МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Мелешко Є.В., Якименко М.С., Поліщук Л.І.

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

Кропивницький
Видавець Лисенко В.Ф.
2019

УДК 004.421(075.8)
М 47

Мелешко Є.В., Якименко М.С., Поліщук Л.І.

М 47 Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. – Кропивницький: Видавець – Лисенко В.Ф., 2019. – 156 с.

У даному навчальному посібнику розглядаються основні алгоритми сортування, пошуку даних, роботи з графами, деревами та хеш-таблицями, а також алгоритми роботи клітинних автоматів. Для кожного алгоритму наведено опис, псевдокод та графічні пояснення. Посібник розрахований на студентів, що навчаються на технічних спеціальностях, пов'язаних зі створенням програмного забезпечення.

УДК 004.421(075.8)

Рецензенти:

Петренюк Володимир Ілліч, кандидат фізико-математичних наук, доцент, доцент кафедри кібербезпеки та програмного забезпечення Центральноукраїнського національного технічного університету

Дрєєв Олександр Миколайович, кандидат технічних наук, доцент кафедри кібербезпеки та програмного забезпечення Центральноукраїнського національного технічного університету

Рекомендовано до друку Вченою радою
Центральноукраїнського національного технічного університету
(протокол № 9 від 15 квітня 2019 р.)

© Центральноукраїнський національний технічний університет, 2019

© Мелешко Є.В., Якименко М.С., Поліщук Л.І., 2019

© Видавець Лисенко В.Ф., 2019

Зміст

Вступ 4
Розділ 1. Поняття алгоритму 5
Розділ 2. Рекурсія та рекурсивні алгоритми 16
Розділ 3. Прості алгоритми внутрішнього сортування...... 28
Розділ 4. Покращені алгоритми внутрішнього сортування 40
Розділ 5. Швидке сортування... 47
Розділ 6. Спеціалізовані алгоритми внутрішнього сортування 52
Розділ 7. Алгоритми зовнішнього сортування.................. 55
Розділ 8. Алгоритми пошуку. Послідовний пошук елементу. Бінарний пошук елементу. Послідовний пошук рядка 70
Розділ 9. Алгоритми пошуку рядка у тексті 75
Розділ 10. Алгоритми роботи з хеш-таблицями............... 93
Розділ 11. Двійкові дерева. Алгоритми обходів дерев..... 101
Розділ 12. Алгоритми на графах. Основні терміни теорії графів. Представлення графів в комп'ютерних програмах.. 117
Розділ 13. Алгоритми на графах. Алгоритми обходу графа.. 125
Розділ 14. Алгоритми на графах. Пошук найкоротших шляхів. Каркасні дерева.. 131
Розділ 15. Клітинні автомати. Поняття, види, застосування, приклади реалізації 136
Список літератури.. 153

Вступ

Створення та реалізація алгоритмів і структур даних є невід'ємною складовою роботи програміста. Знання інуючих алгоритмів дозволяє швидше вирішувати типові задачі.

Запропонований посібник є вичерпним введенням до сучасних комп'ютерних алгоритмів. У ньому надано велику кількість конкретних алгоритмів, які описуються досить глибоко, але таким чином, щоб розробка та аналіз були доступні читачам всіх рівнів підготовки. Ми намагалися обмежитися елементарними поясненнями, але при цьому не завдати шкоди ні глибині викладу, ні математичній строгості.

В кожному розділі наведено певний алгоритм, описані методи його розробки, область застосування та інші пов’язані з ним питання. Алгоритми описуються і простою людською мовою, і за допомогою псевдокоду, розробленого таким чином, щоб його зрозумів кожен, хто має мінімальний досвід програмування. Оскільки один з найважливіших критеріїв розробки алгоритму – це його *ефективність*, кожний опис алгоритму включає в себе ретельний аналіз часу його роботи.

Даний посібник призначений в першу чергу для студентів, які вивчають той чи інший курс по алгоритмам і структурам даних. Він також буде корисний для технічних фахівців, що бажають підвищити свій рівень в цій області, оскільки опис процесу розробки алгоритмів супроводжується викладом технічних і математичних питань.

Сподіваємося, що цей посібник стане хорошим введенням в теорію алгоритмів. Ми намагались викласти кожен алгоритм в доступній і захоплюючій формі. Псевдокод записано таким чином, щоб кожен алгоритм був зрозумілим і лаконічним, використовуючи індивідуальні особливості тієї чи іншої мови програмування.

Розділ 1. Поняття алгоритму

Алгоритм (латинізов. *Algorithmi*, від імені перського математика IX ст. аль-Хорезмі) – послідовність, система, набір систематизованих правил виконання обчислювального процесу, що обов'язково призводить до розв'язання певного класу завдань після скінченного числа операцій. При написанні комп'ютерних програм алгоритм описує логічну послідовність операцій. Для візуального зображення алгоритмів часто використовують блок-схеми.

Єдиного «вірного» визначення поняття «алгоритм» немає.

«Алгоритм – це кінцевий набір правил, що визначає послідовність операцій для рішення конкретної множини задач і володіє п'ятьма важливими рисами: скінченність, визначеність, введення, виведення, ефективність» (Д.Е. Кнут).

«Алгоритм – це будь-яка система обчислень, що виконуються за строго визначеними правилами, яка після якого-небудь числа кроків свідомо призводить до вирішення поставленого завдання» (А. Колмогоров).

«Алгоритм – це точне розпорядження, що визначає обчислювальний процес, що йде від варійованих початкових даних до шуканого результату» (А. Марков).

«Алгоритм – точне приписання про виконання в певному порядку деякої системи операцій, що ведуть до рішення всіх завдань даного типу» (Філософський словник / Під ред. М.М. Розенталя).

«Алгоритм – строго детермінована послідовність дій, що описує процес перетворення об'єкта з початкового стану в кінцевий, записана за допомогою зрозумілих виконавцеві команд» (М.Д. Угринович).

Походження терміну алгоритм

Слово *алгоритм* походить від імені перського вченого, астронома та математика Аль-Хорезмі. Приблизно 825 до н.е. він написав трактат, в якому описав вигадану в Індії позиційну десяткову систему числення.

В першій половині XII століття книжка потрапила до Європи в перекладі латинською мовою під назвою *Algoritmi de numero Indorum*. Вважається, що перше слово в перекладі відповідає невдалій латинізації імені Аль-Хорезмі, а назва перекладу звучить як «Алгоритми про індійську лічбу».

Через невірне тлумачення слова *Algoritmi* як іменника в множині, ним стали називати *метод обчислення*.

Неформальне визначення

Кожен алгоритм передбачає існування *початкових* (вхідних) даних та в результаті роботи призводить до отримання певного *результату.* Робота кожного алгоритму відбувається шляхом виконання послідовності деяких елементарних дій. Ці дії називають *кроками,* а процес їхнього виконання називають *алгоритмічним процесом.* В такий спосіб визначають властивість *дискретності алгоритму*.

Важливою властивістю алгоритмів є *масовість,* або можливість застосування до різних вхідних даних. Тобто, кожен алгоритм покликаний розв'язувати клас однотипних задач.

Необхідною умовою, яка задовольняє алгоритм, є *детермінованість,* або *визначеність.* Це означає, що виконання команд алгоритму відбувається у єдиний спосіб та призводить до одного й того ж результату при одних і тих же вхідних даних.

Вхідні дані алгоритму можуть бути обмежені набором *припустимих вхідних даних.* Застосування алгоритму до неприпустимих вхідних даних може призводити до того, що

алгоритм ніколи не зупиниться, або потрапить в глухий кут (зависання) з якого не зможе продовжити виконання.

Формальне визначення

Різноманітні теоретичні проблеми математики та прискорення розвитку фізики та техніки поставили на порядок денний точніше визначення поняття алгоритму.

Перші спроби уточнення поняття алгоритму та його дослідження здійснювали в першій половині XX століття Алан Тюрінг, Еміль Пост, Жак Ербран, Курт Гедель, Андрій Марков, Алонзо Черч. Було розроблено декілька означень поняття алгоритму, але згодом було з'ясовано, що всі вони визначають одне й те саме поняття.

Машина Тюрінга

Машина Тюрінга – це абстрактна машина (автомат), що працює зі стрічкою окремих комірок, в які записано символи.

Машина також має голівку для запису та читання символів із комірок, яка може рухатись вздовж стрічки (рис. 1.1). На кожному кроці машина зчитує символ з комірки, на яку вказує голівка, та, на основі зчитаного символу й внутрішнього стану робить наступний крок. При цьому, машина може змінити свій стан, записати інший символ в комірку, або пересунути голівку на одну комірку ліворуч або праворуч.

Головна ідея, що лежить в основі машини Тюрінга, дуже проста.

На основі дослідження цих машин було висунуто *тезу Тюрінга* (основна гіпотеза алгоритмів):

Для знаходження значень функції, заданої в деякому алфавіті, тоді і лише тоді існує деякий алгоритм, коли функція обчислюється за Тюрінгом, тобто, коли її можна обчислити на придатній машині Тюрінга.

Ця теза є аксіомою, постулатом, і не може бути доведена математичними методами, оскільки алгоритм не є точним математичним поняттям.

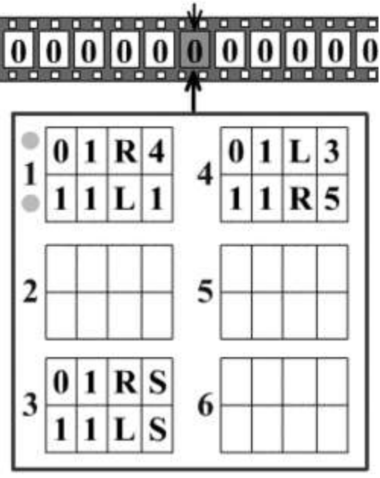


Рисунок 1.1 – Схематична ілюстрація роботи машини Тюрінга

Алгоритмічно нерозв'язні задачі

Функцію *f* називають *обчислюваною* (англ. *computable*), якщо існує машина Тюрінга, яка обчислює значення *f* для всіх елементів множини визначення функції. Якщо такої машини не існує, функцію *f* називають *необчислюваною*. Функція вважатиметься необчислюваною навіть, якщо існують машини Тюрінга, здатні обчислити значення для підмножини з усієї множини вхідних даних.

Випадок, коли результатом обчислення функції *f* є булеве значення *істина* або *хиба* (або множина {0, 1}) називають *задачею*, яка може бути *розв'язною*, або *нерозв'язною* в залежності від обчислюваності функції *f*.

Важливо точно вказувати припустиму множину вхідних даних, оскільки задача може бути розв'язною для однієї множини та нерозв'язною для іншої.

Однією з перших задач, для якої було доведено нерозв'язність є *проблема зупинки*. Формулюється вона наступним чином:

Маючи опис програми для машини Тюрінга, визначити, чи завершить роботу програма за скінченний час, чи працюватиме нескінченно, отримавши будь-які вхідні дані.

Способи запису алгоритмів:

1. графічний (блок-схеми, таблиця 1.1);
2. мовний:

– звичайна мова (наприклад, кулінарний рецепт);

– псевдокод.

Таблиця 1.1 – Основні елементи схем алгоритму

Найменування	**Позначення**	**Функція**
Термінатор	R, 0.5a, b	Елемент відображає вхід у програму або вихід з неї (найбільш часте застосування – початок і кінець програми). Всередині фігури записується відповідна дія.
Процес	a, b	Виконання однієї або декількох операцій, обробка даних будь-якого виду (зміна значення даних, форми подання, розташування). Всередині фігури записують

Найменування	Позначення	Функція
		безпосередньо самі операції.
Рішення	a b	Показує рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення виразів, зазначених всередині цього елементу. Вхід в елемент позначається лінією, що входити зазвичай у верхню вершину елементу. Якщо виходів два чи три, то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижній). Якщо виходів більше трьох, то їх слід показувати однією лінією, що виходить з вершини (частіше нижній) елементу, яка потім розгалужується. Відповідні результати

Найменування	Позначення	Функція
		обчислень можуть записуватися поруч з лініями, що відображають ці шляхи.
Зумовлений процес	0.15a a b	Символ відображає виконання процесу, що складається з однієї або декількох операцій, що визначений в іншому місці програми (у підпрограмі, модулі). Всередині символу записується назва процесу і передані в нього дані.
Дані	a 0.25a b	Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Цей символ не визначає носія даних (для вказівки типу носія

Найменування	Позначення	Функція
		даних використовуються специфічні символи).
Підготовка	a, a, b	Символ відображає модифікацію команди або групи команд з метою впливу на деяку наступну функцію(встановлення перемикача, модифікація індексного регістру або ініціалізація програми)
З'єднувач	0.5a	Символ відображає вихід у частину схеми і вхід з іншої частини цієї схеми. Використовується для обриву лінії та продовження її в іншому місці (наприклад: поділ блок-схеми, що не поміщається на аркуші). Відповідні сполучні символи повинні мати одне (при тому унікальне) позначення.

Найменування	Позначення	Функція
Коментар	5min a	Використовується для більш детальної інформації про кроки, процесу або групи процесів. Опис поміщається з боку квадратної дужки і охоплює їй по всій висоті. Пунктирна лінія йде до елементу, що описується, або до групи елементів (при цьому група виділяється замкнутої пунктирною лінією). Також символ коментаря слід використовувати в тих випадках, коли обсяг тексту в будь-якому іншому символі (наприклад, символ процесу, символ даних та ін.) перевищує його обсяг.

Примітка: значення a вибирається із ряду 10, 15, 20 … кратне 5 мм, відповідно b = 1.5a (допускається b = 2a), R = 0.25a.

Псевдокод – компактна (найчастіше неформальна) мова опису алгоритмів, що використовує ключові слова імперативних мов програмування, але опускає несуттєві подробиці й специфічний синтаксис.

Псевдокод зазвичай опускає деталі, несуттєві для розуміння алгоритму людиною. Такими несуттєвими деталями можуть бути опис змінних, системно-залежний код і підпрограми. Головна мета використання псевдокоду – забезпечити розуміння алгоритму людиною, зробити опис більш сприйманим, ніж початковий код мовою програмування. Псевдокод широко використовується в підручниках і науково-технічних публікаціях, а також на початкових стадіях розробки комп'ютерних програм.

! **Псевдокод** може бути записаний з використанням англійських, або українських слів, або команд структурних мов програмування. У даному посібнику ми будемо використовувати різні варіанти написання псевдокоду.

Таблиця 1.2 – Базові керуючі структури

Назва структури	**Псевдокод**
присвоювання, введення, виведення	змінна = 0, введення (змінна), виведення (змінна)
розгалуження	якщо умова то (серія1 інакше серія 2)
цикл ПОКИ	поки умова поч серія кін

Загальний вигляд псевдокоду функції:

```
функція тип_результату назва_функції(аргументи)
  дано умови застосування алгоритму //не обов'язково
  треба ціль виконання алгоритму    // не обов'язково
початок
|   послідовність команд (тіло функції)
кінець
```

Розглянемо декілька прикладів псевдокоду алгоритмів.

Приклад 1.1. Псевдокод програми «Hello, world!»

```
функція HELLOWORLD()
початок
      виведення ('Hello, World!')
кінець
```

Приклад 1.2. Псевдокод програми «Сума квадратів»

```
функція цілочисельна СумаКвадратів(цілочисельна n)
    дано | n > 0
    треба | S = 1*1 + 2*2 + 3*3 + ... + n*n
початок
змінна цілочисельна i
    введення n; S = 0
    початок для i від 1 до n
      S = S + i * i
    кінець
    повернення S
кінець
ВиведенняНаЕкран СумаКвадратів(7)
```

Питання до розділу:

1. Що таке алгоритм на Ваш особистий погляд?
2. Які основні властивості алгоритмів?
3. Що собою являє машина Тюрінга?
4. Сформулюйте основну гіпотезу алгоритмів.
5. Які існують способи запису алгоритмів?
6. Яку кількість елементів "Термінатор" може мати один алгоритм?
7. Вкажіть, який елемент використовується для перевірки умови $x < 0$?
8. Вкажіть, який елемент використовується для виклику підпрограми?
9. Що таке псевдокод?
10. Яка головна мета використання псевдокоду?

Розділ 2. Рекурсія та рекурсивні алгоритми

Рекурсія – процес повторення чого-небудь самоподібним способом. Наприклад, вкладені віддзеркалення, вироблені двома точно паралельними один одному дзеркалами, є однією з форм нескінченної рекурсії. Даний термін має більш спеціальні значення в різних областях знань – від лінгвістики до логіки.

Рекурсія – це одна з фундаментальних концепцій у математиці й програмуванні. Рекурсія – це одна з форм мислення, цей поужний засіб, ще дозволяє будувати елегантні й виразні алгоритми.

Найбільш загальне застосування рекурсія знаходить у математиці й інформатиці. Тут вона є методом визначення функцій, при якому обумовлена функція застосована в тілі свого ж власного визначення. При цьому нескінченний набір випадків (значень функції) описується за допомогою кінцевого виразу, що для деяких випадків може посилатися на інші випадки, якщо при цьому не виникає циклів або нескінченного ланцюжка посилань. Фактично це спосіб визначення множини об'єктів через самого себе з використанням раніше заданих окремих правил.

Визначення, які використовують рекурсію, називаються *індуктивними*. Одним із прикладів подібного визначення є аксіоматична побудова множини натуральних чисел.

– Факторіал цілого додатного числа n (позначається $n!$) визначається як $n! = n \cdot (n - 1)!$ при ($n > 0$) та $n! = 1$ при ($n = 0$).

– Числа Фібоначі визначаються за допомогою рекурентного співвідношення:

Перше й друге числа Фібоначі рівні 1.

Для $n > 2$, $n - e$ число Фібоначі дорівнює сумі $(n - 1)$-го й $(n - 2)$-го чисел Фібоначі.

– Практично всі геометричні фрактали задаються у формі нескінченної рекурсії (наприклад, трикутник Серпінського).

– Рекурсивні акроніми: GNU (GNU Not Unix), PHP (PHP: Hypertext Preprocessor) і т.д.

Об'єкт називається *рекурсивним*, якщо він містить сам себе або визначений за допомогою самого себе.

Якщо процедура p містить явне звертання до самої себе, то вона називається *явно рекурсивною*. Якщо процедура p містить звертання до деякої процедури q, яка у свою чергу містить пряме або непряме звертання до p, то p – називається *побічно рекурсивною*.

Але рекурсивна програма не може викликати себе нескінченно, інакше вона ніколи не зупиниться, для цього у програмі (функції) повинен бути присутнім ще один важливий елемент – так звана термінальна умова, тобто умова за якої програма припиняє рекурсивний процес.

Рекурсія в програмуванні. Функції

У програмуванні рекурсія – виклик функції (процедури) з неї ж самої, безпосередньо (проста рекурсія) або через інші функції (складна рекурсія), наприклад, функція *A* викликає функцію *B*, а функція *B* – функцію *A*. Кількість вкладених викликів функції або процедури називається *глибиною рекурсії*.

Перевага рекурсивного визначення об'єкта полягає в тому, що таке скінченне визначення теоретично здатне описувати нескінченно велику кількість об'єктів. За допомогою рекурсивної програми ж можливо описати нескінченне обчислення, причому без явних повторень частин програми.

На практиці реалізація рекурсивних викликів функцій у мовах і середовищах програмування, як правило, опирається на механізм стеку викликів – адреси повернення й локальних змінних функцій записуються в стек, завдяки чому кожний

наступний рекурсивний виклик цієї функції користується своїм набором локальних змінних і за рахунок цього працює коректно. Зворотним боком цього досить простого за структурою механізму є те, що на кожний рекурсивний виклик потрібна деяка кількість оперативної пам'яті комп'ютера, і при надмірно великій глибині рекурсії може наступити переповнення стека викликів. Внаслідок цього, зазвичай рекомендується уникати рекурсивних програм, які приводять (або при деяких умовах можуть приводити) до занадто великої глибини рекурсії.

Втім, є спеціальний тип рекурсії, що називається *«хвостовою рекурсією»*. Інтерпретатори й компілятори функціональних мов програмування, що підтримують оптимізацію коду (вихідного та/або що виконується), автоматично перетворять хвостову рекурсію до ітерації, завдяки чому забезпечується виконання алгоритмів із хвостовою рекурсією в обмеженому обсязі пам'яті. Такі рекурсивні обчислення, навіть якщо вони формально нескінченні (наприклад, коли за допомогою рекурсії організовується робота командного інтерпретатора, що приймає команди користувача), ніколи не приводять до вичерпання пам'яті. Однак, далеко не завжди стандарти мов програмування чітко визначають, яким саме умовам повинна задовольняти рекурсивна функція, щоб транслятор гарантовано перетворив її в ітерацію. Один з рідкісних винятків – мова Scheme (діалект мови Lisp), опис якої містить всі необхідні відомості.

Будь-яку рекурсивну функцію можна замінити циклом і стеком.

Рекурентності

Рекурентність – це рекурсивне визначення функції. Вона широко поширена в математиці. Можливо, найбільш знайома Вам з такого роду функцій, як факторіал. Факторіал – це

добуток натуральних чисел від одиниці до деякого даного натурального числа. Він визначається формулою:

$$N! = N((N-1)!, \quad \text{для } N >= 1 \text{ і } 0! = 1.$$

Це прямо відповідає нижченаведеному псевдокоду рекурсивної програми:

```
function int factorial(int N)
 begin
    if N = 0 then
        factorial = 1
    else
        factorial = N * factorial(N - 1)
 end
```

Ця програма демонструє основні властивості рекурсивних програм: програма викликає сама себе (з меншим значенням аргументу), і в неї є термінальна умова, за якою вона прямо обчислює результат.

Необхідно також пам'ятати про те, що це – програма, а не формула: наприклад ні формула, ні програма не працюють із від’ємними *N*, але згубні наслідки спроби зробити обчислення для від’ємного числа більш помітні для програми, ніж для формули. Виклик factorial(-1) призведе до нескінченного рекурсивного циклу. Тому перед викликом даної програми потрібно робити перевірку умови незаперечності.

Друге добре відоме рекурентне співвідношення – співвідношення для визначення чисел Фібоначі. Числа Фібоначі – це елементи числової послідовності 1, 1, 2, 3, 5, 8, ..., у яких кожний наступний елемент дорівнює сумі попередніх.

$$F_N = F_{N-1} + F_{N-2}, \text{ де } N >= 2 \text{ і } F_0 = F_1 = 1.$$

І знову, рекурентність відповідає простій рекурсивній програмі, яку можна реалізувати на основі наступного псевдокоду:

```
function int fibonacci(int N)
 begin
    if N <= 1 then
        fibonacci = 1
    else
   fibonacci = fibonacci(N - 1) + fibonacci(N - 2)
 end
```

Як ми побачимо, багато цікавих алгоритмів можна легко реалізувати за допомогою рекурсивних програм, і багато розроблювачів алгоритмів бажають реалізовувати алгоритми рекурсивно. Але часто трапляється також і так, що настільки ж цікавий алгоритм ховається в деталях нерекурсивної реалізації. Давайте розглянемо такий псевдокод алгоритму:

```
const max = 25
var int i; array of int F[0..max]

procedure fibonacci()
 begin
    F[0] = 1
    F[1] = 1
    for i = 2 to max do
        F[i] = F[i - 1] + F[i - 2]
 end
```

Ця програма обчислює перші max чисел Фібоначі, використовуючи масив розміром max. Цей метод називається *ітераційним*.

Який же метод краще? Точних правил для вибору між рекурсивною й нерекурсивною версіями алгоритму рішення завдання не існує. Стислість і виразність більшості рекурсивних процедур спрощує їхнє читання й супровід. З іншого боку, виконання рекурсивних процедур вимагає великих витрат і

пам'яті, і часу процесора в порівнянні з їхніми ітераційними аналогами.

Поділ навпіл

Велика частина алгоритмів використовує два рекурсивних виклики, кожний з яких працює приблизно з половиною вхідних даних. У дизайні алгоритмів таке явище називають "*поділ навпіл*"; його часто використовують для досягнення істотної економії.

Як приклад, давайте розглянемо завдання нанесення поділок на лінійку: на ній повинна бути мітка в точці 1/2", мітка трохи коротша через кожні 1/4", ще більш коротка через 1/8" і так далі (рис. 2.1).

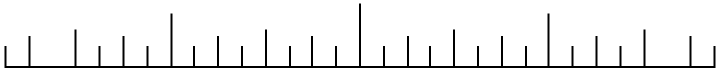


Рисунок 2.1 – Лінійка

Ми також припускаємо, що в нас є процедура mark(*x*, *h*) для нанесення мітки висотою *h* одиниць на лінійку в позицію *x*. Центральна мітка повинна мати висоту n одиниць, мітки в центрах лівої й правої половинок – висоту (*n* – 1) одиниць, і так далі. Наступний псевдокод рекурсивної програми ілюструє прямий шлях досягнення нашої мети:

```
procedure rule(int l, int r, int h)
 var int m
 begin
    if h > 0 then
    begin
        m = (l + r) div 2
        mark( m, h )
        rule( l, m, h - 1)
        rule( m, r, h - 1)
    end
 end
```

Ідея цього методу полягає в наступному: для того, щоб промаркувати лінійку, ми спершу наносимо довгу мітку в її середині. Це ділить її на дві рівні частини. Тепер ми наносимо (більш короткі) мітки в середині кожної із цих половинок, використовуючи ту ж саму процедуру.

Необхідно звертати особливу увагу на термінальну умову рекурсивної програми – у протилежному випадку вона ніколи не зупиниться! У вищенаведеній програмі перевіряємо, коли ми повинні нанести мітку висотою 0. Розглянемо приклад, як результат виклику rule(0, 8, 3). Ми ставимо мітку по середині й викликаємо rule для лівої половини, потім робимо теж саме для лівої половини, і так далі поки висота мітки не стане дорівнювати 0. В остаточному підсумку ми вертаємося до rule і розмічаємо праву половину подібним чином.

