприклад, кількість одиниць у двійковому коді числа $134_{10} = 10000110_2$.

Початкові значення: x=10000110; n=0.

I прохід циклу: $x-1=10000101$;

$$\begin{array}{r} \& \begin{array}{r} 10000110 \\ 10000101 \end{array} \\ \hline x = \quad 10000100 \end{array} \qquad n = 1;$$

II прохід циклу: $x-1=10000011$;

$$\begin{array}{r} \& \begin{array}{r} 10000100 \\ 10000011 \end{array} \\ \hline x = \quad 10000000 \end{array} \qquad n = 2;$$

III прохід циклу: $x-1=01111111$;

$$\begin{array}{r} \& \begin{array}{r} 10000000 \\ 01111111 \end{array} \\ \hline x = \quad 00000000 \end{array} \qquad n = 3;$$

Оскільки x=0, то алгоритм закінчує роботу і результатом обчислень є значення $n=3$.

У середньому, порівняно з попереднім, цей алгоритм збігається швидше, оскільки потребує кількості проходів циклу, рівної кількості одиниць у двійковому поданні числа. Проте в найгіршому випадку, коли в двійковому поданні всі розряди містять одиниці, кількість проходів циклів буде однаковою.

Приклад 6.3.17. Визначення за введеною датою номера дня в році (шляхом додавання кількостей днів за попередні місяці).

При обчисленнях треба врахувати, що високосним є рік, номер якого кратний 4, але не кратний 100, але кратний 400.

Програмно алгоритм визначення номера дня в році можна подати так:

```
/* Визначення номера дня в році.
   У циклі вводяться дати за форматом дд.мм.рррр, доки не
   буде введено будь-який символ, відмінний від цифри */
...
  int d, m=5, r=1959;    // введена інформація
  int k,    //  номер дня в році
      mr;   // робоча змінна - номер попереднього місяця
  printf("Вводьте дати за форматом дд.мм.рррр\
    \nДля закінчення – букву\n");
  while (scanf("%d.%d.%d", &d,&m,&r)){
    while (getchar()!='\n');    // очистка буфера
    /* кількість днів за попередні місяці */
    k=0; mr=1;
    while (mr<m) {
      switch (mr) {
        case 4: case 6: case 9: case 11:
                k=k+30; break;
        case 2: if (r%4==0 && (r%100!=0 || r%400==0))
                  k=k+29;
                else
                  k=k+28;
                break;
        default: k=k+31;
      }
```

```
Вводьте дати за форматом дд.мм.рррр
Для закінчення – букву
30.5
   Номер дня в році - 150 (30.05.1959)
30.5.1900
   Номер дня в році - 150 (30.05.1900)
30.5.2000
   Номер дня в році - 151 (30.05.2000)
кінець
```

```
        mr++;
      }
      /* одержання результату */
      k=k+d;
      printf("   Номер дня в році - %3d (%02d.%02d.%04d)\n",k,d,m,r);
    }
...
```

Як було з'ясовано раніше, дійсні числа з дробовою частиною, не кратною степеневі двійки, зберігаються в пам'яті комп'ютера наближено (особливості роботи з дійсними числами розглянуто в прикладах 3.4.6-3.4.11). Неточність подання дійсних чисел у комп'ютері пов'язана як з тим, що при переведенні чисел з десяткової системи числення у двійкову дроби, не кратні степеневі двійки, стають періодичними, так і з заокругленням чисел через обмеженість розрядної сітки при зберіганні. Також, оскільки дійсні числа подаються в комп'ютері неточно (див. п. 4.2), то замість порівняння $x = y$ треба застосовувати порівняння $|x - y| \le eps$, де *eps* — невелике число, яке вказує на допустиму похибку (в загальному випадку *eps* залежить від *x* і *y*).

Приклад 6.3.18. Табулювання функції $f(x) = \dfrac{\sqrt{3\ln x + a}}{\sin(2x - b)}$ на проміжку $[x_1; x_2]$ з кроком *h*.

Якщо табулювання функції почати з точки x_1 і крок табулювання *h* вибрати, наприклад, рівним 0,1 (або якомусь іншому дробовому числу, не кратному степеневі двійки), то одержувані значення *x* з проміжку $[x_1; x_2]$ будуть не точними і при підході до точки x_2 може бути одержано значення трохи менше або трохи більше від x_2. Тому умовою виконання циклу має бути не $x \le x_2$, а умова $x \le x_2 + eps$.

Так, за умови $x \le x_2$ при $x_1 = 1$, $x_2 = 2$ (а також, якщо x_2 набуває значень 3, 4, від 19 до 75 та інших) і $h = 0{,}1$ значення $x = 2$ (а також 3, 4, від 19 до 75 та інші) у таблицю не виводиться, оскільки за рахунок неточності обчислень останнє значення при додаванні кроку стало трохи більшим від 2. Крім того,

навіть, якщо перевіряти умову $x \le x_2 + eps$, але значення *eps* взяти досить малим, наприклад, 10^{-14}, то при $x_1 = 1$, $x_2 = 20$ і $h = 0{,}1$ значення $x = 20$ у таблицю не виводиться. Для обчислень візьмемо, наприклад, $eps = 10^{-9}$.

У зв'язку з цією ж самою причиною (неточністю подання дробових чисел у комп'ютері) при перевірці входження значення *x* в ОДЗ замість, наприклад, порівняння $\sin(2x - b) = 0$ треба виконати перевірку умови $|\sin(2x - b)| \le eps$.

Проте, якщо значення x_1, x_2 і *h* не матимуть дробових частин чи матимуть дробові частини, кратні степеневі двійки, то ніяких неточностей не виникатиме і порівняння з використанням числа *eps* не потрібні (крім значень тригонометричних функцій, оскільки ірраціональне число π у комп'ютері теж точно не подається).

Функція $f(x) = \dfrac{\sqrt{3\ln x + a}}{\sin(2x - b)}$ буде невизначеною у трьох випадках: підлогарифмічний вираз недодатний, підкореневий вираз від'ємний, у знаменнику нуль. З врахуванням неточності обчислень при роботі з дійсними числами умова порушення ОДЗ з врахуванням порядку обчислень запишеться так: $(x < eps) \vee (3\ln x + a < 0) \vee (|\sin(2x - b)| <= eps)$. Тут підлогарифмічний вираз порівнюється не з 0, а з eps, оскільки при дуже малих значеннях аргументу значення логарифма дуже швидко прямує до мінус нескінченності.

Програмна реалізація алгоритму табулювання функції:

```
/* Табулювання функції f(x) на проміжку [x1; x2] з кроком h */
...
  double x1,x2,h,    // границі проміжку і крок
    a,b,x,      // параметри a і b і змінна x
    y;          // обчислене значення функції
  double eps=1e-9;   // точність обчислень
...
  /* Проведення обчислень і заповнення таблиці */
  x=x1;
```

```
  while (x<=x2+eps) {
    printf("|%6.2f",x);
    if (x<eps || 3*log(x)+a<0 || fabs(sin(2*x-b))<=eps )  // функція
                                                      // не визначена
      printf(" |  не визн ");
    else {   // функція визначена - обчислення її значення
      y=sqrt(3*log(x)+a)/sin(2*x-b);
      if (fabs(y)>100)   // переповнення (велике число)
        printf(" |   переп  ");
      else   // виведення обчисленого значення
        printf(" | %8.2f ",y);
    }
...
    x+=h;   // перехід до наступної точки проміжку
  }
...
```

```
Табулювання функції:
        f(x)=sqrt(3*ln(x)+a)/sin(2x-b)
        Точність обчислень eps=1e-009
Вкажіть кількість колонок таблиці (від 1 до 4): 4
Вкажіть
        проміжок x1, x2: -0.2 6
        крок h: 0.1
        значення параметрів a i b: 3 2
```

x	f(x)	x	f(x)	x	f(x)	x	f(x)
-0.20	не визн	-0.10	не визн	0.00	не визн	0.10	не визн
0.20	не визн	0.30	не визн	0.40	-0.54	0.50	-1.14
0.60	-1.69	0.70	-2.46	0.80	-3.92	0.90	-8.25
1.00	не визн	1.10	9.12	1.20	4.84	1.30	3.45
1.40	2.79	1.50	2.44	1.60	2.25	1.70	2.17
1.80	2.18	1.90	2.28	2.00	2.48	2.10	2.83
2.20	3.43	2.30	4.55	2.40	7.08	2.50	16.99
2.60	-41.49	2.70	-9.57	2.80	-5.58	2.90	-4.07
3.00	-3.32	3.10	-2.90	3.20	-2.68	3.30	-2.58
3.40	-2.59	3.50	-2.71	3.60	-2.96	3.70	-3.41
3.80	-4.19	3.90	-5.73	4.00	-9.58	4.10	-32.37
4.20	23.19	4.30	8.72	4.40	5.52	4.50	4.17
4.60	3.47	4.70	3.08	4.80	2.87	4.90	2.79
5.00	2.83	5.10	2.99	5.20	3.30	5.30	3.85
5.40	4.85	5.50	6.91	5.60	12.82	5.70	переп
5.80	-16.50	5.90	-7.87	6.00	-5.32		

Якщо в обчисленнях з заданими в прикладі значеннями вхідних даних взяти величину $eps = 0$, то при $x = 1$ буде видано повідомлення не про те, що функція «не визначена», а про «переповнення» (при цьому x буде меншим від 1 на $1{,}1 \cdot 10^{-16}$ і результатом буде число з 16-значною цілою частиною -7800463371553962.00).

6.4. Складність і оптимізація циклічних алгоритмів

Циклічний алгоритм може містити кілька операторів циклу, які виконуються послідовно або входять в інші оператори. Найпростіші циклічні алгоритми містять один цикл, а також лінійні фрагменти і розгалуження. У свою чергу цикл може містити лінійні фрагменти і розгалуження. Обчислення складності лінійних фрагментів і розгалужень розглядалося в попередніх розділах.

Складність $t_ц$ циклу з параметром можна обчислити за формулою:

$$t_ц = n \cdot t_т,$$

де n — кількість повторень циклу, яка при крокові 1 визначається як різниця верхньої і нижньої межі циклу плюс 1; $t_т$ — складність тіла циклу, яка є сумою всіх складностей виконуваних конструкцій циклу, які в свою чергу складаються зі складності виконання операторів тіла циклу і складності обчислення умови завершення циклу. Якщо оцінювати складність циклів з передумовою і післяумовою, то можна взяти статистичну оцінку кількості повторень циклу в гіршому і в середньому випадках. Отже, часова складність виконання циклу є величиною $O(n)$. Складність виконання кожного циклу, якщо в алгоритмі їх є кілька, треба визначати окремо.

Оскільки $t_ц = n \cdot t_т$, то оптимізація циклічного алгоритму за часом полягає в зменшенні складності тіла циклу і, якщо це можливо, зменшенні кількості повторень циклу.

Для оптимізації тіла циклу використовують такі прийоми:

— попереднє обчислення поза циклом підвиразів, значення яких не змінюються в циклі;

— використання співвідношень, що зв'язують змінні, які змінюються в циклі, для обчислень однієї зі змінних через інші;

— застосування *мемоїзації* (одного з прийомів методу динамічного програмування, який дає можливість розв'язати складні задачі через розкладання їх на простіші підзадачі) для виключення повторного обчислення результатів попередніх проходів циклів, тобто використання результату попередніх ітерацій циклу для обчислення значень на наступних ітераціях.

Приклад 6.4.1. При обчисленні наближеного значення розкладеної в степеневий ряд показникової функції $a^x = \sum_{n=0}^{\infty} \frac{(\ln a)^n x^n}{n!}$ поза циклом треба обчислити значення логарифму $\ln a$, яке не залежить від номера ітерації. Крім того, можна побачити, що кожний наступний член ряду обчислюється через попередній за таким рекурентним співвідношенням $a_n = \frac{\ln a \cdot x}{n} a_{n-1}$ при $a_0 = 1$; тому поза циклом можна обчислити не тільки $\ln a$, а й добуток $\ln a \cdot x$.

Таким чином, одержимо алгоритм наближеного обчислення значення функції a^x (обчислення продовжуються, доки загальний член ряду більший від деякого досить малого числа *eps* і заданого ліміту кількості ітерацій *k*):

```
/* Обчислення наближеного значення степеневого ряду з
   заданою точністю (функція a^x) */
...
  n=0;
  lnax=log(a)*x;
  an=1;
  s=an;
  do {
    n++;
    an=lnax/n*an;
    s=s+an;
  } while (fabs(an)>=eps && n<k);
...
```

```
  введіть x 3
  введіть a 2
  введіть eps 0.000001
  введіть k 100
результат:
  кількість ітерацій n= 14
  наближене значення функції s= 7.999999948
  точне значення функції f= 8.000000000
  похибка |f-s|= 0.000000052
```

У цьому алгоритмі умовою продовження роботи циклу є не тільки порівняння `fabs(an)>=eps` — абсолютне значення загального члена ряду більше чи рівне деякому невеликому числу *eps*, яке вказує на допустиму похибку (точність), а й порівняння `n<k` — чи не перевищує кількість ітерацій деякого наперед заданого числа *k*. Число *k* — ліміт кількості кроків використовується для підстраховки від зациклювання, яке може виникнути через некоректні вхідні дані — значення змінної *x* і точності *eps*.

Приклад 6.4.2. При обчисленні наближеного значення функції $\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$, розкладеної в степеневий ряд, можна побачити, що кожний наступний член ряду обчислюється через попередній за таким рекурентним співвідношенням $a_n = \frac{-x^2}{2 \cdot n \cdot (2n+1)} a_{n-1}$ при $a_0 = x$. Множник $\frac{-x^2}{2}$, який входить у рекурентне співвідношення для a_n, від *n* не залежить, тому його значення можна обчислити поза циклом і використовувати в циклі; крім того, цей множник також на кожній ітерації буде визначати знак доданка (тобто окремо множити на –1 не треба).

Маємо алгоритм наближеного обчислення значення функції $\sin x$:

```
/* Обчислення наближеного значення степеневого ряду
   (функція sin(x)) з заданою точністю eps і обеженням k
   кількості ітерацій */
...
  n=0;
  an=x;
  s=an;
  r=-x*x/2.;
  do {
    n++;
    an*=r/n/(2.*n+1.);
    s+=an;
  } while (fabs(an)>=eps && n<k);
...
```

```
Функція  sin(x)
ряд E(n=0..+00)(-1)^n*(x^(2n+1))/((2n+1)!)
Введіть значення x 1.57
  введіть eps 1e-5
  введіть ліміт кількості ітерацій k 100
результат:
  кількість ітерацій n= 5
  наближене значення функції s= 0.999999627
  точне значення функції f= 0.999999683
  похибка |f-s|= 0.000000056
```

Також треба мати на увазі, що, наприклад, якщо обчислюється сума n членів арифметичної чи геометричної прогресії, то замість організації циклу треба скористатися відповідною формулою: $\sum_{i=1}^{n} a_i = \frac{a_1 + a_n}{2} \cdot n = \frac{2a_1 + d \cdot (n-1)}{2} n$, де $a_n = a_{n-1} + d = a_1 + d \cdot (n-1)$; зокрема, $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$; $\sum_{i=1}^{n} b_i = \frac{b_1 - b_{n+1}}{1-q} = b_1 \frac{1-q^n}{1-q}$, де $b_n = b_{n-1} \cdot q = b_1 \cdot q^{n-1}$. Якщо обчислюються значення комбінаторних формул, то окремо знаходити факторіали не потрібно, а треба спочатку виконати скорочення. Наприклад: $C_n^2 = \frac{n!}{2!(n-2)!} = \frac{n \cdot (n-1)}{2}$ (детальніше див. п. 10.1).

Якщо в алгоритмі використовуються вкладені цикли, і кожен з них має по n проходів, то складність виконання алгоритму буде оцінюватися як $O(n^2)$.

Крім виконання операторів тіла циклу і обчислення умови завершення циклу, також треба враховувати, що перед початком роботи кожного циклу, наприклад циклу з параметром, виконується ініціалізація — обробка заголовка для визначення початкового і кінцевого значень параметра і кроку приросту параметра, що потребує певного часу. Таким чином, якщо є вкладені цикли і вони виконуються багато разів, то має значення порядок проходження вкладених циклів.

Приклад 6.4.3. Розглянемо вкладені цикли:

```
for (i=1; i<=10000; i++)
  for (j=1; j<=100; j++) {
    // тіло циклу
  }
```

У цьому випадку зовнішній цикл ініціалізується 1 раз, а внутрішній 10000 разів, тобто виконується 10001 ініціалізація.

Якщо поміняти місцями заголовки внутрішнього і зовнішнього циклів:

```
for (j=1; j<=100; j++)
  for (i=1; i<=10000; i++){
    // тіло циклу
  }
```

то внутрішній цикл буде ініціалізуватися тільки 100 разів і в результаті при виконанні вкладених циклів буде виконано 101 ініціалізацію.

Таким чином, при програмуванні вкладених циклів, якщо це можливо, треба цикл з найбільшою кількістю повторень робити найвнутрішнішим, а цикл з найменшою кількістю повторень — найзовнішнішим.

Циклічні алгоритми мають досить велику часову складність і тому при розробці і програмуванні цих алгоритмів треба враховувати і застосовувати всі можливі способи їхньої оптимізації.

Запитання та завдання для самоконтролю

1. Що таке цикл? Для чого використовують цикли? Що називають тілом циклу? Чи може бути тіло циклу порожнім? Які алгоритми називають циклічними? Схарактеризуйте детерміновані й ітераційні цикли. Яку ситуацію називають зациклюванням?

2. Яких типів є оператори циклу і чи є між ними відмінності при виконанні? У яких випадках надають перевагу циклу з параметром, циклу з передумовою чи циклу з післяумовою? Чи можуть бути вкладеними один в оден цикли різних типів? Які вкладені цикли називають залежними, а які незалежними?

3. Що таке «машинний епсилон» і «машинний нуль»? Чи має в певних випадках значення порядок доданків і чому? Для чого використовують метод дотичних Ньютона? Що є умовою виходу з циклу ітерацій у методі Ньютона? Як реалізовано обчислення чисел Фібоначчі за рекурентною формулою другого порядку? Як одержати окремі цифри числа? Подільність на які числа треба перевіряти, щоб виявити, чи є число простим? Які є варіанти алгоритму Евкліда і яка їхня обчислювальна складність? Де застосовують алгоритм Евкліда? Як за текстовим поданням числа одержати число? Як перевірити умову виконання циклу і рівність значень при роботі з дійсними числами?

4. Як визначити часову складність циклів з параметром, з передумовою і післяумовою? Як оптимізувати виконання циклічного алгоритму за часом?

Практичне завдання № 8 «Проектування алгоритмів циклічної структури. Цикли з параметрами»

Мета виконання завдання: навчитися розробляти алгоритми і програмувати циклічні процеси, використовуючи оператор циклу з параметром, і оформляти результати у вигляді таблиць.

Завдання. Розробити алгоритм розв'язування задачі: перші m ($m \geq 1$) членів послідовності $\{x_n\}$ ($n = 1, 2, 3, \ldots$) видати на екран у таблиці по рядках з k ($1 \leq k \leq 6$) колонками; визначити умову переходу на новий рядок в таблиці при будь-якому k; врахувати, що може бути введено $m < k$; якщо результати повністю не заповнюють останній рядок таблиці, то доповнити таблицю порожніми клітинками. В алгоритмі передбачити перевірку правильності введення даних (до першої помилки).

Алгоритм подати програмно і за допомогою узагальненої блок-схеми (без пооператорної деталізації, але в ній зобразити цикл, у якому обчислюються значення членів послідовності). Текст програми структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з 4 різними значеннями m і k.

Вказівки. Зразки використання циклів з параметрами подано в прикладі 6.2.8 та інших. Результати роботи програми у випадку безпомилкового введення даних мають бути такими:

```
Загальний член послідовності:   xn=cos(5n-2)/(3n-1)

Вкажіть кількість членів послідовності (не менше 1) = 7
Вкажіть кількість колонок (від 1 до 6) = 3
-----------------------------------
| -0.49500 | -0.02910 |  0.11343 |
|  0.06003 | -0.03806 | -0.05662 |
| -0.00066 |          |          |
-----------------------------------
Результат видано по рядках
```

Варіанти завдань

1. $x_n = \dfrac{\ln(3n) + 5}{2n + 5}$ 2. $x_n = \dfrac{\sin(2n + 3) - 2}{n + 1}$ 3. $x_n = \dfrac{\sin(n - 1)}{n^2 + 2}$

4. $x_n = \dfrac{\cos(2n-5)+n}{4n+1}$ 5. $x_n = \dfrac{\cos(n-1)}{n^2+1}$ 6. $x_n = \dfrac{\sin(3n-2)}{n+3}$

7. $x_n = \dfrac{\sin n^2 - 5}{3n^2+n}$ 8. $x_n = \dfrac{\cos(n^2-5)+4}{3n^2+1}$ 9. $x_n = \dfrac{\ln(n^2+4)}{5n^2-n}$

10. $x_n = \dfrac{\cos(n+2)}{n^2+3}$ 11. $x_n = \dfrac{\ln(n+1)}{5n^2+3}$ 12. $x_n = \dfrac{\cos(n-1)}{n^2+1}$

13. $x_n = \dfrac{\ln(3n)+5}{n^2}$ 14. $x_n = \dfrac{\sin(n-1)+4}{n^2+n}$ 15. $x_n = \dfrac{\ln(5n)+1}{2n+3}$

16. $x_n = \dfrac{\cos(2n)+5}{3n^2+1}$ 17. $x_n = \dfrac{\sin(n^2)+n}{3n^2-1}$ 18. $x_n = \dfrac{\ln(3n)-1}{2n^2+3}$

19. $x_n = \dfrac{\cos(n-1)}{n^2+4n}$ 20. $x_n = \dfrac{\ln(n+1)}{3n^2-1}$ 21. $x_n = \dfrac{\sin(5n^2)+1}{3+n}$

22. $x_n = \dfrac{\cos(7n)+4}{2n-3}$ 23. $x_n = \dfrac{\sin(3n-2)}{n^2+n}$ 24. $x_n = \dfrac{\ln(5+n)}{3n-2}$

25. $x_n = \dfrac{\cos(2n)-4}{2n+3}$ 26. $x_n = \dfrac{\sin(5n-3)+n}{2n+3}$ 27. $x_n = \dfrac{\ln(n^2+1)}{3n^2}$

28. $x_n = \dfrac{\cos(n+3)-2}{2n+1}$ 29. $x_n = \dfrac{\cos(2n+1)-4}{5n+1}$ 30. $x_n = \dfrac{\sin(2n+3)+4}{n}$.

Практичне завдання № 9 «Проектування алгоритмів з вкладеними циклами. Робота з дійсними числами»

Мета виконання завдання: навчитися розробляти алгоритми і програмувати вкладені циклічні процеси, використовуючи оператори циклу різних видів; навчитися працювати з даними дійсного типу.

Завдання. Розробити алгоритм обчислення значення функції $f(x)$, де аргумент x змінюється на довільно заданому проміжку $[x_1, x_2]$ з кроком h. Результат видати на екран у таблиці з k ($k = \overline{1,4}$) парами колонок $x, f(x)$; заповнену не до кінця таблицю доповнити порожніми клітинками. Якщо функція при якомусь значенні аргументу не визначена, то в таблиці на місці результату подати «*не визн*» (при цьому суму не рахувати). Якщо функція набуває значення, більшого від якогось великого числа (наприклад, 9999,99), то на місці результату подати «*переп*» (переповнення). В алгоритмі передбачити перевірку правильності введення даних (кількості колонок таблиці, границь

проміжку і величини кроку), доки дані не будуть введені правильно. При введенні даних очищати буфер від зайвої інформації.

Алгоритм подати програмно. Текст програми структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з трьома різними наборами вхідних даних.

Вказівки. При обчисленні значення функції як зразок використати алгоритм з прикладу 6.3.18, врахувати, що робота з дійсними числами має певні особливості (див. приклади 3.4.6-3.4.11), а також, що при порівнянні дійсних чисел строгої рівності не може бути (див. п. 4.2). При обчисленні суми, яка є частиною заданої функції, застосувати мемоїзацію (див. п. 6.4) — у додаткову змінну накопичувати значення степеня, а не організовувати додатковий цикл для повного його обчислення. При введенні даних використати алгоритми з прикладів 6.2.4 і 6.2.5; щодо очищення буфера вводу див. приклад 6.2.7. Для побудови таблиці адаптувати алгоритм з попереднього практичного завдання № 8.

Варіанти завдань

1. $f(x)=\sqrt{\dfrac{x+2}{x^2-1}}\sum\limits_{n=2}^{6}\dfrac{3^{n+1}+x}{n}-x\ln|x-5|$

2. $f(x)=\dfrac{\ln(x-2)+2}{x-3}\sum\limits_{n=2}^{3}\dfrac{5^n+x}{4n+3}+\sqrt{\ln|x-9|}$

3. $f(x)=\dfrac{1-x}{1-\sqrt{x+2}}\sum\limits_{n=0}^{8}\dfrac{2^n+x}{2n+1}-5\ln|x-4|$

4. $f(x)=\dfrac{2-\ln(x+4)}{x+2}\sum\limits_{n=2}^{4}\dfrac{3^n+x}{3n-1}+4\sqrt{\ln|x|}$

5. $f(x)=\dfrac{\sqrt{x+2}}{x-3}\sum\limits_{n=1}^{5}\dfrac{4^n-5x}{n^2+n+1}+\ln(x^2-1)$

6. $f(x)=\dfrac{\ln(x+1)+3}{2-x}\sum\limits_{n=2}^{7}\dfrac{2^n+x}{3n-1}+\sqrt{3-x}$

7. $f(x)=\sqrt{\dfrac{x+2}{x-1}}\sum\limits_{n=1}^{5}\dfrac{2^n-x}{n^2+1}+\ln(x+5)$

8. $f(x)=\dfrac{\ln(x-5)}{x-1}\sum\limits_{n=2}^{4}\dfrac{3^n-x}{n+5}+2\sqrt{\ln|x-4|}$

9. $f(x)=\dfrac{1-2x}{2-\sqrt{x+3}}\sum\limits_{n=0}^{5}\dfrac{2^n+x}{n+2}-6\ln|x-3|$

10. $f(x)=\dfrac{3+\ln(x+3)}{3-x}\sum\limits_{n=1}^{7}\dfrac{2^{n+1}+x}{n^2}+\sqrt{x^2-1}$

11. $f(x)=\sqrt{\dfrac{x^2-2}{x-3}}\sum\limits_{n=2}^{5}\dfrac{4^n+x}{4n+3}+7\ln|x-7|$

12. $f(x)=\dfrac{\ln x-2}{x-1}\sum\limits_{n=1}^{4}\dfrac{5^n+x}{n+3}+\sqrt{\ln|x-7|}$

13. $f(x)=\sqrt{\dfrac{x-1}{x-2}}\sum\limits_{n=2}^{7}\dfrac{3^{n+1}+x}{n-1}+\ln|x|$

14. $f(x)=\dfrac{\ln(x+5)}{x+2}\sum\limits_{n=0}^{6}\dfrac{2^n+x}{n+3}+\sqrt{\ln|x-3|}$

15. $f(x)=\sqrt{\dfrac{x+3}{x^2-1}}\sum\limits_{n=0}^{6}\dfrac{2^n+3x}{n+2}-\ln|x-3|$

16. $f(x)=\dfrac{\ln x-4}{x-2}\sum\limits_{n=1}^{3}\dfrac{4^n-x}{n+5}-\sqrt{\ln|x-6|}$

17. $f(x)=\dfrac{2x-1}{1-\sqrt{x+1}}\sum\limits_{n=1}^{4}\dfrac{5^n+x}{(n+1)^2}-\ln((x-1)^2)$

18. $f(x)=\dfrac{\ln(x+2)}{5-x}\sum\limits_{n=1}^{4}\dfrac{3^n+x}{2n-1}+\sqrt{\ln|x|}$

19. $f(x)=\sqrt{\dfrac{x+1}{x-3}}\sum_{n=2}^{5}\dfrac{3^n+x}{n-1}+\ln(x+4)$

20. $f(x)=\dfrac{\ln(x+2)+4}{x-3}\sum_{n=1}^{4}\dfrac{3^n+x}{4n-1}+\sqrt{7-x}$

21. $f(x)=\sqrt{\dfrac{x-3}{x^2-4}}\sum_{n=0}^{3}\dfrac{6^n-x}{n+1}+\ln|10-x|$

22. $f(x)=\dfrac{\ln(x-1)}{4-x}\sum_{n=0}^{6}\dfrac{3^n-2x}{n+2}+\sqrt{8-x}$

23. $f(x)=\dfrac{x+1}{3-\sqrt{x^2-1}}\sum_{n=3}^{5}\dfrac{4^n+x}{n^2+1}-4\ln(x+5)$

24. $f(x)=\dfrac{\ln(x+2)}{x-3}\sum_{n=1}^{5}\dfrac{3^n+x}{n}+\sqrt{x^4-1}$

25. $f(x)=\dfrac{\ln x+3}{x-1}\sum_{n=3}^{5}\dfrac{2^n+x}{n-2}+\sqrt{20-x}$

26. $f(x)=\dfrac{2-x}{1-\sqrt{x+3}}\sum_{n=2}^{5}\dfrac{4^n+x}{n^2}-\ln|x-1|$

27. $f(x)=\dfrac{\ln(x+3)}{x-7}\sum_{n=2}^{5}\dfrac{4^n+x}{4n-1}+\sqrt{x^2-2}$

28. $f(x)=\dfrac{1-5x}{\sqrt{x+1}-2}\sum_{n=2}^{4}\dfrac{4^n+x}{(n-1)^2}-\ln(x^2)$

29. $f(x)=\sqrt{\dfrac{x-2}{x+1}}\sum_{n=0}^{4}\dfrac{3^n+x}{n+1}-\ln(x+4)$

30. $f(x)=\dfrac{\ln(x+4)}{x-3}\sum_{n=2}^{5}\dfrac{4^n+x}{4n-1}+\sqrt{|x|-1}$

Практичне завдання № 10
«Проектування циклічних ітераційних алгоритмів»

Мета виконання завдання: навчитися розробляти ітераційні алгоритми і програмувати їх з використанням циклів з післяумовою.

Завдання. Розробити алгоритм обчислення значення функції $f(x)$, розкладеної в степеневий ряд. Обчислення суми членів ряду проводити доти, доки абсолютна величина члена ряду не стане меншою від деякого досить малого числа ε (наприклад, $\varepsilon=10^{-6}$ чи ще меншого). При цьому порахувати, скільки членів ряду ввійшло в суму. Крім того, для підстраховування від зациклювання, яке може виникнути через некоректні вхідні дані, встановити ліміт кількості кроків. Якщо вихід із циклу відбудеться через вичерпання ліміту, то видати про це повідомлення. Порівняти (знайти абсолютне значення різниці) наближене значення функції, обчислене з використанням ряду, зі значенням, обчисленим за формулою функції. В алгоритмі передбачити перевірку правильності введення даних, доки дані не будуть введені правильно. При введенні даних очищати буфер від зайвої інформації. Передбачити можливість багаторазового виконання алгоритму. Як результат роботи видати: обчислене наближене значення функції, кількість кроків ітерації, обчислене за формулою значення функції, абсолютну різницю наближеного і «точного» значень функції.

Алгоритм подати програмно. Текст програми структурувати. Коментарі обов'язкові — 17-25 %; у коментарях пояснити призначення кожної змінної. Провести розрахунки не менш ніж з трьома різними наборами вхідних даних.

Вказівки. При обчисленні значення функції як зразки використати алгоритми з прикладів 6.4.1, 6.4.2. Для обчислення членів ряду одержати рекурентну формулу і застосувати прийом мемоїзації — використовувати попередній член ряду чи його частину для обчислення поточного, а не організовувати додатковий цикл. При введенні даних використати алгоритми з прикладів 6.2.4 і 6.2.5; щодо очищення буфера вводу див. приклад 6.2.7; для багаторазового виконання алгоритму застосувати алгоритм з діалогом з прикладу 6.2.6. Якщо для функції $f(x)$ не вказано проміжок збіжності, то при розрахунках брати проміжок $[-10,10]$.

Варіанти завдань

1. $shx=\dfrac{e^x-e^{-x}}{2}=\sum\limits_{n=0}^{\infty}\dfrac{x^{2n+1}}{(2n+1)!}$ 2. $\dfrac{1}{\sqrt{1-4x}}=\sum\limits_{n=0}^{\infty}\dfrac{(2n)!x^n}{(n!)^2}$ при $x\in[-0{,}25;0{,}25)$

3. $\dfrac{(1-4x)\sqrt{1-4x}+6x(1-x)-1}{12}=\sum\limits_{n=0}^{\infty}\dfrac{(2n+2)!x^{n+3}}{(n+1)!(n+3)!}$ при $x\in[-0{,}25;0{,}25]$

4. $chx-1=\dfrac{e^x+e^{-x}}{2}-1=\sum\limits_{n=1}^{\infty}\dfrac{x^{2n}}{(2n)!}$ 5. $x\cos 3x=\sum\limits_{n=0}^{\infty}\dfrac{(-1)^n 3^{2n}x^{2n+1}}{(2n)!}$

6. $\dfrac{1}{(1-4x)\sqrt{1-4x}}-1-6x=\sum\limits_{n=2}^{\infty}\dfrac{(2n+1)!x^n}{(n!)^2}$ при $x\in[-0{,}25;0{,}25)$ 7. $e^x-1=\sum\limits_{n=0}^{\infty}\dfrac{x^{n+1}}{(n+1)!}$

8. $\ln\dfrac{2}{1+\sqrt{1-4x}}=\sum\limits_{n=1}^{\infty}\dfrac{(2n-1)!x^n}{(n!)^2}$ при $x\in[-0{,}25;0{,}25]$

9. $\dfrac{1}{(1-4x)\sqrt{1-4x}}=\sum\limits_{n=0}^{\infty}\dfrac{(2n+1)!x^n}{(n!)^2}$ при $x\in[-0{,}25;0{,}25)$ 10. $\cos x=\sum\limits_{n=0}^{\infty}\dfrac{(-1)^n x^{2n}}{(2n)!}$

11. $\dfrac{(1-4x)\sqrt{1-4x}+6x-1}{12}=\sum\limits_{n=0}^{\infty}\dfrac{(2n)!x^{n+2}}{n!(n+2)!}$ при $x\in[-0{,}25;0{,}25]$

12. $\dfrac{1}{2}-x-\dfrac{\sqrt{1-4x}}{2}=\sum\limits_{n=1}^{\infty}\dfrac{(2n)!x^{n+1}}{(n!)^2(n+1)}$ при $x\in[-0{,}25;0{,}25)$ 13. $chx=\dfrac{e^x+e^{-x}}{2}=\sum\limits_{n=0}^{\infty}\dfrac{x^{2n}}{(2n)!}$

14. $\dfrac{1-2x-\sqrt{1-4x}}{4}=\sum\limits_{n=0}^{\infty}\dfrac{(2n+1)!x^{n+2}}{n!(n+2)!}$ при $x\in[-0{,}25;0{,}25]$

15. $\dfrac{1}{2}-\dfrac{\sqrt{1-4x}}{2}=\sum\limits_{n=0}^{\infty}\dfrac{(2n)!x^{n+1}}{(n!)^2(n+1)}$ при $x\in[-0{,}25;0{,}25)$ 16. $\cos x-1=\sum\limits_{n=1}^{\infty}\dfrac{(-1)^n x^{2n}}{(2n)!}$

17. $\ln 2-\ln(1+\sqrt{1-4x})=\sum_{n=0}^{\infty}\frac{(2n+1)!x^{n+1}}{((n+1)!)^2}$ при $x\in[-0,25;0,25]$

18. $1+(x-1)e^x-\frac{x^2}{2}=\sum_{n=1}^{\infty}\frac{(n+1)x^{n+2}}{(n+2)!}$ 19. $shx-x=\frac{e^x-e^{-x}}{2}-x=\sum_{n=1}^{\infty}\frac{x^{2n+1}}{(2n+1)!}$

20. $1+(x-1)e^x=\sum_{n=0}^{\infty}\frac{(n+1)x^{n+2}}{(n+2)!}$ 21. $\frac{e^x-e^{-x}-2\sin x}{4}=\sum_{n=0}^{\infty}\frac{x^{4n+3}}{(4n+3)!}$

22. $x\cos x=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n+1}}{(2n)!}$ 23. $(x-1)e^x+1=\sum_{n=0}^{\infty}\frac{x^{n+2}}{n!(n+2)}$

24. $\frac{1-\sqrt{1-4x}}{2}=\sum_{n=0}^{\infty}\frac{(2n)!x^{n+1}}{n!(n+1)!}$ при $x\in(-0,25;0,25)$ 25. $\frac{e^x-e^{-x}+2\sin x}{4}=\sum_{n=0}^{\infty}\frac{x^{4n+1}}{(4n+1)!}$

26. $x-\sin x=\sum_{n=2}^{\infty}\frac{(-1)^n x^{2n-1}}{(2n-1)!}$ 27. $\frac{e^x+e^{-x}+2\cos x}{4}=\sum_{n=0}^{\infty}\frac{x^{4n}}{(4n)!}$

28. $a^x=\sum_{n=0}^{\infty}\frac{(\ln a)^n x^n}{n!}$ $(a>0,\ a\neq 1)$ 29. $\sin x-x=\sum_{n=1}^{\infty}\frac{(-1)^n x^{2n+1}}{(2n+1)!}$ 30. $e^{x^2}-1=\sum_{n=1}^{\infty}\frac{x^{2n}}{n!}$

7. Алгоритми модульної структури

Поняття модуля і модульного програмування. Проектування й програмування згори вниз і знизу вгору. Організація міжмодульних зв'язків. Функції в мові програмування С. Рекурсивні алгоритми. Складність і оптимізація алгоритмів модульної структури.

7.1. Поняття модуля і модульного програмування

Малі програми на 30-50 рядків можна писати без розбиття на частини, але великі програм на сотні, тисячі і десятки тисяч рядків написати без розбиття на частини дуже важко.

Як було сказано раніше (див. п. 5.3) показником складності алгоритму, а отже, і програми є цикломатична складність (число Мак-Кейба), за допомогою якої можна оцінити той момент, коли алгоритм стає досить складним і код програми потребує розбиття на окремі частини[16] — модулі.

Модуль — послідовність логічно зв'язаних фрагментів, які є невеликими функціонально закінченими частинами алгоритму чи програми. Модуль може бути замінений на еквівалентну частину без зміни решти алгоритму чи програми. Алгоритм чи програму розбивають на модулі у всіх тих випадках, коли та сама функція виконується кілька разів або ця функція досить складна і робить основну логічну схему громіздкою.

Модульне програмування — це написання програм модулями. Програму можуть утворювати один або кілька самостійно розроблених модулів. Модульне програмування виникло ще на початку 1960-х років.

У програмах модулі в більшості випадків оформляють як підпрограми (у різних мовах програмування їх називають по-різному: функції, процедури, методи) чи групи підпрограм, зібрані в окремому файлі. Той самий модуль може використовуватися в одному чи кількох місцях програмної системи, також він може використовуватися в різних програмах.

Виділення окремих функцій алгоритму чи програми в ізольовані одиниці має ряд *переваг*:

[16] Томас Дж. Мак-Кейб рекомендував програмістам розділяти програми на менші частини, якщо їхня цикломатична складність перевищує 10, проте за певних обставин може бути доцільним послабити обмеження й дозволити модулі зі складністю до 15.

— алгоритм модульної структури простіше проектувати і програмно реалізувати, легше тестувати і модифікувати, оскільки можна встановити відповідність структурних одиниць з виконуваними ними функціями;

— великий алгоритм і відповідну програму можуть одночасно розробляти кілька фахівців, що дає можливість раніше закінчити роботу;

— виникає багато реальних контрольних точок для відстежування розробки проекту;

— можна створити бібліотеку алгоритмів і бібліотеку програмних модулів, які використовуються в багатьох інших алгоритмах і програмних системах. Це особливо важливо з погляду надійності, оскільки в бібліотеки, як правило, поміщають перевірені, оптимальні, добре налагоджені, високостабільні алгоритми і програми.

Використання модулів поряд з перевагами має такі *недоліки*:

— може збільшитися час компіляції і час виконання програми;

— проблеми організації міжмодульної взаємодії можуть виявитися досить складними.

Як правило, модулі виконуються як окремо скомпільовані зовнішні підпрограми, хоч у деяких мовах програмування модулі можуть бути внутрішніми підпрограмами.

Основними властивостями модулів є:

— звернення до модуля здійснюється за його іменем;

— модуль активізується операційною системою чи є підпрограмою, яку викликає інший модуль;

— модуль має повертати керування тій програмній одиниці, яка його викликала;

— модуль може звертатися до інших модулів;

— модуль, як правило, має один вхід і один вихід. Це гарантує замкнутість модуля і спрощує супровід програми. Крім того, користувачі надають перевагу кільком схожим модулям, а не одному модулю з кількома входами;

— модуль порівняно невеликий (як правило, код модуля містить від 20 до 2000 рядків);

— модуль не повинен зберігати історію своїх викликів для керування своїм функціонуванням;

— модуль повинен реалізувати одну функцію (самостійну задачу — повністю визначене перетворення вхідних даних у результат), причому цілком.

У більшості випадків дані між модулями передаються списком параметрів (при цьому обов'язково має бути встановлена позиційна відповідність і відповідність щодо типів між фактичними і формальними параметрами), але також можуть передаватися за допомогою глобальних змінних, загальних блоків чи загальних пулів, файлів. Модулі, дані (інформація) яких передаються тільки списком параметрів, називають *повністю закритими*. Такі модулі не мають побічних ефектів на елементах спільної бази даних і використовуються в інших задачах незалежно від наявної бази даних. Така форма модулів найбільш прийнятна і їй треба надавати перевагу. Проте інформаційні потреби модуля, особливо в складних програмах, іноді потребують застосування бази даних, побудованої на глобальних змінних або спільних блоках, замість або одночасно зі списком параметрів. Такі модулі називають *частково закритими*. Внутрішні підпрограми часто використовують цю форму взаємодії за даними, оскільки не треба передавати великі блоки даних.

Модульне програмування полегшує читання програм, коли призначення її модулів відоме. Модульну програму досить легко корегувати при зміні в майбутньому вимог до функцій програми.

Використання модулів дає можливість писати добре структуровані програми, в яких легко відслідковується основний алгоритм, які неважко зрозуміти тим, хто їх читає, які легко налагоджувати і які менш чутливі до помилок програмування. Крім того, окремі модулі можна багаторазово використовувати в тій самій програмі і в різних програмах.

7.2. Проектування й програмування згори вниз і знизу вгору

Проектування *згори вниз* передбачає поділ складного алгоритму на модулі. Ці модулі ділять далі, доки результуючий модуль не буде виконувати зрозумілу функцію, яку вже не треба розкладати на простіші.

При застосуванні підходу *знизу вгору* спочатку проектують найбільш деталізовані фундаментальні частини алгоритму, які потім об'єднують, щоб зробити модуль вищого рівня. Така інтеграція підмодулів у модуль вищого рівня виконується неодноразово до отримання необхідного повного алгоритму.

Традиційно проектування програмної системи здійснюється згори вниз.

Програмування *згори вниз* (низхідне) — це спосіб реалізації і тестування в ієрархічній послідовності, починаючи з модулів, які відображають загальну схему алгоритму.

Програмування *знизу вгору* (висхідне) — це спосіб реалізації і тестування у зворотній ієрархічній послідовності, починаючи з модулів найнижчого рівня, які деталізують алгоритм.

Щоб уявити процеси програмування згори вниз і знизу вгору, треба програму розглядати як структуру з модулів у вигляді дерева чи мережі з верхнім рівнем, який містить керуючі елементи, відповідно до заданих вимог. Як правило, структура розширюється донизу в міру того, як керуючий модуль викликає підпорядковані модулі, які в свою чергу викликають наступні модулі.

Програмування знизу вгору починають з модулів найнижчого рівня. Налагодження, як правило, виконують за допомогою спеціальних налагоджувальних програм, які передають налагоджуваній програмі певні дані й одержують і візуалізують результат її роботи. Хоч налагоджувальні програми порівняно не складні, але на їхнє написання йде час. Оскільки вони не входять в результуючу програму, то витрачувані на них кошти відносять до накладних витрат.

Труднощі при програмуванні знизу вгору виникають, як правило, на заключній стадії розробки. Наприклад, може з'ясуватися, що в структурах да-

них не передбачено зберігання інформації, яка потрібна буде на певному етапі подальшої обробки даних, і це може призвести до необхідності перероблення вже налагоджених програм.

Програмування згори вниз починається з найвищого рівня, потім розробляється наступний рівень і об'єднується в єдину програму шляхом тестування і т.д. до того часу, доки не буде досягнуто низу структури. При цьому підході відповідність функціональних специфікацій системи і складових її програм перевіряється до переходу на нижчий рівень специфікацій. Низхідна розробка концентрує увагу на частково закінченій програмі, інтеграція і тестування йдуть неперервно протягом усієї розробки. Оскільки модулі розробляють згори вниз, то замість програм нижчого рівня використовують *заглушки*, які задовольняють вимогам інтерфейсу, але не виконують потрібних функцій або виконують їх частково. У подальшому заглушки замінюють чи доробляють до справжніх модулів відповідно до алгоритму. Ієрархічна схема проекту з заглушками має, наприклад, вигляд, поданий на рисунку 7.1.

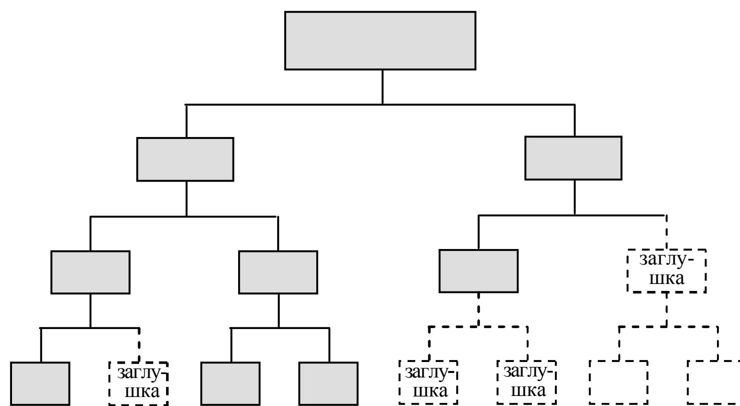


Рисунок 7.1. Ієрархічна схема проекту з заглушками

Оскільки при програмуванні згори вниз кожну частину програми тестують у процесі її включення в програму, то повністю реалізована програма є вже й повністю відтестованою. Застосування програмування згори вниз дає можливість підвищити надійність і зменшити вартість розробки.

При записі алгоритмів теж можна використовувати блоки з загальними діями, які в подальшому будуть деталізуватися. На цьому базується *принцип модульності програмування*.

На практиці обидва методи програмування (низхідне і висхідне) можна застосувати одночасно. При цьому перший крок розробки програми полягає в створенні загальної логічної структури, а потім загальних блоків низького рівня, які найчастіше використовуються, після чого застосовується метод згори вниз. Такий підхід дає кращий результат, ніж кожен метод окремо.

7.3. Організація міжмодульних зв'язків

Як було зазначено вище, дані між модулями у більшості випадків передаються за допомогою списку параметрів. Параметри, описані в заголовку модуля, називають *формальними параметрами*, а параметри, вказані при викликові, — *фактичними*. Фактичними параметрами можуть бути змінні, константи, вирази, вказівники, структури тощо. Фактичні параметри повинні бути записані в тій самій послідовності, що й формальні, і мати відповідний тип.

У загальному випадку при зверненні з одного модуля *A* (який викликає) до іншого модуля *B* (який викликається) використовується ділянка *стека викликів*, довжина якої залежить від модуля *B*. У вершині стека:

— розміщуються значення чи адреси фактичних параметрів модуля *A*;

— виділяється ділянка пам'яті для локальних змінних і констант модуля *B*;

— записується *адреса повернення* (адреса команди, яка має виконуватися в модулі *A* після закінчення роботи викликаного модуля *B*);

— якщо модуль *B* є функцією, то в разі потреби записується *адреса значення* (адреса модуля *A*, за якою треба розмістити значення).

Після цього керування передається першому операторові викликаного модуля *B*.

Після закінчення роботи модуля *B* керування передається модулеві *A*. При цьому виконуються такі дії:

— з вершини стека вибирається адреса повернення;

— якщо модуль *B* є функцією, то її значення запам'ятовується в ділянці пам'яті, на яку вказує вказівник адреси значення;

— ділянка стека викликів модуля *B* вилучається зі стека і в вершині стека опиняється ділянка стека викликів модуля *A*;

— за адресою повернення поновлюється виконання команд модуля *A*.

Локальні змінні і локальні константи доступні тільки в тому модулі, в якому вони описані (а також у вкладених модулях, якщо мова програмування підтримує використання вкладених модулів; у мові програмування C немає вкладених функцій; у Pascal є вкладені підпрограми — функції і процедури).

Апарат фактичних-формальних параметрів служить для передачі даних між модулем *A*, який викликає, і модулем *B*, який викликається. Відповідність між параметрами встановлюється позиційно. Тип фактичного параметра повинен збігатися з типом формального параметра. Імена фактичних і формальних параметрів можуть бути різними.

Існує два способи передачі параметрів — за значенням і за посиланням.

При *передачі параметра за значенням* перед початком виконання модуля *B* обчислюється конкретне значення фактичного параметра. Для формального параметра виділяється нова ділянка пам’яті у стеку викликів. Значення фактичного параметра копіюється в цю ділянку пам’яті. Під час виконання викликаного модуля ніякі зміни значення формального параметра не впливають на значення відповідного фактичного параметра. Після закінчення роботи викликаного модуля *B* ділянка пам’яті формального параметра стає недоступною, а фактичний параметр буде мати те саме значення, яке мав до початку роботи модуля *B*, незалежно від того, що відбувалося з формальним параметром.

При передачі параметра за значенням відповідним фактичним параметром може бути вираз (у широкому розумінні) відповідного типу.

Передача параметрів за посиланням (*за адресою*) використовується тоді, коли треба, щоб модуль повертав кілька значень. При передачі параметрів за посиланням при зверненні до модуля B передається адреса фактичного параметра в пам’яті комп’ютера. У стеку викликів виділяється нова ділянка пам’яті для формального параметра і передана адреса копіюється в

неї. У викликаному модулі B ця адреса надає доступ до фактичного параметра. Тому всі зміни формального параметра відображаються на значенні фактичного параметра.

При цьому фактичним параметром має бути вказівник на змінну, тобто адреса змінної. Відповідним формальним параметром має бути вказівник, тип якого точно збігається з типом змінної, адреса якої передається.

Обмін даними між модулями також може здійснюватися шляхом використання глобальних змінних, загальних блоків, загальних пулів, файлів.

7.4. Функції в мові програмування С

Практично в усіх мовах програмування є засоби створення модульної структури програми. Мови, в яких передбачено такі механізми, називають процедурно-орієнтованими. До них належить і мова С.

У мові програмування С модулі оформляють як функції, всі функції є зовнішніми (у деяких мовах програмування, зокрема в мові Pascal, розрізняють підпрограми-функції і підпрограми-процедури, вони можуть бути як зовнішніми, так і внутрішніми).

Зв'язок між функціями здійснюється через параметри, повернені значення і глобальні змінні. При цьому головна функція описує алгоритм без деталізації, а решта функцій деталізують алгоритм.

Функція — це іменована логічно завершена сукупність оголошень і операторів, призначених для розв'язання певної задачі.

При використанні функцій треба знати, яку саме задачу розв'язує дана функція (виконує відповідну дію і видає результуюче значення) і які параметри їй треба передати.

Кожна функція є самостійним фрагментом програми, пов'язаним з основною програмою лише за допомогою кількох параметрів. Самостійність функцій дає можливість локалізувати в них деталі програмної реалізації тієї чи іншої алгоритмічної дії і тому зміна цих деталей, наприклад, у процесі налагодження, як правило, не призводить до зміни основної програми.

Кожна функція має власне ім'я. Використання цього імені в тексті програми призводить до активізації функції і називається її *викликом*. Зразу після активізації функції починають виконуватися оператори, які в неї входять. Після виконання останнього з них керування повертається назад у функцію, яка викликала, і виконуються команди (оператори чи частини виразу), розміщені безпосередньо за викликом функції. Описаний процес схематично подано на рисунку 7.2.

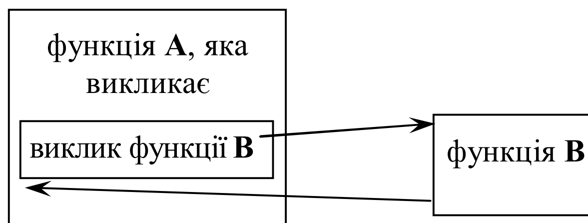


Рисунок 7.2. Схема виклику функції

Для обміну інформацією між основною програмою (функцією, яка викликає) і функцією (виклик якої здійснюється) можуть використовуватися параметри, хоч обмін даними може здійснюватися за допомогою глобальних змінних чи файлів. Функцію називають *функцією з побічним ефектом*, якщо після виходу з неї оновлюється значення хоч би однієї глобальної змінної. Використання глобальних змінних призводить до сильного зв'язку між функціями; хороший стиль програмування вимагає ніколи не використовувати глобальні змінні, якщо немає виняткової потреби.

Якщо функція має параметри, то при зверненні до функції вони вказуються в круглих дужках за іменем функції, якщо функція не має параметрів, то після імені функції розміщуються порожні відкриваюча й закриваюча дужки. Крім використання параметрів, функція може повертати одне значення, яке з функції передається оператором return (у мові Pascal значення функції повертається шляхом присвоювання імені функції певного значення в тілі функції).

У мові C звернення до функції можна здійснювати або у виразі, або окремим оператором (у мові Pascal звернення до функцій записується у виразі, до процедур — окремим оператором). Якщо функція повертає значення і її виклик здійснюється окремим оператором, то повернуте значення ніяк не використовується.

Існують стандартні бібліотеки функцій мови C і відповідні функції є стандартними. Стандартними вони називаються тому, що постачаються разом з компілятором мови C. Наявність бібліотек таких функцій значно спрощує розробку прикладних програм.

Якщо потрібних стандартних функцій нема, то програмістові треба розробити свої нестандартні функції. Розглянемо кілька прикладів.

Приклад 7.4.1. Скорочення звичайних дробів. Щоб скоротити дріб треба знайти найбільший спільний дільник (НСД) чисельника і знаменника. Оформимо алгоритм Евкліда знаходження НСД на основі ділення як функцію.

Програма скорочення дробів має вигляд:

```
/* Скорочення дробів з використанням алгоритму Евкліда */
#include <stdio.h>
#include <stdlib.h>
int nsd(int ch, int zn);   // прототип функції nsd
/* Головна функція */
int main() { // Скорочення дробів
  int a,b,c;   // чисельник, знаменник і НСД
  system("chcp 1251 & cls");
  printf("Введіть дріб a/b: ");
  scanf("%d/%d",&a,&b);
  c=nsd(a,b);
  if (c>1) {
    a=a/c;
    b=b/c;
  }
  printf("НСД=%d. Скорочений дріб: %d/%d",c,a,b);
  printf("\n\n");
  system("pause");
  return 0;
}
/* Функція знаходження НСД.
   Алгоритм Евкліда на основі ділення */
int nsd(int ch, int zn) {
  int r;   // остача від ділення
  while (zn) {
    r=ch%zn;
    ch=zn;
    zn=r;
  }
```

```
Введіть дріб a/b: 6/8
НСД=2. Скорочений дріб: 3/4
```

```
Введіть дріб a/b: 35/21
НСД=7. Скорочений дріб: 5/3
```

```
Введіть дріб a/b: 1/15
НСД=1. Скорочений дріб: 1/15
```

```
Введіть дріб a/b: 0/8
НСД=8. Скорочений дріб: 0/1
```

```
  return ch;
}
```

Функція nsd має два параметри ch і zn, значення яких не змінює (параметри передаються за значенням); має одну локальну змінну r; значенням функції є обчислене значення НСД.

Приклад 7.4.2. Пошук у тексті цілих чисел і виведення їх на екран. Натискання клавіші Enter означає кінець вводу. Останнім перед кінцем вводу має бути число. Якщо останнім перед Enter буде не число, то ввід і обробка тексту продовжаться далі.

При розробці алгоритму модульної структури для розв'язання цієї задачі можна виділити дві основні функції — пропуск символів, відмінних від цифр, і формування цілого числа.

Програма мовою С:

```
/* Пошук у тексті цілих чисел */
#include <stdio.h>
#include <stdlib.h>
void ignoresymb(void);    // прототип функції ignoresymb
int number (void);    // прототип функції number
unsigned char sym;   // символ з введеного тексту (глобальна змінна)
/* Головна функція */
int main() {
  int n;    // ціле число
  system("chcp 1251 & cls");
  printf("Введіть текст з числами\n");
  sym=' ';
  while (sym!='\n') {
    ignoresymb();
    n=number();
    printf("%-8d",n);
  }
  printf("\n\n");
  system("pause");
```

```
  return 0;
}
/* Пропуск символів, відмінних від цифр */
void ignoresymb(void) {
  while ((sym=getchar()) <'0' || sym>'9');
}
/* Формування числа з цифр */
int number (void) {
  int ch=0;    // число, яке формується
  while (sym>='0' && sym<='9') {
    ch=ch*10+sym-'0';
    sym=getchar();
  }
  return ch;
}
```

```
Введіть текст з числами
Найбільша кількість псевдонімів була в О. Кониського - 141, в І. Франка -99,
О. Маковея - 56, у Лесі Українки - 1
141     99      56      1
```

```
Введіть текст з числами
   Мовознавець Іван Ющук у статті "12 фактів про давність української мови"
(20.07.2006) пише, що українська мова вже в 6-7 ст. мала окреслено сучасні
обриси (У 2017 І.П. Ющук видав "Словник української мови 6 століття")
12      20      7       2006    6       7       2017    6
   Україна близько 700 років була розчленована між різними державами, які
насаджували їй свої мови. У складі радянської імперії Україна перебувала з
1922 до 1991
700     1922    1991
```

Обидві функції ignoresymb і number не мають параметрів. Функція ignoresymb не має локальних змінних; функція number має одну локальну змінну ch. Функція number повертає обчислене значення. Змінна sym є глобальною і використовується у всіх функціях (включаючи й головну).

Приклад 7.4.3. Програма виконує впорядкування трьох чисел за зростанням, використовуючи функцію впорядкування двох чисел за зростанням. Програма видає повідомлення «Задані числа впорядковані», якщо було задано впорядковані числа, інакше виводить результат впорядкування.

```
/* Впорядкування трьох цілих чисел за зростанням */
#include <stdio.h>
#include <stdlib.h>
int sort(int *a, int *b);   // прототип функції
/* Головна функція */
int main() {
  int m, n, k;   // вхідні значення
  int i;   // лічильник перевірок
  system("chcp 1251 & cls");
  printf("\tВпорядкування за зростанням трьох чисел\
    \nВведіть значення m, n i k\n");
  while (scanf("%d%d%d", &m, &n, &k))  {
    i=0;
    while (i++, sort(&m,&n)||sort(&n,&k));   // цикл впорядкування
    if (i==1 )
      printf("   Задані числа впорядковані\n");
    else
      printf("   Результат впорядкування: %d %d %d\n", m, n, k);
  }
  printf("\n\n");
  system("pause");
  return 0;
}
/* Функція впорядкування двох чисел */
int sort(int *a, int *b) {
  int c,   // робоча змінна
      ind;   // індикатор результату
  if (*a<=*b)
    ind=0;
  else {
    c=*a;
    *a=*b;
    *b=c;
```

```
    ind=1;
  }
  return ind;
}
```

```
        Впорядкування за зростанням трьох чисел
Введіть значення m, n і k
1 2 3
   Задані числа впорядковані
1 3 2
   Результат впорядкування: 1 2 3
2 1 3
   Результат впорядкування: 1 2 3
2 3 1
   Результат впорядкування: 1 2 3
3 2 1
   Результат впорядкування: 1 2 3
3 1 2
   Результат впорядкування: 1 2 3
1 1 2
   Задані числа впорядковані
1 1 1
   Задані числа впорядковані
_
```

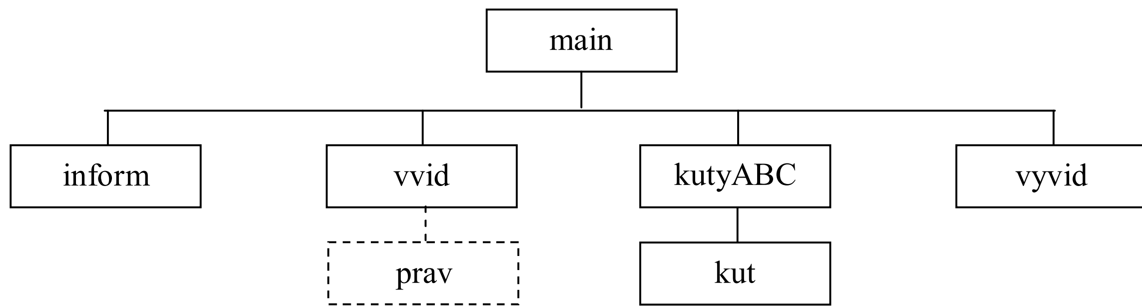
У головній програмі двічі використано звернення до функції впорядкування двох чисел sort. Отже, фрагмент програми, який виконує ті самі дії з різними змінними, не повторюється і за рахунок цього скорочується код. Параметри передаються за адресою. Функція sort набуває значення 1, якщо відбулася перестановка чисел, або значення 0, якщо числа не впорядковувалися.

Приклад 7.4.4. Обчислення кутів трикутника за відомими трьома сторонами.

Для знаходження кутів трикутника можна скористатися формулою косинусів $a^2 = b^2 + c^2 - 2bc\cos A$ (де A — кут, який спирається на сторону a); для подання кута у градусах — використати перетворення $\varphi_{\circ} = \varphi_r \cdot 180/\pi$.

У програмі мають бути такі функції: головна (main), виведення інформації про програму (inform), введення даних і перевірки існування трикутника (vvid, prav), обчислення кута за сторонами (kutyABC) з викликом функції обчислення конкретного кута (kut), виведення результатів (vyvid).

Розроблена ієрархічна схема програми має вигляд:



Програма модульної структури мовою С матиме вигляд:

```
/* Обчислення кутів трикутника за відомими трьома
   сторонами з використанням формули косинусів */
#include <stdio.h>
#include <stdlib.h>
#define _USE_MATH_DEFINES
#include <math.h>
void inform();   // прототипи функцій
void vvid(double *a, double *b, double *c);
void kutyABC(double a, double b, double c,
             double *A, double *B, double *C);
double kut(double a, double b, double c);
void vyvid(double A, double B, double C);
/* Головна функція - організовує обчислення значень кутів
   для багатьох трикутників */
int main() {
  double a, b, c, A, B, C;   // сторони і кути
  int ind;   // індикатор продовження роботи
  char vidp;  // відповідь користувача щодо продовження роботи
  system("chcp 1251 & cls");
  inform();
  ind=1;
  while (ind) {
    fseek(stdin,0,SEEK_END);
    vvid(&a,&b,&c);
    kutyABC(a,b,c,&A,&B,&C);
    vyvid(A,B,C);
    printf("Продовжувати роботу (Y - так)? ");
```

```
    fseek(stdin,0,SEEK_END);
    vidp=getchar();
    fseek(stdin,0,SEEK_END);
    if (!(vidp=='Y'||vidp=='y')) ind=0;
  }
  printf("\n\n");
  system("pause");
  return 0;
}
/* Інформація про програму */
void inform() {
  printf("Програма обчислення кутів трикутника за довжинами його сторін\n");
  printf("Для обчислення величини кута використано формулу косинусів:\n");
  printf("\t\tcosA=(b^2+c^2-a^2)/2bc\n");
  printf("© Кублій Л.І.\n\n");
}
/* Введення даних про трикутник і перевірка їхньої правильності */
void vvid(double *a, double *b, double *c) {
  int ind;
  ind=1;
  do {
    printf("Введіть a, b, c  ");
    while (scanf("%lf %lf %lf",a,b,c)<3) {  // передбачення
                                // механічних помилок вводу
      printf("Неправильний формат даних. Повторіть ввід a, b, c  ");
      fseek(stdin,0,SEEK_END);
    }
    if (*a<=0 || *b<=0 || *c<=0 || *a+*b <= *c ||
        *a+*c <= *b || *b+*c <= *a)
      printf("   Трикутник не існує\n");
    else ind=0;
  } while (ind);
}
```

```
/* Знаходження кутів трикутника за вказаними сторонами */
void kutyABC(double a, double b, double c,
             double *A, double *B, double *C) {
  *A=kut(a,b,c); *B=kut(b,c,a); *C=kut(c,a,b);
}
/* Обчислення кута A навпроти сторони a за
   вказаними сторонами трикутника a, b, c */
double kut(double a, double b, double c) {
  return acos((b*b+c*c-a*a)/(2*b*c))*180./M_PI;;
}
```

```
Програма обчислення кутів трикутника за довжинами його сторін
Для обчислення величини кута використано формулу косинусів:
                cosA=(b^2+c^2-a^2)/2bc
© Кублій Л.І.

Введіть a, b, c  1 3 4
   Трикутник не існує
Введіть a, b, c  1 1 1
Навпроти сторони
   a кут 60.00°
   b кут 60.00°
   c кут 60.00°
Похибка при обчисленні кутів 1.42109e-014°

Продовжувати роботу (Y - так)? y
Введіть a, b, c  3 4 5
Навпроти сторони
   a кут 36.87°
   b кут 53.13°
   c кут 90.00°
Похибка при обчисленні кутів 0°

Продовжувати роботу (Y - так)? y
Введіть a, b, c  8 9 5
Навпроти сторони
   a кут 62.18°
   b кут 84.26°
   c кут 33.56°
Похибка при обчисленні кутів 7.10543e-015°

Продовжувати роботу (Y - так)? n
```

```
/* Виведення значень кутів A, B, C, які
   спираються відповідно на сторони a, b, c */
void vyvid(double A, double B, double C) {
  printf("Навпроти сторони\n");
```

```
  printf("   a кут %0.2f°\n", A);   // символ градуса Alt+0176
  printf("   b кут %0.2f°\n", B);
  printf("   c кут %0.2f°\n", C);
  printf("Похибка при обчисленні кутів %g°\n\n", fabs(180.-A-B-C));
}
```

Функція vvid вводить значення сторін трикутника, а також перевіряє правильність введених даних (чи існує трикутник). Цю функцію можна розбити на дві — функцію вводу даних і функцію перевірки.

Якщо у функції vvid замість блоку перевірки існування трикутника помістити звернення до функції, яка виконує таку перевірку, то в цю функцію можна передавати параметри або за значенням, або за адресою. При цьому програмні коди будуть відрізнятися. Розглянемо обидва випадки.

а) Передача параметрів за значенням. Звернення до функції має вигляд:

```
    ind=prav_zn(*a,*b,*c);   // передаються значення
```

Код функції буде таким:

```
/* Перевірка умови існування трикутника */
int prav_zn(double a, double b, double c) { // параметри-значення
  int ind;
  ind=1;
  if (a<=0 || b<=0 || c<=0 ||a+b <= c || a+c <= b || b+c <= a)
    printf("   Трикутник не існує\n");
  else
    ind=0;
  return ind;
 }
```

б). Передача параметрів за адресою. Звернення до функції має вигляд:

```
    ind=prav_ad(a,b,c);   // передаються адреси
```

Код функції буде таким:

```
/* Перевірка умови існування трикутника */
int prav_ad(double *a, double *b, double *c) { // параметри-адреси
  int ind;
  ind=1;
```

```
  if (*a<=0 || *b<=0 || *c<=0 ||*a+*b <= *c
      || *a+*c <= *b || *b+*c <= *a)
    printf("   Трикутник не існує\n");
  else
    ind=0;
  return ind;
}
```

У мові програмування С вказівники на змінні (крім масивів, — див. розд. 8) записуються з операцією взяття адреси & (унарна операція & дає адресу операнда в пам'яті; наприклад, &n вказує адресу ділянки пам'яті комп'ютера, в якій розміщується значення змінної n, тобто адресу змінної n; &aaa — вказівник на змінну aaa, &ch — вказівник на змінну ch, але mas — вказівник на масив mas[ ], оскільки використання імені масива без квадратних дужок еквівалентне вказуванню адреси його першого елемента (можна також записати &mas[0]), str — вказівник на рядок символів str[ ]).

При передачі параметра за посиланням формальний параметр повинен бути оголошений як вказівник (тип *ім'я змінної вказівника). Наприклад, вказівник p оголошується так: int *p. Тоді вираз *p задає змінну, адресу якої містить p. Унарна операція розіменування вказівника *, застосована до вказівника, забезпечує доступ до вмісту комірки пам'яті, на яку вказує вказівник (тобто доступ до значення змінної, розміщеної за вказаною адресою; наприклад, *p дає доступ до значення, записаного за адресою p).

При використанні вказівників у виразах для уникнення помилок, пов'язаних з пріоритетами операцій, треба вираз *p брати в дужки: (*p). Наприклад, якщо треба вміст комірки за адресою p збільшити на 1, то вираз *p++ дає неправильний результат (береться наступна адреса), хоч, здавалося б, обидві операції * і ++ є унарними і мають однаковий пріоритет і тому повинні виконуватися зліва направо, тобто спочатку *p, а потім інкремент. При цьому правильно працює вираз (*p)++. Також правильно працює оператор *p =*p +1.

7.5. Рекурсивні алгоритми

Рекурсію (кругове визначення) використовують тоді, коли можна виділити самоподібність задачі. Рекурсивні звернення дають можливість порівняно компактно описувати досить складні обчислення. Використання рекурсивних підпрограм також робить опис алгоритму більш наочним. Застосування рекурсії є альтернативою до використання циклів.

Рекурсивна підпрограма — це підпрограма, яка звертається сама до себе. При рекурсивній організації обробки даних підпрограма може викликати сама себе безпосередньо або опосередковано через ланцюжок викликів інших підпрограм. Таким чином, розрізняють пряму і непряму (взаємну) рекурсії.

Підпрограму називають *прямо рекурсивною*, якщо вона містить у своєму тілі виклик самої себе. Якщо ж одна підпрограма викликає іншу підпрограму,

яка у свою чергу викликає першу, то таку підпрограму називають *непрямо рекурсивною* (див. приклади 7.5.6 і 10.5.8). При цьому одна або кілька з викликаних підпрограм на певному етапі звертаються до початкової підпрограми зі зміненим набором параметрів. Рекурсивні підпрограми найчастіше використовують для компактної реалізації алгоритмів.

Рекурсія в мові C, як і в багатьох інших мовах (наприклад, у мові Pascal), можлива завдяки тому, що при викликові функції динамічно створюються нові локальні змінні.

Звернення до рекурсивної підпрограми нічим не відрізняється від виклику будь-якої іншої підпрограми. Коли підпрограма викликає сама себе, в стеку виділяється місце для нових локальних змінних і формальних параметрів, а також запам'ятовується адреса повернення. Код підпрограми працює з цими змінними. Такі копії даних будуть породжуватися до виходу на граничну умову. Рекурсивний виклик нову копію підпрограми не створює. Новими є тільки аргументи. Оскільки кожна рекурсивно викликана підпрограма завершує роботу, то старі локальні змінні і параметри видаляються зі стека і виконання продовжується з точки, в якій було звернення всередині цієї ж підпрограми.

У рекурсивній підпрограмі обов'язково повинна бути умова повернення і оператор повернення без рекурсивного виклику. Якщо це не зробити, то, викликавши підпрограму, вийти з неї буде неможливо (доки не переповниться стек).

Оскільки параметри і локальні змінні підпрограми зберігаються в стекові і кожен новий виклик створює нову копію параметрів і змінних, простір стека може вичерпатися: виникає помилка — *переповнення стека*.

Звернення до рекурсивної підпрограми нічим не відрізняється від виклику будь-якої іншої підпрограми. При кожному новому рекурсивному зверненні до підпрограми в пам'яті створюється нова копія формальних параметрів і виділяється нова ділянка пам'яті у стекові для локальних змінних, а також запам'ятовується адреса повернення. Такі копії даних будуть породжуватися до виходу на граничну умову.

Породження нових копій даних рекурсивної підпрограми до виходу на граничну умову називають *рекурсивним спуском*. Максимальна кількість копій даних рекурсивної підпрограми, які одночасно можуть перебувати в пам'яті комп'ютера, називають *глибиною рекурсії*. Завершення роботи рекурсивних підпрограм і повернення за адресою до точки виклику (аж до першої), з якої було ініційовано рекурсивні виклики, називають *рекурсивним підйомом*.

Глибина вкладеності рекурсії не обмежена — єдиним обмеженням є тільки обсяг пам'яті стека викликів, оскільки пам'ять під параметри й локальні змінні виділяється при кожному зверненні.

Як правило, рекурсія потребує певних витрат часу й пам'яті на створення локальних змінних і задача виконується повільніше і використовує набагато більше пам'яті, ніж у випадку, коли рекурсія не використовується. Але такі витрати найчастіше виправдовуються корисністю самого принципу рекурсії — можна компактно подати алгоритм.

Основною перевагою застосування рекурсивних підпрограм є використання їх для створення більш простих версій деяких алгоритмів порівняно з ітераційними еквівалентами. Наприклад, алгоритм швидкого сортування (див. приклад 8.4.5) досить важко реалізувати ітераційним способом. Для розв'язання деяких проблем, особливо пов'язаних зі штучним інтелектом, також використовують рекурсивні алгоритми. При обробці таких рекурсивних структур даних, як стеки, дерева, графи, автомати, застосовують рекурсивні алгоритми.

Як правило, в основі рекурсивного алгоритму лежить рекурентне визначення якогось поняття, тобто визначення, яке задає деякий об'єкт у термінах більш простого випадку цього ж об'єкта. Рекурентні співвідношення встановлюють залежності між результатами певних дій на n-кроці від результатів аналогічних дій, отриманих на попередньому $(n-1)$-кроці чи кількох попередніх кроках.

