Міністерство освіти і науки України
Запорізький національний університет

С.М. Гребенюк, О.В. Кудін, А.О. Лісняк, А.В. Столярова

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмна інженерія»

Затверджено
вченою радою ЗНУ
Протокол № __ від _____2022 р.

Запоріжжя
2022

УДК: 004.4+004.65] (075.8)
А456

Гребенюк С. М., Кудін О. В., Лісняк А. О., Столярова А. В. Алгоритми та структури даних : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмна інженерія». Запоріжжя : Запорізький національний університет, 2022. 128 с.

У навчальному посібнику в систематизованому вигляді подано програмний матеріал дисципліни «Алгоритми та структури даних». Викладено фундаментальні поняття алгоритму, структур даних, абстрактних типів даних. Розглянуто велику кількість методів розробки та аналізу алгоритмів на прикладі практичних задач. Основна увага приділяється засвоєнню знань з проєктування, розробки та аналізу алгоритмів. До кожного змістового модуля підібрано творчі завдання, запропоновано запитання та тести для самоперевірки, що сприятиме формуванню навичок самостійного аналізу наукової літератури, подальшій науково-дослідній та професійній діяльності.

Видання призначено для здобувачів ступеня вищої освіти бакалавра спеціальності «Інженерія програмної інженерії», які навчаються за освітньо-професійною програмою «Програмна інженерія».

Рецензент
С. І. Гоменюк, доктор технічних наук, професор, декан математичного факультету

Відповідальний за випуск
А. О. Лісняк, кандидат фізико-математичних наук, доцент, завідувач кафедри програмної інженерії

ЗМІСТ

ВСТУП

Поняття алгоритму та структур даних є фундаментальними у галузі інформаційних технологій. Ефективність численних програмних веб та мобільних застосунків залежить від ефективності алгоритмів, які закладено в основу програмного забезпечення. Використання оптимальних структур даних впливає на складність та час розробки. Отже, володіння фундаментальними алгоритмами обробки даних та розв'язання класичних задач є необхідною компетенцією кваліфікованого розробника програмного забезпечення. Цей факт підкреслює необхідність та актуальність вивчення дисципліни «Алгоритми та структури даних».

Курс «Алгоритми та структури даних» є обов'язковим для спеціальності «Інженерія програмного забезпечення». Його опанування передбачає здобуття необхідних знань та вмінь, є основою для вивчення таких дисциплін: основи програмної інженерії, бази даних та ін. Курс надає майбутньому фахівцю фундаментальні компетенції з проєктування та розробки алгоритмів розв'язання практичних задач.

Метою вивчення навчальної дисципліни «Алгоритми та структури даних» є набуття студентами знань про сучасні та ефективні структури даних та алгоритми комп'ютерного оброблення інформації, а також методи їх дослідження та аналізу.

Основними **завданнями** вивчення дисципліни «Алгоритми та структури даних» є: засвоєння базових понять теорії алгоритмів; оволодіння основними алгоритмами розв'язання задач пошуку та сортування даних, алгоритмами на графах; набуття практичних навичок роботи з базовими структурами даних та абстрактними типами даних; створення студентами складних програм різних типів з використанням складних структур даних і алгоритмів їх обробки.

Згідно з вимогами освітньо-професійної програми студенти повинні досягти таких результатів навчання:

знати:

– основні поняття теорії алгоритмів;
– підходи до аналізу алгоритмів;
– основні структури даних: масив, запис, файл;
– основні абстрактні типи даних: список, стек, черга, дерево;
– алгоритми сортування даних;
– алгоритми пошуку даних;
– алгоритми на графах;
– методи розробки алгоритмів;

вміти:

– розробляти алгоритми та проводити їх аналіз;
– застосовувати основні підходи до розробки алгоритмів: метод грубої сили, метод декомпозиції, метод перетворення, жадібні алгоритми;
– проводити аналіз розроблених алгоритмів;
– реалізовувати основні алгоритмічні структури мов програмування

високого рівня;

– реалізовувати парадигму структурного програмування засобами мов програмування високого рівня.

Посібник створено авторами на основі досвіду викладання алгоритмів та структур даних студентам спеціальності «Інженерія програмного забезпечення». Автори сподіваються, що запропоноване видання стане корисним здобувачам вищої освіти, які прагнуть отримати знання та набути навичок з алгоритмів та структур даних, а також викладачам для проведення лекційних та лабораторних занять, організації самостійної роботи студентів.

Тема 1 Поняття алгоритму та його властивості

Мета: ознайомитись з основними поняттями алгоритмізації. Розробити програмні реалізації алгоритмів визначення найбільшого спільного дільника та провести аналіз алгоритмів.

План

1. Інтуїтивне визначення алгоритму
2. Математичні визначення алгоритму
3. Схема розв'язання задач на ЕОМ
4. Важливі типи задач
5. Алгоритм обчислення найбільшого спільного дільника

1.1 Інтуїтивне визначення алгоритму

Алгоритм – це послідовність чітко певних інструкцій, призначених для розв'язання деякої задачі.

Алгоритм має такі властивості:

1. ***Дискретність*** – процес перетворення даних, тобто на кожному кроці алгоритму виконується чергова одна операція.
2. ***Результативність*** – алгоритм повинен давати деякий результат.
3. ***Закінченість*** – алгоритм повинен давати результат за скінченну кількість кроків.
4. ***Детермінованість*** – всі приписи алгоритму повинні бути однозначні, зрозумілі користувачеві.
5. ***Масовість*** – алгоритм повинен давати розв'язки для цілої групи задач із деякого класу, що відрізняються вихідними даними.
6. ***Введення*** – алгоритм має деяку кількість вхідних даних.
7. ***Виведення*** – в алгоритму є одне або декілька вихідних даних, тобто величин, що мають певний зв'язок із вхідними даними.

Алгоритмізація – це загальна послідовність дій, які необхідно виконати для побудови алгоритму розв'язання задачі, у тому числі – виділення конкретних кроків алгоритмічного процесу, визначення виду формального запису для кожного кроку й встановлення певного порядку виконання кожного із цих кроків.

1.2 Математичні визначення алгоритму

Описане вище визначення алгоритму називається інтуїтивним, тому що воно розраховано на людське розуміння. В 30-х роках минулого сторіччя, математикам стало зрозуміло, що необхідно уточнити поняття алгоритму для розв'язання власних проблем математики. Математичні визначення алгоритму

були отримані в середині 30-х років у роботах Д. Гільберта, К. Геделя, А. Черча, С. Кліні, Е. Поста, А. Тюрінга, А. Маркова. Було розроблено три типи моделей.

Перший тип моделей заснований на понятті *рекурсивної функції.*

Другий – на основі опису детермінованого пристрою, що працює по кроках і виконує на кожному кроці заздалегідь визначені операції з елементами пристрою й даними (машини Поста, Тюрінга).

Третій тип моделей пов'язаний з роботою зі словами в деякому фіксованому алфавіті, які за допомогою підстановок переходять в інші слова (нормальний алгоритм Маркова).

Визначення алгоритму за допомогою рекурсивних функцій було зроблено А. Черчем і С. Кліні. Воно достатньо складно й пов'язане з розділом математики – теорією обчислювальних функцій.

Математики Е. Пост (США) і А. Тюрінг (Англія) незалежно один від одного в 1936 році й майже в той же час, що й А. Черч, і С. Кліні спробували уточнити поняття алгоритму, а потім за його допомогою визначити точно й клас обчислювальних функцій. Основна ідея Е. Поста й А. Тюрінга полягала у тому, що алгоритмічні процеси – це процеси, які може виконувати спеціально побудована «машина» (пристрій, автомат). Відповідно до цієї ідеї ними були описані в точних математичних термінах досить вузькі класи машин, але на цих машинах виявилося можливим здійснити (промоделювати) всі алгоритмічні процеси, які на той час коли-небудь, описувалися математиками. Моделі, представлені цими машинами, було запропоновано розглядати як математичні визначення взагалі всіх алгоритмів.

Машинами ці математичні моделі називаються тому, що при побудові використаються деякі поняття реальних електронно-обчислювальних машин – пам'ять, команда, програма.

Машина Поста

Машина Поста складається з необмеженої в обидва боки стрічки, розділеної на осередки, які послідовно пронумеровані цілими числами, як додатними, так і від'ємними. Стрічка відіграє роль пам'яті. У кожному осередку стрічки знаходиться або ознака того, що в осередку присутня мітка, або осередок порожній. Стан стрічки – це дані про те, які осередки зайняті, а які порожні.

Крім стрічки є головка зчитування\запису, що:

- уміє рухатися вперед, назад і стояти на місці;
- уміє читати вміст, стирати й записувати мітку;
- управляється програмою, у яку можуть входити в будь-якій комбінації й будь-якій кількості шість команд:
    1) праворуч;
    2) ліворуч;
    3) поставити мітку;
    4) стерти мітку;
    5) передача керування на один номер команди в програмі, якщо в поточному осередку є мітка, якщо мітки немає – то передача керування на інший номер команди;
    6) припинення роботи.

Стан машини – це стан стрічки й положення голівки зчитування\запису.

Машина Поста, незважаючи на зовнішню простоту, може робити різні обчислення, для чого треба задати початковий стан машини й програму, що ці обчислення зробить.

Машина вирішує таку проблему: якщо для розв'язання задачі можна побудувати машину Поста, то вона алгоритмічно розв'язна, тобто для цієї задачі побудований алгоритм.

Машина Поста – це модель комп'ютера.

Машина Тюрінга

Машина Тюрінга складається з необмеженої в обидва боки стрічки, розділеної на осередки, які послідовно пронумеровані цілими числами, як додатними, так і від'ємними.

У кожному осередку стрічки може стояти будь-який символ із заданого алфавіту, до якого введений «порожній» символ – ознака того, що осередок порожній.

Машина має скінченну множину внутрішніх станів, початковий (з нього починається робота машини) і кінцевий стан, потрапивши в який, машина припиняє роботу.

Крім стрічки є голівка зчитування\запису, що, по-перше, уміє рухатися вперед, назад й стояти на місці, по-друге, вміє зчитувати вміст, стирати й записувати символи з даного алфавіту; по-третє, управляється програмою.

Програма являє собою таблицю, у якій у кожній клітці записана команда. Кожна клітка визначається двома параметрами – символом алфавіту й станом машини. Команда являє собою вказівку, куди пересунути головку зчитування\запису з поточного стану, який символ записати в поточний осередок і в який стан перейде машина (рис. 1.1).

Машина Тюрінга, як і машина Поста – це модель комп'ютера.

Машина Тюрінга, аналогічно машині Поста, вирішує таку проблему: якщо для розв'язання задачі можна побудувати машину Тюрінга, то вона алгоритмічно розв'язна.

І машина Тюрінга, й машина Поста еквівалентні за своїми можливостями.

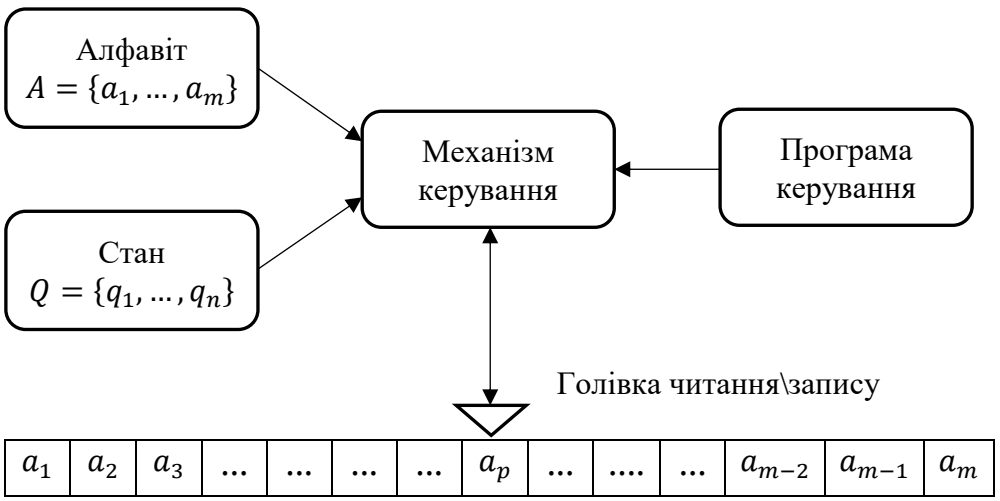


Рисунок 1.1 – Схематичне зображення машини Тюрінга

Основні відмінності машини Поста й машини Тюрінга суто технічні. Доведено їхню еквівалентність.

Нормальний алгоритм Маркова

Нормальний алгоритм Маркова задається алфавітом *A* й нормальною схемою підстановок.

Алфавіт – скінченна, непуста множина елементів, що називаються буквами. Різні комбінації букв утворять слова.

Нормальна схема підстановок – це скінченний набір, що складається з пар слів, де ліве слово переходить у праве (але не навпаки).

Нормальним алгоритмом в алфавіті *A* називається такий алгоритм побудови послідовності слів: як початкове слово береться саме слово *P* і до нього застосовують одну за одною кожну пару із схеми підстановок. Якщо підстановка можлива, то її здійснюють і починають підстановки спочатку. Якщо процес обривається (немає ні однієї допустимої підстановки) на слові *Q* або приходить у кінцеву підстановку, то даний нормальний алгоритм перетворив *P* в *Q*.

Якщо є задача: від *P* перейти до *Q* і доведено, що не можна побудувати нормальну схему, то має місце алгоритмічно нерозв'язна задача.

Алгоритмічно нерозв'язні задачі

Однієї із причин, що змусили математиків зайнятися питаннями формалізації поняття алгоритму, була необхідність одержати спосіб, що дозволяє визначити для деяких задач наявність або відсутність алгоритму для їхнього розв'язання.

Якщо для задачі побудований алгоритм – задача алгоритмічно розв'язна. А якщо не побудований, то є два варіанти:

– алгоритм поки не побудований;

– алгоритм неможливо побудувати, і задача називається алгоритмічно нерозв'язною.

1.3 Схема розв'язання задач на ЕОМ

Алгоритми можна вважати процедурним розв'язанням задач. Причому ці розв'язання є не стільки відповідями, скільки точно певними інструкціями для одержання відповідей. Саме цей акцент на точності визначення конструктивних процедур відрізняє інформатику від інших дисциплін, зокрема від теоретичної математики.

Перелічимо й коротко опишемо послідовність етапів проєктування й аналізу алгоритмів.

1. Розуміння задачі.

Існує кілька типів задач, які при створенні комп'ютерних додатків зустрічаються найчастіше. Виходові дані, що оброблюються алгоритмом, визначають екземпляр задачі (problem's instance), розв'язуваної за допомогою обраного алгоритму. При цьому дуже важливо точно вказати межі вхідних даних, які повинні враховуватися в алгоритмі.

2. Визначення можливостей обчислювального пристрою.

Повністю усвідомивши суть поставленої задачі, необхідно оцінити можливості обчислювального пристрою, для якого створюється алгоритм. Переважна більшість сучасних алгоритмів призначено для створення на їхній основі програми, що працює на комп'ютері, що сильно нагадує за структурою машину фон Неймана. Суть цієї архітектури відображено у її назві – машина з довільним доступом (random-access machine, або RAM). Передбачалося, що команди повинні послідовно вибиратися з пам'яті й виконуватися спеціальним пристроєм, що називається центральним процесором, причому в кожний момент часу в машині Неймана може виконуватися тільки одна команда. Тому алгоритми, розроблені для виконання на такій машині, назвали послідовними (sequential algorithms).

Поява машини Неймана не зупинила прогрес в області обчислювальної техніки. Незабаром були придумані комп'ютери, які могли одночасно (тобто паралельно) виконувати кілька команд. Тому алгоритми, розроблені для таких машин, назвали паралельними (parallel algorithms).

3. Вибір між точним або наближеним методом розв'язання задачі.

Наступне принципове питання – вибір точного або наближеного методу розв'язання задачі. У першому випадку алгоритм називається точним (exact algorithm), а в другому – наближеним (approximation algorithm). Перевага при такому виборі безперечно на боці точних алгоритмів, але існує ряд ситуацій, коли наближені алгоритми є більш затребувані. По-перше, існують задачі, які не можна розв'язати точно. Як приклад, можна привести видобуття квадратного кореня, розв'язання нелінійних рівнянь або обчислення визначених інтегралів. По-друге, існуючі алгоритми точного розв'язання задачі можуть бути неприпустимо повільними, якщо її складність досить висока. Найбільш відомою з таких задач є задача комівояжера (traveling salesman problem), що полягає в пошуку найкоротшого маршруту між *n* містами.

4. Вибір відповідних структур даних.

У деяких алгоритмах не потрібно, щоб Виходові дані були подані в якомусь специфічному форматі. Однак так буває не завжди, більше того, для роботи багатьох алгоритмів потрібні зовсім певні структури даних.

5. Методи проєктування алгоритмів.

Метод проєктування алгоритму (algorithm design technique) (або "стратегія", або "принцип") – це універсальний підхід, що застосовується для алгоритмічного розв'язання широкого кола задач, що виникають у різних областях обчислювальної техніки.

6. Методи подання алгоритмів.

Словесний метод – алгоритм описується словами (тобто у вільній формі у вигляді послідовності виконуваних дій).

Псевдокод (pseudocode) – алгоритм описується сумішшю однієї із природних мов і конструкцій, характерних для мови програмування.

Блок-схема – алгоритм описується за допомогою спеціальних графічних елементів, які позначають операції, потік, дані тощо.

7. Оцінка коректності алгоритму.

Після подання алгоритму в будь-якій формі, необхідно оцінити його коректність (correctness). Це означає, що необхідно довести, що обраний алгоритм за обмежений проміжок часу видає необхідний результат для будь-яких коректних значень вихідних даних.

8. Аналіз алгоритму.

Зазвичай розробники намагаються, щоб алгоритми задовільняли деяким вимогам. Після перевірки коректності алгоритму однією з найважливіших характеристик є оцінка його ефективності. На практиці існує два види оцінки ефективності алгоритму: часова й просторова.

Часова ефективність (time efficiency) є індикатором швидкості роботи алгоритму. Що стосується *просторової ефективності* (space efficiency), то ця оцінка показує кількість додаткової оперативної пам'яті, необхідної для роботи алгоритму.

Ще однією важливою характеристикою алгоритму є його *простота* (simplicity). На відміну від ефективності, яку можна точно визначити й оцінити за допомогою строгих математичних співвідношень, простота алгоритму – поняття дещо суб'єктивне, тому виробити об'єктивні критерії її оцінки досить непросто.

Наступною важливою характеристикою алгоритму є його *загальність*, або *універсальність* (generality). По суті, тут можна виділити два моменти: загальність задачі, для розв'язання якої розроблений алгоритм, і діапазон припустимих значень його вихідних даних.

1.4 Важливі типи задач

Серед величезної кількості задач, що зустрічаються в обчислювальній техніці, можна виділити кілька типів, яким дослідники завжди приділяли особливу увагу. Подібний інтерес викликаний або практичним значенням задачі, або якимись її властивостями, що представляють особливий інтерес для дослідження.

До таких задач можна віднести найбільш важливі типи задач:

- сортування;
- пошук;
- обробка рядків;
- задачі з теорії графів;
- комбінаторні задачі;
- геометричні задачі;
- чисельні задачі.

Сортування. Задача сортування (sorting problem) полягає в упорядкуванні заданого списку яких-небудь елементів у зростаючому (спадаючому) по*рядку.*

Пошук. Задача пошуку пов'язана зі знаходженням заданого значення, яке зветься ключем пошуку (search key), серед заданої множини (або мультимножини).

Обробка рядків. У зв'язку зі швидким збільшенням останнім часом кількості додатків, пов'язаних з обробкою нечислових даних, інтерес фахівців усе більше викликають алгоритми обробки рядків. Рядком (string) називається послідовність символів, узятих із заздалегідь визначеного алфавіту. Практичний інтерес представляють, наприклад, текстові рядки, що складаються з букв, цифр і спеціальних символів; бітові рядки, що складаються з нулів і одиниць; послідовності генів, які можуть бути змодельовані за допомогою рядків символів, узятих із чотирьохсимвольного алфавіту {A, C, G, T}.

Проте варто відзначити, що алгоритми обробки рядків стали важливі для обчислювальної техніки дуже давно – з тих пір, як з'явилися перші мови програмування й відповідні їм програми – компілятори.

Існує одна специфічна задача, що привернула особливу увагу фахівців з інформатики. Мова йде про пошук заданого слова в рядку тексту. Її назвали пошуком підрядків (string matching). Для виконання такого специфічного пошуку розроблено кілька алгоритмів.

Задачі з теорії графів. Однієї із самих давніх і, мабуть, найцікавіших областей алгоритмики є обробка графів. Граф можна визначити як набір точок (вершинами), частина з яких з'єднана відрізками, які називаються ребрами.

Графи є досить цікавим об'єктом для вивчення як з теоретичної, так і із практичної точок зору. За допомогою графів можна змоделювати досить велику кількість процесів, що відбуваються в реальному житті, наприклад функціонування транспортних і комунікаційних мереж, календарне планування проєкту й т.д.

Комбінаторні задачі. Сутність цих задач в остаточному підсумку зводиться до знаходження такого комбінаторного об'єкта, як перестановка, комбінація або підмножина, який би задовольняв певним обмеженням і мав задані властивості (наприклад, дозволяв максимізувати прибуток або мінімізувати витрати).

Загалом кажучи, комбінаторні задачі відносять до класу самих складних як з теоретичної, так і із практичної точок зору. Їхня складність обумовлена наступним. По-перше, кількість комбінаторних об'єктів зазвичай дуже швидко зростає при збільшенні масштабу задачі й досягає неуявних значень вже при досить скромних її масштабах. По-друге, поки не існує алгоритмів для пошуку точного розв'язання подібних задач за прийнятний час.

Геометричні задачі. Геометричні алгоритми пов'язані з такими геометричними об'єктами, як точки, лінії, багатокутники.

Чисельні задачі. Чисельні задачі відносяться до ще однієї досить великої області практичного використання алгоритмів. Вони мають справу з математичними об'єктами, які по своїй суті є неперервними. Приклади типових чисельних задач такі: розв'язання рівнянь і систем рівнянь, обчислення визначених інтегралів і значень функцій і т.д. Переважна більшість таких задач може бути розв'язана тільки наближено. Ще одні принципові труднощі полягають у тому, що при розв'язанні подібних задач зазвичай необхідно виконувати операції з дійсними числами, які в комп'ютері можуть бути представлені лише з певною похибкою.

1.5 Алгоритм обчислення найбільшого спільного дільника

Для двох цілих чисел a і b, усяке число, що ділить без остачі як a, так і b, називається **спільним дільником**.

Найбільшим спільним дільником (НСД) для двох цілих чисел m і n називається найбільший з їхніх спільних дільників.

Як приклади, що ілюструють поняття алгоритму, ми розглянемо два способи розв'язання задачі пошуку НСД двох цілих чисел. Ці приклади допоможуть нам проілюструвати перераховані нижче **важливі властивості алгоритмів**:

– кожний крок алгоритму повинен бути чітко й однозначно визначений (ця вимога є обов'язковою і не повинна порушуватися ні за яких умов);

– повинні бути точно зазначені діапазони припустимих значень вхідних даних, які обробляються за допомогою алгоритму;

– для рішення однієї й тієї ж задачі може існувати кілька різних алгоритмів;

– в основу алгоритмів, для рішення однієї й тієї ж задачі, можуть бути покладені зовсім різні принципи, що може істотно вплинути на швидкість рішення цієї задачі.

Обчислення НСД чисел m і n за допомогою алгоритму Евкліда:

Крок 1. Якщо $n = 0$, повернути m як відповідь і закінчити роботу; інакше перейти до кроку 2.

Крок 2. Поділити без остачі m на n і привласнити значення остачі змінній r.

Крок 3. Привласнити значення n змінній m, а значення r – змінній n. Перейти до кроку 1.

Нижче наведено алгоритм Евкліда у вигляді псевдокоду.

Алгоритм $Euclid(m, n)$
// Алгоритм Евкліда обчислює значення функції $gcd(m, n)$
// Вхідові дані: два натуральних числа m і n
// Числа m та n не можуть одночасно дорівнювати нулю
// Виходові дані: найбільший спільний дільник чисел m і n
while *n!* = 0 **do**
  r = m mod n
  m = n
  n = r
return *m*

Обчислення НСД чисел m і n методом послідовного перебору:

Крок 1. Привласнити значення функції $\min(m, n)$ змінній t.

Крок 2. Розділити m на t. Якщо остача дорівнює нулю, перейти до кроку 3; інакше перейти до кроку 4.

Крок 3. Розділити n на t. Якщо остача дорівнює нулю, повернути t як

відповідь і закінчити роботу; інакше перейти до кроку 4.

Крок 4. Відняти 1 з t. Перейти до кроку 2.

Практичні завдання

1. Реалізувати алгоритм Евкліда.
2. Реалізувати алгоритм послідовного перебору.
3. Протестувати обидва алгоритми для різних вхідних даних.
4. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - вибір алгоритму для обчислення НСД;
   - виводити запрошення на введення чисел m і n;
   - виводити значення НСД.
5. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду обох алгоритмів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

Тестові завдання

1. Що не є прикладом математичного визначення алгоритму?
   а) машина Тюринга;
   б) машина Поста;
   в) алгоритм Маркова;
   г) ланцюг Маркова.
2. Властивість алгоритму, що всі приписи алгоритму повинні бути однозначні, зрозумілі користувачеві, це:
   а) дискретність;
   б) детермінованість;
   в) масовість;
   г) результативність.
3. _____ – це універсальний підхід, що застосовується для алгоритмічного розв'язання широкого кола задач, що виникають у різних областях обчислювальної техніки:
   а) метод проєктування алгоритмів;
   б) метод розробки програмної реалізації алгоритмів;
   в) схема розв'язання задач на ЕОМ;
   г) метод декомпозиції.
4. Який тип задач зводиться до знаходження такого об'єкта, як перестановка, підмножина тощо, який би задовольняв певним обмеженням і мав задані властивості?
   а) сортування;
   б) пошук;
   в) комбінаторні задачі;

г) задачі на графах.

5. Математичне визначення алгоритму, яке використовує модель пов’язану з роботою зі словами в деякому фіксованому алфавіті, які за допомогою підстановок переходять в інші слова

а) машина Тюринга;

б) машина Поста;

в) рекурсивне визначення;

г) алгоритм Маркова.

Контрольні запитання

1. Дайте визначення поняття алгоритму і його основних властивостей.
2. Які Ви знаєте етапи розв’язання задачі на ЕОМ?
3. Які основні типи задач вивчаються в курсі «Алгоритми та структури даних»?
4. Який із двох реалізованих алгоритмів знаходження НСД є найбільш ефективним та чому?
5. Спробуйте розробити альтернативні алгоритми пошуку найбільшого спільного дільника.

Тема 2 Основи аналізу ефективності алгоритмів

Мета: ознайомитись математичний і емпіричний аналіз рекурсивного алгоритму; розробити рекурсивний алгоритм і програму розв'язання задачі про ханойські вежі.

План

1. Поняття аналізу алгоритмів, оцінка розміру вихідних даних
2. Одиниці виміру часу виконання алгоритму
3. Порядок зростання
4. Асимптотичний аналіз алгоритмів
5. Емпіричний аналіз алгоритмів
6. Математичний аналіз алгоритмів
7. Методи розв'язання рекурентних співвідношень

2.1 Поняття аналізу алгоритмів. Оцінка розміру вихідних даних

Термін «аналіз алгоритмів» означає процес дослідження ефективності алгоритмів, яку можна оцінити за двома параметрами: часу виконання алгоритму й необхідного об'єму оперативної пам'яті. Почнемо з розгляду двох видів ефективності: часової й просторової. *Часова ефективність* (time efficiency) є індикатором швидкості роботи алгоритму. *Просторова ефективність* (space efficiency) показує, скільки додаткової оперативної пам'яті необхідно для роботи алгоритму.

Час виконання більшості алгоритмів прямо залежить від розміру вихідних даних (тобто чим більше розмір, тим довше працює алгоритм). Наприклад, досить довго триває процес сортування великих масивів даних, перемножування великих матриць і т.п. Тому цілком логічно описати ефективність алгоритму у вигляді функції від деякого параметра n, пов'язаного з розміром вхідних даних. У більшості випадків вибрати такий параметр не представляє великої складності. Наприклад, для задач, пов'язаних із сортуванням, пошуком, знаходженням найменшого елемента в списку й багатьох інших, пов'язаних з обробкою списків, таким параметром буде розмір списку. Для задачі обчислення значення полінома степеня n таким параметром може бути степінь полінома або кількість його коефіцієнтів, що на одиницю більше степеня.

На вибір належної системи вимірів розміру задачі можуть вплинути виконувані розглянутим алгоритмом дії. Наприклад, як оцінити розмір вхідних даних для алгоритму, що виконує перевірку орфографії? Якщо в алгоритмі перевіряється кожний символ, що вводиться, то оцінити розмір вхідних даних можна, підрахувавши кількість символів у вхідному потоці. Якщо ж в алгоритмі відбувається обробка тексту за словами, то потрібно підрахувати кількість слів у вхідному потоці.

Необхідно зробити спеціальне зауваження із приводу оцінки розміру вихідних даних для алгоритмів, пов'язаних зі знаходженням чисел, що задовольняють певним умовам (наприклад, що перевіряють, чи є задане ціле число n простим). Для подібних алгоритмів кібернетики воліють оцінювати розмір вхідних даних за кількістю бітів b у двійковому поданні числа n:

$$b = \lfloor \log_2 n \rfloor + 1.$$

2.2 Одиниці виміру часу виконання алгоритму

Необхідно розглянути ще одне питання, що стосується одиниць виміру часу виконання алгоритму. Безумовно, для цієї мети можна просто скористатися загальноприйнятими одиницями виміру часу – секундою, мілісекундою й т.д. і з їхньою допомогою оцінити час виконання програми, що реалізує розглянутий алгоритм. Однак у такого підходу існують явні недоліки. Час виконання програми залежить від:

– швидкодії конкретного комп'ютера;
– якості реалізації алгоритму у вигляді програми;
– типу компілятора, використаного для генерації машинного коду;
– точності хронометрування реального часу виконання програми.

Оскільки необхідно вирішити проблему виміру ефективності алгоритму, а не його програми, скористаємося такою системою вимірів, яка б не залежала від наведених вище сторонніх факторів.

Один з можливих способів рішення цієї проблеми – в підрахунку того, скільки разів виконується кожна операція алгоритму. Однак подібний підхід занадто складний і найчастіше не потрібний. Тому необхідно скласти список найбільш важливих операцій, виконуваних в алгоритмі, які називаються *основними*, або *базовими операціями* (basic operation), визначити, які з них вносять найбільший вклад у загальний час виконання алгоритму, і обчислити, скільки разів ці операції виконуються.

Як правило, скласти список основних операцій алгоритму зовсім неважко. Звичайно в нього включають найбільш тривалі за часом операції, виконувані у внутрішньому циклі алгоритму. Наприклад, у більшості алгоритмів сортування використається метод порівняння двох елементів (ключів) списку, що сортується. Для подібного типу алгоритмів основною є операція порівняння ключів. Як ще один приклад розглянемо алгоритми множення матриць і обчислення значення полінома. У них використаються дві основні операції: множення й додавання. На більшості комп'ютерів команда множення двох цілих чисел виконується набагато довше, ніж додавання. Тому вона є безумовним кандидатом на включення в список основних операцій.

Припустимо, що c_{op} – час виконання основної операції алгоритму на конкретному комп`ютері, а $C(n)$ – кількість разів, які ця операція повинна бути виконана при роботі алгоритму. Тоді час виконання програмної реалізації цього

алгоритму на даному комп'ютері $T(n)$ можна приблизно визначити за такою формулою:

$$T(n) \approx c_{op}\, C(n).$$

У процесі аналізу ефективності при досить великих розмірах вихідних даних не враховують постійні множники, а зосереджують увагу на оцінці *порядку зростання* (order of growth) кількості основних операцій з точністю до постійного множника.

2.3 Порядок зростання

При малих розмірах вхідних даних неможливо помітити різницю в часі виконання між ефективним і неефективним алгоритмом.

Для великих значень *n* обчислюють порядок зростання функції. У табл. 2.1 ці значення наведені для деяких функцій, що відіграють особливу роль у процесі аналізу алгоритмів.

Таблиця 2.1 – Порядок зростання функцій

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Як видно з таблиці, найменший порядок зростання має логарифмічна функція. Причому його значення настільки мале, що програми, що реалізують алгоритми з логарифмічною кількістю основних операцій, будуть виконуватися практично миттєво для всіх діапазонів вихідних даних реального розміру. Зверніть також увагу, що, хоча деякі значення при таких обчисленнях, природно, будуть залежати від основи логарифма, наведена нижче формула дозволяє легко переходити від однієї основи логарифма до іншої, зберігаючи при цьому логарифмічну залежність обчислень (при цьому використаються нові постійні множники):

$$\log_a n = \log_a b \log_b n.$$

Тому у випадку, коли потрібно тільки визначити порядок зростання кількості основних операцій алгоритму з точністю до постійного множника, будемо опускати основу логарифма й записувати просто: $\log n$.

За допомогою алгоритмів, у яких кількість виконуваних операцій росте за

експонентним законом, можна розв'язувати лише задачі дуже малих розмірів.

Існує ще один спосіб оцінки якісного розходження в порядку зростання функцій, наведеному в табл. 1. Необхідно розглянути реакцію функції на, скажемо, дворазове збільшення значення параметра n.