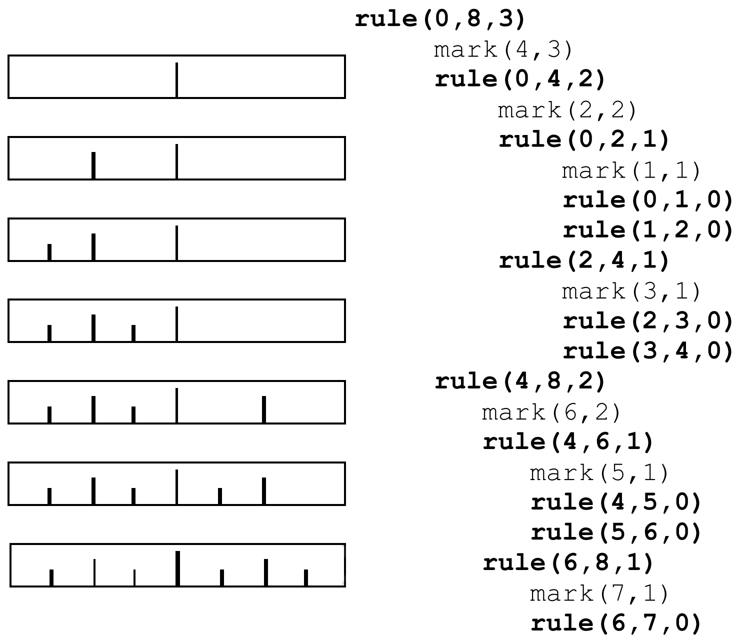


Рисунок 2.2 – Ілюстрація послідовності рекурсивних викликів

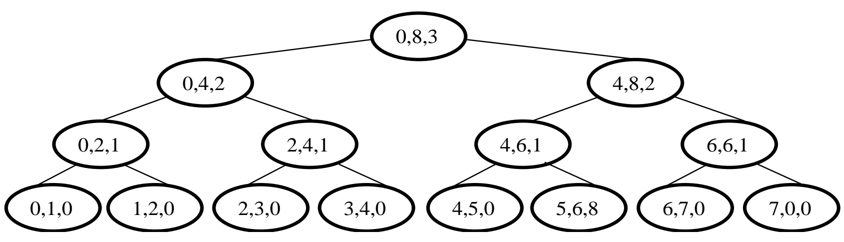


Рисунок 2.3 – Дерево рекурсивних викликів для малювання лінійки

Інший нерекурсивний алгоритм полягає в тому, щоб малювати спершу найкоротші мітки, потім трохи менш короткі й так далі, як показано в наступному, досить компактному, нерекурсивному псевдокоді:

```
procedure draw(int l, int r, int h)
 var  int i, int j
 begin
     j = 1
     for i = 1 to h do
      begin
          for x = 0 to (l + r) div j do
          mark(l + j + x * (j + j), i)
          j = j + j
     end
 end
```

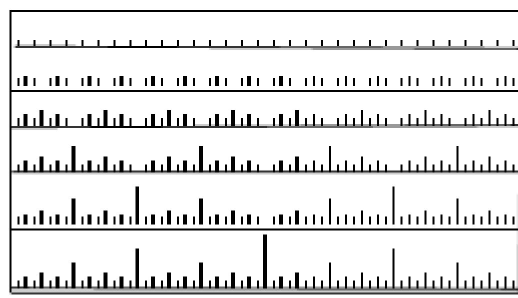


Рисунок 2.4 – Нерекурсивне малювання лінійки

Зараз створимо двомірний візерунок, що демонструє як проста рекурсія може дати рішення тому, що здається складним.

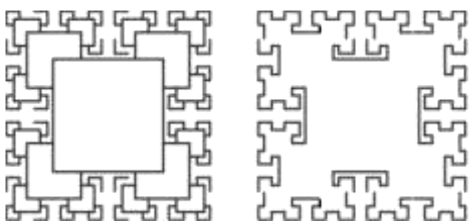


Рисунок 2.5 – Фрактальна зірка (ліворуч) і її обриси (праворуч)

Приклад псевдокоду алгоритму створення фрактальної зірки:

```
var int r

 procedure star(int x, int y, int r)
  begin
     if r > 2 then begin
         star(x + r, y + r, r div 2)
         star(x + r, y - r, r div 2)
         star(x - r, y - r, r div 2)
         star(x - r, y + r, r div 2)
         drawFilledRrectangle(x - r, y - r, x + r, y + r)
     end
  end

 begin

 ...  // тут повинне бути очищення екрану

     write('Enter the length of star: ')
     read(r)

...  // тут повинні бути функції налаштування графіки
```

```
  do
     star(getMaxX() div 2, getMaxY() div 2, r)
     repeat
  until (not keyPressedEsc)
 end
```

Примітив для малювання тут – просто процедура, яка малює квадрат розміром $2r$ із центром в (x, y).

Рекурсивно визначені геометричні візерунки подібні цьому називають *фракталами*. Якщо використовується більш складний примітив для малювання й більш складні рекурсивні виклики (особливо на дійсній і комплексній площинах), то можна розробити візерунки вражаючої краси й складності. Відомим прикладом фракталу є трикутник Серпінського (рис. 2.6).

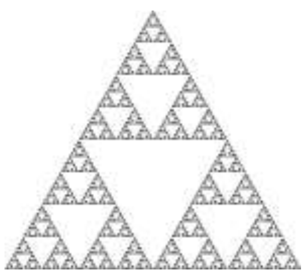


Рисунок 2.6 – Трикутник Серпінського

Рекурсивні структури даних

Опис типу даних може містити посилання на самого себе. Подібні структури використовуються при описі списків і графів. Спосіб створення рекурсивних типів даних дуже залежить від засобів тієї чи іншої мови програмування і, зокрема, реалізації у ній вказівників. Наведемо приклад створення рекурсивної структури даних не на псевдокоді, а на коді реальної мови програмування.

Приклад 2.1. Опис списку на мові C++

```
 class element_of_list
 {
   element_of_list *next; /* посилання на наступний
елемент того ж типу */
   int data; /* деякі дані */
 };
```

Рекурсивна структура даних найчастіше спричиняє застосування рекурсії для обробки цих даних.

Корекурсія

Корекурсія – у теорії категорій та інформатиці тип операції, дуальний до рекурсії. Зазвичай корекурсія використовується (разом з механізмом ледачих обчислень) для генерації нескінченних структур даних.

Загальні зауваження

Правило використання корекурсії на коданих (прикладом яких є потоки) дуальне правилу застосування рекурсії на даних. Замість *згортання* структури даних виходячи з початкового значення аргументу, корекурсія *розгортає* результат на основі початкового значення аргументу. Необхідно відзначити, що корекурсія *створює* потенційно нескінченні структури даних, у той час як звичайна рекурсія *аналізує (розбирає)* по необхідності скінченні структури даних. Звичайна рекурсія незастосовна до коданих, оскільки процес аналізу може ніколи не зупинитися. Відповідно, корекурсія не може продукувати дані, оскільки дані завжди скінченні.

Приклади корекурсії

Приклад використання механізму корекурсії мовою Haskell (обчислення нескінченного списку чисел Фібоначі):

```
fibs = 0:1: zipWith (+) fibs(tail fibs)
```

Інший приклад – обчислення нескінченного списку простих чисел:

```
primes = eratosthenes [2..]
   where
   eratosthenes (x:xs) = x:eratosthenes (filter ((/=
0).('mod' x)) xs)
```

Дана функція реалізує алгоритм «решето Ератосфена», причому робить це самим безпосереднім чином.

Наведені приклади мовою Haskell не зовсім коректні, оскільки в мові немає ідіоми коданих. У зазначених прикладах кодані тільки емулюються за допомогою нескінченного списку.

Питання до розділу:

1. В чому різниця між явно і побічно рекурсивними процедурами?
2. Який важливий елемент повинен бути присутнім у програмі (функції) для її коректної роботи?
3. Які широко поширені рекурентності Ви знаєте з математики?
4. Чому перед викликом програми для обчислення факторіалу потрібно робити перевірку умови незаперечності?
5. Які переваги й недоліки використання рекурсивних процедур?
6. На Ваш погляд, який із наведених вище прикладів для малювання лінійки (а саме рекурсивний розподіл навпіл чи нерекурсивний алгоритм) кращий і чому?
7. Дайте означення фракталу.
8. Дайте означення глибини рекурсії.
9. Чому рекомендують уникати рекурсивних програм?
10. Опишіть порядок роботи “хвостової рекурсії”.
11. Дайте визначення корекурсії.

Розділ 3. Прості алгоритми внутрішнього сортування

Алгоритм сортування – це алгоритм для впорядкування елементів у деякій структурі даних за зростанням чи спаданням. У випадку наявності елементів з однаковими значеннями, в упорядкованій послідовності вони розташовуються поруч один за одним у будь-якому порядку. Однак іноді буває корисно дотримуватися початкового порядку елементів з однаковими значеннями.

В залежності від того, над якою структурою даних здійснюється сортування, буває сортування масиву, зв'язаного списку, дерева, графа, таблиці.

У випадку, коли елемент структури даних має декілька полів, вводиться поняття ключ сортування.

Ключ сортування – це поле структури даних, за значенням якого визначається порядок елементів (тобто за яким відбувається сортування). На практиці як ключ часто виступає число, а в інших полях зберігаються які-небудь дані, що ніяк не впливають на роботу алгоритму.

Простий алгоритм сортування можна розбити на 3 частини:

- порівняння, що визначає впорядкованість пари елементів;
- перестановка, що міняє місцями цю пару елементів;
- повтор перших двох дій доти, поки всі елементи структури даних не будуть впорядковані.

Алгоритми сортування мають велике практичне значення. Їх можна зустріти там, де відбувається обробка й зберігання великих обсягів інформації. Вони використовуються також у криптографії, кодуванні, вирішенні математичних задач тощо.

Жодна інша проблема не породила такої кількості різних рішень, як задача сортування. Універсального, найкращого

алгоритму сортування не існує. Однак, маючи приблизні характеристики вхідних даних, можна підібрати метод, що працює оптимальним чином. Для цього необхідно знати параметри, за якими буде здійснюватися оцінка алгоритмів.

Параметри алгоритмів сортування:

– *Час сортування* – основний параметр, що характеризує швидкодію алгоритму.

– *Пам'ять* – один з параметрів, що характеризується тим, що ряд алгоритмів сортування вимагають виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці використаної пам'яті не буде враховуватися місце, що займає вихідний масив даних і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

– *Стійкість* – це параметр, що відповідає за те, що сортування не змінює взаємного розташування рівних елементів. Наприклад, якщо алфавітний список групи сортується за оцінками, то стійкий метод створює список у якому прізвища студентів з однаковими оцінками будуть впорядковані за алфавітом, а нестійкий метод створить список у якому, можливо, вихідний порядок буде порушений.

– *Природність поводження* – параметр, який вказує на ефективність методу при обробці вже відсортованих, або частково відсортованих даних. Алгоритм поводиться природно, якщо враховує цю характеристику вхідної послідовності й працює краще.

– *Використання операції порівняння.* Алгоритми, що використовують для сортування порівняння елементів між собою, називаються *заснованими на порівняннях*. Мінімальна трудомісткість гіршого випадку для цих алгоритмів становить $O(n \log n)$, але вони відрізняються гнучкістю застосування. Для спеціальних випадків (типів даних) існують більш ефективні алгоритми.

Класифікація алгоритмів сортування

Всі алгоритми сортування можна класифікувати за різними ознаками, наприклад, за стійкістю, за особливостями функціонування, за використанням операцій порівняння, за потребою в додатковій пам'яті, за структурою даних, за шириною області застосування, за потребою в знаннях про структуру даних, що виходять за рамки операцій порівняння ключів тощо.

Розглянемо більш детально класифікацію алгоритмів сортування за використанням пам'яті. У цьому випадку основні типи сортування діляться на:

- *Внутрішнє сортування* – це алгоритм сортування, що у процесі впорядкування даних використовує тільки оперативну пам'ять (ОЗП) комп'ютера. Тобто оперативної пам'яті досить для розміщення в ній масиву даних, що сортується з довільним доступом до будь-якої комірки й власне для виконання алгоритму. Внутрішнє сортування застосовується у всіх випадках, за винятком однопрохідного зчитування даних і однопрохідного запису відсортованих даних. Залежно від конкретного алгоритму і його реалізації дані можуть сортуватися в тій же області пам'яті, або використовувати додаткову оперативну пам'ять.
- *Зовнішнє сортування* – це алгоритм сортування, що при проведенні впорядкування даних використовує зовнішню пам'ять, як правило, жорсткі диски. Зовнішнє сортування розроблене для обробки великих структур даних, які не поміщаються в оперативну пам'ять. Звертання до різних носіїв накладає деякі додаткові обмеження на даний алгоритм: доступ до носія здійснюється послідовним чином, тобто в кожний момент часу можна зчитати або записати тільки елемент наступний за поточним; обсяг даних не дозволяє їм розміститися в ОЗП.

Внутрішнє сортування є базовим для будь-якого

алгоритму зовнішнього сортування – окремі частини масиву даних сортуються в оперативній пам'яті й за допомогою спеціального алгоритму з’єднуються в один масив, упорядкований за ключем.

Слід зазначити, що внутрішнє сортування значно ефективніше зовнішнього, тому що на звертання до оперативної пам'яті затрачується набагато менше часу, ніж до носіїв.

Одна з найбільш розповсюджених класифікацій методів сортування наведена на рисунку 3.1.

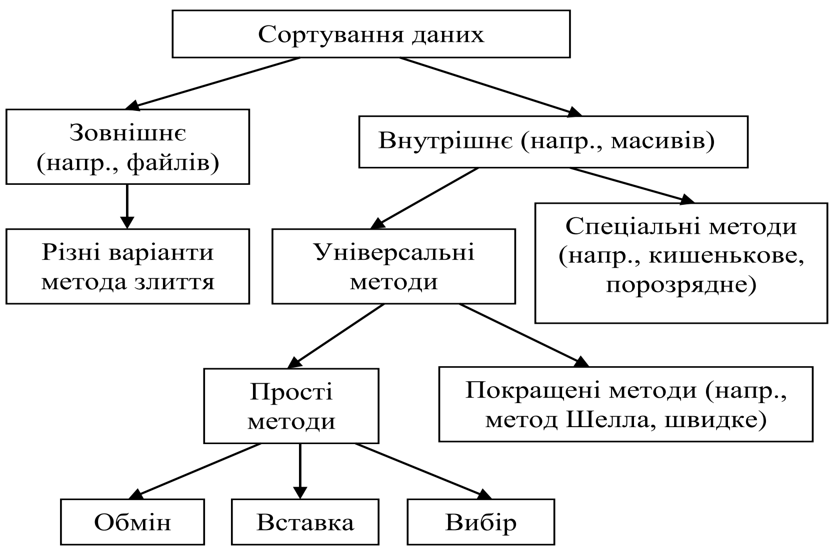


Рисунок 3.1 – Загальна класифікація методів сортування

Класифікація за часом виконання:

За час $O(n^2)$:

- сортування вибором;
- сортування вставкою;
- сортування обміном.

За час $O(n\ log\ n)$:

- пірамідальне сортування;
- швидке сортування;
- сортування злиттям.

За час *O(n)* з використанням додаткової інформації про елементи :

- сортування підрахунком;
- сортування за розрядами;
- сортування комірками.

За час $O(n\ log^2 n)$:

- сортування злиттям модифіковане;
- сортування Шелла.

Алгоритми стійкого впорядкування:

- сортування вибором;
- сортування вставкою;
- сортування обміном;
- сортування злиттям;
- сортування підрахунком;
- сортування за розрядами;
- сортування комірками;
- сортування злиттям модифіковане.

Прості алгоритми внутрішнього сортування

Щоб сконцентруватися на алгоритмічних питаннях, ми будемо працювати з алгоритмами, які сортують масиви цілих чисел. Ці алгоритми легко адаптувати для сортування записів.

В основному програми сортування працюють із записами двома способами: або вони порівнюють і сортують тільки ключі, або пересувають записи повністю. Більшість алгоритмів, які ми вивчимо можна застосовувати, використовуючи їхнє переформулювання в термінах цих двох операцій, для довільних записів.

Якщо записи досить великі, то звичайно намагаються

уникнути їхнього пересування за допомогою "непрямого сортування": при цьому самі записи не сортуються, а замість цього сортується масив вказівників (індексів), так, що перший вказівник вказує на самий маленький елемент і так далі. Ключі можуть зберігатися або із записами (якщо вони великі), або з вказівниками (якщо вони маленькі).

Сортування вибором

Один з найпростіших методів сортування працює в такий спосіб:

1) знаходимо найменший елемент у масиві;

2) міняємо його місцями з елементом, що знаходиться на першому місці;

3) повторюємо процес із другої позиції у файлі й знайдений найменший елемент обмінюємо із другим елементом і так далі поки весь масив не буде відсортований.

У міру просування зліва направо через масив, елементи ліворуч від вказівника перебувають уже в своїй кінцевій позиції (і їх вже не будуть більше пересувати), тому масив стає повністю відсортованим до того моменту, коли вказівник досягає правого краю.

Цей метод називається *сортуванням вибором* оскільки він працює циклічно вибираючи найменший з елементів, що залишилися.

Псевдокод процедури сортування вибором:

```
Процедура СортуванняВибором(масив a, цілочисельна N)
 Початок
     змінні цілочисельні i, j, min, t
     для i від 1 до N-1 //N-розмір масиву
     початок
          min = i
          //цикл знаходження мінімального елементу
          для j від i + 1 до N
               якщо a[j] < a[min] тоді
                   min = j
```

```
          t = a[min] //заміна елементів
          a[min] =a[i]
          a[i] = t
    кінець
 Кінець
```

Приклад сортування масиву з п'яти елементів за даним алгоритмом:

7 5 2 3 1
1 5 2 3 **7**
1 **2** **5** 3 7
1 2 **3** **5** 7

Цей метод – один з найпростіших, і він працює дуже добре для невеликих структур даних. Його "внутрішній цикл" складається з порівняння a[i]<a[min] (плюс код необхідний для збільшення j та перевірки того, щоб він не перевищив N), що навряд чи можна ще спростити.

Більш того, незважаючи на те, що цей метод очевидно є методом "грубої сили", він має дуже важливе застосування: оскільки кожний елемент пересувається не більше ніж раз, то він дуже гарний для великих записів з маленькими ключами.

Характеристика алгоритму:

- Структура даних: Масив.
- Швидкодія: $O(n^2)$.
- Простір: $O(n)$, $O(1)$.
- Оптимальність: Не практичний.

Сортування вставкою

Сортування вставкою – це метод який майже настільки ж простий, що й сортування вибором, але набагато більш гнучкий. Більшість людей при сортуванні колоди гральних карт, використовують метод, схожий на алгоритм сортування вставкою. Суть алгоритму: беремо один елемент і вставляємо

його в потрібне місце серед тих, що ми вже обробили (тим самим залишаючи їх відсортованими). Алгоритм:

1) зліва направо проходимо масив, порівнюючи сусідні елементи, поки не знайдемо елемент, що розташований не в порядку сортування;

2) обмінюємо цей елемент з елементами розташованими ліворуч від нього, поки він не займе потрібну позицію;

3) повторюємо перші дві дії, поки масив не буде відсортовано.

Приклад сортування вставкою масиву з семи елементів за даним алгоритмом зображено на рис. 3.2.

3	8	9	10	6	7	11	j=5, i=4	6<10
3	8	9	6	10	7	11	i=3	6<9
3	8	6	9	10	7	11	i=2	6<8
3	6	8	9	10	7	11	i=1	6>3
3	6	8	9	10	7	11	j=6, i=5	7<10
3	6	8	9	7	10	11	i=4	7<9
3	6	8	7	9	10	11	i=3	7<8
3	6	7	8	9	10	11	i=2	7>6

Рисунок 3.2 – Приклад сортування вставкою

Псевдокод процедури сортування вставкою:

```
Процедура СортуванняВставкою(масив a, цілочисельна N)
 Початок
    Змінні цілочисельні i, j, t
    для i від 2 до N
    початок
        t = a[i]
        j = i-1
        поки j > 0 та t < a[j]
            початок
                 a[j+1] = a[j]
                 j = j-1
```

```
            кінець
        a[j+1] = t
    кінець
 Кінець
```

Також як і в сортуванні вибором, у процесі сортування вставкою елементи ліворуч від вказівника перебувають уже в відсортованому порядку, але вони не обов'язково перебувають у своїй остаточній позиції, оскільки їх ще можуть пересунути праворуч, щоб вставити більш маленькі елементи, які зустрічаються пізніше. Однак масив стає повністю відсортованим, коли вказівник досягає правого краю.

Даний алгоритм простий в реалізації та ефективний для невеликих масивів і є стійким. Він ефективний при сортуванні масивів, дані в яких вже непогано відсортовані: продуктивність рівна $O(n + d)$, де d – кількість інверсій.

Характеристика алгоритму:

– Структура даних: Масив.
– Швидкодія: $O(n^2)$, для найкращого випадку $O(n + d)$.
– Простір: $O(n)$, $O(1)$.
– Оптимальний: Переважно ні.

Бульбашкове сортування (сортування простими обмінами)

Алгоритм працює таким чином – у масиві порівнюються два сусідні елементи. Якщо один з елементів, не відповідає критерію сортування (є більшим, або ж, навпаки, меншим за свого сусіда), то ці два елементи міняються місцями. Прохід по списку продовжується до тих пір, доки дані не будуть відсортованими. Алгоритм отримав свою назву від того, що процес сортування за ним нагадує поведінку бульбашок повітря у резервуарі з водою. Оскільки для роботи з елементами масиву він використовує лише порівняння, це сортування на основі порівнянь.

```
Процедура БульбашковеСортування(масив a,
Цілочисельна N)
 Початок
     Змінні цілочисельні i, j, t
     для i від 1 до N
         для j від 1  до N-1
           якщо a[j]>a[j+1] тоді
                початок
                     t = a[j]
                     a[j] = a[j+1]
                     a[j+1] = t
                кінець
 Кінець
```

Приклад реалізації крок за кроком

Візьмемо масив чисел "5 1 4 2 8", і за допомогою даного алгоритму, відсортуємо його від найменшого до найбільшого елементу. На кожному кроці, елементи, виділені **жирним** шрифтом будуть порівнюватись.

Перший прохід:

(**5** **1** 4 2 8) ⟶(**1** **5** 4 2 8)
(1 **5** **4** 2 8) ⟶(1 **4** **5** 2 8)
(1 4 **5** **2** 8) ⟶(1 4 **2** **5** 8)
(1 4 2 **5** **8**) ⟶(1 4 2 **5** **8**)

Другий прохід:

(**1** **4** 2 5 8) ⟶(**1** **4** 2 5 8)
(1 **4** **2** 5 8) ⟶(1 **2** **4** 5 8)
(1 2 **4** **5** 8) ⟶(1 2 **4** **5** 8)
(1 2 4 **5** **8**) ⟶(1 2 4 **5** **8**)

Тепер наш масив повністю відсортований, однак, алгоритм цього ще не знає. Йому потрібен ще один "пустий" прохід, під час якого він не поміняє місцями жодного елементу

(якщо реалізувати відповідний прапор, інакше проходів буде стільки скільки елементів у масиві – за класичною реалізацією).

Третій прохід:

```
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
```

Нарешті, масив відсортовано, і алгоритм може припинити свою роботу.

Одним з найбільш частих застосувань алгоритмів сортування є сортування рядків (напр., списку назв, списку слів, тощо). Зазвичай воно проводиться так: спочатку множина рядків сортується за першим символом кожного рядка, потім кожна підмножина рядків, що мають однаковий перший символ, сортується по другому символу, і так до тих пір, поки всі рядки не будуть впорядковані. При цьому відсутній символ (при порівнянні рядка довжини N з рядком довжини N + 1) вважається меншим будь-якого символу.

Застосування даного методу до рядків, які представляють собою числа у текстовому записі, видає невірні результати: наприклад, «9» виявляється більше, ніж «11», так як перший символ першого рядка має більше значення, ніж перший символ другого. Для виправлення цієї проблеми алгоритм сортування може перетворювати сортовані рядки в числа і сортувати їх як числа. Такий алгоритм називається «числовим сортуванням», а описаний раніше – «строковом сортуванням».

Питання до розділу:

1. Дайте означення алгоритму сортування.
2. Поясніть, що таке ключ сортування.

3. Де застосовуються алгоритми сортування?
4. Охарактеризуйте параметри алгоритмів сортування.
5. Вкажіть ознаки, за яким можна класифікувати алгоритми сортування.
6. Чим відрізняється внутрішнє сортування від зовнішнього, і як вони пов’язані?
7. Що таке непряме сортування і коли воно використовується?
8. Графічно проілюструйте принцип сортування масиву чисел (2 10 5 3 8) в порядку зростання за методом сортування вибором.
9. Графічно проілюструйте принцип сортування масиву чисел (2 10 5 3 8) в порядку зростання за методом сортування вставкою.
10. Використовуючи бульбашкове сортування, графічно проілюструйте принцип сортування масиву чисел (2 10 5 3 8) за зростанням.
11. Порівняйте використанні в пп. 8-10 методи.

Розділ 4. Покращені алгоритми внутрішнього сортування

Покращення алгоритму бульбашкового сортування

Кролики і черепахи. Позиція елементів, що підлягають сортуванню відіграє велику роль у питанні продуктивності даного алгоритму. Великі елементи на початку списку не викликають проблеми, оскільки вони досить швидко переміщуються на свої місця. Однак, малі елементи у кінці списку переміщуються на його початок дуже повільно. Це призвело до того, що ці типи елементів було названо *кроликами* і *черепахами*, відповідно.

З метою підвищення швидкодії алгоритму, у свій час було здійснено чимало зусиль для зменшення кількості "черепах". Сортування змішуванням є порівняно непоганим, однак, усе ще у своєму найгіршому випадку має складність $O(n^2)$. Сортування гребінцем спершу порівнює великі елементи один з одним, а вже тоді поступово переходить до все менших і менших. Його середньостатистична швидкість приблизно рівна такій, як в алгоритмі Швидке сортування.

Сортування змішуванням – один із різновидів алгоритму сортування бульбашкою. Відрізняється від сортування бульбашкою тим, що сортування відбувається в обох напрямках, міняючи напрямок при кожному проході. Даний алгоритм лише трохи складніший за сортування бульбашкою, однак, вирішує так звану проблему "черепах".

Швидкодія

Ефективність алгоритму змішування рівна $O(n^2)$ для середньостатистичного та найгіршого випадку, водночас, вона прямує до $O(n)$ якщо список вже не погано відсортований, наприклад, якщо кожен елемент знаходиться у позиції, яка

відрізняється від кінцевої більше, ніж на k ($k \geq 1$), то його швидкодія рівна $O(k \cdot n)$.

Відмінності від сортування бульбашкою

Сортування змішуванням мало чим відрізняється від сортування бульбашкою. Єдина його відмінність у тому, що замість багаторазового проходження через список знизу вгору, він проходить по черзі знизу вгору і згори вниз. Він може досягати трохи вищої ефективності, ніж алгоритм сортування бульбашкою. Причиною цьому є те, що алгоритм сортування бульбашкою проходить по списку лише в одному напрямі, а тому за одну ітерацію елементи списку можна перемістити лише на один крок.

Наприклад, для того, щоб відсортувати список (2, 3, 4, 5, 1), алгоритму сортування змішуванням достатньо лише одного проходу, у той час, як алгоритму сортування бульбашкою знадобиться чотири проходи. Однак, один прохід сортування змішуванням слід рахувати за два проходи сортування бульбашкою. Зазвичай, сортування змішуванням удвічі швидше за сортування бульбашкою.

Іншою можливою оптимізацією є запам'ятовування попередніх перестановок. У наступній ітерації, перестановки не повторюватимуться, а тому алгоритм матиме коротші проходи по списку.

Псевдокод сортування бульбашкою:

```
procedure bubble()
 begin
    var byte i, byte j, byte t
    for i = 2 to N do
        for j = N down to i do
            if x[i-1] > x[j] then begin
                t = x[j - 1]; x[j - 1] = x[j]; x[j] = t
            end
 end
```

Сортування гребінцем – спрощений алгоритм сортування, розроблений Влодзімежом Добосєвічем (Wlodzimierz Dobosiewicz) у 1980 році, і пізніше заново досліджений та популяризований Стефаном Лейсі (Stephen Lacey) та Річардом Боксом (Richard Box), котрі написали про нього в журналі Byte Magazine у квітні 1991 р. Сортування гребінцем є поліпшенням алгоритму сортування бульбашкою, і конкурує у швидкодії з алгоритмом швидкого сортування. Основна його ідея полягає в тому, щоб усунути так званих "черепах" (малі значення) ближче до кінця списку, оскільки у сортуванні бульбашкою вони сильно уповільнюють процес сортування. ("Кролики" або великі значення на початку списку у сортуванні бульбашкою не викликають проблеми).