Класичними прикладами використання рекурсії є обчислення сум, факторіалу числа, реалізація операції піднесення до степеня, знаходження значень чисел Фібоначчі. Проте ці приклади мають тільки ілюстративний харак-

тер через їхню зручність для пояснення поняття рекурсії, але практично не дають виграшу в програмній реалізації порівняно з ітераційним способом розв'язання цих задач.

Ідея рекурсивного визначення модуля така. Із сукупності допустимих значень аргументів треба виділити прості випадки, при яких обчислення зводиться до інших, наперед визначених чи незалежно визначених понять. Таким чином, створення рекурсивного модуля передбачає зведення задачі до такої самої задачі, але, як правило, меншої складності чи меншої розмірності.

Приклад 7.5.1. Обчислення суми $\sum_{i=1}^{n} i^2$.

При обчисленні вказаної суми, якщо $n \le 0$, треба вважати, що значення суми дорівнює 0. Для решти випадків треба спробувати виразити значення суми через значення цієї ж суми при інших значеннях аргументів так, щоб у результаті все обчислення звелося до виділених простих випадків. У даному випадку обчислення суми n доданків можна, відокремивши останній доданок, звести до обчислення суми $n-1$ доданка. Отже, можна одержати таке рекурентне визначення функції обчислення суми:

$$\sum_{i=1}^{n} i^2 = n^2 + \sum_{i=1}^{n-1} i^2 \text{, якщо } n > 0;$$

$$\sum_{i=1}^{n} i^2 = 0 \text{, якщо } n \le 0,$$

тобто

$$Suma(n) = n^2 + Suma(n-1) \text{, якщо } n > 0;$$

$$Suma(0) = 0.$$

Наприклад, при $n = 3$ за даним рекурентним визначенням суми буде виконано послідовність дій:

$$Suma(3) = 3^2 + Suma(2);$$

$$Suma(2) = 2^2 + Suma(1);$$

$$Suma(1) = 1^2 + Suma(0);$$

$$Suma(0) = 0.$$

Програмні реалізації функції обчислення суми мовою C з використанням рекурсії і з використанням циклу будуть такими:

```
#include <stdio.h>
#include <stdlib.h>
int suma(int n);
int main() {
  int n;
  printf(" Вкажіть кількість доданків суми i^2  ");
  scanf("%d", &n);
  if (n>=0) printf("S(i^2;i=1,%d)=%d", n, suma(n));
  else printf(" Задано від'ємне число");
  printf("\n\n");
  system("pause");
  return 0;
}
```

```
/* Рекурсивне обчислення суми
   квадратів чисел від 1 до n */
int suma(int n) {
  int s;
  if (n>0)
    s=n*n+suma(n-1);
  else
    s=0;
  return s;
}
```

```
/* Ітераційне обчислення суми
   квадратів чисел від 1 до n */
int suma(int n) {
  int s, i;
  s=0;
  for(i=1; i<=n; i++)
    s=s+i*i;
  return s;
}
```

```
 Вкажіть кількість доданків суми i^2  4
S(i^2;i=1,4)=30
```

Рекурсивна функція suma викликатиме сама себе, постійно зменшуючи значення аргументу наступного виклику на 1, доки аргумент не стане рівним 0. У цьому випадку функція поверне число 0.

Приклад 7.5.2. Обчислення факторіалу цілого невід’ємного числа n визначається так:

$$\begin{cases} n! = n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1; \\ 0! = 1. \end{cases}$$

Зведемо цю задачу до задачі з меншим значенням аргументу:

$$\begin{cases} n! = n \cdot (n-1)!; \\ 0! = 1. \end{cases}$$

Нехай рекурсивне обчислення факторіалу реалізує функція fact. У термінах програми це можна записати так:

$$\begin{cases} fact(n) = n * fact(n-1); \\ fact(0) = 1. \end{cases}$$

Програма мовою С має вигляд:

```
...
int fact(int n);
int main() {
  int n;
  printf(" Введіть число n ");
  scanf("%d", &n);
  if (n>=0) printf("n!=%d", fact(n));
  else printf(" Задано від'ємне число");
...
}
```

```
/* Рекурсивне обчислення
   факторіалу числа n */
int fact (int n) {
  int nf;
  if (n>1)
    nf=n*fact(n-1);
  else
    nf=1;
  return nf;
}
```

```
/* Ітераційне обчислення
   факторіалу числа n */
int fact (int n) {
  int nf, i;
  nf=1;
  for(i=1; i<=n; i++)
    nf=i*nf;
  return nf;
}
```

```
 Введіть число n 6
n!=720
```

Рекурсивна функція fact викликатиме сама себе, постійно зменшуючи значення аргументу наступного виклику на 1, доки аргумент не стане рівним 1 (якщо $n = 0$, то рекурсивний алгоритм теж буде працювати правильно, оскільки $0! = 1! = 1$). У цьому випадку функція поверне число 1.

На прикладі обчислення значення $n!$ (при $n = 3$) з використанням розглянутої вище рекурсивної функції fact можна зобразити загальну схему рекурсивних обчислень як ланцюжок викликів цієї функції з передачею нових значень аргументу і повернення значень функції за попередньою адресою (рисунок 7.3).

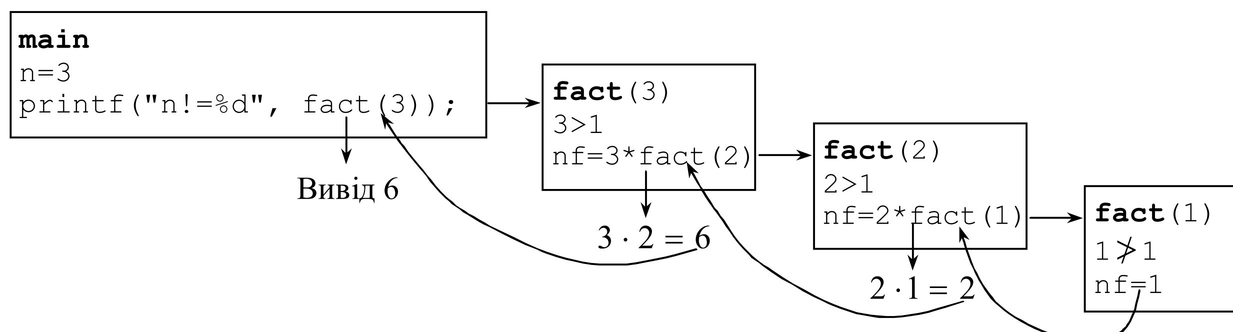


Рисунок 7.3. Схема рекурсивного обчислення значення 3!

У наведених двох прикладах 7.5.1 і 7.5.2 використання рекурсивного алгоритму призведе при кожному викликові функції до копіювання у стек викликів *n* разів значення параметра функції, виділення пам'яті для локальної змінної, запису адреси повернення і адреси значення функції, а при кожному закінченні роботи функції зі стека вибирається адреса повернення, значення функції запам'ятовується в ділянці значення, звільняється пам'ять, виділена під значення параметра і локальну змінну, поновлюється виконання команд за адресою повернення. Застосування ж ітераційного алгоритму потребує тільки використання параметра циклу.

Приклад 7.5.3. Числа Фібоначчі визначаються такими рекурентними співвідношеннями:

$$f_0 = f_1 = 1;$$

$$f_n = f_{n-1} + f_{n-2};\ n = 2,3,...$$

У цьому випадку для обчислення значення наступного числа використовуються два попередні.

Програма обчислення значень з використанням функції може бути такою:

```
...
unsigned long long int fib(int n);
int main() {
  int n, i;
```

```
  printf("Вкажіть номер останнього числа\
    \nФібоначчі, яке треба обчислити  ");
  scanf("%d", &n);
  if (n<=92)
    for (i=0; i<=n; i++)
      printf("f%-2d= %-15llu",i,fib(i));
  else printf("Номер числа перевищує 92\n");
...
}
```

```
/* Рекурсивне обчислення
   чисел Фібоначчі */
unsigned long long int fib(int n) {
  unsigned long long int fn;
  if (n==0 || n==1)
    fn=1;
  else
    fn=fib(n-1)+fib(n-2);
  return fn;
}
```

```
/* Ітераційне обчислення
   чисел Фібоначчі */
unsigned long long int fib(int n) {
  unsigned long long int fn,
      fn_1, fn_2;
  int i;
  if (n==0 || n==1)
    fn=1;
  else {
    fn_2=fn_1=1;
    for (i=2; i<=n; i++) {
      fn=fn_1+fn_2;
      fn_2=fn_1;
      fn_1=fn;
    }
  }
  return fn;
}
```

```
Вкажіть номер останнього числа
Фібоначчі, яке треба обчислити  30
f0 = 1              f1 = 1              f2 = 2              f3 = 3
f4 = 5              f5 = 8              f6 = 13             f7 = 21
f8 = 34             f9 = 55             f10= 89             f11= 144
f12= 233            f13= 377            f14= 610            f15= 987
f16= 1597           f17= 2584           f18= 4181           f19= 6765
f20= 10946          f21= 17711          f22= 28657          f23= 46368
f24= 75025          f25= 121393         f26= 196418         f27= 317811
f28= 514229         f29= 832040         f30= 1346269
```

Алгоритм рекурсивного обчислення значень чисел Фібоначчі порівняно з ітераційним алгоритмом має витончений компактний запис. Але рекурсивний алгоритм потребує $O(2^n)$ викликів, що навіть для невеликих n виконується дуже довго. При цьому багато викликів будуть дублюватися. Ітераційний алгоритм, поступово зсуваючи значення, послідовно обчислює числа Фібоначчі, займаючи лише три комірки пам'яті і не дублюючи виклики з однаковими аргументами.

Приклад 7.5.4. Перевірка правильності введення значення числа з діапазону з використанням рекурсії (ітераційний алгоритм подано в прикладі 6.2.5).

```
#include <stdio.h>
#include <stdlib.h>
void vvid_n(int *n);
int main() {
  int n;
  system("chcp 1251 & cls");
  vvid_n(&n);
  printf("\nВведене значення %d",n);
  printf("\n\n");
  system("pause");
  return 0;
}
/* Ввід значення n */
void vvid_n(int *n) {
  printf("Введіть значення n в межах від 0 до 9:   ");
  scanf("%d",n);
  fseek(stdin,0,SEEK_END);
  if (*n<0 || *n>9) {
    printf("\tнеправильне значення n\n");
    vvid_n(n);
  }
}
```

```
Введіть значення n в межах від 0 до 9:   14
        неправильне значення n
Введіть значення n в межах від 0 до 9:   -3
        неправильне значення n
Введіть значення n в межах від 0 до 9:   7

Введене значення 7
```

У функції vvid_n при наступному зверненні ніякі параметри з попереднього виклику не використовуються, розмірність задачі не зменшується,

умовою виходу з рекурсивного спуску є введення правильного значення змінної. У функцію vvid_n параметр передається за адресою, щоб при виході з рекурсії одержати правильно введене значення. При використанні рекурсивної функції для перевірки правильності введення даних, якщо користувач буде дуже наполегливим, то може переповнитися стек викликів.

Приклад 7.5.5. Розглянемо рекурсивні реалізації різних версій алгоритму Евкліда (див. приклад 6.3.14).

а) Алгоритм Евкліда на основі віднімання, поданий раніше ітераційно:

К1. Ввести числа a, b.

К2. Якщо a=b, то перейти на К5.

К3. Якщо a>b, то a=a–b, інакше b=b–a.

К4. Перейти на К2.

К5. Вивести НСД=a.

з використанням рекурсії матиме вигляд:

К1. Ввести числа a, b.

К2. Якщо a=b, то перейти на К4.

К3. НСД(a, b): Якщо a>b, то НСД(a–b, b);

інакше НСД(a, b–a).

К4. Вивести НСД=a.

Мовою С цей алгоритм можна подати так:

```
/* Алгоритм Евкліда на основі віднімання для знаходження
   найбільшого спільного дільника двох цілих чисел */
...
int nsd(int a, int b);
int main() {
  int a,b;   // введені числа
  printf("   Алгоритм Евкліда\
         \n(на основі віднімання; рекурсивний)");
  while(printf("\nВведіть числа a і b \n"),scanf("%d %d",&a,&b)) {
    a=abs(a);   //функція бібліотеки stdlib.h
    b=abs(b);
```

```
    if (a==0||b==0) // якщо не перевірити, то алгоритм зациклиться
      printf("   НСД=%d",a<b?b:(a!=0?a:-1));
    else
      printf("   НСДе=%d",nsd(a,b));
  }
...
}
/* Рекурсивна реалізація алгоритму Евкліда
   на основі віднімання */
int nsd(int a, int b) {
  int r;
  if(a==b)
    r=a;
  else
   if (a>b)
     r=nsd(a-b, b);
    else
      r=nsd(a, b-a);
  return r;
}
```

```
   Алгоритм Евкліда
(на основі віднімання; рекурсивний)
Введіть числа a і b
107 17
   НСДе=1
Введіть числа a і b
24 40
   НСДе=8
Введіть числа a і b
7 3
   НСДе=1
Введіть числа a і b
-9 12
   НСДе=3
Введіть числа a і b
кінець
```

б) Алгоритм Евкліда на основі ділення, поданий раніше ітераційно:

К1. Ввести числа a, b.

К2. Якщо b=0, то перейти на К5.

К3. r=a mod b, a=b, b=r.

К4. Перейти на К2.

К5. Вивести НСД=a.

з використанням рекурсії можна подати так:

К1. Ввести числа a, b.

К2. НСД(a, b): якщо b=0, то r=a;

інакше r=НСД(b, a mod b).

К3. Вивести r.

Його програмна реалізація має вигляд:

```
/* Алгоритм Евкліда на основі ділення для знаходження
   найбільшого спільного дільника двох цілих чисел */
...
int nsd(int a, int b);
int main() {
  int a,b;    // введені числа
  printf("   Алгоритм Евкліда\
          \n(на основі ділення; рекурсивний)");
  while(printf("\nВведіть числа a і b \n"),scanf("%d %d",&a,&b)) {
    a=abs(a);
    b=abs(b);
    if (a==0||b==0) // без перевірки алгоритм працює
      printf("   НСД=%d",a<b?b:(a!=0?a:-1));
    else
      printf("   НСДе=%d",nsd(a,b));
  }
...
}
/* Рекурсивна реалізація алгоритму Евкліда на основі ділення */
int nsd(int a, int b) {
  int r;
  if(b==0)
    r=a;
  else
    r=nsd(b, a%b);
   return r;
}
```

```
   Алгоритм Евкліда
(на основі ділення; рекурсивний)
Введіть числа a і b
1024 3072
   НСДе=1024
Введіть числа a і b
5 7
   НСДе=1
Введіть числа a і b
7 3
   НСДе=1
Введіть числа a і b
113 17
   НСДе=1
Введіть числа a і b
кінець
```

в) Для поданого раніше двійкового ітераційного алгоритму Евкліда:

К1. Ввести числа a, b.

К2. k=0.

К3. Якщо a і b парні, то a=a/2 і b=b/2; k=k+1; перейти на К3.

К4. Якщо a парне, то a=a/2; перейти на К4. // надалі a завжди непарне

К5. Якщо b парне, то b=b/2; перейти на К5.

К6. Якщо $a > b$, то виконати переприсвоєння a і b; b=b–a; перейти на К5.

К7. Вивести НСД=a*2^k.

рекурсивна програмна реалізація буде такою:

```
/* Алгоритм Евкліда на основі побітових зсувів і
   віднімання непарних чисел для знаходження найбільшого
   спільного дільника двох цілих чисел */
...
int nsdp(int a, int b);
int main() {
  int a,b;   // введені числа
  printf("   Алгоритм Евкліда (на основі \
         \nзсувів і віднімання; рекурсивний)");
  while(printf("\nВведіть числа a і b \n"),scanf("%d %d",&a,&b)) {
    a=abs(a);
    b=abs(b);
    if (a==0||b==0) // якщо не перевірити, то алгоритм зациклиться
      printf("   НСД=%d",a<b?b:(a!=0?a:-1)); // якщо є один 0,
                                // то видає ненульове число,
                                // якщо обидва нулі, то -1
    else
      printf("   НСДп=%d",nsdp(a,b));
  }
...
}
int nsdp(int a, int b) {
  int r; // робоча змінна; обчислений результат
  if (a==b)   // закінчення роботи
    r=a;
  else {   // обчислення НСД
    if ((~a&1)&&(~b&1))    // a і b - парні
      r=nsdp(a>>1, b>>1)<<1;
    else
      if (~a&1)   // a - парне
```

```
    r=nsdp(a>>1, b);
   else
    if (~b&1)   // b - парне
     r=nsdp(a, b>>1);
  // зменшення більшого аргументу
    else {   // a і b - непарні
     if (a>b) {
      r=a; a=b; b=r;
     }
     b=b-a;   // b - парне
     r=nsdp(a, b>>1);
    }
 }
 return r;
}
```

```
   Алгоритм Евкліда (на основі
зсувів і віднімання; рекурсивний)
Введіть числа a і b
20 20
   НСДп=20
Введіть числа a і b
10 20
   НСДп=10
Введіть числа a і b
5 7
   НСДп=1
Введіть числа a і b
3 8
   НСДп=1
Введіть числа a і b
16 3
   НСДп=1
Введіть числа a і b
128 24
   НСДп=8
Введіть числа a і b
кінець
```

Приклад 7.5.6. Обчислити значення виразу $f_n + g_n$, якщо:

$$f_1 = 1; \;\; g_1 = 1;$$

$$f_n = f_{n-1} + 3g_{n-1};$$

$$g_n = 2g_{n-1} + f_{n-1}; \quad n = 2,3,...$$

У цьому прикладі функція *f* визначається рекурентно через функцію *g*, а функція *g* — через функцію *f*. Алгоритм обчислення значення можна реалізувати, використовуючи непряму рекурсію.

Подання алгоритму мовою C має вигляд:

```
/* Реалізація непрямої (взаємної) рекурсії */
...
int f(int n);
int g(int n);
int main() {
 int n;
...
 if (n<=0)
  printf("n має бути >=1");
```

```
  else
    printf("f%d+g%d = %d",n,n,f(n)+g(n));
...
}
/* Рекурсивне обчислення значення функції f */
int f(int n) {
  int fn;
  if (n==1)
    fn=1;
  else
    fn=f(n-1)+3*g(n-1);
  return fn;
}
/* Рекурсивне обчислення значення функції g */
int g(int n) {
  int gn;
  if (n==1)
    gn=1;
  else
    gn=2*g(n-1)+f(n-1);
  return gn;
}
```

```
f(1)=1; g(1)=1;
f(n)=f(n-1)+3*g(n-1);
g(n)=2*g(n-1)+f(n-1);
   Вкажіть значення n  3
f3+g3 = 23
```

Приклад 7.5.7. У прикладі 6.3.8 розглянуто алгоритми з використанням циклів для обчислення значення числа π за такими формулами:

$$\pi = 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{2n-1}, \qquad \pi = 3 + \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n(2n+1)(n+1)}.$$

Розглянемо рекурсивні алгоритми обчислення значення числа π.

```
/* Рекурсивне обчислення суми ряду з заданою точністю eps.
   Члени ряду обчислюються за формулою
            an=((-1)^(n+1))/(2n-1)
   чи       an=((-1)^(n+1))/(n*(2*n+1)*(n+1)),
   поки |an|>=eps. */
```

```
...
double pi_eps(int n, double eps);
int k;  // кількість рекурсивних викликів
int main() {
  double pi;   // наближене значення числа pi
  double eps;   // точність обчислення
  printf("Рекурсивне обчислення значення числа pi за\n\
    першою формулою.\n");
    //другою формулою.\n");
  while (printf("Введіть значення eps: "), k=0,
    scanf("%lf",&eps)) {
      pi=4*pi_eps(1, eps);
      //pi=3+pi_eps(1, eps);
      printf("Наближене значення PI: %.17f\n",pi);
      printf("Кількість членів ряду: %d\n",k);
  }
...
}
/* Обчислення значення числа PI з точністю eps
   за першою чи другою формулами */
double pi_eps(int n, double eps) {
  double an,   // поточний член ряду
    s;    // результат обчислення
  k++;   // номер ітерації (рекурсивного виклику)
  s=0;
  an=1./(2.*n-1.);   // перша формула
  //an=1./(n*(2.*n+1.)*(n+1.));   // друга формула
  n++;
  if (an>=eps) s=pi_eps(n, eps);
  if (n&1) an=-an;
  return s+an;
}
```

```
Рекурсивне обчислення значення числа pi за
    першою формулою.
Введіть значення eps: 0.00012575
Наближене значення PI: 3.14184409939924600
Кількість членів ряду: 3977
Введіть значення eps: 0.001
Наближене значення PI: 3.14358865958578710
Кількість членів ряду: 501
Введіть значення eps: 0.01
Наближене значення PI: 3.16119861298705020
Кількість членів ряду: 51
Введіть значення eps: кінець
```

```
Рекурсивне обчислення значення числа pi за
    другою формулою.
Введіть значення eps: 1e-11
Наближене значення PI: 3.14159265358479710
Кількість членів ряду: 3684
Введіть значення eps: 1e-7
Наближене значення PI: 3.14159270271861150
Кількість членів ряду: 171
Введіть значення eps: 0.001
Наближене значення PI: 3.14125482360776460
Кількість членів ряду: 8
Введіть значення eps: кінець
```
,

При ітераційному обчисленні наближеного значення числа π з точністю 10^{-7} за першою формулою було виконано 5000001 (5 млн.) кроків циклу, за другою — 171 крок і в обох випадках одержано 6 правильних знаків після коми. При використанні рекурсивних функцій результати повинні бути такими самими, як і при використанні циклів. Але максимально можлива кількість рекурсивних викликів відповідних функцій (на моєму комп'ютері при стандартних налаштуваннях) коливається в межах від 3975 до 3979, і в зв'язку з цим при обчисленнях за першою формулою максимальною є точність 0,00012575 (зроблено 3977 рекурсивних викликів), при якій знайдено наближене значення тільки з трьома правильними знаками після коми — 3,*1418*. При обчисленнях за другою формулою використання рекурсивної функції дало ті самі результати, що й застосування циклу. Крім того, при 3684 рекурсивних викликах за другою формулою з точністю 10^{-11} знайдено наближене значення 3,*141592653584* з 11 правильними знаками після коми, але вже при заданні точності 10^{-12} стек переповнюється.

Приклад 7.5.8. У прикладі 6.4.2 подано алгоритм обчислення наближеного значення функції $\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$ з використанням циклу. Аналогічний рекурсивний алгоритм, який реалізує рекурентне співвідношення для обчислення загального члена ряду $a_n = \frac{-x^2}{2 \cdot n \cdot (2n+1)} a_{n-1}$ (при $a_0 = x$) при рекурсивному спускові і накопичення суми при рекурсивному підйомі, можна подати, наприклад, так (глобальні змінні не використовуються; обчислення припиняється, як тільки значення загального члена ряду стане меншим від заданого досить малого числа *eps*):

```
/* Обчислення наближеного значення степеневого ряду
   (функція sin(x)) з використанням рекурсії */
...
double sin_eps(int *n, double an, double r, double eps, int k);
int main() {
...
  n=0;
  an=x;   // при n=0 перший член ряду a0=x,
  s=an;    // початкове значення суми s=a0=x,
  r=-x*x/2.;    // множник -x*x/2. не залежить від n
  s+=sin_eps(&n, an, r, eps, k);
...
}
```

```
Функція  sin(x)
ряд E(n=0..+00)(-1)^n*(x^(2n+1))/((2n+1)!)
Введіть значення x 1.57
  введіть eps 1e-12
  введіть ліміт кількості ітерацій k 100
результат:
  кількість ітерацій n= 9
  наближене значення функції s= 0.999999683
  точне значення функції f= 0.999999683
  похибка |f-s|= 0.000000000
```

```
/* Рекурсивне обчислення наближеного значення функції
   sin(x) з точністю eps, починаючи з другого доданка */
double sin_eps(int *n, double an, double r, double eps, int k) {
  double s;    // накопичує результат обчислення
  (*n)++;   // номер ітерації (рекурсивного виклику)
```

```
  /* Обчислення наступних членів ряду */
  an*=r/(*n)/(2.*(*n)+1.);
  if (fabs(an)>=eps && (*n)<k)    // умова продовження рекурсії
    s=sin_eps(n, an, r, eps, k);    // рекурсивне звернення
  else
    s=0;    // щоб при останньому зануренні повернути тільки an
  return s+an;  // повернення накопичуваної суми доданків
                // (від останнього доданка до другого)
}
```

При рекурсивному спускові кожен раз на початку роботи функції sin_eps обчислюється значення номера ітерації *n* (номера рекурсивного виклику) і останнє його значеня визначає глибину рекурсії. При спускові також обчислюється значення кожного наступного члена ряду a_n, а при підйомі воно додається до суми *s*.

У цьому прикладі, як і в прикладі 6.4.2, перший доданок, абсолютна величина якого, менша від *eps*, ввійде в суму; щоб цей доданок не входив у суму, можна використати такий фрагмент алгоритму:

```
  if (fabs(an)>=eps && (*n)<k)    // умова продовження рекурсії
    s=sin_eps(n, an, r, eps, k)+an;    // рекурсивне звернення
  else
    s=0;    // щоб при останньому зануренні не повертати an
  return s;  // повернення накопичуваної суми доданків
```

При знаходженні наближеної суми ряду використання рекурсивної функції має ту перевагу, що сума накопичується, починаючи з найменшого значення, а це дає точніший результат (див. приклад 6.3.4).

Підсумовуючи сказане, можна зробити певні висновки щодо використання рекурсивних алгоритмів:

— на практиці для простих задач доцільно використовувати нерекурсивні методи розв’язування;

— перед використанням рекурсії треба спочатку впевнитися в тому, що задачу неможливо простіше розв’язати за допомогою ітераційних методів;

— рекурсію варто використовувати тільки тоді, коли нерекурсивний підхід сильно ускладнює програмування або не призводить до сильного зменшення часу роботи алгоритму, обсягу пам'яті тощо.

Але є випадки, коли застосування рекурсивних алгоритмів виправдане. Це, насамперед, задачі штучного інтелекту, також без рекурсії важко реалізувати алгоритм швидкого сортування, обробляти стеки, дерева, графи тощо. Перевагою рекурсії є те, що більшість складних алгоритмів при її використанні можна записати чіткіше й коротше, ніж з використанням циклів. Рекурсія в поєднанні з іншими методами (наприклад, методом розділяй і володарюй) дає можливість будувати ефективні програми.

7.6. Складність і оптимізація алгоритмів модульної структури

Як було сказано раніше (див. п. 7.3), при зверненні до програмного модуля виконуються дії, пов'язані з виділенням пам'яті в стекові викликів. Час на виділення такої пам'яті і її обсяг залежать від кількості і типів параметрів і локальних змінних модуля. За рахунок цього потрібний обяг пам'яті і час виконання програм модульної структури зростають. Особливо сповільнення роботи відчутне при застосуванні рекурсії; крім того, рекурсивні алгоритми можуть мати велику ємнісну складність, що навіть може призвести до досить швидкого переповнення стека викликів, тому глибина рекурсивних викликів не повинна перевершувати максимального розміру стека.

Ємнісна складність модуля, який реалізує певний алгоритм, порівняно з використанням цього алгоритму в програмі збільшується за рахунок створення копій параметрів і зберігання адреси повернення і адреси значення функції в стековій пам'яті; локальні змінні також можуть дублювати деякі змінні викликаючого модуля.

Таким чином, ємнісна складність модуля

$$V_{мод} = V_{алг} + V_{ст},$$

де $V_{алг}$ — ємнісна складність алгоритму, $V_{ст}$ — обсяг пам'яті, виділеної в стекові викликів.

Як і для ємнісної складності, при визначенні часової складності модуля треба врахувати витрати часу на виділення стекової пам'яті і зберігання в ній інформації, а також на її звільнення, перехід за адресою повернення, розміщення значення функції за адресою значення функції.

Таким чином, часова складність модуля

$$t_{мод} = t_{алг} + 2t_{ст},$$

де $t_{алг}$ — часова складність алгоритму, $t_{ст}$ — час на виділення/звільнення (якщо вважати всі пов'язані з цими діями операції елементарними і однаковими за часом виконання) пам'яті в стекові викликів.

Якщо модуль рекурсвний, то дії, пов'язані зі стеком викликів, будуть виконуватися кількість разів, рівну кількості рекурсивних викликів. Нова пап'ять для збегігання команд алгоритму використовуватися не буде, але команди алгоритму будуть виконуватися повторно. Таким чином:

$$V_{мод.рек} = V_{алг} + nV_{ст},$$

$$t_{мод.рек} = n \cdot (t_{алг} + 2t_{ст}),$$

де n — загальна кількість рекурсивних викликів.

Приклад 7.6.1. Для розглянутого у прикладі 7.5.2 рекурсивного алгоритму обчислення факторіалу, часову складність можна визначити так. Здійснюється виклик функції при n і виділяється стекова пам'ять — складність $t_{ст}$. Перевірка умови має складність t_l (у мікропроцесорах VIA Nano 3000 операція порівняння виконується за 2 такти — див. таблицю 4.5). Якщо виконується умова, то здійснюється рекурсивний спуск — виклик функції при $n-1$ і перевірка умови і т.д. Усіх викликів функції буде n; отже, складність при рекурсивному спускові становить $n \cdot (t_{ст} + t_l)$. Якщо умова не виконується, то виконується присвоювання, складність якого $t_{пр}$ (2-3 такти), і звільнення стека зі складністю $t_{ст}$ — загальна часова складність $t_{пр} + t_{ст}$. При рекурсивному підйомі $(n-1)$ раз виконується множення з часовою складністю t_m (3-6 тактів), присвоювання результату зі складністю

$t_{пр}$ і звільнення стека зі складністю $t_{ст}$. Часова складність підйому дорівнює $(n-1)\cdot(t_{т}+t_{пр}+t_{ст})$.

Таким чином, загальна часова складність виконання рекурсивної функції обчислення факторіалу становить:

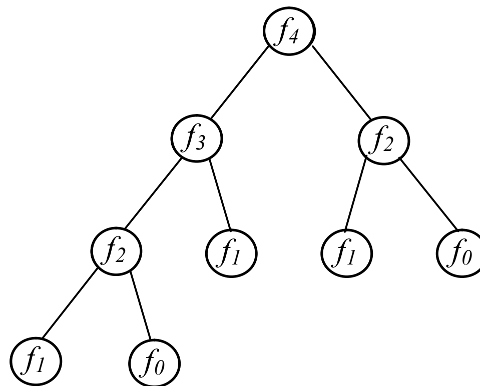
$$t_{факт}=n\cdot(t_{ст}+t_{l})+t_{пр}+t_{ст}+(n-1)\cdot(t_{т}+t_{пр}+t_{ст})=$$
$$=n\cdot(2t_{ст}+t_{l}+t_{т}+t_{пр})-t_{т},$$

що є величиною $O(n)$.

Приклад 7.6.2. Якщо розглянути рекурсивний алгоритм знаходження чисел Фібоначчі (див. приклад 7.5.3), то в ньому кількість рекурсивних викликів $\approx 2^{n-2}$, що вказує на експоненційний час виконання:

$$t_{Фібоначчі}\approx 2^{n-2}\cdot(t_{алг}+2t_{ст})=O(2^{n}).$$

Дерево викликів, яке відповідає операторові `fn=fib(n-1)+fib(n-2);` цього алгоритму, наприклад, при $n=4$ має 9 вершин і 8 ребер (кожне ребро дерева відповідає рекурсивному зверненню):



Максимальним числом Фібоначчі типу int є число $f_{45}=1836311903$ (максимальне значення типу int $2^{31}-1=2147483647$), для його обчислення за допомогою рекурсивного алгоритму здійснено 3672623805 викликів функції, які виконувалися більше ніж 3 хвилини.

Проте, якщо використати *мемоїзацію* (зберігання результатів викликів функції і повернення збереженого результату, якщо виклики на однакових вхідних даних відбуваються повторно), то часова складність обчислення чисел Фібоначчі з використанням рекурсивної функції буде лінійною $O(n)$

(ітераційний алгоритм теж має часову складність $O(n)$, проте працює швидше від рекурсивного).

Для зберігання вже обчислених значень чисел Фібоначчі можна скористатися масивом (див. розд. 8). Програмна реалізація рекурсивного алгоритму з використанням мемоїзації така:

```
#include <stdio.h>
#include <stdlib.h>
int fib(int n);
int memo[46]={1,1,0};   // запам'ятовування обчислених значень
long long int k;   // кількість викликів рекурсивної функції
int main() {
  int n;    // номер числа Фібоначчі
  k=0;
  system("chcp 1251 & cls");
  printf("Вкажіть номер числа Фібоначчі,\
    \n яке треба обчислити  ");
  scanf("%d", &n);
  if (n<=45)
    printf("f%-2d= %-15d",n,fib(n));
  else
    printf("Номер числа перевищує 45\n");
  printf("\nКількість викликів k=%lld",k);
  printf("\n\n");
  system("pause");
  return 0;
}
/* Рекурсивне обчислення чисел Фібоначчі */
int fib(int n) {
  int fn;
  k++;   // обчислення кількості звернень до функції
  if (n==0 || n==1)
    fn=1;
```

```
  else
    if (memo[n])     // значення обчислено раніше
      fn=memo[n];
    else {    // значення треба обчислити
      fn=fib(n-1)+fib(n-2);
      memo[n]=fn;    // запам'ятовування обчисленого значення
    }
  return fn;
}
```

```
Вкажіть номер числа Фібоначчі,
 яке треба обчислити  45
f45= 1836311903
Кількість викликів k=89
```

При використанні мемоїзації час обчислення значення $f_{45} = 1836311903$ був миттєвим (меншим від 1 секунди), зроблено 89 викликів функції.

Таким чином, мемоїзація дала можливість перейти від експоненційного часу виконання $O(2^n)$ рекурсивного алгоритму до лінійного $O(n)$. При цьому трохи збільшився обсяг потрібної пам'яті, що пов'язано зі зберіганням масиву обчислених значень.

Так само, як і в розглягутому прикладі, якщо звернення до підпрограми відбувається багато разів (і не обов'язково рекурсивне) з тими самими даними, то можна запам'ятати всі обчислені її значення і в подальшому їх використовувати.

При використанні модулів до ємнісних і часових складностей самих алгоритмів додаються ще витрати, пов'язані з виділенням стекової пам'яті, що стає особливо відчутним при використанні рекурсії. Але, не зважаючи на це, прозорість і структурованість логіки програм є великою перевагою модульного програмування.

Запитання та завдання для самоконтролю

1. Схарактеризуйте модуль і модульне програмування. Вкажіть переваги й недоліки використання модулів. Вкажіть основні властивості модулів. Що забезпечує міжмодульну взаємодію? Які модулі називають повністю закритими, а які частково закритими? Чи сприяє використання модулів спрощенню логіки програми?

2. Схарактеризуйте проектування й програмування згори вниз і знизу вгору. Якому способові проектування і програмування слід надавати перевагу і чому? У якому випадку зменшується вартість розробки програм? На чому базується принцип модульності програмування?

3. Які параметри називають фактичними, а які формальними? Для чого використовують стек викликів? Яку інформацію містить стек викликів? Що відбувається при зверненні до модуля? Що відбувається після закінчення роботи модуля? Опишіть механізми передачі параметрів за значенням і за посиланням (адресою). Що таке локальні змінні і константи модуля, коли для них виділяється пам’ять і коли вона звільняється? У яких модулях доступні локальні змінні і константи?

4. Які мови програмування називають процедурно-орієнтованими? Як оформляють модулі в мові програмування С? Як здійснюється обмін даними між функціями мови С? Які функції називають функціями з побічним ефектом?

5. Що таке рекурсія? Який алгоритм називають рекурсивним? Яку підпрограму називають прямо рекурсивною? Яку підпрограму називають непрямо рекурсивною? Що таке, рекурсивний спуск, рекурсивний підйом, глибина рекурсії? Завдяки чому можлива реалізація рекурсії в мовах програмування? Що відбувається при рекурсивному зверненні до підпрограми? Чи створює рекурсивний виклик нову копію підпрограми? Чи відрізняється звернення до рекурсивної підпрограми від звернення до звичайної підпрограми? Чи завжди доцільно використовувати рекурсивні алгоритми і чому?

6. Чи збільшуються витрати пам’яті і часу при роботі програм модульної структури і, якщо так, то за рахунок чого? Чи залежать витрати пам’яті і часу від глибини рекурсії? Як обчислити ємнісну і часову складності звичайного модуля? Як обчислити ємнісну і часову складності рекурсивного модуля? Що таке мемоїзація? У чому полягає оптимізація алгоритму з використанням мемоїзації?

Практичне завдання № 11 «Алгоритми модульної структури»

Мета виконання завдання: навчитися розробляти алгоритми модульної структури і програмувати їх з використанням підпрограм (функцій мови С).

Завдання. Алгоритм, розроблений у практичному завданні № 10, подати з використанням модулів: введення даних і перевірки їхньої правильності, обчислення наближеного значення функції (наближеного значення ряду), обчислення точного значення функції, виведення результатів; керуючого модуля, який визначає порядок звернення до вказаних модулів.

Алгоритм подати програмно. Обмін даними здійснювати через параметри без використання глобальних змінних. У кожній підпрограмі після її заголовка в коментарях вказати для всіх змінних і параметрів, які в ній використовуються, які змінні локальні, які параметри передаються за значенням, які за адресою. Текст програми структурувати. Коментарі обов’язкові — 17-25 %. Провести розрахунки не менш ніж з трьома різними наборами вхідних даних.

Вказівки. Як зразки використати приклади 7.4.1-7.4.4.

Функція $f(x)$ та сама, що і в практичному завданні № 10.

Практичне завдання № 12 «Рекурсивні алгоритми»

Мета виконання завдання: навчитися розробляти і програмувати рекурсивні алгоритми.

Завдання. У розробленому в практичному завданні № 11 алгоритмі замінити рекурсивними модуль введення і перевірки правильності введених даних даних і модуль обчислення наближеного значення функції. Модулі для обчислення точного значення функції і виведення результатів залишити без змін.

Алгоритм подати програмно. Обмін даними між підпрограмами здійснювати через параметри без використання глобальних змінних. Текст програми структурувати. Коментарі обов’язкові — 17-25 %. Провести розрахунки не менш ніж з трьома різними наборами вхідних даних.

Вказівки. Як зразки використати приклади 7.5.4 і 7.5.8.

Функція $f(x)$ та сама, що і в практичному завданні № 10.

8. Обробка одновимірних масивів

Поняття масиву; одновимірний масив. Основні типові алгоритми роботи з одновимірними масивами. Пошук елемента в масиві. Сортування масивів. Приклади алгоритмів з використанням одновимірних масивів. Складність і оптимізація алгоритмів обробки одновимірних масивів.

8.1. Поняття масиву. Одновимірний масив

Досі в алгоритмах використовувалися прості структури даних.

Під структурою даних в загальному випадку розуміють множину елементів даних і множину зв'язків між ними. Розрізняють прості й інтегровані структури даних.

Простими (ще кажуть: базовими) є структури даних, які безпосередньо підтримуються процесором. Для таких даних існує достатньо великий набір команд процесора, які оперують з ними «за один захід». З точки зору фізичної структури даних (їхнього подання в пам'яті комп'ютера) для простого типу чітко визначено його розмір і спосіб розміщення в пам'яті комп'ютера. З логічної точки зору прості дані є неподільними одиницями. До базових структур належать такі дані: символи, дані цілого й дійсного типів, вказівники.

Інтегрованими (ще кажуть, складними, структурованими) називають такі структури даних, складовими частинами яких є інші структури даних прості чи інтегровані. Прикладами таких структур є масиви, рядки, структури (записи), списки, черги, дерева тощо.

Масив — це структура даних, яка складається з елементів того самого типу, що розміщуються в послідовних сусідніх ділянках пам'яті єдиним блоком. Найменшу адресу має перший елемент, а найбільшу — останній елемент.

Масиви можна задавати *статично* (кількість елементів масиву вказується при його визначенні в тексті програми; компілятор виділяє необхідну пам'ять; надалі під час виконання програми кількість елементів не змінюється) і *динамічно* (масив не має фіксованої кількості елементів і може створюватися під час виконання програми, що дає можливість працювати з

масивами довільного розміру, не вносячи змін у текст програми). Надалі майже у всіх алгоритмах буде використано статичні масиви.

До всього масиву цілком можна звертатися по імені. Доступ до кожного окремого елемента масиву прямий — можна вибирати будь-який елемент масиву, використавши індексування.

У мові C немає перевірки правильності індекса елемента. Вихід за границі масиву може призвести до непередбачуваних наслідків (наприклад, можна записати значення в будь-яку змінну, яка не належить масиву, або навіть в код програми). Робота щодо перевірки границь масиву покладається на програміста. Тому, розробляючи алгоритми обробки масивів з використанням мови C, треба уважно слідкувати, щоб значення індекса елемента не виходило за межі відведеної для масиву ділянки пам'яті.

Масиви можуть мати одну або кілька вимірностей. При цьому одновимірні масиви можна інтерпретувати як вектори (насправді це списки інформації одного типу), а двовимірні — як матриці.

Опис одновимірного масиву в мові C має вигляд:

тип ім'я_змінної[розмір];

Тут *тип* вказує базовий тип масиву і є типом кожного елемента масиву. Параметр *розмір* задається виразом-константою. Константа може бути числова або може бути визначена директивою препроцесора (наприклад, `#define N 5`), але не може бути оголошеною константою (наприклад, `const int k=2;`) і визначає, скільки елементів містить масив. Наприклад, `int f[7]; double x[N+1]; char c[2*N+1];`, де константу N описано директивою #define.

Масиви досить тісно пов'язані з вказівниками. Ім'я масиву — це вказівник на ділянку пам'яті, в якій розміщується перший елемент масиву *a*[0].

Динамічне виділення пам'яті під масиви. При динамічному виділенні пам'ять резервується не на етапі компіляції, а на етапі виконання програми, що дає можливість ефективніше її використовувати. Якщо пам'ять для масиву буде виділятися динамічно, то масив треба описати, як вказівник. Для динамічного виділення пам'яті використовують функцію malloc (параметром функції є обсяг пам'яті; якщо пам'яті не вистачає, то функція повертає значення 0; операційна система записує службову інформацію перед коміркою, на яку вказує повернутий функцією вказівник) або функцію calloc (працює так само, як і функція malloc, але має два параметри: кількість елементів і розмір одного елемента; також вона обнуляє пам'ять), а для звільнення виділеної пам'яті — функцію free (параметром є вказівник на початок виділеної ділянки). Функція realloc змінює обсяг виділеної функцією malloc, calloc або realloc пам'яті (параметрами є вказівник на початок виділеної ділянки і новий обсяг пам'яті; якщо вказаний в параметрі обсяг пам'яті більший від раніше виділеного і поряд недостатньо пам'яті, щоб утворити неперервну ділянку, то виділяється нова ділянка і туди копіюються існуючі дані). Наприклад:

```
int *a;   // масив описано, як вказівник
  ...
  a=(int *)malloc(n*sizeof(int));    // виділення пам'яті під масив
  if (!a)   // перевірка, чи виділено пам'ять?
    printf("  Недостатньо пам'яті");
  else // виконання алгоритму (звичайна робота з масивом)
  free(a);   // звільнення місця в пам'яті
```

Можливості роботи з масивами. 1) Повний розмір одновимірного масиву (обсяг пам'яті комп'ютера, яку займає масив) в байтах обчислюється так:

sizeof(базовий тип)×кількість елементів.

Наприклад, для масиву int a[5] обсяг пам'яті обчислюється за допомогою виразу: sizeof(int)×5 і дорівнює 20 байтів.

Обсяг пам'яті, яку займає масив, можна визначити за допомогою операції `sizeof(a)`; кількість елементів можна обчислити `k=sizeof(a)/sizeof(a[0])`.

2) Доступ до елементів масиву прямий. Усі його елементи однаково доступні. Доступ до окремого елемента масиву здійснюється за допомогою імені масиву й індексу. Індекс елемента масиву подається в квадратних дужках після імені. Індексом елемента може бути будь-який вираз цілого типу int, типу char чи типу enum. У всіх масивів індексація елементів починається з індексу 0. Наприклад, масив int a[5] має п'ять таких елементів: перший — a[0], другий — a[1], …, п'ятий — a[4].

3) Кожен елемент масиву може використовуватися в будь-якому контексті, де може використовуватися змінна відповідного типу. Наприклад, можна записати оператори: `s=a[i]+a[j]; printf("%d%d", a[3], a[4]); scanf("%d", &a[3]); printf("%d", a[2*i-1]); y=cos((double)a[i])-a[j]; printf("%s", name[Mon])` (де Mon — enum-константа) тощо.

4) Ініціалізація масиву. При оголошенні статичного масиву його елементам можна присвоїти потрібні початкові значення-константи (тобто ініціалізувати масив), тип яких повинен бути сумісним з типом масиву:

```
  int x[5] = {1, 2, 3, 4, 5};
```

Якщо при цьому вказати значення не всіх елементів масиву, а тільки кількох перших (хоч би одного першого), то всі решта ініціалізуються нулем. Наприклад:

```
  int x[5] = {1}; // еквівалентно int x[5] = {1,0,0,0,0};
```

то елементи будуть мати такі значення x[0]=1, x[1]=0, x[2]=0, x[3]=0, x[4]=0.

Обнулити всі елементи масиву можна за допомогою ініціалізації:

```
  int x[5] = {0}; // еквівалентно int x[5] = {0,0,0,0,0};.
```

При оголошенні масиву можна не вказувати його розмір, але при цьому обов'язково потрібно задати початкові значення. Такий масив називають *безрозмірним*. Компілятор визначає розмір масиву автоматично:

```
  int a[ ] = {1, 2, 3};   // автоматичне визначення розміру масиву.
```

Ініціалізація безрозмірних масивів не тільки полегшує програмування, а й дає можливість змінювати кількість елементів, не змінюючи розміру масиву. Це особливо корисно при роботі з рядками символів, які є масивами символів (не треба підраховувати кількість символів).

Ініціалізувати масив можна в циклі, наприклад, значенням 0:

```
  for (i=0; i<N; i++) b[i]=0;
```

Ініціалізувати масив можна також функцією memset(b,0,sizeof(b)).

При тестуванні програм для заповнення масиву можна скористатися генератором випадкових (псевдовипадкових) чисел.

Приклад 8.1.1. Для заповнення одновимірного масиву випадковими числами можна використати такий фрагмент програми мовою C:

```
#include <stdlib.h>    // функції srand, rand
#include <time.h>   // функція time
...
  /* З проміжка -50...50 */
  srand(time(0));
```

```
for (i=0; i<N; i++) a[i]=rand()%101-50;
```

У Visual Studio максимальне згенероване число визначається константою `RAND_MAX = 32767`, мінімальне — `0`.

Генерація цілого числа здійснюється так:

від min до max включно: `n = rand()%(max-min+1)+min;`

(наприклад, числа від 0 до 10: `rand()%11` ; числа –1, 0 чи 1: `rand()%3-1`).

Для генерації дійсних чисел можна застосувати формулу:

`a=(float)rand()/RAND_MAX*(max-min)+min .`

5) Введення й виведення масивів здійснюється поелементно. При введенні елементів з клавіатури значення можна набирати через пробіли, Enter чи Tab:

```
/* Введення й виведення елементів одновимірного масиву */
#define N 5
...
  int a[N];
  int i;
  /* Введення елементів масиву */
  for (i=0; i<N; i++)
    scanf("%d",&a[i]);
  /* Виведення елементів масиву */
  for (i=0; i<N; i++)
    printf("a[%d]= %d;   ",i,a[i]);
...
```

```
3 4
1
0
-3
a[0]= 3;   a[1]= 4;   a[2]= 1;   a[3]= 0;   a[4]= -3;
```

6) Копіювання й порівняння масивів здійснюється поелементно:

```
/* Копіювання масивів */
  for (i=0; i<N; i++) a[i]=b[i];
/* Порівняння масивів */
  ind=1;   // вважається, що масиви рівні
  for (i=0; i<N && ind; i++)
    if (a[i]!=b[i])
      ind=0;
  if (ind)
    printf("Масиви рівні");
  else
    printf("Масиви не рівні");
```

7) Зв’язок між масивами і вказівниками — у мові С масиви і вказівники тісно пов’язані. Так, ім’я масиву є вказівником, і, навпаки, вказівник можна індексувати, як масив. Наприклад, нехай є вказівник p і масив mas:

```
  int *p, mas[5];
  p=mas;   // присвоювання вказівнику адреси масиву
  p[4]=50;   // присвоювання за допомогою індексу — п’ятий
             // елемент mas[4] масиву mas набуває значення 50 */
  *(p+4)=50;   // присвоювання за допомогою адресної арифметики –
               // п’ятий елемент mas[4] масиву mas набуває значення 50
  mas[4]=50;   // п’ятий елемент mas[4] масиву mas набуває значення 50
  *(mas+4)=50;   // п’ятий елемент mas[4] масиву mas набуває значення 50
```

Таким чином, усі чотири оператори присвоювання еквівалентні — виконують ту саму дію: присвоюють п’ятому елементу масиву `mas` значення 50.

Якщо використовувати звернення до елементів масиву `mas[4]`, то програми виходять зрозумілішими, читабельнішими, ніж при використанні вказівників `*(mas+4)`

(при цьому, коли компілятор аналізує вираз типу `mas[i]`, то він перетворює його до виду `*(mas+i)`).

Зауваження. Оскільки вирази a[i] і *(a+i) здійснюють доступ до того самого елемента масиву, то a[i] = *(a+i). Таким чином доступ до i-го елемента масиву можна здійснити й так: i[a] (бо i[a]=*(i+a))

8) Передача параметрів-масивів. Ім'я масиву є вказівником на його перший елемент. Так, якщо масив оголошено int mas[5], то вказівником на його перший елемент є ім'я mas. Адресу першого елемента масиву також можна подати за допомогою операції взяття адреси &: вирази mas і &mas[0] еквівалентні (вираз `mas==&mas[0]` є завжди істинним), але другий варіант не прийнято використовувати у програмуванні.

У мові С весь масив не можна передати як параметр функції, але можна передати вказівник на масив — ім'я масиву без індексів. Наприклад:

```
int main() {
  int mas[5];
  func(mas);      // або func(&mas[0]);
...  }
```

Якщо параметром функції є одновимірний масив, то її формальний параметр можна описати трьома способами:

а) як вказівник — `void func(int *mas);`

б) як масив фіксованого розміру — `void func(int mas[5]);`

в) як масив невизначеного розміру — `void func(int mas[]);`

Ці оголошення еквівалентні, оскільки вони вказують на те саме: в функцію передається вказівник на цілочисельну змінну (на перший елемент масиву). При цьому, як правило (але все залежить від компілятора), розмір масиву, який передається в функцію, не має ніякого значення, оскільки перевірка виходу за межі масиву в мові С не здійснюється.

Продемонструємо всі три способи опису формальних параметрів (все точнісінько те саме, тільки відрізняється заголовок функції). Як видно з наведених прикладів, робота з елементами масиву в головній функції і в підпрограмах-функціях нічим не відрізняється. Доступ до елемента здійснюється за допомогою індексу mas[i] або за допомогою адресної арифметики `*(mas+2)`.

```
/* Одновимірні масиви.
   Передача параметра-масиву */
#include <stdio.h>
#include <stdlib.h>
#define N 2
void func(int *mas);
// або void func(int mas[5]);
// або void func(int mas[]);
int main() {
  int i;
  int mas[2*N+1]={1,4,7,8,9};
  *(mas+2)=1000;   // те саме, що і mas[2]=1000;
  func(mas);
  printf("main:");
  for (i=0; i<2*N+1; i++)
    printf("mas[%d]=%d; ",i,mas[i]);
  printf("\b\b.");
  printf("\n\n");
  system("pause");
  return 0;
}
void func(int *mas) {
  // або void func(int mas[5]) {
  // або void func(int mas[]) {
```

```
  int i;
  printf("func:");
  for (i=0; i<2*N+1; i++)
    printf("mas[%d]=%d; ",i,mas[i]);
  printf("\b\b.\n");
  *(mas+2)=-200;   // те саме, що і mas[2]=-200;
}
```

```
func:mas[0]=1; mas[1]=4; mas[2]=1000; mas[3]=8; mas[4]=9.
main:mas[0]=1; mas[1]=4; mas[2]=-200; mas[3]=8; mas[4]=9.
```

Якщо треба написати надійну програму, в якій функція не повинна змінювати значення елементів масиву, то краще в заголовку цієї функції оголошувати параметр-вказівник як константу, наприклад:

```
int func(const int *mas);
```

У цьому випадку будь-яка спроба змінити значення за адресою, яка міститься в константному вказівнику, буде приводити до помилки компіляції.

9) Робота з рядками символів. На відміну від інших мов мова C не має спеціального типу даних — рядки символів. У мові C рядок символів моделюється за допомогою одновимірного масиву символів. Найважливішою особливістю для масиву, який зберігає рядок символів, є те, що після всіх символів у кінці рядка записується спеціальний нуль-символ '\0' з числовим кодом 0, який відіграє роль ознаки кінця рядка.

При виведенні рядка на екран функцією printf використовується формат %s (наприклад: `char a[5]="abcd"; printf("Текст %s", a);`). Виведення окремого символа рядка здійснюється за форматом %c (наприклад: `printf("%c", a[i]);`).

У функції scanf для введення рядків символів, як і в функції printf, використовується формат %s. При цьому зчитуються всі введені символи до першого пробільного символа (пробіл ' ', Tab '\t', Enter '\n'). Якщо для рядка відведено менше пам’яті, ніж вводиться символів, то, оскільки мова C не підтримує автоматичного контролю виходу за границю масиву, символи будуть записуватися не тільки в пам’ять, виділену під масив, а й у комірки пам’яті безпосередньо після масиву і, отже, можуть зіпсувати значення інших змінних або навіть коди команд. Щоб запобігти цьому, треба в форматі вводу вказати граничну довжину рядка (на 1 меншу від розміру рядка або ще меншу). Наприклад:

```
  char b[10];
  scanf("%9s",b);  // вводяться символи до першого пробільного, але не більше 9
```

решта набраних користувачем символів залишається в буфері вводу. Треба також враховувати, що scanf залишає в буфері пробільний символ.

Існують інші функції для введення й виведення рядків символів і для роботи з ними.

8.2. Основні типові алгоритми роботи з одновимірними масивами

Розглянемо алгоритми роботи з одновимірними масивами, які стосуються знаходження мінімального й максимального елемента, перестановки елементів масиву й циклічних зсувів.

Нехай масив містить *n* елементів і його описано, наприклад, так (у програмах константи, визначені директивою define, прийнято позначати великими буквами, хоч це не обов’язково — див. п. 1.4):

```
#define N 10
  int a[N];   // масив, який має N елементів
```

У мові C при роботі з масивами треба розрізняти індекс і номер елемента:

індекс елемента набуває значень від 0 до $n-1$;

номер елемента набуває значень від 1 до *n*.

При обробці масиву також розглядають його першу й другу половини, які у випадку непарної (при цьому є серединний елемент, який не належить жодній половині) й парної кількості елементів визначаються так:

$$\underbrace{1\ 2\ 3}_{\text{перша}}\ 4\ \underbrace{5\ 6\ 7}_{\text{друга}} \qquad \underbrace{1\ 2\ 3}_{\text{перша}}\ \underbrace{4\ 5\ 6}_{\text{друга}}$$

Приклад 8.2.1. Алгоритм ***знаходження мінімального елемента*** масиву.

Програмна реалізація алгоритму:

```
/* Пошук першого найменшого елемента масиву */
...
  int a[N];   // масив елементів
  int k;   // індекс мінімального елемента
  int i;   // індекс поточного елемента
  int amin;   // мінімальне значення
  amin=a[0]; k=0;
  for (i=1; i<N; i++)
    if (a[i]<amin) {
      amin=a[i]; k=i;
    }
  printf("Мінімальний елемент a[%d]=%d", k, amin);
...
```

Треба звернути увагу на те, що в алгоритмі перед циклом як мінімальне задано значення першого елемента масиву, а в циклі розглядаються елементи, починаючи з другого (оскільки порівнювати значення саме́ з собою не треба). Цикл буде виконуватися $(n-1)=O(n)$ разів.

Якщо в масиві є кілька однакових мінімальних елементів і якщо використано умову `a[i]<amin` (строга нерівність), то буде знайдено індекс першого мінімального елемента. Наприклад, для масиву a[N]={2,4,1,8,9,1,3,9,1,9} буде знайдено a[2]=1. Якщо використати умову `a[i]<=amin` (нестрога нерівність), то буде знайдено індекс останнього мінімального елемента — a[8]=1.

Аналогічний алгоритм можна записати для ***знаходження максимального елемента*** масиву (змінивши в порівняннях знак на протилежний — знак < на знак > і <= на >=):

```
#define N 10
/* Пошук першого найбільшого елемента масиву */
...
  int a[N];   // {2,4,1,8,9,1,3,9,1,9};
  int k;   // індекс максимального елемента
  int i;   // індекс поточного елемента
  int amax;   // максимальне значення
  amax=a[0]; k=0;
  for (i=1; i<N; i++)
  if (a[i]>amax)
      amax=a[i]; k=i;
    }
  printf("Максимальний елемент a[%d]=%d",k,amax);
...
```

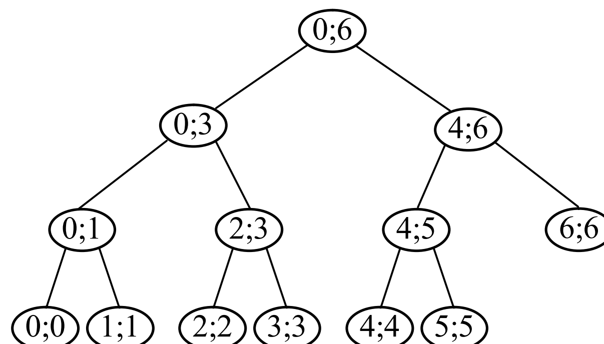
Для пошуку мінімального елемента масиву можна також скористатися рекурсивним алгоритмом. Алгоритм базується на діленні масиву навпіл: якщо масив поділити на дві частини, в обох частинах знайти мінімальні елементи і взяти менший з них, то це і буде мінімальний елемент масиву; якщо масив містить один елемент, то цей елемент є найменшим.

Програмна реалізація рекурсивного алгоритму пошуку мінімуму:

```
/* Рекурсивне знаходження мінімального елемента масиву */
...
#define N 7
int minimum(int a[N], int n1, int n2);
int main() {
  int a[N]={100,21,40,0,10,0,1};
  printf("min=%d", minimum(a, 0, N-1));
...
}
```

```
/* Рекурсивний алгоритм пошуку першого мінімального елемента
   в одновимірному масиві. З масиву елементів a[N] береться
   підмасив елементів з індексами від n1 до n2 і ділиться
   навпіл, доки не залишиться один елемент (з індексом
   n1=n2). Для кожного з двох підмасивів визначається
   мінімальний елемент min1 і min2, із яких береться менший */
int minimum(int a[N], int n1, int n2) {
  int min1, min2, // мінімуми першої і другої половин масиву
      min;   // загальний мінімум
  int nm;    // середина підмасиву
  if (n1==n2)   // у підмасиві один елемент
    min=a[n1];
  else {   // у підмасиві більше одного елемента
    nm=(n1+n2)/2;
    min1=minimum(a, n1, nm);
    min2=minimum(a, nm+1, n2);
    if (min1<min2)   // визначення меншого значення з двох
      min=min1;
    else
      min=min2;
  }
  return min;
}
```

При рекурсивному знаходженні мінімуму виконується $(2n-1)=O(n)$ звернень до функції. Дерево рекурсивних викликів, наприклад для $n=7$ (індекси елементів змінюються від 0 до 6), має вигляд:



Тут у вершинах дерева записано граничні індекси елементів підмножин — значення параметрів n1 і n2.

Обхід цього бінарного дерева програмою здійснюється так:

```
(0;6) (0;3) (0;1) (0;0) (1;1) (2;3) (2;2) (3;3) (4;6) (4;5) (4;4) (5;5) (6;6)
```

,

тобто згори вниз. Таким чином, як і у випадку ітераційного алгоритму, якщо в масиві є кілька однакових мінімальних елементів і якщо використано умову min1<min2 (строга нерівність), то буде знайдено індекс першого мінімального елемента; якщо використати умову min1<=min2 (нестрога нерівність), то буде знайдено індекс останнього мінімального елемента. Наприклад, для масиву a[N]={100,21,40,0,10,0,1} буде відповідно знайдено a[3]=0 і a[5]=0.

Аналогічний рекурсивний алгоритм можна записати для знаходження максимального елемента масиву, змінивши при порівнянні *min1* і *min2* знак на протилежний — знак < на знак > і <= на >=.

Хоч ітераційний і рекурсивний алгоритми мають однакову часову складність $O(n)$, проте рекурсивний алгоритм буде працювати повільніше і за рахунок вдвічі більшої кількості рекурсивних викликів порівняно з кількістю ітерацій, і за рахунок роботи зі стеком викликів. Також при кожному рекурсивному викликові у стекові буде виділятися пам'ять, що збільшує ємнісну складність рекурсивного алгоритму.

Приклад 8.2.2. ***Перестановка двох елементів*** масиву здійснюється так само, як і переприсвоєння значень двох змінних:

```
  ...
  int a[N];    // масив елементів
  int i, j;     // індекси елементів
  int r;     // робоча змінна
  ...
  r=a[i];
  a[i]=a[j];
  a[j]=r;
...
```

Приклад 8.2.3. ***Перестановка*** елементів масиву ***у зворотному порядку***.

Для перестановки елементів у зворотному порядку можна скористатися циклом з двома параметрами; алгоритм працює однаково для масивів з парною і непарною кількістю елементів.

Мовою С алгоритм запишеться так:

```
/* Перестановка елементів масиву у зворотному порядку
   при парній і непарній кількості елементів
   з використанням циклу з двома параметрами */
...
  int a[N]={1,2,3,4,5,6,7,8,9,10};
  int r;    // робоча змінна
  int i, j;    // індекси поточних елементів
  /* Перестановка елементів масиву у зворотному порядку */
  for (i=0, j=N-1; i<j; i++, j--) {
    r=a[i];
    a[i]=a[j];
    a[j]=r;
  }
...
```

```
Початковий масив:
1  2  3  4  5  6  7  8  9  10
Результуючий масив:
10  9  8  7  6  5  4  3  2  1
```

У наведеному алгоритмі цикл for має два параметри i і j для індексації елементів масиву з протилежних кінців. Параметр i в циклі збільшується, а j — зменшується. Ітерації припиняються, коли i стає рівним чи більшим від j. Це забезпечує перестановку всіх елементів незалежно від того, кількість елементів парна чи непарна.

Якщо мова програмування не надає можливості організувати цикл з двома параметрами, то алгоритм перестановки елементів у зворотному порядку буде таким:

```
/* Перестановка елементів масиву у зворотному порядку
   при парній і непарній кількості елементів
   з використанням звичайного циклу з параметром */
...
  for(i=0; i<N/2; i++) {
```

```
    r=a[i];
    a[i]=a[N-i-1];
    a[N-i-1]=r;
  }
...
```

В алгоритмі індекс елемента, який розміщується в другій половині масиву симетрично відносно середини масиву, обчислюється за формулою $N-i-1$. Цей алгоритм, як і алгоритм з двома параметрами, працює для масивів з парною і з непарною кількістю елементів. Перестановки виконуються тільки до середини масиву — інакше все повернеться на свої місця.

Приклад 8.2.4. ***Циклічні зсуви*** елементів масиву.

а) Зсув ***вправо на один елемент*** — останній елемент стає першим. Наприклад: 1 2 3 4 5 —> 5 1 2 3 4. Для здійснення такого зсуву треба останній елемент масиву запам'ятати в робочу змінну і всі елементи масиву, ідучи з кінця масиву до початку, змістити на одну позицію вправо. Після цього запам'ятований останній елемент поставити на місце першого.

Реалізація алгоритму мовою С:

```
/* Циклічний зсув елементів масиву вправо на
   один елемент: останній елемент стає першим */
...
  int a[N]={1,2,3,4,5,6,7,8,9,10};
  int i;    // індекс поточного елемента
  int r;    // робоча змінна
...
  /* Виконання зсуву */
  r=a[N-1];
  for (i=N-1; i>0; i--)
    a[i]=a[i-1];
  a[0]=r;
...
```

```
Початковий масив
1  2  3  4  5  6  7  8  9  10
Масив після зсуву вправо
10  1  2  3  4  5  6  7  8  9
```

б) Зсув ***вліво на один елемент*** — перший елемент стає останнім. Наприклад: 1 2 3 4 5 —> 2 3 4 5 1 . Для здійснення такого зсуву треба перший еле-

мент масиву запам'ятати в робочу змінну і всі елементи масиву, ідучи з початку масиву до кінця, змістити на одну позицію вліво.

Реалізація алгоритму мовою С:

```
/* Циклічний зсув елементів масиву вліво на один елемент:
   перший елемент стає останнім */
...
  int a[N]={1,3,4,1,1,4,1,1,7,5};
  int i;   // індекс поточного елемента
  int r;   // робоча змінна
...
  /* Виконання зсуву */
  r=a[0];
  for (i=0; i<N-1; i++)
    a[i]=a[i+1];
  a[N-1]=r;
 ...
```

```
Початковий масив
1  2  3  4  5  6  7  8  9  10
Масив після зсуву вліво
2  3  4  5  6  7  8  9  10  1
```

в) Зсув ***вправо/вліво на кілька елементів*** — останні/перші елементи стають першими/останніми. Наприклад: 1 2 3 4 5 6 7 8 9 10 —> 8 9 10 1 2 3 4 5 6 7 / 1 2 3 4 5 6 7 8 9 10 —> 4 5 6 7 8 9 10 1 2 3. Реалізувати такий зсув можна або, виконавши кілька разів зсув на один елемент, або зберігаючи ці елементи в іншому (робочому) масиві, а решту елементів переміщати зразу на кілька позицій.

Реалізація алгоритму кількаразового зсуву на один елемент:

```
/* Циклічний зсув елементів масиву вправо/вліво на кілька
   елементів: останні/перші елементи стають на перші/останні
   місця, зберігаючи порядок. Зсув на 1 елемент виконується
   K разів */
...
#define N 10    // кількість елементів масиву
#define K 3    // кількість елементів зсуву
  int a[N]={1,2,3,4,5,6,7,8,9,10};
  int i,   // індекс поточного елемента
```

```
      j,   // номер зсуву
      r;   // робоча змінна
...
  /* Виконання зсуву на 1 елемент K разів */
  for (j=1; j<=K; j++) {
```

```
    /* Зсув вправо */
    r=a[N-1];
    for (i=N-1; i>0; i--)
      a[i]=a[i-1];
    a[0]=r;
  }
...
```

```
    /* Зсув вліво */
    r=a[0];
    for (i=0; i<N-1; i++)
      a[i]=a[i+1];
    a[N-1]=r;
```

```
Початковий масив
1  2  3  4  5  6  7  8  9  10
Масив після 3 зсувів вправо
8  9  10  1  2  3  4  5  6  7
```

```
Початковий масив
1  2  3  4  5  6  7  8  9  10
Масив після 3 зсувів вліво
4  5  6  7  8  9  10  1  2  3
```

Розглянутий алгоритм зсуву на *k* елементів за часом буде виконуватися в *k* разів довше, ніж зсув на один елемент.

Щоб легше було побудувати алгоритми зсуву на *k* елементів вправо чи вліво з використанням допоміжного масиву, треба записати початковий масив і результуючий (наприклад, візьмемо *k* =3):

зсув вправо

1 2 3 4 5 6 7 **8 9 10**

8 9 10 1 2 3 4 5 6 7

зсув вліво

1 2 3 4 5 6 7 8 9 10

4 5 6 7 8 9 10 **1 2 3**

Дивлячись на ці масиви, можна легко розібратися з індексами і записати алгоритми:

```
// запам'ятовування останніх елементів
for (i=0; i<K; i++)
  r[i]=a[N-K+i];
// зсув на K елементів вправо
for (i=N-1; i>0; i--)
  a[i]=a[i-K];
// вставка запам'ятованих елементів
for (i=0; i<K; i++)
  a[i]=r[i];
```

```
// запам'ятовування перших елементів
for (i=0; i<K; i++)
  r[i]=a[i];
// зсув на K елементів вліво
for (i=0; i<N-K; i++)
  a[i]=a[i+K];
// вставка запам'ятованих елементів
for (i=0; i<K; i++)
  a[N-K+i]=r[i];
```

Повна програма для виконання циклічного зсуву на кілька елементів, наприклад вправо, буде такою:

```
/* Циклічний зсув елементів масиву вправо на кілька
   елементів: останні стають на перші місця,
```

```
   зберігаючи порядок. Зсув на K елемент виконується
   з використанням допоміжного масиву розміру K */
...
#define N 10   // кількість елементів масиву
#define K 3   // кількість елементів зсуву
  int a[N]={1,2,3,4,5,6,7,8,9,10},
      r[K];   // допоміжний масив
  int i;   // індекс поточного елемента
...
  /* Виконання зсуву */
  // запам'ятовування останніх елементів
  for (i=0; i<K; i++)
    r[i]=a[N-K+i];
  // зсув на K елементів вправо
  for (i=N-1; i>0; i--)
    a[i]=a[i-K];
  // вставка запам'ятованих елементів
  for (i=0; i<K; i++)
    a[i]=r[i];
...
```

Алгоритм циклічного зсуву на кілька елементів з використанням масиву для запам'ятовування елементів, які переміщаються на початок чи в кінець масиву, буде працювати не набагато довше, ніж циклічний зсув на один елемент і значно швидше, ніж алгоритм з повторним виконанням зсувів. При цьому в найгіршому випадку потрібно буде використати додаткову пам'ять для зберігання не більш ніж $(n-1)$ елемента.

Приклад 8.2.5. а) Елементи, ***відмінні від 0, зібрати на початку масиву***, зберігши їхній порядок, а нулі в кінці масиву. Для розв'язання цієї задачі нулі не треба переставляти в кінець масиву — треба тільки переставити на початок масиву ненульові елементи, а потім кінець масиву заповнити нулями.

Програмна реалізація алгоритму мовою С:

```
/* Елементи, відмінні від 0, зібрати на початку масиву,
   зберігши їхній порядок, а 0 - в кінці */
#define N 10
...
  int a[N]={0,0,1,8,0,1,3,0,1,9};
  int i;    // індекс поточного елемента
  int k;    // новий індекс елемента, не рівного 0
...
  // Ненульові елементи переміщаються на початок масиву
  k=0;
  for (i=0; i<N; i++)
    if (a[i]) {
      a[k]=a[i]; k++;
    }
  // У кінець масиву записуються нулі
  for (i=k; i<N; i++) a[i]=0;
...
```

```
Початковий масив:
0  0  1  8  0  1  3  0  1  9
Результуючий масив:
1  8  1  3  1  9  0  0  0  0
```

```
Початковий масив:
3  5  1  8  7  1  3  9  1  9
Результуючий масив:
3  5  1  8  7  1  3  9  1  9
```

б) Елементи, ***відмінні від 0, зібрати в кінці масиву***, зберігши їхній порядок, а нулі на початку масиву. Для розв'язання цієї задачі треба, ідучи від кінця масиву до початку, перествити на кінець масиву ненульові елементи, а потім початок масиву (ідучи від початку) заповнити нулями.

```
/* Елементи, відмінні від 0, зібрати в кінці масиву,
зберігши їхній порядок, а 0 – на початку */
#define N 10
...
  int a[N]={0,0,1,8,0,1,3,0,1,9};
  int i;    // індекс поточного елемента
  int k;    // новий індекс елемента, не рівного 0
...
  // Ненульові елементи переміщаються в кінець масиву
  k=N;
  for (i=N-1; i>=0; i--)
```

```
    if (a[i]) {
      k--; a[k]=a[i];
    }
  // Початок масиву заповнюється нулями
  for (i=0; i<k; i++) a[i]=0;
...
```

```
Початковий масив:
0  0  1  8  0  1  3  0  1  9
Результуючий масив:
0  0  0  0  1  8  1  3  1  9
```

Приклад 8.2.6. Усі елементи ***з певною ознакою***, наприклад, кратні 5, записати ***на початок масиву***, зберігши їхній порядок, а ***всі решта*** перемістити ***в кінець***, теж зберігши їхній порядок.

Алгоритм розв'язання цієї задачі буде складнішим, ніж двох попередніх. Тут треба використовувати зсув елементів. Як тільки буде знайдено елемент, кратний 5, його значення треба запам'ятати в робочу змінну. Некратні елементи, які розміщуються перед ним, зсунути на одну позицію вправо до його позиції (ідучи вліво від його позиції до позиції, в яку його треба помістити). Запам'ятоване значення елемента помістити у звільнену позицію.

Реалізація алгоритму мовою С:

```
/* Усі елементи, кратні 5, перемістити на початок масиву,
   зберігши їхній порядок, а всі решта - в кінець, теж
   зберігши їхній порядок */
#define N 10
...
  int a[N]={100,21,40,5,1,9,12,18,7,45};
  int i;   // індекс поточного елемента
  int k;   // індекс останнього елемента, кратного 5
  int j;   // індекс елемента, який зсувається
  int r;   // робоча змінна
...
  /* Переміщення елементів: кратних вліво, некратних вправо */
  k=-1;
  for (i=0; i<N; i++)
    if (a[i]%5==0) {
      k++;
```

```
Початковий масив
100  21  40  5  1  9  12  18  7  45
Результуючий масив
100  40  5  45  21  1  9  12  18  7
Таких елементів є 4
```

```
      r=a[i];
      // Зсув некратних елементів вправо
      for (j=i-1; j>=k; j--)
        a[j+1]=a[j];
      a[k]=r;
    }
  printf("\nРезультуючий масив\n");
...
```

```
Початковий масив
101  21  42  8  1  9  12  18  7  49
Результуючий масив
101  21  42  8  1  9  12  18  7  49
Таких елементів є 0
```

Поданий алгоритм можна використати не тільки для переміщення елементів, кратних 5, а й для елементів з будь-якою іншою властивістю.