2.4 Асимптотичний аналіз алгоритмів

2.4.1 Асимптотичні позначення й основні класи ефективності

При викладанні основ аналізу ефективності алгоритмів основна увага зосереджувалась на порядку зростання кількості базових операцій алгоритму, що використовується як основний критерій ефективності алгоритму. Для того щоб можна було порівнювати між собою ці порядки зростання й класифікувати їх, було введено три умовні позначки: O (прописна грецька «О»), Θ (прописна грецьку «тета») і Ω (прописна грецька «омега»). Для початку дамо нестрогий опис цих понять, а потім, розглянувши кілька прикладів, приведемо їхнє строге визначення.

В наведеному нижче описі $t(n)$ й $g(n)$ можуть бути будь-якими невід'ємними функціями, визначеними на множині натуральних чисел. Через $t(n)$ позначимо час виконання алгоритму. Зазвичай він виражається у вигляді числа основних операцій алгоритму, позначуваних через $C(n)$. Під $g(n)$ будемо розуміти деяку просту функцію, з якої буде проводитися порівняння кількості операцій.

Сенс позначення $O\big(g(n)\big)$ – це множина всіх функцій, порядок зростання яких при досить великих n не перевищує (тобто менше або дорівнює) деякої константи, помноженої на значення функції $g(n)$. Нижче кілька прикладів – всі наведені твердження справедливі:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad n(n-1)/2 \in O(n^2);$$
$$n^3 \notin O(n^2), \qquad 0{,}00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

Друге позначення, $\Omega\big(g(n)\big)$ – це множина всіх функцій, порядок зростання яких при досить великих n не менше (тобто більше або дорівнює) деякої константи, помноженої на значення функції $g(n)$. Наприклад:

$$n^3 \in \Omega(n^2), \qquad n(n-1)/2 \in \Omega(n^2), \qquad 100n + 5 \notin \Omega(n^2).$$

Позначення $\Theta\big(g(n)\big)$ – це множина всіх функцій, порядок зростання яких при досить великих n дорівнює деякій константі, помноженої на значення функції $g(n)$.

O-позначення

Говорять, що функція $t(n)$ належить множині $O\big(g(n)\big)$, що записується як $t(n) \in O\big(g(n)\big)$, якщо для всіх великих значень n функція $t(n)$ обмежена зверху

деякою константою, помноженої на значення функції $g(n)$, тобто якщо існує додатна константа c й невід'ємне ціле число n_0 таке, що $t(n) \leq cg(n), \forall n \geq n_0$.

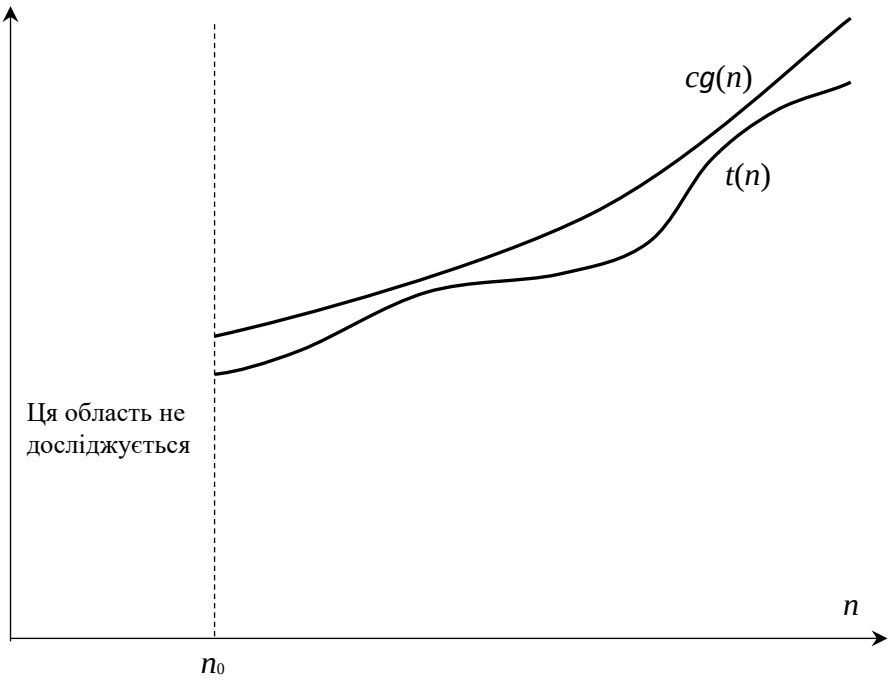


Рисунок 2.1 – Ілюстрація O-позначення: $t(n) \in O\big(g(n)\big)$

Ω-позначення

Говорять, що функція $t(n)$ належить множині $\Omega\big(g(n)\big)$, що записується як $t(n) \in \Omega\big(g(n)\big)$, якщо для всіх великих значень n функція $t(n)$ обмежена знизу деякою константою, помноженої на значення функції $g(n)$, тобто якщо існує додатна константа c й невід'ємне ціле число n_0 таке, що $t(n) \geq cg(n), \forall n \geq n_0$.

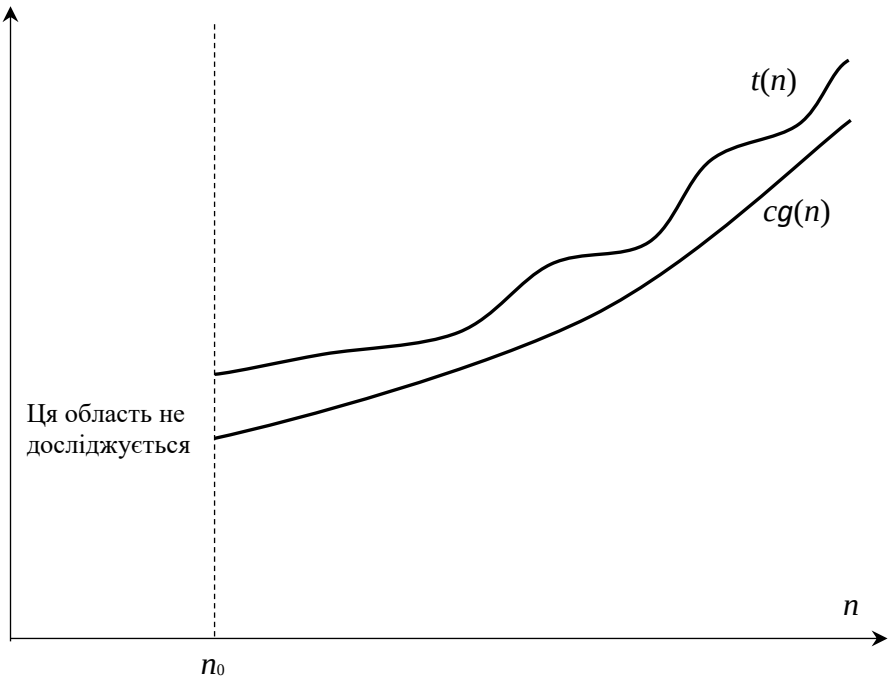


Рисунок 2.2 – Ілюстрація Ω-позначення: $t(n) \in \Omega\big(g(n)\big)$

Θ-позначення

Говорять, що функція $t(n)$ належить множині $\Theta\big(g(n)\big)$, що записується як $t(n) \in \Theta\big(g(n)\big)$, якщо для всіх великих значень n функція $t(n)$ обмежена знизу й зверху деякими константами, помноженими на значення функції $g(n)$, тобто якщо існують додатні константа c_1, c_2 й невід'ємне ціле число n_0 таке, що $c_2 g(n) \le t(n) \le c_1 g(n), \forall n \ge n_0$.

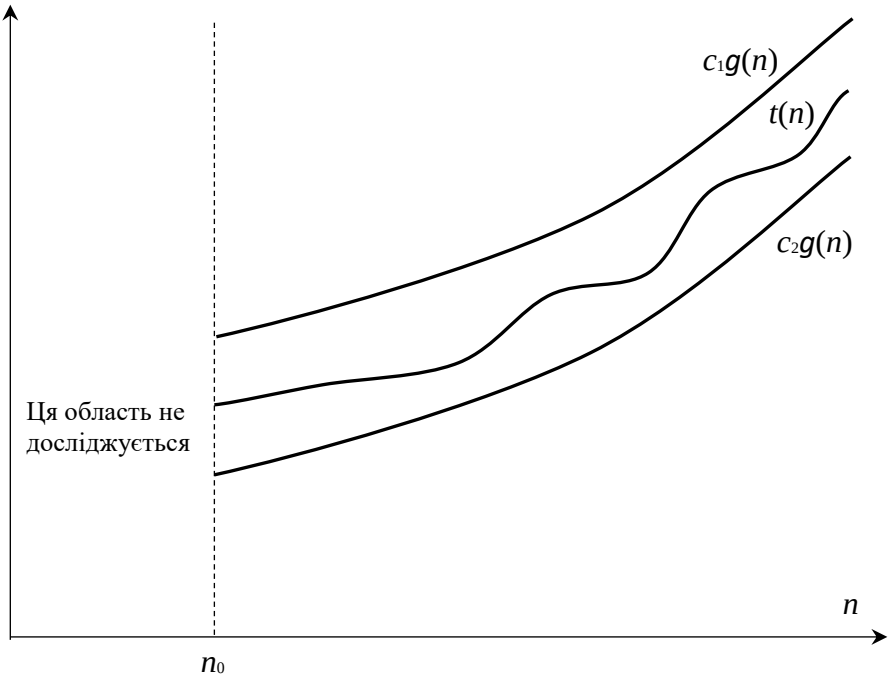


Рисунок 2.3 – Ілюстрація Θ-позначення: $t(n) \in \Theta\big(g(n)\big)$

2.4.2 Властивості асимптотичних позначень

Теорема 1. Якщо $t_1(n) \in O\big(g_1(n)\big)$ й $t_2(n) \in O\big(g_2(n)\big)$, то $t_1(n) + t_2(n) \in O\Big(\max\{g_1(n), g_2(n)\}\Big)$. Аналогічні твердження справедливі також для Ω, Θ.

Отже, ця властивість із погляду аналізу ефективності алгоритмів, що складаються із двох частин, означає, що загальна ефективність алгоритму залежить від тієї частини, що має найбільший порядок зростання, тобто від найменш ефективної його частини:

$$\left.\begin{matrix} t_1(n) \in O\big(g_1(n)\big) \\ t_2(n) \in O\big(g_2(n)\big) \end{matrix}\right\} t_1(n) + t_2(n) \in O\Big(\max\{g_1(n), g_2(n)\}\Big).$$

Наприклад, перевірити, чи містить масив однакові елементи, можна за допомогою двоетапного алгоритму. Спочатку потрібно відсортувати масив, скориставшись одним з відомих алгоритмів сортування. Потім потрібно пройтися по всіх елементах відсортованого масиву й перевірити, чи не дублюють

сусідні елементи один одного. Ефективність цього алгоритму суттєво залежить від вибору алгоритму сортування.

2.4.3 Використання границь для порівняння порядку зростання двох функцій

Незважаючи на те, що без строгих визначень множин O, Ω, Θ, не можна обійтися при доказі їхніх абстрактних властивостей, вони нечасто використаються при порівнянні порядків зростання двох конкретних функцій. Справа в тому, що існує більш зручний метод виконання цієї оцінки, заснований на обчисленні границі відношення двох функцій, що розглядаються. Може існувати три основні випадки:

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0, \text{ якщо } t(n) \text{ має менший порядок зростання, ніж } g(n);\\ c, \text{ якщо } t(n) \text{ має такий саме порядок зростання, як і } g(n);\\ \infty, \text{ якщо } t(n) \text{ має більший порядок зростання, ніж } g(n).\end{cases}$$

2.4.4 Основні класи ефективності

Часову ефективність великої кількості алгоритмів можна віднести всього до декількох класів. Ці класи, їхні назви, а також деякі пояснення, наведені в табл. 2.2 відповідно до збільшення їхнього порядку зростання.

Таблиця 2.2 – Основні асимптотичні класи ефективності

Клас	**Назва**	**Пояснення**
1	*Константа*	Не враховуючи ефективності в найкращому випадку, в цей клас потрапляє досить невелика кількість алгоритмів. Зазвичай, за нескінченного збільшення розміру вхідних даних, час виконання алгоритму також буде прагнути до нескінченності.
$\log n$	*Логарифмічний*	Зазвичай, така залежність виникає в результаті зменшення розміру задачі на постійне значення на кожному кроці ітерації алгоритму. Слід звернути увагу, що в логарифмічному алгоритмі неможлива робота зі всіма вхідними даними (і навіть з їх деякою фіксованою частиною). В цьому випадку час виконання будь-якого алгоритму щонайменше буде лінійно залежним від розміру вхідних даних.
n	*Лінійний*	До цього класу відносяться алгоритми, що виконують сканування списку, який містить n елементів, наприклад, алгоритм пошуку методом послідовного перебору.

Продовження табл. 2.2

Клас	Назва	Пояснення
$n \log n$	*n-log-n*	До даного класу відноситься велика кількість алгоритмів декомпозиції, таких як алгоритми сортування злиттям та швидкого сортування.
n^2	*Квадратичний*	Як правило, подібна залежність характеризує ефективність алгоритмів, що містять два вбудованих цикли. В якості типових прикладів достатньо назвати простий алгоритм сортування та низку операцій, що виконують над матрицями розміру $n \times n$.
n^3	*Кубічний*	Як правило, подібна залежність характеризує ефективність алгоритмів, що містять три вбудованих цикли. До даного класу відносять декілька доволі складних алгоритмів лінійної алгебри.
2^n	*Експоненціальний*	Дана залежність є типовою для алгоритмів, які виконують обробку всіх підмножин деякої множини, що містить n елементів. Зазвичай, термін “експоненціальний” використовується в доволі широкому сенсі й означає дуже високий порядок росту, тобто включає значно швидші в порівнянні з експонентою порядки росту.
$n!$	*Факторіальний*	Така залежність є типовою для алгоритмів, які виконують обробку всіх перестановок деякої множини, що містить n елементів.

Не виключена можливість, коли для вхідних даних реального розміру, алгоритм, що відноситься до гіршого класу ефективності, буде працювати швидше, ніж алгоритм, що відноситься до кращого класу ефективності. Наприклад, якщо час виконання одного алгоритму змінюється за законом n^3, а іншого – за законом $10^9 n^2$, кубічний алгоритм буде працювати швидше за умови, що n не перевищує 10^9.

Як правило, можна припускати, що алгоритм, що відноситься до кращого асимптотичному класу, буде працювати швидше алгоритму, що відноситься до гіршого асимптотичного класу, навіть при обробці вхідних даних середнього розміру.

2.4.5 Ефективність алгоритму в різних випадках

Існує велика кількість алгоритмів, час виконання яких залежить *не тільки від розміру вхідних даних, але також і від особливостей конкретних вхідних даних*. Як приклад, розглянемо задачу послідовного пошуку. Вона розв’язується

за допомогою досить простого алгоритму, що виконує пошук заданого елемента (*ключа пошуку K*) у списку, що складається з *n* елементів, шляхом послідовного порівняння ключа *K* з кожним з елементів списку. Робота алгоритму завершується, або коли заданий ключ знайдений, або коли весь список вичерпаний. Очевидно, що час роботи цього алгоритму може відрізнятися в дуже широких межах для того самого списку розміру *n*. У найгіршому випадку, тобто коли в списку немає шуканого елемента або коли шуканий елемент розташований у списку останнім, в алгоритмі буде виконана найбільша кількість операцій порівняння ключа з усіма *n* елементами списку.

Під ефективністю алгоритму в *найгіршому випадку* (worst-case efficiency) мають на увазі його ефективність для найгіршої сукупності вхідних даних розміром *n*, тобто для такої сукупності вхідних даних розміром *n* серед всіх можливих, для якої час роботи алгоритму буде найбільшим.

Під ефективністю алгоритму в *найкращому випадку* (best-case efficiency) мають на увазі його ефективність для найкращої сукупності вхідних даних розміром *n*, тобто для такої сукупності вхідних даних розміром *n* серед всіх можливих, для якої час роботи алгоритму буде найменшим.

Аналіз ефективності алгоритму для найкращого випадку не так важливий, як для найгіршого випадку, хоча його не можна назвати зовсім марним.

На підставі інформації, отриманої в результаті аналізу алгоритму для найкращого й найгіршого випадків, не можна зробити висновок про те, як поведеться алгоритм при обробці типових або випадково заданих вхідних даних. Щоб одержати подібну інформацію, потрібно виконати аналіз алгоритму для *середнього випадку* (average-case efficiency). Для цього потрібно зробити ряд припущень про сукупність вхідних даних розміром *n*.

Визначення ефективності алгоритму для середнього випадку більше важка задача, ніж для найгіршого й найкращого випадків.

Ефективність алгоритму для середнього випадку не можна одержати, осередненням його ефективності для найгіршого й найкращого випадків. Навіть якщо іноді отримані в такий спосіб результати збігаються з практичними даними для середнього випадку, такий спосіб аналізу не є правильним.

2.5 Емпіричний аналіз алгоритмів

Загальний план емпіричного аналізу ефективності алгоритму:

1. З’ясування мети майбутнього експерименту.
2. Визначення вимірюваної метрики *M* та одиниць виміру (кількість операцій або час роботи алгоритму).
3. Визначення характеристик вхідних даних (їхній діапазон, розмір тощо).
4. Створення програми, що реалізує алгоритм (або алгоритми), для проведення експерименту.
5. Генерація зразка вхідних даних.
6. Виконання алгоритму (або алгоритмів) над зразком вхідних даних і

запис результатів спостережень (наприклад, у таблицю тощо).

7. Аналіз отриманих результатів (побудова графіку (графіків) з подальшим роз'ясненням тощо).

Розглянемо по черзі зазначені кроки.

Є ряд цілей, які можуть бути поставлені перед емпіричним аналізом алгоритмів. Вони включають перевірку точності теоретичних висновків про ефективність алгоритму, порівняння ефективності декількох алгоритмів, призначених для розв'язання однієї й тієї ж проблеми або різних реалізацій того самого алгоритму, висування гіпотези про клас ефективності алгоритму, з'ясування ефективності програмі, що реалізує алгоритм, на даній конкретній машині. Очевидно, що розробка експерименту повинна залежати від того, на яке саме питання він повинен дати відповідь.

Зокрема, ціль експерименту повинна впливати на те, яким чином буде виконуватися вимір ефективності алгоритму. *Перший варіант* полягає у вставці в програму, що реалізує алгоритм, лічильника (або лічильників), які будуть підраховувати кількість виконань алгоритмом базових операцій. Звичайно це досить просто, і необхідно пам'ятати, що базові операції можуть виконуватися не в одному місці програми, і враховувати всі можливі їхні виконання. Також необхідно завжди перевіряти модифіковану в такий спосіб програму, щоб переконатися в її коректності – як у сенсі розв'язання поставленої перед алгоритмом задачі, так і в сенсі коректності роботи внесених у програму лічильників.

Другий варіант полягає у визначенні часу роботи програми, що реалізує досліджуваний алгоритм. Можна також визначати час роботи фрагмента коду, запитуючи системний час безпосередньо перед початком виконання фрагмента й відразу після його завершення, а потім просто обчислювати різницю отриманих значень. Наприклад, у C++ для цієї мети можна використати функцію clock.

Однак дуже важливо не забувати про деякі речі. По-перше, системний час зазвичай не дуже точний, і можна одержати результати, що будуть відрізнятися один від одного при повторних запусках однієї й тої ж програми з тими самими вхідними даними. Очевидним засобом протидії цьому ефекту є запуск програми й виконання вимірів кілька разів, з наступним осередненням отриманих результатів. По-друге, висока швидкість сучасних комп'ютерів може привести до того, що час роботи буде неможливо зареєструвати (будуть виходити нульові значення). Обійти цю неприємність легко, запускаючи програму в циклі багато разів, а потім поділивши зареєстрований час виконання на кількість ітерацій циклу. По-третє, на комп'ютері, що працює під керуванням багатозадачної операційної системи (як, наприклад, UNIX), час, що реєструється, може включати час, витрачений процесором на роботу над іншими програмами, що, зрозуміло, заважає проведенню експерименту.

Емпіричні дані, отримані в ході експерименту, повинні бути записані, а потім представлені для подальшого аналізу. Дані можуть бути представлені в

таблиці або графічно, точками в декартовій системі координат. Непогано використовувати одночасно обидва способи, оскільки для кожного з них характерні свої сильні й слабкі сторони.

Графічне подання даних також може допомогти у висуванні гіпотези про ймовірний клас ефективності алгоритму. У випадку логарифмічного алгоритму графік має опуклий нагору вид. Цей вид графіка відрізняє логарифмічні алгоритми від всіх інших алгоритмів. У випадку лінійного алгоритму експериментальні точки мають тенденцію вибудовуватися уздовж звичайної прямої лінії або, загалом кажучи, розташовуватися між двома прямими лініями. Графіки функцій з $O(n \cdot \ln(n))$ і $O(n^2)$ мають опуклий вниз вид, що робить їхню ідентифікацію більше складної. Графік для кубічного алгоритму також має опуклий вниз вид, але зростає з істотно більш високою швидкістю.

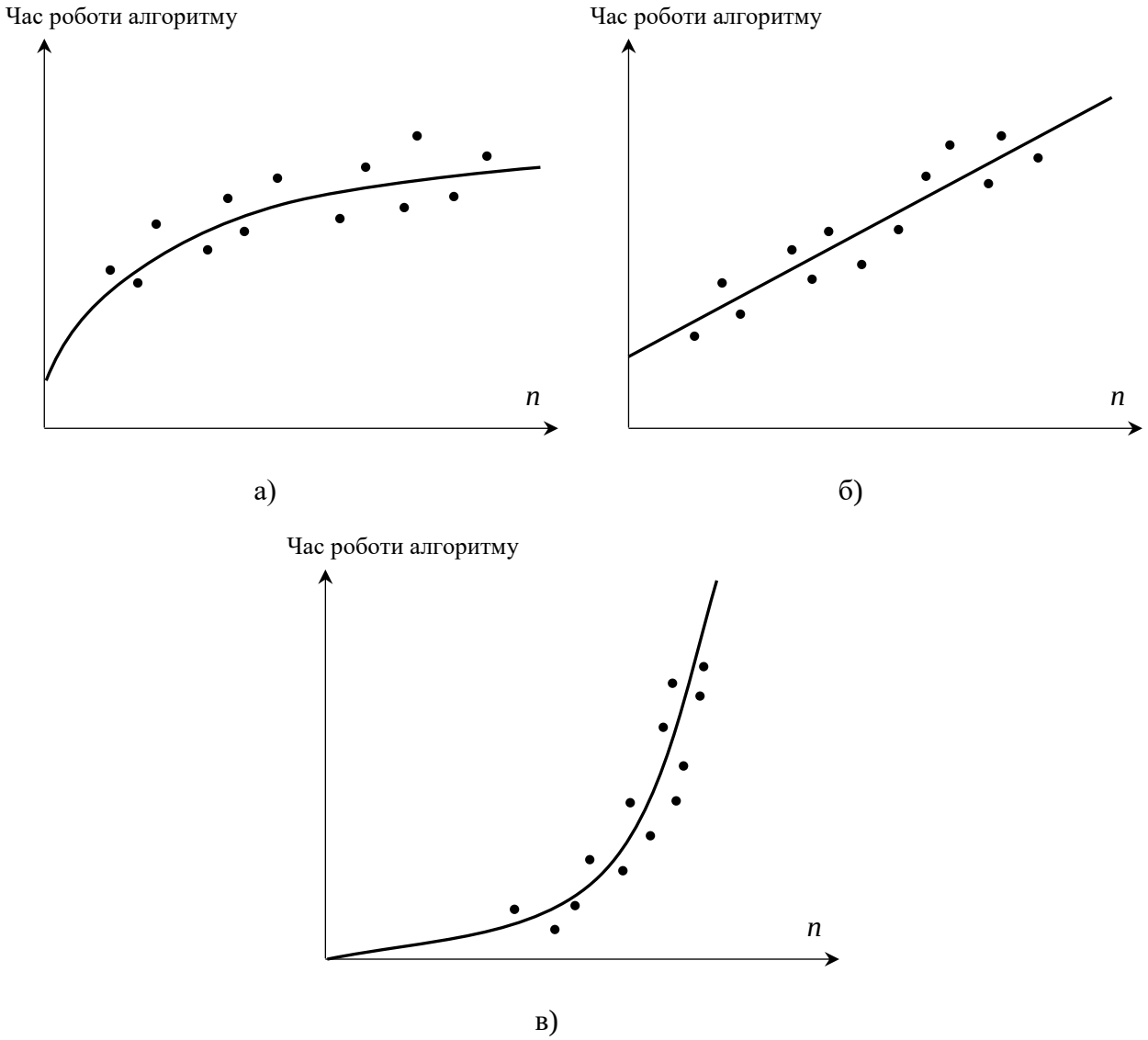


Рисунок 2.4 – Типові графіки функцій: а) логарифмічна; б) лінійна; в) опукла вниз

Одне з можливих застосувань емпіричного аналізу – спроба визначити

продуктивність алгоритму для екземпляра вхідних даних, не включеного в множину екземплярів вхідних даних експерименту.

Таке передбачення називається *екстраполяцією* (extrapolation) на відміну від *інтерполяції* (interpolation), що працює зі значеннями в межах досліджуваного діапазону. Зокрема, не можна нічого сказати про точність таких оцінок. Звичайно, можна спробувати застосувати стандартні методи статистичного аналізу даних, однак помітимо, що вони в основному базуються на певних імовірнісних припущеннях, які можуть виявитися неправильними для розглянутих експериментальних даних.

Головна перевага математичного аналізу алгоритмів – його незалежність від конкретних вхідних даних, а недолік – обмежене застосування, особливо для дослідження ефективності в середньому випадку. Емпіричний аналіз, навпроти, може бути застосований до будь-якого алгоритму, але його результати можуть залежати від конкретних вхідних даних і використаного для проведення експерименту комп'ютера.

2.6 Математичний аналіз алгоритмів

2.6.1 Математичний аналіз ефективності нерекурсивних алгоритмів

Загальний план математичного аналізу ефективності нерекурсивних алгоритмів:

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.
2. Визначте основну операцію алгоритму (як правило, вона перебуває в найбільше глибоко вкладеному внутрішньому циклі алгоритму).
3. Перевірте, чи залежить число виконуваних основних операцій тільки від розміру вхідних даних. Якщо воно залежить і від інших факторів, розгляньте при необхідності, як змінюється ефективність алгоритму для найгіршого, середнього й найкращого випадків.
4. Запишіть суму, що виражає кількість виконуваних основних операцій алгоритму.
5. Використовуючи стандартні формули й правила підсумовування, спростіть отриману формулу для кількості основних операцій алгоритму. Якщо це неможливо, визначте хоча б їхній порядок зростання.

Приклад 1. Розглянемо задачу пошуку найбільшого елемента в списку з n чисел. Для простоти припустимо, що цей список реалізований у вигляді масиву.

Нижче наведений псевдокод алгоритму рішення цієї задачі.

Алгоритм *MaxElement* (A[0..n-1])
// Входові дані: масив дійсних чисел A[0..n-1]
// Виходові дані: повернення значення найбільшого елемента масиву A

```
maxval ← A[0]
for i ← 1 to n-1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Приклад 2. Розглянемо задачу перевірки одиничності елементів. Інакше кажучи, потрібно переконатися, що всі елементи масиву різні.

Цю задачу можна розв'язати за допомогою наведеного нижче нескладного алгоритму.