У сортуванні бульбашкою, коли два елементи порівнюються, вони завжди мають розрив (відстань один від одного) рівну 1. Основна ідея сортування гребінцем полягає у тому, що цей розрив може бути більший одиниці. (Алгоритм сортування Шелла також базується на даній ідеї, однак, він є модифікацією алгоритму сортування вставками, а не сортування бульбашкою).

Розрив починається зі значення, що рівне довжині списку, поділеного на фактор зменшення (зазвичай, 1.3; див. далі), і список сортується з урахуванням цього значення (при необхідності, воно заокруглюється до цілого). Потім розрив знову ділиться на фактор розриву, і список продовжує сортуватись з новим значенням, процес продовжується до тих пір, поки розрив рівний 1. Далі список сортується з розривом рівним 1 до тих пір, доки не буде повністю відсортований. Таким чином, фінальний етап сортування аналогічний такому ж, як у сортуванні бульбашкою, однак, до цього "черепахи" усуваються.

Фактор зменшення

Фактор зменшення справляє великий ефект на швидкість

алгоритму сортування гребінцем. В оригінальній статті, автор пропонує значення 1.3 після багатьох експериментів з іншими значеннями.

Текст описує процес вдосконалення алгоритму використовуючи значення $1/(1-\frac{1}{e^{\varphi}}) \approx 1.247330950103979$ в якості фактора зменшення. Стаття також містить приклад використання алгоритму на псевдокоді.

Псевдокод сортування гребінцем

```
function combSort(array input)
    gap = input.size

    loop until gap <= 1 and swaps == 0
        if gap > 1
            gap = gap / 1.3
            if gap == 10 or gap = 9
                gap = 11
            end
        end
        i = 0
        swaps = 0
        loop until i + gap <= input.size
            if input[i] > input[i+gap]
             swap(input[i], input[i + gap])
                swaps = 1
            end
            i = i + 1
        end
    end
end
```

Приклад 4.1. Сортування гребінцем на мові C++

```
void sort(data *array,dword size )
{
  if(!array||!size)
    return;
```

```
  dword jump=size;
  bool swapped=true;

  while(jump>1||swapped)
  {
    if(jump>1)
      jump=(dword)(jump/1.25);
    swapped=false;
    for(dword i=0; i+jump<size; i+=jump)
      if(array[i]>array[i+jump])
      swap(array,i,i+jump),swapped =true;
  }
}
```

Алгоритм сортування Шелла

Сортування Шелла – це алгоритм сортування, що є узагальненням сортування вставкою.

Алгоритм базується на двох тезах:

– Сортування включенням ефективне для майже впорядкованих масивів.

– Сортування включенням неефективне, тому що переміщує елемент тільки на одну позицію за раз.

Тому сортування Шелла виконує декілька впорядкувань включенням, кожен раз порівнюючи і переставляючи елементи, що знаходяться на різній відстані один від одного.

Сортування Шелла не є стійким.

Сортування Шелла названо на честь автора – Дональда Шелла, який опублікував цей алгоритм у 1959 році. В деяких пізніших друкованих виданнях алгоритм називають *сортуванням Шелла-Мацнера*, за ім'ям Нортона Мацнера. Але сам Мацнер стверджував: «Мені не довелось нічого робити з цим алгоритмом, і моє ім'я не має пов'язуватись з ним».

Ідея алгоритму

На початку обираються *m*-елементів: d_1, d_2, … , d_m, причому, $d_1 > d_2 > \ldots > d_m = 1$.

Потім виконується m впорядкувань методом включення, спочатку для елементів, що стоять через d_1, потім для елементів через d_2 і т. д. до $d_m = 1$.

Ефективність досягається тим, що кожне наступне впорядкування вимагає меншої кількості перестановок, оскільки деякі елементи вже стали на свої місця.

Псевдокод алгоритму сортування Шелла

Сам алгоритм не залежить від вибору *m* і *d*, тому будемо вважати, що вони задані.

```
Shell_Sort(A,N)
 begin
   for k = 1 to m do
       for i = d[k]+1 to N do
        begin
            a = A[i]
            j = i
            while  j-d[k] >= 1 i A[j-d[k]] > a do
             begin
                A[j] = A[j-d[k]]
                j = j – d[k]
                A[j] = a
             end
        end
 end
```

Коректність алгоритму

Оскільки $d_m = 1$, то на останньому кроці виконується звичайне впорядкування включенням всього масиву, а отже кінцевий масив буде впорядкованим.

Час роботи

Час роботи залежить від вибору значень елементів масиву *d*. Існує декілька підходів вибору цих значень:

- При $d_1 = \left\lfloor \frac{N}{2} \right\rfloor, d_2 = \left\lfloor \frac{d_1}{2} \right\rfloor, d_3 = \left\lfloor \frac{d_2}{2} \right\rfloor, \dots, d_m = 1$ виборі час роботи алгоритму, в найгіршому випадку, є $O(N^2)$.

- Якщо d – впорядкований за спаданням набір чисел виду $\frac{3^j - 1}{2}$, $j \in N$, $d_i < N$, то час роботи є $O(N^{\frac{3}{2}})$.
- Якщо d – впорядкований за спаданням набір чисел виду $2^i 3^j$, $i, j \in N$, $d_k < N$, то час роботи є $O(N \, log^2 N)$.

Приклад роботи

Проілюструємо роботу алгоритму на вхідному масиві *A* = (5, 16, 1, 32, 44, 3, 16, 7), *d* = (5,3,1).

1. Масив після впорядкування з кроком в 5: (*3*, **16**, 1, 32, 44, *5*, **16**, 7) – зроблено 1 обмін.
2. Масив після впорядкування з кроком 3: (*3*, **7**, 1, *16*, **16**, 5, *32*, **44**) – зроблено 3 обміну.
3. Масив після впорядкування з кроком 1: (1, 3, 5, 7, 16, 16, 32, 44) – зроблено 5 обмінів.

Отже, весь масив впорядковано за 9 операцій обміну.

Питання до розділу:

1. Поясніть терміни "кролики" і "черепахи".
2. Чим відрізняється алгоритм сортування змішуванням від алгоритму сортування бульбашками?
3. Чому сортування змішуванням удвічі швидше за сортування бульбашками?
4. Поясніть алгоритм сортування гребінцем.
5. Що таке фактор зменшення і яким чином він використовується в алгоритмі сортування Шелла?
6. Графічно проілюструйте принцип сортування масиву чисел *A* = (7,58,10,18,96,5,13,8), *d* = (5,3,1) в порядку зростання за методом Шелла.

Розділ 5. Швидке сортування

Швидке сортування Хоара – удосконалений метод сортування, що базується на обміні. К.Хоар запропонував алгоритм QuickSort сортування масивів, що дає на практиці відмінні результати і дуже просто програмується. Це сортування називають *швидким*, тому що на практиці воно виявляється найшвидшим методом сортування з тих, що оперують порівняннями.

Основна стратегія прискорення алгоритмів сортування – обмін між якомога більш віддаленими елементами вихідного файлу.

Ідея К. Хоара полягає в наступному: на кожному кроці методу ми спочатку вибираємо "середній" елемент, потім переставляємо елементи масиву так, що він поділяється на три частини: спочатку ідуть елементи, менші "середнього", потім рівні йому, а в третій частині – більші. Після такого розподілу масиву залишається тільки відсортувати першу і третю його частини, з якими ми зробимо аналогічно (розділимо на три частини). І так доти, поки ці частини не будуть складатися з одного елементу, а масив з одного елементу завжди відсортований.

Дано	**17**	35	48	52	27	9	15	**13**	89
1-й обмін	13	**35**	48	52	27	9	15	**17**	89
2-й обмін	13	**17**	18	52	27	9	**15**	35	89
3-й обмін	13	15	**18**	52	27	9	**17**	35	89
4-й обмін	13	15	**17**	52	27	**9**	18	35	89
5-й обмін	13	15	9	**52**	27	**17**	18	35	89
6-й обмін	13	15	9	**17**	27	52	18	35	89

Рисунок 5.1 – Ілюстрація роботи швидкого сортування

Вибір "середнього" – задача непроста, тому що потрібно, не виконуючи сортування, знайти елемент зі значенням максимально близьким до середнього. Тут, звичайно, можна просто вибрати довільний елемент (звичайно вибирають елемент, що стоїть посередині підмасиву, що сортується), але можемо вибирати з трьох елементів самого лівого, самого правого і того, що стоїть посередині.

Складність:

Аналіз складності алгоритму в середньому, що використовує гіпотезу про рівну імовірність усіх входів, показує, що:

$$C(n) = O(n \log^2 n), M(n) = O(n \log^2 n).$$

У гіршому випадку, коли в якості базового вибирається, наприклад, максимальний елемент підмасиву, складність алгоритму квадратична.

Швидке сортування є алгоритмом на основі порівнянь, і не є стійким.

Класична реалізація

В класичному варіанті, запропонованому Хоаром, з масиву обирався один елемент, і весь масив розбивався на дві частини по принципу: в першій частині – ті що не більші даного елементу, в другій частині – ті що не менші даного елементу. Процедура *Quicksort*(*A*,*p*,*q*) здійснює часткове впорядкування масиву *A* з *p*-го по *q*-й індекс, псевдокод:

```
Quicksort(A, p, q)
 begin
    if    p >= q   return
    r = A[p]
    i = p - 1
    j = q + 1
    while i < j  do
        repeat
             i = i + 1
        until A[i] >= r
```

```
        repeat
            j = j - 1
        until A[i] <= r
        if i < j then
            // Поміняти місцями значення A[i]та A[j]
        Quicksort(A, p, j)
        Quicksort(A, j + 1, q)
 end
```

Сучасна реалізація

На сьогодні в стандартних бібліотеках використовують таку реалізацію алгоритму, псевдокод:

```
Partition(A, p, q)
 begin
    x  =  A[q]
    i  =  p - 1
    for  i  =  p   to   q - 1  do
    if  A[j] <= x
        then  i = i + 1
            // Поміняти місцями значення A[i] та A[j]
    i = i + 1
    // Поміняти місцями значення A[i]та A[q]
    return i
end

Quicksort(A, p, q)
begin
    if p >= q return
    i = Partition(A, p, q)
    Quicksort(A, p, i  -1)
    Quicksort(A, i + 1, q)
end
```

Аналіз

Час роботи алгоритму сортування залежить від збалансованості, що характеризує розбиття. Збалансованість, у свою чергу залежить від того, який елемент обрано як опорний (відносно якого елементу виконується розбиття). Якщо розбиття збалансоване, то асимптотично алгоритм

працює так само швидко як і алгоритм сортування злиттям. У найгіршому випадку, асимптотична поведінка алгоритму настільки ж погана, як і в алгоритмі сортування включенням.

Найгірше розбиття

Найгірша поведінка має місце у тому випадку, коли процедура, що виконує розбиття, породжує одну підзадачу з n-1 елементами, а другу – з 0 елементами. Нехай таке незбалансоване розбиття виникає при кожному рекурсивному виклику. Для самого розбиття потрібен час $O(n)$. Тоді, рекурентне співвідношення для часу роботи, можна записати так:

$$T(n) = T(n - 1) + T(0) + O(n) = T(n - 1) + O(n)$$

Розв'язком такого співвідношення є $T(n) = O(n^2)$.

Найкраще розбиття

В найкращому випадку процедура *Partition* ділить задачу на дві підзадачі, розмір кожної не перевищує $n/2$. Час роботи, описується нерівністю:

$$T(n) \leq 2T(n / 2) + O(n)$$

Тоді:

$T(n) = O(n \log n)$ – асимптотично найкращий час.

Середній випадок

Математичне очікування часу роботи алгоритму для всіх можливих вхідних масивів є $O(n \log n)$, тобто середній випадок ближчий до найкращого.

Модифікації

В середньому алгоритм працює дуже швидко, але на практиці, всі можливі вхідні масиви мають не однакову імовірність. Тоді, шляхом додання рандомізації вдається отримати середній час роботи в будь-якому випадку.

Рандомізованний алгоритм

В рандомізованному алгоритмі, при кожному розбитті випадковий елемент обирається в якості опорного, псевдокод:

```
Randomized_Partition(A, p, q)
 begin
    i = Random(p, q)
    // Поміняти місцями значення A[i]та A[q]
    return Partition(A, p, q)
 end

Randomized_Partition(A, p, q)
 begin
    if p >= q return
    i = Randomized_Partition(A, p, q)
    Randomized_Quicksort(A, p, i - 1)
    Randomized_Quicksort(A, i + 1, q)
 end
```

Питання до розділу:

1. Чому сортування Хоара називають швидким?
2. Поясніть основну ідею Хоара.
3. Чим сучасна реалізація сортування Хоара відрізняється від класичної?
4. Яку складність має алгоритм сортування Хоара і від чого вона залежить?
5. Що дає рандомізація?

Розділ 6. Спеціалізовані алгоритми внутрішнього сортування

Сортування підрахунком – алгоритм впорядкування, що застосовується при малій кількості різних елементів (ключів) у масиві даних. Час його роботи лінійно залежить, як від загальної кількості елементів у масиві, так і від кількості *різних* елементів.

Ідея алгоритму

Ідея алгоритму полягає в наступному: спочатку підрахувати скільки разів кожен елемент (ключ) зустрічається в вихідному масиві. Спираючись на ці дані можна одразу вирахувати, на якому місці має стояти кожен елемент, а потім за один прохід поставити всі елементи на свої місця.

Псевдокод алгоритму

Для простоти будемо вважати, що всі елементи (ключі) є натуральними числами, що лежать в діапазоні *1..K*. Процедура *Counting_Sort(A)* виконує сортування масиву *A*, псевдокод:

```
Counting_Sort(A)
 begin
    1 // C – масив з K елементів, заповнений нулями
    2 for i = 1 to length[A] do
    3     C[A[i]] = C[A[i]] + 1
    4 for i = 2 to K do
    5     C[i] = C[i] + C[i  -1]
    6 for i = length[A] downto 1 do
    7     B[C[A[i]]] = A[i]
    8        C[A[i]] = C[A[i]] - 1
    9 A = B
 end
```

Аналіз алгоритму

В алгоритмі присутні тільки прості цикли: в рядках 2, 6, 9 – цикл довжиною *N* (довжина масиву), в рядку 4 – цикл довжиною *K* (величина діапазону). Отже складність роботи

алгоритму є *O(N+K)*.

В алгоритмі використовуються два додаткових масиви: *C* і *B*. Тому алгоритм потребує *O(N+K)* додаткової пам'яті.

В такій реалізації алгоритм є стійким. Саме ця його властивість дозволяє використовувати його як частину інших алгоритмів сортування (напр. сортування за розрядами).

Використання даного алгоритму є доцільним тільки у випадку малих *K* (порядку *N*).

Сортування за розрядами (англ. *Radix sort*) – швидкий стійкий алгоритм впорядкування даних. Застосовується для впорядкування елементів, що є ланцюжками над будь-яким скінченним алфавітом (напр. рядки, або цілі числа). В якості допоміжного використовує будь-який інший стійкий алгоритм сортування.

Алгоритм застосовувався для впорядкування перфокарт.

Ідея алгоритму

Ідея полягає в тому, щоб спочатку впорядкувати всі елементи за молодшим розрядом, потім стабільно впорядкувати за другим розрядом, потім за третім і так далі аж до найстаршого. Оскільки, припускається, що кожен розряд приймає значення з невеликого діапазону, то кожен цикл впорядкування можна виконувати швидко і з малими затратами пам'яті.

Приклад роботи

На рис. 6.1 показано, як впорядковувати таким алгоритмом масив трицифрових чисел.

```
572     572     523     266
266     523     349     349
783 --> 783 --> 266 --> 523
523     266     572     572
349     349     783     783
          ^       ^      ^
```

Рисунок 6.1 – Ілюстрація роботи сортування за розрядами

Аналіз

Час роботи кожного циклу сортування залежить від того алгоритму, що використовується в якості допоміжного. Найчастіше використовують сортування підрахунком, що працює за час *O(N+K)* (де *N* – кількість елементів в масиві; *K* – кількість символів у алфавіті, якщо впорядковуються десяткові числа, то *K* = 10) і використовує додатково *O(N+K)* пам'яті. Всього здійснюється стільки циклів впорядкування, скільки розрядів у максимальному елементі.

Загальна складність роботи алгоритму з використанням сортування підрахунком є *O(D · (N + K))* (*D* – кількість розрядів). Якщо впорядковувати цим алгоритмом цілі числа, то складність буде *O(N · log M)*, де *M* – найбільший елемент масиву.

Питання до розділу:

1. Від чого залежить час роботи алгоритму сортування підрахунком?
2. В чому полягає ідея алгоритму сортування підрахунком?
3. Яка складність роботи алгоритму сортування підрахунком і коли його використовують?
4. Навіщо використовується додаткова пам'ять в алгоритмі сортування підрахунком?
5. З якою метою застосовується алгоритм сортування за розрядами?
6. В чому полягає ідея алгоритму сортування за розрядами?
7. Як пов'язані між собою ці два алгоритми?

Розділ 7. Алгоритми зовнішнього сортування

Зі світу "електронних" швидкостей перемістимося в світ менших швидкостей, світ з механічними переміщеннями й впливом інерції. Хоча диск і робить сотні обертів у секунду, проходять мілісекунди до моменту, коли потрібний запам'ятовувальний елемент виявиться під головкою читання/запису. Тому якщо будь-який відомий нам спосіб упорядкування в пам'яті застосувати "механічно" до даних на диску, витрати часу виявляться, швидше за все, неприйнятними.

Сортування даних на магнітних дисках називають *зовнішніми*.

Складнощі при сортуванні на диску

Досліджувати, а тим більш конструювати зовнішнє сортування неможливо, не уявляючи, хоча б трохи, специфіки дисків.

Диски мають запам'ятовуючі магнітні поверхні й обертаються з великою постійною швидкістю під головками читання/запису. Головка обслуговує одну поверхню.

При фіксованому положенні головок доступ відбувається до даних треків (кіл), що перебувають під головками. Сукупність цих треків назвемо поточним циліндром.

Головки кріпляться на штанзі, що подібна до "коромисла" програвача. Після переміщення блоку головок (повороту штанги) поточним стає наступний циліндр. Циліндри нумеруються в порядку їхнього проходження блоком головок при русі штанги в одну сторону. Відстанню між елементами даних назвемо різницю номерів циліндрів, де вони розміщені.

Адресація (нумерація) елементів пам'яті на диску йде в межах циліндра з переходом при вичерпанні його обсягу на наступний, потім на третій і так далі. Файл даних записується послідовно за однією адресою, але якщо вільної ділянки не

вистачило, продовжується на іншій ділянці й, можливо, на іншому циліндрі (випадок фрагментації файлу).

Хоча диски називають пам'яттю прямого доступу, час звертання до даних набагато більшого порядку, ніж час читання/запису в основній пам'яті, через:

а) затримки, пов'язані з очікуванням моменту, коли потрібний елемент циліндра пройде під головкою;

б) переміщення головок, коли потрібні дані не з поточного циліндра.

Прохід – це проходження файлу в одному напрямку зі зчитуванням даних в пам’ять й, можливо, їхнім поверненням у файл.

По можливості, більшу частину роботи з даними слід виконувати використовуючи оперативну пам’ять, не звертаючись до диску до кінця обробки цієї частини.

Стосовно до зовнішнього сортування, це означає впорядкування можливо більших частин файлу внутрішнім сортуванням: ми заміняємо "швидкими" переміщеннями в пам'яті переміщення даних на диску. Зменшуючи число актів їхнього читання й перезапису, ми одночасно виключаємо численні затримки "а", "б".

Збільшення фізичного розміру зчитуваної (записуваної) порції даних скорочує затримку й число актів читання (запису), а разом з ними й число проміжних рухів блоку головок, які назвемо ходами штанги або просто ходами.

Час зовнішнього сортування залежить від:

а) внутрішнього сортування частин файлу;

б) багаторазового зчитування й запису даних на диск;

в) ходів головки між актами зчитування/запису;

г) дій у пам'яті при злитті впорядкованих частин.

Сортування простим злиттям

В основі даного методу сортування лежить процедура злиття двох упорядкованих масивів. Ці масиви повинні бути

об'єднані таким чином, щоб вийшов один упорядкований масив.

Нехай є два масиви *A* і *B*, впорядковані за зростанням ключів. Злиття полягає в почерговому пересиланні елементів з масивів *A* і *B* у зону виведення *C*. Порядок пересилання в зону виведення залежить від результату попарного порівняння ключів масивів *A* і *B*.

Сортування злиттям проводиться на основі використання розглянутої вище процедури. Сутність сортування полягає в тому, що масив, який ми намагаємося відсортувати, розділяється на рівні групи елементів. Групи упорядковуються, а потім попарно зливаються, утворюючи нові групи.

На першому етапі кожна група містить два сусідніх елементи вихідного масиву. Елементи усередині груп упорядковуються (наприклад, методом вставки або включення). Потім відбувається попарне злиття груп. Кількість груп у списку зменшується доти, доки не буде отримана одна упорядкована група. Якщо число елементів непарне, то вводиться додатковий елемент із максимальним значенням. Після сортування він відкидається. Якщо число груп, сформованих на першому етапі, непарне, то непарна група переписується без злиття.

Нехай є 2 масиви, кожен з яких вже відсортований, розглянемо як буде відбуватися сортування злиттям (рис. 7.1).

масив1	**1**	**3**	**5**	**8**	**9**	**11**	**17**							
масив2	2	6	7	10	14	15	19							
результат	**1**	2	**3**	**5**	6	7	**8**	**9**	10	**11**	14	15	**17**	19

Рисунок 7.1 – Ілюстрація роботи сортування злиттям

Зовнішнє сортування злиттям

Зовнішнє сортування сортує файли, які не поміщаються повністю в оперативну пам'ять.

Зовнішнє сортування сильно відрізняється від внутрішнього. Справа в тому, що доступ до файлу є послідовним, а не паралельним як це було в масиві. І тому зчитувати файл можна тільки блоками і цей блок відсортувати в пам'яті й знову записувати у файл.

Принципову можливість ефективно відсортувати файл, працюючи з його частинами і не виходити за межі частини забезпечує алгоритм злиття.

Під **злиттям** розуміється об'єднання двох (або більше) упорядкованих послідовностей в одну впорядковану послідовність за допомогою циклічного вибору елементів, доступних у цей момент.

Злиття – набагато більш проста операція, ніж сортування.

Ми розглянемо 2 алгоритми злиття:

1. Пряме злиття. Алгоритм Боуза-Нельсона
2. Природне (Неймановське) злиття.

Пряме злиття. Алгоритм Боуза-Нельсона

1. Послідовність *a* розбивається на дві половини *b* і *c*.
2. Послідовності *b* і *c* зливаються за допомогою об'єднання окремих елементів в упорядковані пари.
3. Отриманій послідовності привласнюється ім'я *a*, після чого повторюються кроки 1 і 2; при цьому впорядковані пари зливаються в упорядковані четвірки.
4. Попередні кроки повторюються, при цьому четвірки зливаються у вісімки і так далі, поки не буде впорядкована вся послідовність, тому що довжини послідовностей щоразу подвоюються.

Приклад 7.1. Ілюстрація алгоритму Боуза-Нельсона

Початкова послідовність:

A = 44 55 12 42 94 18 06 67

1 *b* = 44 55 12 42
c = 94 18 06 67
a = 44 94' 18 55' 06 12' 42 67'
2 *b* = 44 94' 18 55'
c = 06 12' 42 67'
a = 06 12 44 94' 18 42 55 67'
3 *b* = 06 12 44 94'
c = 18 42 55 67'
a = 06 12 18 42 44 55 67 94

Операція, що однократно обробляє всю множину даних, називається *фазою*.

Найменший підпроцес, що, повторюючись, утворить процес сортування, називається *проходом* або *етапом*.

У нашому прикладі сортування здійснюється за три проходи. Кожний прохід складається з фази розбивки і фази злиття.

Головним мінусом сортування злиттям є подвоєння розміру пам'яті, спочатку зайнятої даними, що сортуються. Розглянемо алгоритм із рекурсивним актом злиття, запропонований Боузом і Нельсоном, що не потребує резерву пам'яті.

Він заснований на очевидній ідеї: злити дві рівні впорядковані частини можна злиттям їхніх початкових половин, злиттям кінцевих і злиттям 2-ї половини 1-го результату з 1-ю половиною 2-го результату, наприклад:

Злиття початків та злиття кінців

1 7 8 11 + 4 5 6 9

Злиття в середній частині

[1 4] 5 7 + 6 8 [9 11] =

[1 4] [5 6 7] [9 11]

Рисунок 7.2 – Пряме злиття. Алгоритм Боуза-Нельсона

Якщо частини не рівні або не діляться точно навпіл, процедуру уточнюють належним чином. Аналогічне злиття "половинок" можна звести до злиття "четвертушок", "восьмушок" і так далі; має місце рекурсія. Псевдокод:

```
Const n = 200

Class srt
 begin
    word kl
    array of real z[1..50]
 end

var Array of srt A[1..n]; word j

Procedure Bose(boolean voz)
begin
    var word m,j; srt x // srt - тип даних, що
сортуються
     Array of srt A[1..65520 div Sizeof(srt)]

     Procedure Sli(word j, r, m)//r - відстань між
початками  частин, що зливаються, де m - їхній
розмір, j - найменший номер запису
      Begin
         if j + r <= n then
             if m = 1 then
               begin
                  if voz Xor (A[j].kl < A[j + r].kl)
```

```
                    then begin
                        x = A[j]
                        A[j] = A[j + r]
                        A[j + r] = x
                    end
              end
              else begin
                  m = m div 2
                  Sli(j, r, m) //Злиття "початків"
                  if j + r + m <= n then
                  Sli(j + m, r, m) //Злиття "кінців"
                  Sli(j + m, r - m, m)
              end //Злиття в центральній частині
     end //процедури Sli
   begin
       m = 1
       repeat
          j = 1 //Цикл злиття списків рівного розміру
          while j + m <= n do
          begin
              Sli(j, m, m); j = j + m + m
          end
          m = m + m //Подвоєння розміру списку перед
початком нового проходу
          until m >= n //Кінець циклу, що реалізує
все дерево злиттів
          end // процедури Bose
begin
    Randomize() // ініціалізація генератора
псевдовипадкових чисел
    for j = 1 to n do begin
        A[j].kl = Random(65535); write(A[j].kl)
    end
    Read() // Очікування натиснення на клавішу
    Bose(A, true)
    for j = 1 to n do
        write(A[j].kl)
     Read() // Очікування натиснення на клавішу
 End
```

Природне (Нейманівське) злиття

Поєднуються впорядковані частини, що спонтанно

виникли у вихідному масиві; вони можуть бути також наслідком попередньої обробки даних. Розраховувати на однаковий розмір частин, що зливаються, не доводиться.