8.3. Пошук елемента в масиві

Пошук елемента в масиві доводиться виконувати досить часто. Найпростіший пошук — за допомогою послідовного перегляду всіх елементів масиву; найшвидший — бінарний пошук у впорядкованому масиві.

Часова складність алгоритмів пошуку прямо пропорційна кількості порівнянь, які виконуються.

Якщо масив, у якому виконується пошук не відсортований, то треба переглядати елементи масиву підряд і порівнювати з заданим значенням (ключем). Такий пошук називають лінійним. У найгіршому випадку з ключем треба порівняти всі елементи масиву. Тому максимальна кількість дій і найбільший час виконання алгоритму прямо (лінійно) пропорційні кількості елементів масиву, тобто описуються лінійною функцією від кількості елементів масиву, а отже, асимптотична складність лінійного пошуку дорівнює $O(n)$.

Якщо масив відсортований за зростанням, то можна застосувати бінарний (ще кажуть: двійковий, дихотомічний, логарифмічний) пошук, ділячи масив приблизно навпіл — при цьому виконується не більше ніж $\log_2 n$ порівнянь (див. приклад 8.6.1). Тому асимптотична складність алгоритму бінарного пошуку дорівнює $O(\log_2 n)$.

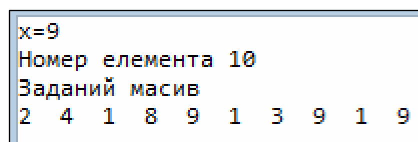
Нехай задано якесь значення *x*, яке треба знайти в масиві. Спочатку розглянемо алгоритми лінійного пошуку.

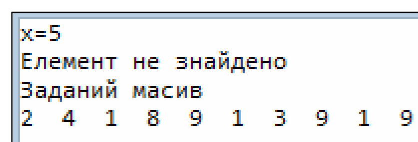
Приклад 8.3.1. ***Пошук*** у масиві ***одного елемента***, рівного заданому значенню.

а) Пошук виконується за допомогою класичного циклу з параметром. При цьому буде знайдено індекс останнього елемента, рівного заданому.

Реалізація алгоритму мовою С:

```
/* Знаходить індекс останнього елемента, рівного x */
#define N 10
...
  int a[N]={2,4,1,8,9,1,3,9,1,9};
  int x;    // задане значення
  int k,    // індекс елемента, рівного x
      i;    // індекс поточного елемента
  printf("x="); scanf("%d", &x);
  k=-1;
  for (i=0; i<N; i++)
    if (a[i]==x) k=i;
  if (k>=0) printf("Номер елемента %d",k+1);
  else printf("Елемент не знайдено");
  printf("\nЗаданий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```





У цьому прикладі, якщо елемент знайдено, то пошук все рівно буде продовжуватися. Ефективніше буде, якщо застосувати цикл з передумовою чи післяумовою, чи цикл з параметром з додатковою умовою.

б) Використання циклу з передумовою. Буде знайдено індекс першого елемента, рівного заданому. Реалізація алгоритму мовою С:

```
/* Знаходить індекс першого елемента, рівного x */
#define N 10
...
```

```
  int a[N]={2,4,1,8,9,1,3,9,1,9};
  int x;   // задане значення
  int k;   // індекс елемента, рівного x
  int i;   // індекс поточного елемента
  printf("x="); scanf("%d", &x);
  k=-1; i=0;
  while (i<N && k<0) {   // for (i=0; i<N && k<0; i++)
    if (a[i]==x) k=i;
    i++;
  }
  if (k>=0) printf("Номер елемента %d",k+1);
  else printf("Елемент не знайдено");
  printf("\nЗаданий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```

```
x=9
Номер елемента 5
Заданий масив
2  4  1  8  9  1  3  9  1  9
```

Відповідно до цього алгоритму, якщо елемент знайдено, то пошук припиняється.

в) Використання циклу з післямовою. Буде знайдено індекс першого елемента, рівного заданому. Реалізація алгоритму мовою С:

```
/* Знаходить індекс першого елемента, рівного x */
#define N 10
...
  int a[N]={2,4,1,8,9,1,3,9,1,9};
  int x;   // задане значення
  int k;   // індекс елемента, рівного x
  int i;   // індекс поточного елемента
  printf("x="); scanf("%d", &x);
  k=-1; i=0;
  do {   // for (i=0; i<N && k<0; i++)
    if (a[i]==x) k=i;
    i++;
  } while (i<N && k<0);
```

```
  if (k>=0) printf("Номер елемента %d",k+1);
  else printf("Елемент не знайдено");
  printf("\nЗаданий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```

Відповідно до цього алгоритму, якщо елемент знайдено, то пошук припиняється.

г) При пошукові елемента до першого входження для спрощеня умови закінчення циклу можна скористатися фіктивним елементом a[N]=x (номер елемента N+1, але індекс N). Цей метод називають алгоритмом ***пошуку з бар'єром***.

Реалізація алгоритму мовою C:

```
/* Використання фіктивного елемента для визначення
   індексу першого елемента в масиві, рівного x */
#define N 10
...
  int a[N+1]={2,4,1,8,9,1,3,9,1,9};
  int x;   // задане значення
  int i;   // індекс поточного елемента
  printf("x="); scanf("%d", &x);
  a[N]=x;   // фіктивному елементу надаємо значення x
  i=-1;
  do
    i++;
  while (a[i]!=x);
  if (i==N) printf("Елемент не знайдено");
  else printf("Номер елемента %d",i+1);
  printf("\nЗаданий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```

```
x=1
Номер елемента 3
Заданий масив
2  4  1  8  9  1  3  9  1  9
```

В алгоритмі з фіктивним елементом нема перевірки на кінець масиву, а є лише перевірка умови пошуку елемента. Вихід із циклу може відбутися або на знайденому елементі, або на фіктивному елементі. Після виходу із циклу

перевіряється, який елемент знайдено: потрібний чи фіктивний. Обчислювальна складність пошуку з фіктивним елементом менша, ніж у розглянутого вище лінійного пошуку, але також є $O(n)$.

Приклад 8.3.2. ***Пошук усіх елементів*** масиву, рівних заданому значенню *x*.

Тут треба переглянути всі елементи масиву. Отже, треба використовувати класичний цикл з параметром (звичайно, його роботу можна промоделювати і циклами з перед- і післяумовою). Для збереження знайдених індексів елементів треба створити новий масив (ще можна в цьому масиві в елементі з індексом 0 зберегти кількість таких елементів).

Реалізація алгоритму мовою С:

```
/* Знаходить індекси всіх елементів, рівних x */
...
#define N 10
  int a[N]={2,4,1,8,9,1,3,9,1,9};
  int x;   // задане значення
  int k[N+1];   // індекси елементів, рівних x; k[0] - кількість
  int kx;   // кількість знайдених елементів
  int i;   // індекс поточного елемента
  printf("x="); scanf("%d", &x);
  kx=0;
  for (i=0; i<N; i++)
    if (a[i]==x) {kx++; k[kx]=i;}
  k[0]=kx;
  if (kx) {
    printf("Кількість знайдених елементів: %d.\nЇхні номери: ", kx);
    for (i=1; i<=kx; i++) printf("%d  ", k[i]+1);
  }
  else
    printf("Елементів не знайдено");
  printf("\nЗаданий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```

```
x=9
Кількість знайдених елементів: 3.
Їхні номери: 5  8  10
Заданий масив
2  4  1  8  9  1  3  9  1  9
```

Приклад 8.3.3. Якщо масив відсортовано за зростанням, то найчастіше застосовують пошук з діленням масиву навпіл (стратегія розділяй і володарюй) — ***бінарний пошук***. При кожному пошукові інтервал між початковим і кінцевим індексами ділиться на дві приблизно рівні частини. При цьому порівнюється з заданим значенням x тільки елемент, розміщений у центрі інтервалу індексів. Асимптотична складність алгоритму бінарного пошуку дорівнює $O(\log_2 n)$ (див. далі приклад 8.6.1).

а) Алгоритм бінарного пошуку з використанням циклу програмно мовою С можна подати так:

```
/* Використання бінарного пошуку для визначення у
   відсортованому масиві номера елемента, рівного x.
   Ітераційний алгоритм */
#define N 9
...
  int a[N]={1,5,7,9,12,18,21,40,45};
  int x;    // задане значення
  int i,j,k;    // індекси
  printf("x=");   scanf("%d",&x);
  i=0;    // початковий індекс
  j=N-1;    // кінцевий індекс
  do {
    k=(i+j)/2;    // середина інтервалу індексів
    if (x>a[k]) i=k+1;
    else j=k-1;
  } while (a[k]!=x && i<=j);
  if(a[k]==x) printf("Номер елемента: %d",k+1);
  else printf("Елемент не знайдено");
  printf("\nЗаданий масив:\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
...
```

```
x=7
Номер елемента: 3
Заданий масив:
1  5  7  9  12  18  21  40  45
```

```
x=4
Елемент не знайдено
Заданий масив:
1  5  7  9  12  18  21  40  45
```

Розглянемо приклад покрокового виконання цього алгоритму.

1. Нехай масив має 9 елементів:

елемент	1	5	7	9	12	18	21	40	45
індекс	0	1	2	3	4	5	6	7	8
номер	1	2	3	4	5	6	7	8	9

і x=7 (у масиві є елемент з таким значенням).

Початкові значення: i=0
j=8

І прохід циклу: k=(0+8)/2=4
(x=7)<(a[4]=12),
отже, j=4-1=3, залишається i=0
оскільки (a[4]=12)≠(x=7) і (i=0)<=(j=3), то виконується

ІІ прохід циклу: k=(0+3)/2=1
(x=7)>(a[1]=5),
отже, i=1+1=2, залишається j=3
оскільки (a[1]=5)≠(x=7) і (i=2)<=(j=3), то виконується

ІІІ прохід циклу: k=(2+3)/2=2
(x=7)!>(a[2]=7),
отже, j=2-1=1, і залишається=2
оскільки (a[2]=7)=(x=7), то цикл закінчує роботу.

Оскільки при k=2 (a[2]=7)=(x=7), то буде виведено номер елемента 2+1=3.

2. Нехай масив має ті самі 9 елементів і x=4 (у масиві нема елемента з таким значенням)

Початкові значення: i=0
j=8

І прохід циклу: k=(0+8)/2=4
(x=4)<(a[4]=12),
отже, j=4-1=3, залишається i=0
оскільки (a[4]=12)≠(x=4) і (i=0)<=(j=3), то виконується

ІІ прохід циклу: k=(0+3)/2=1
(x=4)<(a[1]=5),
отже, j=k-1=0, залишається i=0
оскільки (a[1]=5)≠(x=4) і (i=0)<=(j=0), то виконується

ІІІ прохід циклу: k=(0+0)/2=0
(x=4)>(a[0]=1),
отже, j=0-1=-1, залишається i=0
оскільки (a[2]=7)!=(x=4) і (i=0)>(j=-1), то цикл закінчує роботу.

Оскільки при k=0 (a[0]=1)≠(x=4), то буде виведено повідомлення, що елемент не знайдено.

б) Алгоритм бінарного пошуку можна запрограмувати, використовуючи рекурсію:

```
/* Використання бінарного пошуку для визначення у
   відсортованому масиві номера елемента, рівного x.
   Рекурсивний алгоритм. */
#define N 10
...
  int a[N]={1,5,7,9,12,18,21,40,45,100};
  int x;   // задане значення
```

```
  int k;   // індекс поточного елемента
  printf("x=");
  scanf("%d",&x);
  k=binary(a, x, 0, N-1);
...
/* Бінарний пошук */
int binary(int *a, int x, int i, int j) {
  int k;   // середина інтервалу індексів
  if (i>j) k=-1;
  else {
    k=(i+j)/2;
    if (x<a[k]) k=binary(a, x, i, k-1);
    else
      if (x>a[k]) k=binary(a, x, k+1, j);
  }
  return k;
}
```

При використанні як циклічної, так і рекурсивної реалізації алгоритму бінарного пошуку обчислювальна складність дорівнює $O(\log_2 n)$. Рекурсивний алгоритм потребує додаткового часу і додаткової пам'яті.

8.4. Сортування масивів

Сортування передбачає перегрупування заданої множини об'єктів у певному порядку. Сортування виконують з метою полегшення пошуку даних, їхнього групування, швидкого перегляду і покращення сприйняття, максимально ефективного використання. Програмно сортувати можна числові масиви, рядки символів, масиви записів (структур), списки тощо.

Існує досить багато різних алгоритмів сортування. Є прості алгоритми і є складні. Одні з них дуже повільні навіть на малих обсягах даних, інші з малими масивами працюють досить швидко, але стають повільними при роботі з великими масивами, ще інші досить складні для реалізації, але можуть досить швидко обробляти великі масиви. Маючи приблизні характеристики

вхідних даних, можна підібрати алгоритм сортування, який працює для певного випадку найоптимальніше.

Сортування відбувається за *ключем* — послідовністю символів, яка використовується для порівняння. Ключем може бути число, рядок, довжина рядка, певна координата вектора, відстань від початку координат до точки тощо. Сортувати можна спочатку за одним ключем, а потім за іншим і т. д.

Алгоритми сортування не залежать від того, чи впорядковуються числа, рядки символів або записи (структури) за певним ключем. Але при реалізації цих алгоритмів на практиці треба враховувати природу даних. Наприклад, при сортуванні записів, які містять великі обсяги інформації, щоб при перестановках не переписувати кожен раз цю інформацію, впорядковувати можна масиви вказівників на записи.

Поведінку алгоритму сортування називають природною, якщо час його виконання є мінімальним для відсортованого масиву, час збільшується при зростанні невпорядкованості масиву і час максимальний, коли елементи розташовані у зворотному порядку.

Алгоритм сортування є стабільним, якщо він не змінює взаємного розташування елементів з однаковими ключами.

Алгоритм сортування є адаптивним, якщо він орієнтований на швидше виконання, коли вхідний масив відповідає заздалегідь визначеним критеріям.

Для оцінки часової асимптотичної складності алгоритмів сортування у більшості випадків використовують два показники: кількість порівнянь і кількість перестановок (переприсвоєння виконується значно довше, ніж порівняння).

При сортуванні масивів також, як правило, ставиться вимога мінімального використання додаткової пам'яті, з чого випливає, що сортування має відбуватися в тому самому масиві, як кажуть, «на місці» (у такому випадку ємнісна складність дорівнює $O(1)$). Хоч, наприклад, метод сортування злиттям має дві реалізації — з використанням додаткової пам'яті (старіший алгоритм) і без.

Розрізняють прямі методи сортування і поліпшені.

Прямі методи (наприклад, метод бульбашки, простого вибору з пошуком мінімуму чи максимуму, простих вставок) є найпростішими, їхні програми короткі і їх легко зрозуміти, але їхнім недоліком є велика кількість порівнянь і перестановок. Прямі методи сортування на практиці використовують досить рідко, оскільки вони мають відносно низьку швидкість. Проте вони добре розкривають суть заснованих на них поліпшених методів. Крім того, в деяких випадках (при невеликому обсягові або певному початковому розміщенні елементів масиву) деякі прямі методи можуть навіть перевершити за швидкістю поліпшені методи. Але на великих обсягах елементів прямі методи неефективні, оскільки їхній час виконання $O(n^2)$.

Серед прямих методів виділяють сортування обміном (метод бульбашки, сортування гнома), обміном з вибором (з пошуком мінімуму чи максимуму), вставками (метод простих вставок).

Поліпшені методи сортування (шейкерне сортування, швидке сортування, сортування Шелла, пірамідальне сортування, сортування бінарним деревом; сортування злиттям та інші) ґрунтуються на тих самих принципах, що й прямі, але використовують деякі оригінальні ідеї для прискорення роботи. Вони потребують виконання невеликої кількості операцій, але ці операції, як правило, складніші. Найшвидші серед цих алгоритмів (швидке сортування у середньому, пірамідальне сортування, сортування бінарним деревом, сортування злиттям) виконуються за час $O(n\log n)$. Проте за наявності певної додаткової інформації деякі алгоритми (сортування підрахунком — невелика кількість різних елементів, сортування комірками — вхідні дані розподілені рівномірно) можуть мати часову складність $O(n)$.

Жоден метод сортування не досягає результату за один послідовний перегляд масиву. Як правило, при сортуванні масивів для цього застосовують цикл у циклі, хоч, наприклад, метод гнома реалізується за допомогою одного циклу, але з рухом вперед і назад, безглузде сортування теж використовує один цикл, але після кожної перестановки елементів повертає керування на початок масиву.

В описаних нижче методах сортування вважається, що вхідна (не відсортована) і вихідна (відсортована) множини розміщуються в тій самій ділянці пам'яті (одному масиві). Перед початком сортування вхідна множина займає всю ділянку пам'яті, а вихідна — порожня. У процесі сортування вихідна множина розширюється, а вхідна — звужується.

Розглянемо деякі методи сортування з програмною реалізацією, а деякі — без (їх треба запрограмувати при виконанні практчного завдання № 14).

Метод бульбашки є методом обмінного сортування. Спочатку вихідна множина порожня, вхідною множиною є весь масив. Попарно порівнюються сусідні елементи множини і, якщо їхній порядок не відповідає критерію впорядкованості, то елементи міняються місцями. У результаті одного такого проходу по масиву при сортуванні за зростанням (спаданням) елемент з найбільшим (найменшим) значенням стане останнім у масиві — отже, вихідна множина розміститься в кінці масиву і міститиме один елемент, а вхідна — перші (n–1) елемент. При наступному проході на своє місце стане наступний елемент і т. д. У найгіршому випадку за (n–1) прохід множина буде впорядкована.

Приклад 8.4.1. Програмна реалізація методу бульбашки така:

```
/* Обмінне сортування: метод бульбашки.
   Елементи впорядковуються за зростанням */
...
#define N 10
int po=0, pe=0;   //кількість порівнянь і перестановок
...
/* Обмінне сортування: метод бульбашки - порівняння і
   перестановка кожних двох сусідніх елементів; елементи
   впорядковуються за зростанням; за один прохід циклу
   в кінець масиву випливає найбільший елемент */
void sort(int a[N]) {
  int i;   // індекс поточного проходу
  int j;   // індекс поточного елемента
```

```
  int r;   // робоча змінна
  for (i=0; i<N-1; i++)
    for (j=0; j<N-i-1; j++) {
      po++;   // кількість порівнянь
      if (a[j]>a[j+1]) {
        pe++;   // кількість перестановок
        r=a[j]; a[j]=a[j+1]; a[j+1]=r;
      }
    }
}
```

У програмі додатково використано дві глобальні змінні *po* і *pe* відповідно для обчислення кількостей порівнянь і перестановок елементів. Програму виконано на трьох наборах даних — невпорядкованому, впорядкованому за спаданням і впорядкованому за зростанням:

```
100  21  40  5  1  9  12  18  7  45
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 25
```

```
100  45  40  21  18  12  9  7  5  1
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 45
```

```
1  5  7  9  12  18  21  40  45  100
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 0
```

Як видно зі скриншотів, у всіх трьох випадках при сортуванні за зростанням кількість порівнянь однакова. Кількість перестановок найбільша, якщо елементи впорядковані за спаданням; жодної — якщо за зростанням; проміжна — при випадковому початковому розміщенні елементів.

Якщо треба сортувати не всі елементи масиву, а тільки частину масиву, починаючи з індексу *k*, то у функції сортування замість фрагмента:

```
  for (i=0; i<N-1; i++)
    for (j=0; j<N-i-1; j++)
      if (a[j]>a[j+1]) {
        r=a[j]; a[j]=a[j+1]; a[j+1]=r;
      }
```

треба використати такий фрагмент:

```
// сортування кінця масиву за зростанням від індексу k до n-1
for (i=k; i<N-1; i++)
  for (j=k; j<N+k-i-1; j++)
    if (a[j]>a[j+1]) {
      r=a[j]; a[j]=a[j+1]; a[j+1]=r;
    }
```

У методі бульбашки зовнішній цикл виконується $(n-1)$ раз, а внутрішній — у середньому $n/2$ разів; у найгіршому випадку кількість порівнянь рівна $n(n-1)/2$, кількість перестановок — теж $n(n-1)/2$; у найкращому випадку кількість порівнянь та сама, а кількість перестановок — жодної. Таким чином, кількість порівнянь елементів не залежить від початкової впорядкованості масиву. Часова складність алгоритму становить $O(n^2)$. Метод бульбашки вважають найповільнішим (якщо не враховувати методу безглуздого сортування).

Завдяки тому, що в методі сортування бульбашкою завжди порівнюються і переставляються тільки сусідні елементи, то таке сортування зручне для обробки зв'язних списків — перестановка в зв'язних списках виходить більш економною.

Також метод бульбашки при незначній модифікації — використанні індикатора перестановок і запам'ятовуванні останнього місця перестановки стає чутливим до початкової впорядкованості вхідної множини.

Метод бульбашки з індикатором перестановки. Спочатку вихідна множина порожня, вхідною множиною є весь масив, індикатор перестановки набуває значення false. Попарно порівнюються сусідні елементи множини і, якщо їхній порядок не відповідає критерію впорядкованості, то елементи міняються місцями і індикатор набуває значення true. У результаті одного такого проходу по масиву при сортуванні за зростанням (спаданням) елемент з найбільшим (найменшим) значенням стане останнім у масиві — отже, вихідна множина розміститься в кінці масиву і міститиме один елемент, а вхідна —

перші $(n-1)$ елемент. Якщо значення індикатора після проходу залишилося рівним false, то це означає, що всі елементи відсортовано і сортування закінчено. Якщо ж значення індикатора стало true, то індикаторові треба присвоїти значення false і виконати наступний прохід. При наступному проході на своє місце стане наступний елемент і т. д. За не більш ніж $(n-1)$ прохід множину буде впорядковано. Якщо вхідна множина впорядкована, то буде зроблено всього один прохід.

Метод бульбашки з індикатором перестановки і запам'ятовуванням місця останньої перестановки (алгоритм враховує, що на своє місце за один прохід може стати не лише один, а й більше елементів множини). Спочатку вихідна множина порожня, вхідною множиною є весь масив, позиція останньої перестановки має значення n, індикатор перестановки набуває значення false. Попарно порівнюються сусідні елементи множини і, якщо їхній порядок не відповідає критерію впорядкованості, то позиція перестановки набуває значення номера першого з пари елементів, які переставляються, а індикатор перестановки набуває значення true. У результаті одного такого проходу по масиву при сортуванні за зростанням (спаданням) кілька елементів з найбільшими (найменшими) значеннями стануть останніми в масиві — отже, вихідна множина розміститься в кінці масиву, а вхідна — на початку. Якщо позиція останньої перестановки дорівнює 1 або індикатор має значення false, то це означає, що всі елементи відсортовано і сортування закінчено. Якщо сортування не закінчено, то наступний прохід виконується до позиції останньої перестановки (тобто на кожному проході верхня межа вхідної множини встановлюється не на 1 меншою, а буде збігатися з позицією останньої перестановки, яка може бути значно меншою) і при наступному проході на своє місце стануть наступні елементи і т. д. За не більш ніж $(n-1)$ прохід множину буде впорядковано. Якщо вхідна множина вже впорядкована, то буде виконано тільки один прохід.

Швидкість роботи методу бульбашки залежить від початкового розміщення елементів у масиві. Так, при сортуванні, наприклад, за зростанням ве-

ликі елементи з початку масиву досить швидко переміщуються в його кінець; проте малі елементи з кінця масиву переміщуються на його початок дуже повільно. Ця особливість методу бульбашки одержала назву «проблеми кроликів і черепах». Для підвищення швидкості сортування методом бульбашки розроблено метод сортування перемішуванням — шейкер-сортування.

Метод шейкер-сортування є модифікацією методу бульбашки. Спочатку вихідна множина порожня, вхідною множиною є весь масив. При першому (непарному) проході масив проглядається від початку до кінця: попарно порівнюються сусідні елементи множини і, якщо їхній порядок не відповідає критерію впорядкованості, то елементи міняються місцями (як в методі бульбашки). У результаті такого проходу при сортуванні за зростанням (спаданням) елемент з найбільшим (найменшим) значенням стане останнім у масиві — отже, вихідна множина розміститься в кінці масиву і міститиме один елемент, а вхідна — на початку і міститиме перші $(n-1)$ елемент. При наступному (парному) проході вхідна множина переглядається від кінця до початку і при сортуванні за зростанням (спаданням) елемент з найменшим (найбільшим) значенням стане першим у масиві. Таким чином вихідна множина формується в кінці і на початку масиву. Проходи від початку до кінця і від кінця до початку масиву чергуються, доки не буде досягнуто середини масиву.

Алгоритм шейкер-сортування досить ефективний для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася незначним змінам. Впорядкованість в послідовності з одиночною зміною буде відновлено всього за два проходи.

Шейкер-сортування з індикатором перестановки. Спочатку вихідна множина порожня, вхідною множиною є весь масив, індикатор перестановки набуває значення false. При першому (непарному) проході масив проглядається від початку до кінця: попарно порівнюються сусідні елементи множини і, якщо їхній порядок не відповідає критерію впорядкованості, то елементи міняються місцями і індикатор набуває значення true. У результаті такого

проходу при сортуванні за зростанням (спаданням) елемент з найбільшим (найменшим) значенням стане останнім у масиві — отже, вихідна множина розміститься в кінці масиву і міститиме один елемент, а вхідна — перші $(n-1)$ елемент. Якщо значення індикатора після проходу залишилося рівним false, то це означає, що всі елементи відсортовано і сортування закінчено. Якщо ж значення індикатора стало true, то індикатору присвоюється значення false і виконується наступний прохід. При наступному (парному) проході вхідна множина переглядається від кінця до початку; якщо є перестановки елементів, то індикатор набуває значення true; при сортуванні за зростанням (спаданням) елемент з найменшим (найбільшим) значенням стане першим у масиві. Таким чином вихідна множина формується в кінці і на початку масиву. Проходи від початку до кінця і від кінця до початку масиву чергуються, доки не буде досягнуто середини масиву чи значення індикатора не залишиться рівним false.

Безглузде сортування (нудне сортування; stupid sort) — метод сортування, схожий на метод бульбашки, але після кожної перестановки двох сусідніх елементів керування повертається на початок масиву. Алгоритм дуже неефективний, хоч записується тільки з одним циклом — після кожної перестановки елементів цикл виконується з самого початку.

Приклад 8.4.2. Програмна реалізація методу безглуздого сортування така:

```
/* Безглузде, нудне сортування */
...
#define N 10
int po=0, pe=0;   //кількість порівнянь і перестановок
...
/* Безглузде сортування за зростанням – порівняння двох сусідніх
   елементів, перестановка і повернення після перестановки на
   початок циклу */
void sort(int a[N]) {
  int i;   // індекс поточного елемента
  int r;   // робоча змінна
```

```
  for (i=0; i<N-1; i++) {
    po++;   // кількість порівнянь
    if (a[i]>a[i+1]) {
      pe++;   // кількість перестановок
      r=a[i]; a[i]=a[i+1]; a[i+1]=r;
      i=-1; // для повернення на початок циклу
    }
  }
}
```

```
100  21  40  5  1  9  12  18  7  45
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 115
Кількість перестановок 25
```

```
100  45  40  21  18  12  9  7  5  1
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 174
Кількість перестановок 45
```

```
1  5  7  9  12  18  21  40  45  100
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 9
Кількість перестановок 0
```

Зі скриншотів видно, що при роботі алгоритму з тими самими даними, що і в методі бульбашки (див. приклад 8.4.1), і впорядкуванні елементів за зростанням кількості перестановок ті самі, але кількості порівнянь значно відрізняються — якщо елементи впорядковані за спаданням чи розміщені випадково, то кількості порівнянь значно більші, якщо розміщені за зростанням, то кількість порівнянь менша.

В алгоритмі безглуздого сортування в найгіршому випадку, коли масив впорядковано за спаданням, виконується $n(n-1)/2$ перестановок, кількість порівнянь можна обчислити за формулою $\frac{1}{2}\sum_{i=2}^{n} i(i-1)+n-1$; у найкращому випадку, коли масив впорядковано за зростанням, виконується $(n-1)$ порівняння і не виконується жодної перестановки. Як видно з поданих у прикладі результатів, кількість порівнянь у цьому алгоритмі залежить від початкової впорядкованості масиву. Часова складність алгоритму $O(n^3)$.

Метод гномового сортування (алгоритм працює за аналогією з гномом, який у саду переставляє горщики, впорядковуючи їх за розміром) є одним з

найпростіших алгоритмів сортування. Алгоритм не потребує вкладених циклів. Курсор встановлюється на початку масиву — перед другим елементом. Якщо два сусідні елементи (попередній і поточний) розміщені в правильному порядку, то курсор пересувається на одну позицію вперед; якщо ж порядок двох сусідніх елементів порушено, то елементи переставляються (при цьому може виникнути нове порушення порядку) і курсор пересувається на одну позицію назад, щоб знову перевірити порядок елементів. Якщо курсор опиняється перед першим елементом, то пересувається вперед, якщо після останнього, то масив впорядковано.

Приклад 8.4.3. Програмна реалізація методу гнома:

```
/* Метод гномового сортування.
   Елементи впорядковуються за зростанням */
...
#define N 10
int po=0, pe=0;   //кількість порівнянь і перестановок
...
/* Обмінне сортування: метод гнома - порівнюються два
   сусідні елементи і, якщо перестановка не виконується,
   то здійснюється перехід до наступного елемента, якщо
   виконується - повернення до попереднього елемента.
   Елементи впорядковуються за зростанням */
void sort(int a[N]) {
  int i;   // індекс елемента масиву
  int r;   // робоча змінна для переприсвоєння
  i=1;    // індекс вказує на початок сортування
  while (i < N) {   // доки не досягнуто кінця масиву
    if (i==0)   // індекс вказує на початок масиву
      i=1;
    if (a[i-1]<=a[i]) {   // порядок не порушено
      po++;   // кількість порівнянь
      i++;   // вперед на один елемент
    }
```

```
    else {    // порядок порушено
      pe++;    // кількість перестановок
      r=a[i-1]; a[i-1]=a[i]; a[i]=r;    // перестановка
      i--;    // повернення на один елемент
    }
  }
}
```

```
100  21  40  5  1  9  12  18  7  45
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 31
Кількість перестановок 25
```

```
100  45  40  21  18  12  9  7  5  1
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 45
```

```
1  5  7  9  12  18  21  40  45  100
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 9
Кількість перестановок 0
```

Як видно зі скриншотів виконання програми, алгоритм гнома і за кількістю порівнянь, і за кількістю перестановок (крім найгіршого випадку, коли елементи впорядковані у зворотному порядку) виконується швидше від методу бульбашки.

Хоч в алгоритмі гнома використовується тільки один цикл, проте за рахунок того, що індекс може не тільки збільшуватися, а й зменшуватися, швидкодія алгоритму гнома — $O(n^2)$. Цей алгоритм працює досить швидко, якщо масив частково відсортований — тоді його швидкодія є $O(n)$.

Алгоритм гнома, як і алгоритм бульбашки, можна оптимізувати, ввівши додаткову змінну для запам’ятовування позиції зворотного переміщення, завдяки чому після впорядкування початку масиву (досягнення першого елемента) впорядкування можна продовжити з запам’ятованої позиції.

Алгоритм обміну поточного елемента з розміщеним далі меншим. Спочатку вихідна множина порожня, вхідною множиною є весь масив. Перший елемент порівнюється з усіма наступними. Якщо наступний елемент менший (при сортуванні за зростанням) від першого, то елементи переставля-

ються місцями. Потім береться елемент далі і знову порівнюється і, якщо треба, переставляється з першим і т. д. Після перегляду масиву до кінця на першому місці буде стояти мінімальний елемент — вихідна множина розміститься на початку масиву і міститиме один елемент, а вхідна — решту $(n-1)$ елемент. Після цього другий елемент порівнюється з усіма наступними і т. д.

Приклад 8.4.4. Програмна реалізація алгоритму сортування обміном поточного елемента з подальшим меншим така:

```
/* Обмін поточного елемента з розміщеним далі меншим.
   Елементи впорядковуються за зростанням */
...
#define N 10
int po=0, pe=0;   //кількість порівнянь і перестановок
...
/* Сортування за зростанням - перестановка
   поточного і меншого подальшого елемента */
void sort(int a[N]) {
  int r;    // робоча змінна
  int i;    // індекс поточного елемента
  int j;    // індекс наступного елемента
  for (i=0; i<N-1; i++)
    for (j=i+1; j<N; j++) {
      po++;    // кількість порівнянь
      if (a[j]<a[i]) {
        pe++;    // кількість перестановок
        r=a[i]; a[i]=a[j]; a[j]=r;
      }
    }
}
```

```
100  21  40  5  1  9  12  18  7  45
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 25
```

```
100  45  40  21  18  12  9  7  5  1
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 45
```

```
1  5  7  9  12  18  21  40  45  100
відсортовані
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 0
```

Зі скриншотів видно, що кількість порівнянь і перестановок така ж сама, як і в методі бульбашки.

При роботі алгоритму виконується $n(n-1)/2$ порівнянь у найгіршому випадку, коли масив впорядковано за спаданням, і стільки ж перестановок; у найкращому випадку, коли масив впорядковано за зростанням, — жодної перестановки. Кількість порівнянь елементів не залежить від початкового порядку. Часова складність алгоритму $O(n^2)$.

Якщо в розглянутому алгоритмі обміну поточного елемента з подальшим меншим зробити невелику модифікацію — замість перестановки знайденого меншого елемента з поточним спочатку знайти найменший, а потім його переставити з поточним, то буде одержано описаний нижче метод вибору.

Обмінне сортування простим ***вибором з пошуком мінімуму*** (***максимуму***). Спочатку вихідна множина порожня, вхідною множиною є весь масив, а поточним елементом — елемент з номером 1. У вхідній множині вибирається мінімальний (максимальний) елемент і міняється з поточним. Після цього вихідна множина, яка розміщується на початку масиву, містить один елемент, вхідна — всі решту елементів, а поточним елементом є елемент з номером 2. Знову у вхідній множині вибирається мінімальний (максимальний) елемент і міняється з поточним і т. д., доки не буде досягнуто кінця масиву.

У методі вибору зовнішній цикл виконується $(n-1)$ раз, а внутрішній — у середньому $n/2$ разів; кількість порівнянь, як і в методі бульбашки — $n(n-1)/2$, кількість перестановок у найгіршому випадку — $(n-1)$, у найкращому — жодної. Зауважимо, що в методі сортування бульбашкою кількість перестановок у найгіршому випадку дорівнює $n(n-1)/2$, що значно більше.

Сортування простими вставками. Спочатку вважається, що перший елемент масиву утворює вихідну впорядковану множину, а вхідною множиною є решта невпорядкованих елементів масиву. Перший елемент вхідної множини (спочатку це другий елемент усього масиву) стає поточним елементом. Для вставки поточного елемента в відсортовану за зростанням (спадан-

ням) частину масиву його значення запам'ятовується в робочу змінну, вихідна множина переглядається від кінця до початку і кожен її елемент порівнюється з поточним: якщо поточний елемент менший (більший) від порівнюваного, то порівнюваний елемент зсувається на одну позицію вправо (стає наступним), якщо поточний елемент не менший (не більший) від порівнюваного, то він ставиться на наступну позицію після порівнюваного (цю позицію займав елемент, який перемістився вправо, або це позиція поточного елемента). На кожному кроці сортування вихідна множина збільшується на один елемент, а вхідна зменшується на один елемент.

До речі, метод гномового сортування після незначної модифікації (заміни повернення і перестановок на запам'ятовування невпорядкованого елемента і його вставки у потрібне місце) перетворюється на метод простих вставок.

Часова складність у середньому й найгіршому випадках методу простих вставок щодо порівнянь і перестановок дорівнює $O(n^2)$. На відміну від методів бульбашки і методу вибору кількість порівнянь у методі вставок залежить від початкової впорядкованості масиву — якщо масив впорядковано, то кількість порівнянь дорівнює ($n-1$), а перестановок — жодної. Таким чином, сортування вставками є ідеальним алгоритмом для майже впорядкованих масивів. Порівняно з методом бульбашки метод простих вставок у середньому працює вдвічі швидше.

Наведені вище алгоритми без модифікації чи з модифікацією є досить простими, вони в найгіршому випадку потребують $O(n^2)$ порівнянь елементів (крім методу безглуздого сортування, складність якого становить $O(n^3)$), але кількість перестановок елементів у них може бути різною.

Сортування Шелла[17] (у літературі можна знайти різні варіанти цього методу, наприклад, як узагальнення сортування бульбашкою чи сортування вставками). Дональд Шелл опублікував метод у 1959 році. Основна ідея алгоритму полягає в тому, що спочатку порівнюються далеко розміщені один

[17] Дональд Шелл (Donald Lewis Shell, 1924-2015) — американський вчений у галузі інформатики.

від одного, а не сусідні елементи, як у звичайних сортуваннях. Це призводить до швидкого усунення масової невпорядкованості, завдяки чому на більш пізній стадії сортування залишається менше роботи. Спочатку відстань між порівнюваними елементами, як правило, дорівнює половині довжини всього масиву. Потім ця відстань поступово (кожен раз удвічі) зменшується до одиниці, і при цьому все зводиться до звичайного сортування.

Приклад 8.4.5. Програмна реалізація методу Шелла з використанням як основи методу сортування бульбашкою з індикатором перестановки:

```
/* Сортування Шелла – порівнюються і переставляються
   елементи на великій відстані h (спочатку h=N/2), яка
   поступово зменшується вдвічі */
...
#define N 10
int po=0,pe=0;   //кількість порівнянь і перестановок
...
/* Сортування Шелла для впорядкування за зростанням
   (модифікація методу бульбашки з індикатором) */
void sort(int a[N]) {
  int i, j,   // індекси поточного і наступного елементів
      h,   // відстань між елементами
      ind, // індикатор перестановки
      r;   // робоча змінна
  h=N/2;   // початкова відстань між порівнюваними елементами
  while ( h>0) {
    ind=1;
    // порівняння й перестановка на відстані h
    for (i=0; i<N-h && ind; i++)  { // бульбашка з індикатором
      ind=0; po++;
      for (j=0; j<N-i-h; j++) {
        po++;
        if(a[j]>a[j+h]) {
          pe++;
```

```
            r=a[j]; a[j]=a[j+h]; a[j+h]=r;
            ind=1;   // перестановка відбулася
          }
        }
      }
      h/=2;  // зменшення відстані вдвічі
    }
  }
```

```
Початковий масив
100  21  40  5  1  9  12  18  7  45
   h=5
i=0
 0 5<>  1 6<>  2 7<>  3 8   4 9
i=1
 0 5    1 6    2 7    3 8
   h=2
i=0
 0 2    1 3<>  2 4<>  3 5   4 6    5 7<>  6 8<>  7 9<>
i=1
 0 2<>  1 3    2 4    3 5   4 6<>  5 7    6 8
i=2
 0 2    1 3    2 4<>  3 5   4 6    5 7
i=3
 0 2    1 3    2 4    3 5   4 6
   h=1
i=0
 0 1    1 2    2 3    3 4<> 4 5    5 6<>  6 7    7 8<>  8 9
i=1
 0 1    1 2    2 3    3 4   4 5    5 6    6 7<>  7 8
i=2
 0 1    1 2    2 3    3 4   4 5    5 6    6 7
відсортований (Шелла)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 68
Кількість перестановок 15
```

```
Початковий масив
100  45  40  21  18  12  9  7  5  1
   h=5
i=0
 0 5<>  1 6<>  2 7<>  3 8<>  4 9<>
i=1
 0 5    1 6    2 7    3 8
   h=2
i=0
 0 2<>  1 3<>  2 4<>  3 5    4 6    5 7<>  6 8<>  7 9<>
i=1
 0 2<>  1 3    2 4    3 5    4 6    5 7<>  6 8
i=2
 0 2    1 3    2 4    3 5    4 6    5 7
   h=1
i=0
 0 1    1 2    2 3    3 4    4 5    5 6    6 7    7 8    8 9
відсортований (Шелла)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 45
Кількість перестановок 13
```

```
Початковий масив
1  5  7  9  12  18  21  40  45  100
   h=5
i=0
 0 5    1 6    2 7    3 8    4 9
   h=2
i=0
 0 2    1 3    2 4    3 5    4 6    5 7    6 8    7 9
   h=1
i=0
 0 1    1 2    2 3    3 4    4 5    5 6    6 7    7 8    8 9
відсортований (Шелла)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 25
Кількість перестановок 0
```

На скриншотах подано програмно створені протоколи процесу сортування при різному початковому впорядкуванні 10-елементного масиву. Можна побачити, що крок h набуває значень 5, 2 і 1. При цих кроках параметр зовнішнього циклу методу бульбашки вказує проходи масиву; для кожного проходу виведено пари індексів порівнюваних елементів, символами <> позначено перестановку елементів. З прикладів, наведених на скриншотах видно, що кількість порівнянь і перестановок при початковому випадковому розміщенні елементів більша, ніж при їхньому зворотному початковому впорядкуванні. Отже, поведінка алгоритму сортування Шелла не є природною.

Складність алгоритму сортування Шелла становить $O(n^2)$, але середня кількість порівнянь, визначена емпіричним шляхом, дорівнює $O(n \log^2 n)$.

Швидке сортування. Метод швидкого сортування розробив у 1962 році англійський вчений у галузі інформатики та обчислювальної техніки Чарльз Ентоні Річард Гоар[18]. Цей метод, як і повільний метод бульбашки, є методом обміну. Проте його основною відмінністю є організація обмінів між далекими за розташуванням елементами, що призводить до істотного прискорення роботи. Швидке сортування базується на стратегії розділяй і володарюй. Таким чином, це метод обміну з розділенням.

Основна ідея методу така. Вибирається деякий елемент, який називають медіаною (як правило, медіаною вибирають перший, чи середній, чи останній елемент масиву, чи елемент вибирають випадково). Усі елементи масиву роз-

[18] Чарльз Ентоні Річард Гоар (Charles Antony Richard Hoare, 1934 р. н.) — англійський вчений у галузі інформатики та обчислювальної техніки.

биваються на два підмасиви — елементів менших і елементів більших від медіани. Після цього у підмасивах рекурсивно виконують ті ж самі дії, доки в кожному підмасиві не стане по одному елементу. Масив буде відсортовано.

Нехай медіаною є середній за розташуванням елемент масиву. Розбиття масиву на підмасиви відбувається так. Порівнюється медіана з першим елементом, якщо впорядкованість не порушується, то з другим і т. д. (хід зліва направо), доки не порушиться впорядкованість (тобто буде знайдено елемент, більший або рівний медіані; таким елементом може бути і сама медіана). Далі порівнюється медіана з останнім елементом, якщо впорядкованість не порушується, то з передостаннім і т. д. (хід справа наліво), доки не порушиться впорядкованість (тобто буде знайдено елемент, менший або рівний медіані; таким елементом може бути і сама медіана). Знайдені лівий і правий елементи переставляються місцями. Знову медіана порівнюється з лівими і правими елементами, доки не порушаться впорядкованості. Знайдені елементи переставляються місцями і т. д. Після порівняння медіани з усіма елементами, вона буде розміщуватися так (не обов'язково по центру масиву), що зліва від неї будуть менші елементи, а справа більші. Рекурсивно алгоритм треба застосувати до лівої і правої частин масиву. Швидкість методу залежить від того, чи справді медіана буде ділити масив навпіл.

Приклад 8.4.5. Програмна реалізація методу швидкого сортування:

```
/* Швидке сортування - у діапазоні елементів вибирається
   медіана і відносно неї утворюються дві підмножини -
   з меншими і з більшими елементами. Далі все повторюється
   рекурсивно щодо підмножин */
...
#define N 10
int po=0, pe=0,   //кількість порівнянь і перестановок
    re=0;   //кількість рекурсивних звернень
...
  sort(a, 0, N-1);
...
```

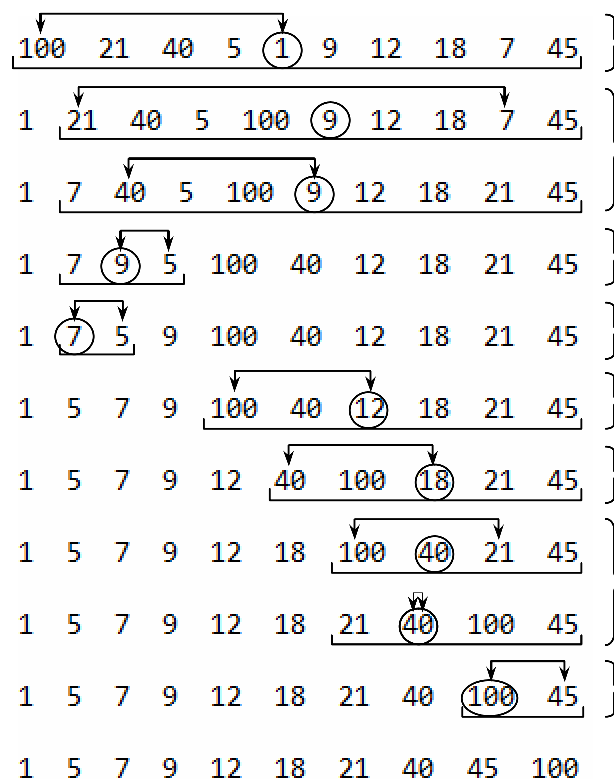
```
/* Швидке сортування - впорядкування за зростанням.
   Використано стратегію ділення масиву – на проміжку від
   індексу p до індексу k вибирається серединний елемент me
   і відносно нього більші елементи і менші елементи взаємно
   переставляються. Алгоритм виконується рекурсивно */
void sort(int a[N], int p, int k) {
  int i, j,    // індекси лівого і правого елементів; ці
               // індекси також визначають межі підмасивів
               // при рекурсивних зверненнях
      me,  // медіана
      r;   // робоча змінна
  re++;   //кількість рекурсивних звернень
  i=p; j=k;    // індекси початку і кінця проміжку
  me=a[(i+j)/2];   // медіана вибирається в центрі проміжку
  while (i<=j) {   // пошук і перестановка елементів
    while (po++, a[i]<me)   // рух зліва направо
      i++;
    while (po++, a[j]>me)   // рух справа наліво
      j--;
    if (i<=j) {   // перестановка елементів відносно медіани
      if (i!=j) {
        pe++;  // кількість перестановок
        r=a[i]; a[i]=a[j]; a[j]=r;
      }
      i++; j--;   // перехід до наступних елементів
    }
  }
  if (p<j)   // рекурсія щодо лівої частини проміжку
    sort(a, p, j);
  if (k>i)   // рекурсія щодо правої частини проміжку
    sort(a, i, k);
}
```

```
Початковий масив
100  21  40  5  1  9  12  18  7  45
відсортований (швидке сортування)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 50
Кількість перестановок 9
Кількість рекурсивних звернень 8
```

```
Початковий масив
100  45  40  21  18  12  9  7  5  1
відсортований (швидке сортування)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 34
Кількість перестановок 5
Кількість рекурсивних звернень 7
```

```
Початковий масив
1  5  7  9  12  18  21  40  45  100
відсортований (швидке сортування)
1  5  7  9  12  18  21  40  45  100
Кількість порівнянь 31
Кількість перестановок 0
Кількість рекурсивних звернень 6
```

Наприклад, для випадково впорядкованого масиву {100, 21, 40, 5, 1, 9, 12, 18, 7, 45} послідовність перестановок (їх виконано 9) є такою (на схемі нижня горизонтальна дужка позначає частину масиву, яка аналізується, кружечками позначено медіанні елементи, верхня квадратна дужка зі стрілками вказує на елементи, які треба переставити місцями, фігурні дужки відповідають окремим рекурсивним зверненням до функції sort):



Зі скриншотів видно, що в методі швидкого сортування порівняно з методом бульбашки кількість порівнянь більша, але кількість перестановок

(які виконуються значно довше, ніж порівняння) значно менша. За рахунок цього метод швидкого сортування працює швидше.

Швидке сортування на деяких простих масивах даних, наприклад, якщо його застосовувати для сортування вже відсортованого масиву, є неефективним. Крім того, рекурсивні виклики є затратними операціями, і тому для невеликих масивів даних (наприклад, з 32 елементами і менше) швидке сортування буде працювати не дуже ефективно — для сортування невеликої кількості елементів треба використовувати нерекурсивні методи, наприклад, сортування вставками або вибором. У найгіршому випадку алгоритм швидкого сортування має складність $O(n^2)$. Проте цей алгоритм на практиці часто виявляється оптимальним завдяки тому, що в середньому час його роботи набагато кращий — становить $O(n \log n)$ з невеликим постійним множником.

Алгоритм швидкого сортування добре досліджений щодо його продуктивності, він вдосконалювався і став найкращим для широкого спектру завдань сортування. Цей алгоритм не використовує додаткової пам'яті, тому він добре працює навіть у середовищах з віртуальною пам'яттю (поєднання оперативної пам'яті комп'ютера з тимчасовим простором на жорсткому диску).

Злиття двох відсортованих масивів. При злитті двох відсортованих (наприклад, за зростанням) масивів a і b відповідно з кількостями елементів n і m буде одержано новий масив ab з $n + m$ елементами. Злиття починається з порівняння перших елементів відповідних масивів a і b. За лічильником i вибираються елементи з масиву a, за лічильником j — з масиву b, а за лічильником k — елементи заносяться до результуючого масиву ab. У масив ab заноситься менший з порівнюваних елементів, а далі в порівнянні бере участь наступний елемент того масиву, елемент якого вже записаний до масиву ab. Ця процедура повторюється, доки елементи одного з масивів не закінчаться. Наприкінці треба переписати до масиву ab всі елементи іншого масиву, які залишилися.

Приклад 8.4.6. Програма злиття двох масивів. Програма має модульну структуру і містить функції: введення елементів масиву (vvid), виведення

масиву (vyvid), сортування масиву методом бульбашки (sortbul), злиття двох відсортованих масивів (zlyttia).

```
    /* Злиття двох масивів */
...
#define N 5
#define M 4
void vvid(int *a, int n);
void vyvid(int *a, int n);
void sortbul(int *a, int n);
void zlyttia(int *a, int n, int *b, int m, int *ab);
int main() {
  int a[N], b[M];   // перший і другий масиви
  int ab[N+M];   // масив, утворений злиттям
  /* Заповнення масивів */
  printf("\Введіть елементи масиву a[%d]:\n",N); vvid(a,N);
  printf("\Введіть елементи масиву a[%d]:\n",M); vvid(b,M);
  /* Виведення масивів */
  printf("\Перший масив:\n"); vyvid(a,N);
  printf("\nДругий масив:\n"); vyvid(b,M);
  /* Сортування масивів івиведення */
  sortbul(a,N); sortbul(b,M);
  printf("\nВідсортований перший масив:\n"); vyvid(a,N);
  printf("\nВідсортований другий масив:\n"); vyvid(b,M);
  /* Злиття відсортованих масивів і виведення */
  zlyttia(a,N,b,M,ab);
  printf("\nРезультуючий злитий масив:\n"); vyvid(ab,N+M);
...
}
/* Ввід елементів масиву */
void vvid(int *a, int n) {
  int i;
  for (i=0; i<n; i++) scanf("%d",&a[i]);
}
```

```
/* Вивід елементів масиву */
void vyvid(int *a, int n) {
  int i;
  for (i=0; i<n; i++) printf("%d  ",a[i]);
}
/* Сортування масиву методом звичайної бульбашки */
void sortbul(int *a, int n) {
  int i, j, r;
  for (i=0; i<n-1; i++)
    for (j=0; j<n-i-1; j++)
      if (a[j]>a[j+1]) { r=a[j]; a[j]=a[j+1]; a[j+1]=r; }
}
/* Злиття двох відсортованих масивів */
void zlyttia(int *a, int n, int *b, int m, int *ab) {
  int i, j, k;   // індекси масивів a, b, ab
  int l;   // індекс для дозапису
  /* Злиття, доки не буде досягнуто кінця одного з масивів */
  for (i=0, j=0, k=0; i<n && j<m; k++)
    if (a[i]<b[j])
      ab[k]=a[i++];
    else
      ab[k]=b[j++];
  /* Дозапис елементів з іншого масиву */
  if (i==n)
    for (l=k; l<n+m; l++)
      ab[l]=b[j++];
  else
    for (l=k; l<n+m; l++)
      ab[l]=a[i++];
}
```

```
Введіть елементи масиву a[5]:
5 0 3 8 4
Введіть елементи масиву b[4]:
6 2 9 1
Перший масив:
  5  0  3  8  4
Другий масив:
  6  2  9  1
Відсортований перший масив:
  0  3  4  5  8
Відсортований другий масив:
  1  2  6  9
Результуючий злитий масив:
  0  1  2  3  4  5  6  8  9
```

Алгоритм злиття двох відсортованих масивів має часову складність $O(n+m)$.

8.5. Приклади алгоритмів з використанням одновимірних масивів

Крім виконання операцій зсувів, пошуку елементів, сортування і злиття, алоритми з масивами використовують для розв’язання багатьох інших задач. Такими задачами є обробка текстів, робота з записами (структурами), робота з великими числами, різноманітні математичні задачі тощо.

Для обробки рядків символів у мові С використовують спеціальні функції. Якщо ж обробляти текст «вручну», то робота з рядком символів виконується, як з масивом — кожен елемент масиву є окремим символом, останній елемент містить нуль-символ '\0' з числовим кодом 0 і відіграє роль ознаки кінця рядка. Для зчитування введеної символьної інформації функцію scanf використовувати не доцільно, оскільки вона зчитує всі символи до першого пробільного символу (пробіл, Tab, Enter). Інформацію краще зчитувати поелементно за допомогою функції getchar з бібліотеки stdio.h. Функція isalpha з бібліотеки ctype.h розпізнає великі й малі латинські букви.

Приклад 8.5.1. Підрахунок кількість англійських слів у введеному тексті. Кінець тексту позначено символом #.

Програмна реалізація може бути такою:

```
/* Підрахунок кількості англійських слів у введеному тексті.
   При описові char str[N] функція isalpha(str[i]) кирилицю
   не сприймає (при обробці видає помилку); якщо задати
   isalpha(str[i]&255), то сприймає. При описові unsigned
   char str[N] функція isalpha(str[i]) кирилицю сприймає */
...
#include <ctype.h>
#define N 80
int main() {
  int i,   // індекс поточного символу
      n,   // кількість слів
      ind;   // індикатор слова: 1 - слово, 0 - роздільник
  char str[N+1];   // рядок довжиною N символів
  printf(" Введіть текст до символа #\n");
```

```
  i=0;
  /* Введення тексту до граничної кількості символів чи символу #
     (після # можуть бути ще символи, але вони не вводяться) */
  do {
    i++; str[i]=getchar();
  } while (i<N && str[i]!='#');
  while(getchar()!='\n');    // очистка буфера
  str[i]=' '; // фіктивний пробіл, щоб однотипно обробляти текст
  str[i+1]='\0';    // позначення кінця рядка
  printf("\n Введений текст:\n%s\n",str);
  /* Підрахунок кількості іноземних слів */
  n=0; i=0; ind=0;
  while (str[i]!='\0') {
    if (isalpha(str[i]&255) && !ind)
      ind=1;
    else
      if (!isalpha(str[i]&255) && ind) {
        ind=0; n++;
      };
    i++;
  }
  printf("\n Кількість іноземних слів у тексті: %d",n);
...
}
```

```
 Введіть текст до символа #
Visual Studio -середовище програмування.
Мови програмування C, C++, C#

 Введений текст:
Visual Studio -середовище програмування.
Мови програмування C, C++, C

 Кількість іноземних слів у тексті: 5
```

Програма вибирає тільки англійські слова (Visual, Studio, C, C, C)

```
 Введіть текст до символа #
Мова С, С++, С##

 Введений текст:
Мова С, С++, С

 Кількість іноземних слів у тексті: 0
```

Весь текст набрано на українській розкладці клавіатури

```
 Введіть текст до символа #
Плагіат не пройде!
Символи буде виявлено #

 Введений текст:
Плагіат не пройде!
Символи буде виявлено

 Кількість іноземних слів у тексті: 11
```

У тексті подібні кириличні символи замінено англійськими. Вони розпізнаються як окремі англійські слова (a, ia, e, o, e, C, o, y, e, e, o)

Згрупувати кілька логічно зв'язаних змінних, типи яких можуть бути різними, і надати їм одне ім'я дає можливість використання структур (записів). У мові програмування С структури можна присвоювати, передавати функціям як аргументи, функції можуть повертати структуру як результат (тобто мати тип структури). Кожен елемент структури може мати будь-який допустимий тип даних, у тому числі масиви й інші структури. Можна визначати масиви структур. Кожен елемент масиву структур є структурою. Доступ до окремої структури в масиві здійснюється за допомогою імені структури й індексу. Доступ до окремих елементів (полів) структури здійснюється за допомогою складних імен: *ім'я структури.ім'я елемента*.

Приклад 8.5.2. Сортування масиву структур. Визначається масив структур, який містить інформацію про n осіб: прізвище, ім'я і вік. Дані про перших $(n-1)$ особу задаються при визначенні масиву структур шляхом ініціалізації, про останню особу — вводяться в діалозі. Усі дані про осіб сортуються за віком за зростанням (застосовано метод бульбашки).

Програма роботи з масивом структур:

```
/* Сортування масиву структур за полем vik */
...
#define N 5
struct dani { // дані про особу
  char prizv[20];   // прізвище
  char imia[20];   // ім'я
  int vik;   // вік
};   // ; обов'язкова в оголошенні структури
struct dani vvids();
void sortbuls(struct dani osoba[], int n);
```

```
void vyvids(struct dani osoba[], int n);
/* Робота зі структурами */
int main() {
  struct dani osoba[N]={
             {"Кублій","Лариса",59},
             {"Ющук", "Іван", 86},
             {"Війтик", "Ольга", 85},
             {"Кублій", "Михайло", 27}
  };
  printf("Введіть дані про останню особу\n");
  printf("\t(прізвище, ім'я і вік особи)\n");
  osoba[N-1]=vvids();
  printf("Початкові дані\n");
  vyvids(osoba, N);
  sortbuls(osoba, N);
  printf("Відсортовані за віком дані\n");
  vyvids(osoba, N);
...
}
/* Ввід даних в елементи структури */
struct dani vvids() {
  struct dani danir;
  scanf("%s%s%d", danir.prizv, danir.imia, &danir.vik);
  return danir;
}
/* Сортування методом бульбашки за полем "vik" */
void sortbuls(struct dani osoba[], int n) {
  struct dani danir;
  int i, j;
  for (i=0; i<n-1; i++)
    for (j=0; j<n-i-1; j++)
      if (osoba[j].vik>osoba[j+1].vik) {
        danir=osoba[j];
```

```
        osoba[j]=osoba[j+1];
        osoba[j+1]=danir;
      }
}
/* Вивід полів структури */
void vyvids(struct dani osoba[], int n) {
  int i;
  for (i=0; i<n; i++)
    printf("\t%s %s %d\n", osoba[i].prizv,
    osoba[i].imia, osoba[i].vik);
}
```

```
Введіть дані про останню особу
        (прізвище, ім'я і вік особи)
Ющук Олег
  55
Початкові дані
        Кублій Лариса 59
        Ющук Іван 86
        Війтик Ольга 85
        Кублій Михайло 27
        Ющук Олег 55
Відсортовані за віком дані
        Кублій Михайло 27
        Ющук Олег 55
        Кублій Лариса 59
        Війтик Ольга 85
        Ющук Іван 86
```

У мові програмування С для чисел типу int максимальне значення дорівнює $2^{31}-1=2147483647$ (10-значне десяткове число), а для чисел типу unsigned long long int — $2^{64}-1=18446744073709551615$ (20-значне десяткове число). Але, наприклад, факторіал малого числа 21 є великим числом, яке дорівнює 51090942171709440000 — воно теж 20-значне, але призводить до арифметичного переповнення навіть при використанні типу unsigned long long int.

При роботі з великими числами для їхнього зберігання використовують масиви цілих чисел чи рядки символів. Розглянемо приклад з використанням масиву цілих чисел.

Приклад 8.5.3. Знаходження ***факторіалу великого числа***.

Велике число будемо зберігати в масиві цілих чисел типу int, де кожен елемент зберігає одну чи кілька цифр числа. Для економії місця в пам'яті

вигідніше розміщувати в одному елементі масиву якомога більше цифр. При цьому треба стежити, щоб результати всіх проміжних операцій не перевищували максимального значення для обраного типу даних (щоб не було арифметичного переповнення). Розглянемо, наприклад 14-значне число і подамо його в системі числення за основою d=1000000:

$$12345678901234 = 12 \cdot (10^6)^2 + 345678 \cdot (10^6)^1 + 901234 \cdot (10^6)^0 .$$

Для його зберігання треба всього 3 елементи масиву A[2], A[1] і A[0].

Розглянемо множення числа

$$A = a_n d^n + a_{n-1} d^{n-1} + ... + a_2 d^2 + a_1 d^1 + a_0$$

на число k, у результаті чого одержимо число

$$A \cdot k = \alpha_m d^m + \alpha_{m-1} d^{m-1} + ... + \alpha_2 d^2 + \alpha_1 d^1 + \alpha_0 .$$

При цьому

$\alpha_0 = (a_0 \cdot k)\%d$ і в наступний розряд додається перенос $r_1 = (a_0 \cdot k)/d$;

$\alpha_1 = (a_1 \cdot k + r_1)\%d$ і перенос $r_2 = (a_1 \cdot k + r_1)/d$

$\alpha_2 = (a_2 \cdot k + r_2)\%d$ і перенос $r_3 = (a_2 \cdot k + r_2)/d$ і т. д.

Обчислення закінчуються, коли одночасно виконуються дві умови:

— усі цифри числа A опрацьовано;

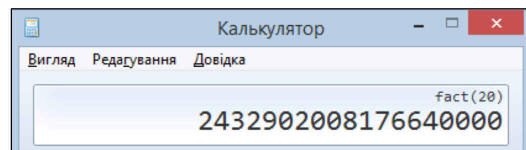
— перенос у наступний розряд дорівнює 0.

Програма, яка реалізує розглянутий алгоритм:

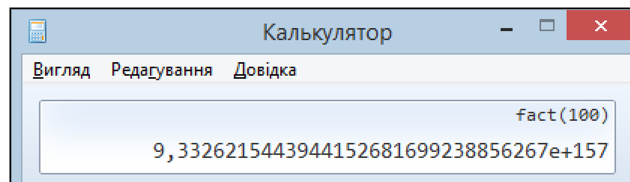
```
/* Знаходження факторіалу великого числа.
   Працюємо з числами за основою 1 000 000 - кожен елемент
   масиву A зберігає одну цифру від 0 до 999999 */
...
#include <stdio.h>
#include <stdlib.h>
#define N 40
int main() {
  const int d = 1000000; // d – основа - 1 000 000, тобто
                         // цифри є від 0 до 999999
  int A[N] = {1}; // A[0]=1 (=1!), решта A[i]=0; розряди числа
```

```
int n; // число, для якого обчислюється факторіал
int i, // поточний номер розряду числа - індекс елемента масиву A
  k, // k - множник при обчисленні факторіалу 2, 3, 4, ..., n
  len, // len – довжина числа (кількість елементів масиву A)
  r, // r – остача – перенос у наступний розряд
  s; // робоча змінна для зберігання A[i]*k + r
printf("Обчислення факторіалу числа n\nВведіть значення n: ");
scanf("%d",&n);
len = 1;
for ( k = 2; k <= n; k ++ ) { // множимо на 2, 3, ..., n
  i = 0; // починаємо з молодшого розряду (0)
  r = 0; // спочатку перенос - 0
  while ( i < len // доки не всі розряди опрацьовано
    || r > 0 ) { // або є перенос
      s = A[i]*k + r; // множимо розряд, додаємо перенос
                      // з попереднього розряду
      A[i] = s % d; // залишається в поточному розряді
      r = s / d; // перенос в наступний розряд
      i ++; // перехід до наступного розряду
  }
  len = i; // змінюємо довжину числа
}
// вивід на екран
printf("Результат:\n");
printf("%d ", A[len-1]); // 123 друкує 123
for (i = len-2; i >= 0; i--) // від старшого розряду
  printf("%06d ", A[i]); // 123 друкує 000123
...
}
```

```
Обчислення факторіалу числа n
Введіть значення n: 20
Результат:
2 432902 008176 640000
```




```
Обчислення факторіалу числа n
Введіть значення n: 100
Результат:
93 326215 443944 152681 699238 856266 700490 715968 264381 621468 592963 895217
599993 229915 608941 463976 156518 286253 697920 827223 758251 185210 916864 000
000 000000 000000 000000
```



Факторіал числа 20 є 3*6+1=19-значним числом, а факторіал числа 100 є 26*6+2=158-значним числом.

Раніше (див. приклад 6.3.13) було розглянуто алгоритм знаходження простих чисел (прості числа діляться лише самі на себе і на 1; число 1 не є простим; першими простими є числа: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, ...) з використанням остачі від ділення. Знайти прості числа також можна, використовуючи ***решето Ератосфена***[19].

Алгоритм Ератосфена полягає в тому, що треба взяти перше просте число — 2; всі числа, кратні 2, вилучити — в масиві замінити їх на 0. Далі в масиві знайти наступне число, не рівне 0; це буде просте число; всі наступні числа, кратні йому, вилучити і т. д.

Приклад 8.5.4. Пошук простих чисел з використанням решета Ератосфена.

Програмна реалізація алгоритму:

```
/* Решето Ератосфена для пошуку простих чисел
   до заданого числа включно */
...
#define N 1000
int main() {
  int a[N];
  int n;    // задане число
  int i,    // індекс елемента масиву
      k;    // індекс простого числа
  int pr, kr;   // просте число і кратне йому число
  printf("Пошук усіх простих чисел до вказаного числа\n");
  printf("Введіть число:  ");
  scanf("%d",&n);
```

[19] Ератосфен (Eratosthenes, близько 275-194 до н. е.) — давньогрецький надзвичайно різнобічний науковець і письменник.