```
Алгоритм UniqueElements (A[0..n-1])
    // Вхідові дані: масив дійсних чисел A[0..n-1]
    // Виходові дані: повернення значення «true», якщо усі елементи
    //                масиву A різні, та «false» у протилежному випадку
    for i ← 1 to n-2 do
        for j ← i+1 to n-1 do
            if A[i] = A[j]
                return false
    return true
```

2.6.2 Математичний аналіз ефективності рекурсивних алгоритмів

Загальний план математичного аналізу ефективності рекурсивних алгоритмів:

1. Виберіть параметр (або параметри), за яким буде оцінюватися розмір вхідних даних алгоритму.
2. Визначте основну операцію алгоритму.
3. Перевірте, чи залежить число виконуваних основних операцій тільки від розміру вхідних даних. Якщо воно залежить і від інших факторів, розгляньте при необхідності, як змінюється ефективність алгоритму для найгіршого, середнього й найкращого випадків.
4. Складіть рекурентне рівняння, що виражає кількість виконуваних основних операцій алгоритму, і вкажіть відповідні початкові умови.
5. Знайдіть розв'язок рекурентного рівняння або, якщо це неможливо, визначте хоча б його порядок зростання.

Рекурсивні алгоритми варто використовувати дуже обережно, оскільки часто за їхньою зовнішньою компактністю ховається вибагливість до ресурсів ЕОМ.

Приклад 1. Обчислимо значення функції факторіала $F(n) = n!$ для довільного цілого невід'ємного значення n.

Враховуючи, що $n! = 1 \cdot \ldots \cdot (n-1) \cdot n$ для всіх $n \geq 1$ і по визначенню $0! = 1$, можна

обчислити $F(n) = F(n-1) \cdot n$ за допомогою наступного рекурсивного алгоритму.

Алгоритм $F(n)$
// Рекурсивне обчислення факторіалу
// Вхідові дані: Ціле невід'ємне число n
// Виходові дані: Значення $n!$
if $n = 0$
return 1
else
return $F(n-1) * n$

Для простоти будемо вважати, що розмір вхідних даних алгоритму буде вказувати саме число n, а не кількість бітів у його двійковому поданні.

Основною операцією цього алгоритму є множення, кількість виконань цієї операції позначимо через $M(n)$. Враховуючи, що для всіх $n > 0$ значення функції $F(n)$ обчислюється за наступною формулою:

$$F(n) = F(n-1) \cdot n,$$

кількість операцій множення $M(n)$, виконуваних при цьому в алгоритмі, повинне задовольняти наступній рівності для всіх $n > 0$:

$$\underset{}{M(n) =} \quad \underbrace{M(n-1)}_{\substack{\text{для обчислення} \\ F(n-1)}} \quad + \quad \underbrace{1}_{\substack{\text{для множення} \\ F(n-1) \text{ на } n}}$$

Останнє рівняння визначає послідовність $M(n)$, що необхідно знайти. Значення для $M(n)$ не задано явно, тобто у вигляді функції від числа n. Воно виражено неявно у вигляді функції, значення якої залежить від значення цієї ж функції на попередньому кроці алгоритму, тобто при n - 1. Подібні рівності називають рекурентними рівняннями, або рекурентними співвідношеннями (recurrence relations), або, для стислості, просто рекурентностями (recurrences). Рекурентні співвідношення відіграють важливу роль не тільки при аналізі алгоритмів, але й у деяких інших областях прикладної математики.

Варто звернути вашу увагу, що існує нескінченна множина послідовностей, що задовольняють розв'язку наведеного вище рекурентного співвідношення. Щоб знайти однозначне рішення, потрібно задати *початкові умови* (initial condition), тобто вказати значення, з якого починається послідовність. Щоб визначити це значення, потрібно проаналізувати умову, при якій в алгоритмі припиняється рекурсивний виклик процедури:

if $n = 0$ **return** 1.

Рекурентне співвідношення й початкові умови для кількості операцій множення $M(n)$ розглянутого нами алгоритму виглядають так:

$$M(n) = M(n-1) + 1 \quad \text{для } n > 0,$$
$$M(0) = 0.$$

Існує кілька методик розв'язання рекурентних рівнянь. Скористаємося однією з них, що називається *методом зворотної підстановки* (method of backward substitutions). Ідея методу та його назва відразу стануть зрозумілі після застосування його до розв'язання нашого рекурентного рівняння:

$$M(n) = M(n-1) + 1 = \quad \text{підставимо } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 =$$
$$= M(n-2) + 2 = \quad \text{підставимо } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 =$$
$$= M(n-3) + 3.$$

Глянувши на перші три рядки, легко помітити закономірність, що дозволить передбачити не тільки вид наступного рядка розв'язання, але також і вивести для нього загальну формулу:

$$M(n) = M(n-i) + i.$$

Залишилося тільки застосувати початкові умови. Оскільки вони задані для $n = 0$, то, щоб одержати останній рядок (або кінцевий результат) розв'язання методом зворотних підстановок, потрібно в наведеній вище загальній формулі виконати підстановку для $i=n$:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Приклад 2. Задача про Ханойські вежі.

Задані три кілочка, на один з яких нанизані диски, причому диски відрізняються розміром і лежать менший на більшому. Задача полягає у тому, щоб перенести цю вежу з дисків за найменшу кількість ходів на другий кілочок, використовуючи третій як допоміжний. За один раз дозволяється перенести тільки один диск, причому неможна покласти більший диск на менший.

Щоб перенести $n > 1$ дисків з першого кілочка на третій (при цьому другий кілочок використовується як допоміжний), спочатку необхідно рекурсивно перенести $n - 1$ дисків з першого кілочка на другий (використовуючи при цьому третій кілочок як допоміжний). Після цього необхідно перенести найбільший диск безпосередньо з першого кілочка на третій і, насамкінець, рекурсивно перенести $n - 1$ дисків з другого кілочка на третій (використовуючи при цьому перший кілочок як допоміжний). Природньо, що при $n=1$, можна безпосередньо перенести єдиний диск з першого кілочка на третій.

Давайте застосуємо наведений вище загальний план аналізу ефективності рекурсивних алгоритмів до задачі про Ханойські вежі. Вочевидь, що в даному випадку розмір вхідних даних потрібно оцінювати за кількістю дисків n, а основною операцією алгоритму буде перенесення одного диску. Зрозуміло, що кількість перенесень $M(n)$ повинна залежати тільки від числа n. Тому для будь-якого $n > 1$ буде правильно таке рекурентне рівняння:

$$M(n) = M(n-1) + 1 + M(n-1).$$

Додамо до нього очевидну початкову умову $M(1) = 1$, отримаємо таке рекурентне співвідношення для кількості перенесень дисків $M(n)$:

$$M(n) = 2M(n-1) + 1 \text{ для } n > 1,$$
$$M(1) = 1.$$

Для розв'язання цього рекурентного рівняння знову скористаємося методом зворотних підстановок:

$$M(n) = 2M(n-1) + 1 = \quad \text{підставимо } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 =$$
$$= 2^2 M(n-2) + 2 + 1 = \quad \text{підставимо } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 =$$
$$= 2^3 M(n-3) + 2^2 + 2 + 1.$$

В перших трьох рядках цього розв'язання простежується очевидна закономірність, так що четвертий рядок буде виглядати так:

$$2^4 M(n-4) + 2^3 + 2^2 + 2 + 1.$$

Щоб отримати узагальнену формулу розв'язання, підставимо замість номера рядка число i. В результаті маємо таке:

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Оскільки початкові умови задані для $n = 1$, то, щоб отримати відповідний цьому числу рядок розв'язання рекурентного відношення, необхідно підставити $i = n - 1$. В результаті отримаємо такий розв'язок рекурентного рівняння:

$$M(n) = 2^{n-1} M\big(n - (n-1)\big) + 2^{n-1} - 1 =$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Отже, з'ясовано, що розглянутий алгоритм відноситься до класу експоненціальних. Це означає, що він буде виконуватися неймовірно тривалий

час навіть для помірних значень n. Причому це не означає, що обрано для розв'язання задачі про ханойські вежі поганий алгоритм. Насправді нескладно довести, що даний алгоритм є самим ефективним серед усіх можливих алгоритмів розв'язання цієї задачі.

Вся суть полягає у складності, притаманній самій задачі, що суттєво ускладнює її розв'язання чисельними методами. Тим не менше цей приклад дозволяє зробити один важливий висновок: рекурсивні алгоритми слід використовувати дуже обережно, оскільки часто за їх зовнішньою компактністю ховається крайня неефективність.

Якщо в рекурсивному алгоритмі виконується більше одного виклику його самого, то для аналізу такого алгоритму корисно побудувати дерево його рекурсивних викликів. Вузли такого дерева будуть відповідати рекурсивним викликам. Їх можна позначити у відповідності до значень параметра (або, в більш загальному випадку, декількох параметрів), який передається рекурсивній функції в момент виклику. Для прикладу з ханойськими вежами дерево рекурсивних викликів буде виглядати так, як показано на рис. 2.5.

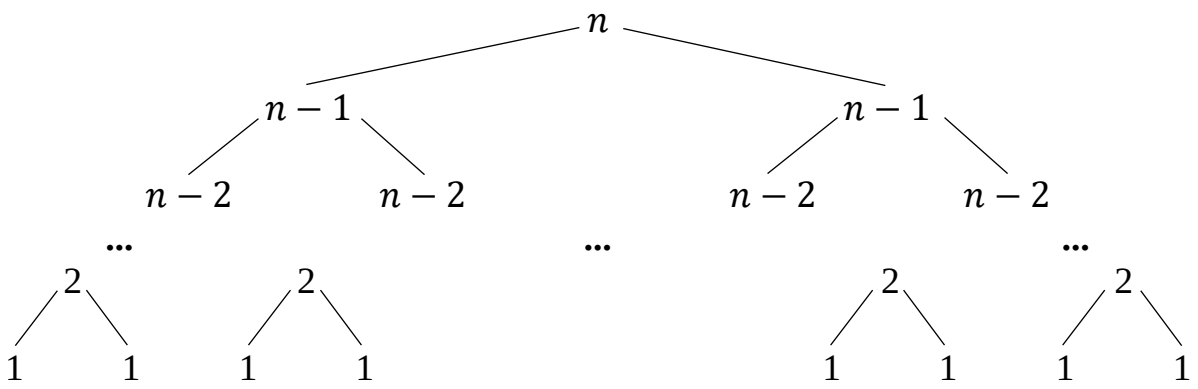


Рисунок 2.5 – Дерево рекурсивних викликів в алгоритмі розв'язання задачі про ханойські вежі

Крім математичного, асимптотичного й емпіричного аналізу є ще один шлях вивчення алгоритмів. Він називається *візуалізацією алгоритму* й може бути визначений як використання зображень для передачі деякої корисної інформації про алгоритми. Ця інформація може бути візуальною ілюстрацією дій, виконуваних алгоритмом, або його продуктивності для різних вхідних даних, або його швидкості виконання в порівнянні з іншими алгоритмами для розв'язання тієї ж задачі. Для досягнення даної мети візуалізація алгоритму використає графічні елементи (точки, відрізки, прямокутники або паралелепіпеди й т.д.) для подання деяких «цікавих подій» у роботі алгоритму. Найбільш популярні системи загального призначення включають BALSA, TANGO, ZEUS, Animated Algorithms.

2.7 Методи розв'язання рекурентних співвідношень

2.7.1 Послідовності і рекурентні співвідношення

Числова *послідовність* (sequence) являє собою упорядкований список чисел. Прилади наведені у таблиці 2.3.

Таблиця 2.3 – Приклади числових послідовностей

2, 4, 6, 8, 10, 12, ...	Додатні парні числа
0, 1, 1, 2, 3, 5, 8, ...	Числа Фібоначі
0, 1, 3, 6, 10, 15, ...	Кількість порівнянь ключів при сортуванні вибором

Послідовність зазвичай позначається літерою (наприклад, x або a) з нижнім індексом (наприклад, n або i) і записується у фігурних дужках (наприклад, $\{x_n\}$) або альтернативний варіант $x(n)$. В останньому випадку підкреслюється той факт, що послідовність – це функція, де n – аргумент, що вказує на позицію числа у списку, а значення функції $x(n)$ – саме число. При цьому, $x(n)$ називається *узагальненим членом послідовності* (generic term).

Послідовність можна задати двома способами:

– *явною формулою*, що виражає узагальнений член послідовності як функцію від n, наприклад, $x(n) = 2n$ для $n \geq 0$;

– *рівнянням*, яке пов'язує узагальнений член послідовності з одним або декількома іншими членами послідовності, в комбінації з одним або декількома відомими значеннями перших членів, наприклад:

$$x(n) = x(n-1) + n \text{ для } n > 0, \tag{2.1}$$

$$x(0) = 0. \tag{2.2}$$

Другий метод особливо важливий для аналізу рекурсивних алгоритмів.

Рівняння виду (2.1) називається *рекурентним рівнянням* (recurrence equation) або *рекурентним співвідношенням* (recurrence relation) (чи просто *рекурентністю* (recurrence)), а (2.2) – *початковою умовою* (initial condition). Початкова умова може бути задана для значення n відмінного від нуля, Наприклад, для $n = 1$ тощо.

Розв'язати рекурентне співвідношення при заданих початкових умовах означає знайти явну формулу узагальненого члена послідовності, яка задовольняє як рекурентному співвідношенню, так і початковим умовам, або довести, що такої послідовності не існує. Наприклад, розв'язок рекурентного співвідношення (2.1) при початковій умові (2.2) має вигляд

$$x(n) = \frac{n(n+1)}{2}, \qquad (n \geq 0). \tag{2.3}$$

Безпосередньою підстановкою у формулу (2.1) можна переконатися, що рівняння виконується для всіх $n > 0$, тобто що

$$\frac{n(n+1)}{2} = \frac{(n-1)(n-1+1)}{2} + n,$$

а підстановкою в початкову умову (2.2) дає

$$\frac{0(0+1)}{2} = 0.$$

Іноді зручно розрізняти загальний та частковий розв'язок рекурентного співвідношення. Зазвичай, рекурентне рівняння має нескінченну кількість послідовностей, що йому задовольняють. *Загальний розв'язок* (general solution) рекурентного рівняння – це формула, що визначає всі такі послідовності. Загальний розв'язок, як правило, включає одну або декілька довільних сталих. Наприклад, загальний розв'язок рекурентного рівняння (2.1) має вигляд:

$$x(n) = c + \frac{n(n+1)}{2}, \tag{2.4}$$

де c – довільна стала. Привласнюючи c різні значення, можна отримати всі розв'язки (2.1), і тільки їх.

Частковим розв'язком (particular solution) рекурентного рівняння є конкретна послідовність, що задовольняє даному рекурентному рівнянню. Зазвичай, нас цікавить частковий розв'язок, який задовольняє заданій початковій умові. Наприклад, послідовність (2.3) – це частковий розв'язок рекурентного рівняння (2.1) за початкових умов (2.2).

2.7.2 Методи розв'язання рекурентних співвідношень

Не існує універсального методу розв'язання, який дозволив би розв'язувати будь-яке рекурентне співвідношення. Розглянемо найпоширеніші підходи.

Метод прямої підстановки. Починаючи з початкового члена (або декількох початкових членів) послідовності для заданих початкових умов, можна використовувати рекурентне співвідношення для генерації декількох перших членів його розв'язку в надії зрозуміти, як саме може виглядати кінцева формула. Якщо така формула знайдена, її коректність має бути перевірена безпосередньою підстановкою в рекурентне рівняння та початкові умови (аналогічно до того, як діяли для (2.1) та (2.2)) або доведена методом математичної індукції.

Наприклад, розглянемо рекурентне співвідношення:

$$x(n) = 2x(n-1) + 1 \text{ для } n > 1, \quad (2.5)$$
$$x(1) = 1. \quad (2.6)$$

Перші члени отримаємо наступним чином:

$$x(1) = 1,$$
$$x(2) = 2x(1) + 1 = 2 \cdot 1 + 1 = 3,$$
$$x(3) = 2x(2) + 1 = 2 \cdot 3 + 1 = 7,$$
$$x(4) = 2x(3) + 1 = 2 \cdot 7 + 1 = 15.$$

Не складно помітити, що отримані числа на 1 менші послідовності степенів числа 2:

$$x(n) = 2^n - 1 \text{ для } n = 1 \ldots 4.$$

Можна довести, що ця гіпотеза дає узагальнений член розв'язку (2.5)–(2.6) або безпосередньою підстановкою формули у (2.5) та (2.6), або за допомогою математичної індукції.

З практичної точки зору цей метод можна використати лише для дуже обмеженої кількості випадків, оскільки зазвичай дуже складно розпізнати вид формули по декільком першим членам послідовності.

Метод зворотної підстановки. Цей метод розв'язування рекурентних співвідношень по своїй суті працює так, як сказано в його назві: використовуючи рекурентне співвідношення, яке розглядається, виражають $x(n-1)$ як функцію $x(n-2)$ і підставляють результат у вихідне співвідношення, щоб отримати $x(n)$ як функцію $x(n-2)$. Повторюючи цей крок для $x(n-2)$, отримують вираз $x(n)$ як функцію $x(n-3)$. Для більшості рекурентних співвідношень, після таких дій як з'являється можливість знайти залежність та представити $x(n)$ як функцію $x(n-i)$ для довільних $i = 1, 2, \ldots$. Обираючи i таким чином, щоб $n - i$ досягало початкової умови і використовуючи одну з формул підсумовування, зазвичай, вдається отримати явну формулу для розв'язку рекурентного співвідношення.

Для прикладу, використаємо описаний метод до рекурентного співвідношення (2.1)–(2.2). Отже, маємо рекурентне співвідношення:

$$x(n) = x(n-1) + n.$$

Замінимо в останньому рівнянні n на $n - 1$. Отримаємо:

$$x(n-1) = x(n-2) + n - 1.$$

Після підстановки отриманого виразу для $x(n-1)$ у вихідне рівняння, отримаємо:

$$x(n) = (x(n-2) + n - 1) + n = x(n-2) + (n-1) + n.$$

Заміна n на $n-2$ дасть $x(n-2) = x(n-3) + n - 2$. Після підстановки отриманого виразу для $x(n-2)$, отримаємо:

$$x(n) = (x(n-3) + n - 2) + (n-1) + n = x(n-3) + (n-2) + (n-1) + n.$$

Порівняння трьох формул для $x(n)$ дає можливість помітити, що загальний вигляд формули після i підстановок має вигляд:

$$x(n) = x(n-i) + (n-i+1) + (n-i+2) + \cdots + n.$$

Оскільки початкова умова (2.2) вказана для $n = 0$, то для його виконання необхідно, щоб виконувалось співвідношення $n - i = 0$, тобто $i = n$:

$$x(n) = x(0) + 1 + 2 + \cdots + n = 0 + 1 + 2 + \cdots + n = n(n+1)/2.$$

Метод зворотної підстановки на диво добре підходить для розв'язування простих рекурентних співвідношень.

Практичні завдання

1. Реалізувати рекурсивний алгоритм розв'язання задачі про ханойські вежі.
2. Провести математичний аналіз рекурсивного алгоритму.
3. Провести емпіричний аналіз рекурсивного алгоритму. Результати обчислювальних експериментів звести в таблицю. Побудувати точковий графік залежності часу роботи алгоритму від кількості дисків n.
4. Порівняти оцінки часу роботи алгоритму, отримані теоретично та експериментально.
5. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення кількості дисків n;
   - виводити послідовність дій по переміщенню дисків (наприклад: «Перемістити диск із кілочка №1 на кілочок № 2»);
   - виводити службову інформацію про параметри роботи алгоритму (час роботи, кількість ітерацій тощо).
6. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму;
   - точковий графік залежності часу роботи алгоритму від кількості дисків n;
   - висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;

– відповіді на контрольні запитання.

Тестові завдання

1. Цей тип операцій алгоритму, зазвичай, розташовуються у найбільш вкладеному циклі

а) базові операції;

б) операції порівняння;

в) арифметичні операції;

г) присвоювання.

2. ____ – це множина всіх функцій, порядок зростання яких при досить великих n не перевищує (тобто менше або дорівнює) деякої константи, помноженої на значення функції $g(n)$:

а) асимптотичне позначення O;

б) асимптотичне позначення Θ;

в) асимптотичне позначення Ω;

г) асимптотичне позначення o.

3. Говорять, що функція $t(n)$ належить множині ____, що записується як $t(n) \in$ ___, якщо для всіх великих значень n функція $t(n)$ обмежена знизу деякою константою, помноженої на значення функції $g(n)$, тобто якщо існує додатна константа c й невід'ємне ціле число n_0 таке, що $t(n) \geq cg(n), \forall n$.

а) $\Omega\big(g(n)\big)$;

б) $\Theta\big(g(n)\big)$;

в) $O\big(g(n)\big)$;

г) жодного з цих варіантів.

4. До якого класу ефективності відноситься алгоритм послідовного пошуку?

а) лінійний;

б) константний;

в) квадратичний;

г) експоненціальний.

5. Для задачі сортування, яка структура вхідних даних відноситься до найгіршого випадку?

а) частково відсортовані дані;

б) дані, що відсортовані у зворотному порядку;

в) випадкова послідовність;

г) дані, які містять багато повторів.

Контрольні запитання

1. Які два основних види ефективності алгоритмів ви знаєте?
2. Дайте визначення асимптотичних позначень O, Θ, Ω.
3. Які Ви знаєте основні класи ефективності алгоритмів?
4. Які Ви знаєте два основних підходи в аналізі алгоритмів? У чому

переваги й недоліки кожного з них?

5. Поясніть поняття ефективності алгоритму для найгіршого, найкращого й середнього випадків.

Тема 3 Структури даних та абстрактні типи даних

Мета: вивчити основні структури даних та абстрактні типи даних; розробити алгоритм і програму реалізації абстрактного типу даних список та стек.

План

1. Поняття про структури даних
2. Абстрактний тип даних «Список»
3. Реалізація списків
4. Стеки
5. Черги

3.1 Поняття про структури даних

Структури даних поряд з алгоритмами є основними складовими частинами створюваних програм. Одну зі своїх книг професор Н. Вірт, винахідник мови програмування Pascal, назвав: «Алгоритми + Структури даних = Програми». Використовувані в програмуванні структури даних можна розділити на дві великі групи:

– дані статичної структури – це такі структури даних, взаємне розташування й взаємозв'язок елементів яких залишаються постійними під час виконання програми;

– дані динамічної структури – це структури даних, внутрішня будова яких формується за певним правилом, а кількість елементів, їх взаємне розташування й взаємозв'язки можуть змінюватися під час виконання програми відповідно до правила формування структури.

Тип даних змінної позначає множину значень, які може приймати ця змінна. Наприклад, змінна булевого (логічного) типу може приймати тільки два значення: значення true (істина) і значення false (лож) і ніякі інші. Набір базових типів даних відрізняється в різних мовах: це типи цілих (integer) і дійсних (real) чисел, булевий (boolean) тип і символьний (char) тип. Правила конструювання складених типів даних (на основі базових типів) також розрізняються в різних мовах програмування: так, легко й швидко будує такі типи.

Базовим будівельним блоком структури даних є ***комірка***, що призначена для зберігання значення певного базового або складеного типу даних. Структури даних створюються шляхом завдання імен сукупностям (агрегатам) комірок і (необов'язково) інтерпретації значення деяких комірок як представників (тобто покажчиків) інших комірок.

Як *найпростіший механізм агрегування* комірок в більшості мов програмування можна застосовувати (одномірний) масив, тобто послідовність комірок певного типу. Масив також можна розглядати як відображення множини індексів (таких як цілі числа) у множину комірок. Посилання на комірку зазвичай

складається з імені масиву й значення із множини індексів даного масиву.

Другим загальним механізмом агрегування комірок у мовах програмування є структура запису. Запис (record) можна розглядати як осередок, що складається з декількох інших комірок (називаних полями), значення в яких можуть бути різних типів. Записи часто групуються в масиви; тип даних визначається сукупністю типів полів запису.

```
ім'я: array[ТипІндексу] of ТипКомірок;
var
    reclist: array[1..4] of record
        data: real;
        next: integer
    end
```

Третій метод агрегування комірок, який можна знайти в Pascal і деяких інших мовах програмування, – це файл. Файл, як і одномірний масив, є послідовністю значень певного типу. Однак файл не має індексів: його елементи доступні тільки в тому порядку, у якому вони були записані у файл. На відміну від файлу, масиви й записи є структурами з «довільним доступом», маючи на увазі під цим, що час доступу до компонентів масиву або запису не залежить від значення індексу масиву або покажчика поля запису. Перевага агрегування за допомогою файлу (частково компенсує описаний недолік) полягає в тім, що файл не має обмеження на кількість складових його елементів і ця кількість може змінюватися під час виконання програми.

Хоча терміни **тип даних** (або просто тип), **структура даних** і **абстрактний тип даних** звучать схоже, але мають вони різний зміст.

Абстрактний тип даних (АТД) визначається як математичну модель із сукупністю операторів, визначених у рамках цієї моделі. Простим прикладом АТД можуть служити множини цілих чисел з операторами об'єднання, перетинання й різниці множин. У моделі АТД оператори можуть мати операндами не тільки дані, певні АТД, але й дані інших типів: стандартних типів мови програмування або визначених в інших АТД.

Термін **реалізація АТД** має на увазі наступне: переклад в оператори мови програмування оголошень, що визначають змінні цього абстрактного типу даних, плюс процедури для кожного оператора (дії), виконуваного над об'єктами АТД. Реалізація залежить від структури даних, що представляють АТД. Кожна структура даних будується на основі базових типів даних застосовуваної мови програмування, використовуючи доступні в цій мові засобу структурування даних.

Як уже вказувалося, можна розробляти алгоритм у термінах АТД, але для реалізації алгоритму в конкретній мові програмування необхідно знайти спосіб подання АТД у термінах типів даних і операторів, підтримуваних даною мовою програмування.

3.2 Абстрактний тип даних «Список»

У програмуванні список являє собою послідовність елементів певного типу, що у загальному випадку будемо позначати як *elementtype* (тип елемента). Будемо часто представляти список у вигляді послідовності елементів, розділених комами: $a_1, a_2, \ldots, a_n$, де $n > 0$ і всі a, мають тип *elementtype*. Кількість елементів n будемо називати довжиною списку. Якщо $n >= 1$, то a_1 називається першим елементом, а a_n – останнім елементом списку. У випадку $n = 0$ маємо порожній список, що не містить елементів.

Для формування абстрактного типу даних на основі математичного визначення списку необхідно задати множину операторів, виконуваних над об'єктами типу *LIST* (Список). Однак не існує однієї множини операторів, виконуваних над списками, що задовольняє відразу всі можливі застосування. Це твердження справедливо й для багатьох інших АТД, які будуть розглядатися далі.

Щоб показати деякі загальні оператори, виконувані над списками, припустимо, що маємо додаток, що містить список поштового розсилання, що необхідно очистити від повторюваних адрес. Концептуально ця задача вирішується дуже просто: для кожного елемента списку видаляються всі наступні елементи, що збігаються з даним. Однак для запису такого алгоритму необхідно визначити оператори, які повинні знайти перший елемент списку, перейти до наступного елемента, здійснити пошук і видалення елементів.

Тепер перейдемо до безпосереднього визначення множини операторів списку. Приймемо позначення: *L* – список об'єктів типу *elementtype*, *x* – об'єкт цього типу, *p* – позиція елемента в списку. Відзначимо, що «позиція» має інший тип даних, її реалізація може бути різною для різних реалізацій списків. Зазвичай позиції розуміються як множина цілих додатних чисел, але на практиці можуть зустрітися інші подання.

1. **INSERT(x, p, L)**. Цей оператор вставляє об'єкт *x* у позицію *p* у списку *L*, переміщаючи елементи від позиції *p* і далі в наступну, більш високу позицію. Таким чином, якщо список *L* складається з елементів $a_1, a_2, \ldots, a_n$ то після виконання цього оператора він буде мати вигляд $a_1, a_2, \ldots, a_{p-1}, x, a_{p+1}, \ldots, a_n$... Якщо *p* приймає значення *END(L)*, то будемо мати $a_1, a_2, \ldots, a_n, x$. Якщо в списку *L* немає позиції *p*, те результат виконання цього оператора не визначений.

2. **LOCATE(x, L)**. Ця функція повертає позицію об'єкта *x* у списку *L*. Якщо в списку об'єкт *x* зустрічається кілька разів, то повертається позиція першого від початку списку об'єкта *x*. Якщо об'єкта *x* немає в списку *L*, то вертається *END(L)*.

3. **RETRIEVE(p, L)**. Ця функція повертає елемент, що стоїть в позиції *p* у списку *L*. Результат не визначений, якщо $p = END(L)$ або в списку *L* немає позиції *p*. Відзначимо, що елементи повинні бути того типу, що у принципі може повертати функція. Однак на практиці завжди можна змінити цю функцію так, що вона буде повертати покажчик на об'єкт типу elementtype.

4. **DELETE(p, L)**. Цей оператор видаляє елемент у позиції *p* списку *L*. Так,

якщо список *L* складається з елементів a_1, a_2,..., a_n то після виконання цього оператора він буде мати вигляд $a_1,...,a_{p-1}$, $a_{p+1},...,a_n$... Результат не визначений, якщо в списку *L* немає позиції *p* або *p = END(L)*.

5. **NEXT(p, L)** і **PREVIOUS(p, L)**. Ці функції повертають відповідно наступну й попередню позиції від позиції *p* у списку *L*. Якщо *p* – остання позиція в списку *L*, то *NEXT(p, L) = END(L)*. Функція NEXT не визначена, коли p = END(L). Функція PREVIOUS не визначена, якщо p = 1. Обидві функції не визначені, якщо в списку *L* немає позиції *p*.

6. **MAKENULL(L)**. Ця функція робить список *L* порожнім і повертає позицію END(L).

7. **FIRST(L)**. Ця функція повертає першу позицію в списку *L*. Якщо список порожній, то вертається позиція END(L).

8. **PRINTLIST(L)**. Друкує елементи списку *L* у порядку розташування.

Приклад. Використовуючи описані вище оператори, створимо процедуру *PURGE* (Очищення), що як аргумент використає список і видаляє з нього однакові елементи. Елементи списку мають тип *elementtype*, а список таких елементів має тип *LIST*. Визначимо функцію same(x, y), де *x* і *y* мають тип *elementtype*, що приймає значення true (істина), якщо *x* і *y* «однакові» (same), і значення false (неправда) у противному випадку. Поняття «однакові», мабуть, вимагає пояснення. Якщо тип elementtype, наприклад, збігається з типом дійсних чисел, то можна покласти, що функція same(x, y) буде мати значення true тоді й тільки тоді, коли x = y. Але якщо тип elementtype є типом запису, що містить поля поштового індексу (acctno), імені (name) і адреси абонента (address), цей тип можна оголосити в такий спосіб:

```
type
    elementtype = record
        acctno: integer;
        name: packed array [1..20] of char;
        address: packed array [1..50] of char;
end
```

Тепер можна задати, щоб функція same(x, y) приймала значення true щораз, коли x.acctno = y.acctno.

У лістингу нижче представлений код процедури PURGE. Змінні *p* та *q* використаються для зберігання двох позицій у списку.

```
procedure PURGE ( var L: LIST );
    var
        p, q: position; { p – «поточна» позиція у списку L,
                          q переміщується вперед від позиції p}
    begin
(1)         p:=FIRST (L);
```

```
(2)          while p < > END (L) do begin
(3)                q:=NEXT (p, L);
(4)                while q < > END (L) do
(5)                     if same (RETRIEVE (p, L), RETRIEVE (q, L)) then
(6)                                DELETE (q, L)
                        else
(7)                           q:=NEXT (q, L);
(8)          p:=NEXT (p, L)
             end
        end; {PURGE}
```

3.3 Реалізація списків

При реалізації списків за допомогою масивів елементи списку розташовуються в суміжних комірках масиву. Це подання дозволяє легко переглядати вміст списку й вставляти нові елементи в його кінець. Але вставка нового елемента в середину списку вимагає переміщення всіх наступних елементів на одну позицію до кінця масиву, щоб звільнити місце для нового елемента. Видалення елемента також вимагає переміщення елементів, щоб закрити комірку, що звільнився.

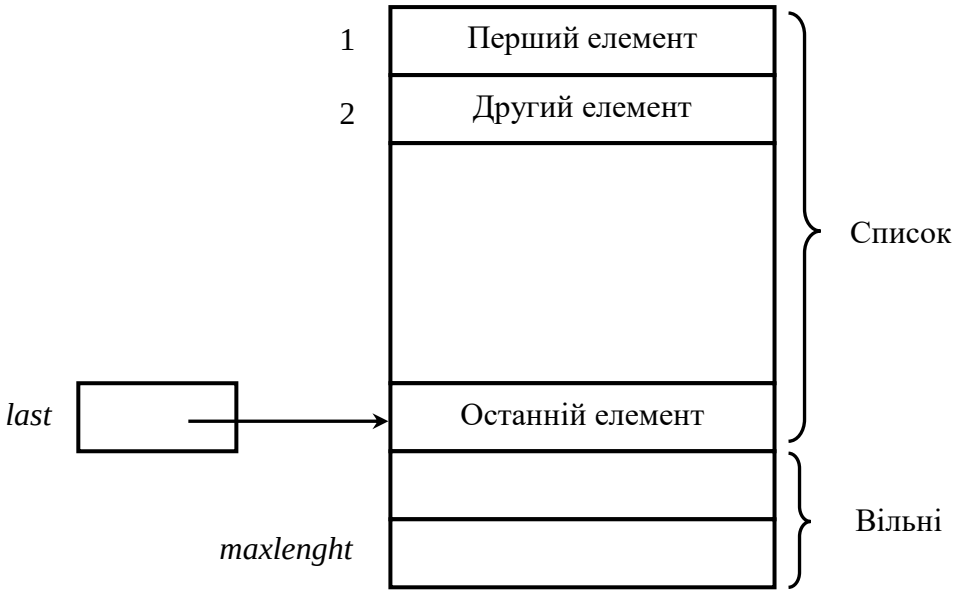


Рисунок 3.1 – Схематичне зображення розміщення елементів списку

Для реалізації односпрямованих списків можуть використатися покажчики, що зв’язують послідовні елементи списку. Ця реалізація звільняє від використання безперервної області пам’яті для зберігання списку й, отже, від необхідності переміщення елементів списку при вставці або видаленні елементів. Однак ціною за цю зручність стає додаткова пам’ять для зберігання покажчиків.

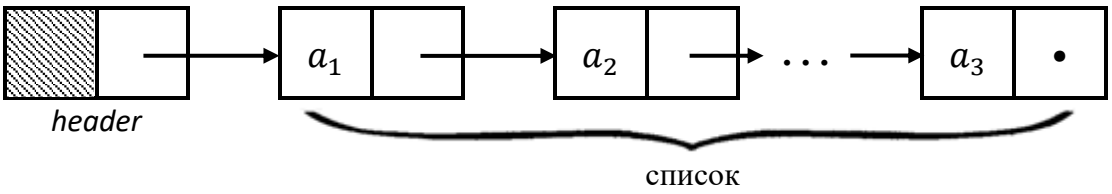
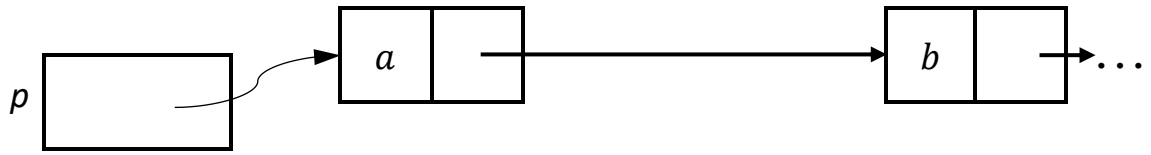
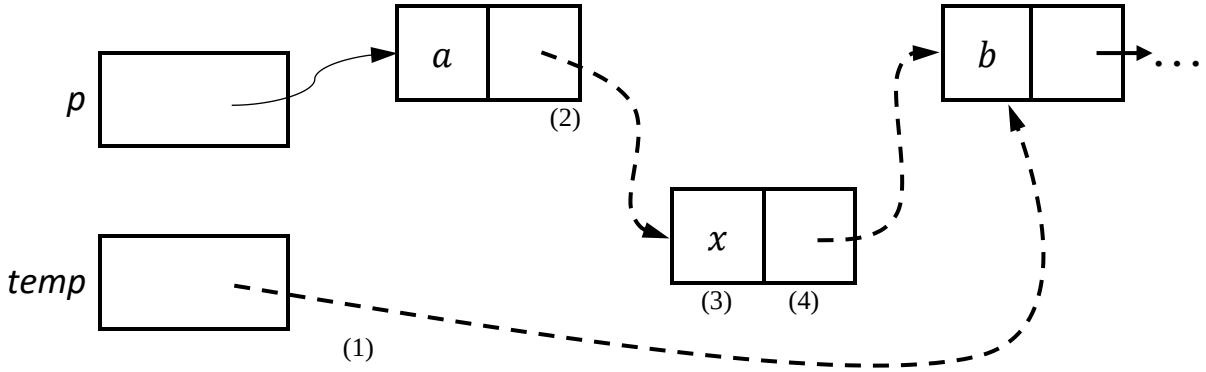


Рисунок 3.2 – Схематичне зображення додаткової пам’яті



а)



б)

Рисунок 3.3 – Ілюстрація процедури Insert

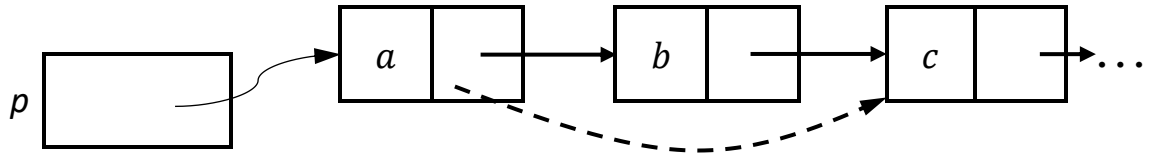
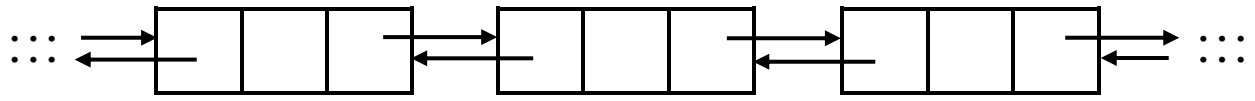


Рисунок 3.4 – Ілюстрація процедури Delete

У багатьох додатках виникає необхідність організувати ефективне переміщення за списком як у прямому, так і у зворотному напрямках. Або за заданим елементом потрібно швидко знайти попередній йому й наступний елементи.

У цих ситуаціях можна дати кожній комірці покажчики й на наступну, і на попередні комірки списку, тобто організувати двічі зв’язний список.