Записи, що йдуть у порядку не зменшення ключів, зчіплюються у підсписок. Мінімальний підсписок один запис.

Приклад 7.2. Ілюстрація Нейманівського злиття

Нехай дані ключі записів: 5 7 8 3 9 4 1 7 6

Шукаємо підсписки (рис. 7.3).

5 → 7 → 8 3 → 9 4 1 → 7 6

Рисунок 7.3 – Пошук підсписків

В один загальний список з'єднуються 1-й, 3-й, 5-й, … підсписки, в інший – 2-й, 4-й, … підсписки.

Зробимо злиття 1 підсписку 1 списку й 1 підсписку 2 списку, 2 підсписку 1 списку й 2 підсписку 2 списку і так далі.

Будуть отримані наступні ланцюги:

3 ---> 5 ---> 7 ---> 8 ---> 9 та 1 ---> 4 ---> 7

Підсписок, що складається із запису "6", пари не має й "примусово" поєднується з останнім ланцюгом, що приймає вигляд 1 ---> 4 ---> 6 ---> 7.

При нашому невеликому числі записів 2-й етап, на якому зливаються два ланцюги, виявиться останнім.

У загальному випадку на кожному етапі підсписок – результат злиття початкових підсписків 1 і 2 списку стає початком нового 1-го списку, а результат злиття наступних двох підсписків – початком 2-го списку. Наступні утворені підсписки по черзі включаються в 1-й і в 2-й список.

Для програмної реалізації створюють масив sp: елемент sp[i] – це номер запису, який слідує за i-м.

Останній запис одного підсписку посилається на перший

запис іншого, а для розрізнення кінців підсписків це посилання доповнюється знаком мінус.

Структура, що проявляє частини деякого початкового масиву (рис. 7.4):

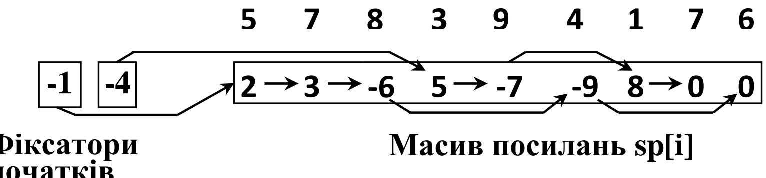


Рисунок 7.4 – Масив посилань

```
repeat //Повторення актів злиття підсписків
    if A[j].kl < A[i].kl //Вибирається менший запис
    then begin
        sp[k] = j; k = j; j = sp[j]
        if j <= 0
            then begin
                sp[k] = i;
                repeat m = i; i = sp[i] until i <= 0
            end
    end
    else begin
        sp[k] = i; k = i; i = sp[i]
        if i<=0 then
            begin sp[k] = j
                repeat m = j; j = sp[j] until j <= 0
            end
    end
    if j <= 0 then begin
        sp[m] = 0; sp[p] = -sp[p]; i = -i; j = -j
        if j != 0 then p = r; k = r; r = m
    end
until j == 0
```

У кінець сформованого підсписку завжди заноситься нульове посилання (sp[m] = 0), тому що він може виявитися

останнім.

Дія sp[*p*] = -sp[*p*] позначає мінусом кінець раніше побудованого підсписку.

У змінних *i* та *j* посилання на початки нових підсписків, що зливаються – зі знаком мінус; його знімаємо. Перехід до нових підсписків вимагає відновлення змінних *p*, *k*, *r*.

Сортування методом поглинання

Маючи кілька частин файлу й почавши зі злиття двох з них, будемо зливати всі наступні з більшою (зростаючою) упорядкованою частиною. Вона наче поглинає частину за частиною. Перед поглинанням чергова частина файлу зчитується в зону "A" пам'яті, там упорядковується й залишається. Початок раніше впорядкованої частини зчитується в зону "B" і починається злиття, що переривається зчитуванням у зону "B", коли вона спорожнюється. У міру заповнення зони "C" записами акту злиття, вміст її переписується у файл (на місце поглинутої частини і далі убік кінця файлу). Якщо при злитті взяті всі записи тієї частини, що поглинається, то поглинання завершується передачею у файл із зони "C" залишку результату. Злиття також завершується, якщо вичерпано раніше впорядковану частину. Поглинання нею чергової частини відбулося.

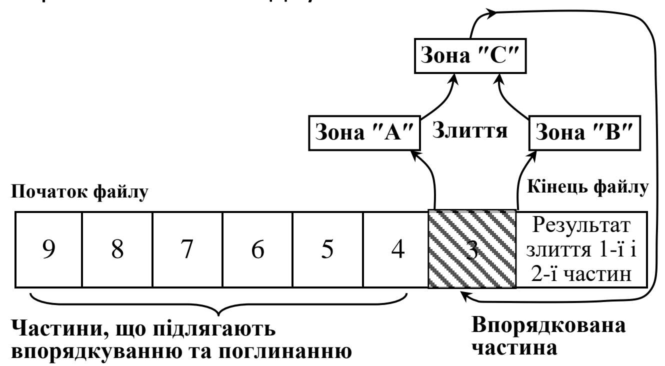


Рисунок 7.5 – Схема сортування поглинанням

Ходів у даному методі буде мало й під час сортування ви не почуєте характерного поскрипування механізму головок. Це дає економію часу й при невеликому розмірі файлу сортування проходить швидко.

Човникове балансове злиття

На 1 етапі внутрішнього сортування частин у файлі створюють M упорядкованих частин, по можливості більшого розміру R. До них застосовують пряме злиття.

Створюють резервний дисковий простір файлу й розташовують його безпосередньо до або після частин, що зливаються, і переміщуються до наступних частин по завершенні злиття. Його розмір не менше розміру R частини. Резерв і простір найближчої частини будуть заповнені результатом злиття двох перших частин, при цьому простір 2-ї частини звільниться і стане резервом, необхідним для злиття наступної пари частин і так далі.

У міру злиття частин, резерв переміщується від початку файлу до кінця, потім назад і так далі, як човник. Оскільки місце результату не відділене від частин, що зливаються, ходи невеликі, поки самі частини невеликі. У процесі сортування "човник" збільшується, тому що ростуть частини:

а) злиття частин розміром *R* (проміжна ситуація):

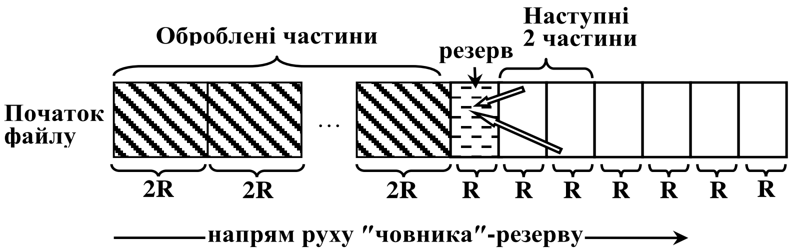


Рисунок 7.6 – Злиття частин розміром *R*

б) наступний "прохід" човника злиття частин розміром 2R:

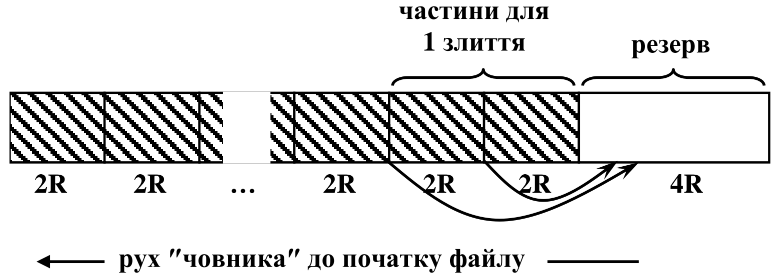


Рисунок 7.7 – Злиття частин розміром 2R

Стрілками показане переміщення даних при злитті.

Відмітимо, що резерв можна збільшувати, коли він наприкінці файлу; це відбувається один раз за 2 прогони човника. Усе було б добре, однак, човник завбільшки половини файлу може займати його початок. Добре б обмежити ріст розміру "човника", тому що цей ріст збільшує розмір ходів.

Шляхом модифікації кінцевих етапів злиття, можна домогтися, щоб цей ріст зупинився на розмірі D = 1/6 розміру файлу.

Для цього однак треба, щоб програма так визначила вихідні розмір і положення резерву (на початку або наприкінці файлу), щоб у момент, коли залишаться всього 3 великі частини, резерв, що має розмір D, стояв на початку файлу.

в) етап злиття (з підетапами), коли у файлі 6 частин:

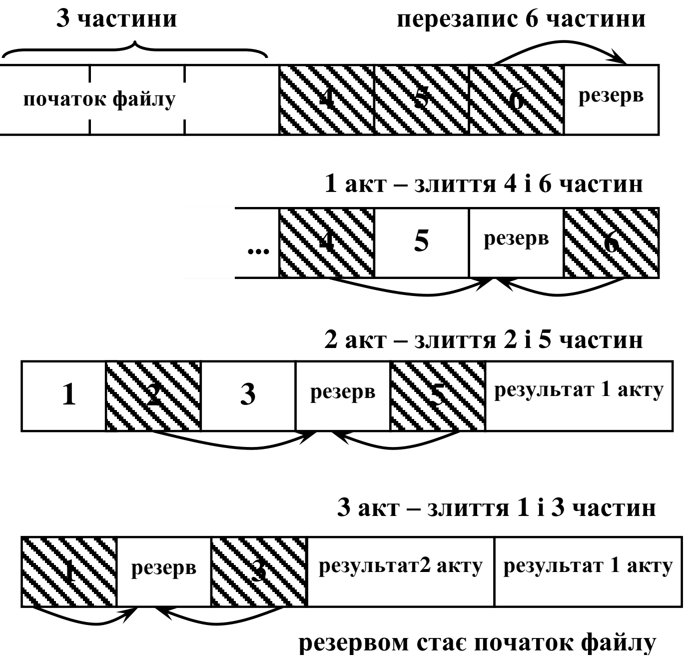


Рисунок 7.8 – Етап злиття (з підетапами)

г) етап злиття "по половинах" (природне злиття), коли частин 3. Кінцева частина файлу на цьому етапі не використовується.

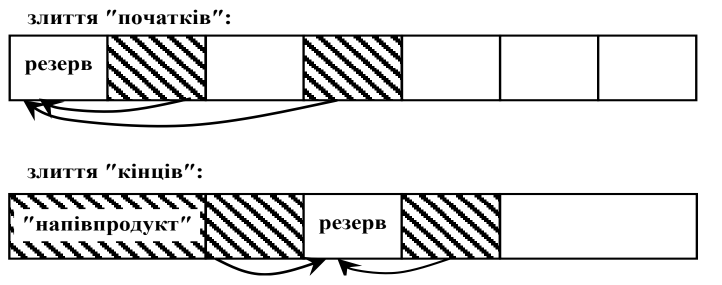
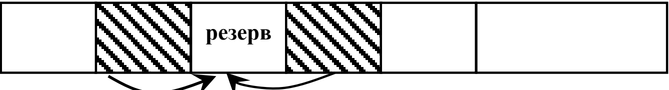
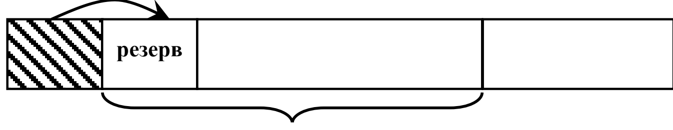


Рисунок 7.9 – Етап злиття "по половинах" (1-ша частина)

заключний 3 акт злиття



витіснення резерву в початок файлу:



остаточне місце результату злиття

Рисунок 7.9 – Етап злиття "по половинах" (2-га частина)

д) заключний етап злиття; спочатку беремо для злиття першу половину кінцевої частини й всю початкову частину:

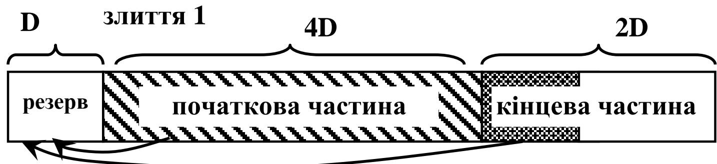


Рисунок 7.10 – Заключний етап злиття

По закінченні злиття 1 з'ясовується, чи треба змістити залишок початкової частини, щоб звільнити місце для кінця результату й продовжити злиття ("підзлиття"). Підзлиття не потрібне, якщо в ході злиття початкова частина вичерпана, треба лише витиснути резерв у кінець файлу, переписавши закінчення.

зміщення залишку і підзлиття:

Рисунок 7.11 – Підзлиття

І при злитті 1, і при підзлитті розмір резерву не менше мінімально необхідного.

Питання до розділу:

1. Поясніть сортування простим злиттям.
2. Чим зовнішнє сортування відрізняється від внутрішнього?
3. Що розуміють під злиттям?
4. Який порядок прямого злиття і в чому його недолік?
5. Яким чином рекурсивний алгоритм злиття Боуза-Нельсона вирішує проблему прямого злиття?
6. Поясніть природне (Нейманівське) злиття на власному прикладі.
7. Яке сортування називається зовнішнім і чому його складнощі?
8. В чому переваги сортування методом поглинання?
9. Яким чином в човниковому балансовому злитті розмір резерву зводиться до мінімуму?

Розділ 8. Алгоритми пошуку. Послідовний пошук елементу. Бінарний пошук елементу. Послідовний пошук рядка

Пошук – знаходження будь-якої конкретної інформації у великому обсязі раніше зібраних даних.

Дані діляться на записи, і кожний запис має хоча б один ключ. Ключ використовується для того, щоб відрізнити один запис від іншого.

Метою пошуку є знаходження всіх записів, що підходять до заданого ключа пошуку.

Пошук елементу в масиві

Для знаходження інформації в неупорядкованому масиві потрібен послідовний пошук, що починається з першого елементу й закінчується при виявленні необхідних даних або при досягненні кінця масиву. Цей метод підходить для пошуку неупорядкованої інформації, але також можна використовувати його й на відсортованих даних. Однак якщо дані вже відсортовані, можна застосувати двійковий пошук, що знаходить дані швидше.

Послідовний пошук

Послідовний пошук дуже легко запрограмувати. Наведений нижче псевдокод ілюструє пошук у масиві символів відомої довжини, поки не буде знайдений елемент із заданим ключем:

```
function int LinearSearch (Array A, int L, int R, int
Key)
 begin
    for X = L to R do
        if A[X] = Key then
            return X
    return -1 // елемент не знайдено
end
```

Функція повертає індекс необхідного елементу, якщо такий існує, або -1 у противному випадку.

Зрозуміло, що послідовний пошук у середньому переглядає (*n / 2)* елементів. У найкращому разі він перевіряє тільки один елемент, а в найгіршому – *n*. Якщо інформація зберігається на диску, пошук може забирати тривалий час. Але якщо дані не впорядковані, послідовний пошук – єдино можливий метод.

Двійковий пошук

Якщо дані, у яких здійснюється пошук, відсортовані, для знаходження елементу можна застосовувати метод, що набагато перевершує попередній – *двійковий пошук*. У двійкового пошуку є й інші назви: *дихотомічний пошук*, *логарифмічний пошук*, *пошук розподілом навпіл*. У ньому застосовується метод половинного розподілу. Спочатку перевіримо середній елемент. Якщо він більший, ніж шуканий ключ, перевіримо середній елемент першої половини, у протилежному випадку – середній елемент другої половини. Будемо повторювати цю процедуру доти, поки шуканий елемент не буде знайдений або поки не залишиться чергового елементу.

Наприклад, знайдемо число 4 у масиві:

1 2 3 4 5 6 7 8 9

При двійковому пошуку спочатку перевіряється середній елемент – число 5. Оскільки воно більше, ніж 4, пошук триває в першій половині:

1 2 3 4 5

Середній елемент тепер дорівнює 3. Це менше, ніж 4, тому перша половина відкидається. Пошук триває в другій

частині:

4 5

Цього разу шуканий елемент знайдений.

У двійковому пошуку кількість порівнянь у найгіршому разі дорівнює $\log^2 n$.

У середньому випадку кількість порівнянь значно нижче, а в кращому – дорівнює 1. Двійковий пошук суттєво швидший за лінійний, відносно простий в реалізації і загальновживаний. Проте, в реальних програмах трапляються випадки помилкового використання лінійного пошуку в упорядкованих даних, що призводять до значного зменшення швидкодії.

Нижче наведений псевдокод функції двійкового пошуку.

```
//Ітеративна версія
BinarySearch(A[0..N-1], value)
 begin
    low = 0
    high = N - 1
    while (low <= high)
     begin
        mid = (low + high) / 2
        if (A[mid] > value)
            high = mid - 1
        else if (A[mid] < value)
            low = mid + 1
        else
        return mid // знайдено
     end
     return -1 // не знайдено
end
```

Одним із варіантів реалізації алгоритму є рекурсивна функція, що отримує масив, шукане значення та початковий і кінцевий індекси елементів в масиві. Далі наведено псевдокод рекурсивної версії двійкового пошуку.

```
//Рекурсивна версія
BinarySearch(A[0..(N - 1)], value, low, high)
 begin
    if (high < low)
        return -1 // не знайдено
        mid = (low + high) / 2
    if (A[mid] > value)
        return BinarySearch(A, value, low, mid - 1)
    else if (A[mid] < value)
        return BinarySearch(A, value, mid+1, high)
    else
        return mid // знайдено
 end
```

Розглянемо псевдокод ітеративної версії з доповненнями для двійкового пошуку, завдяки доповненням даний алгоритм працюватиме швидше.

```
//Ітеративна версія з доповненнями
int first = 0 // Перший елемент у масиві
int last = n  // Елемент, НАСТУПНИЙ ЗА останнім
int mid

if (n == 0)
    begin
         /* масив порожній */
    end
else if (a[0] > x)
    begin
    //елемент не знайдено; якщо вам треба вставити
його зі зсувом - то в позицію 0
    end
else if (a[n - 1] < x)
    begin
    //елемент не знайдено; якщо вам треба вставити
його зі зсувом - то в позицію n
    end
//Якщо переглядається непорожній фрагмент first < last
while (first < last)
    begin
```

```
// УВАГА! На відміну від більш простого
(first + last) / 2, цей код стійкий до переповнень
        mid = first + (last - first) / 2

        if (x <= a[mid])
            begin
                last = mid
            end
        else
            begin
                first = mid + 1
        end
    end

    if (a[last] == x)
        begin
        // Шуканий елемент знайдено. last – шуканий
індекс
    end
else
    begin
        //Шуканий елемент не знайдено. Але якщо треба
його вставити зі зсувом, то його місце – last
    end
```

Питання до розділу:

1. Що таке пошук?
2. Для пошуку якої інформації використовується послідовний метод?
3. В якому випадку використовують двійковий пошук?
4. Поясніть процедуру двійкового пошуку на власному прикладі.
5. Чому дорівнює мінімальна і максимальна кількість порівнянь при двійковому пошуку?

Розділ 9. Алгоритми пошуку рядка у тексті

Пошук послідовності елементів в масиві

Одне з найпростіших завдань пошуку інформації – **пошук точно заданого підрядка у рядку**. Проте, це завдання надзвичайно важливе – воно застосовується в текстових редакторах, СУБД, пошукових машинах тощо.

Пошук рядка формально визначається в такий спосіб. Нехай заданий масив T з N елементами і масив *W* з *M* елементами, причому $0 < M \le N$. Пошук рядка виявляє перше входження *W* у *T*, результатом будемо вважати індекс *i*, що вказує на перший з початку рядка (з початку масиву *T*) збіг зі зразком (словом).

Приклад 9.1. Потрібно знайти всі входження зразка *W* = abaa у тексті *T* = abcabaabcabca.

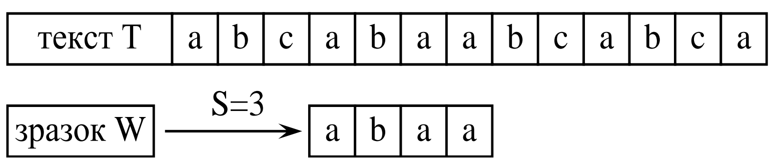


Рисунок 9.1 – Послідовний пошук у тексті

Зразок входить у текст тільки один раз, зі зсувом *S* = 3, індекс *i* = 4.

Алгоритм прямого (послідовного) пошуку

Ідея алгоритму:

1) *i* = 1,

2) порівняти *i*-й символ масиву *T* з першим символом масиву W,

3) збіг → порівняти другий символ і так далі,

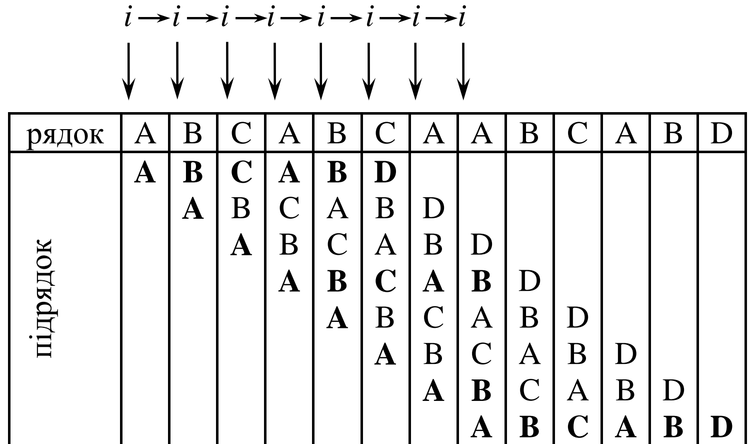
4) розбіжність → *i* = *i* + 1 і перехід до пункту 2.

Умова закінчення алгоритму:

1) підряд *M* порівнянь вдалі,

2) $i + M > N$, тобто слово не знайдене.

Приклад 9.2. Потрібно знайти підрядок *W* = abcabd у тексті *T* = abcabcaabcabd.



рядок	A	B	C	A	B	C	A	A	B	C	A	B	D
підрядок	**A**	**B**	**C**	**A**	**B**	**D**							
		A	B	C	A	B	D						
			A	B	C	A	B	D					
				A	**B**	**C**	**A**	**B**	D				
					A	B	C	A	B	D			
						A	B	C	A	B	D		
							A	**B**	C	A	B	D	
								A	**B**	**C**	**A**	**B**	**D**

Рисунок 9.2 – Ілюстрація послідовного пошуку

Зразок входить у текст тільки один раз, зі зсувом $S = 7$, індекс $i = 8$.

Складність алгоритму:

Гірший випадок. Нехай масив $T \rightarrow$ {AAA....AAAB}, довжина $|T| = N$, зразок $W \rightarrow$ {A….AB}, довжина $|W| = M$. Очевидно, що для виявлення збігу наприкінці рядка буде потрібно зробити порядку $N \cdot M$ порівнянь, тобто $O(N \cdot M)$.

Недоліки алгоритму:

1) висока складність – $O(N \cdot M)$, у найкращому випадку – $O((N - M + 1) \cdot M)$;

2) після розбіжності перегляд завжди починається з першого символу зразка й тому може включати символи T, які раніше вже проглядалися (якщо рядок зчитується із вторинної

пам'яті, то такі повернення займають багато часу);

3) інформація про текст *T*, що отримана при перевірці даного зсуву *S*, ніяк не використовується при перевірці наступних зсувів.

Приклад псевдокоду алгоритму послідовного пошуку рядка у тексті (функція повертає результат пошуку у рядку *S*, підрядка *X*. Place – місце першого входження):

```
Function boolean Search(str S, str X, byte Place)
 begin
    var boolean Res; int i
    Res = FALSE
    i = 1
    while (i <= Length(S) - Length(X) + 1) And Not(Res) do
        if Copy(S,i,Length(X)) = X then
            begin
                Res = TRUE
                Place = i
            end
        else i  = i + 1
        Search = Res
  end
```

Алгоритм Рабіна-Карпа

Алгоритм Рабіна-Карпа – це алгоритм пошуку рядка, який шукає шаблон, тобто підрядок, у тексті використовуючи хешування. Він був розроблений в 1987 році Майклом Рабіном і Річардом Карпом.

Алгоритм рідко використовується для пошуку одиночного шаблону, але має значну теоретичну значимість і дуже ефективний у пошуку збігів множинних шаблонів. Одне з найпростіших практичних застосувань алгоритму Рабіна-Карпа полягає у визначенні плагіату.

Використання хешування для пошуку підрядків зсувом

Алгоритм Рабіна являє собою модифікацію лінійного алгоритму; він заснований на досить простій ідеї.

«Уявимо собі, що в рядку *A*, довжина якого дорівнює *m*, ми шукаємо зразок *X* довжиною *n*. Виріжемо "віконечко" розміром *n* і будемо рухати його по вхідному рядку. Нас цікавить, чи не збігається слово в "віконечку" із заданим зразком. Порівнювати по буквах довго. Замість цього фіксуємо деяку числову функцію на словах довжиною *n*, тоді завдання зведеться до порівняння чисел, що, безсумнівно, швидше. Якщо значення цієї функції на слові в "віконечку" і на зразку різні, то збігу немає. Тільки якщо значення однакові, необхідно перевіряти послідовно збіг по буквах.».

Замість того, щоб використовувати розумний пропуск символів при перевірці, алгоритм Рабіна-Карпа намагається прискорити перевірку еквівалентності зразка з підрядками в тексті використовуючи хеш-функцію. *Хеш-функція* – це функція, що перетворює кожний рядок у числове значення, що називається *хеш-значенням*; наприклад, ми можемо мати hash("hello") = 5. Алгоритм використовує той факт, що якщо два рядки однакові, то і їх хеш-значення також однакові. Таким чином, все що нам потрібно, це порахувати хеш-значення того підрядка, який ми шукаємо й потім знайти підрядок з таким же хеш-значенням.

Однак, існують дві проблеми пов'язані із цим.

Перша, тому що існує дуже багато різних рядків, для того, щоб мати невеликі хеш-значення, ми повинні мати деякі рядки, хеш-значення яких збігаються. Це означає, що незважаючи на те, що хеш-значення збігаються, рядки можуть не збігатися; нам необхідно перевіряти чи це дійсно так, що займає досить багато часу для довгих підрядків. На щастя, гарна хеш-функція забезпечує нам те, що при досить гарних вхідних значеннях це не буде відбуватися дуже часто, і в результаті середній час пошуку буде невеликим.

Псевдокод алгоритму Рабіна-Карпа:

```
 1 function RabinKarp(string s[1..n], string
sub[1..m])
 2     hsub = hash(sub[1..m])
 3     hs = hash(s[1..m])
 4     for i from 1 to ( n − m + 1)
 5         if hs == hsub
 6             if s[i..(i + m − 1)] = sub
 7                 return i
 8         hs = hash(s[(i + 1)..(i + m)])
 9     return not found
```

Рядки 2, 3, і 6 кожний, вимагають часу $O(m)$. Однак, рядки 2 і 3 виконуються тільки один раз, а рядок 6 виконується тільки у випадку, коли хеш-значення збігаються, що не може відбутися частіше, ніж кілька разів. Рядок 5 виконується n разів, але завжди вимагає постійного часу. Тепер розглянемо **другу** проблему: рядок 8.