```
  for (i=0; i<n-1; i++) a[i]=i+2;    // заповнення масиву
                                     // числами від 2 до n
  k=-1;
  for (i=0; i<n-1; i++)    // i<n-1, бо починаємо з числа 2
    if (a[i]!=0) {    // шукаємо наступне просте число
      pr=a[i];    // беремо наступне просте число pr
      for (kr=pr; kr<=n; kr+=pr)
        a[kr-2]=0;    // вилучаємо числа (замінюємо на 0), кратні pr
      k++; a[k]=pr;  // збираємо на початку масиву всі прості числа
    }
  printf("Прості числа:\n ");
  i=-1;
  while (a[++i]!=0)
    printf("%d  ",a[i]);
...
}
```

```
Пошук усіх простих чисел до вказаного числа
Введіть число:  1000
Прості числа:
   2    3    5    7   11   13   17   19   23   29   31   37   41   43   47   53
  59   61   67   71   73   79   83   89   97  101  103  107  109  113  127  131
 137  139  149  151  157  163  167  173  179  181  191  193  197  199  211  223
 227  229  233  239  241  251  257  263  269  271  277  281  283  293  307  311
 313  317  331  337  347  349  353  359  367  373  379  383  389  397  401  409
 419  421  431  433  439  443  449  457  461  463  467  479  487  491  499  503
 509  521  523  541  547  557  563  569  571  577  587  593  599  601  607  613
 617  619  631  641  643  647  653  659  661  673  677  683  691  701  709  719
 727  733  739  743  751  757  761  769  773  787  797  809  811  821  823  827
 829  839  853  857  859  863  877  881  883  887  907  911  919  929  937  941
 947  953  967  971  977  983  991  997
```

Часом виникає потреба, наприклад, для застосування алгоритму сортування підрахунком, визначити кількість різних елементів у поданій сукупності.

Приклад 8.5.5. Визначення ***кількості різних елементів масиву***. Алгоритм такий: береться поточний елемент масиву $a[i]$ і порівнюється з наступними. Якщо серед наступних є рівний, то далі порівняння не виконується і береться наступний елемент $a[i+1]$; інакше, якщо при проході до кінця масиву більше рівних елементів не буде знайдено, то кількість збільшується на 1.

Програмна реалізація алгоритму:

```
/* Визначення кількості різних елементів у масиві */
...
#define N 10
int main() {
  int a[N]={19,2,4,5,8,4,3,1,7,5};
  int i,   // індекс поточного елемента
      j,   // індекс наступного елемента
      k;   // кількість різних елементів
  system("chcp 1251 & cls");
  printf("Початковий масив\n");
  for (i=0; i<N; i++) printf("%d  ",a[i]);
  /* Підрахунок */
  k=1;
  for (i=0; i<N-1; i++) {
    j=i+1;
    while (j<N && a[i]!=a[j]) j++;
    if (j==N) k++;
  }
  printf("\nКількість різних елементів %d", k);
...
```

```
Початковий масив
19  2  4  5  8  4  3  1  7  5
Кількість різних елементів 8
```

```
Початковий масив
1  1  4  1  1  4  1  1  7  5
Кількість різних елементів 4
```

У прикладі 6.3.14 було розглянуто різні алгоритми Евкліда для знаходження найбільшого спільного дільника двох чисел, а в прикладі 7.4.1 алгоритм з використанням ділення було оформлено як функцію. На практиці може виникнути необхідність знайти найбільший спільний дільник для трьох і більшої кількості чисел. Для цього треба використати таку властивість:

$$\text{НСД}(a,b,c,...,d) = \text{НСД}(\text{НСД}(\text{НСД}(\text{НСД}(a,b),c),...),d),$$

тобто спочатку знайти найбільший спільний дільник двох чисел (скориставшись будь-яким з розглянутих раніше алгоритмів), потім знайти найбільший спільний дільник для нього і третього числа і т. д.

Приклад 8.5.6. Знаходження ***найбільшого спільного дільника кількох чисел***, поданих масивом.

Програмно алгоритм можна реалізувати так (для знаходження найбільшого спільного дільника двох чисел використано функцію з прикладу 7.4.1 — алгоритм Евкліда на основі ділення):

```
/* Знаходження найбільшого спільного дільника кількох чисел
   з використанням алгоритму Евкліда */
...
#define N 5
  int a[N]={264, 462, 396, 165, 297};   // масив чисел
  int c;   // НСД
...
  c=nsd(a[0], a[1]);
  for (i=2; i<N; i++)
    c=nsd(c, a[i]);
  printf("\nНСД=%d ",c);
...
/* Знаходження НСД двох чисел.
   Алгоритм Евкліда на основі ділення */
int nsd(int ch, int zn) {
  int r;   // остача від ділення
  while (zn) {
    r=ch%zn; ch=zn; zn=r;
  }
  return ch;
}
```

Приклад 8.5.7. Обчислення ***кута між двома векторами*** (роботу алгоритму реалізовано зі статичними й динамічними масивами). В алгоритмі використовується формула $\cos\alpha = \frac{(\vec{a},\vec{b})}{|\vec{a}|\cdot|\vec{b}|}$; перехід від радіанів до градусів здійснюється за формулою $\varphi_{\circ} = \varphi_{r} \cdot 180/\pi$.

а) Програмна реалізація алгоритму з використанням статичних масивів:

```
/* Знаходження кута між двома векторами з використанням
   статичних одновимірних масивів */
```

```
...
#define _USE_MATH_DEFINES
#include <math.h>
#define N 10   // максимальна кількість елементів масиву
void vvid(int *a, int n);
int dobutok(int *a, int *b, int n);
int main() {
  int a[N],b[N];   // виділення максимальної пам'яті під масиви
  int n;   // реальна довжина вектора
  double alfa;
  printf("Вкажіть розмірність векторів ");
  scanf("%d",&n);
  printf("Введіть координати вектора a "); vvid(a,n);
  printf("Введіть координати вектора b "); vvid(b,n);
  /* Обчислення кута */
  alfa=acos(dobutok(a,b,n)/
       sqrt((double)dobutok(a,a,n)*dobutok(b,b,n)))/M_PI*180.;
  printf("Кут між векторами %0.2f°",alfa);
...
}
/* Ввід координат вектора */
void vvid(int *a, int n) {
  int i;
  for (i=0; i<n; i++) scanf("%d",&a[i]);
}
/* Знаходження скалярного добутку двох векторів */
int dobutok(int *a, int *b, int n) {
  int i;
  int s=0;
  for (i=0; i<n; i++)
    s+=a[i]*b[i];
  return s;
}
```

```
Вкажіть розмірність векторів 5
Введіть координати вектора a 1 2 3 4 5
Введіть координати вектора b 5 4 3 2 1
Кут між векторами 50.48°
```

```
Вкажіть розмірність векторів 3
Введіть координати вектора a 0 1 0
Введіть координати вектора b 0 0 1
Кут між векторами 90.00°
```

У програмі для кожного масиву-вектора статично (ще на етапі компіляції) виділяється пам'ять для зберігання N елементів, а при виконанні програми користувач задає значення *n* (*n* має бути меншим або рівним N) — реальну довжину векторів. При цьому частина пам'яті, виділена під масив, може не використовуватися.

б) Програмна реалізація алгоритму з використанням динамічного виділення пам'яті під масиви:

```
/* Знаходження кута між двома векторами.
   Робота з динамічними одновимірними масивами.
   Обчислення виконуються багато разів */
...
#define _USE_MATH_DEFINES
#include <math.h>
void vvid(int *a, int n);
int dobutok(int *a, int *b, int n);
int main() {
  int *a,*b;
  int ind;   // індикатор виконання програми
  char vidp;   // відповідь щодо продовження роботи
  int n;   // реальна довжина вектора
  double alfa;   // значення кута між векторами
  ind=1;
  /* Багаторазове виконання обчислень */
  while (ind) {
    printf("Вкажіть розмірність векторів:  ");
    scanf("%d",&n);
    /* Виділення пам'яті під масиви */
    a=(int *)malloc(n*sizeof(int));
    b=(int *)malloc(n*sizeof(int));
    if (!a || !b)
      printf("  Недостатньо пам'яті");
    else {
```

```
      printf("Введіть координати вектора a:  ");
      vvid(a,n);
      printf("Введіть координати вектора b:  ");
      vvid(b,n);
      /* Обчислення кута */
      alfa=acos(dobutok(a,b,n)/sqrt((double)dobutok(a,a,n)
           *dobutok(b,b,n)))/M_PI*180.;
      printf("  Кут між векторами %0.2f°",alfa);
    }
    /* Звільнення місця в пам'яті */
    free(a); free(b);
    /* Запит про продовження роботи */
    printf("\n\nПродовжувати роботу (Y - так)? ");
    fseek(stdin,0,SEEK_END);
    vidp=getchar();
    fseek(stdin,0,SEEK_END);
    if (!(vidp=='Y'||vidp=='y'||vidp=='Т'||vidp=='т'))
      ind=0;
  }
...
```

```
Вкажіть розмірність векторів:  5
Введіть координати вектора a:  0 1 0 0 0
Введіть координати вектора b:  0 0 0 0 1
  Кут між векторами 90.00°

Продовжувати роботу (Y - так)? так
Вкажіть розмірність векторів:  3
Введіть координати вектора a:  1 2 3
Введіть координати вектора b:  2 4 6
  Кут між векторами 0.00°

Продовжувати роботу (Y - так)? yes
Вкажіть розмірність векторів:  4
Введіть координати вектора a:  1 2 3 4
Введіть координати вектора b:  5 6 7 8
  Кут між векторами 14.34°

Продовжувати роботу (Y - так)? Y
Вкажіть розмірність векторів:  500000000
  Недостатньо пам'яті

Продовжувати роботу (Y - так)? ні
```

У програмі, яка працює з динамічними масивами, функції vvid і dobutok такі самі, як і при роботі зі статичною пам'яттю, тому їхні тексти тут не повторюються.

Як видно з поданого фрагмента програми, після виділення пам'яті під масиви робота з масивами не відрізняється від випадку, коли пам'ять виділяється статично.

Перевагою динамічних масивів порівняно зі статичними є те, що їхня розмірність може бути змінною і можна працювати з масивами довільного розміру, не вносячи змін до тексту програми. Динамічне виділення пам'яті дає можливість ефективно її використовувати, проте час виконання програми при цьому збільшується за рахунок звернень до функцій виділення і звільнення пам'яті; крім того, можлива ситуація, що наявної вільної пам'яті для масиву вже не вистачає.

Таким чином, статичний масив краще використовувати тоді, коли кількість елементів відома до виконання програми, і найкраще цю кількість визначати директивою препроцесора #define. Якщо кількість елементів масиву треба задавати під час виконання програми, то доцільно створювати динамічний масив.

8.6. Складність і оптимізація алгоритмів обробки одновимірних масивів

Для розглянутих вище алгоритмів пошуку елементів у масивах і сортування масивів було вказано їхні часові складності, для деяких алгоритмів було наведено міркування щодо їхнього обчислення. Слід зауважити, що всі розглянуті алгоритми пошуку і сортування мають ємнісну складність $O(1)$, оскільки не використовують додаткових масивів; тільки алгоритм пошуку всіх елементів масиву, рівних заданому значенню (приклад 8.3.2), має ємнісну складність $O(n)$.

Часова і ємнісна складності усіх алгоритмів обробки масивів залежать від кількості n елементів масиву.

Приклад 8.6.1. Визначення складності алгоритму бінарного пошуку (див. приклад 8.3.3).

При діленні кожен раз навпіл впорядкованого *n*-елементного масиву одержимо послідовність довжин підмасивів (цих підмасивів є *k*):

$$\frac{n}{2},\quad \frac{n}{2^2},\quad \frac{n}{2^3},\quad \frac{n}{2^4},\ \ldots,\quad \frac{n}{2^k},$$

де $\frac{n}{2^k} < 2$, з чого випливає, що

$$n < 2^{k+1}.$$

Оскільки k — перше ціле число, для якого $\frac{n}{2^k} < 2$, то правильною є й нерівність $\frac{n}{2^{k-1}} >= 2$, з чого випливає, що

$$2^k <= n.$$

Отже:

$$2^k <= n < 2^{k+1}$$

Прологарифмуємо за основою 2 цю нерівність:

$$k <= \log_2 n < k + 1,$$

тобто кількість k проходів циклу приблизно рівна $\log_2 n$, а часова складність алгоритму — $O(\log_2 n)$.

Оскільки в алгоритмі не використовується додаткова пам'ять для зберігання масиву чи його частини, то ємнісна складність алгоритму — $O(1)$.

При розгляді методу швидкого сортування (див. п. 8.4, приклад 8.4.5) було вказано його складність в середньому і в найгіршому випадках — відповідно $O(n \log n)$ і $O(n^2)$. Також було сказано, що швидкість методу залежить від того, чи справді медіана буде ділити масив навпіл.

Приклад 8.6.2. Визначення часової складності алгоритму швидкого сортування.

У загальному випадку аналіз ефективності методу швидкого сортування досить складний. Розглянемо два часткові випадки.

Випадок 1. Нехай $n = 2^k$ (тобто $k = \log_2 n$), тобто є степенем числа 2, і медіана розміщується точно посередині кожного підмасиву і розбиває його на два підмасиви приблизно однакового обсягу.

При першому проході масиву виконується $(n-1)$ порівняння, потім утворюються два підмасиви обсягів приблизно $n/2$. У цих підмасивах виконуються порівняння, загальна кількість яких приблизно становить $2 \cdot (n/2) = n$. Потім обробляються 4 підмасиви обсягів приблизно $n/4$ і кількість порівнянь приблизно дорівнює $4 \cdot (n/4) = n$ і т. д. Процес розбиття припиняється після k проходів, коли всі підмасиви містять по одному елементу. Загальна кількість порівнянь приблизно дорівнює

$$\underbrace{n + n + \ldots + n}_{k \text{ разів}} = n \cdot k = n \cdot \log_2 n,$$

тобто складність алгоритму — $O(n \log n)$.

Якщо початковий масив відсортовано за зростанням, то медіана потрапляє на середину кожного підмасиву, після порівнянь перестановки не виконуються. Якщо початковий масив відсортовано за спаданням, то медіана теж потрапляє на середину масиву; при першому проході після порівнянь виконуються попарні перестановки лівих і правих елементів — у результаті масив буде відсортованим; далі в підмасивах виконуються тільки порівняння. Якщо початкові масиви не відсортовані, але в кожному підмасиві медіана потрапляє на середину, то будуть виконуватися і порівняння, і часом перестановки. У всіх цих випадках складність буде $O(n \log n)$.

Випадок 2. Найгірший випадок швидкого сортування, — коли медіанні елементи весь час потрапляють в одноелементні підмасиви (це буде тоді, коли медіанний елемент є найменшим або найбільшим); при цьому n — будь-яке.

При першому проході масиву виконується $(n-1)$ порівняння, медіанний елемент стає першим (чи останнім) і утворюється правий підмасив з $(n-1)$ елементом. При наступному проході виконується $(n-2)$ порівнянь, медіанний елемент стає першим (чи останнім) і утворюється правий підмасив з $(n-2)$

елементами і т. д.; у кінці буде виконано одне порівняння. Загальна кількість порівнянь дорівнює

$$(n-1)+(n-2)+...+1=\frac{(n-1)+1}{2}\cdot(n-1)=\frac{n\cdot(n-1)}{2}.$$

Отже, часова складність методу швидкого сортування у найгіршому випадку дорівнює $O(n^2)$. Проте на практиці такий випадок є малоймовірним.

Доступ до окремих елементів масиву повільніший, ніж до окремих змінних, оскільки кожна змінна має свою адресу, а при доступові до конкретного елемента масиву його адреса кожен раз обчислюється, оскільки відома лише адреса початку масиву, тобто його першого елемента.

Залежно від того, індексація елементів починається з нуля (наприклад, у мовах програмування C, C++, Java, Lisp індексація всіх масивів обов'язково починається з нуля) чи ні (у мовах Pascal, Delphi, Fortran [20] — початковий індекс може бути від'ємним, нулем чи додатним числом), адреса елемента обчислюється за різними формулами. Так, якщо індексація починається з нуля, то адреса елемента $a[i]$ обчислюється за формулою:

*адреса_масиву+i*sizeof(тип_масиву)* ;

якщо ж індексація починається з будь-якого числа *k*, — то за формулою:

*адреса_масиву+(i-k)*sizeof(тип_масиву)* .

Хоч такі обчислення виконуються досить швидко — для обчислення адреси компілятор додає простий машинний код, у якому треба виконати тільки множення й додавання, — проте на нього, все одно, витрачається певний час.

Незалежно від кількості елементів масиву формула для обчислення адреси буде тою самою. Таким чином, одержання доступу до елемента не залежить від обсягу масиву *n* і має складність $O(1)$.

Зважаючи на сказане, щоб оптимізувати роботу з великими масивами, треба, де це можливо, усунути зайві (повторні) звернення до їхніх елементів, використовуючи в розрахунках звичайні робочі змінні.

Приклад 8.6.3. При пошукові мінімального елемента замість фрагмента

[20] У мові Фортран, крім звичайної індексації, використовують індекси-триплети і вектор-індекси.

```
...
  k=0;
  for (i=1; i<N; i++)
    if (a[i]<a[k])
      k=i;
  printf("Мінімальний елемент a[%d]=%d", k, a[k]);
...
```

який має досить компактний вигляд, але в ньому на кожному проході циклу, крім адреси елемента $a[i]$, обчислюється адреса елемента $a[k]$ (навіть, якщо значення k не змінилося) треба подати фрагмент (див. приклад 8.2.1), в якому замість звернення до елемента $a[k]$ використано, наприклад, робочу змінну *amin*:

```
...
  amin=a[0]; k=0;
  for (i=1; i<N; i++)
    if (a[i]<amin) {
      amin=a[i]; k=i;
    }
  printf("Мінімальний елемент a[%d]=%d", k, amin);
...
```

Проте і цей фрагмент можна оптимізувати, ввівши замість двох звернень до елемента $a[i]$, наприклад, робочу змінну *ai*:

```
...
  amin=a[0]; k=0;
  for (i=1; i<N; i++) {
    ai=a[i];
    if (ai<amin) {
      amin=ai; k=i;
    }
  }
  printf("Мінімальний елемент a[%d]=%d", k, amin);
...
```

хоч така реалізація алгоритму має не зовсім компактний вигляд.

Щоб у певних випадках оптимізувати алгоритм, його треба глибше дослідити і виявити певну закономірність, яка може бути досить індивідуальною. Наступний приклад ілюструє такий випадок.

Приклад 8.6.4. Обчислення скалярного добутку вектора $\vec{a} = (a_1, a_2, ..., a_{n-1}, a_n)$ і вектора зі зворотним порядком елементів $\vec{a}' = (a_n, a_{n-1}, ..., a_2, a_1)$.

Не задумуючись, можна знайти скалярний добуток, як суму:

$$(\vec{a}, \vec{a}') = \sum_{i=1}^{n} a_i \cdot a_{n-i+1} = a_1 \cdot a_n + a_2 \cdot a_{n-1} + ... + a_{n-1} \cdot a_2 + a_n \cdot a_1 .$$

Але в цій сумі рівновіддалені доданки рівні. Тому при її обчисленні кількість проходів циклу можна зменшити вдвічі (при цьому треба врахувати парність числа n).

Оптимізований алгоритм програмно можна подати так:

```
/* Знаходження скалярного добутку вектора на
   вектор зі зворотним порядком елементів */
...
  s=0;
  for (i=0; i<N/2; i++)
    s+=a[i]*a[N-i-1];
  s*=2;
  if (N%2)   // n - непарне
    s+=a[N/2]*a[N/2];
...
```

Якщо з масиву треба вилучити чи в масив додати елемент, то при цьому треба виконати зсуви розміщених правіше елементів відповідно вліво чи вправо. Така операція має часову складність $O(n)$. Крім того, при вставці елементів ще треба контролювати, щоб індекс не вийшов за межі масиву. Тому масиви, як правило, використовують для зберігання та обробки даних з довільним доступом, але без вилучення або додавання нових елементів. Якщо ж алгоритм передбачає інтенсивне вилучення чи додавання елементів, то ефективніше вико-

ристовувати зв’язані списки (хоч списки теж мають свої недоліки — наприклад, через якийсь час елементи можуть бути сильно розкидані по пам’яті).

Для прискорення роботи з масивами, якщо це можливо, слід надавати перевагу послідовній обробці їхніх елементів. Це пов’язано з тим, що елементи масиву розміщуються в пам’яті послідовно і послідовна їхня обробка виконується досить швидко за рахунок того, що в одній сторінці пам’яті, яка завантажується з диска, є зразу кілька елементів, а отже, додатковий досить повільний обмін сторінками між диском і пам’яттю комп’ютера не потрібен.

Щодо використання статичних і динамічних масивів, то, якщо в програмі не треба підтримувати, наприклад, більш ніж 100-елементний масив, можна використовувати статичне виділення пам’яті замість повільнішого динамічного. Проте все залежить від конкретних умов і вимог.

Запитання та завдання для самоконтролю

1. Що таке структура даних? Які структури даних є простими, а які інтегрованими? Що таке масив? Як виділяється пам’ять під масиви? Як вибрати окремий елемент масиву?

2. Вкажіть типові алгоритми роботи з масивами.

3. Як здійснюється лінійний пошук елемента в масиві і яка його часова складність? У чому полягає метод пошуку елемента з бар’єром? Як здійснюється бінарний пошук елемента у впорядкованому масиві і яка його часова складність? Яка ємнісна складність алгоритмів пошуку елемента; який алгоритм пошуку має ємнісну складність $O(n)$?

4. У чому полягає і для чого використовують сортування об’єктів? Які є методи сортування? Що може бути ключем сортування? Яку поведінку алгоритму сортування називають природною? Який алгоритм сортування називають стабільним? Який алгоритм сортування називають адаптивним? Які показники у більшості випадків використовують для оцінки часової асимптотичної складності алгоритмів сортування? Вкажіть прямі й поліпшені методи

сортування. Які їхні часові складності? У яких методах сортування використовується тільки один цикл і чи впливає це на їхню часову складність?

5. Для розв'язування яких задач можна використовувати алгоритми роботи з масивами?

6. Від якого показника залежать часова і ємнісна складності алгоритмів роботи з масивами? Як визначити адресу певного елемента масиву? Чи сповільнюється виконання програми при роботі з масивами?

Практичне завдання № 13
«Алгоритми обробки одновимірних масивів»

Мета виконання завдання: навчитися розробляти алгоритми обробки одновимірних масивів, а також створювати програми модульної структури, в яких використовуються ці алгоритми.

Завдання. Розробити алгоритм обробки цілочисельного масиву відповідно до завдання варіанту (всю обробку виконувати в тому самому масиві — робочі масиви не використовувати, введення і виведення інформації не здійснювати) і подати його одним чи двома модулями. Також розробити алгоритми: введення в діалоговому режимі елементів масиву, заповнення масиву випадковими числами, виведення масиву (для виведення початкового та обробленого масивів). У керуючому модулі в діалоговому режимі задати реальну кількість елементів масиву (максимальну кількість елементів масиву визначити, наприклад, директивою #define), передбачити вибір у діалоговому режимі способу заповнення масиву (вручну чи випадковими числами) і звернення до відповідного модуля, звернення до модуля виведення масиву для виведення вхідного масиву, звернення до модуля обробки масиву, виведення знайденого мінімального чи максимального елемента і його номера, звернення до модуля виведення (того самого, що й для виведення початкового масиву) для виведення масиву після обробки; передбачити можливість багаторазового виконання алгоритму.

Алгоритми подати програмно. Обмін даними здійснювати через параметри без використання глобальних змінних. Текст програми структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з трьо-

ма різними наборами вхідних даних, при цьому одним із масивів (чи половини масиву) має бути 5-елементний масив, для якого подати протокол покрокового знаходження мінімального / максимального елемента.

Вказівки. Використати алгоритми, які стосуються введення й виведення елементів одновимірних масивів і генерації випадкових чисел (див. п. 8.1), основні типові алгоритми роботи з одновимірними масивами (див. п. 8.2), зокрема, з прикладів 8.2.1 (знаходження мінімального / максимального елемента), 8.2.2 (перестановка елементів), 8.2.3 (перестановка елементів масиву у зворотному порядку), 8.2.5 (збирання на початку чи в кінці масиву елементів з певними властивостями), а також деякі алгоритми з пункту 6.2.

Варіанти завдань

1. Якщо мінімальний елемент масиву кратний 5 і має непарний номер, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо вони мають різні знаки.

2. Якщо максимальний елемент другої половини масиву має непарний номер і є непарним числом, то виконати таку обробку масиву: елементи, значення яких відмінні від 0, зібрати на початку масиву, зберігши їхній порядок, а нулі розмістити в кінці масиву.

3. Якщо мінімальний елемент масиву має парний номер і розміщується в другій половині масиву, то виконати таку обробку масиву: елементи з непарними номерами зібрати у кінці масиву, зберігши їхній порядок; решту елементів розмістити на початку масиву, замінивши нулями.

4. Якщо максимальний елемент першої половини масиву має парний номер і є додатним числом, то елементи з парними номерами (a_2, a_4, a_6, ...) розмістити в зворотному порядку (..., a_6, a_4, a_2), а елементи з непарними номерами залишити на попередньому місці.

5. Якщо мінімальний елемент масиву має парний номер і є додатним числом, то виконати таку обробку масиву: елементи з парними номерами зібрати на початку масиву, зберігши їхній порядок; решту елементів розмістити у кінці масиву, замінивши нулями.

6. Якщо максимальний елемент масиву має непарний номер і розміщується в першій половині масиву, то елементи першої його половини розмістити в зворотному порядку.

7. Якщо мінімальний елемент масиву має непарний номер і кратний 3, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо жоден із пари не дорівнює нулю.

8. Якщо максимальний елемент другої половини масиву кратний 4 і має парний номер, то виконати таку обробку масиву: елементи, значення яких кратні 3, зібрати на початок масиву, зберігши їхній порядок; решту елементів розмістити в кінці масиву, замінивши нулями.

9. Якщо мінімальний елемент масиву має парний номер і є від'ємним числом, то виконати таку обробку масиву: елементи з непарними значеннями зібрати в кінці масиву, зберігши їхній порядок; решту елементів розмістити на початку масиву, замінивши нулями.

10. Якщо максимальний елемент першої половини масиву має непарний номер і є парним числом, то виконати таку обробку масиву: елементи з непарними значеннями зібрати в кінці масиву, зберігши їхній порядок; решту елементів розмістити на початок масиву, замінивши нулями.

11. Якщо мінімальний елемент другої половини масиву має непарний номер і є додатним числом, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо їхні абсолютні значення відрізняються менш, ніж у 2 рази.

12. Якщо максимальний елемент масиву має непарний номер і розміщується в першій половині масиву, то виконати таку обробку масиву: елементи другої його половини розмістити в зворотному порядку.

13. Якщо мінімальний елемент масиву кратний 10 і має парний номер, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо вони мають різні знаки.

14. Якщо максимальний елемент першої половини масиву має парний номер і є додатним числом, то виконати таку обробку масиву: переставити місцями першу і другу його половини, зберігши в них порядок елементів.

15. Якщо мінімальний елемент масиву кратний 5 і розміщується в другій половині масиву, то виконати таку обробку масиву: елементи, значення яких відмінні від 0, зібрати на початок масиву, зберігши їхній порядок, а нулі розмістити в кінці масиву.

16. Якщо максимальний елемент першої половини масиву має непарний номер і є від'ємним числом, то виконати таку обробку масиву: впорядкувати за спаданням кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо вони мають однакову парність.

17. Якщо мінімальний елемент масиву має парний номер і є парним числом, то виконати таку обробку масиву: елементи першої його половини розмістити в зворотному порядку.

18. Якщо максимальний елемент першої половини масиву має непарний номер і дорівнює 0, то виконати таку обробку масиву: елементи з парними номерами (a_2, a_4, a_6, ...) розмістити в зворотному порядку (..., a_6, a_4, a_2), а елементи з непарними номерами залишити на попередньому місці.

19. Якщо мінімальний елемент масиву має парний номер і дорівнює 0, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо вони відрізняються рівно в 2 рази.

20. Якщо максимальний елемент масиву має парний номер і кратний 3, то виконати таку обробку масиву: елементи другої половини масиву розмістити в зворотному порядку.

21. Якщо мінімальний елемент другої половини масиву має парний номер і є парним числом, то виконати таку обробку масиву: елементи з непарними номерами (a_1, a_3, a_5, ...) розмістити в зворотному порядку (..., a_5, a_3, a_1), а елементи з парними номерами залишити на попередньому місці.

22. Якщо максимальний елемент першої половини масиву має парний номер і є непарним числом, то виконати таку обробку масиву: переставити місцями кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо їхні абсолютні значення відрізняються більш, ніж у 2 рази.

23. Якщо мінімальний елемент другої половини масиву має парний номер і є від'ємним числом, то виконати таку обробку масиву: елементи з парними

значеннями зібрати на початок масиву, зберігши їхній порядок; решту елементів розмістити в кінці масиву, замінивши нулями.

24. Якщо максимальний елемент масиву має непарний номер і є непарним числом, то виконати таку обробку масиву: елементи з парними значеннями зібрати в кінці масиву, зберігши їхній порядок; решту елементів розмістити на початку масиву, замінивши нулями.

25. Якщо мінімальний елемент першої половини масиву має непарний номер і є парним числом, то переставити місцями першу і другу його половини, зберігши в них порядок елементів.

26. Якщо максимальний елемент другої половини масиву має парний номер і дорівнює 0, то виконати таку обробку масиву: впорядкувати за зростанням кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …).

27. Якщо мінімальний елемент першої половини масиву кратний 10 і має парний номер, то виконати таку обробку масиву: елементи, значення яких кратні 3, зібрати в кінці масиву, зберігши їхній порядок; решту елементів розмістити на початку масиву, замінивши нулями.

28. Якщо максимальний елемент першої половини масиву кратний 7 і має непарний номер, то виконати таку обробку масиву: впорядкувати за зростанням кожні два сусідні елементи (a_1 і a_2, a_3 і a_4, …), якщо вони мають різну парність.

29. Якщо мінімальний елемент масиву має непарний номер і розміщується в другій половині масиву, то елементи першої його половини розмістити в зворотному порядку.

30. Якщо максимальний елемент масиву кратний 3 і має парний номер, то виконати таку обробку масиву: елементи, значення яких відмінні від 0, зібрати на початок масиву, зберігши їхній порядок, а нулі розмістити в кінці масиву.

Практичне завдання № 14
«Алгоритми сортування масивів структур»

Мета виконання завдання: навчитися розробляти і програмувати алгоритми сортування одновимірних масивів (на прикладі масиву структур), а також створювати програми модульної структури, в яких реалізуються ці алгоритми.

Завдання. Розробити алгоритм сортування масиву структур, застосовуючи вказаний у варіанті метод, і подати його двома модулями для сортування відповідно за вказаними текстовим і числовим полями. Реалізувати введення в діалоговому режимі елементів структури і виведення масиву структур. У керуючому модулі визначити масив структур даних про об’єкти, який містить вказані в завданні елементи (поля); дані про кілька перших об’єктів задати при визначенні масиву структур шляхом ініціалізації; дані про наступні кілька об’єктів ввести. При цьому реалізувати таку послідовність дій: у діалоговому режимі задати реальну кількість елементів масиву структур (максимальну кількість визначити директивою #define; перевіряти правильність введення кількості елементів масиву структур), заповнити інформацією кілька структур, вивести початковий масив структур, вибрати в діалоговому режимі поле (одне з двох вказаних у завданні), за яким треба виконати сортування, виконати сортування, вивести масив структур після обробки. Передбачити можливість багаторазового виконання алгоритму для здійснення сортування за іншим полем і для доповнення масиву структур новими елементами.

Алгоритми подати програмно. Обмін даними здійснювати через параметри без використання глобальних змінних. Текст програми структурувати. Коментарі обов’язкові — 17-25 %. Провести розрахунки не менш ніж з трьома різними наборами вхідних даних.

Вказівки. Відповідно до варіанту використати вказаний алгоритм сортування (див. п. 8.4) , як правило, з незначною модифікацією. Як зразок використати алгоритм з прикладу 8.5.2.

Варіанти завдань

1. Елементи структури: вакансія, освіта, оплата, кількість робочих годин на тиждень. Сортувати за елементами: вакансія, кількість робочих годин на тиждень. Застосувати метод бульбашки з індикатором перестановки і запам’ятовуванням позиції останньої перестановки для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину на початку масиву.

2. Елементи структури: викладач, посада, наукове звання (доцент, професор…), вік, кількість наукових праць. Сортувати за елементами: посада, кількість наукових праць. Застосувати обмінне сортування простою вибіркою з пошуком мінімуму для впорядкування елементів масиву за спаданням.

3. Елементи структури: вид приміщення, кількість кімнат, площа, рік введення в експлуатацію. Сортувати за елементами: вид приміщення, площа. Застосувати метод бульбашки з індикатором перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

4. Елементи структури: птах, вид, рід, маса. Сортувати за елементами: птах, маса. Застосувати обмінне сортування простою вибіркою з пошуком максимуму для впорядкування елементів масиву за зростанням.

5. Елементи структури: предмет, викладач, асистент, семестр, кількість годин. Сортувати за елементами: предмет, кількість годин. Застосувати шейкер-сортування для впорядкування елементів масиву за зростанням.

6. Елементи структури: товар, виробник, кількість одиниць товару, вартість одиниці товару. Сортувати за елементами: товар, кількість одиниць товару. Застосувати обмінне сортування простою вибіркою з пошуком максимуму для впорядкування елементів масиву за спаданням.

7. Елементи структури: ім’я, стать, вік, зріст. Сортувати за елементами: стать, зріст. Застосувати сортування простими вставками для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину в кінці масиву.

8. Елементи структури: прізвище й ініціали, місце народження, місце проживання, вік, стаж роботи. Сортувати за елементами: прізвище, стаж роботи. Застосувати метод сортування: метод бульбашки з індикатором перестановки і запам’ятовуванням позиції останньої перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

9. Елементи структури: назва фірми, місто, індекс, вулиця, номер будинку, код ЄДРПОУ (як цілочисельна інформація). Сортувати за елементами: назва фірми, код ЄДРПОУ. Застосувати шейкер-сортування з індикатором перестановки для впорядкування елементів масиву за зростанням.

10. Елементи структури: прізвище й ініціали, ідентифікаційний код (як символьна інформація), стать, вік, кількість років навчання. Сортувати за елементами: прізвище, кількість років навчання. Застосувати обмінне сортування простою вибіркою з пошуком максимуму для впорядкування елементів масиву за зростанням.

11. Елементи структури: прізвище, ім'я, по батькові, курс, група, середній бал. Сортувати за елементами: прізвище, середній бал. Застосувати метод бульбашки з індикатором перестановки для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину в кінці масиву.

12. Елементи структури: назва тістечок, виробник, сорт, кількість в упаковці, ціна за упаковку. Сортувати за елементами: назва тістечок, кількість в упаковці. Застосувати метод сортування простими вставками для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

13. Елементи структури: прізвище, ім'я, по батькові, кількість статей, кількість посібників. Сортувати за елементами: прізвище, кількість посібників. Застосувати метод бульбашки з індикатором перестановки і запам'ятовуванням позиції останньої перестановки для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину в кінці масиву.

14. Елементи структури: назва товару, виробник, місто, кількість одиниць, ціна за одиницю. Сортувати за елементами: назва товару, ціна за одиницю. Застосувати обмінне сортування простою вибіркою з пошуком мінімуму для впорядкування елементів масиву за спаданням.

15. Елементи структури: прізвище, ім'я, по батькові, факультет, вік, стипендія. Сортувати за елементами: прізвище, стипендія. Застосувати метод шейкер-сортування для впорядкування елементів масиву за спаданням.

16. Елементи структури: прізвище й ініціали, світа, професія, вік. Сортувати за елементами: прізвище, вік. Застосувати сортування простими вставками для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

17. Елементи структури: назва книги, автор, видавництво, рік видання, кількість сторінок. Сортувати за елементами: автор, рік видання. Застосувати

метод бульбашки з запам'ятовуванням позиції останньої перестановки й індикатором перестановки для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину на початку масиву.

18. Елементи структури: предмет, лектор, викладач на практичних заняттях, кількість лекцій, кількість практичних. Сортувати за елементами: викладач на практичних заняттях, кількість практичних. Застосувати метод шейкер-сортування для впорядкування елементів масиву за зростанням.

19. Елементи структури: викладач, посада, наукове звання (доцент, професор, …), вік, педагогічний стаж. Сортувати за елементами: наукове звання, педагогічний стаж. Застосувати обмінне сортування простою вибіркою з пошуком мінімуму для впорядкування елементів масиву за спаданням.

20. Елементи структури: назва тістечок, виробник, дата виготовлення, ціна одиниці. Сортувати за елементами: назва тістечок, ціна одиниці. Застосувати метод бульбашки з індикатором перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

21. Елементи структури: вид транспорту, прізвище й ініціали водія, номер маршруту, кількість місць. Сортувати за елементами: вид транспорту, кількість місць. Застосувати метод сортування простими вставками для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину в кінці масиву.

22. Елементи структури: назва цукерок, виробник, маса упаковки, сорт, ціна за кілограм. Сортувати за елементами: назва цукерок, маса упаковки. Застосувати метод бульбашки з запам'ятовуванням позиції останньої перестановки й індикатором перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину на початку масиву.

23. Елементи структури: вид тканини, відтінок основного кольору (яскраво-червоний, жовтогарячий, …), ширина, кількість метрів, ціна за метр. Сортувати за елементами: вид тканини, ціна за метр. Застосувати шейкер-сортування з індикатором перестановки для впорядкування елементів масиву за зростанням.

24. Елементи структури: викладач, посада, науковий ступінь, вік, педагогічний стаж. Сортувати за елементами: викладач, педагогічний стаж. Застосува-

ти метод бульбашки з індикатором перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину в кінці масиву.

25. Елементи структури: рік, пора року, характеристика погоди (одним чи кількома словами), кількість опадів за сезон, середня температура. Сортувати за елементами: пора року, середня температура. Застосувати метод сортування простими вставками для впорядкування елементів масиву за спаданням, розміщую чи вихідну множину в кінці масиву.

26. Елементи структури: назва підручника, автор, видавництво, рік видання, кількість сторінок. Сортувати за елементами: назва підручника, рік видання. Застосувати обмінне сортування простою вибіркою з пошуком максимуму для впорядкування елементів масиву за спаданням.

27. Елементи структури: предмет, лектор, асистент, кількість лекцій, кількість практичних. Сортувати за елементами: предмет, кількість лекцій. Застосувати метод бульбашки з індикатором перестановки і запам'ятовуванням позиції останньої перестановки для впорядкування елементів масиву за зростанням, розміщуючи вихідну множину в кінці масиву.

28. Елементи структури: рослина, висота, тип квітки, колір квітки, кількість квіток у суцвітті. Сортувати за елементами: рослина, кількість квіток у суцвітті. Застосувати шейкер-сортування з індикатором перестановки для впорядкування елементів масиву за спаданням.

29. Елементи структури: прізвище, ім'я, курс, середній бал. Сортувати за елементами: прізвище, курс. Застосувати обмінне сортування простою вибіркою з пошуком мінімуму для впорядкування елементів масиву за зростанням.

30. Елементи структури: країна, столиця, площа, кількість населення. Сортувати за елементами: країна, кількість населення. Застосувати метод бульбашки з індикатором перестановки для впорядкування елементів масиву за спаданням, розміщуючи вихідну множину на початку масиву.

9. Обробка двовимірних масивів

Поняття двовимірного масиву. Основні алгоритми роботи з двовимірними масивами. Сортування елементів матриці і рядків символів. Розріджені матриці. Приклади алгоритмів з використанням двовимірних масивів. Складність і оптимізація алгоритмів обробки двовимірних масивів.

9.1. Поняття двовимірного масиву

Багатовимірний масив — це масив із масивів. Наприклад, двовимірний масив для компілятора — це масив одновимірних масивів з однаковою кількістю елементів. Для користувача — це матриця, яка є набором вектор-рядків.

Опис двовимірного масиву в мові С має вигляд:

тип ім'я_змінної[розмір1] [розмір2];

Тут *розмір1* вказує кількість рядків у матриці, а *розмір2* — кількість елементів у рядку (тобто кількість стовпчиків у матриці).

Опис багатовимірного масиву має вигляд:

тип ім'я_змінної[розмір1] [розмір2]...[розмірM];

Індексація кожного розміру починається з нуля. Елементи багатовимірного масиву розташовуються в пам'яті в порядку зростання найправішого індексу. Тому правий індекс буде змінюватися швидше, ніж лівий. Наприклад, якщо описано масив int a[2][3], то в пам'яті його елементи будуть розміщуватися так:

a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2],

тобто елементи відповідної матриці розміщуються по рядках:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{pmatrix}$$

При зверненні до багатовимірних масивів комп'ютер багато часу витрачає на обчислення адрес, оскільки при цьому доводиться враховувати значення кожного індексу. Тому доступ до елементів багатовимірного масиву відбувається трохи повільніше, ніж до елементів одновимірного.

Масиви, які мають більше 3 вимірів, використовують дуже рідко, оскільки задачі, в яких вони потрібні, трапляються не часто.

Можливості роботи з двовимірними масивами. 1) Повний розмір двовимірного масиву (обсяг пам'яті комп'ютера, яку займає масив) в байтах обчислюється так:

sizeof(базовий тип)×розмір1×розмір2.

Наприклад, для двовимірного масиву int a[2][3] обсяг пам'яті обчислюється за допомогою виразу: sizeof(int)×2×3 і дорівнює 24 байти. У програмі обсяг пам'яті, яку займає масив, можна визначити за допомогою операції `sizeof(a)`; кількість елементів можна обчислити `k=sizeof(a)/ sizeof(a[0][0]);`

Відповідно для багатовимірних масивів загальний обсяг пам'яті в байтах обчислюється за формулою:

sizeof(базовий тип)×розмір1×розмір2×...×розмірM.

2) Доступ до елементів масиву прямий. Всі його елементи однаково доступні. Доступ до окремого елемента масиву здійснюється за допомогою імені масиву й індексів. Індекси елемента масиву подаються кожен у своїх квадратних дужках після імені. Індексом елемента може бути будь-який вираз цілого типу int, типу char чи типу enum. У всіх масивів індексація елементів починається з індекса 0.

Двомірні масиви розміщуються в матриці, яка складається з рядків і стовпчиків. Перший індекс вказує номер рядка, а другий — номер стовпчика. У пам'яті комп'ютера рядок матриці розміщується за рядком. Якщо до елементів масиву звертаються в тому порядку, в якому вони розміщені в пам'яті, правий індекс змінюється швидше, ніж лівий.

Доступ до окремих елементів масиву повільніший, ніж до окремих змінних, оскільки кожна змінна має свою адресу, а при доступові до конкретного елемента масиву кожен раз обчислюється його адреса, оскільки відома лише адреса початку масиву, тобто першого елемента масиву.

3) Як і елементи одновимірних масивів, елементи багатовимірних масивів можуть використовуватися в будь-якому контексті, де може використовуватися змінна відповідного типу. Можна записати оператори: `s=a[i][j]*a[j][k]; printf("%d", a[1][0]); scanf("%d", &a[0][2]); printf("%d", a[2*i-1][j-1]);` тощо.

4) Ініціалізація масиву. При оголошенні масиву його елементам можна присвоїти потрібні початкові значення-константи (тобто ініціалізувати масив), тип яких повинен бути сумісним з типом масиву:

```
int x[2][3] = {1, 2, 3, 4, 5, 6};
```

Багатовимірні масиви можна ініціалізувати, розглядаючи їх як масиви масивів — для цього елементи ініціалізації кожного виміру можна взяти в фігурні дужки (застосувати *групування*):

```
int x[2][3] = {{1, 2, 3}, {4, 5, 6}};   // двовимірний масив: містить 2 вектори
int xyz[2][3][4] = { // тривимірний масив: містить 2 блоки, кожен з яких матриця 3×4
  { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} },
  { {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24} }
};
```

Якщо при цьому вказати значення не всіх елементів масиву, а тільки кількох перших (хоч би одного першого), то всі решта ініціалізуються нулем. Також при такому записові, якщо всередині групи недостатньо констант ініціалізації, то решта елементів групи автоматично заповнюються нулями. Наприклад:

```
int x[2][3] = {1,2}; // еквівалентно int x[2][3] = {{1,2,0},{0,0,0}};
int x[2][3] = {{1},{2}}; // еквівалентно int x[2][3] = {{1,0,0},{2,0,0}};
#define N 5
int a[N][N]={0}; // заповнення матриці нулями, при цьому N може бути будь-яким
```

Для багатовимірних масивів, як і для одновимірних, теж підтримується ініціалізація безрозмірних масивів. При оголошенні багатовимірного масиву можна не вказувати його розмір тільки в найлівіших дужках (розміри за всіма решта вимірами вказувати обов'язково — це потрібно компілятору для визначення довжин підмасивів, які утворюють масив), але при цьому обов'язково потрібно задати початкові значення. Компілятор визначає розмір масиву автоматично:

```
int x[][3] = { {1, 2, 3}, {4, 5, 6}};   // автоматичне визначення розміру масиву — 2
int XYZ[][3][4] = {    // пропущений розмір масиву — 2
  { {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
  },
  { {13, 14, 15, 16},
    {17, 18, 19, 20},
    {21, 22, 23, 24}
  }
};
```

Використання безрозмірних масивів дає можливість створювати таблиці змінної довжини, не турбуючись про розмір масиву.

Ініціалізувати масив можна в циклі, наприклад, значенням 0:

```
int b[N][M];
for (i=0; i<N; i++)
  for (j=0; j<M; j++) b[i][j]=0;
```

чи іншими значеннями, наприклад, порядковим номером елемента в пам'яті:

```
int c[N][M];
for(i=0; i<N; i++)
  for(j=0; j<M; j++) c[i][j]=i*M+j+1;
```

Також багатовимірний масив можна ініціалізувати функцією memset(b,0,sizeof(b)).

Для заповнення масиву при тестуванні програм можна скористатися генератором випадкових (псевдовипадкових) чисел

Приклад 9.1.1. Для заповнення двовимірного масиву випадковими числами можна використати такий фрагмент програми мовою С:

```
#include <stdlib.h>    // функції srand, rand
#include <time.h>   // функція time
...
  /* З проміжка -100...50 */
  srand(time(0));
  for (i=0; i<N; i++)
    for (j=0; j<M; j++) b[i][j]=rand()%151-100;
```

5) Введення й виведення масивів здійснюється поелементно. При введенні елементів з клавіатури значення можна набирати через пробіли, Enter чи Tab. Введення двовимірного масиву по рядках здійснюється так:

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++) scanf("%d", &b[i][j]);
```

Для виведення двовимірного масиву у вигляді матриці треба у форматі вказувати модифікатор-специфікацію мінімальної ширини поля (наприклад, %5d, щоб вирівняти стовпчики) і треба переходити на новий рядок після виведення кожного рядка матриці:

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++)
    printf("%5d", b[i][j]);
  printf("\n");   // перехід на новий рядок
}
```

Виведення тривимірного масиву на екран можна виконати, виводячи після кожного двовимірного масиву порожній рядок:

```
int xyz[][3][4] = {
{ {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} }, // 1-й
{ {13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24} } };   // 2-й
for (i=0; i<2; i++) {
  for (j=0; j<3; j++) {
    for (k=0; k<4; k++)
      printf("%4d", xyz[i][j][k]);
    printf("\n");   // перехід на новий рядок
  }
```

```
  printf("\n");   // порожній рядок
}
```

6) Копіювання й порівняння масивів здійснюється поелементно:

```
// Копіювання масивів
for (i=0; i<N; i++)
  for (j=0; j<M; j++) a[i][j]=b[i][j];
// Порівняння масивів
ind=1;   // вважаємо, що масиви рівні
for (i=0; i<N && ind; i++)
  for (j=0; j<M && ind; j++)
    if (a[i][j]!=b[i][j]) ind=0;
if (ind) printf("Масиви рівні");
  else printf("Масиви не рівні");
```

7) Двовимірні масиви і вказівники. Індексувати можна не тільки вказівники на одновимірні масиви, а й на багатовимірні. Наприклад, якщо є масив `int mas[2][5]`, то еквівалентні два вирази `mas[0]` і `&mas[0][0]`, які є адресами масиву — вказівниками на перший елемент масиву. Звернутися до певного елемента масиву можна або, вказавши ім'я масиву і в квадратних дужках індекси елементів, або за допомогою вказівника, наприклад `mas[1][3]` або `*((int*)mas+8)`. У загальному випадку для двовимірного масиву звернення до елемента `mas[i][j]` еквівалентне використанню вказівника:

**((базовий_тип*)mas+(i*довжина_рядка)+j.*

8) Передача багатовимірних параметрів-масивів. Передаючи багатовимірний масив у функцію, в оголошенні параметрів функції обов'язково треба вказати всі розміри вимірів, крім найлівішого. Ці розміри потрібні компілятору для того, щоб всередині функції правильно обчислити адресу елемента масиву. Наприклад, якщо масив оголошено

```
int xyz [2] [3] [4];
```

то формальний параметр функції, в яку передається цей масив, треба описати так:

```
void fm(int xyz[][3][4])
```

При цьому можна вказати розмір першого виміру, але не обов'язково.

У функцію передається тільки вказівник на початковий елемент масиву (тобто на елемент з нульовими індексами).

9.2. Основні алгоритми роботи з двовимірними масивами

Розглянемо алгоритми роботи з двовимірними масивами (матрицями), які стосуються знаходження мінімального й максимального елемента, перестановки рядків і стовпчиків матриць, виконання певних дій у певному фрагменті матриці, перегляду елементів матриці в певному порядку.

Для зручності розробки алгоритмів роботи з двовимірними масивами залежно від розв'язуваної задачі бажано записати прямокутну чи квадратну матрицю індексів для випадків парної й непарної кількості рядків чи стовпчиків. Наприклад:

$$\begin{pmatrix} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \end{pmatrix}_{3\times4} \quad \begin{pmatrix} 00 & 01 & 02 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \\ 30 & 31 & 32 \end{pmatrix}_{4\times3} \quad \begin{pmatrix} 00 & 01 & 02 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \end{pmatrix}_{3\times3}$$

Нехай двовимірний масив оголошено так:

```
#define N 6
#define M 5
  int a[N][M];   // масив, який має N×M елементів
  int i, j;   // індекси поточного елемента
```

Приклад 9.2.1. ***Знаходження мінімального елемента*** масиву.

Програмна реалізація алгоритму:

```
/* Пошук першого найменшого елемента масиву
   при перегляді масиву по рядках */
...
  int mi, mj;   // індекси мінімального елемента
  int amin;   // мінімальне значення
  amin=a[0][0]; mi=0; mj=0;
  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      if (a[i][j]<amin) {
        amin=a[i][j]; mi=i; mj=j;
      };
  printf("Мінімальний елемент a[%d][%d]=%d", mi, mj, amin);
...
```

Треба звернути увагу, що на відміну від алгоритму пошуку мінімального елемента в одновимірному масиві, де, щоб не порівнювати перший елемент сам з собою, порівняння починалися з другого елемента масиву (індекс i=1), в алгоритмі пошуку мінімального елемента в двовимірному масиві цього зробити неможливо, оскільки цикл по i відповідає номеру рядка матриці, а цикл по j — номеру стовпчика; тому, якщо почати цикли з i=1 і j=1, то в матриці буде пропущено перші рядок і стовпчик.

Нижче розглянемо фрагменти програмної реалізації мовою C різноманітних дій з матрицями, їхніми рядками й стовпчиками, діагоналями.

Приклад 9.2.2. ***Перестановка місцями*** елементів

а) рядків *i* й *k*:

```
  int r;    // робоча змінна
```

```
  for (j=0; j<M; j++) {
    r=a[i][j];
    a[i][j]=a[k][j];
    a[k][j]=r;
  }
```

б) стовпчиків *j* й *k*:

```
  for (i=0; i<N; i++) {
    r=a[i][j];
    a[i][j]=a[i][k];
    a[i][k]=r;
}
```

Приклад 9.2.3. ***Знаходження суми*** елементів

а) усього масиву:

```
...
int s=0;    // сума елементів
...
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    s+=a[i][j];
```

б) рядка з номером *i*:

```
for (j=0; j<M; j++)
  s+=a[i][j];
```

в) стовпчика з номером *j*:

```
for (i=0; i<N; i++)
  s+=a[i][j];
```

г) головної діагоналі квадратної матриці (при M=N) :

```
for (i=0; i<N; i++)
  s+=a[i][i];
```

$$\begin{pmatrix} \mathbf{00} & 01 & 02 \\ 10 & \mathbf{11} & 12 \\ 20 & 21 & \mathbf{22} \end{pmatrix}$$

ґ) побічної діагоналі квадратної матриці (при M=N) :

```
for (i=0; i<N; i++)
  s+=a[i][N-i-1];
```

$$\begin{pmatrix} 00 & 01 & \mathbf{02} \\ 10 & \mathbf{11} & 12 \\ \mathbf{20} & 21 & 22 \end{pmatrix}$$

д) кожного непарного рядка (1, 3, 5..., які мають індекси 0, 2, 4...) — працює однаково для парної й непарної кількості рядків:

```
for (i=0; i<N; i+=2) {
  s=0;
  for (j=0; j<M; j++)
    s+=a[i][j];
  printf("рядок %d: сума= %d\n",i+1,s);
}
```

$$\begin{pmatrix} \mathbf{00} & \mathbf{01} & \mathbf{02} \\ 10 & 11 & 12 \\ \mathbf{20} & \mathbf{21} & \mathbf{22} \\ 30 & 31 & 32 \\ \mathbf{40} & \mathbf{41} & \mathbf{42} \end{pmatrix}$$

е) кожного парного рядка (2, 4, 6..., які мають індекси 1, 3, 5...) — порівняно з попереднім алгоритмом внесено зміну — в заголовкові зовнішнього циклу було i=0, замінено на i=1; працює однаково для парної й непарної кількості рядків:

```
for (i=1; i<N; i+=2) {
  s=0;
  for (j=0; j<M; j++)
    s+=a[i][j];
  printf("рядок %d: сума= %d\n",i+1,s);
}
```

$$\begin{pmatrix} 00 & 01 & 02 \\ \mathbf{10} & \mathbf{11} & \mathbf{12} \\ 20 & 21 & 22 \\ \mathbf{30} & \mathbf{31} & \mathbf{32} \\ 40 & 41 & 42 \end{pmatrix}$$

є) кожного непарного стовпчика (1, 3, 5..., які мають індекси 0, 2, 4...) — працює однаково для парної й непарної кількості рядків:

```
for (j=0; j<M; j+=2) {   // для парних стовпчиків – j=1;
  s=0;
  for (i=0; i<N; i++)
    s+=a[i][j];
  printf("стовпчик %d: сума= %d\n",j+1,s);
}
```

$$\begin{pmatrix} \mathbf{00} & 01 & \mathbf{02} & 03 & \mathbf{04} \\ \mathbf{10} & 11 & \mathbf{12} & 13 & \mathbf{14} \\ \mathbf{20} & 21 & \mathbf{22} & 23 & \mathbf{24} \end{pmatrix}$$

Приклад 9.2.4. Знаходження в матриці

а) ***рядків***, ***елементи яких впорядковані*** за зростанням значень:

```
...
#define N 3
#define M 4
  int ind;   // чи впорядковані елементи рядка за зростанням
...
  for (i=0; i<N; i++) {
    ind=1;   // вважаємо, що впорядковані
```

```
    for (j=0; j<M-1; j++)
      if (a[i][j]>a[i][j+1])
        ind=0;   // виявлено невпорядкованість
    if (ind) {
      printf("\nРядок %d:  ",i);
      for (j=0; j<M; j++)
        printf ("%d  ",a[i][j]);
    }
  }
...
```

```
Початкова матриця:
    0    1    9    8
    2    5    6    8
    1    8    9    9

Рядок 1:  2  5  6  8
Рядок 2:  1  8  9  9
```

б) ***стовпчиків, елементи яких впорядковані*** за зростанням значень:

```
...
#define N 3
#define M 4
  int ind;   // чи впорядковані елементи стовпчика
...
  for (j=0; j<M; j++) {
    ind=1;   // вважаємо, що впорядковані
    for (i=0; i<N-1; i++)
      if (a[i][j]>a[i+1][j])
        ind=0;   // виявлено невпорядкованість
    if (ind) {
      printf("\nСтовпчик %d:  ",j);
      for (i=0; i<N; i++)
        printf ("%d  ",a[i][j]);
    }
  }
...
```

```
Початкова матриця:
    0    1    9    8
    2    5    6    8
    1    8    9    9

Стовпчик 1:  1  5  8
Стовпчик 3:  8  8  9
```

Приклад 9.2.5. ***Підняття вгору рядків*** матриці

а) з непарними номерами, обнулення решти нижніх рядків:

```
...
#define N 5
#define M 3
```

```
...
  /* Підняття вгору всіх рядків з непарними номерами;
     перший рядок (з індексом 0) залишається на місці */
  for (i=1; i<(N+1)/2; i++) {
  // (1) 2 (3) 4 (5) при цьому (N+1)/2=6/2=3
  // (1) 2 (3) 4 (5) 6 при цьому (N+1)/2=7/2=3
    i2=i*2;
    for (j=0; j<M; j++)
      a[i][j]=a[i2][j];
  }
  /* Обнулення решти рядків */
  for (i=(N+1)/2; i<N; i++)
    for (j=0; j<M; j++)
      a[i][j]=0;
...
```

```
Початкова матриця:
    3    1    9
    8    2   15
    6    8    1
    8    9    0
    7    2    9

Змінена матриця:
    3    1    9
    6    8    1
    7    2    9
    0    0    0
    0    0    0
```

б) з парними номерами, обнулення решти нижніх рядків:

```
...
#define N 5
#define M 3
...
  /* Підняття вгору всіх рядків з парними номерами */
  for (i=0; i<N/2; i++) {
  // 1 (2) 3 (4) 5 при цьому N/2=5/2=2
  // 1 (2) 3 (4) 5 (6) при цьому N/2=6/2=3
    i2=i*2+1;
    for (j=0; j<M; j++)
      a[i][j]=a[i2][j];
  }
  /* Обнулення решти рядків */
  for (i=N/2; i<N; i++)
    for (j=0; j<M; j++)
      a[i][j]=0;
...
```

```
Початкова матриця:
    3    1    9
    8    2   15
    6    8    1
    8    9    0
    7    2    9

Змінена матриця:
    8    2   15
    8    9    0
    0    0    0
    0    0    0
    0    0    0
```

```
Початкова матриця:
    3    1    9
    8    2   15
    6    8    1
    8    9    0
    7    2    9
    7    8    1

Змінена матриця:
    8    2   15
    8    9    0
    7    8    1
    0    0    0
    0    0    0
    0    0    0
```

Приклад 9.2.6. Обнулення ***рядка і стовпчика***, на перетині яких розміщується найбільший елемент матриці.

Програмна реалізація алгоритму:

```
...
#define N 3
#define M 4
  int amax,im,jm;    // значення й індекси максимального елемента
...
  /* Знаходження максимального елемента */
  amax=a[0][0]; im=0; jm=0;
  for (i=0; i<N; i++)
    for (j=0; j<M-1; j++)
      if (a[i][j]>amax) {
        amax=a[i][j]; im=i; jm=j;
      }
  /* Обнулення рядка im */
  for (j=0; j<M; j++)
    a[im][j]=0;
  /* Обнулення стовпчика jm */
  for (i=0; i<N; i++)
    a[i][jm]=0;
...
```

```
Початкова матриця:
    3    1    9    8
    2   15    6    8
    1    8    9    9

Максимум 15

Змінена матриця:
    3    0    9    8
    0    0    0    0
    1    0    9    9
```

Індекси елементів матриць мають таку властивість, що на діагоналях, паралельних до головної діагоналі, є постійною різниця індексів, а на паралельних до побічної — є постійною сума індексів. Наприклад :

$$\overset{ij}{\begin{pmatrix} \mathbf{00} & 01 & 02 & 03 & \mathbf{04} \\ 10 & \mathbf{11} & 12 & \mathbf{13} & 14 \\ 20 & 21 & \mathbf{22} & 23 & 24 \\ 30 & \mathbf{31} & 32 & \mathbf{33} & 34 \\ \mathbf{40} & 41 & 42 & 43 & \mathbf{44} \end{pmatrix}_{5\times 5}} \quad \overset{i-j}{\begin{pmatrix} \mathbf{0} & -1 & -2 & -3 & -4 \\ 1 & \mathbf{0} & -1 & -2 & -3 \\ 2 & 1 & \mathbf{0} & -1 & -2 \\ 3 & 2 & 1 & \mathbf{0} & -1 \\ 4 & 3 & 2 & 1 & \mathbf{0} \end{pmatrix}} \quad \overset{i+j}{\begin{pmatrix} 0 & 1 & 2 & 3 & \mathbf{4} \\ 1 & 2 & 3 & \mathbf{4} & 5 \\ 2 & 3 & \mathbf{4} & 5 & 6 \\ 3 & \mathbf{4} & 5 & 6 & 7 \\ \mathbf{4} & 5 & 6 & 7 & 8 \end{pmatrix}}$$

Ця властивість вказує алгоритм вибору таких діагоналей.

Приклад 9.2.7. ***Заміна*** в нульовій квадратній матриці ***елементів діагоналей***, на перетині яких розміщується елемент з індексами k і m, пев-

ним значенням (наприклад, числом 5; на перетині діагоналей записується число 1).

Програмна реалізація алгоритму:

```
/* Робота з діагоналями квадратної матриці,
   проведеними через вказаний елемент */
...
#define N 5
  int a[N][N]={0};    // заповнення матриці нулями
  int n,    // робочий розмір матриці
  int k, m,    // індекси вказаного елемента
      kpm, kmm;    // сума і різниця індексів
...
  kpm=k+m; kmm=k-m;
  /* На діагоналях записується число 5 */
  for (i=0; i<n;  i++)
    for (j=0; j<n; j++)
      if (i+j==kpm || i-j==kmm)
        a[i][j]=5;
  a[k][m]=1;    // вказаному елементу присвоюється значення 1
...
```

Для демонстрації роботи алгоритму взято матрицю розміру 5×5, заповнену нулями, і вибрано елементи з індексами: 2 і 2, 3 і 1, 1 і 4.

```
Оброблена матриця:
   5   0   0   0   5
   0   5   0   5   0
   0   0   1   0   0
   0   5   0   5   0
   5   0   0   0   5
```

```
Оброблена матриця:
   0   0   0   0   5
   0   0   0   5   0
   5   0   5   0   0
   0   1   0   0   0
   5   0   5   0   0
```

```
Оброблена матриця:
   0   0   0   5   0
   0   0   0   0   1
   0   0   0   5   0
   0   0   5   0   0
   0   5   0   0   0
```

При обробці матриць розглядають їхні верхні й нижні половини, які у випадку непарної (при цьому є серединний рядок, який не належить жодній половині) й парної кількості рядків визначаються номерами рядків так:

$$\left.\begin{matrix}1\\2\end{matrix}\right\}\text{верхня половина}\qquad\qquad \left.\begin{matrix}1\\2\end{matrix}\right\}\text{верхня половина}$$

$$3$$

$$\left.\begin{matrix}4\\5\end{matrix}\right\}\text{нижня половина}\qquad\qquad \left.\begin{matrix}3\\4\end{matrix}\right\}\text{нижня половина}$$

Кількість рядків верхньої половини, незалежно від парності n, обчислюється за тою самою формулою — $n/2$ (наприклад: 5/2=2, 4/2=2). Першим рядком нижньої половини є рядок з номером $(n+1)/2+1$ (наприклад: (5+1)/2+1=4, (4+1)/2+1=3).

Якщо у випадку непарної кількості рядків треба до верхньої половини матриці віднести серединний рядок, то, незалежно від парності n, кількість рядків верхньої половини обчислюється за тою самою формулою $(n+1)/2$ (наприклад: (5+1)/2=3, (4+1)/2=2); першим рядком нижньої половини, як і у випадку без врахування серединного рядка, є рядок з номером $(n+1)/2+1$.

Якщо у випадку непарної кількості рядків треба до нижньої половини матриці віднести серединний рядок, то, незалежно від парності n, кількість рядків верхньої половини обчислюється за формулою $n/2$; першим рядком нижньої половини є рядок з номером $n/2+1$ (наприклад: 5/2+1=3, 4/2+1=3).

У всіх випадках індекси елементів будуть на 1 меншими.

Аналогічно можна розглядати ліву й праву половини матриці у випадку непарної й парної кількості стовпчиків.

Приклад 9.2.8. Заповнення різних фрагментів матриць[21] :

а) обнулення ***верхньої половини матриці*** (якщо матриця квадратна, то виконується алгоритм обходу сектора 9 без елементів серединного рядка при непарних n — див. Додаток А):

```
for (i=0; i<N/2; i++)
  for (j=0; j<M; j++)
    a[i][j]=0;
```

б) обнулення ***нижньої половини матриці*** (якщо матриця квадратна, то виконується алгоритм обходу сектора 10 без елементів серединного рядка при непарних n — див. Додаток А):

```
for (i=(N+1)/2; i<N; i++)
  for (j=0; j<M; j++)
    a[i][j]=0;
```

[21] Повніший перелік алгоритмів обходу секторів квадратної матриці подано в Додатку А. Зараз розглянемо кілька алгоритмів, щоб зрозуміти принципи їхньої роботи.

в) обнулення елементів квадратної матриці ***над побічною діагоналлю***, якщо діагональ не обнуляється (виконується алгоритм обходу сектора 5 без діагональних елементів— див. Додаток А):

```
for (i=0; i<n-1; i++)
  for (j=0; j<n-i-1; j++)
    a[i][j]=0;
```

$$\begin{pmatrix} \mathbf{00} & \mathbf{01} & \mathbf{02} & 03 \\ \mathbf{10} & \mathbf{11} & 12 & 13 \\ \mathbf{20} & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{pmatrix}$$

і якщо діагональ обнуляється (виконується алгоритм обходу сектора 5 з діагональними елементами — див. Додаток А):

```
for (i=0; i<n; i++)
  for (j=0; j<n-i; j++)
    a[i][j]=0;
```

$$\begin{pmatrix} \mathbf{00} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ \mathbf{10} & \mathbf{11} & \mathbf{12} & 13 \\ \mathbf{20} & \mathbf{21} & 22 & 23 \\ \mathbf{30} & 31 & 32 & 33 \end{pmatrix}$$

г) знаходження суми елементів ***наддіагональної частини*** квадратної матриці (верхній трикутник над головною діагоналлю), не включаючи головної діагоналі (виконується алгоритм обходу сектора 7 без діагональних елементів — див. Додаток А):

```
s=0;
for (i=0; i<n-1;  i++)
  for (j=i+1; j<n; j++)
    s+=a[i][j];
```

$$\begin{pmatrix} 10 & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ 10 & 11 & \mathbf{12} & \mathbf{13} \\ 20 & 21 & 22 & \mathbf{23} \\ 30 & 31 & 32 & 33 \end{pmatrix}$$

і включаючи головну діагональ (виконується алгоритм обходу сектора 7 з діагональними елементами — див. Додаток А):

```
s=0;
for (i=0; i<n;  i++)
  for (j=i; j<n; j++)
    s+=a[i][j];
```

$$\begin{pmatrix} \mathbf{00} & \mathbf{01} & \mathbf{02} & \mathbf{03} \\ 10 & \mathbf{11} & \mathbf{12} & \mathbf{13} \\ 20 & 21 & \mathbf{22} & \mathbf{23} \\ 30 & 31 & 32 & \mathbf{33} \end{pmatrix}$$

ґ) обнулення верхньої частини квадратної матриці ***між головною і побічною діагоналями*** без обнулення діагональних елементів (виконується алгоритм обходу сектора 1 без діагональних елементів — див. Додаток А):

```
for (i=0; i<(n-1)/2;  i++)
  for (j=i+1; j<n-i-1; j++)
    a[i][j]=0;
```

д) ***заміна*** в нульовій квадратній матриці елементів ***нижньої частини між діагоналями*** (діагоналі не враховувати / враховувати), на перетині яких розміщується елемент з індексами k і m, певним значенням (наприклад, числом 5; на перетині діагоналей записується число 1).

Програмна реалізація алгоритму:

```
/* Робота з нижньою частиною квадратної матриці між
   діагоналями (діагоналі враховувати / не враховувати),
   проведеними через будь-який елемент */
...
#define N 5
  int a[N][N]={0};    // заповнення матриці нулями
  int jl,jp;    // індекси колонок зліва і справа
  int k,m;    // індекси вказаного елемента
...
  jl=m;jp=m;
  /* Заповнюється п'ятірками частина матриці з діагоналями */
  for (i=k; i<N;  i++) {    // якщо i=k+1, то без діагоналей
    for (j=jl; j<=jp; j++) {
      a[i][j]=5;  }
    if (jl>0) jl--;
    if (jp<N-1) jp++;  printf("\n");
  }
  a[k][m]=1;
...
```

Для демонстрації роботи алгоритму взято матрицю розміру 5×5, заповнену нулями, і вибрано елементи з індексами: 2 і 2, 3 і 1, 1 і 4. Результат заповнення сектора матриці без діагоналей:

```
Оброблена матриця:
   0   0   0   0   0
   0   0   0   0   0
   0   0   1   0   0
   0   0   5   0   0
   0   5   5   5   0
```

```
Оброблена матриця:
   0   0   0   0   0
   0   0   0   0   1
   0   0   0   0   5
   0   0   0   5   5
   0   0   5   5   5
```

```
Оброблена матриця:
   0   0   0   0   0
   0   0   0   0   0
   0   0   0   0   0
   0   1   0   0   0
   0   5   0   0   0
```

і з діагоналями:

```
Оброблена матриця:
   0    0    0    0    0
   0    0    0    0    0
   0    0    1    0    0
   0    5    5    5    0
   5    5    5    5    5
```

```
Оброблена матриця:
   0    0    0    0    0
   0    0    0    0    1
   0    0    0    5    5
   0    0    5    5    5
   0    5    5    5    5
```

```
Оброблена матриця:
   0    0    0    0    0
   0    0    0    0    0
   0    0    0    0    0
   0    1    0    0    0
   5    5    5    0    0
```

Приклад 9.2.9. ***Обхід елементів матриці в певному порядку***

а) ***ланцюжком*** (змійкою) ***по рядках***: елементи першого рядка переглядаються зліва направо, другого рядка — справа наліво, третього — знову зліва направо і т. д. Обхід виконується в квадратній матриці розміру $n \times n$ (алгоритм не ускладниться, якщо матриця буде прямокутною $n \times m$).

Програмна реалізація алгоритму:

```
/* Нумерація елементів матриці змійкою по рядках */
...
#define N 7
void vyvid(int a[N][N], int n);
void zmiika(int a[N][N], int n);
int main() {
  int a[N][N]={0};    // заповнення матриці нулями
  int n;    // заданий розмір квадратної матриці
  system("chcp 1251 & cls");
  printf("Нумерація елементів масиву змійкою\n");
  printf("Задайте розмір матриці n:  ");
  scanf("%d",&n);
  zmiika(a,n);
  printf("\nРезультат:\n");
  vyvid(a,n);
...
}
/* Побудова витків змійки (спочатку зліва направо,
   потім справа наліво і т.д.) */
void zmiika(int a[N][N], int n) {
  int i, j,    // індекси елементів масиву
         m=1;    // номер елемента при обході
  for (i=0; i<n; i++)     // Побудова витків змійки
```

```
    /* Рядок змійки з непарним індексом */
    if (i&1) for (j=n-1; j>=0; j--) a[i][j]=m++;
    /* Рядок змійки з парним індексом */
    else for (j=0; j<n; j++) a[i][j]=m++;
}
void vyvid(int a[N][N], int n) {
  int i, j;  // індекси елементів масиву
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      printf("%5d",a[i][j]);
    printf("\n");
  }
}
```

```
Нумерація елементів масиву змійкою
Задайте розмір матриці n:  6

Результат:
    1    2    3    4    5    6
   12   11   10    9    8    7
   13   14   15   16   17   18
   24   23   22   21   20   19
   25   26   27   28   29   30
   36   35   34   33   32   31
```

б) ***по спіралі за годинниковою стрілкою***: елементи першого рядка переглядаються і нумеруються зліва направо, потім елементи останнього стовпчика — згори вниз, елементи останнього рядка — справа наліво, першого стовпчика — знизу вгору до другого рядка, потім другий рядок зліва направо до передостаннього елемента і т. д. Якщо n непарне число, то центральному елементу треба присвоїти значення n^2. Якщо n непарне число, то останні два внутрішні рядки будуть заповнюватися аналогічно до попередніх.

Алгоритм мовою С такий:

```
/* Нумерація елементів масиву по спіралі за часовою стрілкою */
void spiral(int a[N][N], int n) {
  int i, j, k, p, m ;  // індекси елементів масиву
  p=n/2;   // кількість витків спіралі
  m=1;   // початкове значення
  for (k=0; k<p; k++) {   // Побудова витків спіралі
    /* Верхній рядок витка (заповнення зліва направо) */
    for (j=k; j<n-k; j++) a[k][j]=m++;
    /* Правий стовпчик витка (заповнення згори вниз)*/
    for (i=k+1; i<n-k; i++) a[i][n-k-1]=m++;
    /* Нижній рядок витка (заповнення справа наліво) */
```

```
      for (j=n-k-2; j>=k; j--) a[n-k-1][j]=m++;
      /* Лівий стовпчик витка (заповнення знизу вгору) */
      for (i=n-k-2; i>k; i--) a[i][k]=m++;
    }
    if (n%2) a[p][p]=n*n;   // заповнення центру при непарному n
}
```

```
Нумерація елементів масиву по спіралі
Задайте розмір матриці n:  1

Результат:
   1
```

```
Нумерація елементів масиву по спіралі
Задайте розмір матриці n:  2

Результат:
   1   2
   4   3
```

```
Нумерація елементів масиву по спіралі
Задайте розмір матриці n:  7

Результат:
   1   2   3   4   5   6   7
  24  25  26  27  28  29   8
  23  40  41  42  43  30   9
  22  39  48  49  44  31  10
  21  38  47  46  45  32  11
  20  37  36  35  34  33  12
  19  18  17  16  15  14  13
```

```
Нумерація елементів масиву по спіралі
Задайте розмір матриці n:  6

Результат:
   1   2   3   4   5   6
  20  21  22  23  24   7
  19  32  33  34  25   8
  18  31  36  35  26   9
  17  30  29  28  27  10
  16  15  14  13  12  11
```

в) ***ланцюжком*** (змійкою) ***паралельно до побічної діагоналі згори вниз***. При виконанні обходу окремо розглядається верхня частина матриці разом з побічною діагоналлю, окремо нижня. Напрямком обходу — згори вниз чи знизу вгору — керує змінна, яка відповідно змінює знак.

```
/* Обхід матриці по діагоналях, паралельних до побічної */
void diahonal(int a[N][N], int n) {
  int d;    // 1 - прохід згори вниз; -1 - прохід знизу вгору
  int i, j, k;  // індекси елементів масиву
  k=1;    // наскрізний номер елемента
  d=1;
  /* Заповнення верхнього трикутника з побічною діагоналлю */
  for (i=0; i<n;  i++) {
    if (d==1)   // згори вниз
      for (j=i; j>=0; j--) a[i-j][j]=k++;
    else //  знизу вгору
      for (j=0; j<=i; j++) a[i-j][j]=k++;
    d=-d;
  }
```

```
  /* Заповнення нижнього трикутника без побічної діагоналі */
  for (i=1; i<n;  i++) {
    if (d==-1)   // знизу вгору
      for (j=i; j<n; j++) a[n-j+i-1][j]=k++;
    else   // згори вниз
      for (j=n-1; j>=i; j--) a[n-j+i-1][j]=k++;
    d=-d;
  }
}
```

```
Нумерація елементів масиву по діагоналях,
паралельних до побічної
Задайте розмір матриці n:  5

Результат:
   1    3    4   10   11
   2    5    9   12   19
   6    8   13   18   20
   7   14   17   21   24
  15   16   22   23   25
```

```
Нумерація елементів масиву по діагоналях,
паралельних до побічної
Задайте розмір матриці n:  6

Результат:
   1    3    4   10   11   21
   2    5    9   12   20   22
   6    8   13   19   23   30
   7   14   18   24   29   31
  15   17   25   28   32   35
  16   26   27   33   34   36
```

Розглянуті алгоритми роботи з двовимірними масивами (матрицями) вказують основні підходи щодо обробки масивів і їхніх фрагментів, вони можуть використовуватися не тільки для «обнулення» чи знаходження «сум» певних елементів, а й при виконанні інших дій з масивами.

9.3. Сортування елементів матриці і рядків символів

Для сортування багатовимірних масивів використовують ті ж самі методи, що і для сортування одновимірних масивів. При цьому багатовимірний масив ніби розгортається в одновимірний з одним наскрізним індексом, а для доступу до конкретних елементів багатовимірного масиву наскрізні індекси треба перерахувати в індекси багатовимірного масиву.

Приклад 9.3.1. Сортування елементів двовимірного масиву, якому відповідає прямокутна матриця $A_{n \times m}$, так, щоб при розгортанні його в одновимірний масив (відповідно до його зберігання в пам'яті комп'ютера — рядок за рядком) елементи були впорядковані за зростанням.

Щоб записати алгоритм сортування елементів всього двовимірного масиву, спочатку згадаємо алгоритм сортування одновимірного масиву, на-

приклад, метод бульбашки для сортування за зростанням (див. приклад 8.4.1; аналогічно можна використати будь-який інший метод).

Для зручності роботи подамо його програмний код:

```
/* Сортування одновимірного масиву методом
   звичайної бульбашки за зростанням */
void sortbul(int *a, int n) {
  int i,   // номер проходу елементів масиву
      k,   // індекс елемента масиву
      r;   // робоча змінна для перестановки елементів
  for (i=0; i<n-1; i++)
    for (k=0; k<n-i-1; k++)
      if (a[k]>a[k+1]) {
        r=a[k];
        a[k]=a[k+1];
        a[k+1]=r;
    }
}
```

Для сортування двовимірного масиву з наскрізним індексом елементів $k=\overline{0,n\cdot m-1}$ треба за цим наскрізним індексом обчислити індекси (i,j) елементів у двовимірному масиві за формулами $i=k/m$, $j=k\%m$ і використовувати їх при сортуванні. Правильність формул можна перевірити на прикладі поданої матриці:

$$\begin{pmatrix} \underset{(0,0)}{0} & \underset{(0,1)}{1} & \underset{(0,2)}{2} \\ \underset{(1,0)}{3} & \underset{(1,1)}{4} & \underset{(1,2)}{5} \end{pmatrix}_{2\times 3} \quad \begin{matrix} n=2 \\ m=3 \end{matrix}$$

Програмна реалізація алгоритму буде такою:

```
/* Сортування двовимірного масиву методом звичайної бульбашки */
void sortbulMatr(int a[N][M], int nm, int m) {
// nm – кількість елементів матриці
// m – кількість елементів у рядку матриці
  int i, k;    // параметри циклів при наскрізному індексі
  int ir, jr;   // індекси (i,j) в матриці для наскрізного індексу k
```

```
  int ir1, jr1; // індекси (i,j) в матриці для наскрізного індексу k+1
  int r;    // робоча змінна для перестановки елементів
  for (i=0; i<nm-1; i++)
    for (k=0; k<nm-i-1; k++) {
      ir=k/m; jr=k%m;
      ir1=(k+1)/m; jr1=(k+1)%m;
      if (a[ir][jr]>a[ir1][jr1]) {
        r=a[ir][jr];
        a[ir][jr]=a[ir1][jr1];
        a[ir1][jr1]=r;
      }
    }
}
```

```
Початкова матриця:
   12    4   18   15    2   13
    9   16    5   17   10    7
    1   11   14    8    3    6

Змінена матриця:
    1    2    3    4    5    6
    7    8    9   10   11   12
   13   14   15   16   17   18
```

```
Початкова матриця:
   12    4   18
   15    2   13
    9   16    5
   17   10    7
    1   11   14
    8    3    6

Змінена матриця:
    1    2    3
    4    5    6
    7    8    9
   10   11   12
   13   14   15
   16   17   18
```

Якщо складність алгоритму сортування одновимірного масиву з кількістю елементів n становить $O(n^2)$, то складність сортування двовимірного масиву розміру $n \times n$ буде дорівнювати $O(n^3)$.

Рядок символів у мові програмування С є одновимірним масивом. Тому масив рядків символів є двовимірним масивом. При програмній реалізації алгоритму сортування мовою С можна з масивом рядків працювати, як з двовимірним масивом символів. Але можна, використовуючи функції мови С, працювати з масивом рядків, як зі звичайним одновимірним масивом елементів. При цьому для порівняння рядків треба використати функцію strcmp, а для присвоєння — strcpy з бібліотеки string.h.

Сортування рядків символів полягає у впорядкуванні їх за алфавітом. При сортуванні рядків треба визначити, який рядок є меншим, а який — більшим. Порівняння рядків здійснюється зліва направо до першого відмінного символу. Той рядок є меншим, у якого поточний порівнюваний символ має менший код. З двох рядків різної довжини, коли один з них збігається з початком іншого, меншим є рядок меншої довжини. Для сортування можна засто-

совувати будь-який з існуючих методів (деякі подано в п. 8.4) залежно від характеристик вхідної інформації.

Приклад 9.3.2. ***Сортування*** масиву ***рядків символів***.

Програмна реалізація алгоритму мовою C з використанням методу бульбашки (див. приклад 8.4.1):

```
/* Сортування масиву рядків символів
   методом бульбашки */
#include <string.h> // функції роботи з рядками
#define N 6
...
  char ms[N][26]={ // масив N рядків символів
    "Іванна", "Іван", "Ольга", "Лариса Ющук",
    "Лариса Кублій", "Іванко"
  };
  char rs[26]; // робочий рядок
...
  /* Сортування */
  for (i=0; i<N-1; i++)
    for (j=0; j<N-i-1; j++)
      if (strcmp(ms[j],ms[j+1])>0) {
        strcpy(rs, ms[j]);
        strcpy(ms[j], ms[j+1]);
        strcpy(ms[j+1], rs);
      }
...
```

```
Початковий масив рядків:
   Іванна
   Іван
   Ольга
   Лариса Ющук
   Лариса Кублій
   Іванко
Відсортований масив рядків:
   Іван
   Іванко
   Іванна
   Лариса Кублій
   Лариса Ющук
   Ольга
```

Оскільки довжина рядка символів не залежить від кількості рядків у масиві і не перевищує деякого числа, то часова асимптотична складність сортування масиву рядків буде тією самою, що й складність застосовуваного алгоритму сортування. Проте при сортуванні масиву рядків символів порівняно з сортуванням масиву чисел час роботи алгоритму буде значно більшим за рахунок виконання посимвольних порівнянь рядків і посимвольного присвоєння рядків.

9.4. Розріджені матриці

Розрідженою називають матрицю, яка переважно складається з нулів і тільки окремі її елементи не нульові. Розріджені матриці виникають, наприклад, при чисельному розв'язуванні диференціальних рівнянь у частинних похідних, при дослідженні коливань у фізиці (коливання пружних конструкцій, коливання в аеродинаміці, коливання в електричних колах, коливання молекул і атомів у фізиці елементарних часток), розрідженою є матриця інцидентності графа, матриця суміжності (наприклад, веб-графа, вузлами якого є веб-сторінки, а ребрами — гіперпосилання) тощо.

Розглянемо, наприклад, таку розріджену матрицю:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 5 & 0 & 0 & 0 & 6 \\ 0 & 2 & 0 & 0 & 0 \end{pmatrix}_{(4\times 5)}$$

Одним зі способів компактного зберігання інформації про розріджену матрицю є її стиснений вигляд — у двовимірному масиві з 3 рядками і n стовпчиками (n — кількість ненульових елементів відповідної розрідженої матриці).

Стиснена матриця має таку структуру:

— у першому рядку зберігаються всі n значень ненульових елементів;

— у другому — індекси їхніх рядків у розрідженій матриці;

— у третьому — індекси стовпчиків у розрідженій матриці.

При цьому треба мати враховувати на увазі, що якщо останній стовпчик початкової розрідженої матриці нульовий, то в подальшому може втратитися інформація про розмір розрідженої матриці.

Крім того, якщо масив має не цілочисельний тип, то інформацію можна зберігати в двох масивах: у першому одновимірному масиві — значення ненульових елементів дійсного типу, у другому двовимірному масиві цілого типу — індекси відповідних елементів.

Для нашого прикладу стиснена матриця має вигляд:

значення	1	3	5	6	2
індекс i	0	1	2	2	3
індекс j	2	3	0	4	1

Як правило, інформацію у стисненій матриці зберігають у впорядкованому вигляді, заносячи елементи відповідно до їхньої появи у розрідженій матриці по рядках. Таким чином, другий рядок стисненої матриці впорядкований за зростанням, а третій — впорядкований за зростанням в межах одного рядка. При цьому кажуть, що стиснена матриця відсортована нормально.

При обробці розрідженої матриці пам'ять виділяється тільки під стиснену матрицю і вся обробка виконується зі стисненою матрицею.

Розріджена матриця виводиться на екран як деяка віртуальна, не існуюча, — на основі інформації стисненої матриці.

При вставці чи вилученні з розрідженої матриці окремих елементів треба у стисненій матриці вилучати чи вставляти елементи, виконуючи кожен раз зсуви стовпчиків стисненої матриці.

Як найпростіший приклад роботи зі стисненими матрицями розглянемо ***виведення розрідженої матриці.*** При виведенні звичайної матриці на екран треба знати значення елементів, кількість рядків і кількість стовпчиків. Вона на екран виводиться так:

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++)
    printf("%3d", b[i][j]);
  printf("\n");
}
```

Якщо стиснена матриця відсортована нормально (якщо ні, то треба відсортувати за другим рядком, а потім за третім), то для виведення розрідженої матриці всю цю інформацію беремо зі стисненої матриці:

— ненульові значення у першому рядку;

— кількість рядків розрідженої матриці — це останній елемент другого рядка a[1][n-1];

— кількість стовпчиків розрідженої матриці — це максимальний елемент третього рядка $\max\limits_{i=\overline{0,n-1}} a[2][i]$.