```
type
    celltype = record
        element: elementtype;
        next, previous: ↑ celltype
```

end;
position = ↑ celltype;

Порівняння реалізацій

1. Реалізація списків за допомогою масивів вимагає вказівки максимального розміру списку до початку виконання програм. Якщо неможливо заздалегідь обмежити зверху довжину оброблюваних списків, то, мабуть, більше раціональним вибором буде реалізація списків за допомогою покажчиків.

2. Виконання деяких операторів в одній реалізації вимагає більших обчислювальних витрат, чим в іншій. Наприклад, процедури INSERT і DELETE виконуються за постійне число кроків у випадку зв'язаних списків будь-якого розміру, але вимагають часу, пропорційного кількості елементів, що знаходяться за тим елементом, що вставляється (або що видаляється), при застосуванні масивів. І навпаки, час виконання функцій PREVIOUS і END постійний при реалізації списків за допомогою масивів, але цей же час пропорційний довжині списку у випадку реалізації, побудованої за допомогою покажчиків.

3. Якщо необхідно вставляти або видаляти елементи, положення яких зазначено за допомогою якоїсь змінної типу position, і значення цієї змінної буде використано пізніше, то недоцільно використати реалізацію за допомогою покажчиків, оскільки ця змінна не «відслідковує» вставку й видалення елементів, як показано вище. Взагалі використання покажчиків вимагає особливої уваги й старанності в роботі.

4. Реалізація списків за допомогою масивів марнотратна відносно комп'ютерної пам'яті, оскільки резервується об'єм пам'яті, достатній для максимально можливого розміру списку незалежно від його реального розміру в конкретний момент часу. Реалізація за допомогою покажчиків використає стільки пам'яті, скільки необхідно для зберігання поточного списку, але вимагає додаткову пам'ять для покажчика кожного осередку. Таким чином, у різних ситуаціях за критерієм використаної пам'яті можуть бути вигідні різні реалізації.

3.4 Стеки

Стек – це спеціальний тип списку, у якому всі вставки й видалення виконуються тільки на одному кінці, називаному вершиною (top). Стеки також іноді називають «магазинами», а в англомовній літературі для позначення стеків ще використається абревіатура LIFO (last-in-first-out – останній увійшов – перший вийшов). Інтуїтивними моделями стека можуть бути колода карт на столі при грі в покер, книги, складені в стопку, або стопка тарілок; у всіх цих моделях взяти можна тільки верхній предмет, а додати новий об'єкт можна, тільки поклавши його на верхній. Абстрактні типи дані сімейства STACK (Стек) звичайно використають наступні п'ять операторів:

1. **MAKENULL(S)**. Робить стек *S* порожнім.

2. **TOP(S)**. Повертає елемент із вершини стека *S*. Звичайно вершина стека ідентифікується позицією 1, тоді TOP(S) можна записати в термінах загальних

операторів списку як RETRIEVE(FIRST(S), S).

3. **POP(S)**. Видаляє елемент із вершини стека (виштовхує зі стека), у термінах операторів списку цей оператор можна записати як DELETE(FIRST(S), S).

4. **PUSH(x, S)**. Вставляє елемент *x* у вершину стека *S* (заштовхує елемент у стек). Елемент, що раніше перебував у вершині стека, стає елементом поза вершиною, і т.д. У термінах загальних операторів списку даний оператор можна записати як INSERT(x, FIRST(S), S).

5. **EMPTY(S)**. Ця функція повертає значення true (істина), якщо стек *S* порожній, і значення false (неправда) у протилежному випадку.

Кожну реалізацію списків можна розглядати як реалізацію стеків, оскільки стеки з їхніми операторами є окремими випадками списків з операторами, виконуваними над списками.

3.5 Черги

Інший спеціальний тип списку – черга (queue), де елементи вставляються з одного кінця, називаного заднім (rear), а віддаляються з іншого, переднього (front). Черги також називають «списками типу FIFO» (абревіатура FIFO розшифровується як first-in-first-out: першим увійшов – першим вийшов). Оператори, виконувані над чергами, аналогічні операторам стеків. Істотна відмінність між ними полягає в тому, що вставка нових елементів здійснюється в кінець списку, а не в початок, як у стеках. Крім того, різна устаткована термінологія для стеків і черг. Будемо використати наступні оператори для роботи із чергами:

1. **MAKENULL(Q)** очищує чергу Q, роблячи її порожньою.

2. **FRONT(Q)** – функція, що повертає перший елемент черги Q. Можна реалізувати цю функцію за допомогою операторів списку як RETRIEVE(FIRST(Q), Q).

3. **ENQUEUE(x,Q)** вставляє елемент x у кінець черги Q. За допомогою операторів списку цей оператор можна виконати в такий спосіб: INSERT(x, END(Q), Q).

4. **DEQUEUE(Q)** видаляє перший елемент черги Q. Також реалізується за допомогою операторів списку як DELETE(FIRST(Q), Q).

5. **EMPTY(Q)** повертає значення true тоді й тільки тоді, коли Q є порожньою чергою.

Як і для стеків, будь-яка реалізація списків припустима для подання черг. Однак з огляду на особливість черги (вставка нових елементів тільки з одного, заднього кінця), можна реалізувати оператор ENQUEUE більш ефективно, ніж при звичайному поданні списків.

Практичні завдання

1. Реалізувати АТД лінійний список та провести обчислювальні експерименти з основними операціями над списком. Для цього:
   1.1 Реалізувати структуру даних для зберігання елементів списку.
   1.2 Реалізувати основні операції роботи зі списком.
   1.3 Провести математичний аналіз операцій роботи зі списком. За необхідності, провести емпіричний аналіз для підтвердження результатів математичного аналізу.
   1.4 Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
      – виводити запит на введення кількості елементів списку n;
      – виводити послідовність операцій роботи зі списком з можливістю вибору користувачем необхідної операції.
   1.5 Зробити висновок про ефективність реалізації списку за допомогою статичних структур даних.
2. Реалізувати АТД стек. Визначити, чи є введена строка паліндромом. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   – виводити запит на введення строки;
   – виводити результат (наприклад, «404» – паліндром).
3. Звіт повинен містити:
   – формулювання завдання;
   – лістинг програмного коду;
   – етапи проведення математичного та, за необхідності, емпіричного аналізу операцій роботи зі списком;
   – скриншоти роботи програми для різних вхідних даних для АТД список та стек;
   – відповіді на контрольні запитання.

Тестові завдання

1. Об’єкт масив це _____
   а) структура даних;
   б) абстрактний тип даних;
   в) базовий тип даних;
   г) тип даних користувача.
2. Які способи організації даних є більш ефективним з точки зору використання оперативної пам’яті, яка доступна програмі?
   а) статична структура даних;
   б) динамічна структура даних;
   в) зовнішній файл;
   г) хмарне сховище.
3. _____ – це математична модель із сукупністю операторів, визначених у рамках цієї моделі.
   а) абстрактний тип даних;

б) базовий тип даних;

в) структура даних;

г) жодного з цих варіантів.

4. Ця функція повертає елемент, що стоїть в позиції *p* у списку *L*.

а) RETRIEVE(p, L);

б) DELETE(p, L);

в) LOCATE(x, L);

г) NEXT(p, L).

5. ____ – це спеціальний тип списку, у якому всі вставки й видалення виконуються тільки на одному кінці.

а) стек;

б) дек;

в) черга;

г) піраміда.

Контрольні запитання

1. Що таке абстрактний тип даних? Які АТД Ви знаєте?

2. Які операції характерні для АТД список?

3. Навести приклади використання списків у програмуванні при вирішенні практичних завдань.

4. Які існують підходи до реалізації списків? Оцініть їхню ефективність.

5. Які абстрактні типи даних та структури даних можна використати для зберігання даних у форматі «ключ-значення»?

Тема 4 Метод грубої сили

Мета: вивчити поняття метод проєктування алгоритмів; ознайомитись з прикладами застосування методу грубої сили.

План

1. Поняття про методи проєктування алгоритмів
2. Метод грубої сили в задачах сортування
3. Послідовний пошук і пошук підрядків методом грубої сили
4. Розв'язання геометричних задач методом грубої сили
5. Метод вичерпного перебору

4.1 Поняття про методи проєктування алгоритмів

Метод проєктування алгоритму (algorithm design technique) (або «стратегія», або «принцип») – це універсальний підхід, що застосовується для алгоритмічного розв'язання широкого кола задач, які ставляться у різних областях обчислювальної техніки.

Метод грубої сили являє собою прямий підхід до розв'язання задачі, звичайно заснований безпосередньо на формулюванні задачі й визначеннях використовуваних нею концепцій.

Перефразувати визначення даної стратегії можна простіше: «Нема чого думати, треба діяти!». Найчастіше стратегія грубої сили виявляється найбільш простою у застосуванні.

Як приклад розглянемо задачу піднесення в ступінь: обчислення a^n для деякого числа a й невід'ємного цілого n. Хоча ця задача може здатися тривіальної, вона дозволяє проілюструвати кілька методів розробки алгоритмів, у тому числі й підхід грубої сили. За визначенням піднесення в ступінь

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n \text{ разів}}.$$

Звідси відразу треба найпростіший алгоритм обчислення a^n – шляхом множення на a початкового значення, рівного 1, n раз.

Раніше було розглянуто алгоритм з використанням грубої сили: послідовна перевірка всіх цілих чисел при пошуку gcd (m,n).

Хоча метод грубої сили нечасто дає красиві або ефективні алгоритми, його розгляд не можна опустити, оскільки даний метод являє собою важливу стратегію розробки алгоритмів. *По-перше*, на відміну від інших стратегій, метод грубої сили застосовується до дуже широкого діапазону задач. Це єдиний підхід, для якого істотно складніше вказати задачу, для розв'язання якої його неможна застосувати. Зокрема, метод грубої сили використається для багатьох елементарних, але важливих алгоритмічних задач, таких як обчислення суми n

чисел, пошук найбільшого елемента в списку й тому подібних. *По-друге*, для деяких важливих задач (наприклад, сортування, пошуку, множення матриць, пошуку підрядків) метод грубої сили дає цілком раціональні алгоритми. *По-третє*, вартість розробки більш ефективного алгоритму може виявитися неприйнятною, якщо потрібно розв'язати тільки кілька екземплярів задачі, а алгоритм, заснований на грубій силі, дозволяє розв'язати їх за прийнятний час. *По-четверте*, навіть будучи неефективним у загальному випадку, метод грубої сили може виявитися корисний для розв'язання невеликих за розмірами екземплярів задачі.

Нарешті, алгоритм, заснований на грубій силі, може виявитися важливим для теоретичних або дидактичних цілей, наприклад мірою для визначення ефективності інших алгоритмів для розв'язання даної задачі.

4.2 Метод грубої сили в задачах сортування

Розглянемо застосування методу грубої сили до задачі сортування, що полягає в наступному: даний список з n елементів (наприклад, чисел, символів деякого алфавіту, символьних рядків), які треба розмістити в неспадному порядку. Є десятки алгоритмів, розроблених для розв'язання цієї дуже важливої задачі.

Бульбашкове сортування

Застосування методу грубої сили до задачі сортування складається в порівнянні сусідніх елементів і їхньому обміні, якщо вони перебувають не в належному порядку. Неодноразове використання цієї дії приводить до того, що найбільший елемент «спливає» у кінці списку. Наступний прохід приведе до спливання другого за величиною елемента, і так доти, поки після n-1 ітерації список не буде повністю відсортований, i-ий прохід $(0 \le i \le n-2)$ бульбашкового сортування можна представити наступною діаграмою:

$$A_0, \dots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \dots, A_{n-i-1} \quad | \quad A_{n-i} \le \dots \le A_{n-1}$$

елементи в остаточних позиціях

Нижче наведений псевдокод даного алгоритму.

Алгоритм *BubbleSort* (*A*[0..*n*-1])
// Сортування масиву бульбашковим методом
// Вхідові дані: масив *A*[0..*n*-1] елементів, що упорядковуються
// Виходові дані: масив *A*[0..*n*-1], відсортований у неспадному порядку
for *i* ← 1 **to** *n*-2 **do**
  for *j* ← 0 **to** *n*-2-*i* **do**
    if *A*[*j*+1] < *A*[*j*]
      Обмін *A*[*j*] та *A*[*j*+1]

Кількість порівнянь ключів у даній версії бульбашкового сортування однаково для всіх масивів розміром n. Воно представляється такою сумою:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] =$$
$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Кількість же обмінів елементів залежить від вхідних даних. У найгіршому випадку вона дорівнює кількості порівнянь:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Найчастіше при використанні методу грубої сили перша версія алгоритму може бути вдосконалена за допомогою досить скромних зусиль. Зокрема, наведену «сиру» версію бульбашкового сортування можна поліпшити, скориставшись наступним спостереженням: якщо при проході за списком не зроблено жодного обміну, виходить, список відсортований, і виконання алгоритму можна припинити.

Сортування вибором

Сортування вибором почнемо з пошуку найменшого елемента в списку й обміну його з першим елементом (таким чином, найменший елемент переміститься в остаточну позицію у відсортованому списку). Потім скануємо список, починаючи із другого елемента, у пошуках найменшого елементу серед n - 1 елементу, які залишилися, і обмінюємо знайдений найменший елемент із другим, тобто переміщуємо другий з кінця за величиною елемент в остаточну позицію у відсортованому списку. У загальному випадку, при i-ому проході за списком $(0 \le i \le n-2)$ алгоритм шукає найменший елемент серед останніх n-i елементів і обмінює його з A_i.

$A_0 \le A_1 \le \cdots \le A_{i-1}$ | $A_i, \overleftrightarrow{\ldots}, A_{min}, \ldots, A_{n-1}$

елементи в остаточних позиціях | останні $n-i$ елементів

Після виконання n - 1 проходу список виявляється відсортований. Нижче наведено псевдокод даного алгоритму, у якому для простоти передбачається, що список реалізований у вигляді масиву.

Алгоритм *SelectionSort* (*A*[0..*n*-1])
// Сортування масиву методом вибору
// Вхідові дані: масив *A*[0..*n*-1] елементів, що упорядковуються
// Виходові дані: масив *A*[0..*n*-1], відсортований у неспадному порядку

for $i \leftarrow 1$ **to** n-2 **do**
  $min \leftarrow i$
  for $j \leftarrow i$+1 **to** n-1 **do**
    if $A[j] < A[min]$
      $min \leftarrow j$
  Обмін $A[i]$ та $A[min]$

Аналіз сортування вибором виконується досить просто. Розмір вхідних даних визначається кількістю n елементів у списку. Базовою операцією алгоритму є порівняння ключів $A[j] < A[min]$. Загальна кількість порівнянь залежить тільки від розміру масиву й визначається наступною сумою:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

Обчислимо зазначену суму:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-i-1) = \frac{(n-1)n}{2}.$$

Таким чином, для будь-яких вхідних даних алгоритм сортування вибором належить $\Theta(n^2)$. Помітимо, однак, що кількість обмінів елементів масиву дорівнює $\Theta(n)$, точніше, дорівнює n-1 – по одному для кожної ітерації циклу i. Ця властивість відрізняє сортування вибором від багатьох інших алгоритмів сортування.

4.3 Послідовний пошук і пошук підрядків методом грубої сили

Послідовний пошук

Цей алгоритм просто по черзі порівнює елементи заданого списку із ключем пошуку доти, поки не буде знайдений елемент із зазначеним значенням ключа (успішний пошук) або весь список буде перевірений, але необхідний елемент не знайдений (невдалий пошук). Найчастіше застосовується простий додатковий прийом: якщо додати ключ пошуку в кінець списку, то пошук обов'язково буде успішним, отже, можна забрати перевірку завершення списку в кожній ітерації алгоритму (швидкий послідовний пошук). Далі наведений псевдокод такої поліпшеної версії; передбачається, що Входові дані мають вигляд масиву.

Алгоритм *SequentialSearch*2 (*A*[0..*n*], *K*)
  // Послідовний пошук з використанням ключа пошуку як обмежника
  // Входові дані: масив *A* з *n* елементів та ключ пошуку *K*

```
// Виходові дані: позиція першого елемента масиву A[0..n-1],
//                значення якого співпадає з K; якщо елемент
//                незнайдений, повертається значення -1
A[n] ← K
i ← 0
while A[i] ≠ K do
   i ← i+1
if i < n
   return i
else
   return -1
```

Якщо вихідний список відсортований, можна скористатися ще одним удосконаленням: пошук у такому списку можна припиняти, як тільки зустрінеться елемент, не менший ключа пошуку.

Алгоритм залишається лінійним як у найгіршому, так і в середньому випадку.

Набір усіх записів іменується таблицею або файлом. Алгоритм пошуку має так званий *аргумент* K, і задача полягає в знаходженні запису, для якого K служить ключем.

Послідовний пошук – почати з початку й продовжувати, поки не буде знайдений шуканий ключ, потім зупинитися.

Алгоритм *S* (Послідовний пошук, *Sequential search*):

// Дана таблиця записів $R_1, R_2, \dots, R_N$ з ключами $K_1, K_2, \dots, K_N$ відповідно

// Алгоритм шукає запис з ключем K

// Виходові дані: позиція у вихідній послідовності запису, для якого ключ співпадає з K

Крок S1. Присвоїти змінній $i = 1$.

Крок S2. Якщо $K = K_i$, повернути i та завершити роботу.

Крок S3. Збільшити i на 1.

Крок S4. Якщо $i<=N$, перейти на крок *S2*. У іншому випадку повернути повідомлення про відсутність шуканого запису та завершити роботу.

Алгоритм S можна покращити.

Алгоритм *Q* (Швидкий послідовний пошук, *Quick sequential search*):

// Дана таблиця записів $R_1, R_2, \dots, R_N$ з ключами $K_1, K_2, \dots, K_N$ відповідно

// Алгоритм шукає запис з ключем K

// Виходові дані: позиція у вихідній послідовності запису, для якого ключ співпадає з K

Крок Q1. Присвоїти змінній $i = 1$ та $K_{N+1} = K$.

Крок Q2. Якщо $K = K_i$перейти на крок *Q4*.

Крок Q3. Збільшити i на 1 і перейти на крок *Q2*.

Крок Q4. Якщо $i<=N$, повернути i та завершити роботу. У іншому випадку повернути повідомлення про відсутність шуканого запису та закінчити роботу ($i = N + 1$).

Практичні завдання

1. Реалізувати алгоритм S.
2. Реалізувати алгоритм Q.
3. Реалізації алгоритмів виконують пошук першого входження аргументу пошуку в послідовності випадкових чисел заданого об'єму.
4. Провести математичний аналіз алгоритмів S і Q.
5. Провести емпіричний аналіз алгоритмів. Зрівняти ефективності алгоритмів на вхідних даних різної структури.
6. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об'єму випадкової послідовності й аргументу пошуку;
   - виводити позицію шуканого елемента в списку;
   - передбачити можливість виводу елемента випадкової послідовності по його позиції для перевірки результатів пошуку.
7. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритмів S і Q;
   - висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

Пошук підрядка

Дано символьний рядок довжиною n, що називається текстом, і рядок довжиною m ($m \leq n$), іменований шаблоном (pattern); потрібно знайти в тексті підрядок, що відповідає шаблону. Говорячи точніше, необхідно визначити i – індекс крайнього ліворуч символу першого підрядка в тексті, що відповідає шаблону – тобто $t_i = p_0, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}$:

t_0	…	t_i	…	t_{i+j}	…	t_{i+m-1}	…	t_{n-1}	Текст T
		↕		↕		↕			
		p_0	…	p_j	…	p_{m-1}			Шаблон P

Якщо потрібно знайти всі такі підрядки, алгоритм пошуку підрядка може просто продовжувати роботу до повної перевірки тексту.

Алгоритм, заснований на грубій силі, очевидний: вирівняємо шаблон з початком тексту й почнемо порівнювати відповідні пари символів зліва направо доти, поки не переконаємося, що символи у всіх m парах рівні (у цьому випадку алгоритм може припиняти роботу), або не зустрінемо пари різних символів. В останньому випадку шаблон зміщається на одну позицію вправо, і порівняння символів триває, починаючи з першого символу шаблона й символу у

відповідній позиції в тексті. Помітимо, що остання позиція в тексті, що ще може виступати в ролі початку шуканого підрядка – *n*-*m* (якщо позиції в тексті індексуються значеннями від 0 до *n*-1). Після цієї позиції в тексті залишається занадто мало символів, щоб вони могли відповідати шаблону. Отже, при досягненні зазначеної позиції алгоритм не повинен робити ніяких порівнянь.

Алгоритм *BruteForceStringMatch* (*T*[0..*n*-1], *P*[0..*m*-1])
```
// Пошук підрядка методом грубої сили
// Вхідові дані: масив T[0..n-1] із n символів, що складають текст;
//                масив P [0..m-1] із m елементів, що складають шаблон
// Виходові дані: позиція першого символу в тексті, з якої починається
//                перший шуканий підрядок, що відповідає шаблону;
//                якщо підрядок не знайдений, повертається -1
for i ← 0 to n-m do
    j ← 0
    while j < m and P[j] = T[i+j] do
        j ← j+1
        if j = m
            return i
return -1
```

N	O	B	O	D	Y	–	N	O	T	I	C	E	D	–	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

Рисунок 4.1 – Приклад роботи алгоритму для пошуку підрядка

Зверніть увагу, що в даному прикладі алгоритм практично завжди зміщує шаблон після першого ж порівняння символу. Однак найгірший випадок набагато неприємніше: алгоритм може виконувати всі *m* порівнянь перед зсувом шаблона, і це відбувається в кожній з *n*-*m*+1 спроб.

Таким чином, у найгіршому випадку алгоритм має час роботи $\Theta(nm)$. Однак у випадку типового пошуку слова в тексті природною мовою можна чекати, що більшість зсувів буде виконуватися після невеликої кількості порівнянь. Таким чином, ефективність у середньому випадку повинна бути істотно вище ефективності в найгіршому випадку. Це так і є насправді: можна показати, що при пошуку у випадкових текстах ефективність виявляється лінійною, тобто рівною $\Theta(n+m) = \Theta(n)$. Для пошуку підрядків є велика кількість більш інтелектуальних, і ефективних алгоритмів.

4.4 Розв’язання геометричних задач методом грубої сили

Пошук пари найближчих точок

Задача пошуку пари найближчих точок полягає в тому, щоб у множині з n точок знайти дві, розташовані друг до друга ближче інших. Для простоти будемо розглядати тільки двовимірний випадок, хоча задача може бути поставлена й для більшого числа вимірів. Точки задаються стандартним способом за допомогою декартових координат (x, y), а відстань між двома точками $P_i(x_i, y_i)$ та $P_j(x_j, y_j)$ являє собою стандартна відстань геометрії Евкліда (евклідова відстань)

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Підхід із застосуванням грубої сили для розв’язання цієї задачі приводить до наступного очевидного алгоритму: обчислити відстані між кожною парою точок і знайти пару з найменшою відстанню. Необхідно уникнути повторного обчислення відстані між тими самими точками, тому розглядаємо тільки пари точок (P_i, P_j), для яких $i < j$.

Алгоритм *BruteForceClosestPoints* (P)
// Вхідові дані**:** список P, що містить $n \geq 2$ точок
// $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
// Виходові дані**:** індекси *index*1 та *index*2 пари найближчих точок
dmin ← ∞
for *i* ← 1 **to** *n*-1 **do**
  for *j* ← *i*+1 **to** *n* **do**
    d ← *sqrt*((*x*$_i$-*x*$_j$)**2+(*y*$_i$-*y*$_j$)**2)
// *sqrt* – функція обчислення квадратного кореня,
// ** – піднесення до степеня
    if *d* < *dmin*
      dmin ← *d*; *index*1← *i*; *index*2← *j*;
return *index*1, *index*2

Базовою операцією алгоритму є обчислення відстані між двома точками. Може здатися, що обчислення квадратного кореня є дуже проста операція, як, скажемо, додавання або множення. Однак це не так. Почнемо з того, що для більшості цілих чисел квадратний корінь є ірраціональним числом, так що він може бути обчислений тільки приблизно. Більше того, такі наближені обчислення – аж ніяк не тривіальна задача. Однак усе виявляється набагато цікавіше: обчислення квадратного кореня в алгоритмі цілком можна уникнути, якщо замінити

d ← *sqrt*((*x*$_i$-*x*$_j$)**2+(*y*$_i$-*y*$_j$)**2)

на

$$d \leftarrow (x_i\text{-}x_j)\text{**}2+(y_i\text{-}y_j)\text{**}2,$$

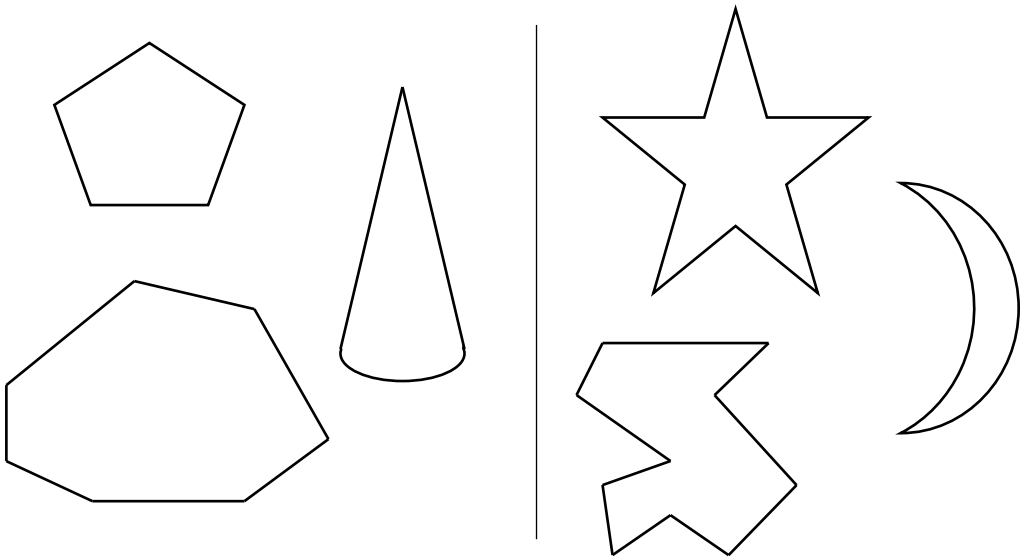
то базовою операцією алгоритму стане піднесення чисел у квадрат. Загальна кількість таких операцій можна обчислити в такий спосіб:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n-i) = 2[(n-1) + (n-2) + \cdots + 1] = \\ = (n-1)n \in \Theta(n^2).$$

Пошук опуклої оболонки

Тепер розглянь іншу задачу – обчислення опуклої оболонки. Почнемо з визначення того, що таке опукла оболонка.

Визначення 1. Множина точок (кінцева або нескінченна) на площині називається опуклою, якщо для будь-яких двох точок *P* та *Q*, що належать даній множині, відрізок з кінцевими точками *P* та *Q* буде повністю належати цій же множині.

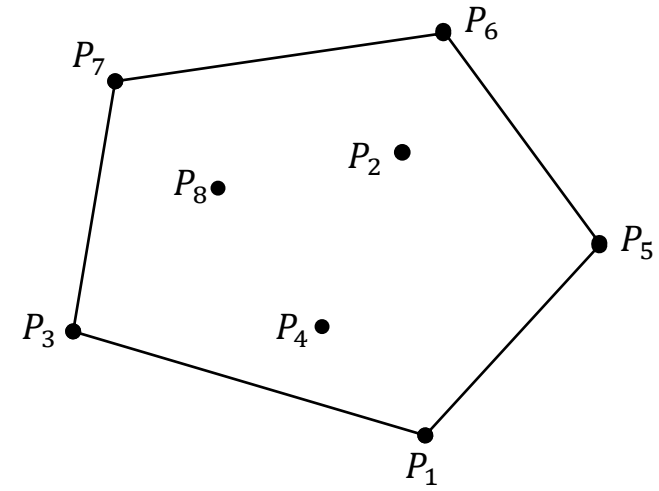


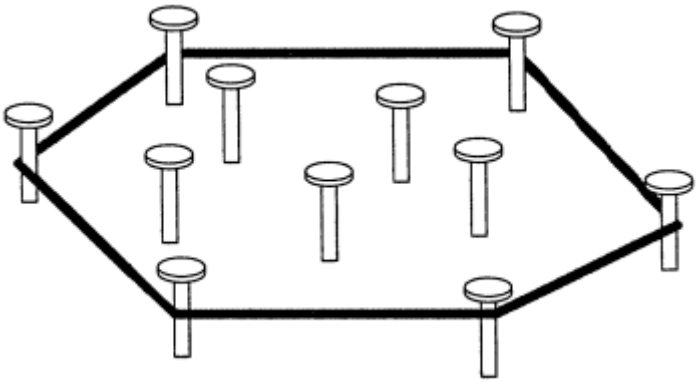
а) б)

Рисунок 4.2 – Множини: а) опуклі; б) які не є опуклими

Інтуїтивно опукла оболонка множини з n точок на площині – це найменший опуклий багатокутник, що містить всі точки множини – усередині або на межі.

Припустимо, що кожна точка представлена цвяхом, вбитим у дерев'яну поверхню, що представляє площину. Візьмемо гумову стрічку, розтягнемо її так, щоб вона охоплювала всі цвяхи, і відпустимо. Опукла оболонка являє собою область, обмежену відпущеною гумовою стрічкою.





Формальне визначення опуклої оболонки може бути застосовано до довільних множин, що включають множини точок, що лежать на одній прямій, і інші подібні екстремальні випадки.

Визначення 2. Опукла оболонка множини точок *S* являє собою найменшу опуклу множину, що містить *S*. («Найменша» означає, що опукла оболонка *S* повинна бути підмножиною будь-якої опуклої множини, що містить *S*).

Теорема 1. Опукла оболонка довільної множини *S*, що складається з $n > 2$ точок, що не належать одній прямій, являє собою опуклий багатокутник з вершинами в деяких точках *S*. (Якщо всі точки лежать на одній прямій, то багатокутник вироджується у відрізок прямої, дві кінцеві точки якого однаково належать множині *S*).

Обчислення опуклої оболонки являє собою задачу побудови опуклої оболонки для множини *S*, що складає з *n* точок. Для її розв'язання необхідно знайти точки, які служать вершинами шуканого багатокутника. Вершини такого багатокутника називаються «кутовими точками» (extreme points). За визначенням, *кутова точка опуклої множини* – точка, що не є внутрішньою точкою будь-якого відрізку з кінцевими точками з даної множини. Наприклад, кутовими точками трикутника є три його вершини; кутовими точками кола – всі точки його кола; кутовими точками опуклої оболонки восьми точок, показаної на рисунку вище – точки P_1, P_5, P_6, P_7, P_3.

Кутові точки мають ряд особливих властивостей, відсутніх в інших точок опуклої множини. Одна з таких властивостей використовується дуже важливим

алгоритмом – так званим симплекс-методом (simplex method). Цей алгоритм призначений для розв'язання задач лінійного програмування, тобто задач пошуку мінімуму або максимуму лінійної функції від n змінних, на які накладаються лінійні обмеження. Зараз же кутові точки нас цікавлять остільки, оскільки їхня ідентифікація вирішує задачу обчислення опуклої оболонки. У дійсності для розв'язання цієї задачі треба знати не тільки кутові точки, але й те, які пари цих точок повинні бути з'єднані для одержання межі опуклої оболонки. Відповідь на це питання може бути отримана перерахуванням кутових точок у порядку обходу по або проти годинникової стрілки.

Є простий, але неефективний алгоритм, заснований на наступній властивості відрізків, що утворять межу опуклої оболонки: *відрізок, що з'єднує точки P_i і P_j множини, що складається з n точок, є частиною межі опуклої оболонки тоді й тільки тоді, коли всі інші точки лежать по один бік від прямої, що проходить через ці дві точки*.

Перевірка кожної пари точок дає список відрізків, які становлять границю опуклої оболонки. Ефективність даного алгоритму дорівнює $0(n^3)$.

4.5 Метод вичерпного перебору

Вичерпний перебір (exhaustive search) являє собою підхід до комбінаторних задач із позиції грубої сили. Він припускає генерацію всіх можливих елементів з області визначення задачі, вибір тих з них, які задовольняють обмеженням, що накладається умовою задачі, і наступний пошук потрібного елемента (наприклад, оптимального значення цільової функції задачі).

Задача комівояжера

Треба знайти такий найкоротший шлях по заданим n містах, щоб кожне місто відвідувалося тільки один раз і кінцевим пунктом виявилися місто, з якого починалася подорож. Проблему зручно змоделювати за допомогою зваженого графа, вершини якого представляють міста, а ваги ребер визначають відстані. Після цього задача може бути сформульована як задача пошуку найкоротшого *гамильтонового циклу* (Hamiltonian circuit) неорієнтованого графа.

Легко бачити, що гамильтонів цикл можна також визначити як послідовність $n + 1$ суміжної вершини $v_{i_0}, \dots, v_{i_{n-1}}, v_{i_0}$, де перша вершина в послідовності збігається з останньої, у той час як всі інші n-1 вершини різні. Далі, без втрати загальності можна припустити, що всі цикли починаються й закінчуються в одній конкретній вершині. Виходить, можна одержати всі можливі маршрути, генеруючи всі перестановки n-1 проміжного міста, обчислюючи довжину відповідних шляхів і знаходячи найкоротший з них. На рис. показаний невеликий екземпляр даної задачі і його розв'язання описаним методом.

Шлях	Довжина	
$a \to b \to c \to d \to a$	l=2+8+1+7=18	
$a \to b \to d \to c \to a$	l=2+3+1+5=11	Оптимальний
$a \to c \to b \to d \to a$	l=5+8+3+7=23	
$a \to c \to d \to b \to a$	l=5+1+3+2=11	Оптимальний
$a \to d \to b \to c \to a$	l=7+3+8+5=23	
$a \to d \to c \to b \to a$	l=7+1+8+2=18	

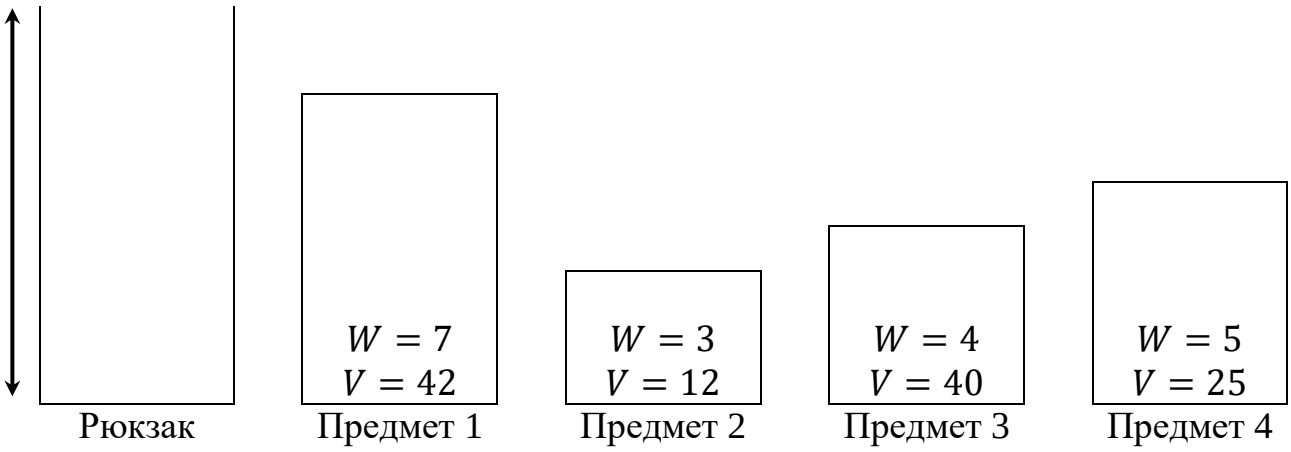
При уважному розгляді рис. можна виявити три пари обходів, які відрізняються друг від друга тільки напрямком. Таким чином, можна знизити кількість перестановок вершин удвічі. Однак це поліпшення алгоритму не дає помітного ефекту. Загальна кількість перестановок залишається рівним (n-1)!/2, що робить вичерпний перебір неприйнятним для всіх значень n, крім невеликих. Помітимо також, що якби не поставлене обмеження, щоб всі шляхи починалися в одній і тій же вершині, то кількість перестановок бути б в n раз більше.

Задача про рюкзак

Дано n предметів вагою $w_1, \dots, w_n$ й ціною $v_1, \dots, v_n$, а також рюкзак, що витримує загальну вагу W. Потрібно знайти підмножину предметів, яку можна розмістити в рюкзаку, і яка має при цьому максимальну вартість.

Вичерпний перебір у цій задачі приводить до розгляду всіх підмножин даної множини з n предметів, обчисленню загальної ваги кожної з них для того, щоб з'ясувати, чи допустимий такий набір предметів (тобто чи не перевершує його загальна вага можливості рюкзака), і вибору із допустимих підмножини з максимальною вагою.

Приклад.



Оскільки загальна кількість підмножин n-елементної множини дорівнює 2^n, то вичерпний перебір приводить до алгоритму порядку $\Omega(2^n)$, незалежно від того, наскільки ефективним методом генеруються підмножини, що розглядаються.

Підмножина	Загальна вага	Загальна вартість
Ø	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	36
{1, 3}	11	Неприпустимо
{1, 4}	12	Неприпустимо
{2, 3}	7	52
{2, 4}	8	37
{3, 4}	**9**	**65**
{1, 2, 3}	14	Неприпустимо
{1, 2, 4}	15	Неприпустимо
{1, 3, 4}	16	Неприпустимо
{2, 3, 4}	12	Неприпустимо
{1, 2, 3, 4}	19	Неприпустимо

Таким чином, застосування методу вичерпного перебору до задач комівояжера й про рюкзак приводить до винятково неефективних алгоритмів для будь-яких вхідних даних. Насправді ці дві задачі являють собою найбільш відомі приклади так званих NP-складних задач (NP-hard problems). Для жодної з NP-складних задач не відомий алгоритм, що вирішує їх за поліноміальний час. Більше того, більшість вчених-кібернетиків сходяться в думці, що такі алгоритми не існують взагалі, хоча це важливе припущення ніким не доведено.

Тестові завдання

1. Клас ефективності алгоритмів сортування, отриманих методом грубої сили:

а) квадратичний;
б) логарифмічний;
в) кубічний;
г) лінійний.

2. Який найбільш вдалий клас ефективності серед різних алгоритмів та задач, що розв’язуються методом грубої сили?

а) квадратичний;
б) логарифмічний;
в) кубічний;
г) лінійний.

3. Яка задача зводяться до розгляду всіх підмножин даної множини з *n* предметів?

а) задача про рюкзак;
б) задача комівояжера;
в) пошук опуклої оболонки;

г) пошук пари найближчих точок.

4. Який з цих алгоритмів відноситься до лінійного класу ефективності?

а) швидкий послідовний пошук;

б) бінарний пошук;

в) бульбашкове сортування;

г) вичерпний перебір у задачі про рюкзак.

5. Для якої задачі вичерпний перебір призводить до алгоритму з факторіальним класом ефективності?

а) задача про рюкзак;

б) задача комівояжера;

в) пошук опуклої оболонки;

г) пошук пари найближчих точок.

Контрольні запитання

1. Що таке метод розробки (проєктування) алгоритмів?
2. У чому полягає особливість методу грубої сили?
3. Сформулюйте задачу пошуку в загальному виді.
4. Назвіть задачі, для яких ефективні алгоритми можуть бути розроблені методом грубої сили.
5. Чи завжди можна розробити алгоритм методом грубої сили для сформульованої задачі?

Тема 5 Метод декомпозиції

Мета: ознайомитись з прикладами застосування методу декомпозиції до задач сортування та пошуку даних.

План

1. Основи методу декомпозиції
2. Бінарний пошук
3. Сортування злиттям
4. Швидке сортування
5. Обхід бінарного дерева
6. Множення великих цілих чисел і алгоритм множення матриць Штрассена

5.1 Основи методу декомпозиції

Метод декомпозиції (він же метод «розділяй і пануй»), імовірно, найбільш популярний метод розробки алгоритмів. Ряд дуже ефективних алгоритмів являють собою реалізації цієї загальної стратегії.

Алгоритми, засновані на методі декомпозиції, працюють у відповідності з таким планом:

1. Екземпляр задачі розбивається на кілька менших екземплярів тієї ж задачі, в ідеалі – однакового розміру.
2. Розв'язуються менші екземпляри задачі (частіше рекурсивно, хоча іноді для невеликих екземплярів застосовується який-небудь інший алгоритм).
3. При необхідності розв'язання вихідної задачі знаходиться шляхом комбінації рішень менших екземплярів.

Схема методу декомпозиції показана на рис. 5.1, де наведений випадок поділу задачі на дві рівні за розмірами підзадачі, що є, мабуть, найпоширенішою ситуацією (принаймні для алгоритмів декомпозиції, розроблених для виконання на однопроцесорному комп'ютері).

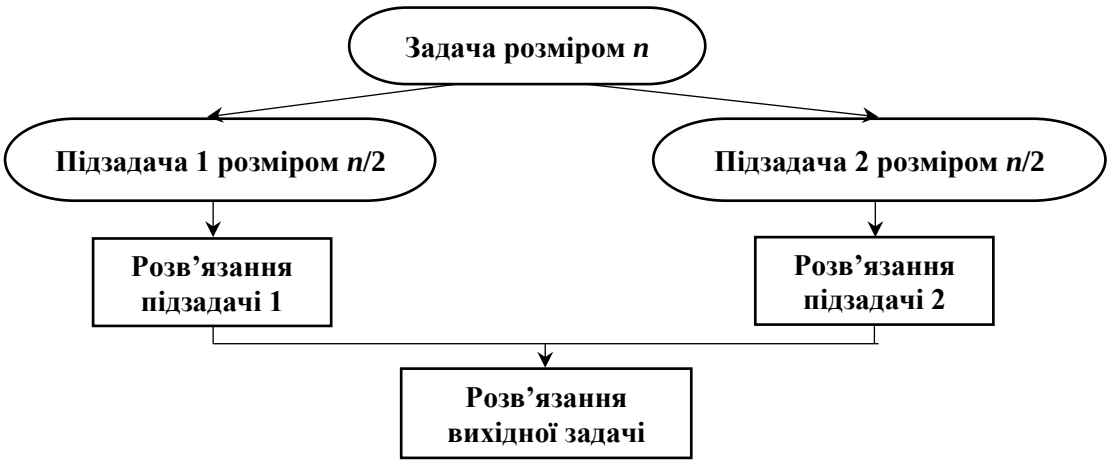


Рисунок 5.1 – Схема розв'язання задачі методом декомпозиції

Як приклад розглянемо задачу обчислення суми чисел $a_0, \ldots, a_{n-1}$.

Якщо $n > 1$, задачу можна розділити на два екземпляри тієї ж задачі: обчислення суми перших $\left\lfloor \frac{n}{2} \right\rfloor$ чисел і обчислення суми $\left\lceil \frac{n}{2} \right\rceil$ чисел, що залишилися (зрозуміло, що при $n = 1$ як відповідь просто вертається значення a_0). Як тільки кожна із цих двох сум буде обчислена (із застосуванням описаного методу, тобто рекурсивно), можна скласти їхні значення, щоб одержати остаточну відповідь:

$$a_0 + \cdots + a_{n-1} = \left(a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}\right).$$

Описаний алгоритм не є ефективним способом підсумовування n чисел.

Таким чином, не кожний алгоритм на основі декомпозиції ефективніше алгоритму, заснованого на грубій силі. Дуже часто час, витрачений на виконання алгоритму на основі декомпозиції, виявляється менше, ніж час розв'язання задачі якимось іншим методом. Метод декомпозиції дав кібернетиці ряд дуже важливих і ефективних алгоритмів. Варто пам'ятати про те, що метод декомпозиції ідеально підходить для паралельних обчислень, коли усі підзадачі можуть одночасно розв'язуватись кожна власним процесором.

Приклад підсумовування ілюструє найбільш типовий випадок декомпозиції: екземпляр задачі розміром n ділиться на два екземпляри розміром $n/2$. У загальному випадку екземпляр задачі розміру n може бути розділений на кілька екземплярів розміром n/b, a з яких потрібно розв'язати (тут a й b – константи; $a > 1$ і $b > 1$). Поклавши для спрощення аналізу, що розмір n дорівнює ступені b, одержуємо наступне рекурентне співвідношення для часу роботи алгоритму $T(n)$:

$$T(n) = aT(n/b) + f(n),$$

де $f(n)$ – функція, що враховує витрати часу на поділ задачі на менші підзадачі й комбінування розв'язків підзадач (у прикладі з підсумовуванням чисел $a = b = 2$ і $f(n) = 1$). Дане рекурентне співвідношення називається узагальненим рекурентним рівнянням декомпозиції (general divide-and-conquer recurrence). Очевидно, що порядок зростання його розв'язків $T(n)$ залежить від значень констант a й b і порядку зростання функції $f(n)$. Аналіз ефективності множини алгоритмів, заснованих на декомпозиції, істотно спрощується наступною теоремою.

Основна теорема. Якщо в рекурентному рівнянні декомпозиції $f(n) \in \Theta(n^d)$, де $d \geq 0$, то

$$T(n) \in \begin{cases} \Theta(n^d) & \text{якщо } a < b^d \\ \Theta(n^d \log n) & \text{якщо } a = b^d \\ \Theta\left(n^{\log_b a}\right) & \text{якщо } a > b^d \end{cases}$$

(Аналогічні результати виконуються й для позначень O, Ω).

Можна вказати клас ефективності алгоритму, не розв'язуючи саме рекурентне рівняння. Звичайно ж, цей підхід дозволяє встановити порядок зростання розв'язків, не визначаючи невідомі множники, які можна знайти, тільки розв'язавши рекурентне рівняння.

5.2 Бінарний пошук

Бінарний пошук являє собою дуже ефективний алгоритм для пошуку у відсортованому масиві. Він працює шляхом порівняння шуканого ключа K із середнім елементом масиву $A[m]$. Якщо вони рівні, алгоритм припиняє роботу. У протилежному випадку та ж операція рекурсивно повторюється для першої половини масиву, якщо $K < A[m]$, і для другої – якщо $K > A[m]$:

$$\underbrace{A[0] \dots A[m-1]} \qquad K \updownarrow A[m] \qquad \underbrace{A[m+1] \dots A[n-1]}$$

Пошук у цій частині, якщо $K < A[m]$

Пошук у цій частині, якщо $K > A[m]$

Як приклад знайдемо ключ, що дорівнює 70, застосовуючи алгоритм бінарного пошуку до масиву:

3	14	27	31	39	42	55	70	74	81	85	93	98

Ітерації алгоритму показані в наступній таблиці:

Індекс	0	1	2	3	4	5	6	7	8	9	10	11	12
Значення	3	14	27	31	39	42	55	70	74	81	85	93	98
Ітерація 1	*l*						*m*						*r*
Ітерація 2								*l*		*m*			*r*
Ітерація 3								*l, m*	*r*				

Хоча зрозуміло, що бінарний пошук заснований на рекурсії, його дуже легко реалізувати як нерекурсивний алгоритм. Псевдокод нерекурсивної версії:

Алгоритм *BinarySearch* (*A*[0..*n*-1], *K*)
// Нерекурсивний бінарний пошук
// Входові дані: масив *A*[0..*n*-1], відсортований у зростаючому порядку,
// та шуканий ключ *K*
// Виходові дані: індекс елементу масиву, рівного *K*,