Якщо ми наївно перераховуємо хеш-значення для підрядка $s[(i + 1) .. (i + m)]$, це буде вимагати час $O(m)$, і тому що це робиться в кожному циклі, алгоритм буде вимагати час $O(m \cdot n)$, тобто такий же, як і в найбільш простих алгоритмах. Прийом для рішення цього завдання полягає в тому, що змінна hs уже містить хеш-значення для $s[i .. (i + m - 1)]$. Якщо ми зможемо використовувати його для підрахунку наступного хеш-значення за постійний час, тоді наше завдання буде вирішене.

Ми будемо робити це використовуючи так званий кільцевий хеш. *Кільцевий хеш* – це хеш-функція, що використовується спеціально для цієї операції. Найпростішим прикладом кільцевого хешу є додавання значень кожного наступного символу в підрядку. Потім, ми можемо використовувати цю формулу для підрахунку кожного наступного хеш-значення за фіксований час:

$$s[(i + 1) .. (i + m)] = s[i..(i + m - 1)] - s[i] + s[i + m]$$

Ця проста функція працює, але в результаті вираз в 6 рядку буде виконуватися частіше, ніж інші більш розумні кільцеві хеш-функції, як ті, що будуть обговорені далі.

Відмітимо, що якщо ми відносимось до розряду невдах, або маємо дуже погану хеш-функцію, таку як постійну функцію, рядок 6 дуже ймовірно буде виконуватися n разів, на кожну ітерацію циклу. Тому що вона вимагає часу $O(m)$, алгоритм повністю буде вимагати час $O(m \cdot n)$.

Використання хеш-функції

Ключем до продуктивності алгоритму Рабіна-Карпа є ефективне обчислення хеш-значення послідовних підрядків тексту. Одна популярна й ефективна кільцева хеш-функція інтерпретує кожний підрядок, як число в деякій системі числення, основа якої є великим простим числом. Наприклад, якщо підрядок "hi" і основа системи числення 101, хеш-значення буде:

$104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCII код 'h' – 104 і 'i' – 105)

Технічно, цей алгоритм тільки подібний до справжнього числа в недесятковій системі запису, тому що для прикладу ми взяли "основу" меншу, ніж одну з його "цифр". Істотна користь досягається таким записом, що можливий для розрахунку хеш-значення наступного підрядку зі значення попереднього шляхом виконання тільки постійного набору операцій, незалежно від довжин підрядка.

Наприклад, якщо ми маємо текст "abracadabra" і шукаємо зразок довжиною 3, ми можемо розрахувати хеш підрядка "bra" з хешу підрядка "abr" (попередній підрядок), віднімаючи число додане для першої букви 'a' з "abr", тобто 97×101^2 (97 – ASCII для 'a' і 101 – основа, що ми використовуємо), множачи на основу й нарешті додаючи останнє число для "bra", тобто $97 \times 101^0 = 97$. Якщо підрядки в запиті довгі, цей алгоритм досягає великої економії порівняно з багатьма

іншими схемами хешування.

Теоретично, існують інші алгоритми, які можуть забезпечити порівнянний обсяг обчислень, наприклад, при множенні ASCII-значень всіх символів, у результаті зсув підрядка буде викликати тільки ділення на перший символ і множення на останній. Обмеженням, однак, є розмір типу даних цілого числа й необхідність використовувати модульну арифметику для зменшення результатів хешування; проте, ці прості хеш-функції, які швидко не роблять великі числа, такі як просте додавання ASCII-кодів, найбільш ймовірно генерують велику кількість хеш-колізій і отже – сповільнюють алгоритм. Із цього випливає, що описана хеш-функція є кращою для алгоритму.

Алгоритм Рабіна-Карпа виконує лінійний прохід по рядку (n кроків) і лінійний прохід по всьому тексту (m кроків), тому, загальний час роботи є $O(n + m)$. При цьому ми не враховуємо часову складність обчислення хеш-функції, тому що, суть алгоритму в тому і полягає, щоб дана функція настільки легко обчислювалась, щоб її робота не впливала на загальну роботу алгоритму. Тоді, час роботи алгоритму лінійно залежить від розміру рядка й тексту, відповідно програма працює швидко. Адже замість того, щоб перевіряти кожну позицію на предмет відповідності зі зразком, ми можемо перевіряти тільки ті, які «нагадують» зразок.

Алгоритм Рабіна-Карпа й алгоритм прямого пошуку є алгоритмами з найменшими трудовитратами, тому вони підходять для використання при рішенні деякого класу завдань. Однак ці алгоритми не є найбільш оптимальними (хоча б тому, що іноді виконують явно марну роботу, про що було сказано вище), тому розглянемо наступний клас алгоритмів. Ці алгоритми з'явилися в результаті ретельного дослідження алгоритму прямого пошуку. Дослідники хотіли знайти способи більш повно використовувати інформацію,

отриману під час сканування (алгоритм прямого пошуку її просто відкидає). Розглянемо алгоритм Кнута-Морріса-Пратта.

Алгоритм Кнута-Морріса-Пратта

Алгоритм був запропонований Дж.Моррісом і В.Праттом, Д.Кнут трохи вдосконалив його. Алгоритм дотепний, але якщо мова йде про звичайний текст, значного виграшу він не дасть – у цьому випадку оцінка числа порівнянь дорівнює $K \cdot n$, де K – невелике число. А от у найгіршому разі виграш буде значний.

Ідея алгоритму заснована на тому, що якщо знайдено частковий збіг, то в деяких випадках можливий зсув зразка (підрядка) більше, ніж на одну позицію й у всіх випадках не потрібно повторно порівнювати символи, що збіглися.

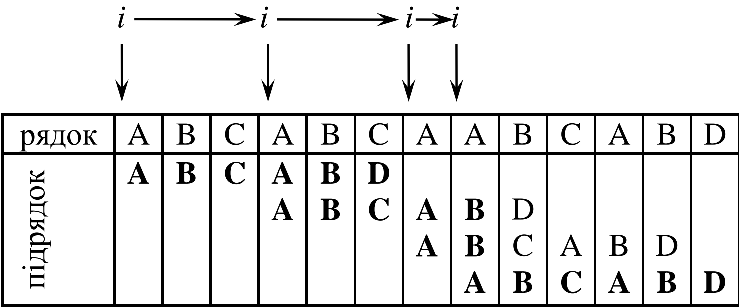


Рисунок 9.3 – Ілюстрація роботи алгоритму Кнута-Морріса-Пратта

Введемо ряд понять. Позначимо рядок як $X = x[1]x[2]...x[n]$, де n – довжина рядка, $x[i]$ – і-й символ рядка X, що належить деякому алфавіту. Тоді рядок X називається **підрядком** рядка Y, якщо знайдуться такі рядки Z_1 і Z_2, що $Y = Z_1XZ_2$, причому Z_1 та/або Z_2 можуть буди порожніми.

Підрядок X називається **префіксом** рядка Y, якщо є такий підрядок Z, що $Y = XZ$. Наприклад, підрядок ab є префіксом рядка *abcfa*.

Підрядок X називається **суфіксом** рядка Y, якщо є такий

підрядок Z, що $Y = ZX$. Наприклад, підрядок bfg є суфіксом рядка vsenfbfg.

Алгоритм КМП можна реалізувати наступним чином:

Метод використовує попередню обробку шуканого підрядка, а саме: на його основі створюється так звана *префікс-функція*. Суть цієї функції в знаходженні для кожного підрядка $S[1..i]$ рядка S найбільшого підрядка $S[1..j]$ ($j < i$), що є присутнім одночасно й на початку, і наприкінці підрядка (як префікс і як суфікс). Наприклад, для підрядка abcHelloabc підрядок abc одночасно є і префіксом, і суфіксом. Зміст префікс-функції в тому, що ми можемо відкинути свідомо невірні варіанти.

Префікс-функція від рядка (позначається $\pi(S, i)$; $S \in \Sigma^+$; $2 \le i \le |S|$) – довжина найбільшого префіксу рядка $S[1…i]$, що не збігається із цим рядком і одночасно є його суфіксом.

Часто префікс-функцію записують у вигляді вектору довжиною $|S|$ - 1. Наприклад, для рядка 'abcdabscabcdabia' префікс-функція буде такою:
π(abcdabscabcdabia)='000012001234560 1'. Іноді для повноти вважають, що $\pi(S, 1) = 0$.

Алгоритм обчислення префікс-функції

Символи рядків нумеруються з 1.

Нехай $\pi(S, i) = k$. Спробуємо обчислити префікс-функцію для $i + 1$.

Якщо $S[i + 1] = S[k + 1]$, то, природно, $\pi(S, i + 1) = k + 1$. Якщо ні – підбираємо менші суфікси. Очевидно, що $S[1…\pi(S,k)]$ також буде суфіксом рядка $S[1…i]$, а для кожного $j \in (k, i)$ рядок $S[1…j]$ суфіксом не буде. Таким чином, маємо алгоритм:

1. При $S[i + 1] = S[k + 1]$ –> $\pi(S, i + 1) = k + 1$.
2. Інакше при $k = 0$ –> $\pi(S, i + 1) = 0$.
3. Інакше – встановити $k: = \pi(S, k)$, GOTO 1.

Для рядка 'abcdabscabcdabia' обчислення буде таким:

'a'!='b' => π=0;
'a'!='c' => π=0;
'a'!='d' => π=0;
'a'=='a' => π=π+1=1;
'b'=='b' => π=π+1=2;
'c'!='s' => π=0;
'a'!='c' => π=0;
'a'=='a' => π=π+1=1;
'b'=='b' => π=π+1=2;
'c'=='c' => π=π+1=3;
'd'=='d' => π=π+1=4;
'a'=='a' => π=π+1=5;
'b'=='b' => π=π+1=6;
's'!='i' => π=0;
'a'=='a' => π=π+1=1;

Приклад 9.3. Псевдокод рефікс-функції

```
Procedure PrefFunc(str P, array Fl)
 begin
    var int n, int i, int j

    n = Length(P)
    Fl[1] = 0
    for i = 2 to n do
        begin
            j = Fl[i - 1]
            while (j != 0) And (P[j] != P[ i - 1]) do
                j = Fl[j]
            Fl[i] = j + 1
    end
 end
```

Таким чином, ми перевіряємо префікс попереднього підрядка, якщо ж той не підходить, то префікс його префіксу, і так далі. Діючи так, знаходимо найбільший шуканий префікс.

Наступне питання, на яке варто відповісти: чому час роботи процедури лінійний, адже в ній є вкладений цикл? Ну, по-перше, присвоєння префікс-функції відбувається чітко m разів, інший час змінюється змінна *k*. Тому що в циклі while вона зменшується ($P[k] < k$), але не стає менше 0, то зменшуватися вона може не частіше, ніж зростати. Змінна *k* зростає на 1 не більше *m* разів. Виходить, змінна *k* міняється всього не більше 2*m* разів. Тому час роботи всієї процедури є $O(m)$.

Псевдокод алгоритму Кнута-Морріса-Пратта

```
function int KMPSearch(str S, str P)
// Алгоритм Кнута-Морріса-Пратта, що встановлює
// входження непорожнього підрядка P в рядок S
 begin
    var array Fl; int i, int j, int n, int m
    n = Length(S)
    m = Length(P)
    PrefFunc(P,Fl)
    j = 1

    for i = 1 to n do
        begin
            while (j != 0) And (P[j] != S[i]) do
                j = Fl[j]
            if j == m then
                break
            j = j + 1
        end
    if (j == m) then Result = i – j + 1
    else Result = 0
 end
```

Довести, що ця програма працює за лінійний час, можна точно так само, як і для префікс-функції. Тому, загальний час роботи програми є $O(n + m)$, тобто лінійний час.

На останок відмітимо, що алгоритм послідовного пошуку й алгоритм КМП крім знаходження самих рядків враховують, скільки символів збіглося в процесі роботи.

Алгоритм Бойєра-Мура

Алгоритм Бойєра-Мура, розроблений двома вченими – Бойєром і Муром, вважається найбільш швидким серед алгоритмів загального призначення, призначених для пошуку підрядків в рядку.

Найпростіший варіант алгоритму Бойєра-Мура складається з наступних кроків. На першому кроці ми будуємо таблицю зсувів для шуканого зразка. Процес побудови таблиці буде описаний нижче. Далі ми сполучаємо початок рядка й зразка і починаємо перевірку з останнього символу зразка. Якщо останній символ зразка й відповідний йому при накладенні символ рядка не збігаються, зразок зсувається щодо рядка на величину, отриману з таблиці зсувів, і знову здійснюється порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, здійснюється порівняння передостаннього символу зразка й так далі. Якщо всі символи зразка збіглися з накладеними символами рядка, то ми знайшли підрядок й пошук закінчений. Якщо ж якийсь (не останній) символ зразка не збігається з відповідним символом рядка, ми зсуваємо зразок на один символ вправо й знову починаємо перевірку з останнього символу. Весь алгоритм виконується доти, поки або не буде знайдене входження шуканого зразка, або не буде кінець рядка.

Величина зсуву у випадку розбіжності останнього символу обчислюється виходячи з наступних міркувань: зсув зразка повинен бути мінімальним, таким, щоб не пропустити входження зразка в рядку. Якщо даний символ рядка зустрічається в зразку, ми зсуваємо зразок таким чином, щоб символ рядка збігся із самим правим входженням цього символу в зразку. Якщо ж зразок взагалі не містить цього символу, ми зсуваємо зразок на величину, рівну його довжині, так що перший символ зразка накладається на наступний за перевіреним символом рядка.

Величина зсуву для кожного символу зразка залежить тільки від порядку символів у зразку, тому зсуви зручно обчислити заздалегідь і зберігати у вигляді одномірного масиву, де кожному символу алфавіту відповідає зсув щодо останнього символу зразка. Пояснимо все вищесказане на простому прикладі. Нехай у нас є алфавіт з п'яти символів: *a*, *b*, *c*, *d*, *e* і ми бажаємо знайти входження зразка “abbad” у рядку “abeccacbadbabbad”. Наступні схеми ілюструють всі етапи виконання алгоритму. Таблиця зсувів буде виглядати так.

a	b	c	d	e
1	2	5	0	5

Рисунок 9.4 – Таблиця зсувів

Початок пошуку.

a	b	e	c	~~c~~	a	c	b	a	d	b	a	b	b	a	d
a	b	b	a	~~d~~											

Рисунок 9.5 – 1 порівняння

Останній символ зразка не збігається з накладеним символом рядка. Зсуваємо зразок вправо на 5 позицій:

a	b	e	c	c	a	~~c~~	**b**	**a**	**d**	b	a	b	b	a	d
					a	~~b~~	**b**	**a**	**d**						

Рисунок 9.6 – 2 порівняння

Три символи зразка збіглися, а четвертий – ні. Зсуваємо зразок вправо на одну позицію:

a	b	e	c	c	a	c	b	a	d	~~**b**~~	a	b	b	a	d
						a	b	b	a	~~**d**~~					

Рисунок 9.7 – 3 порівняння

Останній символ знову не збігається із символом рядка. Відповідно до таблиці зсувів зсуваємо зразок на 2 позиції:

a	b	e	c	c	a	c	b	a	d	b	a	~~**b**~~	b	a	d
								a	b	b	a	~~**d**~~			

Рисунок 9.8 – 4 порівняння

Ще раз зсуваємо зразок на 2 позиції:

a	b	e	c	c	a	c	b	a	d	b	a	b	b	~~**a**~~	d
										a	b	b	a	~~**d**~~	

Рисунок 9.9 – 5 порівняння

Тепер, відповідно до таблиці, зсуваємо зразок на одну позицію, і одержуємо шукане входження зразка:

a	b	e	c	c	a	c	b	a	d	b	**a**	**b**	**b**	**a**	**d**
											a	**b**	**b**	**a**	**d**

Рисунок 9.10 – 6 порівняння

Проілюструємо зазначений алгоритм на псевдокоді.

Насамперед варто визначити тип даних «таблиця зсувів». Для кодової таблиці, що складається з 256 символів, визначення цього типу буде виглядати так: Type TBMTable = Array [0..255] of Integer.

Далі наводиться псевдокод процедури, що обчислює таблицю зсувів для зразка *p*.

```
Procedure MakeMBTable( TBMTable Bmt, Const str p)
 begin
    var  int i
    for i = 0 to 255 do
        Bmt[i] = Length(p)
    for i = Length(p) downto 1 do
        if  Bmt[Byte(p[i])] == Length(p) then
            Bmt[Byte(p[i])] = Length(p) – i
 end
```

Тепер напишемо функцію, що здійснює пошук.

```
function int bmsearch(int startpos; const str s,
const str p, const tbmtable bmt)
 begin
    var int pos, int lp, int i
    lp = length(p)
    pos = startpos + lp – 1
    while pos < length(s) do
        if p[lp] != s[pos] then
            pos = pos + bmt[s[pos]]
        else for i = lp - 1 downto 1 do
            if p[i] != s[pos – lp + i] then
                begin
                    inc(pos)
                    break
                end
        else if i == 1 then
            begin
                result = pos – lp + 1
                exit
        end
     result = 0
 end
```

Параметр StartPos дозволяє вказати позицію в рядку *s*, з якої варто починати пошук. Це може бути корисно в тому випадку, якщо необхідно знайти всі входження *p* в *s*. Для

пошуку із самого початку рядка варто задати StartPos рівним 1. Якщо результат пошуку не дорівнює нулю, то для того, щоб знайти наступне входження p в s, потрібно задати StartPos рівним значенню «попередній результат плюс довжина зразка».

Модифікації Бойєра-Мура

Швидкий пошук

Зсув поганого символу, що використовується в алгоритмі Бойєра-Мура, не дуже ефективний для маленького алфавіту, але, коли розмір алфавіту великий у порівнянні з довжиною зразка, як це часто має місце з таблицею ASCII і при звичайному пошуку в текстовому редакторі, він стає надзвичайно корисний. Використання в алгоритмі тільки його одного може бути досить ефективним.

Після спроби сполучення x і $y[i, i + m - 1]$, довжина зсуву – не менше 1. Таким чином, символ $y[i + m]$ обов'язково буде залучений у наступну спробу, а виходить, може бути використаний у поточній спробі для зсуву поганого символу. Модифікуємо функцію поганого символу, щоб взяти до уваги останній символ x:

$bc[a] = \min \{ j \mid 0 \leq j \leq m$ і $x[m - 1 - j] = a \}$, якщо a зустрічається в x,

$bc[a] = m$ у протилежному випадку.

Порівняння тексту й зразка можуть здійснюватися в будь-якому порядку.

Турбо алгоритм Бойєра-Мура

Турбо БМ також є поліпшенням алгоритму Бойєра-Мура. Ми будемо запам'ятовувати сегмент тексту, що зійшовся із суфіксом зразка під час минулої спроби (і тільки, якщо відбувся зсув гарного суфікса).

Це дасть нам дві переваги:

1. Можливість перескочити через цей сегмент.

2. Можливість застосування «турбо-зсуву».

«Турбо-зсув» може відбутися, якщо ми виявимо, що суфікс зразка, що сходиться з текстом, коротший, ніж той, який був запам’ятований раніше.

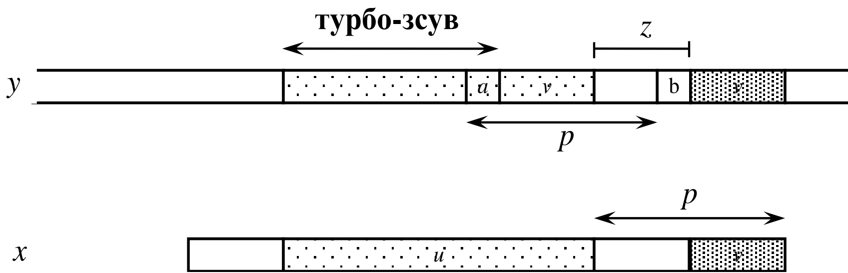


Рисунок 9.11 – Турбо алгоритм Бойєра-Мура

Нехай u – запам’ятований сегмент, а v – суфікс, що збігся під час поточної спроби, такий що uzv – суфікс x. Тоді av – суфікс x, два символи a й b зустрічаються на відстані p у тексті, і суфікс x довжини |uzv| має період довжини p, а значить не може перекрити обидві появи символів a й b у тексті. Найменший можливий зсув має довжину |u| - |v| (його ми й називаємо «турбо-зсувом»).

Питання до розділу:

1. Де застосовують пошук точно заданого підрядка у рядку?
2. Поясніть алгоритм прямого послідовного пошуку. В чому його складність і які він має недоліки?
3. Де застосовують алгоритм Рабіна-Карпа?
4. Поясніть ідею, що лежить в основі алгоритму Рабіна-Карпа?
5. Що таке хеш-функція?
6. Яким чином можна отримати хеш-значення?

7. В чому полягає ідея алгоритму Кнута-Морріса-Пратта?
8. Що таке префікс-функція?
9. Поясніть алгоритм обчислення префікс-функції.
10. За рахунок чого алгоритм Бойєра-Мура є найбільш швидким?

Розділ 10. Алгоритми роботи з хеш-таблицями

Хеш-таблиця – це звичайний масив з незвичайною адресацією, що задається хеш-функцією.

Хеш-функція – функція, що трансформує ключ у деякий індекс у таблиці.

Ситуація, коли два або більше ключів асоціюються з однією й тією ж коміркою називається *колізією* при хешуванні.

Слід зазначити, що гарною хеш-функцією є така функція, що мінімізує колізії й розподіляє записи рівномірно по всій таблиці.

Хеш-таблиці часто застосовуються в базах даних, і, особливо, у мовних процесорах типу компіляторів і асемблерів, де вони вдало обслуговують таблиці ідентифікаторів. У таких додатках, таблиця – найкраща структура даних.

Метод відкритого хешування (інша назва: хешування ланцюжками)

У випадку хешування ланцюжками, елементи з однаковими індексами об'єднуються в зв'язаний список. Наступний рисунок ілюструє це.

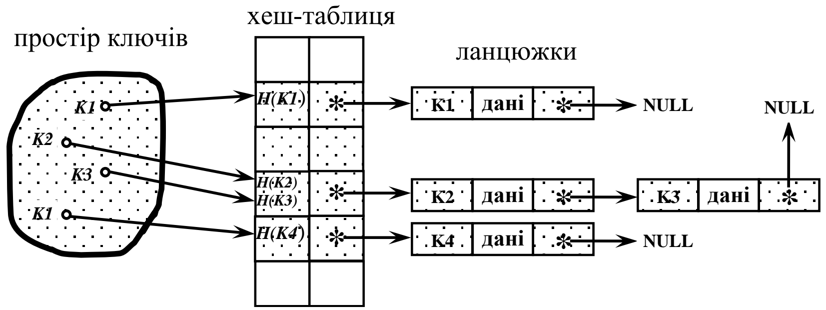


Рисунок 10.1 – Хешування ланцюжками

Тобто, якщо при додаванні в хеш-таблицю в задану комірку ми зустрічаємо посилання на елемент зв'язаного списку, то трапляється колізія. Тоді, ми просто вставляємо наш елемент як вузол у список. При пошуку ми проходимо по ланцюжках, порівнюючи ключі між собою на еквівалентність, поки не доберемося до потрібного. При видаленні ситуація така ж.

Видалення вузла з таблиці, що побудована за методом ланцюжків, полягає просто у виключенні вузла зі зв'язаного списку. Вилучений вузол ніяк не впливає на ефективність алгоритму пошуку. Алгоритм буде працювати так, ніби цей вузол ніколи не вставлявся в таблицю.

Псевдокод алгоритму відкритого хешування:

```
Class node
 begin
    int key
    str st
    link next // посилання на наступний елемент
 end

var link of array mas[0..9]

function int h(int key)
 begin
    h = key mod 10 // mod - залишок від ділення
 end

function link search(int key1, str st1)
begin
    var int i
    link q, link p, link s
    i = h(key1)
    q = nil
    p = mas[i]
    while p != nil do
    begin
        if p.key == key1 then
```

```
            begin
                search = p;
                exit
            end
            q = p
            p = p.link
        end

//Якщо ключ не знайдений, вставляємо новий запис
        new(s)
        s.key = key1
        s.st = st1
        s.next = nil
        if q = nil then
            mas[i] = s
        else
            q.next = s
    search = s
end
```

Метод закритого хешування (інша назва: відкритої адресації)

Закриті хеш-таблиці особливо ефективні, коли максимальні розміри вхідного набору даних вже відомі.

У випадку методу закритого хешування всі елементи зберігаються безпосередньо в хеш-таблиці, без використання зв'язаних списків. На відміну від хешування з ланцюжками, при використанні методу відкритої адресації може виникнути ситуація, коли хеш-таблиця виявиться повністю заповненою, так що буде неможливо додавати в неї нові елементи. Так що при виникненні такої ситуації рішенням може бути динамічне збільшення розміру хеш-таблиці, з одночасною її перебудовою.

Для розв’язання колізій застосовуються кілька підходів. Найпростіший з них – це метод *лінійного дослідження*. У цьому випадку при виникненні колізії наступні за поточною комірки перевіряються одна за одною, поки не знайдеться порожня

комірка, куди й записується елемент. А при досягненні останнього індексу таблиці, відбувається перехід на початок, тобто таблиця розглядається як «циклічний» масив. Ілюстрація цього способу представлена на наступному рисунку:

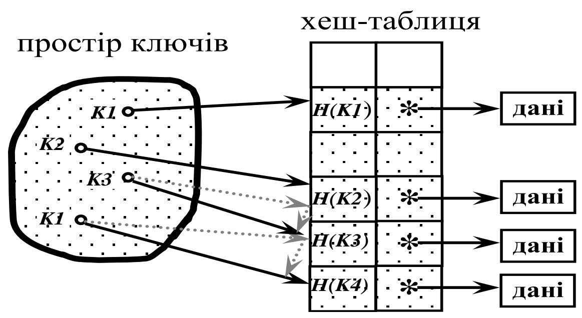


Рисунок 10.2 – Закрите хешування

Лінійне хешування досить просто реалізується, однак з ним зв'язана істотна проблема – *кластеризація*. Це явище створення довгих послідовностей зайнятих комірок, що збільшує середній час пошуку в таблиці. Для зниження ефекту кластеризації використовується інша стратегія вирішення колізій – подвійне хешування. Основна ідея полягає в тому, що для визначення кроку зсуву досліджень при колізії в комірці використовується інша хеш-функція, замість лінійного зсуву на одну позицію.

Розглянемо детальніше реалізацію даного методу. Введемо два масиви:

```
array of T val[0.. n-1]
array of boolean used[0.. n-1]
```

У цих масивах будуть зберігатися елементи множини: вона дорівнює множині всіх val[*i*] для тих і, для яких used[*i*], причому всі ці val[*i*] різні. По можливості будемо зберігати

елемент *t* на місці *h*(*t*), будемо називати це місце «початковим» для даного елементу. Однак може трапитися так, що новий елемент, який ми хочемо додати, претендує на вже зайняте місце (для якого used істинне). У цьому випадку ми відшукаємо найближче праворуч вільне місце й запишемо елемент туди. ("Праворуч" значить "в бік збільшення індексів"; дійшовши до краю, ми перестрибуємо на початок.) За припущенням, число елементів завжди менше n, так що порожні місця гарантовано будуть.