Для виведення розрідженої матриці треба:

— ввести стиснену матрицю c[3][N]. Задамо її при оголошенні:

```
#define N 5
  int c[3][N]= { {1,3,5,6,2},
                 {0,1,2,2,3},
                 {2,3,0,4,1} };
```

— знайти максимальний елемент другого рядка стисненої матриці maxi:

```
  maxi=c[1][N-1];
```

— знайти максимальний елемент третього рядка стисненої матриці maxj:

```
  maxj=c[2][0];
  for (i=1; i<N; i++)
    if (c[2][i]>maxj) maxj= c[2][i];
```

Вивід розрідженої матриці, яка містить лише нулі, може мати вигляд:

```
  p=0;
  for (i=0; i<N; i++) {
    for (j=0; j<M; j++)
      printf("%3d", p);
    printf("\n");
  }
```

Нехай k — номер стовпчика у стисненій матриці. Тоді вивід матриці можна записати так:

```
k=0;
for (i=0; i<=maxi; i++) {
  for (j=0; j<=maxj; j++) {
    if (c[1][k]=i && c[2][k]=j)
      p=c[0][k++];
    else
      p=0;
    printf("%3d", p);
  }
  printf("\n");
}
```

Враховуючи наведені міркування, одержимо код програми виведення розрідженої матриці за заданою стисненою матрицею:

```
/* Виведення розрідженої матриці */
...
#define N 5
int main() {
  int c[3][N]={ // заповнення стисненої матриці
                {1,3,5,6,2},
                {0,1,2,2,3},
                {2,3,0,4,1} };
  int k,   // номер стовпчика в стисненій матриці
      maxi,   // кількість рядків у розрідженій матриці
      maxj,   // кількість стовпчиків у розрідженій матриці
      p,   // значення для виведення
      i,j;   // параметри циклу
  /* Визначення кількості рядків */
  maxi=c[1][N-1];
  /* Визначення кількості стовпчиків */
  maxj=c[2][0];
  for (i=1; i<N; i++)
    if (c[2][i]>maxj) maxj=c[2][i];
  /* Виведення матриці */
  printf("\nРозріджена матриця:\n");
  k=0;
  for (i=0; i<=maxi; i++) {
    for (j=0; j<=maxj; j++) {
      if (c[1][k]==i && c[2][k]==j)
        p=c[0][k++];
      else
        p=0;
      printf("%3d", p);
    }
    printf("\n");
  }
...
}
```

```
Стиснена матриця:
  1  3  5  6  2
  0  1  2  2  3
  2  3  0  4  1

Розріджена матриця:
  0  0  1  0  0
  0  0  0  3  0
  5  0  0  0  6
  0  2  0  0  0
```

9.5. Приклади алгоритмів з використанням двовимірних масивів

Оскільки інтерпретацією двовимірного масиву є матриця, то в основному використання двовимірних масивів пов'язане з матрицями. Розглянемо приклади транспонування матриць, множення матриць, задачу побудови магічного квадрата, а також використання масиву рядків.

Приклад 9.5.1. ***Транспонування матриць.*** Алгоритм транспонування залежить від того, якою є матриця — квадратною чи прямокутною.

а) Транспонування ***квадратної*** матриці $A_{n\times n}$ можна виконати «на місці» — у тій самій матриці. Для одержання транспонованої квадратної матриці треба переставити симетрично відносно головної діагоналі нижню і верхню її частини. При цьому розглядається тільки верхня трикутна матриця без головної діагоналі (виконується алгоритм обходу сектора 7 без діагональних елементів — див. Додаток А) і виконуються переприсвоєння значень елементів, розміщених на симетричних місцях.

Програмна реалізація алгоритму:

```
/* Транспонування матриці. Виконується шляхом перестановки
   відносно головної діагоналі нижньої і верхньої частин
   матриці симетрично: при цьому розглядається тільки верхня
   трикутна матриця без головної діагоналі */
#define N 5
  int a[N][N];
...
  for (i=0; i<N-1;  i++)
    for (j=i+1; j<N; j++) {
      /* Симетрична перестановка */
      r=a[i][j];
      a[i][j]=a[j][i];
      a[j][i]=r;
    }
...
```

```
Початкова матриця:
    1    2    3    4    5
    6    7    8    9   10
   11   12   13   14   15
   16   17   18   19   20
   21   22   23   24   25

Транспонована матриця:
    1    6   11   16   21
    2    7   12   17   22
    3    8   13   18   23
    4    9   14   19   24
    5   10   15   20   25
```

б) Для транспонування ***прямокутної матриці*** треба створити нову матрицю, оскільки розміри початкової і транспонованої матриць будуть різни-

ми: $B_{m\times n}=(A_{n\times m})^T$. Для всіх елементів матриць $A_{n\times m}$ і $B_{m\times n}$ треба виконати присвоєння $b_{ji}=a_{ij}$.

Програмна реалізація алгоритму має вигляд:

```
/* Транспонування прямокутної матриці */
#define N 5
#define M 3
  int a[N][M], b[M][N];
...
  for (i=0; i<N;  i++)
    for (j=0; j<M; j++)
      b[j][i]=a[i][j];
...
```

```
Початкова матриця:
    1    2    3
    4    5    6
    7    8    9
   10   11   12
   13   14   15

Транспонована матриця:
    1    4    7   10   13
    2    5    8   11   14
    3    6    9   12   15
```

Оскільки при транспонуванні квадратних матриць розміру $n\times n$ виконуються два вкладені цикли, то обчислювальна складність алгоритму транспонування матриць становить $O(n^2)$. В алгоритмі не використовується додаткова пам'ять для зберігання масиву чи його частини, тому ємнісна складність транспонування квадратних матриць — $O(1)$.

При транспонуванні прямокутних матриць розміру $n\times m$ теж виконуються два вкладені цикли, тому складність алгоритму транспонування становить $O(n\times m)$. При цьому, якщо n і m близькі за значеннями, то можна говорити про складність $O(n^2)$. В алгоритмі використовується пам'ять для зберігання транспонованого масиву, тому ємнісна складність транспонування прямокутних матриць — $O(n\times m)$ і аналогічно можна говорити про складність $O(n^2)$.

Приклад 9.5.2. ***Множення прямокутних матриць***. Добутком двох прямокутних матриць $A_{(n\times k)}$ і $B_{(k\times m)}$ є матриця $C_{(n\times m)}$, де $c_{ij}=\sum_{l=1}^{k}a_{il}b_{lj}$.

Програмна реалізація має вигляд:

```
/* Множення двох прямокутних матриць */
#define N 2
```

```
#define K 3
#define M 4
...
  int a[N][K]={1,2,3,4,5,6};
  int b[K][M]={3,1,4,3,5,0,3,0,2,1,4,1};
  int c[N][M];   // результуюча матриця
  /* Добуток матриць */
  for (i=0; i<n;  i++)
    for (j=0; j<m; j++) {
      s=0;
      for (l=0; l<k; l++)
        s+=a[i][l]*b[l][j];
      c[i][j]=s;
    }
...
```

```
Матриця A:
   1    2    3
   4    5    6
Матриця B:
   3    1    4    3
   5    0    3    0
   2    1    4    1
Матриця C:
  19    4   22    6
  49   10   55   18
```

Оскільки при множенні матриць виконуються три вкладені цикли, то складність алгоритму множення матриць становить $O(n^3)$; ємнісна складність — $O(n^2)$.

При розв'язуванні задач лінійної алгебри, наприклад, таких як знаходження рангу матриці чи рангу системи векторів, обчисленні визначника, при розв'язуванні систем лінійних алгебраїчних рівнянь методом Гаусса треба матрицю привести до трапецеїдального (трикутного) вигляду.

Приклад 9.5.3. ***Зведення*** прямокутної матриці ***до трапецеїдального (трикутного) вигляду***.

Алгоритм зведення матриці до трапецеїдального вигляду:

— знайти в матриці перший ненульовий стовпчик і в цьому стовпчику вибрати ненульовий елемент (він є ключовим). Рядок з ключовим елементом (він буде ключовим) переставити на перше місце в матриці, а перший рядок на його місце (якщо ключовий рядок є першим, то перестановку виконувати не треба);

— за допомогою ключового рядка під ключовим елементом у всіх решта рядках одержати нулі, виконуючи такі перетворення:

для II рядка $\vec{a}_2 \cdot a_{11} - \vec{a}_1 \cdot a_{21}$;

для III рядка $\vec{a}_3 \cdot a_{11} - \vec{a}_1 \cdot a_{31}$;

………………

для $\underline{\bar{n}}$ рядка $\vec{a}_n \cdot a_{11} - \vec{a}_1 \cdot a_{n1}$.

У результаті після виконання таких дій решта елементів у цих рядках також зміняться;

— у матриці виділити підматрицю без першого рядка і першого стовпчика і для неї виконати попередні дві дії. Ці дії виконувати доти, доки не буде одержано підматрицю, яка містить один рядок, або у якої всі рядки будуть нульовими.

Програмна реалізація алгоритму:

```
/* Зведення матриці a[n][m] до трапецеїдального вигляду */
...
  int n, m,    // робочий розмір матриці
      ik, jk,    // ключовий елемент
      i,j,    // поточні індекси
      ind,    // індикатор пошуку ненульового стовпчика
      r, ri;    // робочі змінні
...
  ik=0; jk=0;    // індекси ключового елемента
  for (; ik<n-1; ik++) {
    /* Пошук першого ненульового стовпчика */
    ind=1;    // треба шукати ненульовий стовпчик
    while (jk<m && ind) {    // пошук ненульового стовпчика
                             // і першого ненульового елемента
      i=ik;
      while (i<n && ind) {
        if(a[i][jk]) ind=0;    // знайдено ненульовий стовпчик
        i++;
      }
      jk++;
    }
```

```
    if (!ind) {   // якщо знайдено ненульовий стовпчик
      i--; jk--; // індекси ненульового елемента
    }
    /* Перестановка рядків, якщо ненульовий елемент
       не в ключовому рядку */
    if (i!=ik)
      for (j=jk; j<m; j++) {
        r=a[i][j]; a[i][j]=a[ik][j]; a[ik][j]=r;
      }
    /* Зведення матриці до трапецеїдального вигляду */
    r=a[ik][jk];    // ключовий елемент
    for (i=ik+1; i<n; i++) {
      ri=a[i][jk];    // перший елемент поточного рядка
      a[i][jk]=0;    // нуль під ключовим елементом
      for (j=jk+1; j<m; j++)
        a[i][j]=a[i][j]*r-a[ik][j]*ri;
    }
  }
...
```

```
Матриця 4*4:
       4       5       2      -1
       3       6       1       0
       2      -3       1       4
       5       0      -1       4

Трапецеїдальна матриця:
       4       5       2      -1
       0       9      -2       3
       0       0     -44     228
       0       0       0   28512
```

```
Матриця 4*3:
       4       5       2
       3       6       1
       2      -3       1
       5       0      -1

Трапецеїдальна матриця:
       4       5       2
       0       9      -2
       0       0     -44
       0       0       0
```

```
Матриця 3*4:
       4       5       2      -1
       3       6       1       0
       2      -3       1       4

Трапецеїдальна матриця:
       4       5       2      -1
       0       9      -2       3
       0       0     -44     228
```

```
Матриця 4*4:
       4       5       2      -1
       4       5       2       1
       2      -3       1       4
       5       0      -1       4

Трапецеїдальна матриця:
       4       5       2      -1
       0     -22       0      18
       0       0     308     -12
       0       0       0  -54208
```

```
Матриця 3*3:
       1       2       3
       4       5       6
       7       8       9

Трапецеїдальна матриця:
       1       2       3
       0      -3      -6
       0       0       0
```

```
Матриця 3*4:
       3       4      -2       5
      -4       1       1       1
       1       5      -3       2

Трапецеїдальна матриця:
       3       4      -2       5
       0      19      -5      23
       0       0     -78    -234
```

Щоб при зведенні матриць до трапецеїдального (трикутного) вигляду, а також в одержаних трапецеїдальних матрицях числа не були великими, можна кожен рядок скоротити на його найбільший спільний дільник, застосувавши алгоритм Евкліда (див. приклад 8.5.6).

Часова складність алгоритму зведення матриць до трапецеїдального вигляду — $O(n^3)$; ємнісна складність — $O(1)$.

У Китаї ще з 2200 р. до н. е. були відомі такі математичні головоломки, як магічні квадрати. *Магічний квадрат* — це квадратна матриця розміру $n \times n$, заповнена числами так, що сума чисел у кожному рядку, кожному стовпчику і на обох діагоналях однакова (цю суму називають магічною сталою). Квадрат, заповнений цілими числами від 1 до n^2, називають нормальним. Нормальні магічні квадрати існують для всіх порядків $n \geq 1$, за винятком $n = 2$, хоч випадок $n = 1$ тривіальний — квадрат складається з одного числа. Мінімальний нетривіальний квадрат має порядок 3. Для нормального магічного квадрата магічна стала залежить тільки від n і дорівнює $n(n^2+1)/2$.

Для побудови нормальних магічних квадратів при парному й непарному n розроблено окремі алгоритми. Алгоритм для випадку непарного n простіший. Розглянемо його.

Приклад 9.5.4. Побудова ***магічного квадрата непарного порядку.***

Алгоритм побудови нормального магічного квадрата непарного порядку базується на таких положеннях.

Нумерація рядків і стовпчиків квадрата збігається з індексами масиву мови C і набуває значень від 0 до $(n-1)$.

У будь-якому магічному квадраті перше число, тобто 1, зберігається в позиції $(n/2, n-1)$. Позначимо цю позицію (i, j). Наступне число зберігається в позиції $(i-1, j+1)$, де при цьому кожен рядок і стовпчик розглядаються як круговий масив, тобто вважається, що останній і перший елементи масиву розміщені поряд.

При заповненні квадрата мають виконуватися три умови:

1) позиція наступного номера розраховується шляхом зменшення номера рядка попереднього числа на 1 і збільшення номера стовпчика попереднього числа на 1. У будь-який час, якщо розрахована позиція рядка набуває значення –1, то вона буде обертатися до $(n-1)$. Так само, якщо розрахункова позиція стовпчика набуває значення n, вона буде обертатися до 0;

2) якщо магічний квадрат вже містить число у розрахунковій позиції, розрахункова позиція стовпчика зменшується на 2, а розрахована позиція рядка збільшується на 1;

3) якщо розрахована позиція рядка набуває значення –1 і обчислена позиція стовпчика дорівнює n, то новою позицією буде $(0, n-2)$.

Програмна реалізація алгоритму:

```
/* Створення магічних квадратів непарних розмірів. Алгоритм
   починає роботу з елемента з індексами (i,j)=(n/2,n-1) */
...
#define N 5    // N 3
int main() {
  int m[N][N] = {0}; // обнуляється квадрат
  int i, j,   // індекси поточної клітинки квадрата
      k;   // поточне значення числа від 1 до N*N
  i = N/2; j = N-1;
  /* Одне за одним розставляються всі числа в магічний квадрат */
  for (k=1; k <= N*N; ) { // проходження клітинок квадрата
    if (i==-1 && j==N) { // умова 3
      i = 0; j = n-2;
    }
    else {
    // умова 1: наступний номер виходить за праву сторону квадрата
      if (j == N) j = 0;
    // умова 1: наступний номер виходить за верхню сторону квадрата
      if (i < 0) i=N-1;
    }
```

```
    if (m[i][j]) { // умова 2
      i++; j -= 2;
    }
    else { // заповнення клітинки числом k - умова 1
      m[i][j] = k++;
      i--; j++;
    }
  }
  // Виведення магічного квадрата
  printf("Магічний квадрат для n=%d: \
    \nСума кожного рядка, колонки, діагоналей %d:\n\n",
    N, N*(N*N+1)/2);
  for(i=0; i<N; i++) {
    for(j=0; j<N; j++)
      printf("%3d ", m[i][j]);
    printf("\n");
  }
  printf("\n");
  system("pause");
  return 0;
...
}
```

```
Магічний квадрат для n=5:
Сума кожного рядка, колонки, діагоналей 65:

  9   3  22  16  15
  2  21  20  14   8
 25  19  13   7   1
 18  12   6   5  24
 11  10   4  23  17
```

```
Магічний квадрат для n=3:
Сума кожного рядка, колонки, діагоналей 15:

  2   7   6
  9   5   1
  4   3   8
```

Розглянутий алгоритм побудови магічного квадрата має часову складність $O(n^2)$, ємнісну складність — $O(1)$.

Приклад 9.5.5. Визначення за введеним номером дня його назви і його статусу — робочий чи неробочий.

Програмна реалізація з використанням перераховуваного типу даних enum і масиву рядків символів така:

```
/* Визначення дня тижня і його статусу за номером */
...
  int dd;    // введена інформація - номер дня тижня
  enum {Mon,Tue,Wed,Thu,Fri,Sat,Sun};
```

```
  char name[][20]={"понеділок", "вівторок", "середа",
       "четвер", "п'ятниця", "субота", "неділя"};
  printf("Введіть номер дня тижня (понеділок - 1)");
  while (printf("\n "),
    scanf("%d", &dd)) {
    dd--;
    switch (dd) {
      case Sat: case Sun:
        printf("  %s - вихідний день",name[dd]);
        break;
      case Mon: case Tue: case Wed: case Thu: case Fri:
        printf("  %s - робочий день",name[dd]);
        break;
      default:  printf("неправильно вказано номер дня");
    }
  }
...
```

```
Введіть номер дня тижня (понеділок - 1)
 8
неправильно вказано номер дня
 2
  вівторок - робочий день
 7
  неділя - вихідний день
 кінець
```

Як видно з поданого прикладу (а також прикладу 9.3.2), якщо в рядку символів не розглядати окремі символи, а брати весь рядок як одиницю інформації, то робота з масивом рядків виконується так само, як і зі звичайним одновимірним масивом.

9.6. Складність і оптимізація алгоритмів обробки двовимірних масивів

Часова і ємнісна асимптотичні складності алгоритмів обробки масивів залежать від кількості їхніх елементів. Так, якщо масив одновимірний і кількість його елементів дорівнює n, то складність залежить від n. Якщо масив двовимірний, то у відповідність йому можна поставити квадратну матрицю розміру $n \times n$ чи прямокутну розміру $n \times m$. При цьому, якщо матриця прямокутна і обидва її розміри є дуже великими числами, то можна вважати, що вони приблизно однакові, тобто $m \approx n$, і розглядати асимптотичну складність, знову ж таки, як функцію від n.

При роботі з масивами алгоритми, як правило, мають поліноміальну складність. Наприклад: $O(n)$ — дослідити властивості окремого рядка, чи стовпчика, чи діагоналей матриці, поміняти місцям два чи кілька рядків чи стовпчиків матриці; $O(n^2)$ — дослідити всі елементи матриці, обійти їх у певному порядку, транспонувати матрицю, знайти елемент чи блок елементів матриці з певними властивостями, виконати дії над парними чи непарними рядками чи стовпчиками, з фрагментом матриці; $O(n^3)$ — помножити дві матриці, звести матрицю до трапецеїдального вигляду. Якщо сортування одновимірного масиву має складність $O(n^2)$ чи $O(n \log n)$, то двовимірного відповідно — $O(n^3)$ чи $O(n^2 \log n)$. Часова складність $O(1)$ буде тоді, коли всі елементи матриці чи її частини переглядати не треба, а треба вибрати тільки елементи з певними номерами.

Якщо в алгоритмі не використовується додаткова пам'ять для зберігання матриць чи їхніх частин, то такі алгоритми мають ємнісну складність $O(1)$; якщо пам'ять використовується для зберігання певної смуги елементів фіксованого розміру, то ємнісна складність — $O(n)$; якщо додатково зберігається вся матриця чи якийсь її сектор, то — $O(n^2)$.

Для оптимізації обробки двовимірних масивів чи масивів з більшою вимірністю треба застосовувати ті ж самі прийоми, що й при обробці одновимірних масивів (див. п. 8.6).

Так, якщо індексація елементів масиву *a*[*n*][*m*] за обома розмірами починається з нуля, то адреса елемента *a*[*i*][*j*] обчислюється за формулою:

*адреса_масиву+i*m*sizeof(тип_масиву) + j*sizeof(тип_масиву)* ;

якщо ж індексація починається з будь-яких чисел *k* і *l*, — то за формулою:

*адреса_масиву+(i-k)*m*sizeof(тип_масиву) +*
*+ (j-l)*sizeof(тип_масиву)* ;

з чого випливає, що треба надавати перевагу індексації елементів з нуля, а також усувати повторні звернення до елементів, використовуючи в розрахунках звичайні робочі змінні.

Для оптимізації алгоритмів обробки двовимірних масивів, як і будь-яких інших алгоритмів, треба якомога глибше аналізувати постановки задач і виявляти певні особливості.

Запитання та завдання для самоконтролю

1. Що таке багатовимірний масив? Який математичний об'єкт можна поставити у відповідність двовимірному масивові? Як розміщуються в пам'яті комп'ютера елементи двовимірного масиву? Як вибрати окремий елемент двовимірного масиву? Що таке масив рядків символів?

2. Вкажіть типові алгоритми роботи з двовимірними масивами. Оцініть їхню часову і ємнісну асимптотичні складності. У чому полягає відмінність алгоритму пошуку мінімального / максимального елемента у двовимірному масиві від пошуку такого елемента в одновимірному масиві? Чи можна алгоритми обробки матриць побудувати так, щоб не розрізняти випадки парного і непарного значення кількості рядків і стовпчиків матриць? Як в алгоритмах обробки матриць можна використати суми і різниці індексів елементів?

3. Як можна сортувати елементи багатовимірного масиву? Як відсортувати рядки символів?

4. Яку матрицю називають розрідженою? Наведіть приклади задач, при розв'язуванні яких виникають розріджені матриці. Як розріджену матрицю можна зберігати в комп'ютері? Навіщо стискають розріджену матрицю? Опишіть можливу структуру стисненої матриці. Як виділяється пам'ять і виконується обробка розрідженої матриці?

5. Для розв'язування яких прикладних задач можна використовувати алгоритми роботи з двовимірними масивами? Чи є різниця при транспонуванні квадратної і прямокутної матриць і чи відрізняються їхні часові і ємнісні складності?

6. Від якого показника залежать часова і ємнісна складності алгоритмів роботи з двовимірними масивами? Як визначити адресу певного елемента двовимірного масиву? Чи сповільнюється виконання програми при роботі з масивами? Вкажіть шляхи оптимізації алгоритмів обробки багатовимірних масивів.

Практичне завдання № 15

«Алгоритми обробки двовимірних масивів»

Мета виконання завдання: навчитися розробляти алгоритми обробки двовимірних масивів, а також створювати програми модульної структури, в яких використовуються ці алгоритми.

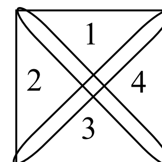
Завдання. Розробити алгоритми обробки двовимірного цілочисельного масиву (квадратної матриці порядку n) відповідно до двох частин завдання, вибираючи частини матриць за поданими схемами. Також розробити алгоритми: введення в діалоговому режимі елементів двовимірного масиву, заповнення масиву випадковими числами, виведення масиву. У керуючому модулі в діалоговому режимі задати реальний порядок матриці n (максимальний порядок визначити, наприклад, директивою #define), передбачити вибір у діалоговому режимі способу заповнення масиву (вручну чи випадковими числами) і звернення до відповідного модуля, звернення до модуля виведення масиву для виведення вхідного масиву (елементи сектора, який відповідає першій частині завдання, виділити, наприклад, символом * — умовою вибирати суму і різницю індексів), звернення до модулів обробки масиву (відповідно до двох частин завдання — застосувати оптимальний перегляд секторів) і вивести одержані результати, звернення до модуля виведення матриці після обробки; передбачити можливість багаторазового виконання алгоритму.

Алгоритми подати програмно. Обмін даними здійснювати через параметри без використання глобальних змінних. Текст програми структурувати. Коментарі обов'язкові — 17-25 %. Провести розрахунки не менш ніж з двома різними наборами вхідних даних для парного і для непарного значень n.

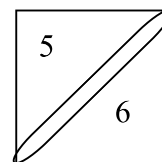
Вказівки. Використати алгоритми, які стосуються введення й виведення елементів двовимірних масивів (див. п. 9.1) і генерації випадкових чисел (див. приклад 9.1.1), основні типові алгоритми роботи з двовимірними масивами (див. п. 9.2), алгоритми обходу секторів матриць (див. Додаток А), а також деякі алгоритми з пункту 6.2. Для елемента (i, j) його номер обчислюється за

формулою $nom = i \cdot n + j + 1$, звідси індекси $i = (nom - 1) / n$, $j = nom - i \cdot n - 1$. У завданнях варіантів використано позначення: $\wedge 4$ — сектор 4 з частинами обох діагоналей; 4\ — з частиною головної діагоналі, /4 — з частиною побічної діагоналі тощо.

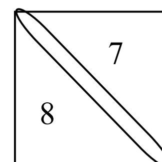
Варіанти завдань



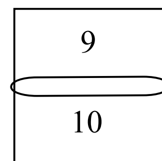
1. У секторі < 4 порахувати кількість елементів масиву, значення яких рівні заданому числу. У секторі \7 порахувати кількість елементів і елементи замінити нулями.



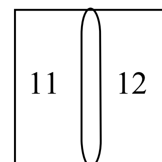
2. У секторі $2 >$ порахувати кількість непарних чисел. У секторі 5/ порахувати кількість елементів і елементи замінити нулями.



3. У секторі $\vee 1$ порахувати кількість елементів з від'ємними значеннями. У секторі 5 порахувати кількість елементів і елементи замінити нулями.



4. У секторі 5 знайти найменший додатний елемент (вказати його номер). У секторі $2 >$ порахувати кількість елементів і елементи замінити нулями.



5. У секторі $\wedge 3$ порахувати кількість елементів, остача від ділення значень яких на задане число дорівнює 3. У секторі 7 порахувати кількість елементів і елементи замінити нулями.

6. У секторі 1 порахувати суму непарних чисел. У секторі 11 (якщо *n* непарне, то з |) порахувати кількість елементів і елементи замінити нулями.

7. У секторі \4 порахувати кількість елементів, значення яких більші від заданого числа. У секторі 10 (якщо *n* непарне, то з —) порахувати кількість елементів і елементи замінити нулями.

8. У секторі $\backslash 7$ знайти елемент (вказати його номер), значення якого найбільш відрізняється від заданого числа. У секторі \1 порахувати кількість елементів і елементи замінити нулями.

9. У секторі $2 >$ порахувати кількість елементів, значення яких рівні нулю. У секторі 6 порахувати кількість елементів і елементи замінити нулями.

10. У секторі 4 порахувати кількість елементів, значеннями яких є парні числа. У секторі $\backslash 7$ порахувати кількість елементів і елементи замінити нулями.

11. У секторі 3 знайти номер першого елемента, значення якого ділиться на задане число без остачі. У секторі $8 \backslash$ порахувати кількість елементів і елементи замінити нулями.

12. У секторі 6 знайти найбільший від’ємний елемент (вказати його номер). У секторі < 3 порахувати кількість елементів і елементи замінити нулями.

13. У секторі 5 порахувати суму елементів, значення яких більші від заданого числа. У секторі 3 порахувати кількість елементів і елементи замінити нулями.

14. У секторі 2/ знайти найменший елемент (вказати його номер). У секторі /6 порахувати кількість елементів і елементи замінити нулями.

15. У секторі /4 порахувати кількість елементів, значеннями яких є парні числа. У секторі 10 (якщо n непарне, то з —) порахувати кількість елементів і елементи замінити нулями.

16. У секторі $\wedge 3$ порахувати кількість елементів, значеннями яких є натуральні числа. У секторі 8 порахувати кількість елементів і елементи замінити нулями.

17. У секторі 1/ порахувати кількість елементів, значення яких більші від заданого числа. У секторі 5/ порахувати кількість елементів і елементи замінити нулями.

18. У секторі 6 порахувати суму значень елементів, які діляться на 3 без остачі. У секторі 4 порахувати кількість елементів і елементи замінити нулями.

19. У секторі $\vee 1$ порахувати кількість елементів з від’ємними значеннями. У секторі 6 порахувати кількість елементів і елементи замінити нулями.

20. У секторі $2 >$ порахувати кількість елементів, значення яких рівні нулю. У секторі 5 порахувати кількість елементів і елементи замінити нулями.

21. У секторі $5/$ знайти суму елементів, значення яких менші від заданого числа. У секторі 1 порахувати кількість елементів і елементи замінити нулями.

22. У секторі \4 порахувати кількість елементів, значеннями яких є непарні числа. У секторі 12 (якщо *n* непарне, то без |) порахувати кількість елементів і елементи замінити нулями.

23. У секторі /3 порахувати кількість елементів, значення яких кратні заданому числу. У секторі 8 порахувати кількість елементів і елементи замінити нулями.

24. У секторі /6 порахувати суму значень елементів, які діляться на 3 без остачі. У секторі 2 порахувати кількість елементів і елементи замінити нулями.

25. У секторі \1 порахувати суму непарних чисел. У секторі 9 (якщо *n* непарне, то без —) порахувати кількість елементів і елементи замінити нулями.

26. У секторі 2\ порахувати кількість елементів, значення яких менші від заданого числа. У секторі 11 (якщо *n* непарне, то з |) порахувати кількість елементів і елементи замінити нулями.

27. У секторі 8\ порахувати кількість чисел, рівних за абсолютною величиною заданому числу. У секторі 2\ порахувати кількість елементів і елементи замінити нулями.

28. У секторі 3\ порахувати кількість елементів, значення яких менші від заданого числа. У секторі 12 (якщо *n* непарне, то без |) порахувати кількість елементів і елементи замінити нулями.

29. У секторі /4 знайти номер першого елемента, значення якого ділиться на задане число без остачі. У секторі 7 порахувати кількість елементів і елементи замінити нулями.

30. У секторі 2 порахувати кількість елементів, значення яких менші від заданого числа. У секторі 10 (якщо *n* непарне, то без —) порахувати кількість елементів і елементи замінити нулями.

10. Алгоритмізація розв'язування задач

Вибір алгоритму розв'язування задачі (на прикладі обчислення кількості сполук). Швидке піднесення чисел до цілих степенів. Генерація випадкових чисел: використання випадкових чисел, генерація рівномірно розподілених випадкових чисел, метод середин квадратів, метод середин добутків, метод перемішування, лінійний конгруентний метод, інші методи генерації випадкових чисел. Генерація бінарних і небінарних кодів Грея. Генерація комбінаторних множин: всіх підмножин множини, перестановок без повторень, перестановок з повтореннями, сполук без повторень, сполук з повтореннями, розміщень без повторень, розміщень з повтореннями. Розкладання натуральних чисел на доданки і прості множники.

10.1. Вибір алгоритму розв'язування задачі (на прикладі обчислення кількості сполук)

Для розв'язання певної задачі, як правило, можна скористатися кількома алгоритмами. Один з них може бути дуже простим, але потребувати великих часових затрат і великого обсягу пам'яті для виконання, інший — не зовсім очевидним, але виконуватися швидко і використовувати незначний обсяг пам'яті при обчисленнях.

Проведемо аналіз використання різних алгоритмів розв'язування тієї самої задачі на прикладі обчислення кількості сполук (біноміальних коефіцієнтів, комбінацій).

Розглянемо приклад розробки алгоритму для оптимального розв'язання задачі, починаючи від розв'язання «напряму́» і до розробки алгоритму, який базується на застосуванні знань з математики і який працює найшвидше.

За означенням кількість сполук обчислюють за формулою:

$$C_n^m = \frac{n!}{m! \cdot (n-m)!}, \text{ де } 0 \le m \le n.$$

Якщо розв'язувати задачу «напряму», тобто реалізовувати подану формулу, то можна написати функцію обчислення факторіалу і тричі до неї звернутися, передаючи як параметри значення n, m і $(n-m)$.

Функція обчислення значення факторіалу числа k матиме, наприклад, такий вигляд:

```
int fact(int k) {
/* Функція знаходження значення факторіалу
  int f;
```

```
  int i;
  f=1;
  for (i=2; i<=k; i++)
    f*=i;
  return f;
}
```

Тоді обчислення значення біноміального коефіцієнта можна реалізувати за допомогою оператора:

```
  c=fact(n)/fact(m)/fact(n-m);
```

Але при такому підході значення факторіалу буде дуже швидко переповнювати пам’ять. Так, оскільки максимальним значенням змінної типу int є 2147483647, то значення $12! = 479001600$ ще буде правильно обчислене, а значення $13! = 6227020800 > 2147483647$ буде вже обчислюватися неправильно і в пам’яті буде подаватися числом 1932053504, значення 17! в пам’яті буде подаватися навіть як від’ємне число -288522240. Таким чином, при обчисленні значення біноміального коефіцієнта, якщо для накопичення значення факторіалу використовувати змінну f типу int, максимально можливим значенням n буде 12.

Якщо тип змінної f буде іншим, то відповідно і для іншого максимально можливого значення n можна буде правильно знайти значення факторіалу. Ці значення можна визначити, наприклад, шляхом порівняння значень, знайдених програмно і значень, обчислених на папері чи за допомогою калькулятора. Так, для типів char і unsigned char змінної f максимально можливим значення n буде 5, для short int — 7, unsigned short int — 8, для unsigned int, як і для int — 12.

Для знаходження максимально можливого значення n для змінної f типів long long int і unsigned long long int скористаємося такою програмою (використаємо максимальні значення-константи для типу long long int — LLONG_MAX, а для типу unsigned long long int — ULLONG_MAX з бібліотеки limits.h):

```
/* Обчислення максимально можливого значення
   факторіалу для певного типу змінних */
...
#include <limits.h>
  int n, ind;
  unsigned long long int f;  // або long long int f
  printf("Обчислення значення факторіалу\n");

  f=1;
  ind=1;
  for (n=2; ind; n++) {
     if (f>ULLONG_MAX/n) ind=0;  // або LLONG_MAX
     else f*=n;
   }
  n-=2;   // останнє значення n, для якого обчислено n!
  printf("Найбільше значення\nдля unsigned long long int:\n\
    %d!=%llu",n,f);  // або long long int
...
```

```
Обчислення значення факторіалу
Найбільше значення
для unsigned long long int:
   20!=2432902008176640000
```

Таким чином, максимально можливим значенням *n* для змінної *f* типів long long int і unsigned long long int буде 20.

Програма обчислення значення біноміального коефіцієнта C_n^m (значення *n* повинно бути не більшим від 20; при цьому результат завжди буде точним) матиме вигляд:

```
/* Обчислення кількості сполук із n по m.
   Значення n повинно бути не більшим від 20.
   Найбільше значення C(20,10)=184756 */
...
long long int fact(int k);
int main() {
  int n, m, c;   // вхідні значення і результат
  printf("Обчислення значення кількості сполук\n\
   Введіть значення n (0<=n<=20) і m (0<=m<=n)\n");
```

```
  while (scanf("%d%d",&n,&m))
    if (n>=0 && m>=0 && n<=20 && m<=n) {
      c=fact(n)/fact(m)/fact(n-m);
      printf("  C(%d,%d)=%d\n",n,m,c);
    }
    else printf("  Помилкові дані\n");
...
}
/* Знаходження значення факторіалу */
long long int fact(int k) {
  long long int f;
  int i;
  f=1;
  for (i=2; i<=k; i++)
    f*=i;
  return f;
}
```

```
Обчислення значення кількості сполук
   Введіть значення n (0<=n<=20) і m (0<=m<=n)
20 19
  C(20,19)=20
20 10
  C(20,10)=184756
21 20
  Помилкові дані
3 5
  Помилкові дані
5 3
  C(5,3)=10
кінець
```

Крім того, що при такому підході біноміальні коефіцієнти можна обчислювати тільки для $n \le 20$, алгоритм також буде працювати неефективно. Для обчислення значення біноміального коефіцієнта буде здійснено три звернення до функції fact, при цьому в циклах буде виконано $(n-1)+(m-1)+(n-m-1)=2n-3$ операцій множення, а в головній функції також 2 операції ділення.

На практиці при обчисленні значення біноміального коефіцієнта ніколи не рахують повні факторіали, наприклад:

$$C_{20}^{2} = \frac{20!}{2!\cdot(20-2)!} = \frac{20!}{2!\cdot 18!} = \frac{2432902008176640000}{2\cdot 6402373705728000} = \frac{2432902008176640000}{12804747411456000} = 190,$$

а спочатку виконують скорочення, причому скорочують на факторіал, який має більше значення, наприклад:

$$C_{20}^{2} = \frac{20!}{2!\cdot(20-2)!} = \frac{20!}{2!\cdot 18!} = \frac{20\cdot 19\cdot 18!}{2!\cdot 18!} = \frac{20\cdot 19}{2\cdot 1} = 190.$$

Отже, якщо у формулі

$$C_n^m = \frac{n!}{m! \cdot (n-m)!}$$

виконати скорочення факторіалів, то, залежно від того, який знаменник $m!$ чи $(n-m)!$ має більше значення, формула набуде вигляду:

$$C_n^m = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (m+1)}{(n-m)!}$$

або

$$C_n^m = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-m+1)}{m!}.$$

Таким чином, іншим — *оптимальним підходом* щодо обчислення значення біноміального коефіцієнта є підхід, у якому реалізовується формула з виконаними скороченнями (у цій формулі в чисельнику і знаменнику буде однакова кількість множників). Візьмемо, наприклад, другу формулу:

$$C_n^m = \frac{n!}{m! \cdot (n-m)!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-m+1)}{m \cdot (m-1) \cdot (m-2) \cdot \ldots \cdot 1}\ , \text{ де } 0 \le m \le n.$$

У цій формулі можна побачити таку закономірність:

$$C_n^m = \frac{n!}{m! \cdot (n-m)!} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdot \ldots \cdot \frac{n-(m-1)}{m} = \prod_{i=0}^{m-1} \frac{n-i}{i+1}.$$

З останньої формули можна одержати рекурентну залежність:

$$C_n^{i+1} = C_n^i \cdot \frac{n-i}{i+1}\ , \text{ де } 0 \le i \le n-1 \text{ і } C_n^0 = 1. \qquad (*)$$

Справді:

$$C_n^{i+1} = \frac{n!}{(i+1)! \cdot (n-(i+1))!} = \frac{n!}{(i+1)! \cdot (n-i-1)!} = \frac{n!}{i! \cdot (n-i)!} \cdot \frac{n-i}{i+1} = C_n^i \cdot \frac{n-i}{i+1}.$$

Також в реалізації алгоритму треба врахувати, що $C_n^m = C_n^{n-m}$, і треба, щоб кількість множників у добутку була меншою, тобто треба вибрати $\min(m, n-m)$.

Маємо алгоритм (алгоритм 19а [4]):

```
if (m>n-m) m=n-m;    // для зменшення кількості множників
c=1;
for (i=0; i<m; i++)
  c=c*(n-i)/(i+1);  // обчислення добутку
```

Якщо проаналізувати цей алгоритм, то видно, що в циклі на кожному кроці обчислюються значення $(n-i)$ і $(i+1)$.

Алгоритм буде працювати ефективніше, якщо цикл, використовуючи можливість запису циклу з параметром у мові C, записати так:

```
for (i=0; i<m;)
   c=c*(n-i)/++i;  // обчислення добутку
```

тут значення параметра циклу змінюється при обчисленні значення біноміального коефіцієнта. Але не кожна мова програмування (наприклад, мова Pascal, у якій значення параметра циклу завжди неявно збільшується чи зменшується на 1) при програмуванні циклу з параметром допускає відсутність зміни значення параметра в заголовку циклу.

З формули

$$C_n^m = \prod_{i=0}^{m-1} \frac{n-i}{i+1},$$

змінивши в добутку значення індекса *i*, можна одержати таку еквівалентну формулу:

$$C_n^m = \prod_{i=1}^{m} \frac{n-(i-1)}{i} = \prod_{i=1}^{m} \frac{n-i+1}{i} = \prod_{i=1}^{m} \frac{(n+1)-i}{i}.$$

Рекурентна залежність замість вигляду (*) у даному випадку буде мати вигляд:

$$C_n^i = C_n^{i-1} \cdot \frac{(n+1)-i}{i}, \text{ де } 1 \le i \le n \text{ і } C_n^0 = 1. \qquad (**)$$

При її програмній реалізації замість обчислення значення $(i+1)$ буде використовуватися значення *i*. Матимемо алгоритм (алгоритм 19б [4]), який буде працювати в будь-якій мові програмування:

```
if (m>n-m) m=n-m;  // для зменшення кількості множників
c=1; n++;
for (i=1; i<=m; i++)
   c=c*(n-i)/i;  // обчислення добутку
```

У цьому алгоритмі на кожному кроці циклу економиться виконання операції додавання (при обчисленні біноміального коефіцієнта у знаменнику замість $(i+1)$ маємо *i*). Тому цей алгоритм працює ефективніше.

Слід зауважити, що не зважаючи на те, що у виразі `c=c*(n-i)/i` здійснюється ділення (при діленні двох цілих чисел результатом буде ціле число, тобто, якщо з'являється дробова частина, то вона відкидається), результат буде одержано точний, оскільки з рекурентної формули випливає, що на кожному кроці результатом має бути значення C_n^i, яке є цілим числом. Наприклад, щоб обчислити значення C_{15}^3 (при $C_{15}^0 = 1$), буде послідовно обчислено значення $C_{15}^1 = 1 \cdot 15/1 = 15$, $C_{15}^2 = 15 \cdot 14/2 = 105$, $C_{15}^3 = 105 \cdot 13/3 = 455$.

Якщо ж розглянутий вираз подати у вигляді `c=c/i*(n-i)` (тобто, спочатку виконувати ділення, а потім множення), то не завжди буде одержано правильне значення: $C_{15}^0 = 1$, $C_{15}^1 = 1/1 \cdot 15 = 15$, $C_{15}^2 = 15/2 \cdot 14 = 7 \cdot 14 = 98$, $C_{15}^3 = 98/3 \cdot 13 = 32 \cdot 13 = 416$.

Для забезпечення можливості одержання якомога більшого результуючого числа скористаємося типом unsigned long long int. При обчисленнях також треба врахувати, що в проміжних обчисленнях може виникати арифметичне переповнення.

У результаті матимемо програму для обчислення кількості сполук:

```
/* Обчислення кількості сполук із n по m. Значення
   n і m повинні бути такими, щоб проміжний результат
   обчислень не перевищував значення константи
   ULLONG_MAX=18446744073709551615 з бібліотеки limits.h */
#include <limits.h>
...
  int n, m;   // вхідні значення
  unsigned long long int c;  // кількість сполук
  int ind;  // індикатор переповнення значення змінної c
  int i;  // параметр циклу
  printf("Обчислення значення кількості сполук\n\
    Введіть значення n (n>=0) і m (0<=m<=n)\n");
  while (scanf("%d%d",&n,&m)) {
    if (n>=0 && m>=0 && m<=n) {
      if (m>n-m) m=n-m;  // береться менше значення
```

```
        /* Обчислення значення кількості сполук */
        c=1; n++;
        ind=1;
        for (i=1; i<=m && ind; i++)
          if (c>ULLONG_MAX/(n-i))
            ind=0;  // переповнення при проміжних розрахунках
          else
            c=c*(n-i)/i;  // обчислення добутку
        if (ind) printf("  C(%d,%d)=%llu\n",--n,m,c);
        else printf("  Арифметичне переповнення\n");
      }
      else printf("  Помилкові дані\n");
  }
...
}
```

```
Обчислення значення кількості сполук
    Введіть значення n (n>=0) і m (0<=m<=n)
5 2
  C(5,2)=10
5 3
  C(5,2)=10
70 21
  C(70,21)=385439532530137800
70 22
  Арифметичне переповнення
100 15
  C(100,15)=253338471349988640
100 16
  Арифметичне переповнення
2000 6
  C(2000,6)=88224108612633000
2000 7
  Арифметичне переповнення
кінець
```

Як видно з наведених прикладів обчислення значень кількості комбінацій, ця програма теж має обмеження щодо значень n і m, але ці обмеження не пов’язані з певним конкретним числом — тут треба, щоб проміжні результати обчислення значення C_n^m, наприклад, для типу unsigned long long int, не перевищували значення константи ULLONG_MAX=18446744073709551615.

Якщо тип unsigned long long int змінної c, в яку накопичується значення кількості сполук, замінити типом double, то проблему з арифметичним переповненням можна «перенести» трохи далі (доки одержуване значення буде меншим від максимального для типу double). При цьому для певних значень

n і m результати обчислень будуть наближеними (це пов'язано з виходом обчислених значень за межі розрядної сітки і їхнім заокругленням). Фрагмент обчислення значення кількості сполук у даному випадку буде таким:

```
...
  double c;
...
    /* Обчислення значення кількості сполук */
    c=1;
    n++;
    for (i=1; i<=m; i++)
        c=c*(n-i)/i;  // обчислення добутку
    printf("  C(%d,%d)=%.0f\n",--n,m,c);
...
```

а результати обчислень — такими:

```
Обчислення значення кількості сполук
   Введіть значення n (n>=0) і m (0<=m<=n)
100 15
  C(100,15)=253338471349988640
100 16
  C(100,16)=1345860629046814700
100 17
  C(100,17)=6650134872937201700
100 18
  C(100,18)=30664510802988208000
кінець
```

Як видно з результатів розрахунків для C_{100}^{15} одержані значення збігаються для типів unsigned long long int і double. Значення C_{100}^{16} для типу unsigned long long int порахувати не вдалося через арифметичне переповнення; для типу double це значення пораховане, але воно вже є наближеним — на 50 більше від знайденого вручну точного значення (1345860629046814650); C_{100}^{17} для типу double на 100 менше від точного (6650134872937201800); C_{100}^{18} для типу double на 300 менше від точного (30664510802988208300).

Складність алгоритму, який реалізує формулу (**), становить $O(n)$.

Кількість сполук також можна обчислити *за рекурентною формулою*, у якій використовується тільки додавання:

$$C_n^m = C_{n-1}^{m-1} + C_{n-1}^m,$$

де $n = 2, 3, 4, \ldots$, $m = \overline{1, n-1}$, $C_0^0 = 1$, $C_1^0 = 1$, $C_1^1 = 1$, $C_n^0 = 1$, $C_n^n = 1$.

Цю формулу легко запрограмувати рекурсивно. Відповідна функція, яка реалізує обчислення за цією формулою, така:

```
/* Рекурсивне обчислення кількості сполук C(n,m) */
int C(int n, int m) {
  int res;   // результат обчислення
  if (n==0 || m==0 || m==n) res=1;
  else res=C(n-1,m-1)+C(n-1,m);
  return res;
}
```

```
   Рекурсивне обчислення C(n,m)
Введіть значення n i m: 10 5
   C(10,5)=252; кількість викликів функції 503
Введіть значення n i m: 15 7
   C(15,7)=6435; кількість викликів функції 12869
Введіть значення n i m: 15 2
   C(15,2)=105; кількість викликів функції 209
Введіть значення n i m: 0 0
   C(0,0)=1; кількість викликів функції 1
Введіть значення n i m: 6 4
   C(6,4)=15; кількість викликів функції 29
Введіть значення n i m: кінець
```

Програмна рекурсивна реалізація рекурентної формули дуже проста. Прихильники рекурсії швидше за все виберуть саме такий підхід. Але кількість викликів цієї функції себе самою з меншими параметрами є величиною порядку $O(2^n)$ — це експоненційна часова складність. При кожному виклику проводяться обчислення і виділяється в стекові нова ділянка пам'яті (при цьому ємнісна складність буде не надто великою, оскільки будуть здійснюватися не тільки рекурсивні спуски, а й рекурсивні підйоми; максимальна глибина стека буде рівною *n*). Наприклад, при обчисленні значення C_{10}^5 буде здійснено 503 виклики функції, а при обчисленні значення C_{15}^7 — 12869 викликів (правда, кількість викликів також залежить від значення *m*; найбільша кількість буде при $m = n/2$). При цьому багато викликів будуть здійснюватися з однаковими параметрами. Таким чином цей алгоритм не є ефективним.

Наведену рекурентну формулу $C_n^m = C_{n-1}^{m-1} + C_{n-1}^m$ використовують при побудові трикутника Паскаля. Трикутник Паскаля містить значення всіх сполук для вказаного *n*, а також всіх попередніх і має вигляд:

ряд 1										1									
ряд 2									1		1								
ряд 3								1		2		1							
ряд 4							1		3		3		1						
ряд 5						1		4		6		4		1					
ряд 6					1		5		10		10		5		1				
ряд 7				1		6		15		20		15		6		1			
ряд 8			1		7		21		35		35		21		7		1		
ряд 9		1		8		28		56		70		56		28		8		1	і т.д

Щоб знайти C_n^m, треба в цьому трикутнику взяти елемент, який стоїть в $(n+1)$ ряду на $(m+1)$ місці. Наприклад, значення C_8^3 можна знайти в 9 ряду на 4 місці — це число 56.

У вигляді двовимірного масиву мови програмування С (матричному поданні) трикутник Паскаля має вигляд:

індекс *m* / *n*	0	1	2	3	4	5	6	7	8	
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	...

Програма, яка будує такий трикутник:

```
/* Побудова трикутника Паскаля */
...
  int a[N][N] = {0}; // обнулення матриці
  int  n,   //
       i, j;
  printf("   Побудова трикутника Паскаля\n");
  while (printf("Введіть значення n: "), scanf("%d",&n)) {
    // присвоєння початкових значень
    a[0][0]=1; a[1][0]=1; a[1][1]=1;
    // обчислення кількостей сполук у матриці
    for (i=2; i<=n; i++) {
```

```
        a[i][0]=1;
        for (j=1; j<i; j++)
          a[i][j]=a[i-1][j-1]+a[i-1][j];
        a[i][i]=1;
      }
      // виведення трикутника
        for (i=0; i<=n; i++) {
          printf("  ряд %2d>",i+1);
          for (j=0; j<=i; j++)
            printf("%5d",a[i][j]);
          printf("\n");
        }
    }
    printf("\n\n");
    system("pause");
    return 0;
}
```

```
  Побудова трикутника Паскаля
Введіть значення n: 9
  ряд  1>    1
  ряд  2>    1    1
  ряд  3>    1    2    1
  ряд  4>    1    3    3    1
  ряд  5>    1    4    6    4    1
  ряд  6>    1    5   10   10    5    1
  ряд  7>    1    6   15   20   15    6    1
  ряд  8>    1    7   21   35   35   21    7    1
  ряд  9>    1    8   28   56   70   56   28    8    1
  ряд 10>    1    9   36   84  126  126   84   36    9    1
Введіть значення n: кінець
```

Для одержання конкретного значення C_n^m треба у сформованому масиві значень вибрати відповідний елемент a[n][m]. У цьому алгоритмі виконується $(n-1)(n-2)/2$ проходів циклу, тобто часова складність алгоритму становить $O(n^2)$. Для зберігання масиву потрібно $(n+1)^2$ ділянок пам'яті по 4 байти, тобто ємнісна складність алгоритму становить теж $O(n^2)$. Тому швидше і з меншим використанням пам'яті буде одержано значення C_n^m з використанням даного алгоритму, ніж з використанням рекурсивного алгоритму. Перевагою тут є те, що не може виникнути арифметичне переповнення при проміжних розрахунках.

З формули

$$C_n^m = \frac{n!}{m! \cdot (n-m)!}, \quad \text{де } 0 \le m \le n,$$

виконавши перетворення

$$C_n^m = \frac{n!}{m!(n-m)!} = \frac{n \cdot (n-1)!}{m \cdot (m-1)! ((n-1)-(m-1))!} = \frac{n}{m} \cdot C_{n-1}^{m-1},$$

можна одержати ще одну рекурентну формулу:

$$C_n^m = \begin{cases} 1 & \text{при } m = 0; \\ \frac{n}{m} \cdot C_{n-1}^{m-1} & \text{при } 1 \le m \le n. \end{cases} \qquad (***)$$

У ній використовується множення й ділення, але на відміну від формули (**) немає віднімання.

Також, щоб оптимізувати цей алгоритм, треба врахувати, що $C_n^m = C_n^{n-m}$, і вибрати замість *m* значення $\min(m, n-m)$.

Якщо обчислення реалізувати ітераційно і вибрати такий порядок обчислень, що на кожному кроці відповідно до (***) значення *m* і *n* зменшуються на 1, то за рахунок того, що при цілочисельному діленні дробова частина результату відкидається, буде одержано неправильне значення C_n^m. Наприклад, при обчисленні C_5^2 буде одержано: $C_5^2 = \frac{5}{2} \cdot C_4^1 \to 2 \cdot C_4^1$, $C_4^1 = \frac{4}{1} \cdot C_3^0 = 4 \cdot 1 = 4$ і результатом буде $2 \cdot 4 = 8$ (правильне значення $C_5^2 = 10$). Якщо ж обчислення почати від $m = 1$, то, оскільки кількість всіх рекурентних кроків дорівнює *m*, початковим значенням кількості сполук є значення C_{n-m}^0 і на першому кроці обчислень за рекурентною формулою (***) дробовий множник має бути $\frac{n-m+1}{1}$. Наприклад, для одержання значення C_5^2 буде виконано таку послідовність обчислень: $\frac{5-2+1}{1} \cdot 1 = \frac{4}{1} \cdot 1 = 4$, $\frac{5}{2} \cdot 4 = 10$, тобто результат буде правильним.

Ітераційна реалізація алгоритму мовою C:

```
/* Ітераційне обчислення значення кількості сполук із n по
   m за рекурентною формулою C(n,m)=C(n-1,m-1)*n/m. Значення
   n i m повинні бути такими, щоб проміжний результат
   обчислень не перевищував значення константи
   ULLONG_MAX=18446744073709551615 з бібліотеки limits.h */
...
```

```
#include <limits.h>
int main() {
  int n, m;   // вхідні значення
  unsigned long long int c;  // кількість сполук
  int ind;  // індикатор переповнення значення змінної c
  int nr, i;  // робочі змінні – поточні значення n i m
  printf("Обчислення значення кількості сполук\n\
    Введіть значення n (n>=0) i m (0<=m<=n)\n");
  while (scanf("%d%d",&n,&m)) {
    if (n>=0 && m>=0 && m<=n) {
      if (m>n-m) m=n-m;  // береться менше значення
      /* Обчислення значення кількості сполук */
      c=1;
      nr=n-m;
      ind=1;
      for (i=1; i<=m && ind; i++) {
        nr++;
        if (c>ULLONG_MAX/nr)
          ind=0;  // переповнення при проміжних розрахунках
        else
          c=c*nr/i;   // обчислення добутку
      }
      if (ind) printf("  C(%d,%d)=%llu\n",n,m,c);
      else printf("  Арифметичне переповнення\n");
    }
  }
...
}
```

Порівняно з алгоритмом (алгоритм 19б [4]), у якому використано формулу (**) і на кожному проході циклу обчислюється значення `c*(n-i)/i`, тут обчислюється значення `c*nr/i` і нема віднімання. Обидва алгоритми дають однакові результати. Даний алгоритм є найоптимальнішим серед розглянутих.

Рекурсивна реалізація рекурентної формули (***) така:

```
/* Рекурсивне обчислення значення кількості сполук із n
   по m за рекурентною формулою C(n,m)=C(n-1,m-1)*n/m.
   Значення n і m повинні бути такими, щоб проміжний результат
   обчислень не перевищував значення константи
   ULLONG_MAX=18446744073709551615 з бібліотеки limits.h */
...
#include <limits.h>
int main() {
  int n, m;   // вхідні значення
  unsigned long long int cnm;  // кількість сполук
...
  while (scanf("%d%d",&n,&m)) {
    if (n>=0 && m>=0 && m<=n) {
      if (m>n-m) m=n-m;  // береться менше значення
      /* Обчислення значення кількості сполук */
      cnm=c(n, m);
      if (cnm)
        printf("  C(%d,%d)=%llu\n",n,m,cnm);
      else printf("  Арифметичне переповнення\n");
    }
  }
 ...
}
unsigned long long int c(int n, int m) {
  unsigned long long int cnm;   // результат обчислення
                                // 0 - переповнення
  if (m) {   // при m>0
    cnm=c(n-1, m-1);
    if (cnm>ULLONG_MAX/n)
      cnm=0;  // переповнення при проміжних розрахунках
    else
      cnm=cnm*n/m;   // обчислення добутку
  }
```

```
  else   // при m=0
    cnm=1;
  return cnm;
}
```

Якби в рекурсивній функції не було перевірки на переповнення, то обчислення значення кількості сполук мало б дуже простий вигляд:

```
  if (m)  // при m>0
    cnm=c(n-1, m-1)*n/m;   // обчислення добутку
  else   // при m=0
    cnm=1;
```

У цій функції кількість рекурсивних звернень збігається зі значенням *m*. Результати рекурсивних обчислень за формулою (***) такі ж, як і попередні.

Розглянутий вище процес вибору алгоритму показує, що ту саму задачу можна розв'язати по-різному.

Звичайно ж, якщо кількість виконань алгоритму незначна, то не варто довго думати і випробовувати різні підходи, щоб у програмі виконувалося на кілька операцій менше. Але, якщо алгоритм має виконуватися дуже велику кількість разів, то його треба оптимізувати, наскільки це можливо.

10.2. Швидке піднесення чисел до цілих степенів

Найпростіший алгоритм піднесення числа *a* до цілого невід'ємного степеня *n* полягає в множенні цього числа самого на себе *n* разів (див. п. 2.2); якщо степінь є цілим від'ємним числом, то можна знайти $a^{|n|}$, а потім обчислити $1/a^{|n|}$.

Якщо *n* — невелике ціле число, то треба робити саме так. Наприклад, для обчислення значення x^2 або x^3 відповідно найраціональніше буде запрограмувати степені, як `x*x` або `x*x*x` (у цьому випадку застосування в мові С функції pow(x, n) не є раціональним навіть через те, що звернення до функції пов'язане з витратами часу і пам'яті, а також виконанням самого алгоритму обчислення степеня). При застосуванні цього алгоритму операцію

множення треба виконати $(n-1)$ раз, а отже, часова складність алгоритму є лінійною і становить $O(n)$.

Для швидкого піднесення числа *a* до степеня *n*, якщо *n* досить велике ціле число (хоч може бути й невеликим, але $n \geq 0$), можна скористатися підходом, який полягає в поданні показника степеня в двійковій системі числення і використанні властивості $a^n \cdot a^m = a^{n+m}$. При цьому швидкість обчислень значно зростає. Так, якщо при звичайному обчисленні степеня *n* треба виконати $(n-1)$ разів множення, при застосуванні даного підходу треба перейти до двійкового подання числа *n*, обчислити приблизно $\log_2 n$ квадратів чисел і виконати приблизно таку саму кількість множень (залежно від кількості одиниць у двійковому поданні показника степеня *n*). Такий алгоритм називають алгоритмом *швидкого піднесення до степеня*, а також дихотомічним чи бінарним алгоритмом піднесення до степеня.

Розглянемо реалізацію вказаного підходу на прикладі. Нехай треба знайти 5^{23}. Якщо порахувати це значення на калькуляторі, то одержимо 17-значне десяткове число $5^{23} = 11920928955078125$ (максимально можливим цілим числом типу long long int є 19-значне число 9223372036854775807, типу unsigned long long int — 20-значне число 18446744073709551615)[22].

Подамо показник степеня 23_{10} у двійковій системі числення $23_{10} = 10111_2$. Це означає:

$$10111_2 = 2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1.$$

Тоді:

$$5^{23} = 5^{16} \cdot 5^4 \cdot 5^2 \cdot 5^1.$$

Степені числа 5 можна одержати так (починаючи з другого степеня як квадрат попереднього степеня):

---

[22] Задача знаходження великих степенів чисел виникає в криптографії. Ключами у сучасних асиметричних криптосистемах, як правило, є цілі числа довжиною 1000 і більше двійкових розрядів. Такі числа зберігають в масивах у вигляді послідовності цифр в системі числення з основою *b* (як правило, основою беруть 2^8, 2^{16} чи 2^{32}). Аналогічний підхід реалізовано при обчисленні факторіалу великого числа (див. приклад 8.5.3).

$$5^1 = 5,$$

$$5^2 = (5^1)^2,$$

$$5^4 = (5^2)^2,$$

$$5^{16} = (5^8)^2 = ((5^4)^2)^2.$$

Розглянуту схему називають *схемою швидкого піднесення до степеня «справа наліво»*. Алгоритм цієї схеми такий:

K1: Подати двійкове число $n = n_k n_{k-1} \ldots n_2 n_1 n_0$ у вигляді:

$$n = n_0 + n_1 \cdot 2 + n_2 \cdot 2^2 + n_3 \cdot 2^3 + \ldots + n_{k-1} \cdot 2^{k-1} + n_k \cdot 2^k,$$

де $n_i \in \{0;1\}$, $i = \overline{0,k}$.

K2: Обчислити a^{2^i} при $i = \overline{0,k}$ шляхом послідовного піднесення до квадрату кожного попереднього значення:

$i = 0$, $b_0 = a$ — значення $a^{2^0} = a^1$;

$i = 1$, $b_1 = b_0^{\ 2}$ — значення $a^{2^1} = a^2$;

$i = 2$, $b_2 = b_1^{\ 2}$ — значення $a^{2^2} = a^4$;

$i = 3$, $b_3 = b_2^{\ 2}$ — значення $a^{2^3} = a^8$;

…

$i = k$, $b_k = b_{k-1}^{\ 2}$ — значення a^{2^k};

K3: Обчислити a^n за формулою:

$$a^n = b_0^{n_0} \cdot b_1^{n_1} \cdot b_2^{n_2} \cdot \ldots \cdot b_k^{n_k}, \text{ де } n_i \in \{0;1\},\ i = \overline{0,k}.$$

Отже, треба знайти всі степені числа $b_i = a^{2^i}$ ($i = \overline{0,k}$), але використати для обчислення шуканого степеня тільки ті, які відповідають одиницям у двійковому поданні показника степеня, а ті, що відповідають нулям, пропустити.

Поданий алгоритм зручно виконувати на папері, а при реалізації на комп'ютері треба зберігати значення всіх обчислених степенів, наприклад, у масиві. Можна алгоритм модифікувати так, щоб обчислення степенів і накопичення результату відбувалися паралельно. При цьому також не треба виконувати жодних дій для подання показника степеня у двійковій системі чис-

лення, оскільки в комп'ютері внутрішнім поданням чисел є двійкове. Крім того, для одержання двійкових розрядів числа можна скористатися дуже швидкими побітовими операціями.

Алгоритм швидкого піднесення числа до степеня «справа наліво»:

К1. Ввести числа a, n.

К2. r=1. // початкове значення результату

К3. Якщо n=0, то перейти на К8.

К4. Якщо наймолодший (правий) розряд числа n дорівнює 1, то r=r*a.

К5. n=n/2. // відкидання наймолодшого розряду

К6. Якщо n$\neq$0, то a=a*a. // квадрат числа для наступного розряду

К7. Перейти на К3.

К8. Вивести r.

У цьому алгоритмі цикл К3-К7 буде виконуватися стільки разів, скільки значущих розрядів має число n, — $\log_2 n$ без дробової частини, тобто $\lfloor \log_2 n \rfloor$. Квадрат буде обчислюватися у кожному проході циклу (крок К5), крім останнього, тобто $(\lfloor \log_2 n \rfloor - 1)$ разів. Добуток при накопиченні результату (крок К4), буде обчислюватися стільки разів, скільки 1 містить двійкове подання числа n, — не більше $\lfloor \log_2 n \rfloor$ разів. Таким чином операцію множення буде виконано не більше $(2 \cdot \lfloor \log_2 n \rfloor - 1)$ разів. Отже, обчислювальна складність алгоритму дорівнює $O(\log_2 n)$. Ємнісна складність алгоритму становить $O(1)$, оскільки обсяг використовуваної пам'яті від величини n не залежить.

Приклад 10.2.1. Програмна реалізація мовою C алгоритму швидкого піднесення до степеня за схемою «справа наліво»:

```
/* Швидке піднесення числа a до цілого невід'ємного
   степеня n за схемою "справа наліво" */
...
unsigned long long int an_p(unsigned long long int a,
                            unsigned int n);
int main() {
  unsigned long long int a; // число
```

```
  unsigned int n;   // показник степеня
  printf("  Швидке піднесення до степеня a^n\n\t'справа наліво'\n");
  while (printf("Задайте значення a і n  "), scanf("%llu%u",&a, &n))
    printf("  %llu^%u=%llu\n",a ,n, an_p(a, n));
...
}
/* Швидке піднесення до степеня a^n "справа наліво" */
unsigned long long int an_p(unsigned long long int a,
                            unsigned int n) {
  unsigned long long int r;   // степінь числа
  r=1;   // початкове значення результату
  while(n) {
    if(n&1)   // якщо наймолодший розряд дорівнює 1
      r*=a;   // накопичення результату
    n>>=1;    // відкидання наймолодшого розряду
    if (n)   // квадрат для наступного розряду
      a*=a;
  }
  return r;
}
```

```
  Швидке піднесення до степеня a^n
        'справа наліво'
Задайте значення a і n  5 0
  5^0=1
Задайте значення a і n  5 1
  5^1=5
Задайте значення a і n  5 3
  5^3=125
Задайте значення a і n  3 5
  3^5=243
Задайте значення a і n  5 23
  5^23=11920928955078125
Задайте значення a і n  кінець
```

Розглянуту вище схему швидкого піднесення до цілого невід'ємного степеня «справа наліво» за допомогою рекурентного співвідношення можна подати так:

$$a^n = \begin{cases} 1, & \text{якщо } n = 0; \\ (a^{n/2})^2, & \text{якщо } n \text{ – парне;} \\ a \cdot (a^{\lfloor n/2 \rfloor})^2, & \text{якщо } n \text{ – непарне.} \end{cases}$$

Приклад 10.2.2. Реалізація мовою С рекурентного співвідношення швидкого піднесення до степеня за схемою «справа наліво» — рекурсивна функція:

```
/* Швидке піднесення до степеня a^n "справа наліво"
   з використанням рекурсії */
unsigned long long int an_p_r(unsigned long long int a,
                              unsigned int n) {
  unsigned long long int r;   // результат обчислення
  if (n)  {   // якщо n>0
    r = an_p_r(a, n>>1);
    if (n&1)   // якщо n непарне
      r=a*r*r;
    else   // якщо n парне
      r=r*r;
  }
  else    // якщо n=0
    r=1;
  return r;
}
```

Код цієї функції можна трохи модифікувати (дистрибутивно винести спільні обчислення з вкладеного умовного оператора):

```
...
  if (n)  {   // якщо n>0
    r = an_p_r(a, n>>1);
    r=r*r;   // якщо n парне
    if (n&1)   // якщо n непарне
      r=a*r;
  }
  else    // якщо n=0
    r=1;
...
```

Обчислювальна складність рекурсивного алгоритму, як і ітераційного, дорівнює $O(\log_2 n)$. Але у цьому випадку ємнісна складність, на відміну від складності нерекурсивного алгоритму $O(1)$, теж становить $O(\log_2 n)$ за рахунок використання стекової пам'яті при $\approx \log_2 n$ рекурсивних викликах.

Якщо використати мультиплікативний аналог схеми Горнера, то степінь 5^{23} можна подати через добуток менших степенів:

$$
\begin{aligned}
5^{23} &= 5^{16} \cdot 5^{4} \cdot 5^{2} \cdot 5^{1} = \\
&= ((((5^{1})^{2} \cdot 5^{0})^{2} \cdot 5^{1})^{2} \cdot 5^{1})^{2} \cdot 5^{1} = \\
&= ((((5)^{2} \cdot 1)^{2} \cdot 5)^{2} \cdot 5)^{2} \cdot 5 = \\
&= (((5^{2})^{2} \cdot 5)^{2} \cdot 5)^{2} \cdot 5 .
\end{aligned}
$$

Таким чином, відповідно до цієї схеми, початковим значенням результату береться число *a*, на кожному кроці накопичуване значення результату підноситься до квадрату і, крім того, якщо поточний розряд показника степеня, ідучи зліва направо, в його двійковому поданні дорівнює 1, то накопичуване значення ще треба помножити на *a*. Таку схему називають *алгоритмом швидкого піднесення до степеня «зліва направо»*.

Алгоритм швидкого піднесення числа до степеня «зліва направо»:

К1. Ввести числа a, n.

К2. У двійковому поданні числа n знайти найстарший (лівий) розряд, рівний 1. Запам'ятати його номер k.

К3. Якщо k=0, то r=1; інакше r=a. // початкове значення результату

К4. Якщо k<0, то перейти на К9.

К5. r=r*r. // піднесення проміжного значення результату до квадрату

К6. Якщо поточний розряд числа n дорівнює 1, то r=r*a.

К7. k=k-1. // перехід до правішого розряду

К8. Перейти на К4.

К9. Вивести r.

В алгоритмі обчислення цілого додатного степеня числа «зліва направо», як і в алгоритмі «справа наліво», кількість виконуваних операцій множення (при піднесенні до квадрату і при множенні) буде такою самою і його складність теж дорівнює $O(\log_2 n)$. Але перевага схеми «зліва направо» в тому, що накопичуване значення домножується на те саме число *a* і, якщо *a* мале, то множення буде виконуватися швидше, за рахунок чого ефективність алгоритму зростає.