```
//                    або -1, якщо такого елементу немає
l ← 0; r ← n-1;
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m]
            return m
    else if K < A[m]
            r ← m-1
    else
            l ← m+1
return -1
```

Стандартний спосіб аналізу ефективності бінарного пошуку складається в підрахунку кількості порівнянь шуканого ключа з елементами масиву. Крім того, з міркувань простоти, будемо вважати, що використаються потрійні порівняння, тобто що одне порівняння K та $A[m]$ дозволяє визначити, чи K менше, чи більше, чи дорівнює значенню $A[m]$.

Знайдемо кількість порівнянь $C_{worst}(n)$ у найгіршому випадку. У найгірший випадок входять всі масиви, які не містять шуканого ключа (крім них у цей розряд попадають і деякі масиви, пошук у які завершується успішно). Оскільки після одного порівняння алгоритм попадає в ту ж ситуацію, що й до порівняння, тільки масив, у якому ведеться пошук, стає вдвічі менше, можна записати наступне рекурентне співвідношення для $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \text{ для } n > 1, C_{worst}(1) = 1.$$
$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$

Складний аналіз показує, що ефективність бінарного пошуку в середньому тільки небагато краща від такої у найгіршому випадку:

$$C_{avg}(n) \approx \log_2 n.$$

Рекурсивну версію алгоритму наведено далі.

Алгоритм *B* (Бінарний пошук, *Binary search*):

// Дана таблиця записів $R_1, R_2, \ldots, R_N$ з ключами $K_1 < K_2 < \ldots < K_N$ відповідно

// Алгоритм шукає запис з ключем K

// Виходові дані: позиція у вихідній послідовності запису, для якого ключ співпадає з K

Крок B1. Присвоїти змінним $l = 1, u = N$.

Крок B2. Якщо $u < l$, алгоритм завершується невдало в іншому випадку присвоїти змінній значення $i = \left\lceil \frac{u+l}{2} \right\rceil$.

Крок B3. Якщо $K < K_i$, перейти на крок *B4*, якщо $K > K_i$, перейти на крок *B5*.

Крок B4. Присвоїти змінній $u = i - 1$ і перейти на крок *B2*.

Крок B5. Присвоїти змінній $l = i + 1$ і перейти на крок *B2*.

Практичні завдання

1. Реалізувати алгоритм B.
2. Провести математичний аналіз алгоритму B.
3. Провести емпіричний аналіз алгоритму B.
4. Для тих самих вхідних даних виконати пошук за алгоритмом B та за алгоритмом Q (швидкий послідовний пошук). Провести емпіричний порівняльний аналіз двох алгоритмів.
5. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об'єму випадкової послідовності й аргументу пошуку;
   - виводити позицію шуканого елемента в списку;
   - передбачити можливість виводу елемента випадкової послідовності по його позиції для перевірки результатів пошуку.
6. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму B;
   - емпіричний порівняльний аналіз алгоритмів B та Q, висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

5.3 Сортування злиттям

Сортування злиттям (*MergeSort*) є чи не найбільш популярним прикладом успішного застосування методу декомпозиції. Сортування заданого масиву $A[0..n-1]$ відбувається шляхом поділу його на дві половини: $A\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]$ та $A\left[\left\lfloor\frac{n}{2}\right\rfloor..n-1\right]$, рекурсивного сортування кожної половини й злиття двох відсортованих половин в один відсортований масив.

Алгоритм *Mergesort* (*A*[0..*n*-1])
// Рекурсивне сортування масиву *A*[0..*n*-1]
// Вхідові дані: масив елементів *A*[0..*n*-1], що упорядковуються

// Виходові дані: відсортований в неспадному порядку масив $A[0..n\text{-}1]$
if $n > 1$
копіювати $A\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]$ у $B\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]$
копіювати $A\left[\left\lfloor\frac{n}{2}\right\rfloor..n-1\right]$ у $C\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]$
$Mergesort\left(B\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]\right)$
$Mergesort\left(C\left[0..\left\lfloor\frac{n}{2}\right\rfloor - 1\right]\right)$
$Merge\ (B, C, A)$

Злиття двох відсортованих масивів можна виконати в такий спосіб:

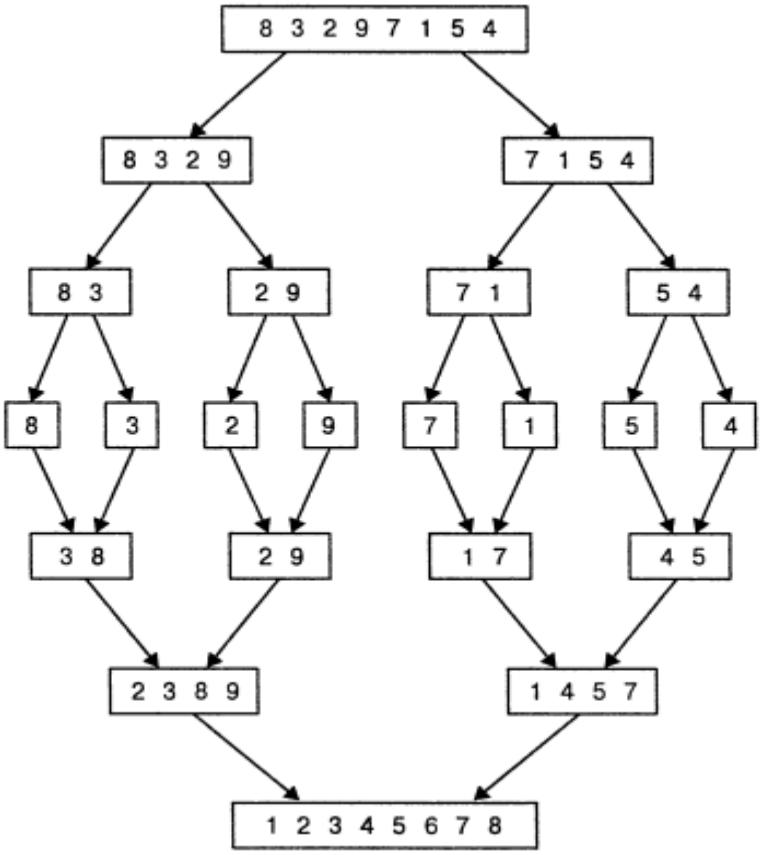
Крок 1. Два покажчики (індекси масивів) після ініціалізації вказують на перші елементи масивів, що зливаються.

Крок 2. Елементи, на які вказують покажчики, порівнюються, і менший з них додається в новий масив.

Крок 3. Індекс меншого елемента збільшується, і він вказує на елемент, що безпосередньо слідує за тільки що скопійованим. Ця операція повторюється доти, поки не буде вичерпаний один з масивів, що зливаються, після чого ті елементи другого масиву, що залишилися, просто додаються в кінець нового масиву.

Алгоритм *Merge* (*B*[0..*p*-1], *C*[0..*q*-1], *A*[0..*p*+*q*-1])
// Злиття двох відсортованих масивів у один
// Входові дані: відсортовані масиви *B*[0..*p*-1] та *C*[0..*q*-1]
// Виходові дані: відсортований масив *A*[0..*p*+*q*-1]),
// що складається із елементів масивів *B* та *C*
$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
if $B[i] \leq C[j]$
$A[k] \leftarrow B[i]; i \leftarrow i+1$
else
$A[k] \leftarrow C[j]; j \leftarrow j+1$
$k \leftarrow k+1$
if $i = p$
копіювати $C[j..q-1]$ у $A[k..p+q-1]$
else
копіювати $B[i..p-1]$ у $A[k..p+q-1]$

Приклад. Робота алгоритму зі списком 8, 3, 2, 9, 7, 1, 5, 4.



Оцінимо ефективність сортування злиттям. Покладемо для простоти, що n є ступенем 2. Рекурентне співвідношення для кількості виконуваних порівнянь ключів $C(n)$ виглядає в такий спосіб:

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ для } n > 1, C(1) = 0.$$

Проаналізуємо величину $C_{merge}(n)$, що дорівнює кількості порівнянь ключів у процесі злиття. На кожному кроці виконується в точності одне порівняння, після якого загальна кількість елементів у двох масивах, що зливаються, зменшується на 1. У найгіршому разі жоден з масивів не вичерпується до самого кінця, поки в іншому масиві не залишиться тільки один елемент (наприклад, найменші елементи надходять із двох масивів по черзі). Отже, у найгіршому випадку $C_{merge}(n) = n - 1$, і одержуємо рекурентне співвідношення

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \text{ для } n > 1, C_{worst}(1) = 0.$$

Відповідно до основної теореми, $C_{worst}(n) = n \log_2 n + n - 1$.

Кількість порівнянь ключів, виконуваних сортуванням злиттям, у найгіршому разі досить близько до теоретичного мінімуму кількості порівнянь для будь-якого алгоритму сортування, заснованого на порівняннях. Основний недолік сортування злиттям – необхідність додаткової пам`яті, кількість якої лінійно пропорційна розміру вхідних даних.

Практичні завдання

1. Реалізувати алгоритм *MergeSort*.
2. Провести математичний аналіз алгоритму.
3. Провести емпіричний аналіз алгоритму.
4. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об'єму випадкової послідовності;
   - виводити генеровану випадковим чином послідовність у файл;
   - виводити відсортовану послідовність у файл із тим же ім'ям, але іншим розширенням.
5. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму;
   - висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

До звіту додається два файли з різними розширеннями (див. п. 4.)

5.4 Швидке сортування

Швидке сортування (quicksort) – ще один важливий алгоритм сортування, заснований на методі декомпозиції. На відміну від сортування злиттям, що розділяє елементи масиву відповідно до їх положення в масиві, швидке сортування розділяє елементи масиву відповідно до їх значень. А саме таке сортування виконує перестановку елементів даного масиву $A[0..n\text{-}1]$ для одержання *розбивки* (partition), коли всі елементи до деякої позиції s не перевищують елемента $A[s]$, а елементи після позиції s не менше $A[s]$:

$$\underbrace{A[0] \dots A[s-1]}_{\text{Всі елементи} \le A[s]} \qquad A[s] \qquad \underbrace{A[s+1] \dots A[n-1]}_{\text{Всі елементи} \ge A[s]}$$

Вочевидь, що після розбивки $A[s]$ перебуває в остаточній позиції у відсортованому масиві, і можна сортувати два підмасива елементів до й після $A[s]$ незалежно (наприклад, тим же самим методом).

Алгоритм *Quicksort* ($A[l..r]$)
// Сортування масиву методом швидкого сортування
// Вхідові дані: підмасив елементів $A[l..r]$ масиву $A[0..n\text{-}1]$, що
// визначається початковим та кінцевим індексами l та r

```
// Виходові дані: підмасив елементів A[l..r],
//                відсортований в неспадному порядку
if l < r
   s ← Partition (A[l..r])
// s – позиція розбиття
   Quicksort(A[l..s − 1])
   Quicksort(A[s + 1..r])
```

Розбиття масиву $A[0..n\text{-}1]$ і, у загальному випадку, його підмасиву $A[l..r]$ $(0 \le l < r \le n-1)$ можна виконати в такий спосіб. Спочатку вибирається елемент, щодо якого буде виконуватися розбиття. У силу важливості ролі цього елемента він називається *опорним* (pivot). Є ряд різних стратегій для вибору опорного елемента. Скористаємося найпростішою стратегією – виберемо як опорний перший елемент підмасиву: $p = A[l]$.

Є також ряд різних процедур перестановки елементів для розбиття. Будемо використати ефективний метод, заснований на двох проходах підмасиву – зліва направо і справа наліво, і при кожному проході елементи масиву будуть порівнюватися з опорним елементом. Прохід зліва направо починається із другого елемента. Оскільки передбачається, що елементи, менші опорного, перебуватимуть у першій частині підмасиву, перший прохід пропускає елементи, менші опорного, і зупиняється, зустрівши перший елемент, що не менше опорного. Прохід справа наліво починається з останнього елемента підмасиву. Оскільки треба, щоб всі елементи, більші опорного, перебували в другій частині підмасиву, при цьому проході опускаються всі елементи, більші опорного, і прохід зупиняється, зустрівши перший елемент, що не перевищує опорний.

Залежно від того, чи перетнулися індекси сканування, можливі три ситуації. Якщо індекси сканування i і j не перетнулися, тобто $i < j$, то просто обмінюються елементи $A[i]$ і $A[j]$ і продовжуємо сканування шляхом збільшення i і зменшення j:

		$\rightarrow i$		$j \leftarrow$	
p	всі елементи $\le p$	$\ge p$	…	$\le p$	всі елементи $\ge p$

Якщо при скануванні відбулося перетинання індексів, тобто $i > j$, то виконується розбиття обміном опорного елементу із $A[j]$:

		$j \leftarrow$	$\rightarrow i$	
p	всі елементи $\le p$	$\ge p$	$\le p$	всі елементи $\ge p$

І, нарешті, якщо при скануванні індекси зупинилися на одному елементі, тобто $i = j$, то значення цього елемента повинне дорівнювати p. Отже, матимемо наступне розбиття масиву:

		$\rightarrow i = j \leftarrow$	
p	всі елементи $\leq p$	$= p$	всі елементи $\geq p$

Останній випадок можна об'єднати з випадком перетинання індексів ($i > j$), обмінюючи опорний елемент із $A[j]$ при $i \geq j$.

Псевдокод, що реалізує описану процедуру розбивки такий:

```
Алгоритм Partition (A[l..r])
    // Розбиття підмасиву з використанням першого елементу як опорного
    // Входові дані: підмасив елементів A[l..r] масиву A[0..n-1], що
    //                визначається лівим та правим індексами l та r (l < r)
    // Виходові дані: розбиття A[l..r]; при цьому позиція розбиття
    //                повертається як значення функції
    p ← A[l]
    i ← l; j ← r+1
    repeat
       repeat
            i ← i+1
       until A[i] ≥ p
       repeat
            j ← j-1
       until A[i] ≤ p
       swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    // Відміна останнього обміну при i ≥ j
    swap(A[l], A[j])
    return j
```

Обговорення ефективності швидкого сортування почнемо із зауваження про те, що кількість порівнянь ключів, виконаних до розбиття, досягає величини $n+1$, якщо індекси перетинаються, і n, якщо збігаються. Якщо всі розбиття виявляються посередині відповідних підмасивів, реалізується найкращий випадок. Кількість порівнянь ключів у найкращому випадку, $C_{best}(n)$, задовольняє наступний рекурентному співвідношенню:

$$C_{best}(n) = 2C_{best}(n/2) + n \text{ при } n > 1, C_{best}(1) = 0.$$

Відповідно до основної теореми, $C_{best}(n) \in \Theta(n \log_2 n)$.

У найгіршому випадку всі розбиття виявляються такими, що один з підмасивів порожній, а розмір другого на 1 менше розміру масиву, що розбивається. Така ситуація виникає, зокрема, у зростаючому масиві, тобто для вхідних даних, для яких задача сортування вже розв'язана.

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2),$$
$$C_{avg}(n) \approx 2n \ln n \approx 1{,}38n \log_2 n.$$

Таким чином, алгоритм швидкого сортування в середньому випадку виконує порівнянь ключів усього на 38 % більше, ніж у найкращому випадку.

Виходячи з важливості швидкого сортування, багато років робилися спроби поліпшити базовий алгоритм. Серед інших удосконалень, відкритих різними дослідниками, – поліпшені методи вибору опорного елемента (наприклад, розбиття на основі медіани трьох елементів, коли як опорний елемент використається медіана крайнього зліва, справа та середнього елементів масиву), перемикання на більш просте сортування для малих підмасивів, видалення рекурсії (так зване нерекурсивне швидке сортування). Згідно із провідним експертом в області швидкого сортування Седжвиком (R. Sedgewick), всі разом ці поліпшення можуть знизити час роботи алгоритму на 20-25 %.

Практичні завдання

1. Реалізувати алгоритм *QuickSort*.
2. Провести порівняльний аналіз алгоритмів сортування вставками, алгоритму сортування злиттям, пірамідальне сортування та швидке сортування.
3. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об’єму випадкової послідовності;
   - виводити генеровану випадковим чином послідовність у файл;
   - виводити відсортовану послідовність у файл із тим же ім’ям, але іншим розширенням;
   - виводити службову інформацію про параметри роботи алгоритму (час роботи, кількість ітерацій тощо).
4. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - порівняльний аналіз алгоритмів сортування вставками, сортування злиттям, пірамідальне сортування та швидке сортування (виконання завдання передбачає заповнення зведеної таблиці даними, отриманими в результаті проведення розрахунків для різних вхідних даних і побудову графіку);
   - висновок щодо отриманих результатів в контексті проведеного порівняльного аналізу;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

До звіту додається два файли з різними розширеннями (див. п. 3.)

5.5 Обхід бінарного дерева

Розглянемо, яким чином метод декомпозиції може бути застосований до бінарних дерев. Бінарне дерево *T* визначається як скінченна множина вузлів, що може бути або порожньою, або складатися з кореня й двох бінарних дерев *TL* і *TR*, що не перетинаються й називаються, відповідно, лівим і правим піддеревами кореня. Зазвичай говоримо про бінарне дерево як про окремий випадок упорядкованого дерева.

Оскільки за визначенням бінарне дерево ділиться на дві менші структури такого ж типу – ліве піддерево й праве піддерево, багато задач, пов'язаних з бінарними деревами, можуть розв'язуватися за допомогою методу декомпозиції. Як приклад розглянемо рекурсивний *алгоритм визначення висоти бінарного дерева*. Згадаємо, що висота дерева визначається як довжина самого довгого шляху від кореня до листа. Отже, її можна обчислити, вибравши максимальне значення серед висот лівого й правого піддеревьев кореня й додавши до нього 1 (щоб урахувати додатковий рівень кореня). Помітимо також, що зручно визначити висоту порожнього дерева рівної -1. Отже, матимемо наступний рекурсивний алгоритм:

Алгоритм *Height* (*T*)
// Рекурсивне обчислення висоти дерева
// Вхідові дані: бінарне дерево *T*
// Виходові дані: висота бінарного дерева *T*
if *T* = *Ø*
return -1
else
return $\max\{Height(T_L), Height(T_R)\} + 1$

Розмір екземпляра даної задачі виміряється кількістю вузлів *n*(*T*) бінарного дерева *T*. Вочевидь кількість порівнянь, виконуваних для обчислення максимального із двох чисел, і кількість додавань *A*(*n*(*T*)), що виконується алгоритмом, однакове. Для *A*(*n*(*T*)) можна записати наступне рекурентне рівняння:

$$A\big(n(T)\big) = A\big(n(T_L)\big) + A\big(n(T_R)\big) + 1 \text{ при } n(T) > 0, A(0) = 0,$$
$$A(n) = n.$$

Найбільш важливі алгоритми бінарних дерев – це три класичних обходи бінарних дерев – у прямому порядку (preorder traversal), симетричний, або центрований (inorder traversal), і у зворотному порядку (postorder traversal). При всіх зазначених обходах вузли дерева відвідуються рекурсивно, тобто відвідуються корінь і його ліве й праве піддерева, і відмінність обходів тільки в послідовності відвідувань:

– при обході в прямому порядку спочатку відвідується корінь дерева, а потім ліве й праве піддерева (у зазначеному порядку);
– при симетричному обході корінь відвідується після лівого піддерева, але перед відвідуванням правого;
– при обході у зворотному порядку корінь відвідується після лівого й правого піддерев (у зазначеному порядку).

5.6 Множення великих цілих чисел і алгоритм множення матриць Штрассена

Множення великих цілих чисел

У деяких додатках, особливо в сучасної криптології, потрібно працювати із цілими числами, довжина яких перевищує 100 десяткових цифр. Зрозуміло, що таке число занадто велике, щоб розмістити його в одному слові сучасного комп'ютера, так що для роботи з ним потрібні спеціальні підходи.

Якщо використати класичний алгоритм множення двох n-значних чисел у стовпчик, то кожна з n цифр одного числа буде множитися на кожну з n цифр другого числа, що в результаті дає нам n^2 множень цифр.

Для демонстрації ідеї, що лежить в основі алгоритму, розглянемо множення двох двозначних чисел, 23 і 14. Ці числа можна представити в такий спосіб:

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ та } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Перемножимо їх:

$$23 \cdot 14 = (2 \cdot 10^1 + 3 \cdot 10^0) \cdot (1 \cdot 10^1 + 4 \cdot 10^0) =$$
$$= (2 \cdot 1) \cdot 10^2 + (3 \cdot 1 + 2 \cdot 4) \cdot 10^1 + (3 \cdot 4) \cdot 10^0.$$

Можна обчислити середній член за допомогою тільки одного множення, скориставшись тим, що нам відомі добутки (2·1) і (3·4), які однаково будуть обчислені:

$$3 \cdot 1 + 2 \cdot 4 = (2 + 3) \cdot (1 + 4) - (2 \cdot 1) - (3 \cdot 4).$$

Для будь-якої пари двозначних чисел $a = a_1 a_0$ і $b = b_1 b_0$, їхній добуток можна обчислити за формулою

$$c = a \cdot b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0,$$

де $c_2 = a_1\, b_1$ – добуток перших цифр, $c_0 = a_0\, b_0$ – добуток других цифр, $c_1 = (a_1 + a_0)(\, b_1 + b_0) - (\, c_2 + c_0)$.

При добутку двох n-значних чисел a й b, де n – додатне парне число, необхідно розділити числа навпіл.

Використаємо для першої половини цифр a запис a_1 а для другий – a_0. Для половин числа b будуть використані відповідно позначення b_1 і b_0. При використанні такого запису матимемо:

$$a = a_1 \cdot 10^{n/2} + a_0,$$
$$b = b_1 \cdot 10^{n/2} + b_0.$$

Таким чином, скориставшись описаним раніше методом, для добутку чисел a й b одержимо

$$c = a \cdot b = \left(a_1 \cdot 10^{n/2} + a_0\right) \cdot \left(b_1 \cdot 10^{n/2} + b_0\right) =$$
$$= (a_1 \cdot b_1) \cdot 10^n + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 10^{n/2} + (a_0 \cdot b_0) =$$
$$= c_2 \cdot 10^n + c_1 \cdot 10^{n/2} + c_0,$$

де $c_2 = a_1\, b_1$ – добуток перших цифр, $c_0 = a_0\, b_0$ – добуток других цифр, $c_1 = (a_1 + a_0)(\, b_1 + b_0) - (\, c_2 + c_0)$.

Якщо $mod(n/2)=0$, той же спосіб можна застосувати й для обчислення добутків c_2, c_1 і c_0. Отже, якщо n являє собою ступінь 2, одержуємо рекурсивний алгоритм для обчислення добутку двох n-значних цілих чисел. У чистому виді рекурсія завершується, коли n стає рівним одиниці, але її можна припинити й раніше – коли n стане досить малим для безпосереднього перемножування чисел такого розміру.

Оскільки перемножування n-значних чисел вимагає трьох множень $n/2$-значних чисел, рекурентне співвідношення для кількості множень $M(n)$ має вигляд

$$M(n) = 3M(n/2) \text{ при } n > 1, M(1) = 1,$$
$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1{,}585}.$$

Не слід забувати, що для чисел середньої довжини цей алгоритм, імовірно, буде працювати довше, ніж класичний. Брассард (Brassard) і Брейтли (Bratley) пишуть, що в їхніх експериментах алгоритм декомпозиції перевершував метод множення в стовпчик тільки для чисел довжиною більше 600 цифр. Якщо використовуються мови програмування Java, C++ або Smalltalk, то для цих мов є спеціальні класи для роботи з великими цілими числами.

Алгоритм Штрассена для множення матриць

Основна ідея, що лежить в основі цього алгоритму, полягає у відкритті, що добуток двох матриць A та B розміром 2×2 можна обчислити за допомогою тільки 7, а не 8 множень, які необхідні при використанні алгоритму грубої сили.

Цього можна досягти, використовуючи наступні формули:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} =$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

де

$m_1 = (a_{00} + a_{11}) \cdot (b_{00} + b_{11})$,
$m_2 = (a_{10} + a_{11}) \cdot b_{00}$,
$m_3 = a_{00} \cdot (b_{01} - b_{11})$,
$m_4 = a_{11} \cdot (b_{10} - b_{00})$,
$m_5 = (a_{00} + a_{01}) \cdot b_{11}$,
$m_6 = (a_{10} - a_{00}) \cdot (b_{00} + b_{01})$,
$m_7 = (a_{01} - a_{11}) \cdot (b_{10} + b_{11})$.

Таким чином, для множення двох матриць розміром 2×2 алгоритм Штрассена виконує сім множень і 18 додавань/віднімань, у той час як алгоритм на основі грубої сили вимагає восьми множень і чотирьох додавань.

Нехай *A* та *B* – дві матриці розміром $n \times n$, де *n* – ступінь двійки (якщо *n* не є ступенем двійки, можна додати до матриць необхідну кількість рядків і стовпців, заповнених нулями). Матриці *A* та *B* і їхній добуток *C* можна розділити на чотири підматриці розміром $\frac{n}{2} \times \frac{n}{2}$ кожну в такий спосіб:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}.$$

Для одержання коректного добутку ці підматриці можна розглядати як числа з формул Штрассена. Алгоритм Штрассена полягає в тому, що необхідні сім добутків підматриць розміром $\frac{n}{2} \times \frac{n}{2}$ обчислюються рекурсивно з використанням описаного методу.

Якщо $M(n)$ – кількість множень, виконуваних алгоритмом Штрассена для множення двох матриць розміром $n \times n$ (де *n* – ступінь двійки), то матимемо наступне рекурентне співвідношення для $M(n)$:

$$M(n) = 7M(n/2) \text{ при } n > 1, M(1) = 1,$$
$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2{,}807}.$$

Це менше, ніж n^3, притаманне для алгоритму на основі грубої сили.

Оскільки економія кількості множень досягається ціною більшої кількості додавань, варто розглянути кількість додавань $A(n)$, виконуваних алгоритмом Штрассена. Для множення двох матриць порядку $n > 1$ алгоритму потрібно сім множень і 18 додавань матриць розміром $\frac{n}{2} \times \frac{n}{2}$; при $n = 1$ додавань взагалі не

потрібно, оскільки в цьому випадку задача вироджується до перемножування двох чисел. Все це приводить до наступного рекурентного рівняння:

$$A(n) = 7A(n/2) + 18(n/2)^2 \text{ при } n > 1, A(1) = 0.$$

Відповідно до основної теореми, $A(n) \in \Theta\left(n^{\log_2 7}\right)$.

Тестові завдання

1. Яка максимальна кількість порівнянь, які виконає алгоритм бінарного пошуку, при успішному пошуку у списку 2, 3, 7, 5, 6, 9, 1, 20?

а) 1;
б) 2;
в) 3;
г) 4.

2. Яка з наведених функцій має найбільший асимптотичний порядок росту?

а) (n-2)!;
б) 5ln(n+100);
в) $22n^{100}$;
г) $1.5n^2$.

3. До якого класу ефективності відноситься класичний алгоритм множення матриць?

а) кубічний;
б) квадратичний;
в) логарифмічний;
г) лінійний.

4. Який можна досягти клас ефективності при застосуванні бінарного пошуку у випадковому списку з *n* елементів?

а) *nlog(n)*;
б) *log(n)*;
в) n^2;
г) *n*.

5. Який з наведених алгоритмів сортування демонструє кращі результати обчислюваних експериментів у найгіршому випадку (при відповіді слід використовувати https://www.toptal.com/developers/sorting-algorithms)?

а) сортування вибором;
б) сортування злиттям;
в) швидке сортування;
г) пірамідальне сортування.

Контрольні запитання

1. Назвіть основні етапи методу декомпозиції.

2. З яким типом рекурентних рівнянь зазвичай пов’язані алгоритми розроблені методом декомпозиції?

3. В чому особливості математичного аналізу алгоритмів метода декомпозиції.

4. Назвіть задачі, для яких ефективним є метод грубої сили порівняно з методом декомпозиції.

5. Назвіть задачі, для яких неможливо сформулювати алгоритм згідно з підходом декомпозиції.

Тема 6 Метод перетворення

Мета: ознайомитись з прикладами застосування методу перетворення.

План

1. Попереднє сортування
2. Збалансовані дерева пошуку
3. Піраміди й пірамідальне сортування
4. Сортування вставкою

6.1 Попереднє сортування

Деякі автори називають цю загальну технологію «перетворюй та володарюй», оскільки такі методи працюють у дві стадії. Спочатку, на стадії перетворення, екземпляр задачі перетвориться в інший, який по тій або іншій причині легше піддається розв'язанню, після чого на стадії «володарювання» розв'язується отриманий у результаті перетворення екземпляр задачі.

Є три основних варіанти цього методу, що відрізняються способом перетворення (рис. 6.1):

– перетворення в більш простий або більш зручний для розв'язання екземпляр тієї ж задачі – спрощення екземпляра;

– зміна подання наявного екземпляра задачі;

– приведення задачі, тобто перетворення до екземпляра іншої задачі, для якої є алгоритм розв'язання.

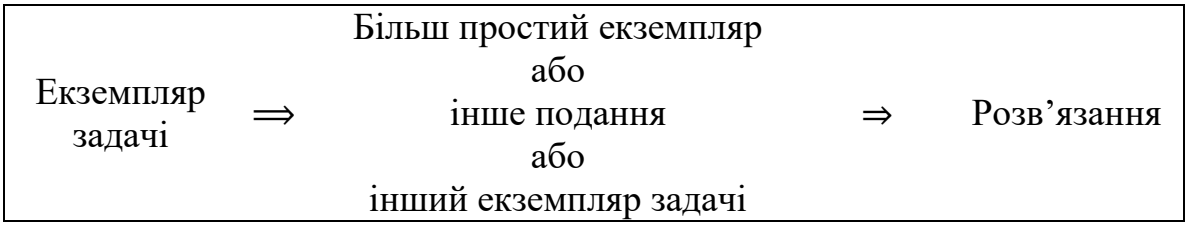


Рисунок 6.1 – Стратегія «перетворюй та володарюй»

Попереднє сортування – стара ідея в кібернетиці. Насправді інтерес до алгоритмів сортування в значній мірі обумовлений саме тим, що ряд задач за участю списків розв'язується істотно простіше, якщо списки відсортовані. Зрозуміло, що часова ефективність алгоритму, що включає як етап сортування, може залежати від ефективності використаного алгоритму сортування. Для простоти вважатимемо, що всі списки реалізовані у вигляді масивів, оскільки багато алгоритмів сортування реалізуються простіше при використанні цього подання.

Алгоритми сортування – сортування вибором, бульбашкове сортування й сортування вставкою – є квадратичними як у найгіршому, так і в середньому

випадку, і два більше ефективних алгоритми – сортування злиттям, ефективність якого в будь-якому випадку дорівнює $\Theta(n \log n)$, і швидке сортування, ефективність якого в середньому випадку також дорівнює $\Theta(n \log n)$, але в гіршому – квадратична. У загальному випадку жоден алгоритм сортування, заснований на порівнянні, не може мати ефективність, що перевищує $n \log n$ у найгіршому або в середньому випадку.

Нижче наведені три приклади використання попереднього сортування.

Приклад 1 (Перевірка одиничності елементів масиву).

Алгоритм на основі грубої сили для перевірки того, що всі елементи масиву різні, попарно порівнює всі елементи цього масиву, поки не будуть знайдені два однакових або поки не будуть переглянуті всі можливі пари. У найгіршому випадку ефективність такого алгоритму дорівнює $\Theta(n^2)$.

До розв'язання задачі можна підійти й по-іншому – спочатку відсортувати масив, а потім порівнювати тільки послідовні елементи: якщо в масиві є однакові елементи, то вони повинні бути розташовані у відсортованому масиві один за одним.

Алгоритм *PresortElementUniqueness* (*A*[0..*n*-1])
// Перевірка одиничності елементів масиву
// Вхідові дані: масив *A*[0..*n*-1] упорядкованих елементів
// Виходові дані: **«true»**, якщо в масиву *A* немає однакових елементів
// та **«false»**, якщо є
Сортування масиву *A*
for *i* ← 0 **to** *n*-2 **do**
  if *A*[*i*] = *A*[*i*+1]
    return false
return true

Час роботи даного алгоритму являє собою суму часу, витраченого на сортування, і часу на перевірку сусідніх елементів. Оскільки для сортування потрібно як мінімум $n \log n$ порівнянь, а для перевірки сусідніх елементів – не більше *n*-1, саме сортування й визначає загальну ефективність алгоритму. Якщо використати квадратичний алгоритм сортування, то алгоритм у цілому виявиться не ефективнішим алгоритму на основі методу грубої сили. Але якщо скористатися гарним алгоритмом сортування, таким як сортування злиттям, ефективність якого в найгіршому разі становлять $\Theta(n \log n)$, то весь алгоритм перевірки одиничності елементів масиву також буде мати ефективність $\Theta(n \log n)$:

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n).$$

Приклад 2 (Обчислення моди). Модою (mode) називається значення, що зустрічається в даному списку частіше інших. Наприклад, у випадку значень 5, 1, 5, 7, 6, 5, 7 модою є значення 5 (якщо однаково часто зустрічається кілька

значень, модою може бути обране кожне з них). Алгоритм на основі грубої сили сканує весь список і обчислює кількість появ у списку кожного з різних значень, після чого розшукується найбільше зі знайдених кількостей появ у списку. При реалізації такого підходу значення, що зустрічаються, й кількість їхніх появ можна зберігати в окремому списку. При кожній ітерації *i*-ий елемент вихідного списку порівнюється зі значеннями елементів, що вже зустрічалися, шляхом сканування допоміжного списку. Якщо значення *i*-го елемента є в допоміжному списку, збільшується лічильник кількості елементів; якщо – ні, елемент додається в допоміжний список, а його лічильнику привласнюється значення 1. Неважко побачити, що в найгіршому випадку Вхідові дані являють собою список з неповторюваних елементів. У такому списку його *i*-й елемент рівняється з (i - 1)-им різним елементом допоміжного списку, перед тим як бути доданим до цього списку. У результаті кількість порівнянь, виконуваних алгоритмом у найгіршому випадку, при створенні допоміжного списку становить

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Крім того, для виявлення елемента з найбільшим значенням лічильника в допоміжному списку потрібно виконати n - 1 порівняння, але вони не впливають на квадратичний характер розглянутого алгоритму в найгіршому випадку.

Розглянемо альтернативний варіант, який починається із сортування списку. У такому випадку всі рівні значення будуть знаходитися один поряд з одним, і для обчислення моди треба тільки знайти найбільшу підпослідовність однакових сусідніх значень у відсортованому списку.

Алгоритм *PresortMode* (*A*[0..*n*-1])
// Обчислює моду масиву з використанням попереднього сортування
// Вхідові дані: масив *A*[0..*n*-1] упорядкованих елементів
// Виходові дані: мода масиву
Сортування масиву *A*
i ← 0
// Поточне сканування починається з позиції *i*
modefrequency ← 0
// Максимальна кількість однакових елементів
while *i* ≤ *n*-1 **do**
  runlenght ← 1; *runvalue* ← *A*[*i*]
  while *i*+*runlenght* ≤ *n*-1 **and** *A*[*i*+*runlenght*] = *runvalue* **do**
    runlenght = *runlenght*+1
  if *runlenght* > *modefrequency*
    modefrequency ← *runlenght*; *modevalue* ← *runvalue*

i ← *i*+*runlenght*
return *modevalue*

Аналіз цього алгоритму аналогічний аналізу алгоритму з прикладу 1: час роботи алгоритму визначається часом сортування, оскільки час виконання іншої частини алгоритму – лінійний. Отже, при використанні сортування, що належить класу ефективності $n \log n$, ефективність описаного алгоритму в найгіршому випадку буде вище ефективності в найгіршому випадку алгоритму з використанням грубої сили.

Задача пошуку перетину двох множин. Нехай $A = \{a_1, a_2, \dots, a_n\}$ і $B = \{b_1, b_2, \dots, b_m\}$ – дві множини чисел. Розглянемо задачу пошуку такої множини $C = \{c_1, c_2, \dots, c_k\}$ – їхнього перетину, яка містить всі числа, які входять як в A, так і у B.

Практичні завдання

1. Розробіть алгоритм *Intersect_bruteforce* методом грубої сили для вирішення даної задачі й визначите клас його ефективності.
2. Розробіть алгоритм *Intersect_conversion* з використанням попереднього сортування для рішення даної задачі й визначите клас його ефективності.

Для виконання поставленої мети, необхідно вирішити наступні задачі:

1. Реалізувати алгоритм *Intersect_bruteforce*.