Формально говорячи, у будь-який момент повинна дотримуватися така вимога: для будь-якого елементу множини ділянка праворуч від його «початкового» місця до його фактичного місця повністю заповнена.

Завдяки цьому перевірка приналежності заданого елементу *t* здійснюється легко: вставши на *h*(*t*), рухаємося праворуч, поки не дійдемо до порожнього місця або до елементу *t*. У першому випадку елемент *t* відсутній у множині, у другому є присутнім. Якщо елемент відсутній, то його можна додати на знайдене порожнє місце. Якщо є присутнім, то можна його видалити (привласнивши used = false).

Одним зі складних питань реалізації хешування з відкритою адресацією є операція видалення елементу. Справа в тому, що при видаленні елементу необхідна властивість "відсутності порожнеч" може порушитися. Тому будемо робити так. Створивши пусте місце, будемо рухатися праворуч, поки не натрапимо на ще одне порожнє місце (тоді на цьому можна заспокоїтися) або на елемент, що стоїть не на «початковому» місці. У другому випадку подивимося, чи не потрібно цей елемент поставити на порожнє місце. Якщо ні, то продовжуємо пошук, якщо так, то закриваємо ним порожнє місце, що з’явилося після операції видалення. При цьому утвориться нова діра, з якою робимо все те ж саме.

Псевдокод алгоритму закритого хешування:

```
function int h(int key)
 begin
    var array of int K[0..999]
    h = key mod 1000 // mod - залишок від ділення
 end

function int rh(int i)
 begin
    rh = i + 1 mod 1000 // mod - залишок від ділення
 end

procedure insert(int key)
 begin
    var int i
    i = h(key) //хешуємо ключ
    while ((k(i) != key) and (k(i) != 0)) do
        i = rh(i) //повторне хешування
    if k(i) == 0 then //вставляємо запис у порожню
позицію
        k(i) = key
 end
```

Вибір хеш-функції

Звернемося тепер до питання про те, як вибрати гарну хеш-функцію. Ясно, що ця функція повинна створювати якнайменше колізій при хешуванні, тобто вона повинна рівномірно розподіляти ключі на наявні індекси в масиві. Звичайно, не можна визначити, чи буде деяка конкретна хеш-функція розподіляти ключі правильно, якщо ці ключі заздалегідь не відомі. Однак, хоча до вибору хеш-функції рідко відомі самі ключі, деякі властивості цих ключів, які впливають на їхній розподіл, звичайно відомі.

1) метод ділення. Деякий цілий ключ ділиться на розмір таблиці й залишок від ділення береться як значення хеш-функції. Ця хеш-функція позначається $h(key) = key \bmod m$.

2) метод середини квадрата. Ключ множиться сам на себе і як індекс використовується декілька середніх цифр цього квадрата. Псевдокод:

```
function int h(int key)
 begin
    key = key * key //Піднести до квадрата
    key = key shl 11 //Відкинути 11 молодших бітів
    H = key mod 1024 //Повернути 10 молодших бітів
 end
```

3) адитивний метод для рядків (розмір таблиці дорівнює 256). Для рядків цілком розумні результати дає додавання всіх символів і повернення залишку від ділення на 256. Псевдокод:

```
function h(st: string): integer
 begin
    var sum: longint
    i: integer
    for i =0 to length(st) do
        sum = sum + ord(st[i])
    H = sum mod 256;
 end
```

4) виключаюче АБО для рядків (розмір таблиці дорівнює 256). Цей метод аналогічний адитивному, але успішно розрізняє схожі слова й анаграми (адитивний метод дасть одне значення для *XY* і *YX*). Метод полягає в тому, що до елементів рядка послідовно застосовується операція "виключаюче або". В алгоритм додається випадковий компонент, щоб ще поліпшити результат. Псевдокод:

```
var
 array of int rand8[0..255]
procedure init
 begin
```

```
    var
     int i
    randomize()
    for i = 0 to 255 do
        rand8[i] = random(255)
 end

function int h(str st)
 begin
    var
     longint sum
     int i
    for i = 0 to length(st) do
        sum = sum + ord(st[i]) xor rand8[i]
    h = sum mod 256 // mod - залишок від ділення
 end
```

Питання до розділу:

1. Як хеш-таблиці пов’язані з хеш-функціями?
2. Що таке колізія?
3. Де використовуються хеш-таблиці?
4. Поясніть метод відкритого хешування.
5. Поясніть метод закритого хешування.
6. Як обрати хеш-функцію?

Розділ 11. Двійкові дерева. Алгоритми обходів дерев

Дерево – це структура даних у програмуванні, що представляє собою сукупність вузлів (що називаються ***вершинами*** дерева), з'єднаних між собою ребрами (***гілками***). Дерево є одною зі структур даних, що найбільш часто застосовуються у програмуванні для вирішення широкого кола задач.

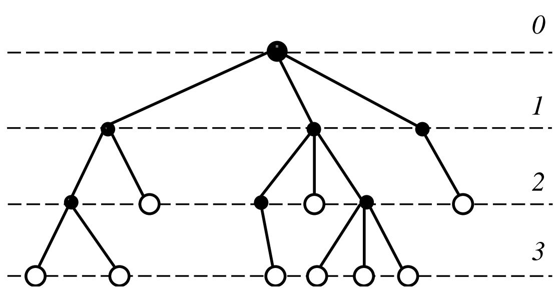


Рисунок 11.1 – Приклад дерева

Кількість рівнів називається ***глибиною*** дерева.

Кожна вершина нижнього рівня з'єднується рівно з однією вершиною попереднього рівня.

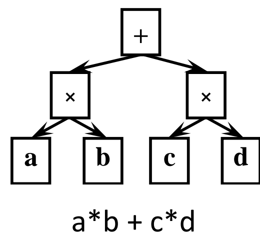
Єдина вершина на рівні 0 називається **коренем** дерева.

Вона не має вершин-предків.

Вершини, що не мають нащадків, називають **листками** дерева, а сукупність всіх листів утворить **крону** дерева.

Приклади структури даних дерево:

- Дерево папок на диску.
- Дерево рішень в інтелектуальних системах.
- Рекурсивне дерево викликів.
- Дерево двійкового пошуку.
- Суфіксні дерева для роботи з рядками.
- Дерево розбору виразів (рис. 11.2).



a*b + c*d

Рисунок 11.2 – Дерево розбору виразів

Тепер дамо *рекурсивне* визначення **дерева**:

Дерево ::= корінь список_піддерев
| ε

Список піддерев ::= список_піддерев дерево
| ε

// ε означає «порожньо»

Визначення

Дерево називається ***бінарним*** (***двійковим***), якщо кожна його вершина має не більше двох нащадків.

(Далі бінарні дерева будемо скорочувати як БД).

Двійкове дерево ::= корінь ліве піддерево праве піддерево
| ε

Ліве піддерево ::= двійкове дерево

Праве піддерево ::= двійкове дерево

БД називається ***ідеально збалансованим***, якщо для кожного вузла кількість вузлів у його правому піддереві відрізняється від кількості вузлів у його лівому піддереві максимум на одиницю.

Повним називають БД, у якого кожна вершина, що не є листом, має рівно двох нащадків, і всі листи перебувають на останньому рівні.

Кількість вузлів (*u*) і кількість ребер (*v*) у довільному дереві зв'язані простою формулою: ***u* = *v* + 1**.

Кількість вузлів у *повному БД* обчислюється за формулою $K = 2^{n+1} - 1$, де *n* – глибина дерева.

Алгоритми обходів дерев

Потрібно скласти алгоритм, що обходить всі вузли дерева в деякому порядку.

Існує декілька варіантів обходів, кожний з яких описується рекурсивним алгоритмом.

Розглянемо ці алгоритми для *звичайного* й *бінарного* дерев:

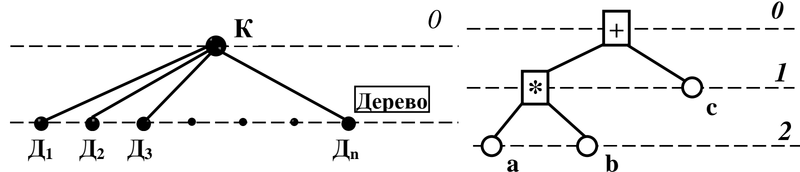


Рисунок 11.3 – Алгоритми обходу дерев

1. Інфіксний обхід (симетричний)

- для будь-якого дерева:

$Д_1$ К $Д_2$ $Д_3$ $Д_n$

- для двійкового дерева:

Ліве_піддерево Корінь Праве_піддерево

2. Префіксний обхід (прямий)

- для будь-якого дерева:

К $Д_1$ $Д_2$ $Д_3$ ... $Д_n$

- для двійкового дерева:

Корінь Ліве_піддерево Праве_піддерево

+ * a b c – **префіксний** запис виразу

Примітка. Для обчислення виразів, записаних у *префіксній* формі, застосовується *рекурсивний* алгоритм.

3. Постфіксний обхід (зворотній)

- для будь-якого дерева:

$Д_1$ $Д_2$ $Д_3$ ... $Д_n$ К

- для двійкового дерева:

Ліве_піддерево Праве_піддерево Корінь

a b * c + – зворотний польський бездужковий запис виразу.

Зворотний польський запис (постфіксна нотація, бездужковий запис) – форма запису математичних виразів, в якій операнди розташовані перед знаками операцій.

Псевдокод реалізації класу бінарного дерева:

```
class TreeNodeT
 begin
    str data
    left, right: TreeNodeT

    constructor (str d, TreeNodeT l, TreeNodeT r)
     begin
        data = d
        left = l
        right = r
     end
 end
```

Псевдокод створення ідеально-збалансованого бінарного дерева:

```
function TreeNode CreateTree(int n)
 begin
    if n <= 0 then
        Result = nil
    else
        Result = new TreeNode(Random(100),
                 CreateTree((n - 1) div 2),
                 CreateTree(n - 1 - ( n - 1) div 2)
 end
```

Розглянемо нижче псевдокоди обходів бінарних дерев.

Псевдокод алгоритму інфіксного обходу

```
procedure InfixPrintTree(TreeNode root)
 begin
     if root == nil then
          exit

     InfixPrintTree(root.left)
     write(root.data, ' ')
     InfixPrintTree(root.right)
 end
```

Відмітимо, що крім *виведення* root.data, над ним можна робити ще масу дій (наприклад, зменшувати на 1, або виводити його квадрат).

Тому необхідно передавати в процедуру обходу дію, яка виконується. Для цього визначимо процедурний тип і внесемо в процедуру відповідні зміни, псевдокод:

```
type
  IntAction = procedure(int data)

procedure InfixTraverseTree(TreeNode root, IntAction
Action)
 begin
     if root == nil then
         exit

     InfixTraverseTree(root.left, Action)
     Action(root.data)
     InfixTraverseTree(root.right, Action)
 end
```

Псевдокод алгоритму префіксного обходу

```
procedure PrefixTraverseTree(TreeNode root, IntAction
Action)
 begin
    if root == nil then
```

```
        exit

    Action(root.data)
    PrefixTraverseTree(root.left, Action)
    PrefixTraverseTree(root.right, Action)
 end
```

Псевдокод алгоритму постфіксного обхіду

```
procedure PostfixTraverseTree(TreeNode root,
IntAction Action)
 begin
     if root == nil then
         exit

     PostfixTraverseTree(root.left, Action)
     PostfixTraverseTree(root.right, Action)
     Action(root.data)
 end
```

Зауваження. Як видно, процедури відрізняються тільки моментом виклику Action. Тому можна уникнути дублювання коду, написавши процедуру, якій як параметр також передається порядок обходу, псевдокод:

```
type
  /// Порядок обходу
  TraversalOrder = (Infix, Prefix, Postfix)

procedure TraverseTree(TreeNode root, IntAction
Action, TraversalOrder order = Infix)
 begin
    if root == nil then
        exit

    if order == Prefix then //якщо префіксний обхід
        Action(root.data) //то спочатку обробляємо
корінь

    TraverseTree(root.left, Action, order)
```

```
    if order == Infix then //якщо інфіксний, то
корінь треба обробити після лівого піддерева, але
перед правим
        Action(root.data)

    TraverseTree(root.right, Action, order)

    if order == Postfix then   // і якщо порядок
постфіксний, то корінь обробляється останнім
        Action(root.data)
 end
```

Зв'язок дерев і рекурсії

Нехай у нас є таке бінарне дерево:

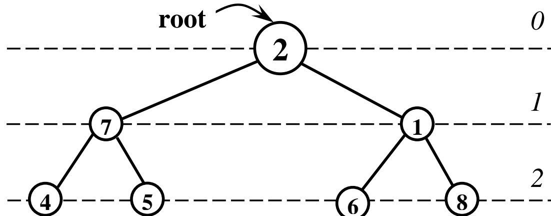


Рисунок 11.4 – Приклад бінарного дерева

Викликаємо процедуру InfixPrintTree(root).

Відмітимо, що її *дерево рекурсивних викликів* співпадає з нашим деревом:

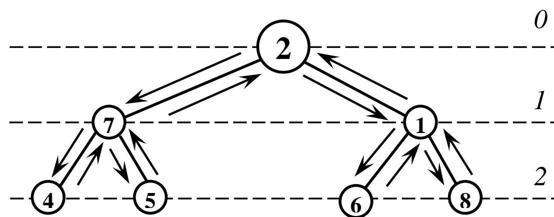


Рисунок 11.5 – Дерево рекурсивних викликів

З ним же співпадає й дерево рекурсивних викликів кожної із процедур: InfixPrintTree, PrefixPrintTree або PostfixPrintTree.

Зробимо кілька зауважень:

- Форма дерева рекурсивних викликів *не залежить від порядку обходу*.
- У кожний момент часу глибина рекурсії збігається з поточною глибиною дерева.
- Стрілки вниз відповідають рекурсивному спуску, а стрілки вгору – рекурсивному поверненню.
- Всі алгоритми на деревах найбільш компактно записуються в рекурсивній формі.

Псевдокод пошуку елементу в бінарному дереві:

```
function TreeNode Find(int root, int k)
 begin
    if root == nil then
        begin
            Result = nil
            exit
        end

    if root.data == k then  // якщо знайшли елемент,
то повертаємо його й припиняємо пошук
    begin
        Result = root
        exit
    end

    // шукаємо елемент у лівому піддереві
    Result = Find(root.left, k)
   // якщо не знайшли в лівому піддереві, то шукаємо
в правому
    if Result == nil then
        Result = Find(root.right, k)
 end
```

Примітка. Якщо в списку перебувають декілька шуканих елементів, то повертається посилання на перший з них, а саме – який знаходиться в «самому лівому» піддереві.

Приклади псевдокодів визначення мінімальної суми від кореня до листа

Рішення 1. Очевидний рекурсивний алгоритм. Псевдокод:

```
function int SimplePathSum(TreeNode root)
 begin
    if root == nil then
    Result = integer.MaxValue
    else if (root.Left == nil) and (root.Right ==
nil) then // виправлення
        result = root.Data
    else
        result = root.Data +
                 min(SimplePathSum(root.Left),
                 SimplePathSum(root.Right))
 end
```

Тут здійснюється повний обхід дерева (відвідування всіх вузлів).

Рішення 2. Алгоритм перебору з поверненням

Змінимо стратегію знаходження мінімальної суми. Будемо накопичувати суму в глобальній змінній sum і щораз по досягненні листа порівнювати її із глобальною змінною min.

Помітимо, що щоразу ми, додаючи до суми чергове значення, здійснюємо **перебір**, і якщо рішення неповне, то продовжуємо рекурсивні виклики. Після рекурсивних викликів ми здійснюємо **повернення** суми до попереднього значення. У результаті в кожній точці в змінній sum зберігається сума елементів від кореня до поточного вузла.

Подібний алгоритм називається **алгоритмом перебору з поверненнями**.

Псевдокод 2-го рішення:

```
var
 int sum = 0
 int min = MaxInt

procedure MinSumPath1(TreeNode r)
 begin
    if r == nil then
        exit
    sum += r.data
    if (r.left == nil) and (r.right == nil) and (sum
< min) then
        min = sum
    MinSumPath1(r.left)
    MinSumPath1(r.right)
    sum -= r.data
 end
```

Даний метод працює приблизно з тією ж швидкістю, що й попередній (тому що здійснюється повний перебір). Однак, на відміну від попереднього методу, він може бути істотно прискорений за рахунок відсікання свідомо неоптимальних рішень. У цьому випадку якщо значення sum перевищить min, то далі рекурсивні виклики можна не робити – рішення не оптимальне (мінімум на цьому шляху досягнутим не буде).

Змінимо даний алгоритм, здійснюючи вихід з рекурсії якщо сума на деякому кроці виявиться не менше min.

Рішення 3. Алгоритм перебору з поверненням. Метод гілок і границь

Метод гілок і границь є варіацією повного перебору, але робить відсівання підмножин припустимих рішень, що свідомо не містять оптимальних рішень. Псевдокод:

```
var
 int sum = 0
 int min = MaxInt
```

```
procedure MinSumPath1(TreeNode r)
 begin
    if r == nil then
        exit
    sum += r.data
    if (r.left == nil) and (r.right == nil) and (sum
< min) then
        min = sum
    if sum < min then
        begin
            MinSumPath1(r.left)
            MinSumPath1(r.right)
        end
    sum -= r.data
 end
```

Даний алгоритм працює практично миттєво.

Ось час роботи при n=20000000

- Побудова дерева: 6.5 с.
- Алгоритм перебору з поверненням: 0.828 с.
- Метод гілок і границь: 0.015 с.

Кількість рекурсивних викликів при різних запусках: 25000, 16000, 28000 (в алгоритмі 1 кількість рекурсивних викликів дорівнює n=20000000)

Бінарні дерева пошуку

Бінарне дерево називають **бінарним деревом пошуку** (**БДП**), якщо для кожного вузла дерева виконане наступне:

всі елементи, що перебувають у *лівому* піддереві, *менші* елементу в корені, а всі елементи, що перебувають у *правому* піддереві – *більші*.

Зауваження. При такому формулюванні визначення, БДП не має повторюваних елементів і називається ***бінарним деревом пошуку без повторюваних елементів***.

А якщо у визначенні замінити всі *строгі* нерівності на *нестрогі*, то БДП буде називатися ***бінарним деревом пошуку з повторюваними елементами***.

Приклад БДП:

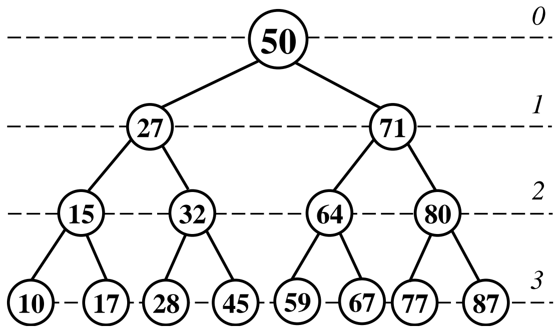


Рисунок 11.6 – Приклад бінарного дерева пошуку

Обійдемо наше БДП за допомогою *інфіксного* обходу:

10 15 17 27 28 32 45 50 59 64 67 71 77 80 89

Як бачимо, при *інфіксному обході* БДП одержуємо *відсортовану* послідовність даних.

Тобто, у кожний момент часу БДП зберігає відсортовані дані.

Яка ж *кількість дій* при обході БДП?

Оскільки при обході ми проходимо по кожному ребру двічі (один раз – вниз, на рекурсивному спуску, і один раз – вгору, на рекурсивному поверненні), то ми здійснюємо кількість дій, у два рази більшу, ніж кількість ребер.

Так як кількість ребер на 1 менше кількості вершин, то всього витрачається **2(*n* – 1)** дій, де *n* – кількість вершин (для порівняння: у масиві обхід займає ***n*** дій.

Алгоритм додавання елементу до БДП

Алгоритм додавання елементу до БДП *із* повторюваними елементами буде не набагато відрізнятися від алгоритму додавання до БДП *без* повторюваних елементів – у другому

випадку, якщо елемент, що ми хочемо додати, у дереві вже є, ми нічого не робимо.

Цей випадок і розглянемо. Псевдокод:

```
procedure Add(TreeNode root, int x)
 begin
    if root == nil then
        begin
            root = new TreeNode(x, nil, nil)
            exit
        end

    if x < root.data then
        Add(root.left, x)
    else if x > root.data then
        Add(root.right, x)
 end
```

Тепер створення дерева не є складним, псевдокод:

```
var n = 15
var TreeNode r = nil

for var i = 1 to n do   // створення випадкового БДП
із кількістю вершин, рівною n
    Add(r, Random(100))
```

Згадаємо, що при інфіксному виведенні БДП одержуємо відсортовану послідовність даних, значить при n викликах процедури додавання елементу до БДП, одержимо відсортовану послідовність із *n* елементів.

Тому даний алгоритм називається ***алгоритмом сортування деревом***.

Розберемося, скільки дій зажадає одне додавання *в середньому*.

Нехай дерево, у яке ми додаємо, є *ідеально-збалансованим*, тоді кількість його рівнів $k \approx log^2 n$, де n – кількість вузлів.

Бачимо, що кількість операцій в Add визнається глибиною дерева *k* (при кожному рекурсивному виклику Add ми опускаємося вниз по дереву на 1 рівень, і вставка здійснюється в *лист* дерева).

Тобто, при додаванні *n* елементів, у випадку, якщо дерево щоразу залишається близьким до *ідеально-збалансованого*, витрачається $n\log^2 n$ дій (стільки ж, скільки при *швидкому сортуванні*).

Зауваження 1. Дана оцінка справедлива *в середньому*.

Зауваження 2. Дана оцінка збігається з оцінкою кількості операцій при швидкому сортуванні, а значить є *оптимальною для довільних даних*.

Оцінка кількості операцій при сортуванні деревом у найгіршому випадку

Будемо додавати елементи в БДП у зростаючому порядку:

```
Add(10)
Add(15)
Add(17)
...
Add(87)
```

Одержимо "однобоке" дерево (рис. 11.7):

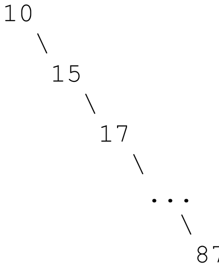


Рисунок 11.7 – Найгірший випадок

При такому порядку додавань, кількість операцій становить *приблизно* n^2 (як і в найгіршому разі швидкого сортування).

Зауваження. Щоб зберегти асимптотичну оцінку $n\log^2 n$ і в найгіршому випадку, щоразу, при додаванні в дерево, треба здійснювати так зване **перебалансування**, що зберігає властивість дерева бути деревом пошуку, але зменшує, по можливості, його глибину до мінімальної.

Пошук елементу в БДП

```
function boolean Find(TreeNode root, int x)
 begin
    if root = nil then
        Result = false
    else if root.data == x then
        Result = True
    else if x < root.data then
        Result = Find(root.left, x)
    else // if x > root.data then
        Result = Find(root.right, x)
 end
```

Очевидно, кількість операцій збігається із глибиною БДП $k \approx \log^2 n$ у середньому.

Помітимо, що алгоритм ***бінарного пошуку*** у відсортованому масиві займає стільки ж дій.

Питання до розділу:

1. Поясніть, що таке дерево, глибина дерева, корінь дерева і листи.
2. Що таке бінарне (двійкове) дерево і за якої умови воно буде ідеально збалансованим і повним?
3. Які існують алгоритми обходів дерев?
4. Що таке зворотний польський запис виразів?
5. Який зв'язок між деревами і рекурсіями?

6. Як здійснюється пошук елементу в бінарному дереві?
7. Які є алгоритми визначення мінімальної суми від кореня до листа?
8. Яке бінарне дерево називають бінарним деревом пошуку?
9. Яким чином додають елемент до бінарного дерева пошуку?

Розділ 12. Алгоритми на графах. Основні терміни теорії графів. Представлення графів в комп'ютерних програмах

Теорія графів останнім часом широко використовується в різних галузях науки й техніки. Швидкий розвиток дана теорія одержала зі створенням електронно-обчислювальної техніки, що дозволила вирішити багато задач алгоритмізації.

Ключові терміни

Граф – це сукупність двох скінченних множин: множини точок і множини ліній, що попарно з'єднують деякі із цих точок. Множина точок називається *вершинами (вузлами) графа*. Множина ліній, що з'єднують вершини графа, називається *ребрами (дугами) графа*.

Орієнтований граф (орграф) – граф, у якого всі ребра орієнтовані, тобто ребрам якого привласнений напрямок.

Неорієнтований граф (неорграф) – граф, у якого всі ребра неорієнтовані, тобто ребрам якого не заданий напрямок.

Змішаний граф – граф, що містить як орієнтовані, так і неорієнтовані ребра.

Петлею називається ребро, що з'єднує вершину саму із собою. Дві вершини називаються *суміжними*, якщо існує з'єднуюче їх ребро. Ребра, що з'єднують ті самі пари вершин, називаються *кратними*.

Простий граф – це граф, у якому немає ні петель, ні кратних ребер.

Мультиграф – це граф, у якого будь-які дві вершини з'єднані більш ніж одним ребром.

Маршрутом у графі називається скінченна послідовність суміжних вершин і ребер, що з'єднують ці вершини.

Маршрут називається *відкритим*, якщо його початкова й кінцева вершини різні, у протилежному випадку він називається *замкнутим*.

Маршрут називається **ланцюгом**, якщо всі його ребра різні. Відкритий ланцюг називається **шляхом**, якщо всі його вершини різні.

Замкнутий ланцюг називається **циклом**, якщо різні всі його вершини, за винятком кінцевих.

Граф називається *зв'язним*, якщо для будь-якої пари вершин існує з'єднуючий їх шлях.

Вага вершини – число (дійсне, ціле або раціональне), поставлене у відповідність даній вершині (інтерпретується як вартість, пропускна здатність і так далі). *Вага (довжина) ребра* – число або декілька чисел, які інтерпретуються відносно ребра як довжина, пропускна здатність і так далі.

Зважений граф – граф, кожному ребру якого поставлено у відповідність якесь значення (вага ребра).

Способи представлення графа в комп'ютерній програмі

Вибір структури даних для зберігання графа в пам'яті комп'ютера має принципове значення при розробці ефективних алгоритмів. Розглянемо кілька **способів представлення графа**.

Нехай заданий граф (наприклад, рис. 12.1), у якого кількість вершин дорівнює *n*, а кількість ребер – *m*. Кожне ребро й кожна вершина мають вагу – ціле позитивне число. Якщо вага ребра не вказана, то вважається, що вага дорівнює одиниці.