Приклад 10.2.3. Програмна реалізація алгоритму швидкого піднесення до степеня «зліва направо» мовою С:

```
/* Швидке піднесення числа a до цілого невід'ємного
   степеня n за схемою "зліва направо" */
...
unsigned long long int an_l(unsigned long long int a,
                          unsigned int n);
int main() {
  unsigned long long int a; // число
  unsigned int n;   // показник степеня
  printf("  Швидке піднесення до степеня a^n 'зліва направо'\n");
  while (printf("Задайте значення a, n\n"),scanf("%llu%u",&a, &n))
    printf("%llu^%u=%llu\n",a ,n, an_l(a, n));
 ...
}
/* Швидке піднесення до степеня a^n "зліва направо" */
unsigned long long int an_l(unsigned long long int a,
                          unsigned int n) {
  unsigned long long int r;   // степінь числа
  int k,   // поточний номер розряду
    ind;   // індикатор виявлення першої 1 у двійковому поданні n
  /* Пошук першої зліва 1 у двійковому поданні n */
  k=sizeof(n)*8; ind=1;
  do {
    k--;
    if ((n>>k)&1)
      ind=0;
  } while (ind && k);
  /* Задання початкового значення результату */
  if (n)   // якщо n>0
    r=a;
  else   // якщо n=0
    r=1;
```

```
  /* Одержання результату a^n */
  while(k) {
    r*=r;   // піднесення до квадрату
    k--;
    if ((n>>k)&1)
      r*=a;   // якщо поточний розряд дорівнює 1
  }
  return r;
}
```

У наведених алгоритмах швидкого піднесення до степеня число a може бути як цілим, так і дійсним (хоч в криптографії використовують тільки цілі додатні числа), але показник степеня — тільки цілим невід'ємним числом.

У криптографічних застосуваннях частіше, ніж степені цілих чисел, використовують остачі від ділення степенів на деякий модуль, тобто *піднесення до степеня за модулем*:

$$r \equiv a^n \bmod m.$$

При обчисленні такого значення не треба спочатку шукати степінь a^n (що дасть дуже велике число), а досить скористатися такою властивістю порівнянь за модулем:

$$r \equiv (a \cdot b) \bmod m \equiv (a \cdot (b \bmod m)) \bmod m.$$

Таким чином обчислення значення $a^n \bmod m$ можна запрограмувати так:

```
r=1;
for (i=1; i<n; i++)
  r=(r*a)%m;
```

Але цей алгоритм має часову лінійну складність $O(n)$.

Як і для звичайного піднесення до степеня, для піднесення до степеня за модулем можна застосувати підхід, який використовує двійковове подання показника степеня. Алгоритми швидкого піднесення до степеня за модулем «справа наліво» і «зліва направо» можна одержати з відповідних алгоритмів швидкого піднесення до степеня без модуля, замінивши в них дії r=r*a, r=r*r і a=a*a (у програмних реалізаціях оператори `r*=a;`, `r*=r;` і `a*=a;`) на дії

r=(r*a) mod m, r=(r*r) mod m і a=(a*a) mod m (у програмних реалізаціях на оператори `r=(r*a)%m;`, `r=(r*r)%m;` і `a=(a*a)%m;`). При цьому, якщо проміжні результати ($r \cdot a$, r^2 і a^2) не будуть перевищувати максимально допустимого значення типу unsigned int (`4294967295`), то в програмній реалізації відпадає необхідність використовувати змінні типу unsigned long long int, а досить використати змінні типу unsigned int. Наприклад, при обчисленні значення $73^{11} \bmod 231$ проміжні результати обчислень (добутки r*a, r*r, a*a) не перевищують 5329 (при застосуванні алгоритму «справа наліво») і 43264 (при застосуванні алгоритму «зліва направо» і рекурсивного алгоритму), хоч саме значення $73^{11} = 31372668556835970837$ є 21-значним числом.

Приклад 10.2.4. Для випадку застосування схеми «справа наліво» піднесення до степеня за модулем програмна реалізація мовою C буде такою:

```
/* Швидке піднесення числа a до цілого невід'ємного
   степеня n за модулем m за схемою "справа наліво" */
...
unsigned int anm_p(unsigned  int a, unsigned int n,
                   unsigned int m);
int main() {
  unsigned int a,   // число
               n,   // показник степеня
               m;   // модуль
  printf("  Швидке піднесення до степеня за модулем\n\
         (a^n)mod m 'справа наліво'\n");
  while (printf("Задайте значення a, n і m  "),
                            scanf("%u%u%u",&a, &n, &m))
    printf("  (%u^%u)mod%u=%u\n",a ,n, m, anm_p(a, n, m));
...
}
/* Швидке піднесення до степеня за модулем
   (a^n)mod m "справа наліво" */
unsigned int anm_p(unsigned int a, unsigned int n,
                   unsigned int m) {
```

```
  unsigned int r;   // степінь числа
  r=1;   // початкове значення результату
  while(n) {
    if(n&1)   // якщо наймолодший розряд дорівнює 1
      r=(r*a)%m;   // накопичення результату
    n>>=1;    // відкидання наймолодшого розряду
    if (n)
      a=(a*a)%m;   // квадрат для наступного розряду
  }
  return r;
}
```

Відповідна рекурсивна функція буде такою:

```
/* Швидке піднесення до степеня за модулем (a^n)mod m
   "справа наліво" з використанням рекурсії */
unsigned int anm_p_r(unsigned int a, unsigned int n,
                     unsigned int m) {
  unsigned int r;   // результат обчислення
  if (n)  {   // якщо n>0
    r = anm_p_r(a, n>>1, m);
    r=(r*r)%m;   // якщо n парне
    if (n&1)   // якщо n непарне, треба r домножити на а
      r=(a*r)%m;
  }
  else    // якщо n=0
    r=1;
  return r;
}
```

```
 Швидке піднесення до степеня за модулем
       (a^n)mod m 'справа наліво'
Задайте значення a, n і m  5 3 35
  (5^3)mod35=20
Задайте значення a, n і m  73 11 231
  (73^11)mod231=40
Задайте значення a, n і m  111 87 134
  (111^87)mod134=45
Задайте значення a, n і m  кінець
```

На даний час алгоритмів піднесення до степеня з кращою оцінкою, ніж $O(\log_2 n)$, не існує, хоч в деяких часткових випадках кількість операцій можна скоротити.

Проблемі швидкого піднесення до степеня приділяють досить багато уваги, оскільки ці алгоритми використовують у задачах шифрування з відкритим ключем (генерація параметрів, шифрування, розшифровування), при факторизації чисел (розкладанні на прості множники), дискретному логарифмуванні (знаходженні показника степеня за результатом) тощо.

10.3. Генерація випадкових чисел

Випадковими є числа, які вибираються випадково. Також інтуїтивно можна визначити випадкову послідовність чисел як послідовність, яка має властивості, притаманні всім послідовностям незалежних вибірок випадкових рівномірно розподілених величин.

Використання випадкових чисел. Випадкові числа використовують для формування випадкових вибірок при дослідженні різноманітних явищ, моделюванні різноманітних процесів, при тестуванні програм, у криптографії, стеганографії, в обчислювальних методах, в основі яких лежить принцип рандомізації (наприклад, в методі статистичних випробувань, який ще називають методом Монте-Карло[23]), при контролі якості виробів, у комп'ютерних іграх і в багатьох інших випадках. При цьому розподіл випадкових чисел може бути різним — рівномірним, нормальним, логнормальним, експоненційним, Пуассона, Фішера-Снедекора, Стьюдента, біноміальним, геометричним, Коші тощо.

Найчастіше на практиці застосовують рівномірно розподілені на певному інтервалі випадкові числа (інші розподіли випадкових чисел, як правило, одержують з рівномірного розподілу). При цьому, якщо згенеровані числа зміщуються в якийсь бік інтервалу, то результат розв'язування зада-

[23] В основі методу статистичних випробувань (Монте-Карло) лежить генерація рівномірно розподілених випадкових чисел. Метод передбачає індивідуальний підхід при розв'язуванні кожної задачі, залежно від моделі, яка використовує випадкові числа. За допомогою цього методу можна обчислювати значення визначених інтегралів, знаходити площі чи об'єми фігур, які є перетином різних геометричних тіл, задачі оптимізації та інші. У комп'ютерних експериментальних дослідженнях метод статистичних випробувань дає змогу імітувати випадкові явища, які відбуваються в змодельованих системах, багаторазово проводити обчислення і одержувати досить точні результати. Дональд Кнут [20, с. 218] рекомендує для одержання стабільного розв'язку методом Монте-Карло використовувати принаймні два різні генератори випадкових чисел.

чі буде неправильним. Тому проблема створення хорошого генератора є досить актуальною.

У комп'ютерних розрахунках замість випадкових використовують псевдовипадкові (ще кажуть: квазівипадкові; «псевдо-» і «квазі-» означають «нібито», «майже», «несправжній», «уявний», «фальшивий») числа, які досить просто генеруються програмно; при цьому кожне наступне згенероване число залежить від попереднього, тобто $x_{n+1} = f(x_n)$.

Функція random мови Pascal генерує дробові числа з проміжку 0..1 і цілі із заданого користувачем проміжку; ініціалізація генерації здійснюється процедурою randomize. Мова C++ має бібліотеку random, функції якої генерують значення випадкових величин з різними стандартними розподілами.

У мові C функція rand з бібліотеки stdlib.h генерує рівномірно розподілені цілі числа з проміжку 0..RAND_MAX (у середовищі Visual Studio — 0..32767). Якщо треба одержати цілі числа з меншого за довжиною проміжку a..b, то їх можна обчислити за формулою `r%(b-a+1)+a`. Наприклад, для одержання цілих чисел від 0 до 10 включно — `rand()%11`; чисел –1, 0 чи 1 — `rand()%3-1`. Проте при цьому в ряді випадків менші числа будуть дещо ймовірнішими за рахунок того, що значення `RAND_MAX` не кратне довжині проміжку a..b. Тому, якщо в розрахунках треба не просто шукати випадкові числа, а суттєво, щоб не порушувалася рівномірність розподілу, треба враховувати тільки менші або рівні `RAND_MAX-RAND_MAX%(b-a+1)` числа, а згенеровані більші числа просто ігнорувати[24].

Для генерації дійсних чисел застосовують формулу `(float)rand()/RAND_MAX*(b-a)+a.`

Щоб при повторних виконаннях програми кожен раз не генерувалася та сама послідовність чисел, треба ініціалізувати генерацію, скориставшись функцією srand (бібліотека stdlib.h). Але, якщо аргументом функції srand взяти якусь константу і її не змінювати, то при кожному виконанні програми також

[24] За допомогою критерію χ^2-Пірсона при рівнях значущості $\alpha = 0{,}05$ і $\alpha = 0{,}01$ для 100 і більше чисел з проміжку 0..9 при багаторазових генеруваннях перевірено й підтверджено гіпотезу про рівномірність одержаного розподілу.

буде генеруватися та сама послідовність. Щоб уникнути таких повторень, можна значення параметра функції srand задавати в діалозі, проте найкраще при ініціалізації скористатися поточним часом (який весь час змінюється) для чого треба звернутися до функції time (бібліотека time.h), яка повертає поточний календарний час системи в секундах. Таким чином, оператор ініціалізації матиме вигляд: `srand(time(0));`.

При комп'ютерній генерації псевдовипадкових чисел їхні послідовності повторюються або стають передбачуваними з певним періодом. Цю характеристику називають *довжиною періоду* (*циклу*) *повторення* і вона є дуже важливою — чим більша довжина періоду, тим краще. Величина періоду залежить від використовуваного методу генерації, від вибраного початкового значення. Так, при використанні псевдовипадкових чисел в криптосистемах довжина періоду повторення практично визначає можливу кількість ключів криптосистеми — чим ця довжина більша, тим складніше підібрати ключ. Слід зауважити, що у всіх послідовностях, у яких значення наступного члена залежить від значення попереднього $x_{n+1} = f(x_n)$, утворюються підпослідовності повторюваних елементів.

Надалі програмно згенеровані числа для спрощення будемо називати випадковими.

Розглянемо приклади використання псевдовипадкових чисел як цілих, так і дійсних.

Приклад 10.3.1. Генерація 100 випадкових чисел, виведення тільки парних чисел і визначення їхньої кількості.

Алгоритм мовою С має вигляд:

```
/* Генерація 100 чисел з проміжку від 0 до 32767 і
   підрахунок кількості парних чисел (моделювання
   випадання герба і номіналу при підкиданні монети) */
#include <stdlib.h>   // для одержання випадкових чисел
#include <time.h>     // для srand(time(0));
...
```

```
  int r,    // згенероване випадкове число
      n,    // номер числа
      k;    // кількість парних чисел
  /* Число генерується на проміжку від 0 до 32767 */
  srand(time(0));
  n=0; k=0;
  printf("\tЗгенеровані  парні  числа  (номер  згенерованого
числа – число)\n");
  while (n<100) {
    n++;
    r=rand();

    if (!(r&1)) {   // чи є число парним
      k++;
      printf("%3d - %5d     ",n,r);
    }
  }
  printf("\nКількість парних чисел серед 100 згенерованих= %d", k);
...
```

```
        Згенеровані парні числа (номер згенерованого числа – число)
  1 - 26572       7 - 28760       9 - 19846      10 - 30412      11 - 28542
 14 - 14018      15 - 17696      17 - 13732      19 - 30804      24 - 28946
 25 - 32408      26 - 11926      32 - 12952      33 - 10872      34 - 19996
 36 - 25446      37 -  2280      39 - 32136      40 - 22126      41 - 10622
 42 - 22004      46 - 10756      47 - 25290      51 - 10632      54 - 24580
 55 - 10760      56 - 22740      57 - 14230      58 - 25808      61 -  5932
 62 -  1398      65 - 19534      66 -  2012      70 - 17060      72 - 12390
 75 - 29632      76 - 32528      82 -  4964      84 -  6820      85 -  6690
 87 - 25250      88 -  5252      91 - 13132      92 - 29624      93 -  8734
 95 -  5126      97 -  6710      99 - 14246
Кількість парних чисел серед 100 згенерованих= 48
```

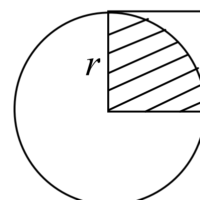
```
        Згенеровані парні числа (номер згенерованого числа – число)
  2 -  1934       3 - 10342       5 - 20970       6 - 29004       7 - 13794
  8 - 27296       9 - 17624      10 -  3336      11 - 12124      12 -  6498
 13 - 25402      14 - 15652      15 - 23426      18 - 20128      19 -  4638
 21 - 16692      24 - 21134      25 - 14938      27 -  4956      29 -  8474
 31 -  5462      34 - 24000      36 -  9508      37 -  1092      39 - 16118
 40 -  8318      43 -  8924      44 - 20712      49 - 14544      50 -  5186
 51 -   710      53 - 10532      60 - 14718      61 - 32364      64 -  7148
 65 -  7432      66 - 18876      67 -  3320      68 -  6888      69 -  3868
 72 -  8408      73 - 16324      74 - 22406      79 - 10486      80 - 20614
 82 -  7682      83 - 12630      88 -  4124      90 - 17558      91 -  7912
 92 - 10530      94 - 26846      95 - 29402      96 -  8230      97 - 22644
 98 - 18730     100 -  8746
Кількість парних чисел серед 100 згенерованих= 57
```

Програма, по суті, моделює випадання герба і номіналу при підкиданні монети, якщо, наприклад, парне число інтерпретувати як герб, а непарне як номінал. Комп'ютерне моделювання підкидання монети дає результати, близькі до отриманих дослідниками на практиці[25].

Приклад 10.3.2. Обчислення наближеного значення числа π із застосуванням методу Монте-Карло.

Формулу для розрахунків одержимо з таких міркувань. Нехай є коло, через центр якого проходять осі координат. Щоб працювати тільки з цілими додатними числами, розглянемо четвертину круга (прямокутний сектор). Навколо сектора опишемо квадрат. При цьому сторона квадрата і радіус кола будуть дорівнювати RAND_MAX=32767. Точки, які потрапляють на границі, будемо враховувати; щоб точки, які потраплять на обмежуючі сектор радіуси, двічі не враховувалися в повному крузі, то, наприклад, при $x=0$ або при $y=0$ їх не будемо враховувати — врахуємо тільки при $y=0$. Обчислення проведемо кілька разів (до введення числа 0), вказуючи різну кількість точок, які генеруються.

Тоді:



$$S_{круга} = \pi r^2;\ S_{сектора} = \frac{\pi r^2}{4};\ S_{квадрата} = r^2.$$

Отже, ймовірність потрапляння точки в сектор дорівнює:

$$p = \frac{S_{сектора}}{S_{квадрата}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \approx \frac{n}{N}.$$

Звідки

$$\pi \approx 4n/N,$$

де N — кількість усіх випробувань; n — кількість точок, які потрапили в сектор.

Алгоритм мовою С має вигляд:

```
/* Обчислення числа Pi за методом Монте-Карло.
   Враховуються граничні точки кола */
...
```

[25] При $\alpha = 0,05$ і $\alpha = 0,01$ багаторазова статистична перевірка рівномірності розподілу частот між двома значеннями за критерієм χ^2 (з 1 ступенем свободи і поправкою на неперервність) жодного разу не дала негативного результату для 20 і більше «підкидань».

```
  int x, y;  // згенеровані координати точки
  long long int N,   // кількість усіх згенерованих точок у
                     // квадраті; при N=0 робота закінчується
       n,   // кількість точок, які потрапили в четвертину круга
       i;   // лічильник згенерованих точок
  double Pi;   // обчислене значення
  srand(time(0));
  printf("  Наближене* значення числа Pi\n");
  printf("Вкажіть кількість генерацій\nвипадкових точок");
  while (printf("\n  N="), scanf("%lld",&N), N) {
    n=0;
    for (i=0; i<N; i++)  {
      x=rand(); y=rand();  // згенеровані координати точки
      if (x!=0 && x*x+y*y<=RAND_MAX*RAND_MAX) n++;
    }
    Pi=4.*(double)n/(double)N;
    printf ("    число Pi=%.4f", Pi);
  }
  printf(" *Використано метод Монте-Карло");
...
```

```
  Наближене* значення числа Pi
Вкажіть кількість генерацій
випадкових точок
  N=100
    число Pi=2.8800
  N=100
    число Pi=3.1200
  N=1000
    число Pi=3.1280
  N=100000
    число Pi=3.1378
  N=1000000
    число Pi=3.1412
  N=1000000
    число Pi=3.1426
  N=100000000
    число Pi=3.1415
  N=1000000000
    число Pi=3.1414
  N=0
 *Використано метод Монте-Карло
```

```
  Наближене* значення числа Pi
Вкажіть кількість генерацій
випадкових точок
  N=100
    число Pi=2.9200
  N=1000
    число Pi=3.2120
  N=100000
    число Pi=3.1442
  N=1000000
    число Pi=3.1424
  N=1000000
    число Pi=3.1437
  N=100000000
    число Pi=3.1414
  N=1000000000
    число Pi=3.1414
  N=1000000000
    число Pi=3.1414
  N=0
 *Використано метод Монте-Карло
```

Генерація рівномірно розподілених випадкових чисел. Існують різні алгоритми генерації випадкових чисел — досить прості і складніші, ті, що працюють швидко, і повільніші. Але при цьому слід зауважити, що внаслідок дискретності подання в комп'ютері неперервних чисел і періодичності алгоритмічно генерованої послідовності, ідеальну послідовність випадкових чисел при комп'ютерній генерації одержати неможливо.

Для одержання дійсних випадкових чисел U_n з проміжку $[0;1)$ можна генерувати цілі числа X_n з проміжку $[0;m)$ і застосовувати перетворення $U_n = X_n / m$. Як правило, число *m* вибирають рівним значенню 2^k, де *k* — кількість розрядів, які відводяться під машинне слово (довжина машинного слова дорівнює розрядності регістрів процесора і / або розрядності шини даних). Проте *k* не обов'язково має відповідати машинному слову. Таким чином, якщо розрахунки проводити з використанням цілих чисел типу short int, то $k = 15$, а $m = 32768$ (хоч максимально можливим значенням типу short int є число 32767), якщо unsigned short int — $k = 16$, $m = 65536$, якщо unsigned int — $k = 32$, $m = 4294967296$, якщо використовувати цілі числа типу unsigned long long int, то *k*=64, а *m*=18446744073709551616. Також значенням *m* можна брати і будь-яке інше число $m > 0$.

Крім того, якщо користувачеві потрібні не числа $U_n \in [0;1)$, а числа $U'_n \in [a;b)$, то треба скористатися формулою $U'_n = (b - a) \cdot U_n + a$.

Розробкою ефективних алгоритмів генерації надійних і якісних випадкових чисел займалися і займаються багато вчених. На основі математичних і експериментальних досліджень розроблено ряд методів побудови, які відрізняються один від одного оперативністю, доступністю, періодом випадкової послідовності, іншими характеристиками.

Розглянемо деякі методи генерації випадкових чисел.

Метод середин квадратів (алгоритм фон Неймана). Вперше алгоритм генерації випадкових чисел запропонував Джон фон Нейман у 1946 році. Ідея алгоритму дуже проста — піднести до квадрату попереднє (або початкове) число і виділити середні цифри. Але такий підхід при невдалому виборі по-

чаткового числа може закінчитися дуже коротким циклом повторюваних чисел (наприклад, з довжиною періоду повторення 4) або виродитися в 0 або якесь інше число.

Якщо початкове число вибрати вдало, то максимальна довжина циклу буде рівною b^k, де b — основа системи числення (наприклад, двійкова $b = 2$ чи десяткова $b = 10$), k — кількість розрядів числа, яке підноситься до квадрату.

Приклад 10.3.3. Генерація випадкових 8-значних десяткових чисел (якщо старші розряди в числах відсутні, то значеннями цих розрядів є нулі) з проміжку $[0;100000000)$ з використанням алгоритму фон Неймана. Для піднесення до квадрату беруться 4 середні цифри числа. Початковим випадковим числом беруться останні 8 цифр поточного системного часу.

Алгоритм мовою С:

```
/* Алгоритм Фон Неймана генерації псевдовипадкових
   чисел. Генеруються восьмирозрядні числа */
#include <time.h>
...
  int n,  // кількість чисел
    r,  // "випадкове число"
    i;  // лічильник циклу
  printf("Вкажіть кількість чисел,\n\
    які треба згенерувати  ");
  scanf("%d",&n);
  /* Одержання першого числа */
  r=time(0)%100000000;  // останні 8 цифр поточного часу
  printf ("%10d",r);
  /* Генерація наступних n чисел */
  for (i=1; i<=n; i++) {
    r=(r/100)%10000;  // 4 середні цифри
    r=r*r;  // квадрат 4-значного числа
    printf ("%10d",r);
  }
...
```

(1)

```
Вкажіть кількість чисел,
    які треба згенерувати  50
  99824470  67963536  92833225  69422224  17825284  68095504    912025  83174400
   3041536    172225   2965284  93161104   2595321  35438209  19201924   4076361
    582169  33884041  78145600   2119936   1437601  19149376   2229049   5244100
   5958481  91853056  72760900  57896881  80425024  18062500    390625  15256836
   6594624  35354916  12595401  35450116  20259001   6708100  50140561   1974025
  94867600  75272976   7447441  20016676     27556     75625    571536  32661225
  43718544  51624225  38962564
```

(2)

```
Вкажіть кількість чисел,
    які треба згенерувати  50
  99824528  67980025  96040000    160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000  92160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000  92160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000  92160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000  92160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000  92160000   2560000  31360000  12960000  92160000
   2560000  31360000  12960000
```

(3)

```
Вкажіть кількість чисел,
    які треба згенерувати  50
  99825001  68062500    390625  15256836   6594624  35354916  12595401  35450116
  20259001   6708100  50140561   1974025  94867600  75272976   7447441  20016676
     27556     75625    571536  32661225  43718544  51624225  38962564  92640625
  41036836    135424   1833316  69438889  19254544   6477025  22752900  56685841
  47032164    103041   1060900    370881  13749264  56130064   1690000  47610000
  37210000   4410000  16810000  65610000  37210000   4410000  16810000  65610000
  37210000   4410000  16810000
```

(4)

```
Вкажіть кількість чисел,
    які треба згенерувати  50
  99825103  68079001    624100  38950081  90250000   6250000   6250000   6250000
   6250000   6250000   6250000   6250000   6250000   6250000   6250000   6250000
   6250000   6250000   6250000   6250000   6250000   6250000   6250000   6250000
   6250000   6250000   6250000   6250000   6250000   6250000   6250000   6250000
   6250000   6250000   6250000   6250000   6250000   6250000   6250000   6250000
   6250000   6250000   6250000   6250000   6250000   6250000   6250000   6250000
   6250000   6250000   6250000
```

(5)

```
Вкажіть кількість чисел,
    які треба згенерувати  50
  99825923  68211081   4452100  20439441  19307236   9437184  19105641   1115136
   1324801  10549504  30195025   3802500  64400625  16048036    230400   5308416
   9511056  26112100   1256641   6584356  34140649   1976836  95413824  17123044
   1512900  26306641   9400356  16024009     57600    331776  11002489       576
        25         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0
         0         0         0
```

Для одержання різних послідовностей випадкових чисел програму було виконано кілька разів і кожен раз як початкове значення взято поточний системний час (який, звичайно ж, змінювався) і згенеровано по 50 чисел. Як видно зі скриншотів, одержано: (1) усі різні числа (але при такому початковому значенні, починаючи з 69 числа числа по 4 циклічно повторюються); (2) 4 унікальні числа, а решта по 4 циклічно повторюються; (3) 40 унікаль-

них чисел, а решта по 4 циклічно повторюються; (4) 5 унікальних чисел і послідовність вироджується в те саме число; (5) 33 унікальних числа і послідовність вироджується в нуль.

У 1950-ті роки ряд вчених експериментували з методом середин квадратів, досліджуючи поведінку рядів згенерованих чисел як в десятковій системі числення, так і в двійковій. Вони встановили, що більшість початкових значень дають послідовності, які досить швидко вироджуються в короткі цикли, а деякі в нулі. Метод середин квадратів є дуже недосконалим і його тепер практично не використовують.

Метод середин добутків. Метод середин добутків досить подібний до методу середин квадратів, тільки в ньому для одержання наступного числа беруть добуток середин двох попередніх чисел (або двох початкових чисел).

Приклад 10.3.4. Генерація випадкових 8-значних десяткових чисел (якщо старші розряди в числах відсутні, то значення цих розрядів — нулі) з проміжку $[0;100000000)$ з використанням методу середин добутків. Для множення беруться 4 середні цифри останніх двох згенерованих чисел. Початковими випадковими числами беруться відповідно перші 4 і останні 8 цифр поточного системного часу.

Алгоритм мовою С:

```
/* Метод середин добутків генерації псевдовипадкових
   чисел. Генеруються восьмирозрядні числа */
#include <stdio.h>
...
  int n,   //кількість чисел
    r1, r2,   // чотирирозрядні середини двох попередніх чисел
    r,   // "випадкове число"
    i;   // лічильник циклу
  printf("Вкажіть кількість чисел,\n\
    які треба згенерувати  ");
  scanf("%d",&n);
  /* одержання перших двох чисел */
```

```
  r1=time(0);   // початкове 10-значне число
  r2=r1%10000;   // беремо останні 4 цифри числа
  r=r1%100000000;   // беремо останні 8 цифр числа
  printf ("%10d%10d", r2, r);
  /* Генерація наступних n чисел */
  for (i=1; i<=n; i++) {
    r1=r2;
    r2=(r/100)/10000;  // 4 середні цифри
    r=r1*r2;  // добуток середин двох чисел
    printf ("%10d",r);
  }
...
```

Для одержання різних послідовностей випадкових чисел програму було виконано кілька разів і кожен раз як початкові значення для першого числа взято останні 4 цифри, а для другого числа — останні 8 цифр поточного системного часу і згенеровано по 100 чисел.

(1)

```
Вкажіть кількість чисел,
    які треба згенерувати  100
      3221  11503221  16208072  10466560   9703200  32804280  56551344  44335546
  18496115  16644155  31953801  61434258  41413996  17971538  40210385  20430645
   9055518   2389830   2163390   6365434   5966982  35330526  31956045  31595800
  56958480  57101472   9718176   7281534  20214515   6038175    817245   3113532
   9275220   3123520   3398720   4923945  36835893  77219562  18345810   7590310
  20412574  24349875  14429250  15013416    575128    770634  44317206  24443432
  14064648   2864364   5583378  50414619  24183618   7612056  11236320  14461560
  10905245  41774980  70143948  11150811   2170012   2563600   9581200  32756432
  43961968  72758116  72921639  69866496  79847424  73418736  35480638  20122722
   5896962  11004963    439481    215306   9460282   9908106  41790762  71803467
  63524838  42162432   8522752   8488648  25539122  26340426  18350964  11944636
  33146014  13791160  11550060  43510500  28077500   3956375   7411325  39332619
  13679838  22610148  41474598  28949245  45039540   3749340
```

(2)

```
Вкажіть кількість чисел,
    які треба згенерувати  100
      3633  11503633  18295788  14891452  26358698  31965604  34626416  60485184
  30386664  18753966  29145774  10984323  14341251  33584316  19936316  54708009
  66290040  20532000  15428000  22769600  32938880  72250048  23470000  11750000
  35250000  18750000  18750000  56250000  18750000  18750000  56250000  18750000
  18750000  56250000  18750000  18750000  56250000  18750000  18750000  56250000
  18750000  18750000  56250000  18750000  18750000  56250000  18750000  18750000
  56250000  18750000  18750000  56250000  18750000  18750000  56250000  18750000
  18750000  56250000  18750000  18750000  56250000  18750000  18750000  56250000
  18750000  18750000  56250000  18750000  18750000  56250000  18750000  18750000
  56250000  18750000  18750000  56250000  18750000  18750000  56250000  18750000
  18750000  56250000  18750000  18750000  56250000  18750000  18750000  56250000
  18750000  18750000  56250000  18750000  18750000  56250000  18750000  18750000
  56250000  18750000  18750000  56250000  18750000  18750000
```

(3)
```
Вкажіть кількість чисел,
    які треба згенерувати  100
      3873  11503873  19512174  25799598  40942395  75336885  31736664  24808688
  59561476  45394804  22164072   6474720   7785080  37263950  20716150  18897879
  64291458  26161892   4714852  11565464  40414792  23447138  18541237  24197052
  10661640  13033520   2216360    724605  15673098  48765580  51518150  39660555
  34220505  14564025  12436200  24601680  26241792  14540672  13066302   3584178
   3872583  50962725  83995575  95836785  83293485  24548778  16098858   5421156
   4160468   6754444  12100576   7581720   5846085  49211820  17918280  19447476
  41080268   3588148   4716562  42137365   9837545  11498875  41774500  38632060
  48948400  59938880  89035792   3351516   1254855   8956220  24363976  34796118
  28970079  77221700  21504900  11193633   9774864  15000128      7748        77
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0
```

Як видно зі скриншотів, одержано: (1) усі різні числа (але при таких початкових значеннях, починаючи з 183 числа (при n=181) послідовність вироджується в 0); (2) 25 унікальних чисел, а решта, починаючи з 26 числа (n=24), по 3 циклічно повторюються; (3) 80 унікальних чисел, а починаючи з 81 числа (n=79), послідовність вироджується в нуль.

Порівняно з методом серединквадратів послідовність чисел, згенерована за допомогою методу серединдобутків, не так швидко вироджується в цикли чи нулі.

Метод перемішування. У цьому методі виконуються циклічні зсуви числа на 1/4 його довжини вліво і вправо і результати додаються. Якщо одержане після додавання число не буде поміщатися у відведені розряди, то від нього треба відкинути зайвий старший розряд — оскільки може з’явитися тільки один старший розряд (він є передбачуваним і може дорівнювати тільки 0 чи 1), то треба відкинути тільки його. Наприклад, якщо є 8-значне десяткове число 12345678, то після виконання зсувів на 2 позиції утворяться числа 34567812 і 78123456; їхня сума є 9-значним числом $34567812 + 78123456 = 112691268$, тому відкинемо старший розряд і матимемо число 12691268. В утворених після виконання зсувів числах ліва половина першого дорівнює правій половині другого і навпаки; крім того, число, яке є результатом додавання одержаних чисел має або рівні ліву й праву половини (наприклад, якщо при генерації 8-значних чисел було одержано 8-значне число і старший розряд не відкидався), або ліва половина

на 1 більша від правої (наприклад, якщо при генерації 8-значних чисел було одержано 9-значне число і відкинуто старший розряд). Метод дає змогу генерувати числа з кількістю значущих цифр, не меншою ніж половина використовуваних розрядів + 1 (наприклад, якщо генеруються 8-значні числа, то кількість значущих цифр буде не меншою від $4+1=5$).

Якщо алгоритм перемішування реалізувати для десяткового і для двійкового подання чисел, то при тому самому початковому числі буде одержано різні послідовності. При використанні двійкового подання у випадку виникнення переповнення старший біт, який вийде за розрядну сітку заданого типу даних, відкинеться автоматично.

Приклад 10.3.5. Генерація випадкових 8-значних десяткових і 32-розрядних двійкових чисел з використанням методу перемішування.

а) Для генерації випадкових 8-значних десяткових чисел (якщо старші розряди в числах відсутні, то значеннями цих розрядів є нулі) зсуви будемо виконувати на дві позиції вліво і на дві позиції вправо в десяткових поданнях чисел; після додавання залишатимемо 8 молодших десяткових розрядів. Початковим випадковим числом візьмемо останні 8 цифр поточного системного часу.

Програмна реалізація алгоритму:

```
/* Метод перемішування для генерації псевдовипадкових
   чисел. Генеруються восьмизначні десяткові числа */
#include <time.h>
...
  int n,   //кількість чисел
    i;   // лічильник циклу
  unsigned int r,   // згенероване "випадкове" число
    rl, rp;   // числа з циклічними зсувами вліво і вправо
  printf("  Генерація псевдовипадкових чисел\n\
         з використанням методу перемішування\n");
  printf("Вкажіть кількість чисел, \n\
         які треба згенерувати  ");
```

```
  scanf("%d",&n);
  /* Одержання першого числа */
  r=time(0);    // початкове 10-значне число
  r=r%100000000;    // беремо останні 8 цифр числа
  printf ("Початкове випадкове число %10u\n", r);
  /* Генерація наступних n 8-значних чисел:
  зсув здійснюється у десятковому поданні числа */
  printf("8-значні числа, одержані з використанням \n\
         зсувів у десятковому поданні\n");
  for (i=1; i<=n; i++) {
    rl=r/1000000+r%1000000*100;    // циклічний зсув вліво
    rp=r%100*1000000+r/100;    // циклічний зсув вправо
    r=rl+rp;  // сума двох зміщених чисел
    r=r%100000000;    // беремо останні 8 цифр числа
    printf ("%10u",r);
  }
...
```

Для одержання різних послідовностей випадкових чисел програму було виконано багато разів; при цьому початковими значеннями взято останні 8 цифр системного часу, а також інші довільні числа.

(1)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  100
Початкове випадкове число   11577561
8-значні числа, одержані з використанням
         зсувів у десятковому поданні
  18871886  73377336  74467446  93489348  97869786  73957394  90479046  94809480
  61896188  78237822  46564656  12931292  85258524  50705070  41014100   1820182
  64036402   6280628  56125612  25122512  24502450    490048  97009700   1940194
  88038802   6760676  52135212  26042604   8520852   4170416  33083308  16661666
  32333232  65646564  29312930  61586158  17231722  45344534  68906890  81378136
  74627462  25492548  97509750   1950194  89038902   6780678  56135612  26122612
  24522452   4490448  97089708  17941794  88358834  70767076  53415340  82068206
  13641364  28272826  53565356  13071306  13261326  52265226  53045304   9060906
  12181218  36243624  48724872  44974496  93899388  78877886  74577456  14491448
  97289728  57945794  89158914  30783078  56615660  22132212  25442544  88508850
   1770176  53035302   6060606  12121212  24242424  48484848  96969696  93939392
  86878686  74737472  46494648  97929792  85958594  90719070  42814280  61856184
  70237022  46404640  80928092  85618560
```

(2)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  100
Початкове випадкове число   11590645
8-значні числа, одержані з використанням
         зсувів у десятковому поданні
   4180417  35083508  16701670  40334032  65806580  61316130  62226222  45244524
  48904890  80978096  94619460  22892288  77457744  90549054   9810980  61196118
  38223822  44764476  52895288  78057804  10561056  12211220  41244124  48824882
  64976496  94299428  58885888  77177716  34543454   8690868  37173716  33743374
  48674866  33973396  93679366  34873486  73697368  38473846  93769376  53875386
  74077406  14481448  96289628  57925792  85158514  30703070  40614060  21812180
  61436142  86228622  45724572  44914490  81898188  78637862  26572656  13531352
   5270526  53105310  21062106  12421242  84248424  49684968  36993698  97739772
  46954694  89938992  86798678  58735872  46174616  33923392  84678466  34693468
  37693768  37753774  49754974  49994998  97999798  98959894  90979096  94819480
  62896288  78257824  50565056  13011300   1260126  52025202   5040504   8100810
  20162016  32403240  80648064  29612960  21592158  17431742  85348534  69706970
  41394138  77827782  65556554  10311030
```

(3)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  100
Початкове випадкове число   12345678
8-значні числа, одержані з використанням
         зсувів у десятковому поданні
  12691268  37253724  49744974  48994898  97979796  94959494  90899088  78817880
  62576256  14251424  49284928  56985698  97139712  26942694  88538852   6770676
  53135312  26062606  12521252   4250424  49084908  16981698  96339632  66926692
  85338532  66706670  41334132  65826582  65316530  62306230  61246124  49224922
  44984498  96899688  78937892  86578656  14731472  45294528  57905790  81158114
  30623062  24612460  21492148  97429742  85948594  89718970  42794278  57855784
  70157014  30403040  80608060  21612160  21432142  85428542  85708570  41714170
  41834182  65836582  66316630  62326232  65246524  49304930  60986098  97219720
  42944294  88858884  70777076  54415440  82088208  17641764  28352834  69566956
  13391338  77267726  53545354   9070906  13181318  36263626  52725272  45054504
   9900990  80198018  38603860  20772076  53415340  82068206  13641364  28272826
  53565356  13071306  13261326  52265226  53045304   9060906  12181218  36243624
  48724872  44974496  93899388  78877886
```

Як видно зі скриншотів, одержано: (1) у згенерованій послідовності циклів явно не видно, але при повнішому дослідженні виявлено, що вже перше згенероване число є початком циклу довжиною 244; (2) усі різні числа (при повнішому дослідженні виявлено, що унікальними є 621 число, після яких з числа 33283328 з номером 622 починається цикл довжиною 244); (3) початковим вибрано не пов’язане з системним часом число 12345678 і одержано 84 унікальних числа, а число 53415340 з номером 85 є початком циклу довжиною 244.

Крім поданих прикладів (1)-(3), для дослідження поведінки десяткових 8-значних чисел, які генеруються з використанням методу перемішування і зсуви застосовуються до десяткового подання чисел, програму було вико-

нано з початковими значеннями від 1 до 10 включно і одержано цикли довжиною 162, але для числа 11, як і для розглянутих прикладів, а також при багаторазовому виконанні програми з іншими початковими числами довжина циклу становила 244. Також усі згенеровані числа, крім, може, першого, є парними. При багаторазовому виконанні програми не було виявлено, щоб метод призводив до виродження послідовності в 0 чи якесь інше число.

б) Для генерації випадкових 32-розрядних двійкових чисел типу unsigned int (якщо старші розряди в числах відсутні, то значеннями цих розрядів є нулі), тобто чисел, менших від 4294967296, але з кількістю значущих двійкових цифр, не меншою ніж $16+1=17$, зсуви будемо виконувати на вісім розрядів вліво і на вісім розрядів вправо в двійкових поданнях чисел; початковим випадковим числом візьмемо поточний системний час.

Програмна реалізація алгоритму:

```
/* Метод перемішування для генерації псевдовипадкових
   чисел. Генеруються і 32-розрядні двійкові числа типу
   unsigned int (від 0 до 4294967295) */
#include <time.h>
...
  int n,   //кількість чисел
    i;   // лічильник циклу
  unsigned int r,   // згенероване "випадкове" число
    rl, rp;   // числа з циклічними зсувами вліво і вправо
  printf("  Генерація псевдовипадкових чисел\n\
  з використанням методу перемішування\n");
  printf("Вкажіть кількість чисел, \n\
         які треба згенерувати  ");
  scanf("%d",&n);
  /* одержання першого числа */
  r=time(0);   // початкове 10-значне число
  printf ("Початкове випадкове число %10u\n", r);
  /* генерація наступних n 32-розрядних чисел:
  зсув здійснюється у двійковому поданні числа */
```

```
  printf("32-розрядні числа, одержані з використанням\n\
         зсувів у двійковому поданні\n");
  for (i=1; i<=n; i++) {
    rl=(r<<8)+(r>>24);
    rp=(r>>8)+(r<<24);
    r=rl+rp;   // сума двох зміщених чисел; зайвий
               // розряд відкидається автоматично
    printf ("%13u",r);
    if (i%6==0)
      printf ("\n");
  }
...
```

Програму було виконано багато разів; при цьому початковими значеннями взято системний час, а також інші довільні числа.

(1)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  50
Початкове випадкове число 1611638804
32-розрядні числа, одержані з використанням
         зсувів у двійковому поданні
    604513288    273158216   2418053152   1092632864   1082278018     75564160
     17367304    285348098     69338146   1141392392    277352584    270602272
   1092632864   1082278018     75564160     17367304    285348098     69338146
   1141392392    277352584    270602272   1092632864   1082278018     75564160
     17367304    285348098     69338146   1141392392    277352584    270602272
   1092632864   1082278018     75564160     17367304    285348098     69338146
   1141392392    277352584    270602272   1092632864   1082278018     75564160
     17367304    285348098     69338146   1141392392    277352584    270602272
   1092632864   1082278018
```

(2)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  50
Початкове випадкове число 1611639507
32-розрядні числа, одержані з використанням
         зсувів у двійковому поданні
   3809141514    365303238   2351664170   1444435480    816591020   1482774624
   3249586608   1636000130     96667074   2232124682    369759754    338433068
   1479039016   1353732272   1621188768   1103184320   2172879234    100861442
     84673804    403314698    338695216   1613258792   1354780864   2158067872
   1107378688     25428356    134416386     84935952    537534474    339743808
   2150137896   1358975232     10616994   1140933632     25690504    268636162
     85984544   1074413578    343938176      2687016   1358975232     10616994
   1140933632     25690504    268636162     85984544   1074413578    343938176
      2687016   1358975232
```

(3)

```
  Генерація псевдовипадкових чисел
  з використанням методу перемішування
Вкажіть кількість чисел,
         які треба згенерувати  50
Початкове випадкове число   12345678
32-розрядні числа, одержані з використанням
         зсувів у двійковому поданні
    174197345   3272917780    696658310    206769234   2769855768    826945866
   2489488482   3307783464   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584   1368019338   346231970   1160332584   1368019338
    346231970   1160332584
```

Як видно зі скриншотів, одержано: (1) 3 унікальні числа, а далі повторюється цикл довжиною 9 чисел; (2) 31 унікальне число, а далі ідуть елементи циклу довжиною 9; (3) початковим вибрано не пов’язане з системним часом число 12345678 і одержано 8 унікальних чисел, а далі йдуть елементи циклу, в який входять всього 3 числа.

При застосуванні методу перемішування до 32-розрядних двійкових чисел і багаторазовому виконанні програми згенеровані послідовності містили дуже мало унікальних чисел — в основному 8-13, після яких повторювалися цикли по 9 чисел. Щоб одержати послідовність (2) з 31 унікальним числом довелося досить багато разів запускати програму на виконання. При заданні початкового числа 1 цикл довжиною 9 почався з першого згенерованого числа, тобто унікальних чисел у послідовності взагалі не було. Цикл з 3 числами одержано тільки в (3). Усі згенеровані числа, крім, може, першого, є парними При багаторазовому виконанні програми вироджування послідовності в число 0 чи якесь інше не було виявлено.

Таким чином серед розглянутих вище методів найкращі результати дав метод перемішування для десяткового подання чисел.

Лінійний конгруентний метод. Найпопулярнішою в даний час є дуже проста схема, запропонована Д. Г. Лехмером (D. H. Lehmer) у 1949 році, відповідно до якої випадкові числа обчислюють за формулою:

$$r_{n+1} = (ar_n + c) \bmod m,$$

де модуль $m > 0$, a, c, $r_0 \in [0, m)$ (при $c = 0$ формулу називають мультипліка-

тивною конгруентною формулою; при $c \neq 0$ — змішаною конгруентною формулою). Згенеровані таким чином числа можуть набувати значень від 0 до $(m-1)$. Цикл не може містити більше, ніж m чисел. Ця формула дає досить хороші випадкові числа.

Проте і ця схема потребує підбору параметрів, щоб при генерації одержувати прийнятні значення (достатньо випадкові, рівномірно розподілені, незалежні) і прийнятні довжини циклів.

Приклад 10.3.6. При $m=10$, $a=c=r_0=7$ буде одержано таку послідовність чисел: $6,9,0,7,6,9,0,7,...$, розподіл значень в якій не є рівномірним і цикл має довжину, рівну всього 4. Виконання поданої нижче програми підтверджує це.

```
/* Лінійний конгруентний метод для генерації
   псевдовипадкових чисел. Генеруються 32-розрядні
   числа типу unsigned int (від 0 до 4294967295) */
#include <time.h>
...
  unsigned long long int m,   //значення модуля
    n,   //кількість чисел
    i;   // лічильник циклу
  int  a, c;   // параметри методу
  unsigned int r;   // згенероване "випадкове" число
...
  /* генерація наступних n 32-розрядних чисел */
  printf("Згенеровані 32-розрядні числа\n");
  for (i=1; i<=n; i++) {
    r=(a*r+c)%m;   // r=(a*r+c) при m=2^32
    printf ("%5u",r);   // при великих числах %12u
    if (i%6==0)
      printf ("\n");
  }
...
```

```
  Генерація псевдовипадкових чисел
  з використанням лінійного конгруентного методу
        r(n+1) = (a*r(n)+c) mod m
Задайте значення
        модуля m  10
        множника a  7
        приросту c  7
Задайте початкове число r0  7
Вкажіть кількість чисел,
         які треба згенерувати  15
Виконується генерація:
  n= 15
  початкове число 7
  формула методу r(n+1) = (7*r(n)+7) mod 10
Згенеровані 32-розрядні числа
    6    9    0    7    6    9
    0    7    6    9    0    7
    6    9    0
```

При виборі модуля m треба врахувати те, що період не може містити більше ніж m елементів. Тому m треба вибирати якомога більшим. Крім того, m має бути таким, щоб значення $(ar_n + c) \bmod m$ обчислювалося досить швидко, оскільки мультиплікативна операція знаходження остачі від ділення виконується досить довго. Компенсувати цей недолік ділення можна, вибравши значення m відповідно до довжини машинного слова. Якщо комп’ютер є k-розрядним, то найбільше значення, яке можна одержати, буде рівним $2^k - 1$, тоді можна взяти $m = 2^k$ (беззнакове ціле число). При цьому, якщо працювати з беззнаковими величинами, то дію ділення навіть не треба буде виконувати, оскільки при виході за розрядну сітку старші розряди будуть автоматично втрачатися. Якщо k взяти меншим ніж розрядність комп’ютера, то знаходження остачі від ділення на $m = 2^k$ зведеться до виконання порозрядної логічної операції &, яка виконується дуже швидко. Також часто вибирають найбільше просте число m, менше від 2^k, наприклад, часто беруть одне з чисел Мерсенна $(2^{31} - 1) = 2147483647$, яке є простим.

Як було сказано вище, довжина періоду має бути якомога більшою. Довжина періоду залежить не тільки від величини модуля m, а також і від величини множника a, і приросту c, і від співвідношень між ними.

Теорема. Лінійна конгруентна послідовність, визначена формулою $r_{n+1} = (ar_n + c) \bmod m$, має період довжини m тоді і тільки тоді, коли:

— числа m і c взаємно прості (не мають спільного дільника, крім 1);

— число $b = a - 1$ кратне p для кожного простого p, яке є дільником m;

— число b кратне 4, якщо m кратне 4.

Із теореми випливає, що якщо значення $c = 0$, то максимального періоду досягнути неможливо, але при цьому пришвидшуються обчислення. Крім того, щоб при $c = 0$ послідовність не виродилася в 0, треба, щоб r_n і m були взаємно простими числами для всіх n; тоді максимально можлива довжина періоду буде визначатися функцією Ейлера $\varphi(m) = m \cdot \prod_{i=1}^{k}\left(1 - \frac{1}{p_i}\right)$ — кількістю взаємно простих з m чисел, які розміщуються між 0 і m (тут $m = p_1^{d_1} \cdot p_2^{d_2} \cdot \ldots \cdot p_k^{d_k}$ $(i = \overline{1,k})$, p_i — прості множники, степені d_i множників не враховуються. Наприклад, якщо взяти $c = 0$ і $m = 2^{32}$, то $\varphi(2^{32}) = 2^{32} \cdot \left(1 - \frac{1}{2}\right) = 2^{31}$, тобто максимальна довжина періоду зменшиться вдвічі.

Задачу вибору значень параметрів r_0, m, a, c для генерації надійних випадкових чисел за рекурентною формулою $r_{n+1} = (ar_n + c) \bmod m$ досліджували протягом десятиліть багато вчених-практиків і теоретиків. Дональд Кнут [20, с. 213-214], підсумовуючи ці дослідження, формулює вимоги щодо вибору величин r_0, m, a, c; основними є такі:

— число r_0 можна взяти будь-яке, наприклад, системний машинний час;

— число m має бути великим, хоч би 2^{30}; зручно, щоб воно відповідало довжині машинного слова;

— якщо m є степенем 2, то a треба вибрати таким, щоб $a \bmod 8 = 5$; якщо m є степенем 10, то a треба вибрати таким, щоб $a \bmod 200 = 21$;

— множник a бажано вибрати між $0{,}01m$ і $0{,}99m$ і його десяткові чи двійкові цифри не повинні мати просту регулярну структуру, наприклад, можна вибрати число $a = 3141592621$;

— число c не повинно мати спільного множника з m, якщо m відповідає довжині машинного слова. Можна вибрати $c = 1$ чи $c = a$. Вибір $c = 0$ при $m = 2^k$ зменшує довжину періоду вдвічі;

— можна генерувати не більше $m/1000$ чисел, після чого треба змінити схему генерації (наприклад, взяти нове значення a).

При виборі параметрів треба враховувати те, що для практичного використання генератор випадкових чисел повинен генерувати послідовність, близьку до випадкової. Для цього треба, щоб період був досить великим, навіть якщо генерується незначна кількість чисел. Дональд Кнут наводить приклади найпопулярніших генераторів [20, с. 129-133], розроблених рядом вчених; деякі з них:

— при $m = 2^{35} = 34359738368$, $a = 5^{13} = 1220703125$ (успішно використовували на початку 1950-х років на комп'ютерах, наприклад DECsystem 20, розрахованих на ефективну роботу з 35-розрядними даними);

— при $m = 2^{31} = 2147483648$, $a = 2^{16} + 3 = 65539$, r_0 — непарне, $c = 0$ (генератор RANDU використовувався в 1990-і роки на комп'ютерах, але, як виявилося, він не відповідає критеріям випадковості);

— при $m = 2^{31} - 1 = 2147483647$: $a = 62089911$ (найкращий щодо статистичного спектрального критерію); $a = 7^5 = 16807$, $c = 0$ (міститься в міжнародній математичній бібліотеці програм IMSL); $a = 48271$, $c = 0$;

— при $m = 2^{31} - 249 = 2147483399$, $a = 40692$;

— при $m = 2^{32} = 4294967296$, $a = 1 \bmod 4$, зокрема: $a = 1664525$, $c = 1013904223$ (параметри визначили за допомогою комп'ютера, використовується в бібліотеках мов програмування C для Unix, 1C, Пролог); $a = 69069$, $c = 1$ (бібліотека машини Vax); $a = 2650845021$ (найкращий серед генераторів при $m = 2^{32}$);

— при $m = 2^{48} = 281474976710656$: $a = 31167285$, (параметри визначили за допомогою комп'ютера); $a = 44485709377909$, $c = 0$ (бібліотека суперкомп'ютера Cray X-MP);

— при $m = 2^{64} = 18446744073709551616$, $a = 6364136223846793005$.

Встановлено [16, с.181-192], що при $c = 0$ і $m = 2^k$ (де k — будь-яке число) найбільший період $m/4$ буде при $a = 3 \bmod 8$ і при $a = 5 \bmod 8$ (при цьому

треба врахувати, що послідовність буде мати вигляд більш випадкової, якщо значення a велике) і непарному початковому значенні r_0. Якщо $m = 2^k$, число c непарне і $a = 1 \bmod 4$ (наприклад, це числа $a = 69069$ і $a = 71365$), то період буде мати довжину m (при цьому період починається з першого згенерованого числа).

Слід зауважити, що в Інтернеті широко розповсюджено таку неперевірену інформацію: при $m = 2^{31} - 1 = 2147483647$, $a = 5^{13} = 1220703125$, $c = 7$ і $r_0 = 7$ буде одержано цикл з 7 мільйонами чисел. З цими даними було виконано програму з прикладу 10.3.6 і встановлено, що згенерована послідовність має 42351 унікальне число, після яких починається цикл довжиною 139145; таким чином можна одержати тільки $42351 + 139145 = 181496$ різних чисел, а не 7 мільйонів.

Лінійний конгруентний метод часто реалізують у сучасних системах програмування. У багатьох системах програмування використовують конгруентні генератори відповідно до алгоритму, запропонованого Національним бюро стандартів США, — генерується послідовність з довжиною періоду $2^{24} = 16777216$, яка має досить хороші статистичні характеристики. Але для криптографії така довжина періоду надто мала. Крім того, було доведено, що послідовності, згенеровані конгруентними генераторами, не є криптографічно стійкими — якщо є частина такої послідовності певної довжини, то всі параметри моделі можна відновити. Крім криптографії, цей метод також майже непридатний для застосування в статистиці, де вимоги до випадковості значно вищі. Проте для прикладних задач моделювання випадкової поведінки об'єктів лінійний конгруентний метод генерації випадкових чисел використовують досить широко.

Для покращення статистичних властивостей генерованої числової послідовності в багатьох генераторах використовують тільки частину розрядів результату. Так, у стандарті ISO/IEC 9899 мови C подано приклад функції rand, у якій відкидаються один старший і 16 молодших розрядів згенерованого числа (тут у формулі $r_{n+1} = (ar_n + c) \bmod m$ використано параметри $m = 2^{15} = 32768$, $r_0 = 1$, $a = 1103515245$, $c = 12345$):

```
#define RAND_MAX 32767
static unsigned long int next=1;
void srand(unsigned int seed) {
  next = seed;
}
int rand(void) {
  next=next*1103515245+12345;
  return (unsigned int)(next/65536)%(RAND_MAX+1);
}
```

Лінійний конгруентний метод часто реалізують у сучасних системах програмування, проте він майже непридатний у статистиці чи криптографії, в яких вимоги до випадковості значно вищі.

Інші методи генерації випадкових чисел. Лінійний конгруентний метод можна узагальнити до квадратичного конгруентного методу, відповідно до якого обчислення проводять за формулою:

$$x_{n+1} = (dx_n^2 + ax_n + c) \bmod m .$$

Щоб генерована послідовність мала максимальний період довжини *m*, умови щодо вибору значень *d*, *a*, *c* не жорсткіші, ніж у лінійному випадку.

Серед багатьох схем, в яких використовують не одне, а кілька попередніх чисел, можна вказати схему Фібоначчі із запізненням:

$$r_n = (r_{n-24} + r_{n-55}) \bmod m ,$$

де m — непарне, r_0, …, r_{54} — будь-які цілі не всі парні числа. На комп’ютері метод ефективно можна реалізувати за допомогою циклічного масиву. Генеровані випадкові числа мають хороші статистичні характеристики, довжина періоду при $m = 2^k$ точно дорівнює $2^{k-1} \cdot (2^{55} - 1)$. Крім зміщення (24, 55), для одержання максимальних періодів можна використати зміщення (37, 100), (83, 258), (273, 607) та ін. Генератор працює досить швидко, оскільки у формулі немає операції множення. Цей метод використовують у системі Matlab.

Обернена конгруентна послідовність, яка визначається формулою:

$$r_{n+1} = (ar_n^{-1} + c) \bmod 2^k,\ r_0 = 1,\ k \ge 3,$$

при $a \bmod 4 \equiv 1$ і $c \bmod 4 \equiv 2$ має період 2^{k-1}.

Алгоритм Блюма-Блюма-Шуба (BBS) — генератор з квадратичною остачею, запропонований у 1986 році, дає можливість генерувати двійкові послідовності b_n:

$$r_{n+1} = r_n^2 \bmod m, \quad b_n = r_n \bmod 2,$$

де $m = p \cdot q$, p і q — великі прості числа виду $(3 + 4 \cdot i)$; початкове число r_0 має бути взаємно простим з m. Особливістю цього методу є те, що для одержання r_n не обов'язково генерувати всі попередні числа; його можна обчислити за формулою:

$$r_n = r_0^{2^n \bmod ((p-1)\cdot(q-1))} \bmod m.$$

Згенерована послідовність задовольняє всім статистичним критеріям випадковості. Алгоритм було розроблено спеціально для криптографії. Проте для моделювання, виконання обчислень в реальному часі, а також при потоковому шифруванні він не підходить, оскільки працює досить повільно.

Якщо є згенеровані (або генеруються) послідовності випадкових чисел $0 \le x_n < m$ і $0 \le y_n < m' < m$, їхні початкові числа x_0 і y_0 вибрано незалежно, то для підвищення випадковості і для збільшення періоду можна, наприклад, їх об'єднати, скориставшись формулою:

$$r_n = (x_n - y_n) \bmod m,$$

де початкові значення x_0.і y_0 вибираються незалежно. Наприклад, при

$$x_{n+1} = 48271 x_n \bmod (2^{31} - 1),$$

$$y_{n+1} = 40692 y_n \bmod (2^{31} - 249),$$

$$r_n = (x_n - y_n) \bmod (2^{31} - 1)\ .$$

послідовність r_n має період довжиною $(2^{31} - 2) \cdot (2^{31} - 250) / 62 \approx 7 \cdot 10^{16}$.

Якщо послідовності генеруються більшою кількістю схем, то одна із послідовностей може керувати об'єднанням інших. Такий підхід дає можливість отримувати послідовності з довгими періодами.

Методів алгоритмічної генерації випадкових чисел є досить багато. Вони модифікуються і вдосконалюються, розробляються нові методи, оскільки проблема генерації випадкових чисел в наш час є досить актуальною.

10.4. Генерація кодів Грея

Коди Грея розглядалися на початку курсу як приклад непозиційної системи кодування цілих невід'ємних чисел (див. п. 3.3). Ці коди можуть бути як бінарними, так і небінарними. У кодуванні Грея два послідовні коди відрізняються лише значенням одного розряду, при цьому перший і останній коди теж є сусідніми.

Вони мають досить широке застосування на практиці, насамперед при передачі інформації каналами зв'язку, оскільки дають змогу зменшити кількість помилок від перешкод. Їх використовують у сучасній алгоритмізації при кодуванні генетичних ознак в генетичних алгоритмах. Надалі (див. п. 10.5) буде розглянуто застосування кодів Грея при генерації комбінаторних множин.

Існують різні способи *переходу до стандартних кодів Грея*. Один з них полягає в тому, що якщо число подано в двійковій системі числення $b = b_n b_{n-1} b_{n-2} .. b_1 b_0$ (а саме так подаються числа в комп'ютері), то код Грея $g = g_n g_{n-1} g_{n-2} ... g_1 g_0$ можна знайти, використовуючи операцію $\oplus$ порозрядного (без переносу в старші розряди) додавання за модулем 2:

$$g_i = b_i \oplus b_{i+1}, \text{ або } g(b) = b \oplus \lfloor b/2 \rfloor,$$

де $0 \le i \le n$, $b_{n+1} = 0$, $\lfloor * \rfloor$ — заокруглення до меншого. Щоб обчислити значення $\lfloor b/2 \rfloor$, треба у двійковому поданні числа відкинути наймолодший розряд, тобто здійснити зсув на один розряд вправо.

Таке перетворення легко реалізувати програмно; крім того, воно буде виконуватися дуже швидко за рахунок швидкого виконання порозрядних операцій. Часова складність алгоритму — $O(n)$, ємнісна — $O(1)$.

Приклад 10.4.1. Програма, в якій реалізовано перетворення двійкового подання числа в бінарний стандартний код Грея з використанням порозрядних операцій, має вигляд:

```
/* Одержання послідовності бінарних кодів Грея
   із двійкових подань чисел */
...
#define N 10 // але не більше 32
void code_Gray(unsigned long long int i, unsigned int *gr);
void vyvid_Gray(unsigned long long int i, int n, unsigned int gr);
int main() {
/* Знаходження всіх n-розрядних кодів Грея (їх буде 2**n)
   з використанням побітових операцій */
  int n;  // кількість розрядів коду Грея
  unsigned int gr;  // код Грея (максимум 32 розряди)
  unsigned long long int ng,  // кількість всіх n-розрядних кодів
    i;  // лічильник циклу (номер коду)
  printf("\tЗнаходження всіх n-розрядних кодів Грея\n");
  printf("\t(з використанням в алгоритмі побітових операцій)\n");
  printf("Вкажіть розрядність коду Грея n (0<=n<=%d<=32) \n",N);
  while(scanf("%d",&n)) {
    if (!(n>=0 && n<=N && n<=32))
      printf("\tПомилкові дані\n");
    else { /* обчислення кількості кодів */
      ng=1; // при n=32 максимальне ng=2**32=4294967296 -
            // це вже число типу long long int
      for (i=1; i<=n; i++)
        ng*=2;
     /* одержання і вивід кодів Грея */
      for (i=0; i<ng; i++) {
        code_Gray(i, &gr);   // одержання коду
        vyvid_Gray(i, n, gr);   // виведення коду
      }
      printf("\n");
```

```
    }
  }
...
}
/* Одержання коду Грея з використанням порозрядних операцій */
void code_Gray(unsigned long long int i, unsigned int *gr) {
  *gr=(unsigned int)i^((unsigned int)i>>1);
}
/* Виведення розрядів коду Грея – правих n розрядів цілого числа */
void vyvid_Gray(unsigned long long int i, int n, unsigned int gr) {
  int j;  // лічильник циклу для доступу до окремих розрядів коду
  if (i!=0 && i%4==0)   // умова виведення по 4 числа в рядку
    printf("\n");
  printf("\t%4llu:  ",i);
  for (j=n-1; j>=0; j--) {
    if ((gr&(1<<j)))   // вибір j-го розряду справа
      printf("%d",1);
    else
      printf("%d",0);
  }
}
```

```
        Знаходження всіх n-розрядних кодів Грея
        (з використанням в алгоритмі побітових операцій)
Вкажіть розрядність коду Грея n (0<=n<=10<=32)
0
           0:
1
           0:  0          1:  1
2
           0:  00         1:  01         2:  11         3:  10
3
           0:  000        1:  001        2:  011        3:  010
           4:  110        5:  111        6:  101        7:  100
4
           0:  0000       1:  0001       2:  0011       3:  0010
           4:  0110       5:  0111       6:  0101       7:  0100
           8:  1100       9:  1101      10:  1111      11:  1110
          12:  1010      13:  1011      14:  1001      15:  1000
5
           0:  00000      1:  00001      2:  00011      3:  00010
           4:  00110      5:  00111      6:  00101      7:  00100
           8:  01100      9:  01101     10:  01111     11:  01110
          12:  01010     13:  01011     14:  01001     15:  01000
          16:  11000     17:  11001     18:  11011     19:  11010
          20:  11110     21:  11111     22:  11101     23:  11100
          24:  10100     25:  10101     26:  10111     27:  10110
          28:  10010     29:  10011     30:  10001     31:  10000
кінець
```

Для *декодування коду Грея* треба код спочатку перетворити у звичайне двійкове число, а потім одержати відповідне десяткове. Можна скористатися формулою:

$$b_i = \bigoplus_{j=i}^{n} g_j ,$$

де $i = n-1, n-2, ..., 0$.

Часова складність алгоритму — $O(n^2)$, ємнісна — $O(1)$.

Приклад 10.4.2. Програму, в якій реалізовано перетворення коду Грея у двійкове число з використанням порозрядних операцій, подано нижче.

У програмі генерується ціле число (максимальне згенероване число визначається константою RAND_MAX = 32767, мінімальне число — 0), його двійкове подання інтерпретується як двійковий код Грея, який переводиться у відповідне двійкове число. Зауважимо, що число 32767 є максимальним для типу двобайтових даних short int і містить максимум 15 розрядів (без специфікатора unsigned один розряд відводиться під знак числа).