2. Реалізувати алгоритм *Intersect_conversion*.
3. Провести математичний аналіз алгоритмів.
4. Провести порівняльний емпіричний аналіз алгоритму *Intersect_bruteforce* і алгоритму *Intersect_conversion*.
5. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об’єму та елементів множин A і B (передбачити можливість автоматичної генерації множин випадковим чином та ручного введення з клавіатури);
   - виводити множину C;
   - виводити службову інформацію про параметри роботи алгоритмів (час роботи, кількість ітерацій тощо).
6. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу обох алгоритмів;
   - висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

6.2 Збалансовані дерева пошуку

Бінарне дерево пошуку – це бінарне дерево, вузли якого містять елементи множини елементів, що впорядковуються, по одному елементі у вузлу, причому всі елементи у лівому піддереві менше елемента в корені піддерева, а елементи в правому піддереві – більше його. Відзначимо, що таке перетворення множини в бінарне дерево пошуку являє собою приклад *методу зміни подання*. В цьому випадку одержуємо більш *високу ефективність пошуку, вставки й видалення* – час виконання всіх цих операцій дорівнює $\Theta(\log n)$, але тільки в середньому випадку. У найгіршому випадку ці операції виконуються за час $\Theta(n)$, оскільки дерево може виродитися в повністю незбалансоване, з висотою, рівної n - 1.

Розглянемо структуру даних, що зберігає важливі властивості класичних бінарних дерев пошуку, – у першу чергу логарифмічну ефективність словникових операцій і відсортованість елементів, – але при цьому уникає виродженості в найгіршому випадку. Для цього використаються два підходи.

Перший підхід являє собою варіант спрощення екземпляра задачі – незбалансоване бінарне дерево пошуку перетвориться в збалансоване. Конкретні реалізації цієї ідеї розрізняються за їх визначеннями того, що таке збалансованість. *AVL-дерево* (AVL tree) вимагає, щоб різниця висот лівого й правого піддерев кожного вузла не перевищувала 1. *Червоно-чорне дерево* (red-black tree) допускає, щоб висота одного піддерева була у два рази більше висоти іншого піддерева того ж самого вузла. Якщо вставка нового вузла або видалення наявного приводить до того, що порушується умова збалансованості, таке дерево перебудовується за допомогою одного із сімейств спеціальних перетворень, які називаються поворотами (rotation) і які відновлюють умови збалансованості.

Другий підхід являє собою варіант зміни подання: допускається наявність більш ніж одного елемента у вузлу дерева пошуку. Окремими випадками таких дерев є *2-3-дерева*, *2-3-4-дерева* й більш загальний і важливий випадок – *B-дерева*. Вони розрізняються кількістю елементів, які допустимі в одному вузлу дерева пошуку, але всі вони є ідеально збалансованими.

AVL-дерева

AVL-дерева були відкриті в 1962 р. двома радянськими математиками – Г.М. Адельсон-Вельским і Е.М. Ландисом; винайдена структура одержала назву по перших буквах їхніх прізвищ.

Визначення 1. AVL-дерево являє собою бінарне дерево пошуку, у якому показник збалансованості (balance factor) кожного вузла, обумовлений як різниця висот лівого й правого піддерев вузла, дорівнює 0, +1 або -1 (висота порожнього дерева вважається рівною -1).

Так, бінарне дерево пошуку на рис. 6.2, а – AVL-дерево, у той час як бінарне дерево пошуку на рис. 6.2, б таким не є.

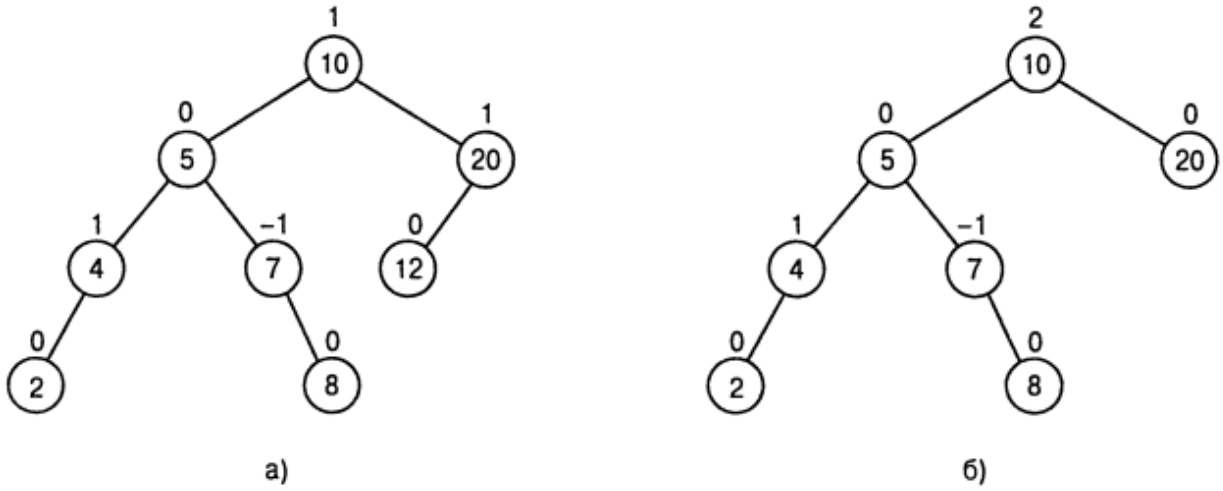
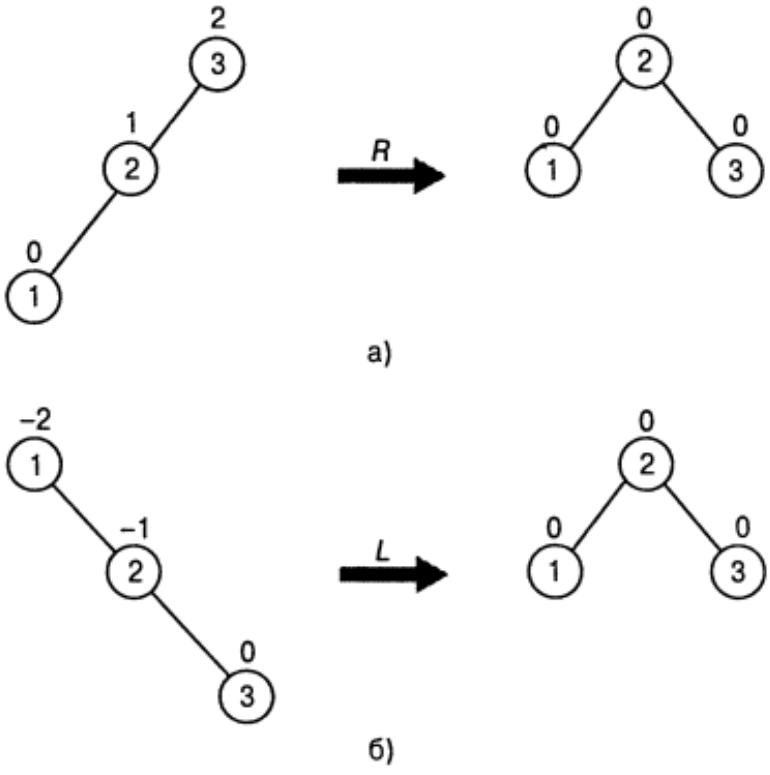


Рисунок 6.2 – Бінарні дерева: а) AVL-дерево; б) бінарне дерево пошуку, що не є AVL-деревом

Показники збалансованості показані над вузлами дерев.

Якщо вставка нового вузла робить AVL-дерево незбалансованим, воно перетворюється за допомогою *повороту*. Поворот в AVL-дереві являє собою локальне перетворення піддерева, корінь якого має показник збалансованості, рівний +2 або -2; якщо таких вузлів небагато, повертаємо дерево з незбалансованим коренем, який найбільш близький до знову вставленого листа. Усього є тільки чотири типи поворотів, причому два з них являють собою дзеркальне відображення двох інших. У найпростішій формі чотири можливих повороти показані на рис. 6.3.



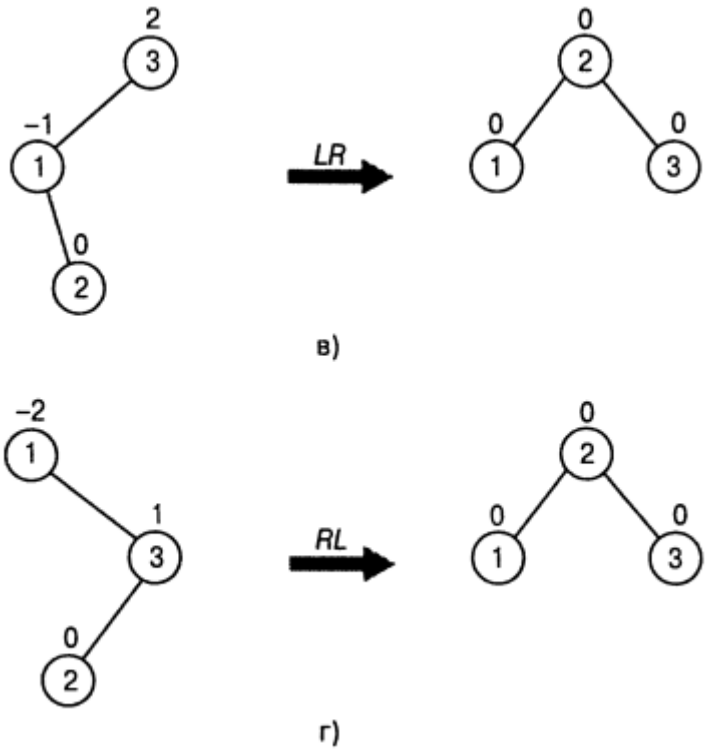
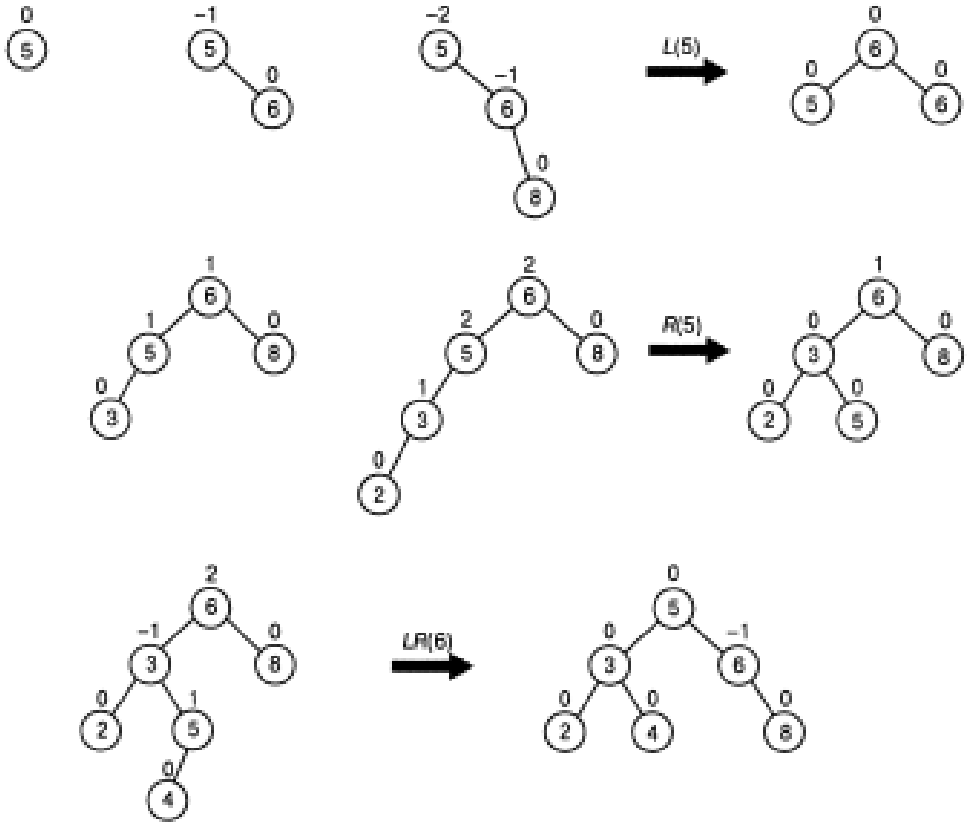


Рисунок 6.3 – Чотири типи поворотів AVL-дерев з трьома вузлами:
а) одиночний *R*-поворот; б) одиночний *L*-поворот;
в) подвійний *LR*-поворот; г) подвійний *RL*-поворот

На рисунку 6.4 представлено побудову AVL-дерева шляхом послідовної вставки чисел із списку 5, 6, 8, 3, 2, 4, 7. Число в дужках після вказівки типу повороту – корінь дерева, що перебудовується.



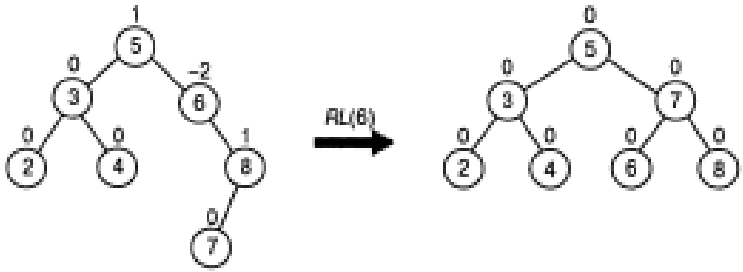
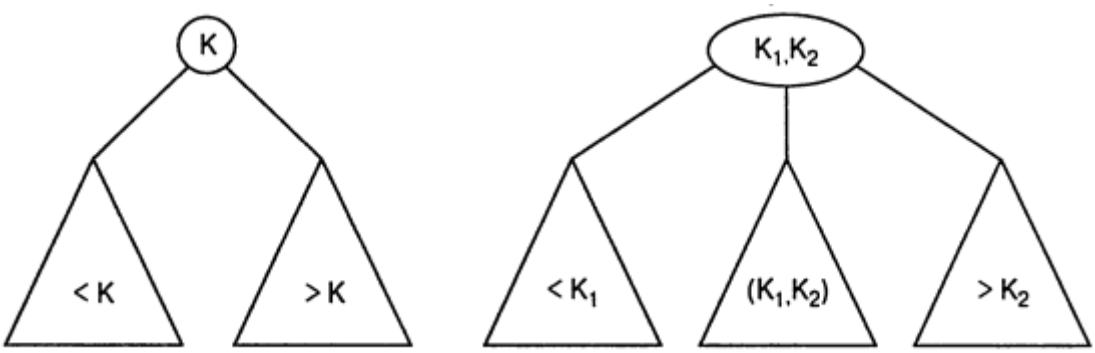


Рисунок 6.4 – Побудова AVL-дерева шляхом послідовної вставки чисел із списку 5, 6, 8, 3, 2, 4, 7

2-3-дерева

Друга ідея балансування дерев пошуку полягає в тому, щоб дозволити вузлу одночасно містити кілька ключів. Найпростішою реалізацією цієї ідеї є 2-3-дерева, розроблені в 1970 р. американським кібернетиком Дж. Хопкрофтом (John Hopcroft). *2-3-дерево* являє собою дерево, що може мати вузли двох видів – 2-вузли й 3-вузли. *2-вузол* містить єдиний ключ K та має два нащадки: лівий дочірній вузол служить коренем піддерева, всі ключі в якому менше за K, а правий – коренем піддерева, всі ключі в якому більші за K. (Інакше кажучи, 2-вузол точно такий же, як і вузол у класичному бінарному дереві пошуку.) *3-вузол* містить два впорядкованих ключі K_1 і K_2 ($K_1 < K_2$) і має три дочірніх вузли. Лівий дочірній вузол служить коренем піддерева, ключі в якому менші за K_1, середній – коренем піддерева, ключі в якому більші за K_1 і менші за K_2, а правий – коренем піддерева, всі ключі в якому більші за K_2 (рис. 6.5).



а) б)

Рисунок 6.5 – Два типи вузлів 2-3-дереві: а) 2-вузол; б) 3-вузол

Остання вимога до 2-3-дерева полягає в тому, що всі його листи повинні перебувати на одному рівні, тобто 2-3-дерево *завжди збалансовано за висотою* (height-balanced): довжина шляху від кореня дерева до листа повинна бути однакова для всіх листів дерева. Ця властивість досягається ціною дозволу мати вузли із трьома дочірніми вузлами.

Пошук заданого ключа K у 2-3-дереві досить простий. Він починається з кореня. Якщо корінь являє собою 2-вузол, то діємо так само, як і у випадку бінарного дерева пошуку, – або припиняємо пошук, якщо значення K дорівнює

значенню ключа кореня, або продовжуємо пошук у левом або правом піддереві, залежно від того, чи менше значення *K*, ніж ключ кореня, чи більше. Якщо ж корінь являє собою 3-вузол, то після не більше ніж двох порівнянь визначаємо, чи варто припинити пошук (якщо *K* дорівнює одному із ключів 3-вузла) або в якому з трьох піддерев він повинен бути продовжений.

Вставка нового ключа в 2-3-дерево виконується в такий спосіб. Новий ключ *K* завжди вставляється у лист, за винятком випадку порожнього дерева. Відповідний лист знаходиться при виконанні пошуку ключа *K*. Якщо шуканий лист – 2-вузол, *K* вставляється або як перший, або як другий ключ – залежно від того, чи менше *K*, ніж старий ключ, чи більше. Якщо ж шуканий лист – 3-вузол, то розділяємо його на два: найменший із трьох ключів (двох старих і нового) міститься в перший лист, найбільший – у другий лист, а середній ключ переноситься у вузол, батьківський по відношенню до старого листа (якщо лист – корінь дерева, то для вставки нового ключа створюється новий корінь). Помітимо, що переміщення середнього ключа в батьківський вузол може привести до переповнення батьківського вузла (якщо він був 3-вузлом) і, отже, викликати ряд поділів вузлів уздовж ланцюжка предків листа.

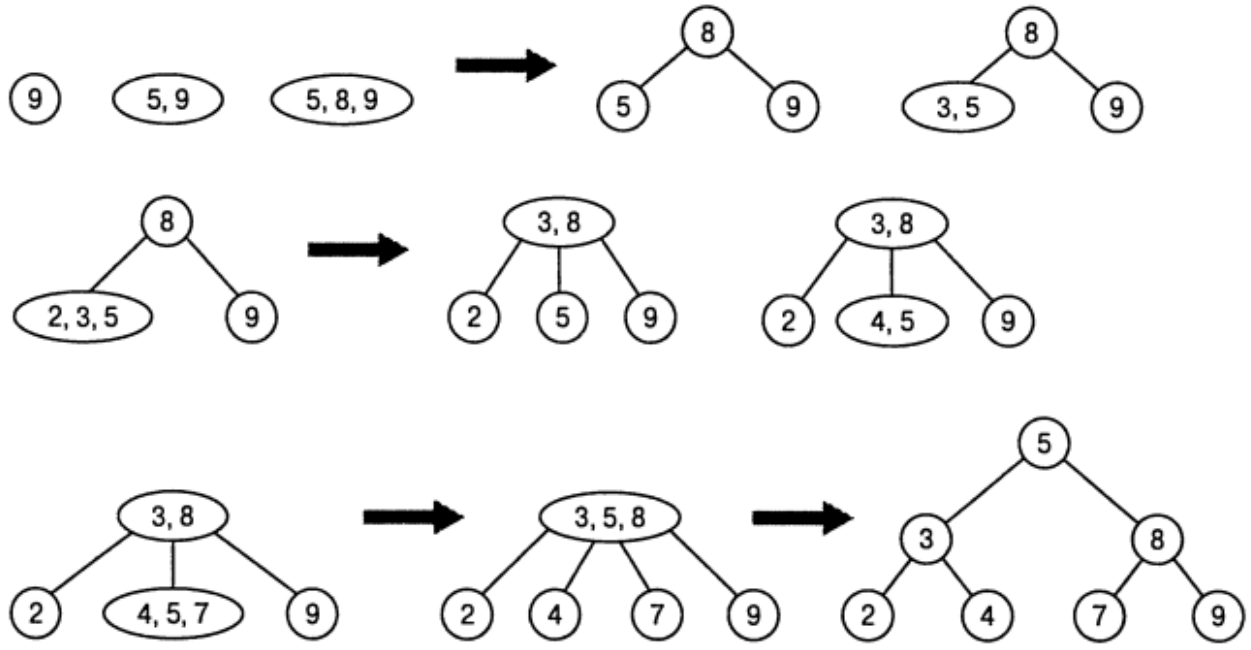


Рисунок 6.6 – Побудова 2-3-дерева шляхом внесення ключів 9, 5, 8, 3, 2, 4, 7

Як і в будь-якому дереві пошуку ефективність словникових операцій залежить від його висоти. Знайдемо верхню границю ефективності. Для будь-якого 2-3-дерева висотою *h* з *n* вузлами одержуємо нерівність

$$h \leq \log_2(n+1) - 1.$$

Для будь-якого 2-3-дерева з *n* вузлами

$$h \geq \log_3(n+1) - 1.$$

З отриманих у такий спосіб верхньої й нижньої границь висоти h

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

випливає, що часова ефективність пошуку, вставки й видалення як у найгіршому, так і в середнє випадку – $\Theta(\log n)$.

6.3 Піраміди й пірамідальне сортування

Структура даних – піраміда (heap) – являє собою частково впорядковану структуру даних, що особливо добре підходить для реалізації черг із пріоритетами. Згадаємо, що *черга із пріоритетами* (priority queue) являє собою множину елементів з упорядкованою характеристикою, що називається пріоритетом (priority) елемента, і, що забезпечує виконання наступних операцій:

– друк елемента з найвищим (тобто з найбільшим) пріоритетом;

– видалення елемента з найбільшим пріоритетом;

– додавання нового елемента в множину.

Піраміди становлять особливий інтерес у першу чергу завдяки ефективній реалізації перерахованих операцій. Піраміда також є структурою даних, що служить наріжним каменем теоретично важливого алгоритму – пірамідального сортування (heapsort).

Поняття піраміди

Визначення 1. *Піраміда* (heap) може бути визначена як бінарне дерево з ключами, призначеними її вузлам (по одному ключу на вузол), для якого виконуються наступні дві умови:

1. Вимога до форми дерева. Бінарне дерево практично повне (essentially complete) або просто повне (complete), тобто всі його рівні заповнені, за винятком, можливо, останнього рівня, у якому можуть бути відсутніми деякі крайні справа листи.
2. Вимога домінування батьківських вузлів. Ключ у кожному вузлу не менше ключів у його дочірніх вузлах (вважається умова автоматично виконується для всіх листів). Деякі автори вимагають, щоб ключ у кожному вузлу не перевищував ключів в дочірніх вузлах.

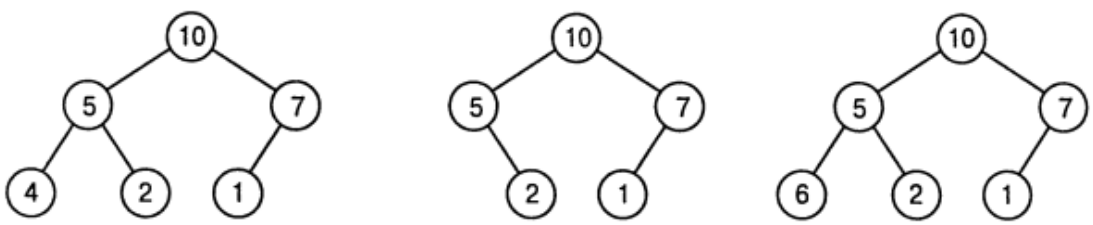
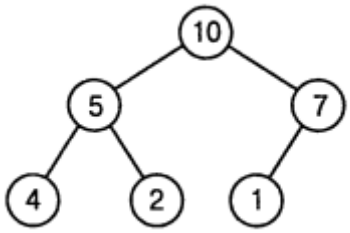


Рисунок 6.7 – Ілюстрації до визначення піраміди

Розглянемо дерева на рис. 6.7. Лише перше дерево є пірамідою. Друге дерево – не є пірамідою, оскільки порушено вимогу до форми дерева. Третє дерево також не є пірамідою, оскільки в ньому порушена вимога домінування батьківських вузлів для вузла із ключем 5.

Звертає на себе увагу впорядкованість значень у піраміді зверху вниз – тобто послідовність значень на будь-якому шляху від кореня до листа спадна (незростаюча, якщо допускається наявність однакових ключів). Однак упорядкованості ключів зліва направо немає, тобто немає ніяких співвідношень між значеннями ключів у вузлах на одному рівні дерева або, у загальному випадку, у лівому і правому піддеревах одного вузла.

Перелік важливих властивостей пірамід (для більш точного розуміння, див. рис. 6.8):



Подання у виді масиву

Індекс	0	1	2	3	4	5	6
Значення		10	5	7	4	2	1

Батьківські вузли *Листя*

Рисунок 6.8 – Піраміда та її подання у виді масиву

1. Є рівно одне практично повне бінарне дерево з n вузлами. Його висота дорівнює $\lfloor \log_2 n \rfloor$.

2. Корінь піраміди завжди містить її найбільший елемент.

3. Будь-який вузол піраміди з усіма його нащадками також є пірамідою.

4. Піраміда може бути реалізована у вигляді масиву шляхом запису її елементів зверху вниз зліва направо. Зручно зберігати елементи піраміди в позиціях такого масиву з 1 по n, залишаючи $H[0]$ або невикористовуваним, або розміщаючи в ньому обмежник, значення якого перевищує значення будь-якого елемента піраміди. При використанні такого подання:

а) ключі батьківських вузлів займають перші $\lfloor n/2 \rfloor$ позицій у масиві, а ключі листів – останні $\lceil n/2 \rceil$ позицій.

б) дочірні ключі стосовно батьківського в позиції i $(0 \le i \le \lfloor n/2 \rfloor)$ перебувають у позиціях $2i$ і $2i+1$, відповідно; батьківський ключ для ключа в позиції i $(2 \le i \le n)$ перебуває в позиції $\lfloor i/2 \rfloor$.

Таким чином, можна визначити піраміду як масив $H[1..n]$, у якому кожний елемент у позиції i у першій половині масиву більше або дорівнює елементам у позиціях $2i$ і $2i+1$, тобто

$$H[i] \geq \max\{H[2i], H[2i+1]\} \text{ для } i = 1, 2, \ldots, \lfloor n/2 \rfloor.$$

(Зазвичай, якщо $2i+1>n$, то виконуватися повинне тільки нерівність $H[i] \geq H[2i]$) У той час як ідеї, що лежать в основі алгоритмів з використанням пірамід, простіше для розуміння при поданні пірамід у вигляді бінарних дерев, реальні реалізації цих алгоритмів звичайно істотно простіше й ефективніше при використанні подання у вигляді масиву.

Є два основних методи побудови піраміди для заданої множини ключів. Перший називається висхідною побудовою піраміди (bottom-up heap construction) і проілюстрований на рис. 6.9.

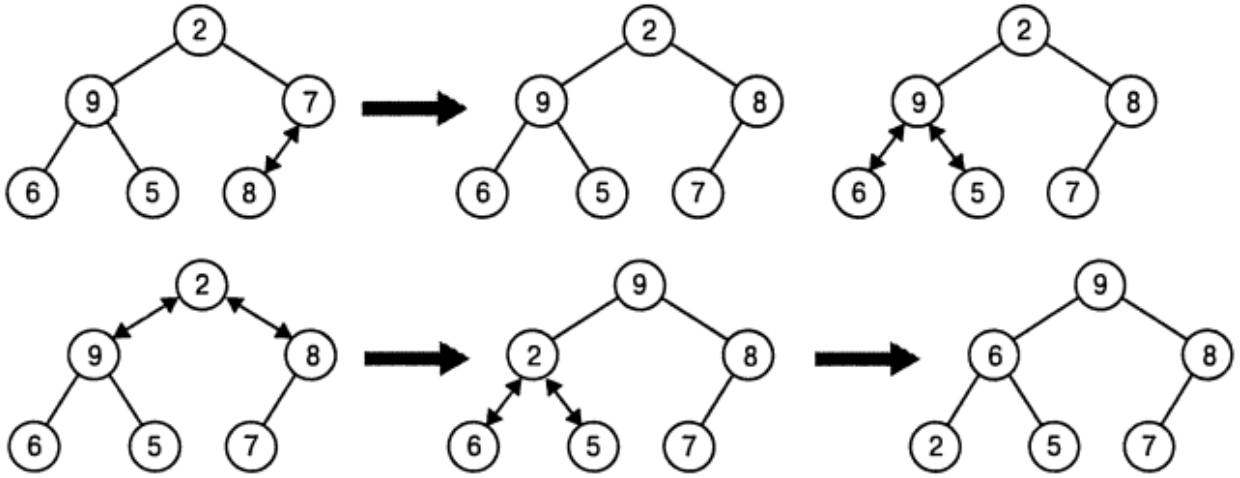


Рисунок 6.9 – Висхідна побудова піраміди для множини ключів 2, 9, 7, 6, 5, 8

При цьому практично повне бінарне дерево ініціалізується шляхом розміщення n ключів у заданому порядку, а потім дерево «пірамідизується» у такий спосіб. Починаючи з останнього батьківського вузла й закінчуючи коренем алгоритм перевіряє, чи виконується для розглянутого вузла вимога домінування батьківського вузла. Якщо ні, то алгоритм обмінює ключ вузла K з найбільшим ключем серед його дочірніх вузлів і перевіряє виконання вимоги домінування батьківського вузла для ключа K у нової позиції. Цей процес триває доти, поки для ключа K не буде виконана вимога домінування батьківського вузла (в остаточному підсумку ця вимога буде виконана, тому що вона завжди виконується для ключів у листах). Після завершення «пірамідизації» піддерева, коренем якого є даний батьківський вузол, алгоритм виконує такі ж дії з безпосереднім предком цього вузла. Алгоритм завершує свою роботу після обробки кореня дерева.

Тут варто зробити одне зауваження, оскільки значення ключа не змінюється при його переміщенні вниз по дереву, немає необхідності виконувати проміжні обміни. Таке покращення алгоритму можна представити як обмін порожнього вузла з більшими ключами серед нащадків (або як переміщення порожнього вузла вниз по дереву) до досягнення кінцевої позиції, куди й

вставляється раніше збережений ключ.

Алгоритм *HeapBottonUp* (*H*[1..*n*])
// Побудова піраміди з елементів заданого масиву
// за допомогою висхідного алгоритму
// Вхідові дані: масив *H*[10..*n*] упорядкованих елементів
// Виходові дані: піраміда *H*[10..*n*]
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
  $k \leftarrow i$; $v \leftarrow H[k]$
  heap = **false**
  while not *heap* **and** $2*k \leq n$ **do**
    $j \leftarrow 2*k$
    if $j < n$
    // Маємо два дочірні вузли
      if $H[j] < H[j+1]$
      $j \leftarrow j+1$
  if $v \geq H[j]$
    heap ← **true**
  else
    $H[k] \leftarrow H[j]$; $k \leftarrow j$
$H[k] \leftarrow v$

Ефективність даного алгоритму у найгіршому випадку:

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\substack{\text{ключі на} \\ \text{рівні } i}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)\,2^i = 2(n - \log_2(n+1)).$$

Альтернативний (і менш ефективний) алгоритм будує піраміду шляхом послідовних вставок нового ключа в раніше побудовану; деякі автори називають цей алгоритм *спадною побудовою піраміди* (top-down heap construction). Розглянемо процедуру додавання нового ключа в піраміду. Почнемо з додавання нового вузла із ключем *K* після останнього листа наявної піраміди, а потім перемістимо *K* у відповідне його значенню місце в новій піраміді в такий спосіб. Порівняємо *K* с батьківським ключем: якщо він не менше *K*, алгоритм припиняє роботу (отримана структура є пірамідою). У протилежному випадку обміняємо ці два ключі й будемо порівнювати *K* с новим батьківським вузлом. Цей процес триває доти, поки *K* не перестане перевищувати значення ключа в батьківському вузлу або не досягне кореня (цей процес проілюстрований на рис.). У цьому алгоритмі також можна переміщувати порожній вузол до досягнення їм коректної позиції, а потім привласнити йому значення *K*.

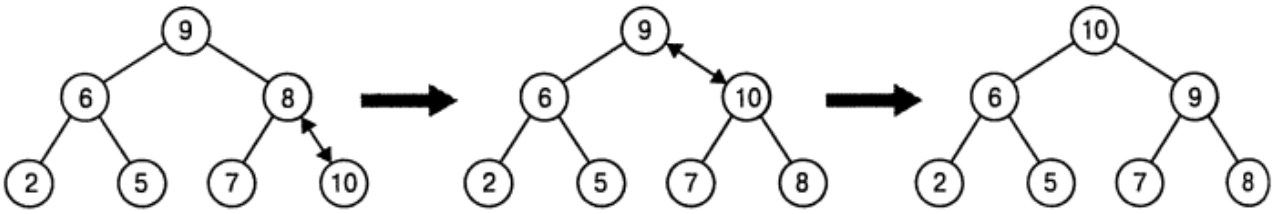


Рисунок 6.10 – Вставка ключа 10 у піраміду, побудовану на рис 6.9

Вочевидь, така вставка не може вимагати більшої кількості порівнянь ключів, чим висота піраміди. Оскільки висота піраміди з n вузлами дорівнює близько $\log_2 n$, часова ефективність вставки становить $O(\log n)$.

Розглянемо тільки найбільш важливий випадок видалення ключа з кореня. Отже, видалення кореневого ключа з піраміди можна виконати за допомогою наступного покрокового алгоритму (проілюстрованого на рис.).

Крок 1. Обміняти ключ у корені з останнім ключем піраміди.

Крок 2. Зменшити розмір піраміди на 1.

Крок 3. «Пірамідизувати» зменшене дерево шляхом переміщення *K* вниз по дереву так само, як у висхідному алгоритмі побудови піраміди, – тобто перевіряючи виконання вимоги домінування батьківських вузлів: якщо воно виконується, алгоритм завершує роботу, якщо ні – обмінюємо *K* із найбільшим з дочірніх вузлів і повторюємо дану операцію доти, поки в черговій позиції *K* вимоги домінування батьківських вузлів не виявиться виконаним.

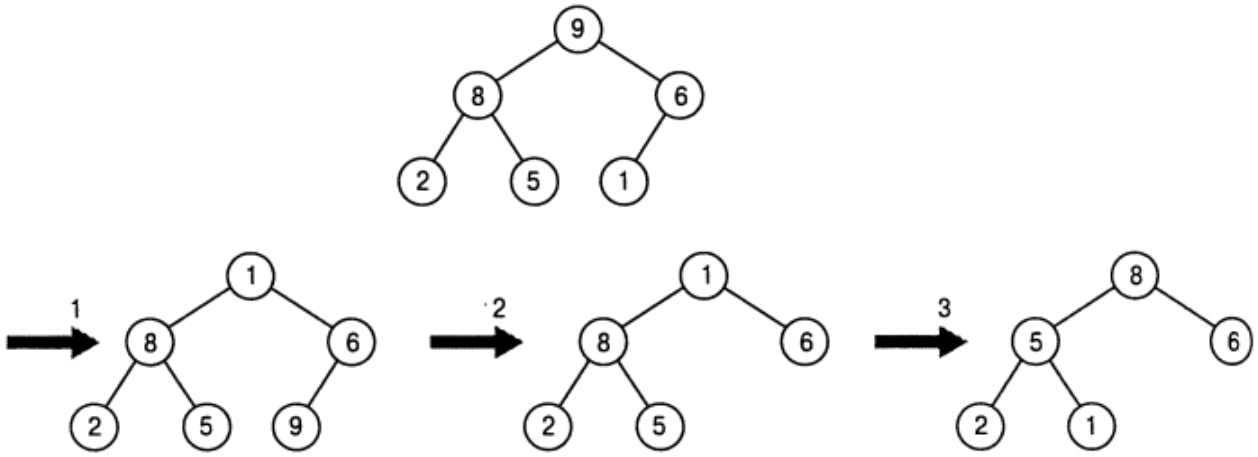


Рисунок 6.11 – Видалення кореневого ключа з піраміди

Ефективність операції видалення визначається кількістю виконуваних порівнянь ключів, необхідних для «пірамідизації» дерева після того, як був зроблений обмін і розмір піраміди був зменшений на 1. Оскільки не може знадобитися порівнянь більше, ніж подвоєна висота піраміди, часова ефективність видалення з піраміди – $O(\log n)$.

Тепер можна описати пірамідальне сортування (*HeapSort*) – цікавий алгоритм сортування, відкритий Дж. Вільямсом. Цей двоетапний алгоритм

працює в такий спосіб.

Етап 1 (побудова піраміди). Будуємо піраміду для заданого масиву.

Етап 2 (видалення найбільших елементів). Застосовуємо операцію видалення кореня n - 1 раз.

У результаті елементи масиву видаляються в порядку зменшення. Але оскільки при реалізації піраміди з використанням масиву видаляється елемент, що, розташовується останнім, масив, що виходить у результаті пірамідального сортування, виявляється відсортований у порядку зростання.

Оцінимо часову ефективність пірамідального сортування:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i \le$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n.$$

Це означає, що для другого етапу пірамідального сортування $C(n) \in O(n \log n)$. Більше докладний аналіз показує, що в дійсності часова ефективність пірамідального сортування дорівнює $\Theta(n \log n)$ як у середньому, так і в найгіршому випадках. Таким чином, часова ефективність пірамідального сортування попадає в той же клас, що й сортування злиттям, але, на відміну від останнього, виконується «на місці», без залучення додаткової пам'яті. Експерименти, проведені над випадковими файлами, показують, що пірамідальне сортування працює повільніше швидкого сортування, однак цілком може суперничати із сортуванням злиттям.

Визначення 2. *Пірамідою* (купою) називається двійкове дерево таке, що

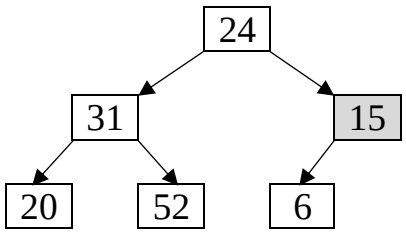
$$a[i] \le a[2i+1],$$
$$a[i] \le a[2i+2],$$

де $a[0]$ – найменший (мінімальний) елемент і розміщується на верхівці піраміди.

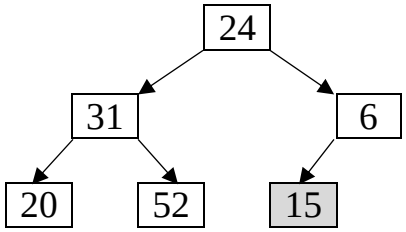
Приклад. Реалізуємо поетапне виконання алгоритму пірамідального сортування для масиву 24, 31, 15, 20, 52, 6.

Перший етап – побудова піраміди. Визначається права частина дерева, починаючи з $N/2 - 1$ (нижній рівень дерева). Береться елемент лівіше за цю частину масиву і його «просіюють» через піраміду по тому шляху, де розташовані елементи, менші за нього, які, в свою чергу, одночасно підіймаються вгору. Із двох можливих маршрутів обирається той, на шляху якого стоїть «менший» елемент.

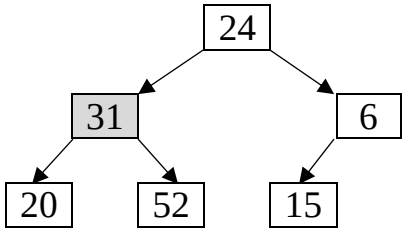
Розмістимо елементи у вигляді вихідної піраміди. Кількість елементів у масиві $N = 6$, тому позиція елементу правої частини $(6/2 - 1) = 2$ – це елемент 15.



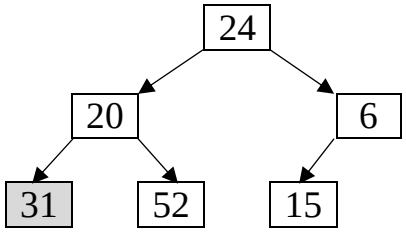
Результат просіювання елемента 15 через піраміду:



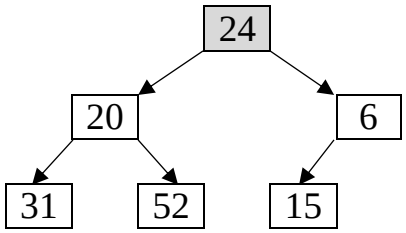
Позиція наступного елементу, який буде просіюватися – 1, це елемент 31.



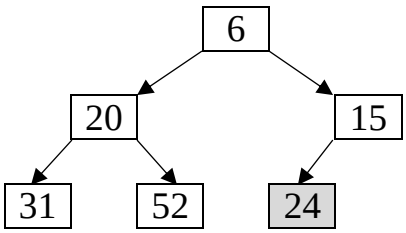
Результат просіювання елемента 31 через піраміду:



Наступним буде елемент на нульовій позиції, це елемент 24.

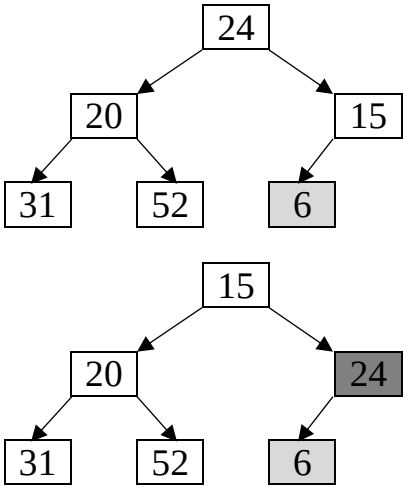


Результат просіювання елемента 24 через піраміду:

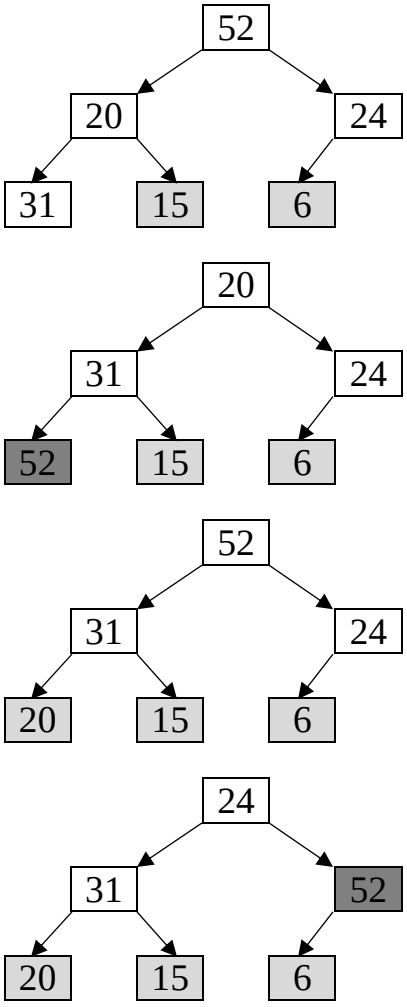


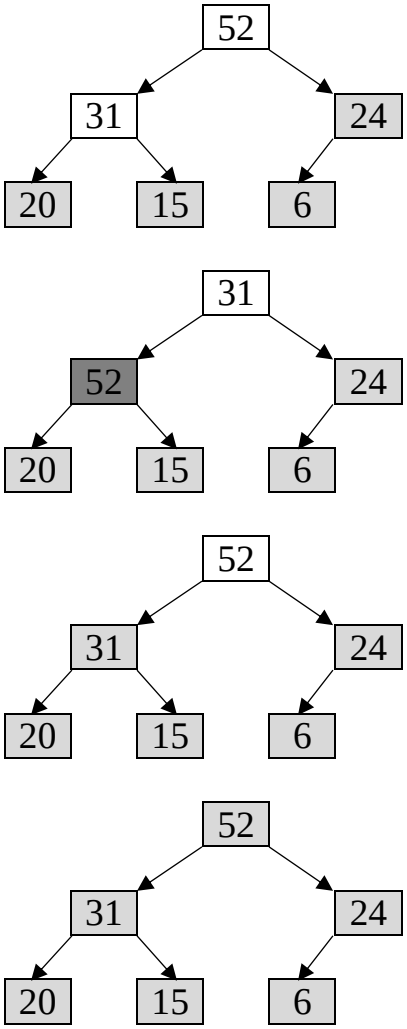
Зрозуміло, що масив ще не до кінця відсортовано. Однак, така процедура просіювання – це основа для пірамідального сортування. В результаті просіювання, найменший елемент розташовується на вершині піраміди.

Другий етап – сортування на побудованій піраміді. Беремо останній елемент масиву в якості поточного. Міняємо верхній (найменший) елемент масиву і поточний місцями. Поточний елемент (він тепер верхній) просіюємо через $(N-1)$-елементну піраміду. Потім береться передостанній елемент і т.д.



Продовжимо процес. В результаті масив буде відсортовано за спаданням.





Переваги пірамідального сортування:

– має доведену оцінку найгіршого випадку;

– сортує «на місці», тобто необхідно всього $O(1)$ додаткової пам'яті (якщо дерево організовувати саме так, як показано вище).

Приклад реалізації (на мові Python):