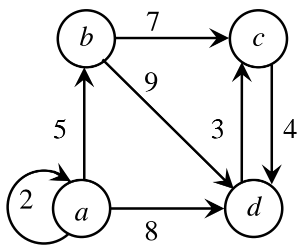


Рисунок 12.1 – Граф

1. *Список ребер* – це множина, утворена парами суміжних вершин (рис. 12.2). Для його зберігання звичайно використовують одномірний масив розміром m, що містить список пар вершин, суміжних з одним ребром графа. Список ребер більш зручний для реалізації різних алгоритмів на графах у порівнянні з іншими способами.

a	*a*	*a*	*b*	*b*	*c*	*d*
a	*b*	*d*	*c*	*d*	*d*	*c*
2	*5*	*8*	*7*	*9*	*4*	*3*

Рисунок 12.2 – Список ребер графа

2. *Матриця суміжності* – це двовимірний масив розмірності *n* x *n*, значення елементів якого характеризуються суміжністю вершин графа (рис. 12.3). При цьому значенню елементу матриці привласнюється кількість ребер, які з'єднують відповідні вершини. Даний спосіб зручний, коли треба перевіряти суміжність або знаходити вагу ребра за двома заданими вершинами.

	a	*b*	*c*	*d*
a	2	5	0	8
b	0	0	7	9
c	0	0	0	4
d	0	0	3	0

Рисунок 12.3 – Матриця суміжності графа

3. *Матриця інцидентності* – це двовимірний масив розмірності *n* x *m*, у якому вказуються зв'язки між інцидентними елементами графа (ребро й вершина). Стовпці матриці відповідають ребрам, рядки – вершинам (рис. 12.4). Ненульове значення в комірці матриці вказує зв'язок між

вершиною й ребром. Даний спосіб є самим ємним для зберігання, але полегшує знаходження циклів у графі.

	a	*b*	*c*	*d*
(a, a)	2	0	0	0
(a, b)	0	5	0	0
(a, d)	0	0	0	8
(b, c)	0	0	7	0
(b, d)	0	0	0	9
(c, d)	0	0	0	4
(d, c)	0	0	3	0

Рисунок 12.4 – Матриця інцидентності графа

4. Списки суміжності. Цей спосіб представлення *графів* має на увазі, що для кожної *вершини* буде зазначений список всіх суміжних з нею *вершин* (для *орграфа* – список *вершин*, що є кінцями вихідних *дуг*). Конкретний формат вхідного файлу, що містить *списки суміжності*, необхідно обговорювати окремо. Наприклад, можна використати наступний формат – початкова *вершина* відділена від *списку суміжності* двокрапкою:
<номер_початкової_вершини>: <номери_суміжних_вершин>

Найбільш природно застосовувати цей спосіб для представлення *орграфів*, однак і для інших варіантів він теж підходить.

a: a 2 b 5 d 8
b: c 7 d 9
c: d 4
d: c 3

Рисунок 12.5 – Список суміжності графа

5. *Ієрархічний список.* Цей спосіб представлення графів є всього лише внутрішньою реалізацією списку суміжності: в одному лінійному списку містяться номери "початкових вершин", а в інших – номери суміжних вершин або вказівники на ці вершини.

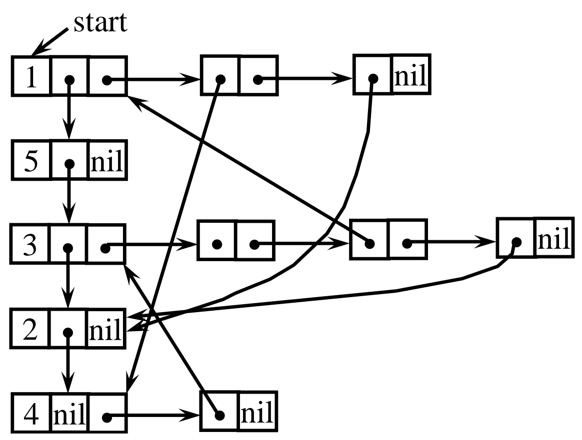


Рисунок 12.6 – Приклад ієрархічного списку

Очевидна перевага такого способу представлення *графів* полягає в економічному використанні пам'яті. І навіть невелика надмірність даних, до якої доводиться прибігати у випадку неорієнтованого *графа*, задаючи кожне *ребро* як дві *дуги*, окупається гнучкістю всієї структури, що особливо зручно при необхідності частих перебудовувань у процесі роботи програми.

Якщо в наведені описи типів даних додати поля, які могли б зберігати вагу *вершин* і *дуг*, то таким же способом можна задавати й зважені *графи*.

Дерева

Дерево – це окремий випадок *графа*, що найбільш широко застосовується у програмуванні.

Основні визначення

Існує досить багато рівносильних визначень *дерев*, от лише деякі з них.

1. *Дерево* – це зв'язаний *граф* без *циклів*.

2. *Дерево* – це зв'язаний *граф*, у якому при *N вершинах* завжди рівно (*N* – 1) *ребро*.

3. *Дерево* – це *граф*, між будь-якими двома *вершинами* якого існує рівно один *шлях*.

Аналогічним чином визначається й **орієнтоване** *дерево* – як *орграф*, у якому між будь-якими двома *вершинами* існує не більше одного *шляху*.

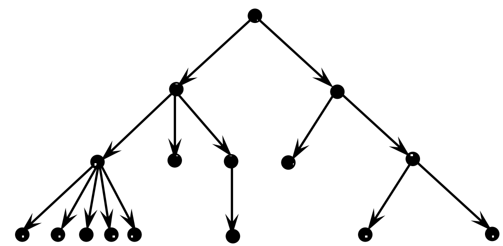


Рисунок 12.7 – Кореневе дерево з висотою 3

Кореневе дерево – це орієнтоване *дерево*, у якому можна виділити *вершини* трьох видів: *корінь*, *листи* (інша їхня назва: ***термінальні вершини***) й інші *вершини* (***нетермінальні***); причому повинні виконуватися дві обов'язкових умови:

1) з *листів* не виходить жодна *дуга*; з інших *вершин* може виходити безліч *дуг*;

2) у *корінь* не заходить жодна *дуга*; в усі інші *вершини* заходить рівно по одній *дузі*.

Традиційно в математиці й близьких їй науках (у тому числі й у теоретичному програмуванні) *дерева* "ростуть" вниз

головою: це робиться просто для зручності нарощування *листів* якщо буде потреба. Таким чином, на малюнках *корінь дерева* виявляється самою верхньою *вершиною*, а *листи* – самими нижніми.

Предок *вершини* v – це *вершина*, з якої виходить *дуга*, що заходить у *вершину* v. *Нащадок вершини v* – це *вершина*, у яку заходить *дуга*, що виходить із *вершини v*. У цих термінах можна дати інші визначення поняттям *корінь* і *лист*: у *кореня* немає *предків*, в *листа* немає *нащадків*.

Висота *кореневого дерева* – це максимальна кількість *дуг*, що відокремлюють *листи* від кореня. Якщо *дерево* не зважене, то його *висота* – це проста *відстань* від кореня до самого віддаленого листа.

Каркас графа – це *дерево*, отримане після викидання із *графа* деяких ребер (рис. 12.8).

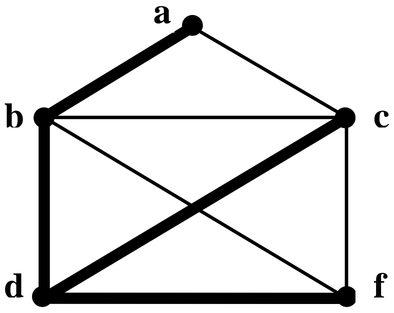


Рисунок 12.8 – Каркас графа

Прикладом *каркаса* є (*кореневе*) *дерево* найкоротших *шляхів* від деякої виділеної *вершини* (вона буде коренем *каркаса*) до всіх інших *вершин графа*.

Питання до розділу:

1. Поясніть поняття граф, вузли графа, дуги графа, корінь графа, орграф, неорграф, змішаний граф, петля, суміжні вершини, кратні ребра, простий граф, мультиграф, маршрут у графі, відкритий маршрут, замкнутий маршрут, ланцюг, шлях, цикл, зв’язний граф, вага вершини, вага ребра, зважений граф.
2. Назвіть способи представлення графів в комп’ютерній програмі. Які переваги має кожний із цих способів?
3. Дайте означення дерева, орієнтованого дерева, кореневого дерева, термінальних вершин, нетермінальних вершин.
4. Чому в математиці і близьких їй науках дерева "ростуть" вниз головою?
5. Що таке предок, нащадок, висота кореневого дерева, каркас?

Розділ 13. Алгоритми на графах. Пошук найкоротших шляхів. Каркасні дерева

Позначення

Орієнтований граф (скорочено *орграф*) $G = (V, E)$ складається із множини вершин V і множини дуг E. Вершини також називають *вузлами*, а дуги – *орієнтованими ребрами.* Дуги записуються у вигляді впорядкованої пари вершин (v, w), де вершина v називається початком, а w – кінцем дуги.

Неорієнтований граф $G = (V, E)$ складається з кінцевої множини вершин V і множини ребер E. На відміну від орієнтованого графа, тут кожне ребро (v, w) відповідає *неупорядкованій* парі вершин: якщо (v, w) – неорієнтоване ребро, то $(v, w) = (w, v)$.

Алгоритм Дейкстри

Алгоритм Дейкстри вирішує задачу про найкоротші шляхи з однієї вершини для зваженого орієнтованого графа $G = (V, E)$ з початковою вершиною s, у якій вага всіх ребер не негативна.

Псевдокод алгоритму Дейкстри

$d[u]$ – відстань до вершини u з початкової вершини s.

Q – черга із пріоритетами. Витягується вершина, для якої $d[u]$ мінімальне

```
function InitializeSingleSource(G, s)
 begin
    for (для) всіх вершин v з V[G] do
        d[v] = ∞
    батько[v] = NIL
    d[s] = 0

    Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] = d[u] + w(u, v)
```

```
    предок[v] = u
 end

function Dijkstra(G, w, s)
 begin
    InitializeSingleSource(G, s)
    S = O
    Q < V[G]
    while Q != Ø do
        u ← витягти вершину, для якої d[u] мінімальне
    S = S U {u}
    for (для) всіх вершин v суміжних з u do
        Relax(u, v, w)
 end
```

Суть алгоритму. *d*[*u*] – це оцінка шляху по кожній з вершин на даному кроці. Спочатку нам відомо, що до початкової вершини шлях 0, до інших оцінка на першому кроці нескінченність ∞. На кожному кроці беремо вершину *u*, для якої оцінка мінімальна. Для неї можна довести, що оцінка є найкоротшим шляхом. Для суміжних з нею вершин *v*, якщо оцінка через вершину *u* виявляється менше, ніж поточна оцінка вершини *v*, то зменшуємо оцінку (релаксація).

Оцінка часу роботи. Оцінка часу роботи залежить від ефективності реалізації черги із пріоритетами. При використанні масиву, оцінка буде $O(V^2)$; при використанні двійкової купи оцінка буде $O(E \cdot \log V)$; при використанні фібоначієвої купи оцінка $O(V \cdot \log V + E)$.

Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда вирішує задачу про найкоротшу вагу з однієї вершини для випадку, коли вагам ребер дозволено бути негативними. Цей алгоритм повертає TRUE, якщо в графі немає циклу негативної ваги, досяжного з початкової вершини, і FALSE, якщо такий цикл є. У першому випадку алгоритм знаходить найкоротші шляхи і їхню вагу; у

другому випадку найкоротших шляхів (принаймні, для деяких вершин) не існує.

Псевдокод алгоритму Беллмана-Форда

```
d[u] – відстань до вершини u з початкової вершини s

InitializeSingleSource(G, s)
 begin
    for (для) всіх вершин v з V[G] do
        d[v] = ∞
    батько[v] = NIL
    d[s] = 0
 end

Relax(u, v, w)
 begin
    if d[v] > d[u] + w(u, v) then
        d[v] = d[u] + w(u, v)
    предок[v] = u
 end

BellmanFord(G, w, s)
 begin
    InitializeSingleSource(G, s)
    for i = 1 to |V[G]| - 1 do
        for (для) кожного ребра (u, v) з E[G] do
            Relax(u, v, w)
    for (для) кожного ребра (u, v) з E[G] do
        if d[v] > d[u] + w(u, v) then
            return FALSE
    return TRUE
 end
```

Оцінка часу роботи. Час роботи алгоритму – $O(V \cdot E)$.

Алгоритм Флойда-Воршолла

Може виникати задача, коли потрібно знайти найкоротшу відстань між всіма парами вершин. Алгоритм Флойда-Воршолла використовує ідею динамічного програмування й дозволяє за $O(V^3)$ знайти найкоротші шляхи

між всіма парами вершин орієнтованого зваженого графа. Вага ребер може бути негативною, але не допускається існування циклів негативної ваги.

Псевдокод алгоритму Флойда-Воршолла

```
W – матриця ваги n x n
D – матриця відстаней n x n
FloydWarshall(W)
 begin
    D = W
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
               D[i, j] = min(D[i, j], D[i, k] + D[k, j])
 end
```

Результатом роботи алгоритму буде матриця *D* найкоротших відстаней між вершинами.

Каркасні (остовні) дерева

Зв'язаний підграф графа *G*, що є деревом і містить всі його вершини, називають **каркасним деревом** цього графа.

Мінімальним каркасним деревом є каркасне дерево, сума ваги ребер якого мінімальна.

Алгоритми, запропоновані нижче, будуть ґрунтуватися на наступній властивості. Нехай $G = (V, E)$ – зв'язний граф із заданою функцією вартості, визначеної на множині ребер. Позначимо через *U* підмножину множини вершин *V*. Якщо (u, v) – таке ребро найменшої вартості, що $u \in U$ і $v \in V/U$, тоді для графа G існує каркасне дерево мінімальної вартості, що містить ребро (u, v).

Алгоритм Прима

Q – черга із пріоритетами, ключем у якій є величина *key*[*v*] рівна мінімальній вазі ребра з вершини *v* у вершину із множини вже оброблених.

r – корінь каркасного дерева.

```
MSTPrim(G, w, r)
 begin
    Q = V[G]
    for (для) кожної вершини u з Q do
        key[u] = ∞
    key[r] = 0
    предок[r] = NIL
    while Q != Ø do
      u ← витягти вершину, для якої key[u] мінімальне
    for (для) кожної вершини v суміжної з u do
        if (v Є Q) AND (w(u, v) < key[v]) then
            предок[v] = u
            key[v] = w(u, v)
 end
```

Оцінка часу роботи. Оцінка часу роботи залежить від ефективності реалізації черги із пріоритетами. При використанні двійкової купи оцінка буде $O(E \cdot \log V)$; при використанні купи Фібоначі оцінка $O(E + V \cdot \log V)$.

Псевдокод алгоритму Крускала

```
MSTKruskal(G, w)
 begin
     A < O
     for (для) кожної вершини v з V[G] do
        Make-Set(v)
     упорядкувати ребра E за вагами
     for (для) (u, v) з E (у порядку зростання ваги) do
        if FindSet(u) != FindSet(v) then
           A ← A U {(u, v)}
           Union(u, v)
     return A
 end
```

Суть алгоритму. На початку кожної роботи алгоритму кожна вершина графа лежить у своїй множині (має свій колір). По ходу роботи алгоритму проглядаються ребра в порядку зростання ваги. Якщо ребро з'єднує вершини з різних множин

(різного кольору), то воно не створює циклу й, відповідно, може бути додане в дерево, яке ми будуємо.

Оцінка часу роботи. Оцінка дорівнює $O(E \cdot \log E)$, тобто основний час іде на сортування. Передбачається, що для зберігання непересічних множин використовується метод з об'єднанням за рангом і стиском шляхів.

Питання до розділу:

1. Чому в неорієнтованому графі ребро відповідає неупорядкованій парі вершин?
2. Яку задачу вирішує алгоритм Дейкстри?
3. В чому суть алгоритму Дейкстри?
4. Яку задачу вирішує алгоритм Беллмана-Форда?
5. Яку задачу вирішує алгоритм Флойда-Воршолла?
6. Що таке каркасне дерево?
7. Поясніть алгоритм Прима.
8. Поясніть алгоритм Крускала.

Розділ 14. Алгоритми на графах. Алгоритми обходу графа

Існує багато алгоритмів на графах, в основі яких лежить систематичний перебір вершин графа, такий що кожна вершина проглядається (відвідується) у точності один раз. Тому важливою задачею є знаходження гарних методів пошуку в графі.

Під **обходом графів (пошуком на графах)** розуміється процес систематичного перегляду всіх ребер або вершин графа з метою відшукання ребер або вершин, що задовольняють деякій умові.

При рішенні багатьох задач, що використовують графи, необхідні ефективні методи регулярного обходу вершин і ребер графів. До стандартних й найпоширеніших методів відносяться:

- пошук у глибину (Depth First Search, DFS);
- пошук у ширину (Breadth First Search, BFS).

Ці методи найчастіше розглядаються на орієнтованих графах, але вони можуть застосовуватися і для неорієнтованих, ребра яких вважаються двоспрямованими. Алгоритми обходу в глибину і в ширину лежать в основі рішення різних задач обробки графів, наприклад, побудови каркасного лісу, перевірки зв'язності, ациклічності, обчислення відстаней між вершинами тощо.

Пошук у глибину

При пошуку в глибину відвідується перша вершина, потім необхідно йти уздовж ребер графа, до потрапляння в глухий кут. Вершина графа є *глухим кутом*, якщо всі суміжні з нею вершини вже відвідані. Після влучення в глухий кут потрібно вертатися назад уздовж пройденого шляху, поки не буде виявлена вершина, у якої є суміжна ще не відвідана

вершина, а потім необхідно рухатися в цьому новому напрямку. Процес є завершеним при поверненні в початкову вершину, причому всі суміжні з нею вершини вже повинні бути відвідані.

Таким чином, основна ідея пошуку в глибину – коли можливі шляхи по ребрах, що виходять із вершин, розгалужуються, потрібно спочатку повністю досліджувати одну гілку й тільки потім переходити до інших гілок (якщо вони залишаться нерозглянутими).

Алгоритм пошуку в глибину

Крок 1. Всім вершинам графа привласнюється значення «не відвідана». Вибирається перша вершина й позначається як відвідана.

Крок 2. Для останньої позначеної як відвідана вершини вибирається суміжна вершина, що є першою позначеною як не відвідана, і їй привласнюється значення відвідана. Якщо таких вершин немає, то береться попередня позначена вершина.

Крок 3. Повторити крок 2 доти, поки всі вершини не будуть позначені як відвідані (рис. 14.1).

Псевдокод алгоритму пошуку в глибину

```
function DepthFirstSearch(int n, int **Graph, bool
*Visited, int Node)
 begin
    visited[Node] = true
    write(Node + 1)

    for int i = 0 ; i < n ; i++
        if Graph[Node][i] And (Not Visited[i])
            DepthFirstSearch(n, Graph, Visited, i)

 end
```

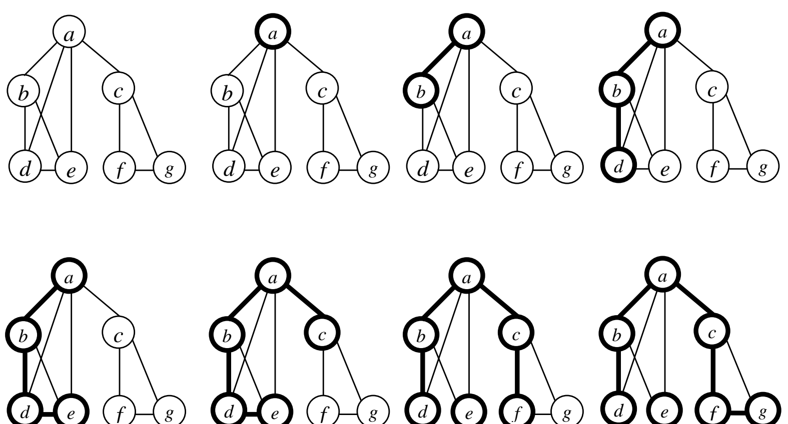


Рисунок 14.1 – Демонстрація алгоритму пошуку в глибину

Також часто використовується нерекурсивний алгоритм пошуку в глибину. У цьому випадку рекурсія заміняється на стек. Як тільки вершина переглянута, вона поміщається в стек, а використаною вона стає, коли більше немає нових вершин, суміжних з нею.

Часова складність залежить від представлення графа. Якщо застосовано матрицю суміжності, то часова складність дорівнює $O(n^2)$, а якщо нематричне представлення – $O(n+m)$: розглядаються всі вершини й всі ребра.

Пошук в ширину

При пошуку в ширину, після відвідування першої вершини, відвідуються всі сусідні з нею вершини. Потім відвідуються всі вершини, що перебувають на відстані двох ребер від початкової. При кожному новому кроці відвідуються вершини, відстань від яких до початкової на одиницю більша попередньої. Щоб запобігти повторному відвідуванню вершин, необхідно вести список відвіданих вершин. Для зберігання тимчасових даних, необхідних для

роботи алгоритму, використовується черга – упорядкована послідовність елементів, у якій нові елементи додаються в кінець, а старі видаляються із початку.

Таким чином, основна ідея пошуку в ширину полягає в тому, що спочатку досліджуються всі вершини, суміжні з початковою вершиною (вершина з якої починається обхід). Ці вершини перебувають на відстані 1 від початкової. Потім досліджуються всі вершини на відстані 2 від початкової, потім всі на відстані 3 і так далі. При цьому для кожної вершини відразу визначається довжина найкоротшого маршруту від початкової вершини.

Алгоритм пошуку в ширину

Крок 1. Всім вершинам графа привласнюється значення не відвідана. Вибирається перша вершина й позначається як відвідана (і заноситься в чергу).

Крок 2. Відвідується перша вершина із черги (якщо вона не позначена як відвідана). Всі її сусідні вершини заносяться в чергу. Після цього вона видаляється із черги.

Крок 3. Повторюється крок 2 доти, поки черга не є порожньою (рис. 14.2).

Псевдокод алгоритму пошуку в ширину

```
void BreadthFirstSearch(int n, int **Graph,
                        bool *Visited, int Node)
 begin
    int *List = new int[n] //черга
    int Count, Head        // вказівники черги
    int i
    for i = 0; i < n ; i++
        List[i] = 0 // початкова ініціалізація
    Count = Head = 0
  // розміщення в чергу вершини Node
    List[Count++] = Node
    Visited[Node] = true
    while (Head < Count)
```

```
    begin
       //взяття вершини із черги
       Node = List[Head++]
       write(Node + 1)
    //перегляд всіх вершин, в'язаних з вершиною Node
       for i = 0 ; i < n ; i++
         // якщо вершина раніше не переглянута
         if (Graph[Node][i] && !Visited[i])
         begin
           // заносимо її в чергу
           List[Count++] = i
           Visited[i] = true
         end
     end
 end
```

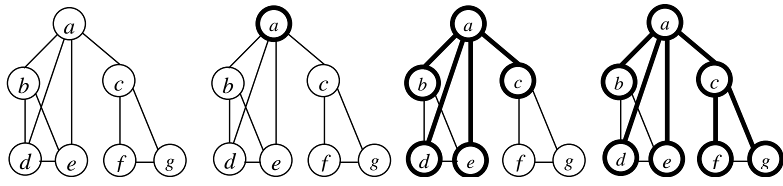


Рисунок 14.2 – Демонстрація алгоритму пошуку в ширину

Складність пошуку в ширину при нематричному представленні графа дорівнює $O(n + m)$, тому що розглядаються всі n вершин і m ребер. Використання матриці суміжності приводить до оцінки $O(n^2)$.

Питання до розділу:

1. Що розуміють під терміном "обхід графу" ?
2. В чому полягає основна ідея методу пошуку в глибину?
3. Поясніть алгоритм пошуку в глибину?
4. В чому полягає основна ідея методу пошуку в ширину?
5. Поясніть алгоритм пошуку в ширину?

Розділ 15. Клітинні автомати. Поняття, види, застосування, приклади реалізації

Клітинний автомат (КА) – сукупність, до якої входять:

1. Набір клітинок, які утворюють періодичну решітку.
2. Задані правила переходу, що визначають стан клітини за теперішнім станом самої клітинки та тих її сусідів, що знаходяться від неї на певній відстані, яка не перевищує максимальну.

Можливі визначення

Поняття клітинних автоматів доволі обширне, тому можна знайти доволі багато різних визначень. Найпоширенішими є:

- математичний об’єкт з дискретним простором та часом;
- регулярна структура двійкових скінченних автоматів з однаковими правилами переходів, що виражені у вигляді булевих функцій від станів сусідніх автоматів;
- стилізовані, синтетичні світи, що визначені простими правилами, подібно правилам настільної гри;
- математична ідеалізація фізичної системи, в якій час та простір дискретні, а фізичні величини приймають скінченну множину значень.

Отже, що ж потрібно для клітинного автомата? По-перше, це нескінченна (або кінцева) площина, розділена на клітини (квадратні або інші форми). Кожна із клітин може бути в одному зі станів з певного набору станів (тобто стан клітини дискретний). По-друге, годинники. Час у клітинному автоматі теж дискретний. І по-третє, набір правил, що визначає поводження автомата.

Клітинні автомати є дискретними динамічними системами, поведінка яких повністю визначається в

термінах локальних залежностей, в значній мірі те саме відноситься і до великого класу безперервних динамічних систем, визначених рівняннями в частинних похідних. У цьому сенсі клітинні автомати в інформатиці є аналогом фізичного поняття «поле».

Клітинний автомат може мислитися як стилізований світ. Простір представлено рівномірною сіткою, кожна комірка якої містить декілька бітів даних, час йде вперед дискретними кроками, а закони світу виражаються єдиним набором правил, скажімо, невеликою довідковою таблицею, за якою будь-яка клітина на кожному кроці обчислює свій новий стан за станам її близьких сусідів. Таким чином, закони системи є локальними і всюди однаковими.

Якщо заданий відповідний набір правил, то такий простий операційний механізм достатній для підтримання цілої ієрархії структур і явищ. Клітинні автомати дають корисні моделі для багатьох досліджень в природничих і обчислювальних науках і комбінаторній математиці, вони, зокрема, представляють природний шлях вивчення еволюції великих фізичних систем. Клітинні автомати до того ж утворюють загальну парадигму паралельних обчислень, подібно тому, як це роблять машини Тюрінга для послідовних обчислень.

Клітинні автомати винаходилися багато разів під різними назвами, і трохи різні один від одного поняття вживалися під однією і тією ж назвою. У чистій математиці їх можна виявити як один з розділів топологічної динаміки, в електротехніці вони іноді називаються ітеративними масивами, а в інформатиці це вид гри на домашньому комп'ютері.

Клітинний автомат працює так:

1. Задається початковий стан всіх клітин КА.
2. Для кожної із клітин за певними правилами зі стану клітин-сусідів і самої клітини обчислюється наступний її стан.
3. На кожному кроці клітини міняють свій стан одночасно.

Далі будемо розглядати автомати із квадратними клітинами, а трикутні, шестикутні й інші залишимо в спокої (як не дивно, але КА із клітинами неквадратної форми можна моделювати за допомогою КА із квадратними клітинами).

Особливо треба визначити сусідів клітини (або її окіл). Якщо в розрахунок беруться сама клітина й клітини, що стикаються з нею тільки сторонами (їх чотири), такий окіл називається околом фон Неймана (рис. 15.1).

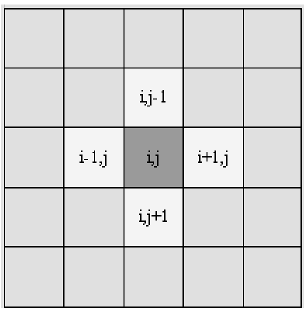


Рисунок 15.1 – Окіл фон Неймана

Якщо ж у розрахунок беруться сама клітина й клітини, що стикаються з нею не тільки сторонами, але й кутами, така околиця називається околом Мура (рис. 15.2).