```
/* Переведення стандартних двійкових кодів Грея у звичайні
   двійкові числа і відповідні двійковим числам десяткові числа.
   Для цього генерується випадкове число типу int (функція rand
   генерує цілі числа від 0 до 32767), його двійкове подання
   інтерпретується як код Грея. Код Грея перетворюється в
   двійкове число */
...
#include <time.h>     // для srand(time(0));
#define N 15 // але не більше 15
void vyvid(int n, int gr);
void decode_Gray(int gr, int *b);
int main() {
  int n;  // кількість розрядів коду Грея (не більше 15)
  int gr, b,  //код Грея і двійкове число (до 15 розрядів)
    ng,  // кількість всіх n-розрядних кодів
    i;  // лічильник циклу
```

```
  srand(time(0));
  printf("\tГенерація випадкових n-розрядних кодів Грея\n");
  printf("\ti переведення їх у звичайні двійкові й десяткові числа\n");
  printf("Вкажіть розрядність коду Грея n (0<=n<=%d<=15) \n",N);
  while(scanf("%d",&n)) {    // цикл працює до введення букви
    if (!(n>=0 && n<=N && n<=15))
      printf("\tПомилкові дані\n");
    else {
      /* Обчислення кількості кодів */
      ng=1; // при n=15 максимальне ng=32768 - це число типу int
      for (i=1; i<=n; i++)
        ng*=2;
      /* Одержання випадкового n-розрядного коду Грея */
      gr=rand()%ng;
      printf("  Код Грея ");
      vyvid(n, gr);
      decode_Gray(gr, &b);
      printf(", двійкове число ");
      vyvid(n, b);
      printf(" (десяткове %u)\n", b);
    }
  }
...
}
void decode_Gray(int gr, int *b) {
  /* Одержання двійкового числа за стандартним кодом Грея */
  for (*b=0; gr; gr>>=1)
    *b^=gr;
}
void vyvid(int n, int a) {
  /* Виведення двійкових розрядів числа */
  int i;  // лічильник циклу для доступу до окремих
          // розрядів коду
```

```
  for (i=n-1; i>=0; i--) {
    if ((a&(1<<i)))
      printf("%d",1);
    else
      printf("%d",0);
  }
}
```

```
        Генерація випадкових n-розрядних кодів Грея
        і переведення їх у звичайні двійкові й десяткові числа
Вкажіть розрядність коду Грея n (0<=n<=15<=15)
16
        Помилкові дані
15
  Код Грея 101011111110111, двійкове число 110010101011010 (десяткове 25946)
2
  Код Грея 01, двійкове число 01 (десяткове 1)
2
  Код Грея 10, двійкове число 11 (десяткове 3)
3
  Код Грея 011, двійкове число 010 (десяткове 2)
4
  Код Грея 1101, двійкове число 1001 (десяткове 9)
5
  Код Грея 10011, двійкове число 11101 (десяткове 29)
5
  Код Грея 11101, двійкове число 10110 (десяткове 22)
7
  Код Грея 1111111, двійкове число 1010101 (десяткове 85)
10
  Код Грея 1110101000, двійкове число 1011001111 (десяткове 719)
кінець
```

У недвійкових (небінарних) кодах Грея $g = g_n g_{n-1} \ldots g_1 g_0$ кожен розряд може мати свою основу, тобто кожна координата g_i $(n+1)$-вимірного вектора $(g_n, g_{n-1}, \ldots, g_1, g_0)$ може набувати значень у діапазоні з іншою верхньою межею (від 0 до d_i, $i = \overline{0,n}$). Кількість усіх таких кодів дорівнює $\prod_{i=0}^{n}(d_i + 1)$. У такому коді, як і в бінарному, найшвидше змінюється остання координата g_0 (відповідає наймолодшому розряду), почергово спочатку зростаючи, а потім спадаючи, після неї аналогічно змінюється передостання координата g_1 і т.д., перша координата g_n змінюється найповільніше. Наприклад, якщо в коді Грея $g = g_2 g_1 g_0$ координата g_2 є бінарною, g_1 — п’ятірковою, а g_0 — тернарною, то буде одержано таку послідовність кодів (їх є $2 \cdot 5 \cdot 3 = 30$):

000, 001, 002, 012, 011, 010,

020, 021, 022, 032, 031, 030,

040, 041, 042, 142, 141, 140,

130, 131, 132, 122, 121, 120,

110, 111, 112, 102, 101, 100.

Розглянемо *алгоритм генерації недвійкових рефлексивних кодів Грея*. В алгоритмі при генерації *n*-розрядних кодів $g = g_{n-1}g_{n-2}...g_1g_0$ буде використано такі чотири масиви: значень розрядів коду g[0], g[1], …, g[n-1], основ розрядів d[0], d[1], …, d[n-1] ($d_i \geq 2$), керування діями f[0], f[1], …, f[n-1], f[n], напрямків зміни значень розрядів o[0], o[1], …, o[n-1], елементи якого почергово набувають значень +1 (вказує на зміну значень від 0 до d_i), –1 (вказує на зміну значень від d_i до 0). Елементи масиву керування діями f спочатку ініціюються послідовними значеннями від 0 до *n*, але поступово їхні значення змінюються так, щоб, коли буде пройдено діапазон певної позиції, змінювалося значення сусідньої (наступної чи попередньої) позиції коду. Елемент f[0], одержаний на поточному кроці, визначає номер елемента масиву, який має бути змінений на наступному кроці; і якщо завдяки поступовим змінам елемент f[0] набуде значення *n*, то це означає, що побудовано всі коди Грея і алгоритм закінчує роботу.

Алгоритм генерації недвійкових рефлексивних кодів Грея:

K1. Виконати ініціалізацію масивів: $g_i = i$, $o_i = 1$ при $i = \overline{0, n-1}$, $f_i = i$ при $i = \overline{0, n}$. Ввести значення d_i при $i = \overline{0, n-1}$.

K2. Одержано поточний код Грея (вивести його чи використати по-іншому).

K3. $i = f_0$, $f_0 = 0$.

K4. Якщо $i = n$, то перейти на К6, інакше $g_i = g_i + o_i$.

K5. Якщо $g_i = i$ або $g_i = d_i - 1$, то $o_i = -o_i$, $f_i = f_{i+1}$, $f_{i+1} = i + 1$. Перейти на К2.

K6. Кінець.

Часова складність алгоритму одержання окремого n-розрядного коду становить $O(n)$, ємнісна складність за рахунок використання пам'яті для зберігання допоміжних значень у масивах d, f, o — теж $O(n)$.

Приклад 10.4.3. Програмна реалізація мовою С алгоритму генерації недвійкових рефлексивних кодів Грея:

```
/* Недвійкові рефлексивні коди Грея. Знаходження всіх
   n-розрядних кодів Грея з різними основами розрядів */
...
#define N 7  // кількість розрядів не обмежується
int code_Gray_d(char gr[], char d[], char o[], char f[], int n);
void vyvid_Gray_d(char gr[], int n, int k);
int main() {
  int n,  // кількість розрядів коду Грея
    i,  // лічильник циклу - номер розряду
    ind;  // індикатор помилкового вводу
  char gr[N],  //розряди коду Грея
    d[N],  // верхня межа діапазону значення розряду не більше 9
    o[N],  // напрямок зміни значення розряду
    f[N+1];  // керування розрядами
  printf("\tЗнаходження всіх n-розрядних недвійкових рефлексивних кодів\n\
        Грея з різними діапазонами (від 0 до вказаної для кожного розряду\n\
        верхньої межі) значень розрядів\n");
  printf("Вкажіть розрядність коду Грея n (1<=n<=%d) \n",N);
  while(scanf("%d",&n)) {
    if (!(n>=1 && n<=N))
      printf("\tПомилкові дані\n");
    else {
      /* Ініціалізація масивів */
      printf("  Вкажіть верхню межу (від 1 до 9) діапазону значень розрядів\n");
      for (i=0; i<n; i++) {
        gr[i]=0;
        ind=0;
        do {
```

```
          if (ind)
            printf("\tПомилка\n");
          printf("  d[%d]=",i);
          scanf("%d",&d[i]);
        } while (ind=(d[i]<1 || d[i]>9));
        o[i]=1;
        f[i]=i;
      }
      f[n]=n;
      vyvid_Gray_d(gr, n, 1);  // виведення першого коду Грея
      /* Одержання і вивід наступного коду Грея */
      while (code_Gray_d(gr, d, o, f, n))
        vyvid_Gray_d(gr, n, 0);
    }
    printf("\n");
  }
...
}
int code_Gray_d(char gr[], char d[], char o[], char f[], int n){
/* Одержання окремого недвійкового (з діапазонами розрядів)
   коду Грея з використанням масиву розрядів.
   Масиви ініціалізовано:
     розряди коду Грея gr - всі 0;
     основи розрядів d - задає користувач;
     напрямки зміни значень розрядів o - всі 1;
     керування діями f - від 0 до n. */
  int ind,  // індикатор продовження генерації
    i;  //номер розряду, значення якого змінюється
    ind=1;
  /* Вибір номера поточного розряду */
  i=f[0];
  f[0]=0;
  if (i==n) // чи не побудовано останній код?
    ind=0;
```

```
  else  // зміна значення розряду
    gr[i]=gr[i]+o[i];
  /* Чи не пройдено весь діапазон? */
  if (gr[i]==0 || gr[i]==d[i]) {
    /* Зміна напрямку і значень керуючих елементів */
    o[i]=-o[i];
    f[i]=f[i+1];
    f[i+1]=i+1;
  }
  return ind;
}
```

```
        Знаходження всіх n-розрядних недвійкових рефлексивних кодів
        Грея з різними діапазонами (від 0 до вказаної для кожного розряду
        верхньої межі) значень розрядів
Вкажіть розрядність коду Грея n (1<=n<=7)
0
        Помилкові дані

3
  Вкажіть верхню межу (від 1 до 9) діапазону значень розрядів
  d[0]=1
  d[1]=0
        Помилка
  d[1]=1
  d[2]=1
         0:  000         1:  001         2:  011         3:  010
         4:  110         5:  111         6:  101         7:  100
3
  Вкажіть верхню межу (від 1 до 9) діапазону значень розрядів
  d[0]=1
  d[1]=1
  d[2]=5
         0:  000         1:  001         2:  011         3:  010
         4:  110         5:  111         6:  101         7:  100
         8:  200         9:  201        10:  211        11:  210
        12:  310        13:  311        14:  301        15:  300
        16:  400        17:  401        18:  411        19:  410
        20:  510        21:  511        22:  501        23:  500
3
  Вкажіть верхню межу (від 1 до 9) діапазону значень розрядів
  d[0]=1
  d[1]=5
  d[2]=1
         0:  000         1:  001         2:  011         3:  010
         4:  020         5:  021         6:  031         7:  030
         8:  040         9:  041        10:  051        11:  050
        12:  150        13:  151        14:  141        15:  140
        16:  130        17:  131        18:  121        19:  120
        20:  110        21:  111        22:  101        23:  100
кінець
```

```
void vyvid_Gray_d(char gr[],int n, int k) {
/* Виведення розрядів коду Грея - елементів масиву
   у зворотному порядку */
  static unsigned long long int i;  // номер коду
  int j;  // лічильник циклу для доступу до елементів-розрядів коду
  if(k)
    i=0;
  if (i!=0 && i%4==0)
    printf("\n");
  printf("\t%4llu:  ",i);
  for (j=n-1; j>=0; j--)
    printf("%d",gr[j]);
  i++;
}
```

Подана в прикладі 10.4.3 програма може генерувати також бінарні коди Грея, якщо всі верхні межі діапазону значень розрядів задати рівними 1. Таким чином дана програма реалізує ще один метод генерації бінарних стандартних кодів Грея. Відмінністю реалізованого тут алгоритму від алгоритму, реалізованого в прикладі 10.4.1 є те, що тут може генеруватися тільки послідовність кодів, тобто наступний код будується на основі попереднього, а реалізований раніше алгоритм дає можливість зразу одержати код Грея для окремого двійкового числа.

Незалежно, які коди Грея розглядати — бінарні чи небінарні, кодові відстані між двома сусідніми кодами завжди дорівнюють 1. У цьому можна пересвідчитися, проаналізувавши подані у скриншотах прикладів 10.4.1 і 10.4.3 результати.

10.5. Генерація комбінаторних множин

Ряд практичних задач (наприклад, задача комівояжера, обробка природномовного тексту, задача про повні підграфи, про існування в графі гамільтонового циклу) можна розв’язати шляхом перебору ріноманітних комбінацій

даних, що, як правило, потребує експоненційного $O(k^n)$ чи навіть факторіального $O(n!)$ часу. Розв'язання таких задач можна звести до задач генерації різноманітних комбінаторних множин, зокрема: утворення всіх підмножин і підмножин з певними властивостями, перестановки елементів множин у різному порядку тощо.

Генерація всіх підмножин множини. Якщо множина містить *n* елементів, то всіх підмножин (включаючи невласні — порожню і саму множину) буде 2^n. Для одержання всіх підмножин даної множини можна застосувати різні підходи.

Якщо числа від 0 до $2^n - 1$ подати в двійковій системі числення, то наявність 0 в певній позиції відповідає відсутності певного елемента множини, а наявність 1 — його присутності в підмножині. Такий бітовий шаблон називають *маскою*. Отже, перебираючи двійкові подання чисел від 0 до $2^n - 1$, будемо одержувати маски різних підмножин.

Розглянемо підхід з *використанням побітових операцій* — побітового логічного «і» і зсуву вліво. Будемо послідовно розглядати двійкові подання десяткових чисел від 0 до $2^n - 1$ і визначати розряди, в яких розміщуються двійкові одиниці. При цьому справа розміщуватиметься наймолодший розряд, зліва — найстарший. Наприклад, якщо розглядається 5-лементний масив, то числу (масці) $0_{10} = 0_2 = 00000_2$ відповідає порожня підмножина, числу $25_{10} = 11001_2$ відповідають елементи масиву з номерами 1, 4 і 5, а числу $31_{10} = 11111_2$ відповідають елементи з номерами 1, 2, 3, 4, 5. Для визначення розрядів з одиницями скористаємося логічним виразом `i&(1<<j)`, який набуває значення істина, якщо в розряді з номером *j* двійкового подання числа *i* міститься 1, і набуває значення хибність, якщо міститься 0.

Приклад 10.5.1. Реалізація мовою С алгоритму генерації всіх підмножин заданої множини із застосуванням побітових операцій до двійкового подання десяткового числа:

```
/* Генерація всіх підмножин n-елементної множини з
```

```
   використанням побітових операцій для визначення
   позицій одиниць у двійковому поданні числа */
#define N 10 // але не більше 63
...
  int a[N]={1,2,3,4,5,6,7,8,9,10}, // елементи множини
    b[N];  // масив елементів підмножини
  unsigned long long int nsub,  // кількість всіх підмножин
    i;  // лічильник циклу (номер підмножини)
  int n,  // кількість елементів множини
    j,  // лічильник циклу (номер елемента множини)
    k;  // кількість елементів поточної підмножини
...
  while(scanf("%d",&n)) {
    if (!(n>=0 && n<=N && n<=63))
      printf("\tПомилкові дані\n");
    else {
      /* Обчислення кількості підмножин */
      nsub=1;
      for (j=1; j<=n; j++)
        nsub*=2;
      /* Генерація підмножин */
      for (i=0; i<nsub; i++) {
        /* Одержання елементів підмножини */
        k=0;
        for (j=0; j<n; j++)
          if ((i&((unsigned long long int)1<<j))!=0) { // або 1lu
            b[k]=a[j];
            k++;
          }
        /* Виведення елементів підмножини */
        printf(" %4llu:  {",i+1);
        for (j=0; j<k; j++)
          printf(" %d,",b[j]);
```

```
        if (k) printf("\b");
        printf(" }\n");
      }
    }
  }
...
```

```
        Знаходження всіх підмножин n-елементної множини
        (з використанням в алгоритмі побітових операцій)
Вкажіть кількість елементів множини n (0<=n<=10<=63)
11
        Помилкові дані
0
    1:  { }
1
    1:  { }
    2:  { 1 }
2
    1:  { }
    2:  { 1 }
    3:  { 2 }
    4:  { 1, 2 }
3
    1:  { }
    2:  { 1 }
    3:  { 2 }
    4:  { 1, 2 }
    5:  { 3 }
    6:  { 1, 3 }
    7:  { 2, 3 }
    8:  { 1, 2, 3 }
кінець
```

Як бітову маску підмножини можна *використати бінарні коди Грея*. Цей підхід схожий на попередній, але порівняно з ним змінюється порядок одержання підмножин; при цьому кожна наступна підмножина відрізняється наявністю, чи відсутністю, чи зміною лише одного елемента.

Приклад 10.5.2. Реалізація мовою C алгоритму генерації всіх підмножин заданої множини із застосуванням бінарних кодів Грея і побітових операцій:

```
/* Генерація всіх підмножин n-елементної множини з
   використанням кодів Грея і побітових операцій для
   визначення позицій одиниць у двійковому поданні числа */
#define N 10 // але не більше 63
...
  int a[N]={1,2,3,4,5,6,7,8,9,10}, // елементи множини
      b[N];  // масив елементів підмножини
```

```
  unsigned long long int nsub,  // кількість всіх підмножин
    i,  // лічильник циклу (номер підмножини)
    gi;  //код Грея
  int n,  // кількість елементів множини
    j,  // лічильник циклу (номер елемента множини)
    k;  // кількість елементів поточної підмножини
...
  while(scanf("%d",&n)) {
    if (!(n>=0 && n<=N && n<=63))
      printf("\tПомилкові дані\n");
    else {
      /* Обчислення кількості підмножин */
      nsub=1;
      for (i=1; i<=n; i++)
        nsub*=2;
      /* Генерація підмножин */
      for (i=0; i<nsub; i++) {
        /* Одержання наступного коду Грея */
        gi=i^(i>>1);
        /* Одержання елементів підмножини */
        k=0;
        for (j=0; j<n; j++)
          if ((gi&((unsigned long long int)1<<j))!=0) { // або 1lu
            b[k]=a[j];
            k++;
          }
...
      }
    }
  }
...
```

```
        Знаходження всіх підмножин n-елементної множини з
        використанням в алгоритмі кодів Грея і побітових операцій
Вкажіть кількість елементів множини n (0<=n<=10<=63)
0
    1:  { }
1
    1:  { }
    2:  { 1 }
2
    1:  { }
    2:  { 1 }
    3:  { 1, 2 }
    4:  { 2 }
3
    1:  { }
    2:  { 1 }
    3:  { 1, 2 }
    4:  { 2 }
    5:  { 2, 3 }
    6:  { 1, 2, 3 }
    7:  { 1, 3 }
    8:  { 3 }
4
    1:  { }
    2:  { 1 }
    3:  { 1, 2 }
    4:  { 2 }
    5:  { 2, 3 }
    6:  { 1, 2, 3 }
    7:  { 1, 3 }
    8:  { 3 }
    9:  { 3, 4 }
   10:  { 1, 3, 4 }
   11:  { 1, 2, 3, 4 }
   12:  { 2, 3, 4 }
   13:  { 2, 4 }
   14:  { 1, 2, 4 }
   15:  { 1, 4 }
   16:  { 4 }
кінець
```

Якщо мова програмування не має засобів роботи з окремими бітами пам'яті, то можна *як маску використати масив*, елементи якого набувають значень 0 і 1 і є розрядами двійкового подання числа. При цьому в першому елементі масиву розміщуватиметься наймолодший розряд, в останньому — найстарший. Додавання до числа 1 призводить до того, що якщо в наймолодшому розряді був 0, то він замінюється на 1, якщо була 1, то вона заміню-ється на 0 і в цьому випадку в попередньому розряді знову можливий один з двох розглянутих випадків і т.д.

Приклад 10.5.3. Реалізація мовою С алгоритму генерації всіх підмножин заданої множини із застосуванням масиву-маски:

```
/* Знаходження всіх підмножин n-елементної множини з вико-
   ристанням масиву, елементи якого містять значення розрядів
   двійкового подання числа (у зворотному порядку) */
#define N 10 // але не більше 63
...
  int a[N]={1,2,3,4,5,6,7,8,9,10}, // елементи множини
    b[N]={0};  // масив 0 і 1 - відповідає двійковому поданню числа
  unsigned long long int nsub,  // кількість всіх підмножин
    i;  // лічильник циклу (номер підмножини)
  int n,  // кількість елементів множини
    j,  // лічильник циклу (номер елемента множини)
    ind;  // індикатор завершення операції додавання 1 до
          // числа, поданого масивом
...
  while(scanf("%d",&n)) {
    if (!(n>=0 && n<=N && n<=63))
      printf("\tПомилкові дані\n");
    else {
      /* Обчислення кількості підмножин */
      nsub=1;
      for (j=1; j<=n; j++)
        nsub*=2;
      /* Генерація підмножин */
      for (i=0; i<nsub; i++) {
        ind=0;
        /* Формування маски поточної підмножини - додавання 1 */
        for (j=0; j<n && !ind; j++) {
          if (b[j]==0) {
            b[j]=1;
            ind=1;
          }
          else b[j]=0;
        }
```

```
...
            }
        }
    }
...
```

```
        Знаходження всіх підмножин n-елементної множини
        (з використанням в алгоритмі масиву, який
        містить цифри двійкового подання числа)
Вкажіть кількість елементів множини n (0<=n<=10<=63)
15
        Помилкові дані
0
    1:  { }
1
    1:  { 1 }
    2:  { }
2
    1:  { 1 }
    2:  { 2 }
    3:  { 1, 2 }
    4:  { }
3
    1:  { 1 }
    2:  { 2 }
    3:  { 1, 2 }
    4:  { 3 }
    5:  { 1, 3 }
    6:  { 2, 3 }
    7:  { 1, 2, 3 }
    8:  { }
кінець
```

Для визначення елементів підмножин перший алгоритм працює з послідовністю звичайних цілих чисел, подання кожного з яких у двійковій системі числення (а це його подання в пам'яті комп'ютера) є маскою для окремої підмножини. Тут треба визначити тільки номери розрядів, в яких містяться значення 1, це робиться за допомогою використання побітових операцій. Другий алгоритм двійкові подання цілих чисел переводить у коди Грея і теж працює з двійковою маскою. Хоч другий алгоритм і має «зайвий» крок, але при цьому порівняно з першим алгоритмом послідовність одержання підмножин змінюється — кожна наступна підмножина відрізняється від попередньої лише одним елементом. Відповідно до третього алгоритму маскою для підмножини є масив, який містить одиниці й нулі і в якому імітується виконання операції додавання 1 до числа, поданого в двійковій системі числення. Таким чином, третій алгоритм потребує виконання більшої кількості операцій і є менш ефективним, але його можна використовувати в мовах програмування, які не мають засобів роботи з окремими бітами пам'яті.

Генерація перестановок без повторень. Перестановками є послідовності, утворені з тих самих n різних елементів множини, які відрізняються тільки порядком розташування елементів. Кількість усіх перестановок елементів n-елементної множини дорівнює $n!$.

Нехай є множина елементів $A = \{1,2,3,...,n\}$ (числа тут є тільки позначками — замість чисел у множині можуть бути елементи будь-якої природи, наприклад, слова, індекси чи будь-що інше). З елементів цієї множини будемо утворювати $n!$ усіх можливих послідовностей $a_1\, a_2\, a_3 ... a_n$, $b_1\, b_2\, b_3 ... b_n$, …

Лексикографічний порядок послідовностей визначається так: $a_1a_2a_3...a_n < b_1b_2b_3...b_n$, якщо $a_1 = b_1$, $a_2 = b_2$, $a_3 = b_3$, …, $a_{k-1} = b_{k-1}$, але $a_k < b_k$, де $1 \le k \le n$. Послідовність $b_1b_2b_3...b_n$ називають *лексикографічно безпосередньо наступною* за послідовністю $a_1a_2a_3...a_n$, якщо не існує такої (проміжної) послідовності $c_1c_2c_3...c_n$, що $a_1a_2a_3...a_n < c_1c_2c_3...c_n$ і $c_1c_2c_3...c_n < b_1b_2b_3...b_n$. Так, якщо $A = \{1,2,3,4\}$, то $1234 < 2134$, $1234 < 1423$, $1234 < 4321$ тощо; серед наведених послідовностей тільки послідовність 2134 є лексикографічно безпосередньо наступною за 1234 (а, наприклад, $1234 < 1243 < 1423$, $1234 < 2134 < 4321$).

Алгоритм Нараяни[26] будує лексикографічно безпосередньо наступну перестановку за даною перестановкою. Щоб побудувати всі перестановки елементів заданої множини, треба почати з лексикографічно найменшої перестановки, тобто 123…n (елементами множини не обов’язково мають бути числа від 1 до n — це можуть бути будь-які впорядковані за зростанням значення) і $(n!-1)$ раз виконати описані нижче дії (при цьому факторіал рахувати не треба).

Алгоритм Нараяни побудови перестановок у лексикографічному порядку.

К1. Переглядаючи елементи поточної перестановки справа наліво (з кінця до початку), знайти першу пару таких елементів, що $a_{i-1} < a_i$. Якщо таких елементів нема (це означає, що при попередній побудові одержано останню перестановку), то перейти на К4.

[26] Нараяна Пандит (Narayana Pandita , 1325-1400) — індійський математик.

К2. Переглядаючи елементи поточної перестановки справа наліво (з кінця до початку), знайти перше число таке, що a_{i-1} буде меншим від нього; переставити a_{i-1} і знайдене число місцями.

К3. Записати числа a_i, a_{i+1}, a_{i+2}, …, a_n у зворотному порядку (у даному випадку числа будуть впорядковуватися за зростанням).

К4. Кінець

Роботу алгоритму продемонструємо на прикладі 4-елементної множини $\{1,2,3,4\}$. Для 4-елементної множини кількість перестановок дорівнює $4! = 24$. Одержимо кілька перших перестановок.

I перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 2\ 3\ 4}$ — лексикографічно найменша перестановка, задана при ініціалізації.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_3 = 3) < (a_4 = 4)$.

К2. Першим справа числом, що $a_3 = 3$ менше від нього, є $a_4 = 4$; міняємо ці числа місцями, тобто маємо $a_3 = 4$ і $a_4 = 3$:

$a_1a_2a_3a_4 = 1\ 2\ 4\ 3$.

К3. Після числа $a_3 = 4$ є тільки число $a_4 = 3$, тому записувати у зворотному порядку немає чого. Одержано перестановку номер 2.

II перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 2\ 4\ 3}$.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_2 = 2) < (a_3 = 4)$. Першим справа числом, що $a_2 = 2$ менше від нього, є $a_4 = 3$. Міняємо ці числа місцями, тобто маємо $a_2 = 3$ і $a_4 = 2$:

$a_1a_2a_3a_4 = 1\ 3\ 4\ 2$.

К3. Після числа $a_2 = 3$ є числа $a_3 = 4$ і $a_4 = 2$; запишемо їх у зворотному порядку і матимемо $a_3 = 2$ і $a_4 = 4$. Одержано перестановку номер 3.

III перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 3\ 2\ 4}$.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_3 = 2) < (a_4 = 4)$. Першим справа числом, що $a_3 = 2$ менше від нього, є $a_4 = 4$. Міняємо ці числа місцями, тобто маємо $a_3 = 4$ і $a_4 = 2$:

$a_1a_2a_3a_4 = 1\ 3\ 4\ 2$.

К3. Після числа $a_3 = 4$ є тільки число $a_4 = 2$, тому записувати у зворотному порядку немає чого. Одержано перестановку номер 4.

IV перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 3\ 4\ 2}$.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_2 = 3) < (a_3 = 4)$. Першим справа числом, що $a_2 = 3$ менше від нього, є $a_3 = 4$. Міняємо ці числа місцями, тобто маємо $a_2 = 4$ і $a_3 = 3$:

$a_1a_2a_3a_4 = 1\ 4\ 3\ 2$.

К3. Після числа $a_2 = 4$ є числа $a_3 = 3$ і $a_4 = 2$; запишемо їх у зворотному порядку і матимемо $a_3 = 2$ і $a_4 = 3$. Одержано перестановку номер 5.

V перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 4\ 2\ 3}$.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_3 = 2) < (a_4 = 3)$. Першим справа числом, що $a_3 = 2$ менше від нього, є $a_4 = 3$. Міняємо ці числа місцями, тобто маємо $a_3 = 3$ і $a_4 = 2$:

$a_1a_2a_3a_4 = 1\ 4\ 3\ 2$.

К3. Після числа $a_3 = 3$ є тільки число $a_4 = 2$, тому записувати у зворотному порядку немає чого. Одержано перестановку номер 6.

VI перестановка: $a_1a_2a_3a_4 = \mathbf{1\ 4\ 3\ 2}$.

К1. Перегдядаючи елементи перестановки справа наліво, маємо $(a_1 = 1) < (a_2 = 4)$. Першим справа числом, що $a_1 = 1$ менше від нього, є $a_4 = 2$. Міняємо ці числа місцями, тобто маємо $a_1 = 2$ і $a_4 = 1$:

$a_1a_2a_3a_4 = 2\ 4\ 3\ 1$.

К3. Після числа $a_1 = 2$ є числа $a_2 = 4$, $a_3 = 3$ і $a_4 = 1$; запишемо їх у зворотному порядку і матимемо $a_2 = 1$, $a_3 = 3$ і $a_4 = 4$. Одержано перестановку номер 7.

VII перестановка: $a_1a_2a_3a_4 = \mathbf{2\ 1\ 3\ 4}$.

. . .

XXIV перестановка: $a_1a_2a_3a_4 = \mathbf{4\ 3\ 2\ 1}$.

К1. При перегляді елементів перестановки справа наліво не знайдено таких елементів, що $a_{i-1} < a_i$. Отже, перестановка номер 24 є останньою. Процес генерації перестановок множини $\{1, 2, 3, 4\}$ закінчено.

Приклад 10.5.4. Реалізація мовою С алгоритму Нараяни генерації перестановок у лексикографічному порядку:

```
/* Побудова всіх перестановок елементів n-елементної
   множини (алгоритм Нараяни побудови перестановок
   у лексикографічному порядку) */
...
#define N 10  // кількість елементів масиву
int perest_Naraiana(char a[N], int n);
void vyvid(char a[N], int n, int nom);
int main() {
  char a[N];  // множина елементів
  int n,  // кількість елементів масиву
    i;  // лічильник циклу - індекс елемента
...
```

```
  while(scanf("%d",&n)) {
    if (!(n>=0 && n<=N))
      printf("\tПомилкові дані\n");
    else {
      /* Ініціалізація масиву числами 1, 2, 3, ..., n */
      for (i=0; i<n; i++)
        a[i]=i+1;
      vyvid(a, n, 1);  // виведення першої перестановки
      /* Одержання і виведення всіх решти перестановок  */
      while (perest_Naraiana(a, n))
        vyvid(a, n, 0);
    }
    printf("\n");
  }
...
}
/* Алгоритм Нараяни одержання лексикографічно впорядкованих
   перестановок елементів n-елементної множини. При одному
   зверненні до функції генерується одна перестановка на
   основі попередньої перестановки */
int perest_Naraiana( char a[N], int n) {
  int ind,  // індикатор продовження генерації перестановок
    i,  // параметр циклу - індекс елемента масиву
    j,  // індекс елемента масиву
    aj_1,  // поточний елемент
    r;  // робоча змінна – індикатор; для обміну
  ind=1;  // ще будуть утворюватися перестановки
  /* Пошук першої справа пари елементів a[i-1]<a[i] */
  j=0;
  for (i=n-1; i>0 && j==0; i--) {
    if (a[i-1]<a[i])
      j=i;
  }
```

```
  if (j==0)
    ind=0;  // знайдено всі перестановки
  else {
    /* Пошук a[i]>a[j-1] у кінці масиву */
    aj_1=a[j-1];
    r=1;  // перестановки не було
    for (i=n-1; i>=j && r; i--)
      if (aj_1<a[i]) {
        /* Переставляємо a[j-1] і a[i] */
        a[j-1]=a[i];
        a[i]=aj_1;
        r=0;  // перестановка була
      }
      /* Перевертання масиву від a[j] до кінця */
      for(i=n-1; i>j; i--, j++) {
        r=a[i];
        a[i]=a[j];
        a[j]=r;
      }
  }
  return ind;
}
/* Виведення елементів перестановки */
void vyvid(char a[N], int n, int nom) {
  static int k;  // номер перестановки n-елементної множини
  int i;  // параметри циклу - індекс елемента масиву
  if (nom) k=0;  // якщо nom=1, то виводиться перша перестановка
  k++;
  printf("\t%3d:   ", k);
  for (i=0; i<n; i++)
    printf("%d  ",a[i]);
   if (k%3==0)
    printf("\n");
}
```

```
       Знаходження всіх лексикографічно впорядкованих
       перестановок елементів множини
Вкажіть кількість елементів множини (0<=n<=10)
3
         1:   1  2  3          2:   1  3  2          3:   2  1  3
         4:   2  3  1          5:   3  1  2          6:   3  2  1

0
         1:
1
         1:   1
2
         1:   1  2      2:   2  1
4
         1:   1  2  3  4       2:   1  2  4  3       3:   1  3  2  4
         4:   1  3  4  2       5:   1  4  2  3       6:   1  4  3  2
         7:   2  1  3  4       8:   2  1  4  3       9:   2  3  1  4
        10:   2  3  4  1      11:   2  4  1  3      12:   2  4  3  1
        13:   3  1  2  4      14:   3  1  4  2      15:   3  2  1  4
        16:   3  2  4  1      17:   3  4  1  2      18:   3  4  2  1
        19:   4  1  2  3      20:   4  1  3  2      21:   4  2  1  3
        22:   4  2  3  1      23:   4  3  1  2      24:   4  3  2  1

кінець
```

Розглянемо алгоритм одержання *перестановок за допомогою циклічних зсувів* елементів масиву (перший елемент блоку елементів стає останнім). Цей алгоритм дає найменший розмір коду програми, але він формує перестановки, послідовність яких не є лексикографічно впорядкованою. Алгоритм особливо ефективно працює, якщо використовувати не масив, а регістри комп'ютера.

Для побудови всіх перестановок елементів заданої *n*-елементної множини, треба, маючи першу перестановку $123\ldots n$, виконати $(n!-1)$ раз (при цьому факторіал рахувати не треба) вказану нижче послідовність дій.

Алгоритм побудови перестановок з використанням циклічних зсувів.

K1. $k = n$.

K2. Циклічно зсунути вліво перші *k* елементів перестановки, тобто замінити перестановку $a_1 a_2 a_3 \ldots a_{k-1} a_k a_{k+1} \ldots a_n$ на перестановку $a_2 a_3 \ldots a_{k-1} a_k a_1 a_{k+1} \ldots a_n$.

K3. Якщо $a_k \neq k$, то перейти на K6.

K4. $k = k - 1$.

K5. Якщо $k > 1$, то на K2; інакше на попередньому кроці побудовано останню перестановку.

K6. Кінець.

Тут змінна k визначає кількість елементів поточного блоку, в якому здійснюється зсув.

Щоб краще зрозуміти, як алгоритм працює, розглянемо його виконання на прикладі 4-елементної множини (перша перестановка 1234 задається при ініціалізації; перестановки, подані в дужках правіше від пунктирної лінії, є проміжними; стрілки над перестановками показують, куди буде переміщуватися перший елемент для одержання наступної перестановки; довжина кожної такої стрілки відповідає значенню змінної k):

Номер послідовності зсувів	Результуючі перестановки	Проміжні перестановки
I	1234 → 2341 → 3412 → 4123 →	(1234)
II	→ 2314 → 3142 → 1423 → 4231 →	(2314)
III	→ 3124 → 1243 → 2431 → 4312 →	(3124) → (1234)
IV	→ 2134 → 1342 → 3421 → 4213 →	(2134)
V	→ 1324 → 3241 → 2413 → 4132 →	(1324)
VI	→ 3214 → 2143 → 1432 → 4321 →	(3214) → (2134) → (1234)X

Алгоритм не після кожного одержання перестановки буде закінчувати роботу. Так, якщо на кроці К3 виконується рівність $a_k = k$, то побудована перестановка є проміжною — вона збігається з одержаною раніше перестановкою, причому в кожній послідовності зсувів кожна $(n+1)$ перестановка (на поданому вище рисунку — перша після пунктирної лінії) збігається з першою перестановкою поточної послідовності циклічних зсувів. Якщо одержано проміжну перестановку, то треба зменшити блок і в ньому виконати циклічний зсув. Одержана при цьому нова перестановка теж може бути проміжною (послідовності III і VI).

Приклад 10.5.5. Програмна реалізація алгоритму генерації перестановок з використанням циклічних зсувів:

```
/* Побудова всіх не впорядкованих лексикографічно
   перестановок з елементів n-елементної множини з
   використанням циклічних зсувів вліво (перший
   елемент блоку стає останнім) */
...
int perest_cycl(char a[N], int n);
void vyvid(char a[N], int n, int nom);
int main() {
  char a[N];  // множина елементів
  int n,  // кількість елементів масиву
    i;  // лічильник циклу - індекс елемента
...
  while(scanf("%d",&n)) {
    if (!(n>=1 && n<=N))
      printf("\tПомилкові дані\n");
    else {
      /* Ініціалізація масиву числами 1, 2, 3, ..., n */
      for (i=0; i<n; i++)
        a[i]=i+1;
      vyvid(a, n, 1);  // виведення першої перестановки
      /* Одержання і виведення наступної перестановки  */
      while (perest_cycl(a, n))
        vyvid(a, n, 0);
    }
    printf("\n");
  }
...
}
/* Одержання лексикографічно невпорядкованих
   перестановок елементів n-елементної множини з
   використанням циклічних зсувів вліво */
int perest_cycl(char a[N], int n) {
  int ind,  // індикатор продовження утворення перестановок
```

```
    ind_c,  // чи виконувати повторно зсув
    i,  // параметри циклу - індекс елемента масиву
    k, // кількість елементів блоку, в якому здійснюється зсув
    r;  // робоча змінна - запам'ятовує перший елемент блоку
  ind=1;  // ще будуть утворюватися перестановки
  k=n;
  do {
    ind_c=0;  // повторно зсув не виконувати
    r=a[0];
    /* Виконання циклічного зсуву вліво - перший
       елемент блоку стає останнім */
    for (i=1; i<k; i++)
      a[i-1]=a[i];
    a[k-1]=r;
```

```
        Знаходження всіх лексикографічно невпорядкованих
        перестановок елементів множини
Вкажіть кількість елементів множини (1<=n<=10)
3
          1:   1  2  3          2:   2  3  1          3:   3  1  2
          4:   2  1  3          5:   1  3  2          6:   3  2  1

1
          1:   1
2
          1:   1  2      2:   2  1
4
          1:   1  2  3  4       2:   2  3  4  1       3:   3  4  1  2
          4:   4  1  2  3       5:   2  3  1  4       6:   3  1  4  2
          7:   1  4  2  3       8:   4  2  3  1       9:   3  1  2  4
         10:   1  2  4  3      11:   2  4  3  1      12:   4  3  1  2
         13:   2  1  3  4      14:   1  3  4  2      15:   3  4  2  1
         16:   4  2  1  3      17:   1  3  2  4      18:   3  2  4  1
         19:   2  4  1  3      20:   4  1  3  2      21:   3  2  1  4
         22:   2  1  4  3      23:   1  4  3  2      24:   4  3  2  1

кінець
```

```
    /* Якщо одержана перестановка є повторною, то вона
    не виводиться і змінюваний блок зменшується */
    if (a[k-1]==k) {
      k=k-1;  // зменшення блоку
      if (k>1)    // якщо в блоці більше ніж 1 елемент,
        ind_c=1;  // то треба виконати циклічний зсув
```

```
      else
         ind=0;  // одержано всі перестановки
    }
  } while (ind_c);
  return ind;
}
...
```

Генерація перестановок з повтореннями. Якщо у заданій сукупності n елементів деякі елементи повторюються по n_1, n_2, …, n_k разів і всі елементи сукупності переставляють місцями, то таких перестановок буде $P_n^{n_1, n_2, \ldots, n_k} = \dfrac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$. Сукупність, у якій елементи повторюються, ще називають мультимножиною.

Алгоритм Нараяни без жодних модифікацій генерує перестановки з повтореннями, тобто цей алгоритм можна застосовувати не тільки для множин, а й для мультимножин. Єдине що — в головній функції (див. приклад 10.5.4) треба задати, крім кількості елементів мультимножини, ще й самі елементи і впорядковувати їх за зростанням (це пов'язано з тим, що алгоритм дає лексикографічно впорядковану послідовність перестановок, і генерацію треба починати з мінімальної перестановки). Для впорядкування невеликої кількості заданих користувачем елементів можна скористатися, наприклад, найпростішим методом сортування звичайною бульбашкою (див. приклад 8.4.1) чи якимось іншим.

Приклад 10.5.6. Програма побудови всіх перестановок елементів n-елементної мультимножини відрізняється від програми, поданої в прикладі 10.5.4, тільки введенням елементів мультимножини в головній функції і використанням функції сортування за зростанням для одержання мінімальної перестановки (якщо користувач введе елементи не відсортованими за зростанням).

Результати роботи цієї програми:

```
        Знаходження всіх лексикографічно впорядкованих
        перестановок елементів мультимножини
Вкажіть кількість елементів мультимножини (0<=n<=10)
0
  Задайте значення елементів

          1:
2
  Задайте значення елементів
1 1
          1:   1  1
2
  Задайте значення елементів
7 4
          1:   4  7       2:   7  4
5
  Задайте значення елементів
3 7 3 5 3
          1:   3  3  3  5  7      2:   3  3  3  7  5      3:   3  3  5  3  7
          4:   3  3  5  7  3      5:   3  3  7  3  5      6:   3  3  7  5  3
          7:   3  5  3  3  7      8:   3  5  3  7  3      9:   3  5  7  3  3
         10:   3  7  3  3  5     11:   3  7  3  5  3     12:   3  7  5  3  3
         13:   5  3  3  3  7     14:   5  3  3  7  3     15:   5  3  7  3  3
         16:   5  7  3  3  3     17:   7  3  3  3  5     18:   7  3  3  5  3
         19:   7  3  5  3  3     20:   7  5  3  3  3
5
  Задайте значення елементів
3 1 3 1 2
          1:   1  1  2  3  3      2:   1  1  3  2  3      3:   1  1  3  3  2
          4:   1  2  1  3  3      5:   1  2  3  1  3      6:   1  2  3  3  1
          7:   1  3  1  2  3      8:   1  3  1  3  2      9:   1  3  2  1  3
         10:   1  3  2  3  1     11:   1  3  3  1  2     12:   1  3  3  2  1
         13:   2  1  1  3  3     14:   2  1  3  1  3     15:   2  1  3  3  1
         16:   2  3  1  1  3     17:   2  3  1  3  1     18:   2  3  3  1  1
         19:   3  1  1  2  3     20:   3  1  1  3  2     21:   3  1  2  1  3
         22:   3  1  2  3  1     23:   3  1  3  1  2     24:   3  1  3  2  1
         25:   3  2  1  1  3     26:   3  2  1  3  1     27:   3  2  3  1  1
         28:   3  3  1  1  2     29:   3  3  1  2  1     30:   3  3  2  1  1

кінець
```

Генерація сполук без повторень. *Сполуками* (ще кажуть: комбінаціями) є послідовності, утворені з n різних елементів по m $(m \le n)$ елементів, які відрізняються хоча б одним елементом. Кількість сполук із n елементів по m дорівнює $C_n^m = \dfrac{n!}{m! \cdot (n-m)!}$.

Розглянемо два алгоритми генерації сполук без повторень, які дають можливість одержувати сполуки в різному порядку.

Перший алгоритм генерує *лексикографічно впорядковану послідовність сполук*. Він дуже простий.

Сполука містить m елементів і, якщо вважати, що значеннями елементів

множини є натуральні числа від 1 до n і вони впорядковані за зростанням, то першою сполукою є множина:

$$1,\ 2,\ 3,\ \ldots,\ m-2,\ m-1,\ m.$$

Наступну сполуку буде одержано на основі даної шляхом збільшення останнього елемента на 1:

$$1,\ 2,\ 3,\ \ldots,\ m-2,\ m-1,\ m+1.$$

При цьому для одержання кожної наступної сполуки аналізуються, починаючи з останнього, елементи кожної попередньої сполуки. Якщо в останній одержаній сполуці останній елемент набув свого можливого найбільшого значення, тобто n, то для одержання поточної сполуки треба змінити значення попереднього елемента, але, якщо цей попередній елемент має своє найбільше можливе значення значення, тобто $n-1$, то треба змінити значення елемента перед ним, і т. д. Таким чином, якщо k-й з кінця сполуки елемент має значення $n-k+1$, а $(k+1)$-й з кінця сполуки елемент має значення менше від $n-k$, то його значення треба збільшити на 1, а значення всіх наступних елементів мають утворювати зростаючу послідовність сусідніх натуральних чисел, починаючи з наступного за ним, тобто на 1 більшого, числа. Таким чином одержимо лексикографічно впорядковану послідовність сполук.

Останньою буде така сполука:

$$n-m+1,\ n-m+2,\ n-m+3,\ \ldots,\ n-2,\ n-1,\ n.$$

Нехай, наприклад, треба одержати всі сполуки із 4 елементів по 3 (таких сполук буде $C_4^3=4$). Їх неважко записати, використовуючи поданий вище алгоритм:

```
1 2 3
1 2 4
1 3 4
2 3 4
```

Приклад 10.5.7. Програмна реалізація мовою С алгоритму генерації лексикографічно впорядкованої послідовності сполук:

```
/* Генерація лексикографічно впорядкованої послідовності
   сполук із n елементів по m */
```

```
...
#define M 10
int comb(int a[M], int n, int m);
void vyvid(int a[M], int m, int nom);
int main() {
  int a[M];     // елементи поточної сполуки
  int n,    // кількість елементів множини
      m,    // кількість елементів сполуки
      i;    // параметр циклу
...
  while(scanf("%d%d",&n,&m)) {
    if (!(n>=0 && m<=n && m<=M))
      printf("\tПомилкові дані\n");
    else { // ініціалізація  масиву - генерація першої сполуки
      for (i=0; i<m; i++)
        a[i]=i+1;
      vyvid(a,m,1);  // виведення першої сполуки
     /* Одержання сполук із n елементів по m і їхнє виведення */
      while(comb (a, n, m))
        vyvid(a, m, 0);
      printf("\n");
    }
  }
...
}
/* Одержання сполук із n елементів по m */
int comb(int a[M], int n, int m) {
  int i,  // параметр зовнішнього циклу
      j,  // параметр внутрішнього циклу зміни
          // значень попередніх елементів
      ind;  // індикатор зміни останніх елементів масиву
  ind=0;  // масив не змінювався
  for (i=m-1; i>=0 && !ind; i--) {
```

```
    /* Перегляд елементів з кінця масиву і визначення номера
       елемента, починаючи з якого треба зробити зміни */
    if (a[i]<n-m+i+1) {  // елемент менший від максимального
      a[i]++;
      ind=1;  // елемент масиву змінено
      for (j=i+1; j<m; j++)  // зміна останніх елементів
        a[j]=a[j-1]+1;  // утворюється зростаюча послідовність
    }
  }
  return ind;
}
...
```

```
      Знаходження всіх сполук із n елементів по m
      (послідовність впорядкована лексикографічно)
 Введіть значення n (0<=n) і m (0<=m<=n), де m<=10
5 2
          1:   1  2      2:   1  3      3:   1  4
          4:   1  5      5:   2  3      6:   2  4
          7:   2  5      8:   3  4      9:   3  5
         10:   4  5
5 3
          1:   1  2  3         2:   1  2  4         3:   1  2  5
          4:   1  3  4         5:   1  3  5         6:   1  4  5
          7:   2  3  4         8:   2  3  5         9:   2  4  5
         10:   3  4  5
5 4
          1:   1  2  3  4      2:   1  2  3  5      3:   1  2  4  5
          4:   1  3  4  5      5:   2  3  4  5
4 3
          1:   1  2  3         2:   1  2  4         3:   1  3  4
          4:   2  3  4
кінець
```

Розглянемо покрокове виконання функції comb. Наприклад, одержимо всі сполуки з 4 елементів по 3 (їх буде $C_4^3 = 4$). Таким чином, буде 4 звернення до функції comb:

Перше звернення до функції comb: a[0]=1, a[1]=2, a[2]=3, n=4, m=3.
ind=0.
I прохід зовнішнього циклу: i=3–1=2 $\geq$ 0, а також ind=0.
(a[2]=3)<(n–m+i+1=4–3+2+1=4), отже:
a[2]=3+1=4, а також ind=1.
I прохід внутрішнього циклу: (j=2+1=3) $\not<$ (m=3) — цикл не виконується.
i=2–1=1.

II прохід зовнішнього циклу: i=1 $\geq$ 0, а також ind=1 — цикл закінчує роботу. Функція comb повертає значення ind=1, а також змінений масив a[0]=1, a[1]=2, a[2]=4.

Друге звернення до функції comb: a[0]=1, a[1]=2, a[2]=4, n=4, m=3.

ind=0.

I прохід зовнішнього циклу: i=3–1=2 $\geq$ 0, а також ind=0.

(a[2]=4) $\nless$ (n–m+i+1=4–3+2+1=4) — ніяка дія не виконується.

i=2–1=1.

II прохід зовнішнього циклу: i=1 $\geq$ 0, а також ind=0.

(a[1]=2)<(n–m+i+1=4–3+1+1=3), отже:

a[1]=2+1=3, а також ind=1.

I прохід внутрішнього циклу: (j=1+1=2)<(m=3).

a[2]=a[1]+1=3+1=4.

II прохід внутрішнього циклу: (j=3) $\nless$ (m=3) — цикл закінчує роботу.

i=1–1=0.

III прохід зовнішнього циклу: i=0 $\geq$ 0, а також ind=1 — цикл закінчує роботу. Функція comb повертає значення ind=1, а також змінений масив a[0]=1, a[1]=3, a[2]=4.

Третє звернення до функції comb: a[0]=1, a[1]=3, a[2]=4, n=4, m=3.

ind=0.

I прохід зовнішнього циклу: i=3–1=2 $\geq$ 0, а також ind=0.

(a[2]=4) $\nless$ (n–m+i+1=4–3+2+1=4) — ніяка дія не виконується.

i=2–1=1.

II прохід зовнішнього циклу: i=1 $\geq$ 0, а також ind=0.

(a[1]=3) $\nless$ (n–m+i+1=4–3+1+1=3) — ніяка дія не виконується.

i=1–1=0.

III прохід зовнішнього циклу: i=0 $\geq$ 0, а також ind=0.

(a[0]=1)<(n–m+i+1=4–3+0+1=2), отже:

a[0]=1+1=2, а також ind=1.

I прохід внутрішнього циклу: (j=0+1=1)<(m=3).

a[1]=a[0]+1=2+1=3.

II прохід внутрішнього циклу: (j=2)<(m=3).

a[2]=a[1]+1=3+1=4.

III прохід внутрішнього циклу: (j=3) $\nless$ (m=3) — цикл закінчує роботу.

i=0–1=–1.

IV прохід зовнішнього циклу: i=–1 $\ngeq$ 0, а також ind=1 — цикл закінчує роботу. Функція comb повертає значення ind=1, а також змінений масив a[0]=2, a[1]=3, a[2]=4.

Четверте звернення до функції comb: a[0]=2, a[1]=3, a[2]=4, n=4, m=3.

ind=0.

I прохід зовнішнього циклу: i=3–1=2 $\geq$ 0, а також ind=0.

(a[2]=4) $\nless$ (n–m+i+1=4–3+2+1=4) — ніяка дія не виконується.

i=2–1=1.

II прохід зовнішнього циклу: i=1 $\geq$ 0, а також ind=0.

(a[1]=3) $\nless$ (n–m+i+1=4–3+1+1=3) — ніяка дія не виконується.

i=1–1=0.

III прохід зовнішнього циклу: i=0 $\geq$ 0, а також ind=0.

(a[0]=2) $\nless$ (n–m+i+1=4–3+0+1=2) — ніяка дія не виконується.

i=0–1=–1.

IV прохід зовнішнього циклу: i=–1 $\ngeq$ 0, а також ind=0 — цикл закінчує роботу. Функція comb повертає значення ind=0, що означає, що всі можливі сполуки побудовано.

Інший алгоритм — *алгоритм обертових дверей* (один зайшов — один вийшов) для одержання всіх сполук із *n* елементів по *m* базується на ідеї кодів Грея — два послідовні коди відрізняються лише значенням одного розряду. Він генерує сполуки в тому ж порядку, що якби маскою служили *n*-розрядні коди Грея і було вибирано серед них тільки ті, які містять *m* одиниць.

Алгоритм працює для $1 < m \le n$ (при $m = 1$ даний алгоритм не працює) і генерує сполуки в почерговому лексикографічному порядку. В алгоритмі використовується $(m+1)$-елементний масив a[1], a[2], …, a[m], a[m+1], у якому останній елемент є допоміжним; масив ініціалізується числами від 0 до $(m-1)$ і значенням *n*: $a_i = i - 1$ при $i = \overline{1, m}$, $a_{m+1} = n$. Значення перших *m* елементів утворюють першу сполуку, яку можна вивести чи використати з будь-якою іншою метою. Для одержання інших сполук треба виконати дії, вказані нижче.

Алгоритм обертових дверей одержання сполук без повторень.

К1. Якщо *m* — непарне, то, якщо $a_1 + 1 < a_2$, то $a_1 = a_1 + 1$ і перейти на К4, інакше $i = 2$ і перейти на К2.

Якщо *m* — парне, то, якщо $a_1 > 0$, то $a_1 = a_1 - 1$ і перейти на К4, інакше $i = 2$ і перейти на К3.

К2. Якщо $a_i \ge i$, то $a_i = a_{i-1}$, $a_{i-1} = i - 2$ і перейти на К4, інакше $i = i + 1$ і перейти на К3.

К3. Якщо $a_i + 1 < a_{i+1}$, то $a_{i-1} = a_i$, $a_i = a_i + 1$ і на К4, інакше $i = i + 1$. Якщо $i <= m$, то перейти на К2, інакше перейти на К4.

К4. Кінець.

Поданий алгоритм має досить складну логіку переходів. Її можна програмно реалізувати один до одного з використанням оператора безумовного переходу. Але це суперечить принципові структурного програмування. Хорошим розв'язанням проблеми є використання непрямої (взаємної) рекурсії.

Приклад 10.5.8. Програмну реалізацію мовою С алгоритму обертових дверей для генерації сполук без повторень подано нижче. У програмі функції К2 і К3 відповідають однойменним крокам алгоритму; вони рекурсивно ви-

кликають одна одну, доки не буде згенеровано поточну сполуку. Враховуючи, що в алгоритмі на кроці К3 використовується значення *m*, то, щоб не передавати це значення окремо через параметри при реалізації непрямої рекурсії, воно буде зберігатися в елементі a[0] масиву. Таким чином, масив буде містити $m+2$ елементи.

```
/* Алгоритм "обертові двері" - генерація сполук без
   повторень із n елементів по m */
...
#define M 10
int comb_od(int a[M+2], int m);
void k2(int a[M+2], int *i, int *ind);
void k3(int a[M+2], int *i, int *ind);
void vyvid(int a[M+2], int m, int nom);
int main() {
  int a[M+2];   // елементи поточної сполуки і значення m, n
  int n,   // кількість елементів множини
      m,   // кількість елементів сполуки
      i;   // номер поточного елемента сполуки
...
  while(scanf("%d%d",&n,&m)) {
    if (!(n>=2 && m>=2 && m<=n && m<=M))
      printf("\tПомилкові дані\n");
    else { // ініціалізація  масиву - генерація першої сполуки
      for (i=1; i<=m; i++)
        a[i]=i-1;
      a[0]=m;
      a[m+1]=n;
      vyvid(a,m,1);  // виведення першої сполуки
      /* Одержання послідовності сполук із n елементів
         по m і їхнє виведення */
      while(comb_od(a,m))
        vyvid(a,m,0);
```

```
      printf("\n");
    }
  }
...
}
/* Алгоритм "обертові двері" одержання сполук
   із n елементів по m */
int comb_od(int a[M+2], int m) {
  int i,  // номер елемента масиву
    ind;  // індикатор зміни останніх елементів масиву
  ind=1;  // масив буде змінюватися
  if (m&1)   // m непарне
    if (a[1]+1<a[2])
      a[1]++;   // і перейти на кінець (К4)
    else {
      i=2;
      k2(a, &i, &ind);   // і перейти на кінець (К4)
    }
  else   // m парне
    if (a[1]>0)
      a[1]--;   // і перейти на кінець (К4)
    else {
      i=2;
      k3(a, &i, &ind);   // і перейти на кінець (К4)
    }
  return ind;
}
/* Реалізація кроку К2 алгоритму.
   Спроба зменшення a[i]. На даному етапі a[i]=a[i-1]+1 */
void k2(int a[M+2], int *i, int *ind) {
  if (a[*i]>=*i) {
    a[*i]=a[*i-1];
    a[*i-1]=*i-2;
  }
```

```
  else {
    (*i)++;
    k3(a, i, ind);    // неявна рекурсія
  }
}
```

```
        Знаходження всіх сполук із n елементів по m
        (алгоритм 'обертові двері')
 Введіть значення n (2<=n) і m (2<=m<=n), де m<=10
5 2
          1:   0  1       2:   1  2       3:   0  2
          4:   2  3       5:   1  3       6:   0  3
          7:   3  4       8:   2  4       9:   1  4
         10:   0  4
5 3
          1:   0  1  2          2:   0  2  3          3:   1  2  3
          4:   0  1  3          5:   0  3  4          6:   1  3  4
          7:   2  3  4          8:   0  2  4          9:   1  2  4
         10:   0  1  4
5 4
          1:   0  1  2  3       2:   0  1  3  4       3:   1  2  3  4
          4:   0  2  3  4       5:   0  1  2  4
5 5
          1:   0  1  2  3  4
5 6
        Помилкові дані
4 3
          1:   0  1  2          2:   0  2  3          3:   1  2  3
          4:   0  1  3
кінець
```

```
/* Реалізація кроку К3 алгоритму.
   Спроба збільшення a[i]. На даному етапі a[i-1]=i-2 */
void k3(int a[M+2], int *i, int *ind) {
  if (a[*i]+1<a[*i+1]) {
    a[*i-1]=a[*i];
    a[*i]++;
  }
  else {
    (*i)++;
    if (*i<=a[0])
      k2(a, i, ind);    // неявна рекурсія
    else
```

```
        *ind=0;   // усі сполуки згенеровано
    }
}
...
```

Генерація сполук з повтореннями. Якщо з елементів n-елементної множини утворювати набори з однаковою кількістю m елементів, повторюючи ці елементи будь-яку кількість разів, але при цьому порядок елементів не враховувати, то буде одержано сполуки з повтореннями. Сполуки з повтореннями на відміну від сполук без повторень можуть містити більше елементів, ніж їх є в початковій множині, тобто m може бути будь-яким — і меншим чи рівним, і більшим за n. Кількість таких сполук обчислюють за формулою $f_n^m = C_{n+m-1}^m$.

Розглянемо алгоритм, який дає можливість генерувати лексикографічно впорядковану послідовність сполук з повтореннями. Його легко одержати з розглянутого вище алгоритму генерації лексикографічно впорядкованої послідовності сполук без повторень. У сполуках без повторень елементи не повторюються, а в даному випадку можуть повторюватися будь-яку кількість разів. Масив сполук містить m елементів і, якщо вважати, що значеннями елементів є натуральні числа від 1 до n і елементи масиву впорядковані за зростанням, то першою сполукою є мультимножина:

$$\underbrace{1,\ 1,\ 1,\ \ldots,\ 1,\ 1}_{m \text{ разів}}.$$

Наступну сполуку буде одержано на основі даної шляхом збільшення останнього елемента на 1:

$$\underbrace{1,\ 1,\ 1,\ \ldots,\ 1}_{m-1 \text{ раз}},2.$$

Останній елемент при генерації кожної наступної сполуки буде збільшуватися на 1, доки не стане рівним n. Після цього передостанній елемент буде збільшено на 1 (набуде значення 2), останній елемент набуде такого ж значення (теж 2). Знову останній елемент буде збільшуватися на 1, доки не стане рівним n, і т. д.

У загальному, для одержання кожної наступної сполуки аналізуються, починаючи з останнього, елементи кожної попередньої сполуки. Якщо в останній одержаній сполуці останній елемент масиву набув свого можливого найбільшого значення, тобто n, то для одержання поточної сполуки треба змінити значення попереднього елемента, але, якщо цей попередній елемент набуде свого найбільшого можливого значення, тобто n, то треба змінити значення елемента перед ним, і т. д. Таким чином, якщо k-й з кінця масиву елемент має значення n, а $(k+1)$-й з кінця масиву елемент має значення менше від n, то його значення треба збільшити на 1, а значення всіх наступних елементів мають стати рівними цьому одержаному значенню. Тобто елементи мультимножини впорядковуються так, що після кожного елемента може розміщуватися або рівний йому або більший від нього елемент. У результаті буде одержано лексикографічно впорядковану послідовність сполук з повтореннями.

Останньою буде сполука:

$$\underbrace{n,\ n,\ n,\ \ldots,\ n,\ n}_{m\ \text{разів}}.$$

Нехай, наприклад, треба одержати всі сполуки з повтореннями із 4 елементів по 3 (таких сполук буде $f_4^3 = C_6^3 = 20$). Їх неважко записати, використовуючи поданий вище алгоритм (послідовність комбінацій подано по стовпчиках):

```
1 1 1     1 2 3     2 2 2     2 4 4
1 1 2     1 2 4     2 2 3     3 3 3
1 1 3     1 3 3     2 2 4     3 3 4
1 1 4     1 3 4     2 3 3     3 4 4
1 2 2     1 4 4     2 3 4     4 4 4
```

Приклад 10.5.9. Програмна реалізація мовою C алгоритму генерації лексикографічно впорядкованої послідовності сполук з повтореннями (зміни, внесені в програму з прикладу 10.5.7, відмічено коментарями `//***`):

```
/* Генерація всіх лексикографічно впорядкованих
   сполук з повтореннями із n елементів по m */
...
#define M 10
int comb_p(int a[M], int n, int m);
```

```
void vyvid(int a[M], int m, int nom);
int main() {
  int a[M];
  int i, n, m;
...
  while(scanf("%d%d",&n,&m)) {
    if (!(n>=1 && m>=1 && m<=M))  //***
      printf("\tПомилкові дані\n");
    else { // ініціалізація  масиву - генерація першої сполуки
      for (i=0; i<m; i++)
        a[i]=1;  //***
      vyvid(a,m,1);  // виведення першої сполуки
     /* Одержання сполук з повтореннями із n елементів
        по m і їхнє виведення */
      while(comb_p(a,n,m))
        vyvid(a,m,0);
      printf("\n");
    }
  }
...
}
/* Одержання лексикографічно впорядкованої послідовності
   сполук з повтореннями із n елементів по m */
int comb_p(int a[M], int n, int m) {
  int i,  // параметр зовнішнього циклу
    j,    // параметр внутрішнього циклу зміни
          // значень попередніх елементів
    ind;  // індикатор зміни останніх елементів масиву
  ind=0;  // масив не змінювався
  for (i=m-1; i>=0 && !ind; i--) {
    /* Перегляд елементів з кінця масиву і визначення номера
       елемента, починаючи з якого треба зробити зміни */
    if (a[i]<n) {  //*** елемент менший від максимального
```

```
        a[i]++;
        ind=1;  // елемент масиву змінено
        for (j=i+1; j<m; j++)  // зміна останніх елементів
          a[j]=a[i];  //***
      }
    }
  return ind;
}
...
```

```
       Знаходження всіх лексикографічно впорядкованих
       сполук з повтореннями із n елементів по m
 Введіть значення n (1<=n) і m (1<=m<=10)
1 1
     1:   1
1 3
     1:   1  1  1
1 0
       Помилкові дані
2 3
     1:   1  1  1       2:   1  1  2       3:   1  2  2
     4:   2  2  2
4 3
     1:   1  1  1       2:   1  1  2       3:   1  1  3
     4:   1  1  4       5:   1  2  2       6:   1  2  3
     7:   1  2  4       8:   1  3  3       9:   1  3  4
    10:   1  4  4      11:   2  2  2      12:   2  2  3
    13:   2  2  4      14:   2  3  3      15:   2  3  4
    16:   2  4  4      17:   3  3  3      18:   3  3  4
    19:   3  4  4      20:   4  4  4
3 4
     1:   1  1  1  1       2:   1  1  1  2       3:   1  1  1  3
     4:   1  1  2  2       5:   1  1  2  3       6:   1  1  3  3
     7:   1  2  2  2       8:   1  2  2  3       9:   1  2  3  3
    10:   1  3  3  3      11:   2  2  2  2      12:   2  2  2  3
    13:   2  2  3  3      14:   2  3  3  3      15:   3  3  3  3

кінець
```

Генерація розміщень без повторень. *Розміщеннями* є послідовності, утворені з n різних елементів по m ($m \le n$) елементів, які відрізняються або складом елементів, або їхнім порядком. Кількість розміщень без повторень дорівнює $A_n^m = \dfrac{n!}{(n-m)!} = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-m+1)$.

Для генерування розміщень без повторень можна спочатку згенерувати сполуки, скориставшись, наприклад, алгоритмом генерації лексикографічно впорядкованої послідовності сполук, а потім скористатися, наприклад, алгоритмом

Нараяни генерації лексикографічно впорядкованої послідовності перестановок. Ці алгортми розглянуто раніше. При цьому будуть генеруватися поблочно лексикографічно впорядковані розміщення — кожен блок визначається лексикографічно впорядкованими сполуками, в межах кожного блоку розміщення теж впорядковуються лексикографічно. У такому комбінованому алгоритмі треба використати два масиви з m елементами — один для генерації сполук, другий — для генерації перестановок (якщо використовувати тільки один масив, то не буде збережено попередню сполуку для генерації наступної).

Приклад 10.5.10. Програмна реалізація мовою С алгоритму генерації поблочно лексикографічно впорядкованих розміщень без повторень із n елементів по m (програма звертається до функцій perest_Naraiana і comb, поданих відповідно в прикладах 10.5.4 і 10.5.7):

```
/* Генерація поблочно лексикографічно впорядкованих
   розміщень із n елементів по m без повторень */
...
#define M 10
int comb(int b[M], int n, int m);
int perest_Naraiana(int b[M], int n);
void vyvid(int b[M], int m, int nom);
int main() {
  int a[M], // масив одержуваних сполук
      b[M];  // масив одержуваних перестановок
  int n,  // кількість елементів множини
      m,  // кількість елементів розміщення
      i,  // параметр циклу
      ind;  // індикатор виведення
...
  while(scanf("%d%d",&n,&m)) {
    if (!(n>=0 && m>=0 && m<=n && m<=M))
      printf("\tПомилкові дані\n");
    else { // ініціалізація  масиву - генерація першої сполуки
      for (i=0; i<m; i++)
        a[i]=i+1;
```

```
      /* Одержання сполук із n елементів по m і одержання
         для них перестановок, тобто одержання розміщень */
      ind=1;
      do {
        for (i=0; i<m; i++)
          b[i]=a[i];
        do {
          vyvid(b, m, ind);
          ind=0;
        } while (perest_Naraiana(b, m));
        // printf("\n");  // відокремлення блоку розміщень
      } while(comb(a,n,m));
      printf("\n");
    }
  }
...
}
```

```
        Знаходження всіх розміщень із n елементів по m
 Введіть значення n (0<=n) і m (0<=m<=n), де m<=10
0 0
          1:
2 0
          1:
2 1
          1:   1         2:   2
2 2
          1:   1  2      2:   2  1
3 2
          1:   1  2      2:   2  1      3:   1  3
          4:   3  1      5:   2  3      6:   3  2

3 3
          1:   1  2  3          2:   1  3  2          3:   2  1  3
          4:   2  3  1          5:   3  1  2          6:   3  2  1

4 3
          1:   1  2  3          2:   1  3  2          3:   2  1  3
          4:   2  3  1          5:   3  1  2          6:   3  2  1
          7:   1  2  4          8:   1  4  2          9:   2  1  4
         10:   2  4  1         11:   4  1  2         12:   4  2  1
         13:   1  3  4         14:   1  4  3         15:   3  1  4
         16:   3  4  1         17:   4  1  3         18:   4  3  1
         19:   2  3  4         20:   2  4  3         21:   3  2  4
         22:   3  4  2         23:   4  2  3         24:   4  3  2

кінець
```

Генерація розміщень з повтореннями. Розміщення з повтореннями на відміну від розміщень без повторень можуть містити більше елементів, ніж їх є в початковій множині, тобто *m* може бути будь-яким — і меншим чи рівним, і більшим за *n*. При цьому всіх розміщень буде $\tilde{A}_n^m = n^m$.

При генерації всіх лексикографічно впорядкованих розміщень з повтореннями із *n* елементів по *m* першим буде розміщення, всі елементи якого мають найменше значення, тобто значення 1 — $\underbrace{111...1}_{m \text{ разів}}$. Для побудови всіх решти розміщень з повтореннями елементів заданої *n*-елементної множини, треба виконати $\tilde{A}_n^m$ разів поданий нижче алгоритм. Алгоритм передбачає, що початкова множина містить елементи 1, 2, 3, …, *n*.

Алгоритм генерації лексикографічно впорядкованих розміщень з повтореннями із *n* елементів по *m*.

К1. Переглядаючи елементи поточного розміщення справа наліво (з кінця до початку), знайти перший елемент a_i такий, що $a_i < n$. Якщо такого елемента нема (це означає, що при попередній побудові одержано останнє розміщення), то перейти на К4.

К2. $a_i = a_i + 1$.

К3. Якщо $i = m$, то поточне розміщення побудовано. Якщо $i < m$, то $a_{i+1} = 1$, $a_{i+2} = 1$, …, $a_m = 1$.

К4. Кінець.

Приклад 10.5.11. Програмна реалізація мовою C алгоритму генерації лексикографічно впорядкованих розміщень із *n* елементів по *m* з повтореннями:

```
/* Генерація лексикографічно впорядкованих розміщень
   з повтореннями із n елементів по m */
...
#define M 10
int comb(int a[M], int n, int m);
int rozmishchennia_pv(int a[M], int m, int n);
void vyvid(int a[M], int m, int nom);
```

```
int main() {
  int a[M], // масив одержуваних розміщень
    i,  // параметр циклу
    n,  // кількість елементів множини
    m,  // кількість елементів розміщення
    ind;  // індикатор виведення
...
  while(scanf("%d%d",&n,&m)) {
    if (!(n>=1 && m>=0 && m<=M))
      printf("\tПомилкові дані\n");
    else { // ініціалізація  масиву - генерація першої сполуки
      for (i=0; i<m; i++)
        a[i]=1;
      /* Одержання розміщень із n елементів по m
         з повтореннями */
      ind=1;
      do {
        vyvid(a, m, ind);
        ind=0;
      } while (rozmishchennia_pv(a, m, n));
      printf("\n");
    }
  }
...
}
/* Одержання лексикографічно впорядкованих розміщень
   елементів n-елементної множини по m з повтореннями */
int rozmishchennia_pv(int a[], int m, int n) {
  int i, j,  // параметри циклу - індекси елементів масиву
      ind;  // індикатор продовження побудови розміщень
  ind=1;
  /* Пошук першого справа елемента a[i]<n */
  for (i=m-1; i>=0 && a[i]==n; i--);
```

```
  if (i>=0) {
    a[i]++;
    for (j=i+1; j<m; j++)
      a[j]=1;
  }
  else
    ind=0;  // знайдено всі перестановки
  return ind;
}
...
```

```
        Знаходження всіх лексикографічно впорядкованих
        розміщень із n елементів по m з повтореннями
 Введіть значення n (0<=n) і m (0<=m<=10)
2 3
          1:   1  1  1          2:   1  1  2          3:   1  2  1
          4:   1  2  2          5:   2  1  1          6:   2  1  2
          7:   2  2  1          8:   2  2  2
0 3
        Помилкові дані
3 0
          1:
3 1
          1:   1        2:   2        3:   3

3 2
          1:   1  1      2:   1  2      3:   1  3
          4:   2  1      5:   2  2      6:   2  3
          7:   3  1      8:   3  2      9:   3  3

3 3
          1:   1  1  1          2:   1  1  2          3:   1  1  3
          4:   1  2  1          5:   1  2  2          6:   1  2  3
          7:   1  3  1          8:   1  3  2          9:   1  3  3
         10:   2  1  1         11:   2  1  2         12:   2  1  3
         13:   2  2  1         14:   2  2  2         15:   2  2  3
         16:   2  3  1         17:   2  3  2         18:   2  3  3
         19:   3  1  1         20:   3  1  2         21:   3  1  3
         22:   3  2  1         23:   3  2  2         24:   3  2  3
         25:   3  3  1         26:   3  3  2         27:   3  3  3

кінець
```

Серед розглянутих алгоритмів генерації комбінаторних множин деякі будують лексикографічно впорядковані послідовності множин. Лексикографічна впорядкованість дає, наприклад, можливість ефективно виконати направлений перебір можливих варіантів при пошукові оптимального розв'язку в дискретних задачах оптимизації, здійснити відбір елементів за пев-

ними умовами, від вдалого вибору лексикографічного впорядкування змінних задачі залежить ефективність роботи точних і наближених методів лексикографічного напрямленого перебору.

Генерація комбінаторних множин не обмежується розглянутими вище алгоритмами. Існує досить багато інших алгоритмів, наведених у літературі, зокрема, в [22].

10.6. Розкладання натуральних чисел на доданки і прості множники

Розглянемо ще алгоритми, пов'язані з теорією чисел, — розкладання натурального числа на доданки і на прості множники (другу задачу називають факторизацією чисел). Ці алгоритми мають експоненційну складність.

Розкладання натурального числа на доданки. У ряді математичних задач (зокрема в математичному аналізі) виникає необхідність розкласти натуральне число n на доданки a_i $(i = \overline{1,m},\ m = \overline{1,n})$. Розкладання полягає в знаходженні послідовності натуральних чисел таких, що $a_1 \ge a_2 \ge \ldots \ge a_m$ і $a_1 + a_2 + \ldots + a_m = n$. При цьому часом може виникнути необхідність одержати всі можливі доданки, а часом одержати розкладання на певну кількість доданків або на доданки, які не перевищують певного числа, чи набувають значень в певних межах, чи на доданки накладаються якісь інші обмеження.

Зі збільшенням значення n кількість розкладань збільшується досить швидко — експоненційно $O(e^{\sqrt{n}} / n)$. Визначення кількості розкладань є однією із задач теорії чисел. Так, при знаходженні всіх можливих доданків для числа 1 є одне розкладання, для 2 — 2, 3 — 3, 4 — 5, 5 — 7, 6 — 11, 7 — 15, 8 — 22, 9 — 30, 10 — 42, 20 — 627, 30 — 5604, 40 — 37338, 50 — 204226, 100 — 190569292, 120 — 1844349560, 140 — 15065878135 і т. д.

Існують різні алгоритми розкладання чисел на доданки. Розглянемо простий і досить швидкий алгоритм одержання всіх можливих розкладань на доданки у зворотному лексикографічному порядку, починаючи від n і закінчу-

ючи всіма одиницями. Алгоритм дає можливість безпосередньо формувати масиви доданків.

Алгоритм працює для будь-якого натурального числа $n \geq 1$ і генерує розкладання у зворотному лексикографічному порядку. В алгоритмі використовується $(n+1)$-елементний масив a[0], a[1], a[2], …, a[n], який містить доданки поточного розкладання. Доданки кожного наступного розкладання генеруються на основі доданків попереднього розкладання.

При ініціалізації роботи алгоритму в масиві перший елемент a[0], який є допоміжним, набуває значення *n* і зберігає це значення незмінним (при виконанні алгоритму значення *n* змінюється — набуває значення поточного доданка, який розкладається на нові доданки); елемент a[1] теж набуває значення *n*, а решта елементів a[2], …, a[n] набувають значення 1. Кількість доданків *m* у першому розкладанні дорівнює 1. При генерації наступних розкладань кількість доданків *m* буде то збільшуватися, то зменшуватися; останнє розкладання буде містити *n* доданків — всі 1. Номер поточного елемента *k* залежить від значення *n* і набуває значення 0 (якщо $n = 1$) чи 1 (якщо $n \geq 2$). Якщо $k = 0$, то це означає, що згенеровано всі розкладання і алгоритм закінчує роботу.

Отже, в результаті ініціалізації буде одержано: $a_0 = n$, $a_1 = n$, $a_i = 1$ при $i = \overline{2,n}$, $k = 1 \cdot (n \geq 2)$, $m = 1$. Першим розкладанням є задане число *n*.

Для одержання решти розкладань треба циклічно виконати дії, вказані в поданому нижче алгоритмі.

Алгоритм генерації окремих (але пов'язаних між собою) розкладань натурального числа *n* у зворотному лексикографічному порядку:

К1. Якщо $a_k \neq 2$, то перейти на К3.

К2. $a_k = 1$, $k = k - 1$, $m = m + 1$. Перейти на К6.

К3. Якщо $k = 0$, то перейти на К6; інакше $r = a_k - 1$, $a_k = r$, $n = m - k + 1$, $m = k + 1$.

К4. Якщо $n < r$, перейти на К5; інакше $a_m = r$, $m = m + 1$, $n = n - r$ і перейти на К4.

К5. $a_m = n$. Якщо $n = 1$, то $k = m - 1$; інакше $k = m$. // $k = m - (n == 1)$.

К6. Кінець — одержано поточне (а при $k = 0$ — останнє) розкладання з m доданками.

Приклад 10.6.1. Реалізація мовою С алгоритму генерації всіх розкладань числа n на доданки:

```
/* Генерація всіх розкладань числа n на доданки у зворотному
   лексикографічному порядку з використанням алгоритму,
   який безпосередньо формує масиви доданків */
...
#define N 20 // може бути більшим
void dodanky(int a[N+1], int *n, int *m, int *k);
void vyvid(int a[N+1], int m);
int main() {
  int a[N+1]; // елементи розкладання
  int n;  // число, для якого генеруються розкладання
  int m,  // кількість доданків у поточному розкладанні
    k,  // номер поточного елемента в розкладанні;
        // при k=0 генерація закінчується
    i;   // параметр циклу
...
  while(scanf("%d",&n)) {
    if (n<1 || n>N)
      printf("\tПомилкове значення\n");
    else {
      /* Ініціалізація масиву - генерація першого розкладання */
      a[0]=n;  // запам'ятовує значення n, яке надалі змінюється
      a[1]=n;  // перше розкладання
      m=1;  // у першому розкладанні один доданок
      for (i=2; i<=n; i++)
        a[i]=1;   // правіше від поточного доданка всі 1
      vyvid(a, m);   // виведення першого розкладання
      /* Одержання послідовності розкладань */
      k=m-(n==1);   // номер поточного доданка
```

```
      while (k) {   // доки k не 0
        dodanky(a, &n, &m, &k);
        vyvid(a, m);
      }
    }
  }
...
}
/* Одержання поточого розкладання */
void dodanky(int a[N+1], int *n, int *m, int *k) {
  int r;   // зменшений поточний доданок, який
           // може повторюватися в розкладанні
  /* Заміна доданка 2 на 1+1 */
  if (a[*k]==2) {
    a[*k]=1;
    (*k)--;
    (*m)++;
  }
  /* Перед найлівішою 1 розміщується число >2 */
  else {
    if(*k) {  // якщо k!=0 (при k=0 всі розкладання згенеровано)
      r=a[*k]-1;   // зменшення поточного доданка
      a[*k]=r;
      *n=*m-*k+1;
      *m=*k+1;
      while ((*n)>r) {  // заміна кількох 1 поточним значенням
        a[*m]=r;
        (*m)++;
        *n=*n-r;
      }
    }
    a[*m]=*n;
    *k=*m-(*n==1);
  }
}
```

```
/* Виведення згенерованого поточного
   розкладання числа n на доданки */
void vyvid(int a[N+1], int m) {
  int i;   // параметр циклу
  printf(" %4d = ",a[0]);
  for (i=1; i<=m; i++)
    printf(" %d +",a[i]);
  printf("\b \n");
}
```

```
        Знаходження всіх розкладань числа n на доданки
        (у зворотному лексикографічному порядку)
Вкажіть число n (0<=n<=20)
1
   1 =  1
7
   7 =  7
   7 =  6 + 1
   7 =  5 + 2
   7 =  5 + 1 + 1
   7 =  4 + 3
   7 =  4 + 2 + 1
   7 =  4 + 1 + 1 + 1
   7 =  3 + 3 + 1
   7 =  3 + 2 + 2
   7 =  3 + 2 + 1 + 1
   7 =  3 + 1 + 1 + 1 + 1
   7 =  2 + 2 + 2 + 1
   7 =  2 + 2 + 1 + 1 + 1
   7 =  2 + 1 + 1 + 1 + 1 + 1
   7 =  1 + 1 + 1 + 1 + 1 + 1 + 1
кінець
```

У розглянутому алгоритмі на значення n обмеження не накладаються, але, наприклад, при $n = 120$ всі розкладання без виведення на екран генеруються (на моєму комп'ютері) більш ніж 2 хвилини, а при $n = 140$ — більш ніж 18 хвилин.

Факторизація натурального числа — це його розкладання на прості множники. Відповідно до основної теореми арифметики на прості множники можна однозначно розкласти будь-яке натуральне число $n > 1$. Задача факторизації є досить складною. На трудомісткості факторизації довгих чисел базується стійкість багатьох криптографічних алгоритмів з відкритим ключем, зокрема, алгоритму RSA, алгоритму Рабіна та інших.

Алгоритми факторизації дають можливість розкласти число на прості множники за допомогою їхнього повторного (рекурсивного) застосування.

Існує багато алгоритмів факторизації, але більшість сучасних алгоритмів використовують ідею методу Ферма.

Алгоритм Ферма дає можливість розкласти будь-яке непарне натуральне число на два множники:

$$n = p \cdot q,$$

де n — відоме число, а p і q треба знайти. Для цього треба знайти числа A і B такі, що $n = A^2 - B^2$. Тоді $p \cdot q = A^2 - B^2 = (A - B) \cdot (A + B)$, звідки $p = A - B$, $q = A + B$.

Алгоритм Ферма:

К1. $m = \sqrt{n}$ із заокругленням до більшого.

К2. Для $x = 0, 1, 2, ...$ треба обчислювати значення $r(x) = (m + x)^2 - n$, доки $r(x)$ не стане повним квадратом.

К3. Нехай $r(x) = B^2$, отже, $B = \sqrt{r(x)}$. Оскільки $n = A^2 - B^2$, то $B^2 = A^2 - n$, отже, $A = m + x$. Тоді $p = A - B$, $q = A + B$.

Якщо одне із чисел p і q дорівнює 1, то інше є простим, якщо немає 1, то до обох чисел треба знову застосувати алгоритм.

Алгоритм Ферма базується на переборі варіантів, і він має експоненційну часову складність.

Приклад 10.6.2. Програмна реалізація мовою C алгоритму факторизації методом Ферма:

```
/* Факторизація (розкладання на прості множники)
   непарного числа n з використанням методу Ферма */
...
#include <limits.h>
#define N 20 // для непарного числа типу int
             // (INT_MAX=2^31-1=2147483647) кількість
             // множників не перевищує 20
             // (для непарного числа типу long long int
             // (LLONG_MAX=2^63-1=9223372036854775807)
             // кількість множників не перевищує 40)
```

```
void mnozhnyky_Ferma(int a[N], int n, int *k);
void vyvid(int a[N], int n, int k);
int main() {
  int a[N];   // елементи розкладання
  int n;   // число, для якого генерується розкладання
  int k;   // кількість простих множників у розкладанні
...
  while(scanf("%d",&n)) {
    if (n<1)
      printf("\tПомилкове значення\n");
    else {
      if (!(n&1)) printf("\tЗадайте непарне число\n");
      else {
        /* Розкладання непарного числа на множники */
        k=-1;
        mnozhnyky_Ferma(a, n, &k);
        vyvid(a, n, k);
      }
    }
  }
...
}
/* Рекурсивна факторизація цілого непарного
   числа n>=1 методом Ферма */
void mnozhnyky_Ferma(int a[N], int n, int *k) {
  int m,   // заокруглений до більшого корінь з n
    r,   // має стати повним квадратом
    x,   // доданок для одержання повного квадрату
    sr,   // корінь з r
    A, B,   // параметри алгоритму Ферма
    p, q;   // одержані множники
  m=ceil(sqrt((double)n));
  x=-1;
```

```
  /* Знаходження значень A і B */
  do {
    x++;
    A=m+x;
    r=A*A-n;     // одержано B^2
    sr=sqrt((double)r);
  } while (r!=sr*sr);
  B=sqrt((double)r);
  /* Обчислення значень множників */
  p=A-B;
  q=A+B;
  /* Закінчити чи продовжити розкладання? */
  if (p==1) {   // розкладання закінчено
    (*k)++;
    a[*k]=q;
  }
  else {    // треба розкласти обидва множники
    mnozhnyky_Ferma(a, p, k);
    mnozhnyky_Ferma(a, q, k);
  }
}
```

```
 Розкладання непарного числа n на прості множники
Вкажіть непарне число n (1<=n<=2147483647)
-3
        Помилкове значення
8
        Задайте непарне число
1
    1 =  1
7
    7 =  7
27
   27 =  3 · 3 · 3
1875
 1875 =  5 · 5 · 5 · 3 · 5
12345
 12345 =  3 · 5 · 823
1234567
 1234567 =  127 · 9721
1621477
 1621477 =  23 · 29 · 17 · 11 · 13
2147483647
 2147483647 =  11 · 3 · 5 · 1091 · 139 · 1459
кінець
```

```
/* Виведення розкладання числа n на множники */
void vyvid(int a[N], int n, int k) {
  int i;   // параметр циклу
  printf(" %4d = ", n);
  for (i=0; i<=k; i++)
    printf(" %d ·",a[i]);   // Alt+0183
  printf("\b \n");
}
```

У функції mnozhnyky_Ferma при обчисленні значень A і B на кожній ітерації циклу двічі обчислюються квадрати немалих чисел `A*A` і `sr*sr`, що потребує значних витрат часу.

Щоб оптимізувати обчислення, виведемо рекурентну формулу для знаходження значення виразу $r(x) = (m + x)^2 - n$ на наступному кроці через значення попереднього кроку. На кожному кроці x збільшується на 1. Тоді:

$$r(x+1) = (m + (x+1))^2 - n = ((m + x) + 1)^2 - n = (m + x)^2 + 2(m + x) + 1 - n =$$
$$= ((m + x)^2 - n) + 2(m + x) + 1 = r(x) + 2(m + x) + 1,$$

де $r(0) = m^2 - n$.

Таким чином, досить один раз обчислити квадрат числа m, а в циклі до попереднього обчисленого значення додавати $2(m + x) + 1$, де множення цілого числа на 2 виконується дуже швидко шляхом заміни його зсувом на один розряд вліво.

Щодо перевірки умови виконання циклу — доки значення $r(x)$ не стане повним квадратом, — `r!=sr*sr`. Якщо число r стане повним квадратом, то $\sqrt{r}$ має бути цілим числом. Тому, крім змінної sr цілого типу, введемо змінну дійсного типу dsr і виконаємо такі дії: $dsr = \sqrt{r}$, $sr = dsr$. Якщо dsr містить дробову частину (не є повним квадратом), то порівняння `dsr!=sr` дасть той самий результат, що й порівняння `r!=sr*sr`, але квадрат числа обчислювати не треба.

Отже, замість початкового фрагмента функції можна використати оптимізований:

```
/* Фрагмент без оптимізації */
m=ceil(sqrt((double)n));
x=-1;
/* Знаходження A і B */
do {
  x++;
  A=m+x;
  r=A*A-n;
  sr=sqrt((double)r);
} while (r!=sr*sr);
B=sqrt((double)r);
```

```
/* Фрагмент після оптимізації */
int ind; //шукати повний квадрат?
double dsr;   // корінь з r
m=ceil(sqrt((double)n));
r=m*m-n;   // значення r(0)
ind=1;
while (ind){
  dsr=sqrt((double)r);
  sr=dsr;
  if(sr==dsr) ind=0;
  else r=r+(m<<1)+1;  // r(x+1)
  m++;
}
A=--m;
B=sqrt((double)r);
```

Алгоритм Ферма дає можливість розкладати на множники тільки непарні числа. Щоб на множники розкласти будь-яке число — парне чи непарне — треба спочатку скоротити його на степінь числа 2 (див. приклад 6.3.6).

Приклад 10.6.3. Програмна реалізація мовою C факторизації будь-якого натурального числа з використанням методу Ферма може бути такою (для пришвидшення виконання скорорчення використаємо побітові операції):

```
/* Факторизація (розкладання на прості множники) будь-якого
   натурального числа n. Спочатку виконується скорочення цілого
   числа на множники 2, а потім застосовується метод Ферма */
...
#include <limits.h>
#define N 31 // для будь-якого числа типу int
             // (INT_MAX=2^31-1=2147483647) кількість
             // множників не перевищу 30
             // (для будь-якого числа типу long long int
             // (LLONG_MAX=2^63-1=9223372036854775807)
             // кількість множників не перевищує 62)
```

```
void mnozhnyky_Ferma(int a[N+1], int n, int *k);
void mnozhnyky_Dva(int a[N+1], int n, int *k);
void vyvid(int a[N+1], int n, int k);
int main() {
  int a[N+1];   // елементи розкладання
  int n;   // число, для якого генерується розкладання
  int k;   // кількість простих множників у розкладанні
...
  while(scanf("%d",&n)) {
    if (n<1)
      printf("\tПомилкове значення\n");
    else {
      a[0]=n;   // запам'ятовування значення n
      k=0;
      /* Розкладання числа на множники */
      if (!(n&1))
        mnozhnyky_Dva(a, &n, &k);
      mnozhnyky_Ferma(a, n, &k);
      vyvid(a, n, k);
    }
  }
...
}
/* Пошук множників 2 і скорочення числа n на ці множники */
void mnozhnyky_Dva(int a[N], int *n, int *k) {
  do {   // доки залишається парне число
    (*k)++;
    a[*k]=2;
    *n=*n>>1;
  } while(!(*n&1));
}
/* Рекурсивна факторизація цілого непарного числа
   n>=1 методом Ферма (оптимізований варіант) */
void mnozhnyky_Ferma(int a[N], int n, int *k) {
```

```
  int m,    // заокруглений до більшого корінь з n
    r,    // має стати повним квадратом
    ind,    // чи шукати повний квадрат
    sr,    // ціла частина кореня з r
    A, B,    // параметри алгоритму Ферма
    p, q;    // одержані множники
  double dsr;    // дійсний корінь з r
  m=ceil(sqrt((double)n));
  r=m*m-n;    // початкове значення r
  ind=1;    // треба шукати повний квадрат
  while (ind){
    dsr=sqrt((double)r);    // дійсний корінь
    sr=dsr;    // відкидання дробової частини
    if(sr==dsr)
      ind=0;    // повний квадрат знайдено
    else
      r=r+(m<<1)+1;    // наступне значення r+2m+1
    m++;
  } ;
  A=--m;
  B=sqrt((double)r);
  /* Обчислення значень множників */
  p=A-B;
  q=A+B;
  /* Закінчити чи продовжити розкладання? */
  if (p==1) {  // розкладання закінчено
    (*k)++;
    a[*k]=q;
  }
  else {    // треба розкласти обидва множники
    mnozhnyky_Ferma(a, p, k);
    mnozhnyky_Ferma(a, q, k);
  }
}
```

. . .