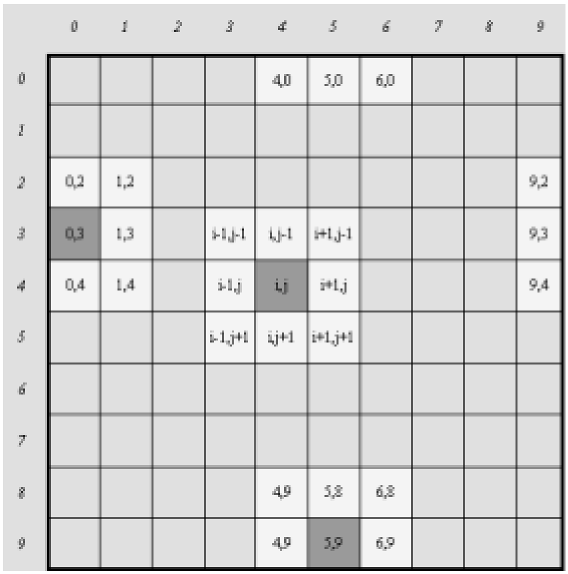


Рисунок 15.2 – Окіл Мура

Клітинний автомат гра “Життя”

Найвідоміший із всіх КА – це звичайно ж гра “Життя”, винайдена Дж. Конвеем.

Ми будемо розглядати клітинні автомати у двомірному просторі (на площині), однак, ніхто не обмежує автомати двома вимірами. Правила гри можна перенести й у три, і в більшу кількість вимірів. Наприклад, існують тривимірні версії гри “Життя”.

Правила гри ”Життя” дуже прості:

1. Клітини можуть перебуває у двох станах – “жива” або “мертва” клітина.
2. Окіл клітини – окіл Мура (сама клітина й 8 сусідів).
3. Жива клітина, що має 2 або 3 живих сусідів залишається живою на наступному кроці.

4. Будь-яка клітина, що має менше 2 або більше 3 живих сусідів, вмирає.

5. Мертва клітина, що має 3 живих сусідів на наступному кроці перетворюється в живу.

Дані правила отримали назву *генетичних законів Конвея*, вони задовольняють трьом основним умовам:

1) не має бути жодної початкової конфігурації, для якої існувало б просте доведення можливості необмеженого росту популяції;
2) мають існувати такі початкові конфігурації, які заздалегідь володіють властивістю безмежно розвиватися;
3) мають існувати прості початкові конфігурації, які протягом значного проміжку часу ростуть, перетерплюють різноманітні зміни та закінчують свою еволюцію одним з трьох наступних способів:
   a. повністю щезають;
   b. переходять у стійку конфігурацію та перестають змінюватися взагалі;
   c. виходять у коливальний режим з певним періодом.

Гравець не бере прямої участі у грі, а лише розставляє початкову конфігурацію «живих» клітин, які потім взаємодіють відповідно до правил уже без його участі.

Фігури

Ці прості правила призводять до виникнення величезної кількості різноманітних форм, кожна з яких має дещо спільне з попередньою. На цей час склалася така система їхньої класифікації:

Стійкі фігури – фігури, які залишаються незмінними за кожної ітерації

Періодичні фігури – фігури, стан яких повторюється через деяку кількість поколінь.

Фігури, що рухаються – фігури у яких стан повторюється, але з деяким зсувом у просторі.

Гармати – фігури у яких стан повторюється, але кожен цикл вони додатково створюють фігури, що рухаються.

Паротяги – фігури, що рухаються, які залишають за собою сліди у вигляді стійких або періодичних фігур.

Пожирачі – стійкі фігури, які при зіткненні з деякими фігурами, що рухаються, зберігають свій стан, знищуючи рухому фігуру.

Приклади

Незабаром після публікації правил, було виявлено декілька цікавих фігур, зокрема: *r*-пентаміно, глайдер (англ. *glider*).

Нерухомі фігури

Нерухомі фігури не змінюються з плином часу. Найпростіший приклад нерухомої фігури – блок.

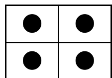


Рисунок 15.3 – Нерухома фігура "Блок"

Осцилятори

Осцилятор – така фігура, що має певну періодичність. Приклад: лінія з 3-х клітин.

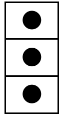


Рисунок 15.4 – Періодична фігура "Смужка"

Планери

Планери (англ. *glider*) – рухомі фігури, які є періодичними, але з кожним циклом руху вони зміщуються на декілька клітин у певному (зазвичай сталому) напрямку.

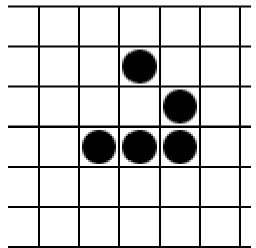


Рисунок 15.5 – Класичний планер

Фігура планер в 2003 році була запропонована в якості емблеми хакерів.

Гармата планерів

Гармата планерів – періодична фігура, яка за повний цикл генерує один чи декілька планерів.

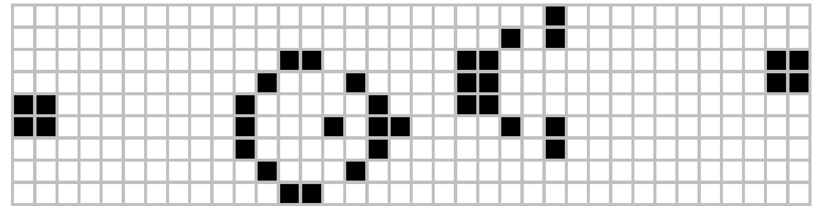


Рисунок 15.6 – Гармата планерів (глайдерна гармата)

Конвей спочатку припустив, що ніяка початкова комбінація не може привести до необмеженого розмноження і запропонував премію в 50 доларів тому, хто доведе або спростує цю гіпотезу. Приз був отриманий

групою з Массачусетського технологічного інституту, яка придумала нерухому повторювану фігуру, яка періодично створювала рухомі «планери». Таким чином, кількість живих клітин могла рости необмежено. Потім були знайдені рухомі фігури, що залишають за собою «сміття» з інших фігур.

Едемський сад

Едемським садом називається таке розташування клітин, у якого не може бути попереднього покоління. Практично для будь-якої гри, стан кліток, в якій визначається декількома сусідами на попередньому кроці, можна довести існування садів Едему, хоча побудова конкретної фігури є набагато складнішою.

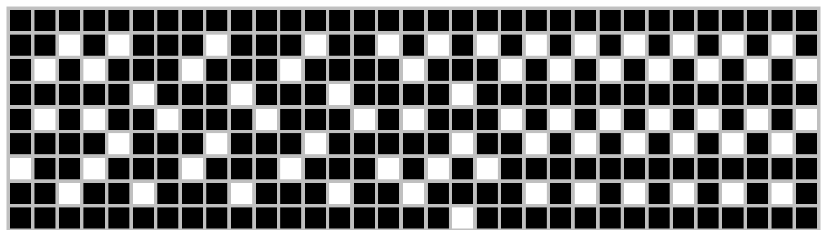


Рисунок 15.7 – Приклад Едемського саду

Каркас програми

Для початку нам потрібний масив, що буде зберігати стан кожної клітини. Назвемо його old[0 .. *size* – 1, 0 .. *size* – 1] (*size*, *size* – розмір поля відповідно по горизонталі й вертикалі.

Можна було б використовувати координати клітин починаючи з 1, але з нуля починати зручніше). Потім, нам потрібний ще один масив new (такий же як і old), де ми будемо зберігати значення, які одержать наші клітини на наступному кроці.

Псевдокод:

```
ЗадатиПочатковуКонфігурацію(old)
 Початок
   для i = 0 до size-1
   для j = 0 до size-1
   new[i, j] = Правило(i, j)
   old = new
   Вивести(new)
 Кінець
```

Спочатку потрібно задати початкову конфігурацію клітин. Це може бути й заповнення масиву випадковим чином, або з файлу, або користувачем вручну.

Основний цикл програми такий – проходимо по кожній клітині, за певним правилом обчислюємо новий стан клітини й записуємо його в масив new. Потім переписуємо масив new у масив old і виводимо на екран стан клітин.

Залишається тільки написати функцію Правило(i, j). Позначимо 0 – мертва клітина, 1 – жива. Нам потрібно обчислити суму станів всіх сусідів нашої клітини, вона ж буде кількістю живих клітин в околі. А як бути, якщо клітина перебуває на границі масиву (тобто $i = 0$ або $i = (size - 1)$, $j = 0$ або $j = size - 1$). Можна, звичайно, перевіряти всі ці умови. Але є й інший варіант. Згорнемо наше клітинне поле в тор. Тобто лівий край поля буде стикатися із правим, а верхній – з нижнім.

Псевдокод реалізації правил гри "Життя":

```
Функція Правило(i, j)
 Початок // операція mod - залишок від ділення
   im1 = ((i - 1) + size) mod size
   ip1 = (i + 1) mod size
   jm1 = ((j - 1) + size) mod size
```

```
   jp1 = (j + 1) mod size
   sum = old[im1, jm1] + old[im1, j] + old[im1, jp1]
         + old[i, jm1] + old[i, jp1] + old[ip1, jm1]
         + old[ip1, j] + old[ip1, jp1]
   якщо sum == 3 то повернути 1
   інакше якщо sum == 2 то повернути old[i,j]
   інакше пвернути 0
 Кінець
```

im1 – це (i - 1) у наших координатах, ip1 – це (i + 1), jm1 – (j – 1), jp1 – (j + 1), mod позначає узяття залишку від ділення. Результат функції обчислюємо за вищенаведеним правилом для гри “Життя”.

Циклічний автомат

Циклічний автомат був відкритий Девідом Гріффітом. Починаючи роботу з випадковим чином обраним клітинним простором, такий автомат поступово перетворить поле клітин в обертові ромбовидні спіралі (рис. 15.8).

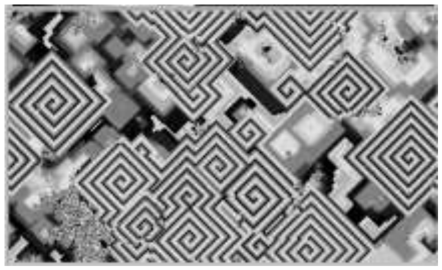


Рисунок 15.8 – Циклічний автомат

Циклічний автомат – це клітинний автомат з кількістю станів n (від 0 до n – 1). Правило його дуже просте. Якщо клітина на даний момент перебуває в стані k, то на наступному кроці всі сусідні з нею клітини, що перебувають

у стані ($k - 1$), переходять у стан k. Можна сказати, що ця клітина “з'їдає” всі клітини по сусідству, які перебувають на “нижчому” рівні розвитку. Якщо клітина перебуває в стані 0, то вона “поглинає” всі клітини зі станом ($n - 1$). Давайте напишемо функцію “Правило” для такого автомата (використовуємо окіл фон Неймана), псевдокод:

```
Функція Правило(i,j)
 початок
    im1 = ((i - 1) + size) mod size
    ip1 = (i + 1) mod size
    jm1 = ((j - 1) + size) mod size
    jp1 = (j + 1) mod size
    sp1 = (old[i, j] + 1) mod n
    якщо (old[im1, j] = sp1) або (old[ip1, j] = sp1)
або (old[i, jm1] = sp1) або (old[i, jp1] = sp1) то
        Результат == sp1
    інакше результат == old[i,j]
 Кінець
```

Перші чотири рядки цієї функції стандартні – обчислюються координати сусідніх клітин на торі. Змінна sp1 – це номер стану клітини, що може “з'їсти” нашу поточну клітину. Цей номер більше поточного стану клітини на 1 по модулі n. Тобто якщо клітина перебуває в стані ($n - 1$), то “з'їсти” її може клітина в стані 0.

Далі перевіряється, чи може яка або із сусідніх клітин “з'їсти” нашу клітину, і якщо це так, то стан поточної клітини стає рівним стану клітини, яка її “з'їла”.

Килимок

Розглянемо ще один простий КА, що називається «Килимок» (рис. 15.9).

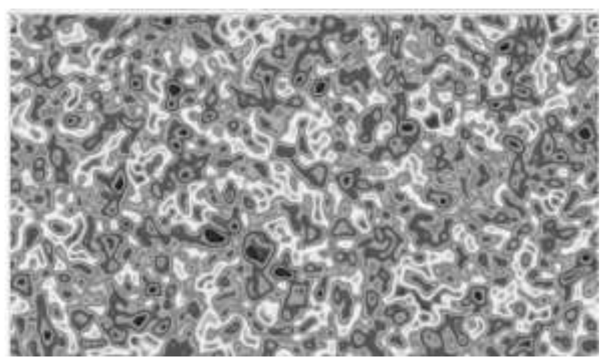


Рисунок 15.9 – Клітинний автомат «Килимок»

У ньому наступний стан клітини визначається так:

```
Функція Правило(i, j)
 Початок
    im1 = ((i - 1)+size) mod size
    ip1 = (i + 1) mod size
    jm1 = ((j - 1) + size) mod size
    jp1 = (j + 1) mod size
    повернути (((old[im1, jm1] + old[i, m1] +
old[ip1, jm1] + old[im1, j] + old[ip1, j] + old[im1,
jp1] + old[i,jp1] + old[ip1,jp1])/8 + shift) and
mask) mod n
 Кінець
```

Ми визначаємо середнє значення стану сусідніх клітин, додаємо зсув shift, маскуємо маску mask (тобто виконуємо логічне додавання з маскою) і потім робимо так, щоб отримане значення не перевищувало n-1 за допомогою взяття залишку від ділення.

Перемішуюча машина

Цей автомат уже набагато складніше. Він моделює хімічні реакції, у яких дві різних речовини реагують за допомогою каталізатора, наприклад знамениту реакцію

Білоусова-Жаботинського. У цій реакції взаємодіючі речовини утворять складні хвилеподібні структури (див. рис. 15.10).

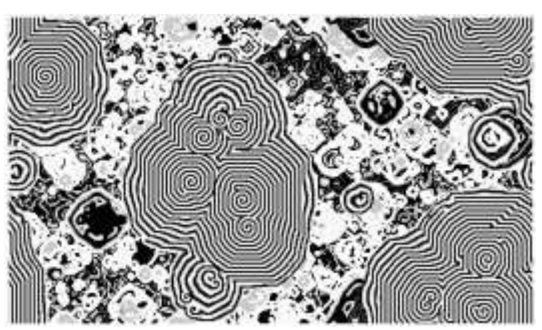


Рисунок 15.10 – Клітинний автомат «Перемішуюча машина»

Приймемо за $n + 1$ кількість станів клітини в КА. Клітину в стані 0 будемо вважати "здоровою", у стані n – хворий. Проміжні стани відображають ступінь зараженості клітини – чим ближче до n, тим сильніше клітина хвора.

Якщо клітина здорова, то на наступному кроці вона перейде в стан, що залежить від стану навколишніх її сусідів. Нехай A – число заражених клітин навколо поточної клітини, B – число хворих клітин навколо її. Тоді наступний стан даної клітини визначиться так: $A/k1 + B/k2$. Причому розподіл тут цілочисельний, тобто дробова частина відкидається. Результат не повинен перевищувати n.

Якщо ж клітина заражена, її стан обчислюється так: $S/(A + 1) + g$, де S – сума станів самої клітини й всіх її сусідів, A – кількість заражених сусідів, g – швидкість поширення інфекції серед заражених клітин. Точно так само результат не повинен перевищувати n.

Ну і якщо клітина хвора, на наступному кроці вона видужує (переходить у стан 0).

Приклад псевдокоду:

```
Функція Правило(i, j)
 Початок
    im1 = ((i - 1) + size) mod size
    ip1 = (i + 1) mod size
    jm1 = ((j - 1) + size) mod size
    jp1 = (j + 1) mod size
    a = КількістьЗараженихКлітин(i, j)
    b = КількістьХворихКлітин(i, j)
    s = СумаВсіхСусідніх(i, j)
    якщо old[i, j] == N то
        повернути 0
    інакше якщо old[i, j]=0 то
        повернути min((a div k1) + (b div k2),N)
    інакше
        повернути min((s div (a + 1)) + g, N)
 Кінець
```

Спочатку порахуємо кількість заражених і хворих клітин в околиці, а також суму станів всіх сусідів і самої клітини. Далі, перевіряємо, по яких формулах обчислювати наступний стан клітини. У формулах замість того, щоб перевіряти, чи не перевищує результат n, скористаємося функцією min, позбуваючись у такий спосіб від зайвого умовного оператора.

Можна також оптимізувати даний код. Адже якщо поточний стан клітини дорівнює n, нема рації обчислювати змінні a, b і s.

За замовчуванням значення параметрів такі: $n = 100$, $k1 = 2$, $k2 = 3$, $g = 30$.

Довільні клітинні автомати

Чи можна зробити так, щоб не перекомпілюючи програму, додавати нові клітинні автомати? Виявляється можна. Правда для автоматів з невеликою кількістю станів.

Кількість станів кожної клітини дорівнює n. Отже, всіх можливих варіантів станів самої клітини й всіх її сусідів дорівнює 9^n для околу фон Неймана й 5^n для околу Мура.

Клітинний автомат “Wire World”

З його допомогою можна моделювати цифрові електронні схеми й теоретично на його основі можна створити комп'ютер будь-якої потужності (як відомо, будь-який комп'ютер можна зібрати, використовуючи лише логічні схеми тільки одного типу, наприклад “логічне і-не”).

Кожна клітина цього автомата може перебувати в одному із чотирьох станів: фон, провідник, голова електрона, хвіст електрона. Голова й хвіст електрона, розташовані поруч, утворять електрон, що рухається по провіднику в напрямку від хвоста до голови. Правила автомата прості:

1. На наступному кроці клітина голови електрона переходить у клітину хвоста електрона.
2. Клітина хвоста електрона стає клітиною провідника.
3. Клітина фону завжди залишається без змін.
4. Клітина провідника, навколо якої тільки 1 або 2 голови електрона, стає клітиною голови електрона.

Щоб почати конструювання електронних схем, на клітинах фону потрібно намалювати клітини провідника, по яких будуть рухатися електрони, а потім увести електрон. Після запуску автомата електрон буде рухатися уздовж провідника зліва на право.

Таким чином, логічний “0” буде зображуватися відсутністю електрона, а “1” – його наявністю.

Одномірні клітинні автомати

В даних автоматах в клітини є тільки сусіди ліворуч і праворуч і всі клітини вибудувані в одну лінію. Звичайно, якщо стан клітин буде просто мінятися від кроку до кроку, нічого цікавого ви не побачите, тому слід “розгортати” цей рядок клітин у часі. Стан КА в початковий момент часу намалюємо зверху, у наступний момент – трохи нижче й так далі. Одержимо діаграму розвитку КА в часі в напрямку зверху вниз (рис. 15.11).

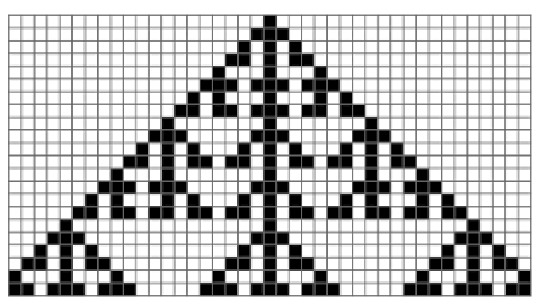


Рисунок 15.11 – Одномірні клітинні автомати (з розгортанням в часі)

Нехай кількість станів кожної клітини одномірного КА дорівнює *n*, а радіус околиці – *r* (тобто кількість сусідніх клітин ліворуч або праворуч. Наприклад для *r* = 1 у клітини два сусіди – ліворуч і праворуч, при *r* = 2 у клітини вже 4 сусіда – 2 праворуч і 2 ліворуч). Наступний стан клітини залежить від стану самої клітини й стану її сусідів.

Кількість можливих клітинних автоматів колосально. Спробуємо обмежити його. Візьмемо автомати, у яких стан клітини залежить від суми станів самої клітини й клітин

околиці (такі КА називають тоталістичними). Найбільше значення цієї суми $s = n \cdot (r \cdot 2 - 1) - 1$. Тоді опис самого КА можна звести до одномірної таблиці. Дуже цікавий автомат виходить, наприклад, для $n = 2$ та $r = 2$ (s змінюється від 0 до 5) і такій таблиці правил (рис. 15.12):

Сума	**5**	**4**	**3**	**2**	**1**	**0**
Наступний стан	0	1	0	1	0	0

Рисунок 15.12 – Таблиця станів клітинного автомату

Помітьте, що число у двійковій системі, записане в другому рядку – це 20. Як і у двовимірних клітинних автоматах, наприклад, грі "Життя", у такому автоматі можуть існувати конфігурації клітин, які постійно рухаються в одному напрямку – планери. Для автомата "20" – це комбінації 10111011і 1001111011, вони рухаються вправо, якщо їх розгорнути навпаки, вони будуть рухатися вліво.

Є ще декілька цікавих КА, наприклад автомат з $n = 2$, $r = 3$, кодовий номер 88. У ньому є катапульта, що через кожні 238 кроків стріляє планерами вліво й вправо).

Початкова конфігурація катапульти така: 111111111101.

Питання до розділу:

1. Що таке клининні автомати?
2. Що таке гра «Життя»?
3. Які правила у гри життя?
4. Для чого потрібні клітинні автомати?
5. Наведіть приклади різних клітинних автоматів.

Список літератури

1. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений / В.Е. Алексеев, В.А. Таланов. - М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2018. - 320 с.
2. Альфред В. Ахо Структуры данных и алгоритмы / Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. - М.: Вильямс, 2016. - 400 с.
3. Бабенко, М. А. Введение в теорию алгоритмов и структур данных / М.А. Бабенко. - М.: МЦНМО, 2016. - 243 с.
4. Белов, В. В. Алгоритмы и структуры данных. Учебник / В.В. Белов, В.И. Чистякова. - М.: КУРС, Инфра-М, 2016. - 240 с.
5. Борисенко В.В. Основы программирования Интернет-университет информационных технологий – ИНТУИТ.ру, 2005.
6. Вирт Алгоритмы и структуры данных / Вирт, Никлаус. - М.: СПб: Невский Диалект; Издание 2-е, испр., 2018. - 352 с.
7. Гагарина, Л.Г. Алгоритмы и структуры данных / Л.Г. Гагарина. - М.: Финансы и статистика, 2015. - 787 с.
8. Голицына О.Л., Попов И.И. Основы алгоритмизации и программирования: Учебное пособие. – М.: Форум: Инфра-М, 2004.
9. Гринченков, Д.В. Математическая логика и теория алгоритмов для программистов: Учебное пособие / Д.В. Гринченков, С.И. Потоцкий. - М.: КноРус, 2013. - 206 с.
10. Заковряшин А.И. Алгоритмизация и программи–рование вычислительных задач. – Санкт-Петербург, 2002 г.- 80 с.
11. Игошин В.И. Теория алгоритмов: Учебное пособие / В.И. Игошин. - М.: ИНФРА-М, 2013. - 318 с.
12. Канцедал, С.А. Алгоритмизация и программирование : Учебное пособие / С.А. Канцедал. - М.: ИД ФОРУМ, НИЦ ИНФРА-М, 2013. - 352 с.
13. Касьянов В.Н., Евстегнеев В.А. Графы в программировании: обработка, визуализация и программирование. – Санкт-Петербург:

БХВ-Петербург, 2003 . – 1104 с.

14. Кнут Д. Искусство программирования для ЭВМ, т. 1. Основные алгоритмы. – С.-П.: Вильямс, 2000.

15. Кнут Д. Искусство программирования для ЭВМ, т. 2. Получисленные алгоритмы. – С.-П.: Вильямс, 2000.

16. Кнут Д. Искусство программирования для ЭВМ, т. 3. Сортировка и поиск. – С.-П.: Вильямс, 2000.

17. Крупский, В.Н. Математическая логика и теория алгоритмов: Учебное пособие для студентов учреждений высшего проф. образования / В.Н. Крупский, В.Е. Плиско. - М.: ИЦ Академия, 2013. - 416 с.

18. Лавров С.С. Программирование. Математические основы, средства, теория. – Санкт-Петербург: БХВ-Петербург, 2001 г.- 320 с.

19. Лафоре Р. Структуры данных и алгоритмы в Java / Р. Лафоре. - М.: Питер, 2016. - 704 с.

20. Локазюк В.М. Надійність, помилки і тестування програмного забезпечення комп'ютерних пристроїв та систем: Навчальний посібник. – Хмельницький: ТУП, 2003. – 74 с.

21. Паронджанов, В. Учись писать, читать и понимать алгоритмы / В. Паронджанов. - М.: ДМК Пресс, 2016. - 458 с.

22. Непейвода Н.Н. Стили и методы программирования Интернет-университет информационных технологий – ИНТУИТ.ру, 2005.

23. Семакин И.Г. Основы алгоритмизации и программирования: Учебник для студ. учреждений сред. проф. образования / И.Г. Семакин, А.П. Шестаков. - М.: ИЦ Академия, 2013. - 304 с.

24. Скиена С.С. Алгоритмы. Руководство по разработке. – Санкт-Петербург: БХВ-Петербург, 2011 г.- 719 с.

25. Співаковський О. В., Осипова Н. В., Львов М. С., Бакуменко К. В. Основи алгоритмізації та програмування. Обчислювальний експеримент. Розв'язання проблем ефективності в алгоритмах пошуку та сортування: Навчальний почібник. – Херсон: Айлант. – 2010. – 100 с.: іл.

26. Терехов А.Н. Технология программирования БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий – ИНТУИТ.ру, 2007

27. Тюкачев, Н.А. C#. Алгоритмы и структуры данных / Н.А. Тюкачев. - М.: Лань, 2017. - 187 с.

28. Харченко В.С., Скляр В.В., Тарасюк О.М. Методы моделирования и оценки качества и надежности программного обеспечения. – Харьков: НАУ «ХАИ», 2004. – 159 с.

НАВЧАЛЬНЕ ВИДАННЯ

МЕЛЕШКО Єлизавета Владиславівна
ЯКИМЕНКО Микола Сергійович
ПОЛІЩУК Людмила Іванівна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник

Редактор Якименко М.С.
Технічний редактор Лисенко В.Ф.
Комп’ютерна графіка Поліщук Л.І.
Обкладинка Мелешко Є.В.
Комп'ютерний набір та верстка Мелешко Є.В.

На обкладинці навчального посібника використане зображення, розміщене на сайті freepik.com користувачем @fullvector, яке розповсюджується за ліцензією creative commons "free for commercial use with attribution"

Формат 60x84 1/16. Ум. др. арк. 9,06.
Облік. видав. арк. 4,18. Тираж 100. Зам. 524.

Видавець і виготовлювач СПД ФО Лисенко В.Ф.
25029, м. Кропивницький, вул. Пацаєва, 14, корп. 1, кв. 101
Свідоцтво суб’єкта видавничої справи ДК № 3904 від 22.10.2010
тел.: (0522) 322-326