```
  Розкладання будь-якого числа n на прості множники
Вкажіть натуральне число n (1<=n<=2147483647)
1
    1 =  1
2
    2 =  2
1024
 1024 =  2 · 2 · 2 · 2 · 2 · 2 · 2 · 2 · 2 · 2
123456
 123456 =  2 · 2 · 2 · 2 · 2 · 2 · 3 · 643
1234567
 1234567 =  127 · 9721
123456780
 123456780 =  2 · 2 · 5 · 3 · 3 · 47 · 14593
кінець
```

Алгоритм Ферма пов'язаний з досить довгим перебором значень $r(x) = (m + x)^2 - n$ при пошукові повного квадрату. За рахунок цього особливо великими будуть часові витрати, якщо один множник дуже близький до 1, а інший до n.

Алгоритм має експоненційну часову складність, і для розкладання великих чисел його застосовувати дуже неефективно.

Швидкість факторизації з використанням алгоритму Ферма можна підвищити, якщо, наприклад, спочатку виконати пробні ділення на числа від 2 до якогось досить невеликого числа X (див. приклад 6.3.13), а після цього застосувати метод Ферма. Існує також вдосконалений метод Ферма з просіюванням (метод решета), в якому враховується, що останньою цифрою квадрату числа може бути тільки 0, 1, 4, 5, 6, 9, а також деякі інші властивості чисел.

Майже всі інші алгоритми факторизації мають експоненційну чи субекспоненційну (час виконання зростає швидше за будь-який поліноміальний, але залишається істотно меншим від експоненційного) складності. Квантовий алгоритм Шора має складність $O(\log^3 n)$, меншу за поліноміальну, але потребує для виконання спеціального квантового комп'ютера, який ще не створено. Проте для виконання на звичайному комп'ютері поки що поліноміального алгоритму факторизації не розроблено.

Запитання та завдання для самоконтролю

1. Чи буде найкращим алгоритм, який дає можливість розв'язувати задачу «напряму»? Як впливає розрядність використовуваних змінних на універсальність застосування алгоритму? Чи варто для розв'язання реальних практичних задач в програмній реалізації алгоритму виконувати перевірку щодо діапазону можливих значень не тільки остаточного результату, а й проміжних результатів обчислень? Чи завжди доцільно з метою розширення діапазону можливих значень результатів замість змінних цілого типу використовувати змінні дійсного типу? Чи завжди доцільно використовувати рекурсію? Чи завжди доцільно шукати чи розробляти найоптимальніший алгоритм?

2. Чи можна застосувати подання цілого додатного числа у двійковій системі числення для обчислення цілих степенів? Де виникає задача знаходження великих степенів чисел? Який алгоритм називають алгоритмом швидкого піднесення до степеня? Чим відрізняються схеми швидкого піднесення до степеня «справа наліво» і «зліва направо»? Чому схема «зліва направо» працює швидше від схеми «справа наліво»? Чи можна подати алгоритмом швидкого піднесення до степеня рекурсивно?

3. Де використовують випадковіф числа? Що лежить в основі методу статистичних випробувань (Монте-Карло) і де його використовують? Що таке псевдовипадкові числа? Що таке довжина періоду повторення генерованих чисел? Що є найважливішими характеристиками при генерації чисел? Що впливає на утворення повторюваних чисел? Вкажіть і схарактеризуйте методи генерації псевдовипадкових чисел. У чому полягає лінійний конгруентний метод? Який підхід можна застосувати для підвищення випадковості і для збільшення періоду при генерації чисел?

4. Що є характерною особливістю кодів Грея? Як можна одержати двійковий код Грея із двійкового подання числа? Чи існують алгоритми декодування кодів Грея? Чи можна згенерувати недвійковий код Грея і чи може кожен розряд коду мати різну основу?

5. Які задачі можна звести до задач генерації комбінаторних множин? Яка складність алгоритмів генерації комбінаторних множин? Як можна за-

стосувати бітові маски для генерації всіх підмножин? Чи можна застосувати як маски бінарні коди Грея? Що таке лексикографічний порядок послідовностей? Який алгоритм генерує у лексикографічному порядку перестановки без повторень? В якому алгоритмі застосовують циклічні зсуви? Який алгоритм генерує у лексикографічному порядку перестановки з повтореннями? Чи можна згенерувати лексикографічно впорядковану послідовність сполук? Який тип рекурсії використано в програмній реалізації алгоритму обертових дверей генерації сполук? В якій послідовності генеруються сполуки при використанні алгоритму обертових дверей? Що спільного і які відмінності в алгоритмах генерації лексикографічно впорядкованих послідовностей сполук без повторень і з повтореннями? На основі яких алгоритмів можна згенерувати поблочно лексикографічно впорядковані розміщення? Як згенерувати лексикографічно впорядковані розміщення з повтореннями? Які переваги має лексикографічна впорядкованість послідовності множин?

6. У чому полягає розкладання цілого числа на доданки? Як залежить кількість розкладань від значення самого числа? Як згенерувати послідовність розкладань числа у зворотному лексикографічному порядку? Що таке факторизація числа? Де використовують факторизацію великих чисел? Опишіть алгоритм Ферма? Які числа можна розкладати на множники з використанням алгоритму Ферма? Як можна оптимізувати обчислення з використанням алгоритму Ферма? Як розкласти на множники парне число? Яка обчислювальна складність алгоритму Ферма? Які складності інших агоритмів факторизації? Чи існує поліноміальний алгоритм факторизації чисел?

Практичне завдання № 16
«Виконання додаткового завдання»

Мета виконання завдання: продемонструвати вміння самостійно працювати і застосовувати набуті протягом семестру знання.

Завдання. Додаткове завдання кожен студент виконує за бажанням і вибирає самостійно із поданого нижче списку. На вибір для програмування пропонуються алгоритми дискретної математики:

1) виконання теоретико-множинних операцій над множинами;

2) виконання операцій над нечіткими множинами;

3) побудова замикань відношень відносно рефлексивності, симетричності й транзитивності (алгоритм Уоршалла);

4) побудова ДДНФ і ДКНФ за вектором значень логічної функції;

5) побудова полінома Жегалкіна методом переходу від вектора значень функції до вектора коефіцієнтів полінома;

6) побудова полінома Жегалкіна методом невизначених коефіцієнтів;

7) побудова скороченої ДНФ за вектором значень функції методом Квайна;

8) метод Петрика для побудови тупикових ДНФ за імплікантною таблицею Квайна;

9) алгоритм визначення дводольності графа;

10) алгоритм побудови ейлерового циклу в графі;

11) знаходження діаметра, радіуса і центра графа (матричний метод);

12) побудова каркасного дерева графа методом пошуку вглиб;

13) побудова каркасного дерева графа методом пошуку вшир;

14) метод Краскала побудови мінімального каркасу зваженого графа;

15) алгоритм Дейкстри знаходження найкоротшого шляху (ланцюга) у зваженому графі;

16) алгоритм Уелша-Пауелла розфарбовування графів;

17) алгоримт Дейкстри побудови зворотного польського запису
тощо.

Написати програму, яка, крім реалізації алгоритму, має модулі введення даних і виведення результатів обчислень у зручному для сприйняття вигляді. Передбачити можливість багаторазового виконання програми. Глобальні змінні необґрунтовано не використовувати. Коментарі в програмі обов’язкові (17-25 %).

Вибране завдання кожен студент виконує протягом семестру, а при захисті пояснює і обґрунтовує використання методів алгоритмізації, вказує на нюанси і труднощі, які виникли при програмній реалізації алгоритму, демонструє роботу програми з різними вхідними даними.

Список використаної літератури

1. Ахо А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман. — Москва : Мир, 1979. — 536 с.

2. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. — Москва : Изд. дом «Вильямс», 2010. — 384 с.

3. Бакнелл Дж. М. Фундаментальные алгоритмы и структуры данных в Delphi / Дж. М. Бакнелл. — Санкт-Петербург : ДиаСофтЮП, 2003. — 560 с.

4. Библиотека алгоритмов 1б-50б : справочное пособие / М. И. Агеев [и др.]. — Москва : Советское радио, 1975. — 176 с.

5. Бородкіна І. Л. Теорія алгоритмів: посіб. для студентів вищих навчальних закладів / І. Л. Бородкіна, Г. О. Бородкін. — Київ : Національний університет біоресурсів та природокористування України, 2018. — 231 с.

6. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих / А. Бхаргава. — Санкт-Петербург : Питер, 2017. — 288 с.

7. Вирт Н. Алгоритмы + структуры данных = программы / Н. Вирт. — Москва : Мир, 1985. — 406 с.

8. Гаврилович Я. Швидке піднесення до степеня [Електронний ресурс] / Я. Гаврилович. — Режим доступу: https://codeguida.com/post/2119 (дата звернення: 20.02.2022). — Назва з екрана.

9. Гарасимчук О. І. Генератори псевдовипадкових чисел, їх застосування, класифікація, основні методи побудови і оцінка якості [Електронний ресурс] / О. І. Гарасимчук, В. М. Максимович // Науково-технічний журнал «Захист інформації». — 2003. — № 30. — С. 29-36. — Режим доступу: http://jrnl.nau.edu.ua/index.php/ZI/article/viewFile/4270/4405 (дата звернення: 12.02.2022). — Назва з екрана.

10. Гласс Р. Руководство по надежному программированию / Р. Гласс. — Москва : Финансы и статистика, 1982. — 256 с.

11. Глибовець М. М. Основи комп’ютерних алгоритмів / М. М. Глибовець. — Київ : Видавничий дім «КМ Академія», 2003. — 452 с.

12. Горлова Т. М. Теорія алгоритмів: конспект лекцій [для студентів напряму підготовки 6.050101 «Комп’ютерні науки» денної та заочної форм навчання] [Електронний ресурс] / Т. М. Горлова, К. Є. Бобрівник, Н. В. Ліманська. — Київ : НУХТ, 2015. — 95 с. — Режим доступу: http://library.nuft.edu.ua/ebook/file/ M51.21.pdf (дата звернення: 05.02.2022). — Назва з екрана.

13. Губенко Н. Е. Эффективность использования генераторов псевдослучайных чисел в криптографических системах [Электронный ресурс] / Н. Е. Губенко, Е. А. Назаров // Информационные управляющие системы и компьютерный мониторинг (ИУС и КМ). — 2012. — С. 199-203. — Режим доступа: http://ea.donntu.org:8080/jspui/bitstream/123456789/30268/1/%D0%A1%D1%82%D0%B0%D1%82%D1%8C%D1%8F%205.pdf (дата обращения: 18.01.2022). — Название с экрана.

14. Дасгупта С. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани. — Москва : МЦНМО, 2014. — 320 с.

15. Ершов А. П. Введение в теоретическое программирование. Беседы о методе / А. П. Ершов. — Москва : Наука, 1977. — 288 с.

16. Жельников В. Криптография от папируса до компъютера / В. Жельников. — Москва : ABF, 1996. — 336 с.

17. Ишмухаметов Ш. Т. Методы факторизации натуральных чисел / Ш. Т. Ишмухаметов. — Казань: Казанский ун-т, 2011. — 190 с.

18. Керниган Б. Язык программирования Си / Б. Керниган, Д. Ритчи. — 3-е изд., испр. — Санкт-Петербург : Невский Диалект, 2001. — 352 с.

19. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 3-е изд. — Москва : Изд. дом «Вильямс», 2002. — Т. 1 : Основные алгоритмы. — 720 с.

20. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 3-е изд. — Москва : Изд. дом «Вильямс», 2001. — Т. 2 : Получисленные алгоритмы. — 832 с.

21. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 2-е изд. — Москва : Изд. дом «Вильямс», 2018. — Т. 3 : Сортировка и поиск. — 824 с.

22. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — Москва : Изд. дом «Вильямс», 2013. — Т. 4А : Комбинаторные алгоритмы, часть 1. — 960 с.

23. Кублій Л. І. Алгоритмізація та програмування. Практикум : навч. посіб. [для здобувачів ступеня бакалавра за спеціальністю 122 «Комп'ютерні науки»] [Електронний ресурс] / Л. І. Кублій. — Київ : КПІ ім. Ігоря Сікорського, 2019. — 209 с. — Режим доступу: ela.kpi.ua/handle/123456789/28216 (дата звернення: 27.04.2022). — Назва з екрана.

24. Кублій Л. І. Комп'ютерна дискретна математика (Частина 1): Розрахункова робота : навч. посіб. [для здобувачів ступеня бакалавра за освітньою програмою «Інженерія програмного забезпечення розподілених систем» спеціальності 121 «Інженерія програмного забезпечення»] [Електронний ресурс] / Л. І. Кублій. — Київ: КПІ ім. Ігоря Сікорського, 2020. — 165 с. — Режим доступу: ela.kpi.ua/handle/123456789/32323 (дата звернення: 27.04.2022). — Назва з екрана.

25. Лавров С. С. Введение в программирование / С. С. Лавров. — Москва : Наука, 1977. — 367 с.

26. Львов М. С. Основи алгоритмізації та програмування: навч. посіб. / М. С. Львов, О. В. Співаковський. — Херсон, 1997. — 371 с.

27. Мейер Б. Методы программирования: в 2 т. / Б. Мейер, К. Бодуэн. — Москва : Мир, 1982. — Т. 1. — 356 с.

28. Мікропроцесорна техніка: електронний підручник [Електронний ресурс] / Жуйков В. Я. [та ін.]. — К.: КПІ, 2016. — 440 с. — Режим доступу: https://ela.kpi.ua/bitstream/123456789/18969/1/МПТ_електронне_видання.pdf (дата звернення: 27.04.2022). — Назва з екрана.

29. Мороз Л. Прості алгоритми ділення чисел з плаваючою комою [Електронний ресурс] / Л. Мороз, А. Гринчишин, Я. Мірецька // Lviv Polytechnic National University Institutional Repository http://ena.lp.edu.ua. — 2016. — С. 35-38. — Режим доступу: http://195.22.112.37/bitstream/ntb/36472/1/6_35-38.pdf (дата звернення: 27.04.2022). — Назва з екрана.

30. Мороз Л. Модифікований метод швидкого обчислення оберненого квадратного кореня з використанням магічної константи [Електронний ресурс] / Л. Мороз, А. Гринчишин // Комп'ютерні технології друкарства. — 2015. — № 33. — С. 64-69. — Режим до-

ступу: http://nbuv.gov.ua/UJRN/ Ktd_2015_33_11 (дата звернення: 27.04.2022). — Назва з екрана.

31. Новотарський М. А. Алгоритми та методи обчислень : навч. посіб. [для студ. спеціальностей 121 «Інженерія програмного забезпечення», спеціалізації «Програмне забезпечення високопродуктивних комп'ютерних систем та мереж» та 123 «Комп'ютерна інженерія», спеціалізації «Комп'ютерні системи та мережі»] [Електронний ресурс] / Новотарський М. А. — Київ : КПІ ім. Ігоря Сікорського, 2019. — 407 с. — Режим доступу: https://ela.kpi.ua/bitstream/123456789/27864/1/Alhorytmy_ta_metody_obchislenn.pdf (дата звернення: 27.04.2022). — Назва з екрана.

32. Панкратова И. А. Теоретико-числовые методы в криптографии: учебное пособие / И. А. Панкратова. — Томск: Томский государственный университет, 2009. — 120 с.

33. Рисований О. М. Системне програмування: підручник / О. М. Рисований. — Харків : НТУ «ХПІ», 2010. — 912 с.

34. Свами М. Графы, сети и алгоритмы / М. Свами, К. Тхуласираман. — Москва : Мир, 1984. — 455 с.

35. Селін О. Алгоритми та структури даних: конспект лекцій [Електронний ресурс] / О. Селін. — Київ : Національний технічний університет України "Київський політехнічний інститут", 1998. — Режим доступу: http://mmsa.kpi.ua/sancho/ASD_HTM/index.html (дата звернення: 27.04.2022). — Назва з екрана.

36. Сліпченко В. Г. Структури даних мови Паскаль / В. Г. Сліпченко, Н. В. Ревинська. — Київ : КПІ, 1998. — 200 с.

37. Ткачук В. М. Алгоритми і структури даних: навч. посіб. / В. М. Ткачук. — Івано-Франківськ: Вид-во Прикарпатського національного університету ім. Василя Стефаника, 2016. — 286 с.

38. Хайнеман Дж. Алгоритмы. Справочник с примерами на C, C++, Java и Python / Дж. Хайнеман, Г. Пояяис, С. Сеяков. — 2-е изд. — Санкт-Петербург : ООО «Альфа-книга», 2017. — 432 с.

39. Хьюз Дж. Структурний подход к программированию / Дж. Хьюз, Дж. Мичтом. — Москва : Мир, 1980. — 280 с.

40. Curtis S. LeetCode 31 Next Permutation: Narayana Pandita's algorithm [Electronic resource] / Steven Curtis. — Mode of access: https://stevenpcurtis.medium.com/leetcode-31-next-permutation-narayana-panditas-algorithm-eb1dcb4686a7 (date of access: 27.04.2022). — Title from screen.

41. Fog Agner. 4. Instruction tables [Electronic resource]. — Technical University of Denmark, 22.03.2021. — 424 p. — Режим доступу: https://www.agner.org/optimize/instruction_tables.pdf (date of access: 23.12.2021). — Title from screen.

42. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985. — NY: IEEE, 1985. — 20 p.

43. IEEE Standard for Binary Floating-Point Arithmetic. IEEE Std 754-2008 (Revision of IEEE Std 754-1985). — NY: IEEE, 2008. — 70 p.

Посилання на список використаної літератури за розділами

1. Поняття алгоритму. Подання алгоритмів
[1, с. 11-15], [2, с. 16-37], [6, с. 18-39], [11, с. 14-18, 32-36, 170-183, 210-219, 241-245], [14, с. 49-74, 127-148, 156-175], [19, с. 27-36, 138-142], [31, с. 47-53], [38, с. 29-69].

2. Типи обчислювальних процесів. Різноманітність алгоритмів
[15, с. 14-27], [23, с. 15-17], [39, с. 72-81].

3. Позиційні і непозиційні системи числення
[14, с. 15-19, 49-52], [20, с. 226-303, 361-372], [22, с. 331-350], [23, с. 9-15], [24, с. 72-80], [29], [30] , [42, с. 7-14], [43, с. 6-38].

4. Лінійні алгоритми
[18, с. 61- 74, 251-252], [23, с. 18-20, 27-35], [26, с. 32-34], [28, с. 107-118], [33, с. 320-343, 879-881], [41, с. 177-188, 415-417].

5. Алгоритми з розгалуженнями
[23, с. 44-50], [26, с. 40-45], [34, с. 78-83].

6. Алгоритми з циклами
[5, с. 48-69], [23, с. 50-57, 59-86], [25, с. 99-107], [26, с. 47-50], [31, с. 34-39].

7. Алгоритми модульної структури
[1, с. 70-75], [2, с. 266-273], [7, с. 150-169], [10, с. 82-91], [12, с. 64-84], [14, с. 8-10], [23, с. 86-98], [26, с. 79-89], [37, с. 227-233] , [39, с. 29- 35, 232-238].

8. Обробка одновимірних масивів
[3, с. 49-51], [7, с. 14-30, 44-48, 74-89], [11, с. 170-183], [21, с. 19-28, 82-192, 425-456], [23, с. 102-123, 162-165, 199-202], [26, с. 51-62, 66-72, 77-78], [36, с. 10-30, 61-73].

9. Обробка двовимірних масивів
[23, с. 123-145], [26, с. 53-54], [27, с. 335-339], [36, с. 19-28, 69-70].

10. Алгоритмізація розв'язування задач
[4, с. 43], [19, с. 82-99], [35]; [8], [32, с. 7-11]; [9, с. 29-36], [13, с 199-203.], [16, с. 181-192], [20, с. 19-223]; [22, с. 331-349]; [11, с. 160-169], [22, с. 329-433], [40]; [11, с. 162-164], [17, с. 52-54], [22, с. 446-462], [32, с. 72-89].

Список рекомендованої літератури

1. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. — Москва : Изд. дом «Вильямс», 2010. — 384 с.

2. Вирт Н. Алгоритмы + структуры данных = программы / Н. Вирт. — Москва : Мир, 1985. — 406 с.

3. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 3-е изд. — Москва : Изд. дом «Вильямс», 2002. — Т. 1 : Основные алгоритмы. — 720 с.

4. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 3-е изд. — Москва : Изд. дом «Вильямс», 2001. — Т. 2 : Получисленные алгоритмы. — 832 с.

5. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — 2-е изд. — Москва : Изд. дом «Вильямс», 2018. — Т. 3 : Сортировка и поиск. — 824 с.

6. Кнут Д. Искусство программирования : в 4 т. / Д. Кнут. — Москва : Изд. дом «Вильямс», 2013. — Т. 4А : Комбинаторные алгоритмы, часть 1. — 960 с.

7. Кублій Л. І. Алгоритмізація та програмування. Практикум : навч. посіб. [для здобувачів ступеня бакалавра за спеціальністю 122 «Комп'ютерні науки»] [Електронний ресурс] / Л. І. Кублій. — Київ : КПІ ім. Ігоря Сікорського, 2019. — 209 с. — Режим доступу: ela.kpi.ua/handle/123456789/28216 (дата звернення: 27.04.2022). — Назва з екрана.

8. Новотарський М. А. Алгоритми та методи обчислень : навч. посіб. [для студ. спеціальностей 121 «Інженерія програмного забезпечення», спеціалізації «Програмне забезпечення високопродуктивних комп'ютерних систем та мереж» та 123 «Комп'ютерна інженерія», спеціалізації «Комп'ютерні системи та мережі»] [Електронний ресурс] / Новотарський М. А. — Київ : КПІ ім. Ігоря Сікорського, 2019. — 407 с. — Режим доступу: https://ela.kpi.ua/bitstream/123456789/27864/1/Alhorytmy_ta_metody_obchislenn.pdf (дата звернення: 27.04.2022). — Назва з екрана.

9. Хайнеман Дж. Алгоритмы. Справочник с примерами на C, C++, Java и Python / Дж. Хайнеман, Г. Поляис, С. Сеяков. — 2-е изд. — Санкт-Петербург : ООО «Альфа-книга», 2017. — 432 с.

Предметний покажчик

А
абсолютна величина числа, 20
адреса
значення, 234
повернення, 234
алгоритм, 9
Аль-Хорезмі, 15, 98
асимптотична складність, 13
Блюма-Блюма-Шуба (BBS), 442
визначення
кількості
різних елементів масиву, 329
одиниць у двійковому поданні, 212
номера дня в році, 214
типу трикутника, 151
однаковості
знаків чисел, 56
парності двох цілих чисел, 54
парності цілого числа, 52
властивості, 12
впорядкування
двох чисел, 30
трьох чисел, 240
графічне подання, 20
декодування коду Грея, 446
детермінований, 13
ділення
дійсних чисел, 102
цілих чисел, 99
додавання чисел, 94
Евкліда, 205, 256
двійковий (бінарний, Стайна), 209, 258
на основі віднімання (оригінальний), 205, 206, 256
на основі ділення, 205, 207, 238, 257
Ератосфена решето, 328
етапи створення, 9
ефективність, 11
ємнісна складність, 11
ітераційний, 167
зациклювання роботи програми, 176
зведення матриці до трапецеїдального (трикутного) вигляду, 379
злиття двох відсортованих масивів, 318
знаходження
максимуму, 130, 280
мінімуму, 130, 279, 280, 356
«машинного епсилона», 185
з циклами, 167
калькулятор, 184
Карацуби, 99
лінійний, 113
множення
прямокутних матриць, 378
чисел, 96
Нараяни, 461, 470
недетермінований, 13
обертових дверей, 476
обчислення
значення числа π, 194, 261
квадратного кореня, 197
кореня цілого степеня, 198
кубів цілих чисел, 181
номера дня в році, 214
оберненого квадратного кореня, 105
суми, 92, 94
факторіалу, 79, 251
великого числа, 325
перевірки правильності введення значення, 176, 255
переприсвоєння значень, 48
перестановка елементів масиву, 282, 356
на початок масиву, 287, 289
у кінець масиву, 288
у зворотному порядку, 283
переходу
від двійкового подання числа до
вісімкового, 70
десяткового, 64
шістнадцяткового, 71
від десяткового подання числа до двійкового, 67
до стандартних кодів Грея, 75
подання
графічне, 20
програмне, 24
словесне, 19
пошуку елемента в масиві, 290
бінарний, 295
з бар'єром (фіктивним елементом), 293
лінійний, 290
рекурсивний, 247
решето Ератосфена, 328

розв'язування квадратного
рівняння, 132, 138, 152
розгалужений, 143
розкладання натурального числа на
доданки, 489
множники, 493, 499
словесне подання, 19
скорочення
звичайного дробу, 237
цілого числа на 2, 192
сортування, 297
безглузде, 305
бульбашкою, 300
з індикатором перестановки, 302
вибором з пошуком мінімуму
(максимуму), 310
гномове, 306
масиву структур, 323
простими вставками, 310
швидке, 314
шейкерне, 304
з індикатором перестановки, 304
Шелла, 311
способи подання, 19
транспонування матриці
квадратної, 377
прямокутної, 377
факторизації натурального числа, 493, 499
Ферма, 494, 499
фон Неймана, 424
циклічний, 167, 218
цикломатична складність, 147
часова складність, 11
швидке піднесення чисел до цілих
степенів, 407
алгоритмічна структура
лінійна, 39
розгалуження, 39
цикл, 39
арифметичне переповнення, 47, 55, 124,
127, 190, 202, 398
арифметичний
вираз, 119
зсув, 124

Б

безглузде сортування, 305
бінарний
пошук, 293
алгоритм
Евкліда, 209, 258
піднесення до степеня, 407
бітова маска, 454, 458, 476
блок-схема, 20, 30, 33, 40, 148, 162, 510

В

визначення
кількості одиниць у двійковому
поданні числа, 212
однаковості
знаків чисел, 56
парності цілих чисел, 54
парності цілого числа, 52
вимірність масиву, 274, 352
випадкові (псевдовипадкові) числа, 275,
354, 419
рівномірно розподілені, 418, 424
вираз, 113
арифметичний, 119
логічний, 54, 120, 157
обчислення, 122, 159
віртуальна пам'ять, 318
віртуальний комп'ютер, 49, 69, 92
вісімкова система числення, 70
властивості алгоритму, 12
впорядкування
двох чисел, 240
трьох чисел, 240
втрата молодших розрядів, 47, 49

Г

генератор випадкових (псевдовипадкових)
чисел, 275, 354
генерація
випадкових чисел, 275, 354, 418
дійсних випадкових чисел, 276
комбінаторних множин, 453
недвійкових рефлексивних кодів
Грея, 77, 449
перестановок
алгоритм Нараяни, 461, 470
без повторень, 461
за допомогою циклічних зсувів, 466
з повтореннями, 470
підмножин множини, 453
з використанням
бінарних кодів Грея, 456
маски, 454
масиву-маски, 458
побітових операцій, 454
розміщень
без повторень, 483
з повтореннями, 486

сполук
алгоритм обертових дверей, 475
без повторень, 471
з повтореннями, 480
глибина рекурсії, 249
гномове сортування, 306
границі циклу, 178
граф керування алгоритму, 148
графічне подання алгоритму, 20

Д

двійкова система числення, 63, 78
двійковий дріб
неперіодичний, 67
нескінченний, 67, 88
періодичний, 67, 88
скінченний, 67
денормалізований експоненційний вигляд числа, 83
дерево рекурсивних викликів, 268, 281
десяткова система числення, 62
детермінований алгоритм, 13
динамічне програмування, 35
динамічний масив, 273, 333, 335, 341
діалог з користувачем, 177
ділення
навпіл, 35
дійсних чисел, 102
цілих чисел, 99
довжина періоду (циклу) повторення псевдовипадкових чисел, 420
додатковий код числа, 78
доступ до елементів масиву, 274, 352

Е

експонента, 82
зміщена, 83
елемент масиву, 273
індексування, 274
етапи розв’язування задачі, 9
ефективність алгоритму, 11

Є

ємнісна складність алгоритму, 11, 18

З

затримка переходу, 135
захист від механічних помилок, 175
зациклювання, 167
роботи програми, 176
зворотний код числа, 78
злиття відсортованих масивів, 318
змінна
глобальна, 28, 115, 236, 264
локальна, 115
змішана система числення, 63
зсув
логічний, 124
арифметичний, 124

І

ідентифікатор, 27
ітераційний алгоритм, 167

К

кількість сполук, 392
ключ сортування, 298
ключове слово, 27
код числа додатковий, 78
коди Грея, 62, 74, 443
бінарні (двійкові), 74, 456
віддзеркалені (стандартні), 75, 443
декодування, 446
небінарні (недвійкові), 77, 449
рефлексивні десяткові, 77, 449
коментар, 24, 29
комп’ютерне експериментальне дослідження, 418
корінь цілого степеня, 198
криптографія, 415, 440, 493
крок дискретизації, 84

Л

лексикографічно
безпосередньо наступна послідовність, 461
впорядкована послідовність, 461
логічний
вираз,120
зсув, 124

М

«магічне» число, 102, 196
магічний квадрат, 382
непарного порядку, 382
максимум двох чисел, 130
мантиса, 82
залишок, 83
масив, 273
безрозмірний, 275, 353
багатовимірний, 352

введення, 276, 354
виведення, 276, 354
двовимірний, 352
динамічний, 273, 333, 335, 341
доступ до елементів, 274, 353
ініціалізація, 275, 353
копіювання, 276, 355
одновимірний, 274
передача параметра, 277, 355
порівняння, 276, 355
статичний, 273
маска бітова, 454, 458, 476
матриця трапецеїдального (трикутного) вигляду, 379
«машинний епсилон», 185
«машинний нуль», 187
мемоїзація, 36, 219, 225, 268
метод
бульбашки, 300
з запам'ятовуванням місця останньої перестановки, 303
з індикатором перестановки, 16, 302
лінійний конгруентний, 435
Ньютона (дотичних), 73, 102, 196
перемішування (шейкер-сортування), 304
простого вибору з пошуком мінімуму чи максимуму, 310
простих вставок, 310
середин
добутків, 427
квадратів (алгоритм фон Неймана), 424
статистичних випробувань (Монте-Карло), 418, 422
метрика Мак-Кейба, 147, 229
мінімум двох чисел, 130
множення
алгоритм Аль-Хорезмі, 15, 98
метод Карацуби, 99
множина
вихідна (відсортована), 300
вхідна (не відсортована), 300
мова програмування
Ада, 184
Delphi, 338
Оберон-2, 169, 184
Pascal, 31, 69, 505
C, 28, 505
Fortran, 338
модуль, 148, 229
властивості, 230
з побічним ефектом, 231, 237
недоліки, 230
переваги, 229
повністю закритий, 231
рекурсивний, 247
непрямо (взаємно), 248
прямо, 247
частково закритий, 231
модульне програмування, 229

Н

найбільший спільний дільник (НСД), 205
двох чисел, 205
кількох чисел, 330
недетермінований алгоритм, 13
неперіодичний (скінченний) двійковий дріб, 67
непозиційна система числення, 61
нескінченний двійковий дріб, 67
неявний розряд, 83, 86
нормалізований експоненційний вигляд числа, 83

О

операції
над двійковими числами, 72
побітові (порозрядні), 50, 123
оптимізація
алгоритму
з розгалуженнями, 157
ітераційного, 218
лінійного, 133
модульної структури, 266
обробки двовимірних масивів, 385
обробки одновимірних масивів, 335
обчислень, 133

П

параметр
масив, 277, 355
фактичний, 234
формальний, 234
перегляд елементів матриці в певному порядку, 366, 367, 368
передача параметра за
адресою, 235
значенням, 235
посиланням, 235
переповнення
арифметичне, 47, 55, 124, 127, 190, 202, 398
стека, 18, 248, 266
перепривсоєння значень двох змінних, 47

перестановка елементів масиву, 282
у зворотному порядку, 283
перестановки, 461
перехід від
двійкового подання числа до
вісімкового, 70
десяткового, 64
шістнадцяткового, 71
десяткового подання числа до
двійкового, 67
періодичний двійковий дріб, 67
піднесення до степеня, 407
за модулем, 415
швидке, 408
підпрограма, 230, 247
подання алгоритму
графічне, 20
програмне, 24
словесне, 19
позиційна система числення, 61
помилка
двійкового подання дробів, 68
заокруглення, 69, 85
точності подання, 86
помилкове щодо числового формату
значення, 176
порядок
доданків, 92
лексикографічний, 461
похибка, 85, 91, 104, 112
абсолютна, 197
відносна, 197
допустима, 120, 215
пошук, 290
бінарний, 295
з бар'єром (фіктивним елементом), 293
лінійний, 290
усіх однакових елементів, 294
правило заокруглення, 85
правильність введення даних, 176, 242, 255
принцип модульності програмування, 233
природна поведінка алгоритму
сортування, 298
«проблема кроликів і черепах», 304
програмне подання алгоритму, 24
програмування
динамічне, 35
згори вниз, 232
знизу вгору, 232
модульне, 229
структурне, 39, 46, 476
проектування
згори вниз, 232
знизу вгору, 232
простими вставками сортування, 310

Р

рекурентне визначення, 249
рекурсивна підпрограма, 247
рекурсивний
алгоритм, 247
підйом, 249
спуск, 249
рекурсія, 247
глибина, 249
непряма (взаємна), 248
пряма, 247
рефлексивний десятковий код Грея, 77
решето Ератосфена, 328
розгалуження, 143
розділяй і володарюй, 35, 99, 295, 314
розкладання натурального числа на
доданки, 489
прості множники, 418, 493
розміщення, 483
з повтореннями, 486
розріджена матриця, 373
виведення, 374
розряд числа
знаковий, 79
молодший, 49, 53, 79, 454
старший, 79, 454
розрядна сітка, 79
рядок символів, 275, 278, 371

С

сектор матриці, 363, 522
система числення, 61
вісімкова, 63
двійкова, 63
десяткова, 63
змішана, 63
коди Грея, 62, 443
непозиційна, 61
позиційна, 61
факторіальна, 63
Фібоначчі, 63
шістнадцяткова, 63
скінченний двійковий дріб, 67
складність
алгоритму, 13
асимптотична, 13, 14

бінарного пошуку, 336
в найгіршому випадку, 16
в найкращому випадку, 16
в середньому, 16
ємнісна, 11, 18
з розгалуженнями, 157
лінійного, 133
часова, 11, 16
модульної структури, 266
обробки
двовимірних масивів, 385
одновимірних масивів, 335
поліноміальна, 13
рекурсивного, 267
цикломатична, 147
циклічного, 218
швидкого сортування, 336
алгоритмів
пошуку, 335
сортування, 335
виразу, 133
скорочення
звичайного дробу, 237
цілого числа на 2, 192
словесне подання алгоритму, 19
сортування, 297
адаптивний алгоритм, 298
безглузде, 305
вставками, 299
гномове, 306
елементів двовимірного масиву, 369
з вибором, 299
ключ, 298
масиву
рядків символів, 372
структур, 323
«на місці», 298
обміном, 299
простими вставками, 310
прямий метод, 299
поліпшений метод, 299
стабільний алгоритм, 298
швидке, 314
співвідношення рекурентне, 63, 75, 103, 201, 219, 249, 396
сполука, 392, 471
сполуки з повтореннями, 480
способи подання алгоритму, 19
стандарт IEEE 754, 82
стандартні (віддзеркалені) коди Грея, 75
статичний масив, 273
стек
викликів, 234, 253
переповнення, 18, 248, 266
стиль
подання коду програми, 28, 144
програмування, 29, 146, 173, 237
стиснена матриця, 373
стратегія алгоритмізації, 34
динамічне програмування, 35
ділення навпіл, 35
жадібна, 34
розділяй і володарюй, 35, 99, 295, 314
структура даних, 273
інтегрована (складна), 273
проста, 273
структурне програмуванням, 39, 46, 476
схема
Горнера, 65, 137, 140, 193
мультиплікативний аналог, 413
Фібоначчі із запізненням, 441

Т

таблиця Келі, 180
таблиця символів
ANSI, 190
ASCI, 59, 114, 116, 190
табулювання функції, 215
такт, 133
теорема Вієта, 138
теорія чисел, 201
тіло циклу, 167
транспонування матриці,377
трикутник Паскаля, 401

У

умова (логічний вираз)
проста, 120
складена, 121

Ф

факторизація натурального числа, 418, 493
факторіал великого числа, 325
Ферма алгоритм, 494, 499
Фібоначчі система числення, 63
формат
дійсного числа, 82
цілого числа, 78
формула
Герона, 20, 131, 197
рекурентна, 63, 75, 103, 201, 219, 249, 396

фрагмент (сектор) матриці, 363, 522
функція
  Ейлера, 438
  задана логічно, 24, 158
  знака, 151

Ц

цикл, 167
  детермінований, 167
  з двома параметрами, 181
  з параметром, 168
  з передумовою, 168
  з післяумовою, 168
  з порожнім тілом, 171, 177
  ітераційний, 167
цикли
  вкладені, 167, 170, 180, 221
    залежні, 170, 180
    незалежні, 170, 180
циклічний
  алгоритм, 167, 218
  зсув, 284
    вліво, 284, 340, 430, 466
    вправо, 284, 340, 430
цикломатична складність алгоритму, 147
цикломатичне число графа, 148

Ч

часова складність алгоритму, 11, 16
числа Фібоначчі, 63, 201, 253, 268
число
  π, 194, 261
  просте, 204, 328
  цикломатичне гафа, 148

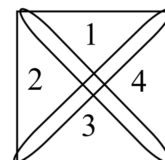
Ш

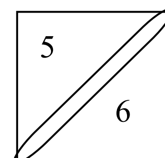
швидке
  обчислення оберненого квадратного кореня, 105
  піднесення до степеня, 407
    за схемою
      «зліва направо», 413
      «справа наліво», 409
  сортування, 314
шейкерне сортування, 304
  з індикатором перестановки, 304
Шелла сортування, 311
шістнадцяткова система
  числення, 62, 70

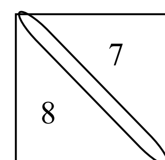
Додаток А

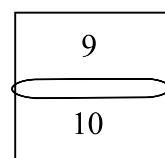
Обхід секторів квадратної матриці (мови С і Pascal)

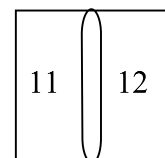
Розглянемо для квадратної матриці розміру $n \times n$ алгоритми вибору елементів з її частин-секторів (деякі із цих алгоритмів розглянуто в п. 9.2). Незалежно від того, порядок матриці n є парним чи непарним числом, в алгоритмах використано ту властивість, що в мові С результатом ділення двох цілих чисел є ціле число — дробова частина відкидається, а в мові Pascal використано операцію цілочисельного ділення, результатом якої теж є число з відкинутою дробовою частиною.











Подамо алгоритми обходу секторів квадратної матриці, мовами С і Pascal. Особливості цих алгоритмів полягають у тому, що в мові С індексація елементів завжди починається з індекса 0, а в мові Pascal нижньою границею індексів може бути будь-яке число (найчастіше беруть 1; також візьмемо 1).

Щоб краще зрозуміти, як працюють ці алгоритми, можна скористатися матрицями індексів, наприклад, розмірів 4×4 (n — парне) і 5×5 (n — непарне).

Мова С

$$ij\begin{pmatrix} \mathbf{00} & 01 & 02 & \mathbf{03} \\ 10 & \mathbf{11} & \mathbf{12} & 13 \\ 20 & \mathbf{21} & \mathbf{22} & 23 \\ \mathbf{30} & 31 & 32 & \mathbf{33} \end{pmatrix} \quad ij\begin{pmatrix} \mathbf{00} & 01 & 02 & 03 & \mathbf{04} \\ 10 & \mathbf{11} & 12 & \mathbf{13} & 14 \\ 20 & 21 & \mathbf{22} & 23 & 24 \\ 30 & \mathbf{31} & 32 & \mathbf{33} & 34 \\ \mathbf{40} & 41 & 42 & 43 & \mathbf{44} \end{pmatrix}$$

Мова Pascal

$$ij\begin{pmatrix} \mathbf{11} & 12 & 13 & \mathbf{14} \\ 21 & \mathbf{22} & \mathbf{23} & 24 \\ 31 & \mathbf{32} & \mathbf{33} & 34 \\ \mathbf{41} & 42 & 43 & \mathbf{44} \end{pmatrix} \quad ij\begin{pmatrix} \mathbf{11} & 12 & 13 & 14 & \mathbf{15} \\ 21 & \mathbf{22} & 23 & \mathbf{24} & 25 \\ 31 & 32 & \mathbf{33} & 34 & 35 \\ 41 & \mathbf{42} & 43 & \mathbf{44} & 45 \\ \mathbf{51} & 52 & 53 & 54 & \mathbf{55} \end{pmatrix}$$

Опис масиву (матриця $A_{n \times n}$)

```
int a[n][n];
```

```
type matr=array[1..n,1..n] of
          integer;
var a: matr;
```

Сектор 1 без елементів діагоналей

```
for (i=0; i<(n-1)/2; i++)
  for (j=i+1; j<n-i-1; j++) a[i][j]=1;
```

```
for i:=1 to ((n-1) div 2) do
  for j:=i+1 to n-i do a[i,j]:=1;
```

Сектор 1 з елементами діагоналей

```
for (i=0; i<(n+1)/2; i++)
```

```
for i:=1 to ((n+1) div 2) do
```

```
  for (j=i; j<n-i; j++) a[i][j]=1;     |  for j:=i to n-i+1 do a[i,j]:=1;
```

Сектор 2 без елементів діагоналей

```
for (i=1; i<n-1; i++) {
  if (i<(n+1)/2) jr=i;
  else jr=n-i-1;
  for (j=0; j<jr; j++) a[i][j]=2;
}
```

```
for i:=2 to n-1 do
  begin
    if i<=((n+1) div 2)
      then jr:=i-1
      else jr:=n-i;
    for j:=1 to jr do a[i,j]:=2;
  end;
```

Сектор 2 без елементів діагоналей
(як і сектор 1, тільки в заголовках циклів поміняно *i* й *j*)

```
for (j=0; j<(n-1)/2; j++)
  for (i=j+1; i<n-j-1; i++) a[i][j]=2;
```

```
for j:=1 to ((n-1) div 2) do
  for i:=j+1 to n-j do a[i,j]:=2;
```

Сектор 2 з елементами діагоналей

```
for (i=0; i<n; i++) {
  if (i<(n+1)/2) jr=i+1;
  else jr=n-i;
  for (j=0; j<jr; j++) a[i][j]=2;
}
```

```
for i:=1 to n do
  begin
    if i<=((n+1) div 2)
      then jr:=i
      else jr:=n-i+1;
    for j:=1 to jr do a[i,j]:=2;
  end;
```

Сектор 2 з елементами діагоналей
(як і сектор 1, тільки в заголовках циклів поміняно *i* й *j*)

```
for (j=0; j<(n+1)/2; j++)
  for (i=j; i<n-j; i++) a[i][j]=2;
```

```
for j:=1 to (n+1) div 2 do
  for i:=j to n-j+1 do a[i,j]:=2;
```

Сектор 3 без елементів діагоналей

```
for (i=n/2+1; i<n; i++)
  for (j=n-i; j<i; j++) a[i][j]=3;
```

```
for i:=(n div 2)+2 to n do
  for j:=n-i+2 to i-1 do a[i,j]:=3;
```

Сектор 3 з елементами діагоналей

```
for (i=n/2; i<n; i++)
  for (j=n-i-1; j<i+1; j++) a[i][j]=3;
```

```
for i:=((n div 2)+1 to n do
  for j:=n-i+1 to i do a[i,j]:=3;
```

Сектор 4 без елементів діагоналей

```
for (i=1; i<n-1; i++) {
  if (i<(n+1)/2) jr=n-i;
  else jr=i+1;
  for (j=jr; j<n; j++) a[i][j]=4;
}
```

```
for i:=2 to n-1 do
  begin
    if i<=((n+1) div 2)
      then jr:=n-i+2
      else jr:=i+1;
    for j:=jr to n do a[i,j]:=4;
  end;
```

Сектор 4 без елементів діагоналей
(як і сектор 3, тільки в заголовках циклів поміняно *i* й *j*)

```
for (j=n/2+1; j<n; j++)
  for (i=n-j; i<j; i++) a[i][j]=4;
```

```
for j:=(n div 2)+2 to n do
  for i:=n-j+2 to j-1 do a[i,j]:=4;
```

Сектор 4 з елементами діагоналей

```
for (i=0; i<n; i++) {
  if (i<(n+1)/2) jr=n-i-1;
  else jr=i;
  for (j=jr; j<n; j++) a[i][j]=4;
}
```

```
for i:=1 to n do
  begin
    if i<=((n+1) div 2)
      then jr:=n-i+1
      else jr:=i;
    for j:=jr to n do a[i,j]:=4;
  end;
```

Сектор 4 з елементами діагоналей

(як і сектор 3, тільки в заголовках циклів поміняно *i* й *j*)

```
for (j=n/2; j<n; j++)
  for (i=n-j-1; i<j+1; i++) a[i][j]=4;
```

```
for j:=(n div 2)+1 to n do
  for i:=n-j+1 to j do a[i,j]:=4;
```

Сектор 5 без елементів діагоналі

```
for (i=0; i<n-1; i++)
  for (j=0; j<n-i-1; j++) a[i][j]=5;
```

```
for i:=1 to n-1 do
  for j:=1 to n-i do a[i,j]:=5;
```

Сектор 5 з елементами діагоналі

```
for (i=0; i<n; i++)
  for (j=0; j<n-i; j++) a[i][j]=5;
```

```
for i:=1 to n do
  for j:=1 to n-i+1 do a[i,j]:=5;
```

Сектор 6 без елементів діагоналі

```
for (i=1; i<n; i++)
  for (j=n-i; j<n; j++) a[i][j]=6;
```

```
for i:=2 to n do
  for j:=n-i+2 to n do a[i,j]:=6;
```

Сектор 6 з елементами діагоналі

```
for (i=0; i<n; i++)
  for (j=n-i-1; j<n; j++) a[i][j]=6;
```

```
for i:=1 to n do
  for j:=n-i+1 to n do a[i,j]:=6;
```

Сектор 7 без елементів діагоналі

```
  for (i=0; i<n-1; i++)
for (j=i+1; j<n; j++) a[i][j]=7;
```

```
for i:=1 to n-1 do
  for j:=i+1 to n do a[i,j]:=7;
```

Сектор 7 з елементами діагоналі

```
for (i=0; i<n; i++)
  for (j=i; j<n; j++) a[i][j]=7;
```

```
for i:=1 to n do
  for j:=i to n do a[i,j]:=7;
```

Сектор 8 без елементів діагоналі

```
for (i=1; i<n; i++)
  for (j=0; j<i; j++) a[i][j]=8;
```

```
for i:=2 to n do
  for j:=1 to i-1 do a[i,j]:=8;
```

Сектор 8 з елементами діагоналі

```
for (i=0; i<n; i++)
  for (j=0; j<i+1; j++) a[i][j]=8;
```

```
for i:=1 to n do
  for j:=1 to i do a[i,j]:=8;
```

Сектор 9 без елементів серединного рядка

```
for (i=0; i<n/2; i++)
  for (j=0; j<n; j++) a[i][j]=9;
```

```
for i:=1 to (n div 2) do
  for j:=1 to n do a[i,j]:=9;
```

Сектор 9 з елементами серединного рядка при непарних *n*

```
for (i=0; i<(n+1)/2; i++)
  for (j=0; j<n; j++) a[i][j]=9;
```

```
for i:=1 to ((n+1) div 2) do
  for j:=1 to n do a[i,j]:=9;
```

Сектор 10 без елементів серединного рядка

```
for (i=(n+1)/2; i<n; i++)
  for (j=0; j<n; j++) a[i][j]=10;
```

```
for i:=((n+1) div 2)+1 to n do
  for j:=1 to n do a[i,j]:=10;
```

Сектор 10 з елементами серединного рядка при непарних *n*

```
for (i=n/2; i<n; i++)
  for (j=0; j<n; j++) a[i][j]=10;
```

```
for i:=(n div 2)+1 to n do
  for j:=1 to n do a[i,j]:=10;
```

Сектор 11 без елементів серединного стовпчика

```
for (i=0; i<n; i++)
  for (j=0; j<n/2; j++) a[i][j]=11;
```

```
for i:=1 to n do
  for j:=1 to (n div 2) do a[i,j]:=11;
```

Сектор 11 з елементами серединного стовпчиком при непарних *n*

```
for (i=0; i<n; i++)
  for (j=0; j<(n+1)/2; j++) a[i][j]=11;
```

```
for i:=1 to n do
  for j:=1 to ((n+1) div 2) do a[i,j]:=11;
```

Сектор 12 без елементів серединного стовпчика

```
for (i=0; i<n; i++)
  for (j=(n+1)/2; j<n; j++) a[i][j]=12;
```

```
for i:=1 to n do
  for j:=((n+1) div 2) to n do a[i,j]:=12;
```

Сектор 12 з елементами серединного стовпчика при непарних *n*

```
for (i=0; i<n; i++)
  for (j=n/2; j<n; j++) a[i][j]=12;
```

```
for i:=1 to n do
  for j:=(n div 2)+1 to n do a[i,j]:=12;
```

Подані алгоритми вибору секторів квадратних матриць є найефективнішими, оскільки в них цикли виконуються тільки для тих елементів, які мають бути вибрані, і в них нема зайвих перевірок. Але, як правило, такий алгоритм важко запам’ятати і згадати.

Існує інший, менш ефективний, але дуже простий спосіб вибору елементів матриці, в якому використовуються властивості суми й різниці індексів — переглядаються всі можливі n^2 комбінацій індексів і серед них вибираються потрібні (при цьому проходів циклів виконується приблизно в 2 чи 4 рази більше і, крім того, виконуються перевірки в цих циклах, тобто машинний час витрачається неефективно). Розглянемо суми й різниці індексів для матриць 4×4 і 5×5.

У мові C:

$$ij\ \begin{pmatrix} \mathbf{00} & 01 & 02 & \mathbf{03} \\ 10 & \mathbf{11} & \mathbf{12} & 13 \\ 20 & \mathbf{21} & \mathbf{22} & 23 \\ \mathbf{30} & 31 & 32 & \mathbf{33} \end{pmatrix} \quad i+j\ \begin{pmatrix} 0 & 1 & 2 & \mathbf{3} \\ 1 & 2 & \mathbf{3} & 4 \\ 2 & \mathbf{3} & 4 & 5 \\ \mathbf{3} & 4 & 5 & 6 \end{pmatrix} \quad i-j\ \begin{pmatrix} \mathbf{0} & -1 & -2 & -3 \\ 1 & \mathbf{0} & -1 & -2 \\ 2 & 1 & \mathbf{0} & -1 \\ 3 & 2 & 1 & \mathbf{0} \end{pmatrix}$$

$$ij\ \begin{pmatrix} \mathbf{00} & 01 & 02 & 03 & \mathbf{04} \\ 10 & \mathbf{11} & 12 & \mathbf{13} & 14 \\ 20 & 21 & \mathbf{22} & 23 & 24 \\ 30 & \mathbf{31} & 32 & \mathbf{33} & 34 \\ \mathbf{40} & 41 & 42 & 43 & \mathbf{44} \end{pmatrix} \quad i+j\ \begin{pmatrix} 0 & 1 & 2 & 3 & \mathbf{4} \\ 1 & 2 & 3 & \mathbf{4} & 5 \\ 2 & 3 & \mathbf{4} & 5 & 6 \\ 3 & \mathbf{4} & 5 & 6 & 7 \\ \mathbf{4} & 5 & 6 & 7 & 8 \end{pmatrix} \quad i-j\ \begin{pmatrix} \mathbf{0} & -1 & -2 & -3 & -4 \\ 1 & \mathbf{0} & -1 & -2 & -3 \\ 2 & 1 & \mathbf{0} & -1 & -2 \\ 3 & 2 & 1 & \mathbf{0} & -1 \\ 4 & 3 & 2 & 1 & \mathbf{0} \end{pmatrix}$$

З поданих матриць видно, що для $i+j$ на побічній діагоналі стоять числа $(n-1)$, вище від неї — числа, менші від $(n-1)$, а нижче — більші від $(n-1)$; для $i-j$ на головній діагоналі стоять нулі, вище від неї — від’ємні числа, а нижче — додатні числа.

У мові Pascal:

$$ij\ \begin{pmatrix} \mathbf{11} & 12 & 13 & \mathbf{14} \\ 21 & \mathbf{22} & \mathbf{23} & 24 \\ 31 & \mathbf{32} & \mathbf{33} & 34 \\ \mathbf{41} & 42 & 43 & \mathbf{44} \end{pmatrix} \quad i+j\ \begin{pmatrix} 2 & 3 & 4 & \mathbf{5} \\ 3 & 4 & \mathbf{5} & 6 \\ 4 & \mathbf{5} & 6 & 7 \\ \mathbf{5} & 6 & 7 & 8 \end{pmatrix} \quad i-j\ \begin{pmatrix} \mathbf{0} & -1 & -2 & -3 \\ 1 & \mathbf{0} & -1 & -2 \\ 2 & 1 & \mathbf{0} & -1 \\ 3 & 2 & 1 & \mathbf{0} \end{pmatrix}$$

$$
\underset{ij}{\begin{pmatrix} \mathbf{11} & 12 & 13 & 14 & \mathbf{15} \\ 21 & \mathbf{22} & 23 & \mathbf{24} & 25 \\ 31 & 32 & \mathbf{33} & 34 & 35 \\ 41 & \mathbf{42} & 43 & \mathbf{44} & 45 \\ \mathbf{51} & 52 & 53 & 54 & \mathbf{55} \end{pmatrix}}
\quad
\underset{i+j}{\begin{pmatrix} 2 & 3 & 4 & 5 & \mathbf{6} \\ 3 & 4 & 5 & \mathbf{6} & 7 \\ 4 & 5 & \mathbf{6} & 7 & 8 \\ 5 & \mathbf{6} & 7 & 8 & 9 \\ \mathbf{6} & 7 & 8 & 9 & 10 \end{pmatrix}}
\quad
\underset{i-j}{\begin{pmatrix} \mathbf{0} & -1 & -2 & -3 & -4 \\ 1 & \mathbf{0} & -1 & -2 & -3 \\ 2 & 1 & \mathbf{0} & -1 & -2 \\ 3 & 2 & 1 & \mathbf{0} & -1 \\ 4 & 3 & 2 & 1 & \mathbf{0} \end{pmatrix}}
$$

У цьому випадку для $i+j$ на побічній діагоналі стоять числа $(n+1)$, вище від неї — числа, менші від $(n+1)$, а нижче — більші від $(n+1)$; для $i-j$ на головній діагоналі стоять нулі, вище від неї — від'ємні числа, а нижче — додатні числа.

Використовуючи властивості сум і різниць індексів, можна вибирати певні сектори матриць. Розглянемо, наприклад, алгоритми для секторів 1-4, а для решти секторів, пов'язаних з діагоналями матриці, — аналогічно. Алгоритми обходу секторів 1-4 мають однаковий вигляд і відрізняються лише умовами.

Мова C	Мова Pascal

Загальний вигляд алгоритму (*k* — номер сектора)

```
for (i=0; i<n; i++)                    | for i:=1 to n do
  for (j=0; j<n; j++)                  |   for j:=1 to n do
    if (умова) a[i][j]=k;              |     if умова then a[i,j]:=k;
```

Сектор 1 без елементів діагоналей

```
if (i+j<n-1 && i-j<0) a[i][j]=1;       | if (i+j<n+1)and(i-j<0) then a[i,j]:=1;
```

Сектор 1 з елементами діагоналей

```
if (i+j<=n-1 && i-j<=0) a[i][j]=1;     | if (i+j<=n+1)and(i-j<=0) then a[i,j]:=1;
```

Сектор 2 без елементів діагоналей

```
if (i+j<n-1 && i-j>0) a[i][j]=2;       | if (i+j<n+1)and(i-j>0) then a[i,j]:=2;
```

Сектор 2 з елементами діагоналей

```
if (i+j<=n-1 && i-j>=0) a[i][j]=2;     | if (i+j<=n+1)and(i-j>=0) then a[i,j]:=2;
```

Сектор 3 без елементів діагоналей

```
if (i+j>n-1 && i-j>0) a[i][j]=3;       | if (i+j>n+1)and(i-j>0) then a[i,j]:=3;
```

Сектор 3 з елементами діагоналей

```
if (i+j>=n-1 && i-j>=0) a[i][j]=3;     | if (i+j>=n+1)and(i-j>=0) then a[i,j]:=3;
```

Сектор 4 без елементів діагоналей

```
if (i+j>n-1 && i-j<0) a[i][j]=4;       | if (i+j>n+1)and(i-j<0) then a[i,j]:=4;
```

Сектор 4 з елементами діагоналей

```
if (i+j>=n-1 && i-j<=0) a[i][j]=4;     | if (i+j>=n+1)and(i-j<=0) then a[i,j]:=4;
```

З поданих алгоритмів видно, що якщо елементи діагоналі входять в сектор матриці, то використовується порівняння <= чи >=, а якщо не входять, — то < чи >.

Додаток Б

Орієнтовні завдання до модульної контрольної роботи

Мета модульної контрольної роботи полягає в перевірці засвоєння матеріалу дисципліни. На контрольну роботу виносяться завдання, пов’язані з програмуванням арифметичних виразів, вкладених логічних операторів, обробкою одновимірних і двовимірних масивів, алгоритми сортування. Робота виконується без використання комп’ютера.

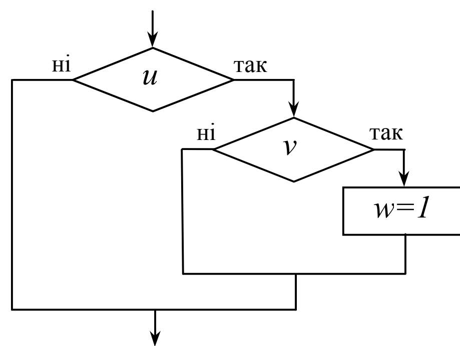
Зразок завдання.

1. Запишіть вираз з мінімальною кількістю дужок, дотримуючись правил і використовуючи функції мови С:

$$\frac{1}{8}\sqrt{\frac{5x-a^3}{x^2+9}}+ctg\left|x-e^{0,8x}\right|+\frac{1}{9}$$

Порядок операндів не змінюйте. Вважайте, що змінні і результат мають тип double.

2. За поданою блок-схемою запишіть умовний оператор мовою С[27]:



3. Запишіть результат виконання поданого фрагменту програми:

```
#define N 5
int main() {
  int i, k;
  int a[N]={-3,-6,7,-3,-5};
  ..........
  for(i=0, k=0; i<N && k<2; i++)
    if (a[i]>0) k++;
```

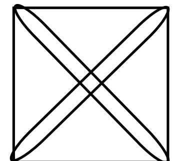
[27] У завданнях варіантів блок-схеми відповідають вкладеним умовним операторам різної складності.

```
  if (k==3) printf("%d, %d",i-1,a[i-1]);
  else printf("Немає двох додатних елементів");
  ..........
}
```

4. Ідентифікуйте метод сортування; вкажіть, з якого кінця масиву формується вихідна множина; подайте результат виконання фрагменту програми:

```
#define N 5
int main() {
  int i, j;
  char r;
  char a[N]={'b','c','a','d','e'};
  ..........
  for(i=1; i<N; i++)
    for (j=N-1; j>=i; j--)
      if (a[j-1]>a[j]) {
        r=a[j-1]; a[j-1]=a[j]; a[j]=r; }
  for (i=0; i<N; i++) printf("%3c,", a[i]);
  ..........
}
```

5. Перемалюйте шаблон матриці і акуратно заштрихуйте в ньому сектор (з частинами діагоналей чи без), який обробляється поданим фрагментом програми:



```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    if (i+j??n-1 && i-j??0)
      a[i][j]=0;
```

Додаток В

Запитання і орієнтовні завдання до залікової контрольної роботи

Залікова контрольна робота виконується для покращення сумарного залікового балу. Залік проводиться у письмовому вигляді. Залікова контрольна робота містить 4 завдання (2 теоретичні і 2 практичні), які забезпечують перевірку результатів навчання. Завдання передбачають інтегроване застосування знань матеріалу навчальної дисципліни.

Теоретичні запитання охоплюють весь матеріал курсу. При написанні відповіді на запитання треба чітко й лаконічно розкрити його суть і, бажано, подати приклад. Перелік теоретичних запитань:

1. Етапи розв'язування задачі за допомогою комп'ютера.
2. Поняття алгоритму.
3. Властивості алгоритмів.
4. Складність алгоритмів.
5. Способи подання алгоритмів.
6. Лінійні обчислювальні процеси.
7. Розгалужені обчислювальні процеси.
8. Циклічні обчислювальні процеси.
9. Різноманітність алгоритмів розв'язування задачі.
10. Залежність алгоритму і реалізації алгоритму від використовуваної мови програмування.
11. Позиційні й непозиційні системи числення.
12. Подання чисел у позиційних системах.
13. Двійкова, вісімкова і шістнадцяткова системи числення.
14. Переходи між поданнями чисел у різних системах.
15. Подання цілих чисел у комп'ютері.
16. Подання дійсних чисел у комп'ютері.

17. Особливості роботи з дійсними числами.
18. Бінарні й небінарні коди Грея, їхнє використання.
19. Поняття лінійного алгоритму.
20. Програмування виразів.
21. Структура лінійного алгоритму.
22. Оцінка складності виразів; оптимізація обчислень.
23. Поняття алгоритму з розгалуженнями.
24. Логічні вирази. Обчислення значень логічних виразів.
25. Розробка алгоритмів і програмування розгалужених обчислювальних процесів: послідовні умови, вкладені умови.
26. Розробка алгоритмів і програмування розгалужених обчислювальних процесів: використання оператора вибору.
27. Цикломатична складність алгоритму.
28. Алгоритм повного розв’язування квадратного рівняння.
29. Оцінка обчислювальної складності алгоритмів з розгалуженнями.
30. Особливості обчислення логічних виразів.
31. Оптимізація алгоритмів з розгалуженнями.
32. Поняття алгоритму з циклами.
33. Реалізація алгоритму з циклами.
34. Особливості використання циклів з передумовою, післяумовою, з параметром.
35. Вкладені цикли.
36. Ітераційні циклічні процеси.
37. Запобігання механічним помилкам під час вводу інформації.
38. Зациклювання роботи програми.
39. Оригінальний алгоритм Евкліда на основі віднімання.
40. Алгоритм Евкліда на основі ділення.
41. Двійковий алгоритм Евкліда (алгоритм Стайна).
42. Обчислення сум з дуже малими і великими числами.
43. Реалізація схеми Горнера.

44. Обчислення «машинного епсилона».

45. Метод дотичних Ньютона обчислення кореня цілого степеня.

46. Ітераційні алгоритми обчислення значення числа π .

47. Виконання обчислень з точністю до «машинного нуля».

48. Складність алгоритмів з циклами; оптимізація виконання, застосування мемоїзації.

49. Поняття модуля і модульного програмування.

50. Властивості модулів.

51. Модульне програмування: його переваги й недоліки.

52. Проектування й програмування великих програм згори вниз і знизу вгору.

53. Оцінка складності модульного алгоритму.

54. Поняття рекурсії.

55. Пряма й непряма рекурсія.

56. Рекурсивний спуск, підйом, глибина рекурсії.

57. Переваги й недоліки використання рекурсії.

58. Оцінка складності рекурсивного алгоритму.

59. Поняття структури даних. Прості й інтегровані структури даних.

60. Поняття масиву. Одновимірний масив.

61. Основні типові алгоритми роботи з одновимірними масивами: знаходження мінімуму / максимуму.

62. Основні типові алгоритми роботи з одновимірними масивами: перестановка елементів; циклічні зсуви.

63. Пошук елемента в масиві.

64. Пошук з бар’єром.

65. Бінарний пошук у відсортованому масиві.

66. Сортування масивів: метод бульбашки і його модифікації.

67. Сортування масивів: шейкер-сортування.

68. Сортування масивів: обмінне сортування простим вибором з пошуком мінімуму.

69. Сортування масивів: метод простих вставок.

70. Сортування масивів: сортування Шелла.

71. Злиття відсортованих масивів.

72. Обчислення факторіалу великого числа.

73. Решето Ератосфена.

74. Поняття двовимірного масиву.

75. Типові алгоритми роботи з двовимірними масивами.

76. Сортування елементів матриці.

77. Робота з секторами матриці, обхід елементів матриці.

78. Розріджені матриці.

79. Генерація бінарних і небінарних кодів Грея.

80. Використання випадкових чисел.

81. Алгоритми генерації рівномірно розподілених випадкових чисел: метод середин квадратів і метод серединних добутків.

82. Алгоритми генерації рівномірно розподілених випадкових чисел: метод перемішування.

83. Алгоритми генерації рівномірно розподілених випадкових чисел: вибір параметрів лінійного конгруентного методу генерації випадкових чисел.

84. Генерація всіх підмножин множини.

85. Генерація перестановок без повторень.

86. Генерація перестановок з повтореннями.

87. Генерація сполук без повторень.

88. Генерація сполук з повтореннями.

89. Генерація розміщень без повторень.

90. Генерація розміщень з повтореннями.

Практичні завдання. При виконанні практичних завдань треба написати програми з коментарями; глобальні змінні не використовувати.

Перше практичне завдання пов’язане з обробкою одновимірних масивів, друге — з обробкою двовимірних масивів.

Зразки першого практичного завдання. Одновимірний масив містить *n* елементів, де *n* може бути як парним, так і непарним числом. У масиві:

— елементи з певною ознакою зібрати на початку (в кінці масиву), зберігши їхній порядок, решту елементів розмістити в кінці (на початку масиву), замінивши нулями;

— переставити місцями певні частини масиву, зберігши в них порядок елементів;

— переставити місцями кожні два сусідні елементи (1 і 2, 3 і 4 і т. д.);

— елементи з парними (непарними) номерами всього (першої половини; другої половини) масиву розмістити у зворотному порядку тощо.

Усі дії виконувати в тому самому масиві.

Зразки другого практичного завдання: квадратна матриця має порядок *n*, де *n* може бути як парним, так і непарним числом; у вказаному секторі матриці (див. Додаток А) з фрагментом головної чи побічної діагоналі порахувати кількість (суму; добуток) елементів з певною ознакою.