```
def heapSort(li):
    def downHeap(li, k, n):
        new_elem = li[k]
        while k <= n/2:
            child = 2*k;
            if child < n and li[child] < li[child+1]:
                child += 1
            if new_elem >= li[child]:
                break
            li[k] = li[child]
            k = child
        li[k] = new_elem
    size = len(li)
```

```
for i in range(round(size/2-1),-1,-1):
        downHeap(li, i, size-1)
for i in range(size-1,0,-1):
        temp = li[i]
        li[i] = li[0]
        li[0] = temp
        downHeap(li, 0, i-1)
return li
```

Практичні завдання

1. Реалізувати алгоритм *HeapSort*.
2. Протестувати алгоритм для різного об’єму вхідних даних.
3. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об’єму випадкової послідовності;
   - виводити генеровану випадковим чином послідовність у файл;
   - виводити відсортовану послідовність у файл із тим же ім’ям, але іншим розширенням;
   - виводити службову інформацію про параметри роботи алгоритму (час роботи, кількість ітерацій тощо).
4. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - висновок щодо отриманих результатів в контексті проведеного тестування для різних вхідних даних (виконання завдання передбачає заповнення зведеної таблиці даними, отриманими в результаті проведення розрахунків для різних вхідних даних і побудову графіку);
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

До звіту додається два файли з різними розширеннями (див. п. 3.)

6.4 Сортування вставкою

Метод зменшення розміру задачі заснований на використанні зв’язку між рішенням даного екземпляру задачі й рішенням зменшеного екземпляру тієї ж задачі. Якщо такий зв’язок установлений, його можна використати або зверху вниз (рекурсивно), або знизу вгору (без використання рекурсії).

Є три основних варіанти методу зменшення розміру:

1) зменшення на постійну величину;

2) зменшення на постійний множник;

3) зменшення змінного розміру.

При зменшенні на постійну величину розмір екземпляра задачі знижується на ту саму постійну величину при кожній ітерації алгоритму.

Сортування вставкою. Алгоритм сортування вставкою (*InsertionSort*) – це приклад алгоритму, розробленого з використанням методу зменшення розміру задачі.

Алгоритм сортування вставкою у вигляді псевдокоду наведено нижче.

Алгоритм *InsertionSort* ($A[0..n-1]$)

// Сортує масив $A[0..n-1]$ методом сортування вставками

// Входові дані: масив $A[0..n-1]$ елементів, що потребують сортування

// Виходові дані: відсортований масив $A[0..n-1]$ у неспадному порядку

for *i* = 1 **to** *n*-1 **do**

v = *A*[*i*]

j = *i*-1

while *j*>=0 **and** *A*[*j*]>*v* **do**

A[*j*+1] = *A*[*j*]

j = *j*-1

A[*j*+1] = *v*

Практичні завдання

1. Реалізувати алгоритм *InsertionSort*.
2. Провести математичний аналіз алгоритму.
3. Провести емпіричний аналіз алгоритму.
4. Провести порівняльний аналіз алгоритму сортування вставками й алгоритму сортування злиттям.
5. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення об’єму випадкової послідовності;
   - виводити генеровану випадковим чином послідовність у файл;
   - виводити відсортовану послідовність у файл із тим же ім’ям, але іншим розширенням.
6. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму;
   - висновок щодо отриманих результатів (зокрема, в частині

порівняльного аналізу алгоритмів сортування вставками та злиттям); скріншоти роботи програми для різних вхідних даних;
- відповіді на контрольні запитання.

До звіту додається два файли з різними розширеннями (див. п. 5.)

Тестові завдання

1. Який з підходів дає найбільш ефективний алгоритм розв'язання задачі пошуку перетину двох множин?

а) метод грубої сили;
б) метод декомпозиції;
в) метод перетворення;
г) динамічне програмування.

2. Який клас ефективності пошуку в бінарному дереві в найгіршому випадку?

а) кубічний;
б) квадратичний;
в) логарифмічний;
г) лінійний.

3. Які значення допустимі для показника збалансованості AVL-дерева?

а) 1;
б) -1;
в) 0;
г) 2.

4. Які значення допустимі для показника збалансованості 2-3-дерева?

а) 1;
б) -1;
в) 0;
г) 2.

5. В якій деревовидній структурі даних корінь містить найбільший елемент?

а) 2-3 дерево;
б) AVL-дерево;
в) піраміда;
г) жодне з наведених.

Контрольні запитання

1. У чому полягає особливість методу зменшення розміру задачі?
2. Оцінка часу роботи алгоритму сортування вставками для найгіршого, найкращого й середнього випадків.
3. Недоліки методу сортування вставками.
4. В якому випадку дерево пошуку стає не збалансованим?

5. Чим поняття піраміда відрізняється від поняття список?

Тема 7 Динамічне програмування

Мета: ознайомитись з прикладами застосування методу динамічного програмування до обчислювальних задач та задач на графах.

План

1. Основи динамічного програмування
2. Обчислення біноміальних коефіцієнтів
3. Найдовша спільна послідовність
4. Алгоритм Флойда

7.1 Основи динамічного програмування

Динамічне програмування – це метод розв'язання задач з підзадачами, що перекриваються. Зазвичай такі задачі виникають з рекурентних співвідношень. Коли розв'язок даної задачі пов'язаних з розв'язками підзадач того ж типу, але меншого розміру. При цьому, може виникнути потреба обчислювати розв'язок однієї з підзадач декілька разів. При застосуванні методу динамічного програмування пропонується запам'ятовувати розв'язки задач, що повторюються у таблицю. При необхідності дані зчитуються з таблиці без додаткових втрат часу.

Наприклад, при обчислені *n*-го числа Фібоначчі, дерево рекурсивних викликів має такий вигляд (рис. 7.1). Можна побачити, що зберігання проміжних значень функцій *F*(4), *F*(3), *F*(2), *F*(1) може суттєво прискорити швидкість обчислень *F*(6).

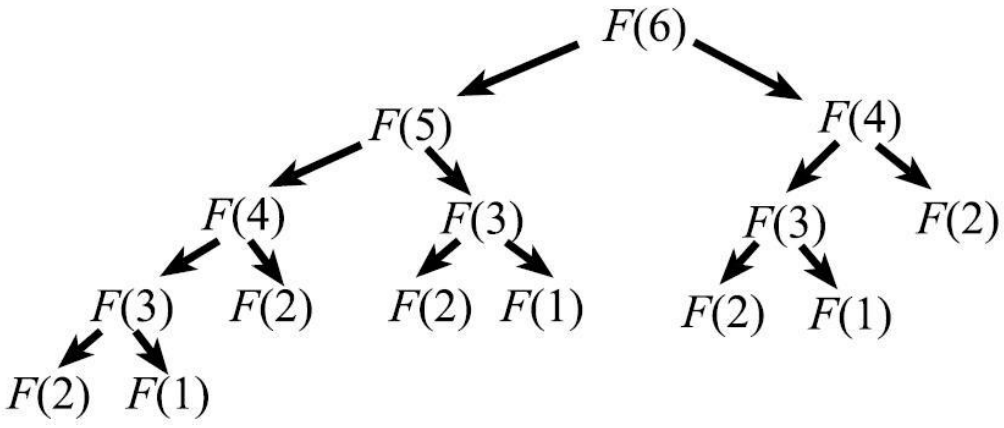


Рисунок 7.1 – Дерево викликів функції обчислення 6-го числа Фібоначчі

Такою таблицею може виступати асоціативний масив з ключами – можливими аргументами функції, а значеннями – відповідними значеннями шуканої функції.

Динамічне програмування можна застосовувати, лише коли маємо

оптимальну підструктуру: оптимальний розв'язок задачі складається з оптимальних розв'язків її підзадачі.

7.2 Обчислення біноміальних коефіцієнтів

При обчисленні біноміальних коефіцієнтів застосування підходу динамічного програмування може суттєво підвищите швидкість розрахунків. Біноміальним коефіцієнтом $C(n, k)$ називається кількість комбінацій (підмножин) з k елементів n-елементної множини ($0 \leq k \leq n$).

Залежність між коефіцієнтами, яка може використовуватись при обчисленнях:

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k,$$
$$C_n^0 = C_n^n = 1.$$

Візуально, біноміальні коефіцієнти та зв'язок між ними можна зобразити трикутником Паскаля (рис. 7.2).

n \ k	0	1	2	3	4	5	6	7	8
0	C_0^0								
1	C_1^0	C_1^1							
2	C_2^0	C_2^1	C_2^2						
3	C_3^0	C_3^1	C_3^2	C_3^3					
4	C_4^0	C_4^1	C_4^2	C_4^3	C_4^4				
5	C_5^0	C_5^1	C_5^2	C_5^3	C_5^4	C_5^5			
6	C_6^0	C_6^1	C_6^2	C_6^3	C_6^4	C_6^5	C_6^6		

n \ k	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

Рисунок 7.2 – Трикутник Паскаля

Алгоритм обчислення коефіцієнтів може бути таким, як представлено нижче.

Алгоритм *Binomial* (*n*,*k*)
// Обчислення *C*(*n*,*k*) за допомогою методу
// динамічного програмування
// Вхідові дані: пара позитивних чисел $0 \leq k \leq n$
// Виходові дані: значення *C*(*n*,*k*)
for *i* ← 0 **to** *n* **do**
  for *j* ← 0 **to** min(i, k) **do**
    if *j = 0* **or** *j = i*
      C[*i*, *j*] ← 1
    else
      C[*i*, *j*] ← C[*i-1*,*j-1*]+ C[*i-1*, *j*]
return C[*n*, *k*]

Часова складність цього алгоритму – $\Theta(nk)$.

7.3 Найдовша спільна підпослідовність

Задачі з обробки послідовності символів можуть виникати у застосунках, які працюють з текстами природніх або синтезованих мов, у біоінформатиці при кодуванні генетичних послідовностей ДНК тощо.

Розглянемо довільну **послідовність** – список елементів, у якому порядок елементів важливий. Будемо вважати, що елементи послідовності – це символи. Підпослідовність Z рядка X – це рядок X з вилученими з нього деякими елементами. Наприклад, якщо X – це рядок ГАЦ, то в нього вісім підпослідовностей:

– ГАЦ (жоден символ не вилучено);
– ГА (Ц вилучено);
– ГЦ (А вилучено);
– АЦ (Г вилучено);
– Г (А, Ц вилучено);
– А (Г, Ц вилучено);
– Ц (Г, А вилучено);
– порожній рядок (всі символи вилучено).

Якщо X та Y – рядки, то Z – **спільна підпослідовність** рядків X та Y, якщо Z – підпослідовність їх обох. Наприклад для X зі значенням ЦАТЦГА та Y зі значенням ГТАЦЦГТЦА рядок ЦЦА спільна підпослідовність X та Y, що містить три символи. А **найдовша спільна підпослідовність** (НСП) для цього прикладу – ЦТЦА.

Якщо X – рядок $x_1x_2x_3…x_m$, то i-й префікс рядка X – це рядок $x_1x_2x_3…x_i$ ($i <= m$), позначатимемо його як X_i. Наприклад, якщо X=ЦАТЦГА, то X_4=ЦАТЦ.

Можна помітити, що НСП двох рядків містить у собі НСП префіксів цих двох рядків. Розглянемо для рядки $X= x_1x_2x_3…x_m$ та $Y= y_1y_2y_3…y_n$. Позначимо їхню НСП $Z= z_1z_2z_3…z_k$ певної довжини k, $k \in [0, \min(m, n)]$.

При роботі алгоритму суттєвим етапом є аналіз останніх символів в X та Y.

Якщо останні символи однакові, то останній символ рядка Z, z_k має бути таким же символом. Решта символів Z, тобто $Z_{k-1} = z_1z_2z_3…z_{k-1}$ має бути НСП решти рядків X і Y, а саме рядків $X_{m-1}=x_1x_2x_3…x_{m-1}$ та $Y_{n-1}=y_1y_2y_3…y_{n-1}$. Наприклад, якщо X=ЦАТЦГА, Y=ГТАЦЦГТА, Z=ЦТЦА, то останній символ А співпадає для всіх трьох рядків.

Якщо останні символи не однакові, то z_k може бути або останнім символом рядка X, x_m, або останнім символом рядка Y, y_n.

Отже, треба розв’язати одну з цих двох підзадач, залежно від того, чи останні два символи X та Y однакові. Якщо так, треба розв’язати лише одну підзадачу: знайти НСП рядків X_{m-1} та Y_{n-1}, а тоді додати цей останній символ, щоб отримати НСП рядків X та Y. Якщо останні символи рядків X та Y різні, то необхідно розв’язати дві підзадачі: знайти НСП X_{m-1} та Y, знайти НСП X та Y_{n-1}, і вибрати довшу з цих двох підпослідовностей як НСП рядків X та Y. Якщо ці дві

найдовші спільні підпослідовності мають однакову довжину, можна використати довільну з них.

Розглянемо пошук НСП двох рядків в два етапи. По-перше, визначається довжина НСП. Для цього застосовується алгоритм *Evaluate LCS Table(X, Y)*.

Будемо розв'язувати задачу пошуку НСП рядків *X* та *Y* у два етапи. По-перше, визначимо довжину НСП заданих рядків, також довжину найдовших спільних підпослідовностей усіх префіксів *X* та *Y*. Після обчислення довжин НСП, відновимо за допомогою «зворотного розроблення», як відбувалося це обчислення, щоб відновити саму НСП.

Позначимо довжину НСП префіксів X_i та Y_j як *l*[*i, j*]. Відповідно, довжина НСП рядків *X* та *Y* позначена як *l*[*m, n*]. Почнімо з індексів *i* та *j*, які дорівнюють 0, бо коли один з префіксів має довжину 0, їхня НСП відома: це порожній рядок. Іншими словами, *l*[*0, j*] *l*[*i, 0*] дорівнюють 0 для всіх значень *i* та *j*. Коли обидва індекси (*i* та *j*) додатні, визначимо *l*[*i, j*] за допомогою менших значень *i* та\або *j*.

Якщо *i* та *j* додатні й x_i дорівнює y_j, то *l*[*i, j*] становить l[*i-1, j-1*] +1.

Якщо *i* та *j* додатні й x_i відрізняється від y_i, то *l*[*i, j*] дорівнює більшому з *l*[*i, j-1*] та *l*[*i-1, j*].

Розглянемо таблицю, де будуть зберігатися значення *l*[*i, j*] (табл. 7.1). Наприклад, *l*[5, 8] дорівнює 3. Це означає, що НСП рядків X_5=ЦАТЦГ та Y_8=ГТАЦЦГТЦ має довжину 3.

Таблиця 7.1 – Довжини НСП

		j	0	1	2	3	4	5	6	7	8	9
		yj		Г	Т	А	Ц	Ц	Г	Т	Ц	А
i	xi											
0			0	0	0	0	0	0	0	0	0	0
1	Ц		0	0	0	0	1	1	1	1	1	1
2	А		0	0	0	1	1	1	1	1	1	2
3	Т		0	0	1	1	1	1	1	2	2	2
4	Ц		0	0	1	1	2	2	2	2	3	3
5	Г		0	1	1	1	2	2	3	3	3	3
6	А		0	1	1	2	2	2	3	3	3	4

Щоб обчислити значення таблиці за збільшенням індексів, перед обчисленням *l*[*i, j*] треба обчислити елементи *l*[*i, j-1*], *l*[*i-1, j*].

Алгоритм *Evaluate LCS Table(X, Y)*

// Обчислення довжини найдовшої спільної підпослідовності
// Вхідові дані: X та Y – два рядки завдовжки m та n відповідно
// Виходові дані: масив *l[0..m, 0..n]*. Значення *l[m, n]* – це довжина
// найдовшої спільної підпослідовності рядків X та Y
Нехай *l[0..m, 0..n]* – новий масив
for *i* ← 0 **to** *m* **do**
  l[i, 0] ← 0

```
for j ← 0 to n do
   l[0, j] ← 0
for i ← 1 to m do
   for j ← 1 to n do
         if x_i=y_i
               l[i, j] ← l[i-1, j-1]+1
         else
               l[i, j] ← max(l[i, j-1], l[i-1, j])
return l
```

На заповнення кожного елемента таблиці потрібен сталий час, таблиця містить $(m+1)(n+1)$ елементів, тож час роботи становить $\Theta(nm)$. Елемент $l[m, n]$ міститиме НСП рядків X та Y. Для конструювання самої НСП з рядків X та Y необхідно додатково витратити $O(n+m)$ часу, для цього використовується алгоритм Restore *LCS(X,Y,l,i,j).*

Алгоритм Restore *LCS(X,Y,l,i,j)* використовується для відновлення символів найбільшої спільної послідовності.

```
Алгоритм Restore LCS(X, Y, l, i, j)
   // Поелементне відновлення найдовшої спільної підпослідовності
   // Вхідові дані: X та Y – два рядки завдовжки m та n відповідно
   // l – масив, заповнений алгоритмом Evaluate LCS Table(X, Y)
   // i, j – індекси в X та Y відповідно, а також у l.
   // Виходові дані: найдовша спільна підпослідовність рядків X та Y
   LCS ← ‘ ’
   if l[i, j] = 0
      return LCS
   else
      if x_i=y_i
            LCS ← Restore LCS(X, Y, l, i-1, j-1)
            LCS ← LCS + x_i
            return LCS
      else
            if l[i, j-1] > l[i-1, j]
                  LCS ← Restore LCS(X, Y, l, i, j-1)
                  return LCS
            else
                  LCS ← Restore LCS(X, Y, l, i-1, j)
                  return LCS
```

7.4 Алгоритм Флойда

Нехай *V* – множина деяких об’єктів, на яких задається відношення між парами елементів цієї множини. Тобто відома множина *E* пар (*i*, *j*) таких, що *i*, *j*

належать множині *V*. Множини *V* та *E* утворюють граф (*V, E*).

Елементи множини *V* називають вершинами графа, а множини *E* – ребрами. Вершини, що з'єднані ребром, називаються суміжними.

Якщо (*i, j*) – невпорядкована пара, тобто (*i, j*) та (*j, i*) – одна й та сама пара, граф вважається неорієнтованим, якщо ж (*i, j*) та (*j, i*) – різні пари, граф орієнтований. Ребра орієнтованого графа називають дугами.

Задача пошуку найкоротших шляхів між всіма парами вершин складається в пошуку для даного зваженого зв'язного графа (орієнтованого або неорієнтованого) відстаней від кожної вершини до всіх інших вершин.

Більше формальна постановка цієї задачі наступна: є орієнтований граф $G = (V, E)$, кожній дузі $v \to w$ цього графа зіставлена не негативна вартість $C[v, w]$. Загальна задача знаходження найкоротших шляхів полягає в знаходженні для кожної впорядкованої пари вершин (v, w) будь-якого шляху від вершини v у вершину w, довжина якого мінімальна серед всіх можливих шляхів від v до w.

Алгоритм Флойда (R. W. Floyd) полягає в наступному. Для визначеності покладемо, що вершини графа послідовно пронумеровані від 1 до n.

Алгоритм використовує матрицю A розміру nxn, у якій обчислюються довжини найкоротших шляхів. Спочатку $A[i, j] = C[i, j]$ для всіх $i \neq j$. Якщо дуга $i \to j$ відсутня, то $C[i, j] = \infty$. Кожний діагональний елемент матриці *A* дорівнює 0.

Над матрицею *A* виконується *n* ітерацій. Після *k-ій* ітерації *A[i, j]* містить значення найменшої довжини шляхів з вершини *i* у вершину *j*, які не проходять через вершини з номером, більшим *k*. Інакше кажучи, між кінцевими вершинами шляху *i* і *j* можуть перебувати тільки вершини, номера яких менше або рівні *k*.

На *k-ій* ітерації для обчислення матриці *A* застосовується наступна формула:

$$A_k[i, j] = min(A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]).$$

Нижній індекс *k* позначає значення матриці *A* після *k-ій* ітерації, але це не означає, що існує *n* різних матриць, цей індекс використається для скорочення запису.

Для обчислення $A_k[i, j]$ проводиться порівняння величини $A_{k-1}[i, j]$ (тобто вартість шляху від вершини *i* до вершини *j* без участі вершини *k* або іншої вершини з більше високим номером) з величиною $A_{k-1}[i, k] + A_{k-1}[k, j]$ (вартість шляху від вершини *i* до вершини *k* плюс вартість шляху від вершини *k* до вершини *j*). Якщо шлях через вершину *k* дешевше, ніж $A_{k-1}[i, j]$, то величина $A_k[i, j]$ змінюється.

На рис. 7.3 показаний позначений орієнтований граф, а на рис. 7.4 – значення матриці *A* після трьох ітерацій.

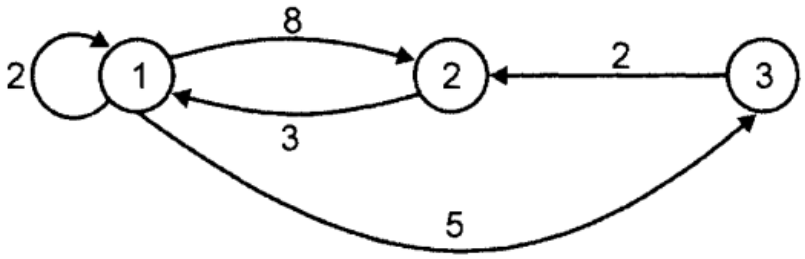


Рисунок 7.3 – Орієнтований граф

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_2[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

Рисунок 7.4 – Матриці кроків алгоритму

Процедура, що реалізує алгоритм Флойда, представлена в наступному лістингу.

```
procedure Floyd ( var A: array[1..n, 1..n] of real;
                      C: array[1..n, 1..n] of real);
    var
        i, j, k: integer;
    begin
        for i:= 1 to n do
            for j:= 1 to n do
                A[i, j]:= C[i, j];
        for i:= 1 to n do
            A[i, i]:= 0;
        for k:= 1 to n do
            for i:= 1 to n do
                for j:= 1 to n do
                    if A[i, k] + A[k, j] < A[i, j] then
                        A[i, j]:= A[i, k] + A[k, j]
    end; { Floyd }
```

Часова складність алгоритму Флойда – $\Theta(n^3)$, а просторова – $\Theta(n^2)$.

Практичні завдання

1. Реалізувати алгоритм *Флойда*.
2. Провести математичний аналіз алгоритму.
3. Провести емпіричний аналіз алгоритму *Флойда*.
4. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на уведення матриці ваг орієнтованого графа;
   - виводити результат – матрицю міток найкоротших шляхів;
   - виводити службову інформацію про параметри роботи алгоритмів (час роботи, кількість ітерацій і т.д.).
5. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму та висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

Тестові завдання

1. Який з алгоритмів відноситься до динамічного програмування?
   а) алгоритм Флойда;
   б) алгоритм Гафмена;
   в) алгоритм Прима;
   г) алгоритм Крускала.
2. Клас ефективності алгоритму обчислення біноміальних коефіцієнтів методом динамічного програмування з *k* елементів *n*-елементної множини?
   а) $\Theta(nk)$;
   б) $\Theta(n+k)$;
   в) $\Theta(\log(nk))$;
   г) $\Theta((n-1)/k)$.
3. В чому передумови застосування алгоритму Флойда?
   а) зважений орієнтований граф;
   б) планарний граф;
   в) остовний граф;
   г) бінарне дерево.
4. Математичний об’єкт «трикутник Паскаля» виникає під час роботи алгоритму?
   а) алгоритм Дейкстри;
   б) алгоритм Хорспула;
   в) алгоритм Боєра-Мура;
   г) алгоритм обчислення біноміальних коефіцієнтів.
5. Просторова складність алгоритму Флойда?
   а) $\Theta(n^2)$;

б) $\Theta(n^3)$;
в) $\Theta(\log(n))$;
г) жодне з наведених.

Контрольні запитання

1. У чому полягає особливість алгоритмів динамічного програмування?
2. Приклади задач, які ефективно вирішуються методом динамічного програмування.
3. Способи подання графів у пам’яті ЕОМ.
4. Наведіть приклади задач, які ефективніше вирішити методом повного перебору, ніж з використання динамічного програмування.
5. Чи можна розв’язати за допомогою динамічного програмування задачу пошуку строки за зразком?

Тема 8 Жадібні алгоритми

Мета: ознайомитись з прикладами застосування жадібних алгоритмів до задач оптимального кодування.

План

1. Основи жадібних алгоритмів
2. Алгоритм Гафмена

8.1 Основи жадібних алгоритмів

Жадібні алгоритми – це поширений клас алгоритмів та метод розробки. На кожному етапі роботи такий алгоритм обирає той варіант розв'язку, який є локально оптимальним у сенсі конкретної задачі. Слід зауважити, що іноді не вдається отримати повністю оптимальне рішення при такому підході, однак, отриманий розв'язок може бути «достатньо оптимальним».

Даний метод полягає у тому, щоби протягом пошуку найкращого розв'язку алгоритм відшукував все кращі та кращі варіанти розв'язку. Якщо ввести деяку кількісну оцінку якості шуканого розв'язку, то такий метод подібний на здолання все нової та нової висоти при сходженні на вершину.

Розглянемо задачу розміну монет: як виплатити суму n за допомогою найменшої кількості монет номіналом $d_1>d_2>\ldots>d_m$. Нехай, існують такі монети d_1=25, d_2=10, d_3=5, d_4=1. Як у такому разі видати суму n=48? При жадібному підході будемо на кожному етапі використовувати монети максимальним номіналом. Спочатку використаємо монету 25, що зменшить суму до 23. Наступним можливим максимальним значенням номіналу є 10. Після цього сума знизиться до 13, що каже про те, що монету номіналом 10 можна використати ще раз, а потім три рази по 1. Отже 48=25+10+10+1+1+1.

8.2 Алгоритм Гафмена

Алгоритм стиснення даних (ефективного кодування), заснований знанні статистичного розподілу даних. Д.А. Гафменом у 1952 році. Ідея алгоритму полягає в тому, що знаючи ймовірності входження символів в повідомлення, можна описати процедуру побудови кодів змінної довжини, що складаються з цілої кількості бітів. Символам з більшою ймовірністю будуть присвоєні більш короткі коди. Коди Гафмена мають унікальну префікс, що дозволяє однозначно розшифровувати їх, незважаючи на їх змінну довжину.

Класичний алгоритм Гафмена на вході отримає таблицю частот входження символів в повідомленні. Далі на основі цієї таблиці будується бінарне дерево кодування Гафмена (*H*-дерево). **Алгоритм побудови *H*-дерева наступний**:

1. Символи вхідного алфавіту утворюють список вільних вузлів дерева

(листів). Кожен лист має вагу, яка дорівнює або ймовірності, або кількості входжень символу в повідомленні.

2. Обираються два вільних вузла дерева з мінімальними вагами.

3. Створюється новий батьківський вузол з вагою, яка дорівнює сумі ваг двох вузлів.

4. Новий батьківський вузол додається в список вільних вузлів, а два вузла-потомки видаляються з цього списку.

5. Одне ребро дерева, що виходить з батьківських вузлів родителя, помічається бітом 1, а інше – 0.

6. Кроки, що починаються з другого, повторюються до тих пір, поки в списку вільних вузлів не залишиться тільки один вільний вузол. Він буде вважатися коренем дерева.

Припустимо, таблиця частот деякого повідомлення виглядає так, як представлено в табл. 8.1.

Таблиця 8.1 – Частоти символів повідомлення

А	Б	В	Г	Д
15	7	6	6	5

На першому кроці з листя дерева вибираються два з найменшими вагами – Г і Д. Вони приєднуються до нового вузла-батька, вага якого встановлюється в 5 +6 = 11. Потім вузли Г і Д видаляються зі списку вільних. Вузол Г відповідає гілці 0 батька, вузол Д – гілці 1.

На наступному кроці те ж саме відбувається з вузлами Б і В, так як тепер ця пара має найменшу вагу в дереві. Створюється новий вузол з вагою 13, а вузли Б і видаляються зі списку вільних і так далі. На рисунку 8.1 зображено дерево кодування після проведених перетворень.

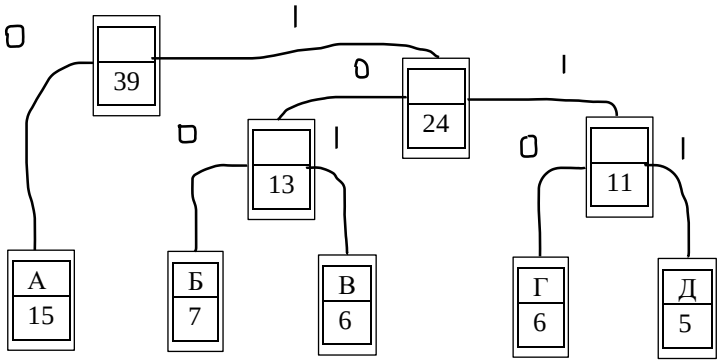


Рисунок 8.1 – Дерево Гафмена

Щоб визначити код для кожного з символів, що входять у повідомлення, ми повинні пройти шлях від листа дерева, що відповідає цьому символу, до кореня дерева, накопичуючи біти під час переміщення по гілках дерева. Отримана таким чином послідовність бітів є кодом даного символу, записаним у зворотному порядку.

Для таблиці символів 8.1 коди Гафмена виглядатимуть наступним чином

(табл. 8.2).

Таблиця 8.2 – Коди Гафмена

А	0
Б	100
В	101
Г	110
Д	111

Оскільки жоден з отриманих кодів не є префіксом іншого (), вони можуть бути однозначно декодовані під час читання їх з потоку. Крім того, найчастіший символ повідомлення А закодований найменшою кількістю бітів, а найрідкісніший символ Д – найбільшим.

Класичний алгоритм Гафмена має *один істотний недолік*. Для відновлення вмісту стисненого повідомлення декодер повинен знати таблицю частот, якою користувався кодер. Отже, довжина стисненого повідомлення збільшується на довжину таблиці частот, яка повинна посилатися попереду даних, що може звести нанівець всі зусилля зі стиснення повідомлення. Крім того, необхідність наявності повної частотної статистики перед початком власне кодування вимагає двох проходів за повідомленням: одного для побудови моделі повідомлення (таблиці частот і H-дерева), іншого для власне кодування.

Оскільки в результаті роботи алгоритму будується двійкове дерево, час роботи становить $O(nlog(n))$.

Практичні завдання

1. Реалізувати алгоритм Гафмена.
2. Виконати математичний та емпіричний аналіз ефективності.
3. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення повідомлення;
   - передбачити можливість кодування файлу;
   - виводити на екран таблицю кодів;
   - виводити службову інформацію про параметри роботи алгоритму (час роботи, кількість ітерацій тощо).
4. Звіт повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму та висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

Тестові завдання

1. Який недолік класичного алгоритму Гафмена?

а) експоненціальний час роботи;

б) необхідність зберігання дерева Гафмена;

в) не може бути однозначно декодований;

г) генерований код не є префіксним.

2. Нехай дано значення частот символів А=10, Б=5, В=2. Які коди будуть відповідати символам, якщо при побудові дерева Гафмена вільні вузли розташовуються в алфавітному порядку, а порядок міток гілок ліва – 0, права – 1?

а) А=0, Б=10, В=11;

б) А=1, Б=01, В=00;

в) А=11, Б=10, В=0;

г) А=1, Б=11, В=00.

3. Який клас ефективності алгоритму Гафмена?

а) $\Theta(n^2)$;

б) $\Theta(n^3)$;

в) $\Theta(\log(n))$;

г) жодне з наведених.

4. Яка з наведених функцій має найбільший асимптотичний порядок росту?

а) (10n)!;

б) 100ln(n);

в) n^{10};

г) $0.5n^2$.

5. Просторова складність алгоритму Гафмена?

а) $\Theta(n^2)$;

б) $\Theta(n^3)$;

в) $\Theta(\log(n))$;

г) жодне з наведених.

Контрольні запитання

1. Чи завжди жадібні алгоритми знаходять оптимальний розв’язок?

2. Які особливості аналізу жадібних алгоритмів?

3. Недоліки алгоритму Гафмена.

4. Чи можна розробити жадібний алгоритм пошуку найкоротшого шляху в графі?

5. Особливості емпіричного аналізу жадібних алгоритмів.

Тема 9 Евристичні алгоритми

Мета: ознайомитись з прикладами застосування генетичних алгоритмів до обчислювальних задач.

План

1. Поняття евристичних алгоритмів
2. Генетичні алгоритми

9.1 Поняття евристичних алгоритмів

Експоненціально складні задачі вимагають занадто багато часу та ресурсів для розв'язання. Загалом, багато практичних задач потерпають від так званого «прокляття розмірності» – часова та просторова складність розв'язку дуже швидко зростає при збільшенні кількості параметрів, що характеризуються стан системи. У таких випадках доцільним є використання евристичних алгоритмів, які маються такі особливості:

1. Вони дозволяють знайти добрі, хоча і не завжди найкращі розв'язки з усіх, що існують.
2. Метод пошуку або побудови розв'язку звичайно значно простіший, ніж той що гарантує оптимальність розв'язку.

Розв'язання задач дискретної оптимізації, таких як задачі про призначення робіт та задачі теорії розкладів, може відбуватися стандартними методами дискретного програмування, наприклад, методом гілок і меж, динамічного програмування тощо.

Поняття «добрий розв'язок» змінюється від задачі до задачі, тому його важко визначити точно. Припустимість використання евристики залежить від співвідношення часу та складності пошуку розв'язку обома способами (евристичних та класичний підхід) та співвідношення якості обох розв'язків.

Еволюційні методи набули широкої популярності при розв'язанні обчислювально складних задач через їх інтуїтивну зрозумілість та відносну простоту програмної реалізації.

9.2 Генетичні алгоритми

Методи генетичного пошуку отримані в процесі узагальнення та імітації в штучних системах таких властивостей живої природи, як природний відбір, пристосовність до змінюваних умов середовища, спадкоємність нащадками життєво важливих властивостей від батьків і т.і.

Однією з перших робіт у галузі еволюційних обчислень, які застосовували принципи біологічної еволюції до вирішення різних технічних проблем, була робота Дж. Г. Голланда. В ній він використав генетичний алгоритм для

вирішення деяких оптимізаційних задач та обґрунтував його ефективність.

Стандартний методи генетичного пошуку можуть бути описані у вигляді такої функції:

$$Gen = Gen(P0, N, L, f, \Omega, \Psi, \theta, T),$$

де $P_0 = \{H_{10},H_{20},...,H_{N0}\}$ – початкова популяція – множина рішень задачі, поданих у вигляді хромосом; $H_{j0}=\{h_{1j0}, h_{2j0},..., h_{Lj0}\}$ – j-та хромосома популяції P_0 набір значень незалежних змінних, поданих у вигляді генів; h_{ij0} – i-ий ген j-ої хромосоми популяції P_0 – значення i-го оптимізованого параметру задачі, що входить в j-те рішення; N – кількість хромосом в популяції; L – довжина хромосом, кількість генів; f – цільова функція; Ω – оператор відбору; Ψ – оператор схрещування; θ – оператор мутації; T – критерії зупинення.

Загальну схему роботи генетичного алгоритму зображену на рис. 9.1.

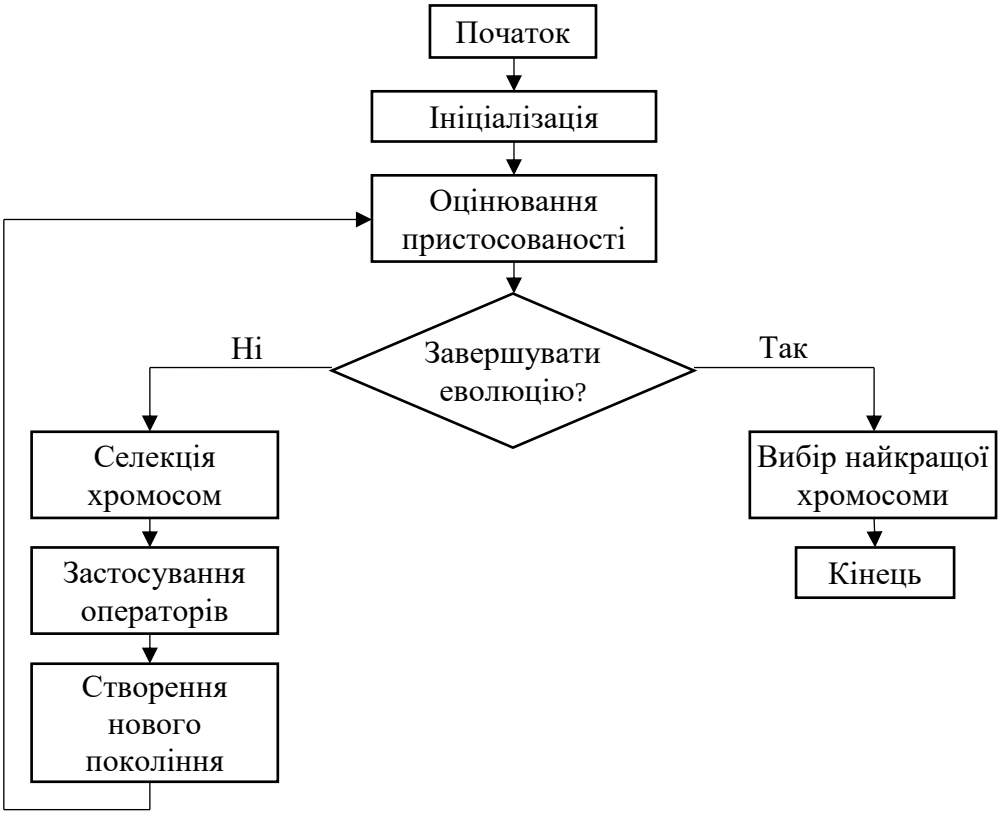


Рисунок 9.1 – Схема роботи генетичного алгоритму

Сутність генетичного пошуку полягає в циклічній заміні однієї популяції наступною, більш пристосованою. Таким чином, популяція існує не тільки в просторі, але і в часі. Часто можна вважати, що вся популяція складається в просторі і в часі з дискретних поколінь (генерацій, епох) P_0, P_1, P_2, ..., P_T.

Покоління P_{t+1} – це сукупність особин, батьки яких належать поколінню P_t. Покоління P_0 є початковою популяцією. Процес формування покоління P_0 називається ініціалізацією. Кожне покоління є результатом циклу роботи генетичного методу.

Кожна хромосома (особина, точка в просторі пошуку) оцінюється мірою її пристосованості відповідно до того, наскільки є гарним відповідне їй рішення задачі. Пристосованість визначається як обчислена цільова функція (фітнес-функція) для кожної з хромосом. Правила відбору (селекції) прагнуть залишити лише ті точки-рішення, де досягається оптимум цільової функції. Найбільш пристосовані особини дістають можливість відтворювати нащадків за допомогою перехресного схрещування з іншими особинами популяції. Це призводить до появи нових особин, які поєднують в собі деякі характеристики, успадковані ними від батьків. Найменш пристосовані особини з меншою ймовірністю зможуть відтворити нащадків, внаслідок чого ті властивості, якими вони володіли, поступово зникатимуть з популяції в процесі еволюції. Таким чином, з покоління в покоління, гарні характеристики розповсюджуються по всій популяції. Комбінація гарних характеристик від різних батьків іноді може призводити до появи пристосованого нащадка (або мутанта), чия пристосованість більша, ніж пристосованість будь-якого з його батьків. Схрещування найбільш пристосованих особин призводить до того, що досліджуються найбільш перспективні ділянки простору пошуку. Зрештою, популяція збігатиметься до оптимального рішення задачі. Після схрещування інколи відбуваються мутації – спонтанні зміни в генах, які випадковим чином розкидають точки по всій множині пошуку. Процес збіжності за рахунок відбору повинен бути більш виражений порівняно з розкидом точок за рахунок мутації і інвертування, інакше збіжність до екстремумів не матиме місця.

Подальша робота генетичного методу є ітераційним процесом застосування генетичних операторів до особин наступного покоління. Генетичні оператори необхідні, щоб застосувати принципи спадковості і мінливості до популяції.

Критерієм зупинки роботи генетичного алгоритму може бути одна з трьох подій:

1. Було сформовано задана користувачем кількість поколінь.
2. Популяція досягла заданого користувачем якості (наприклад, значення якості всіх особин перевищило заданий поріг).
3. Був досягнутий деякий рівень збіжності. Тобто особини в популяції сталі настільки подібними, що подальше їх поліпшення відбувається надзвичайно поволі.

Отже, узагальнений метод генетичного пошуку можна записати таким чином:

Крок 1. Встановити лічильник ітерацій (часу): $t = 0$. Виконати ініціалізацію (initialization) початкової популяції особин:

$$Pt = \{H_1, H_2, ..., H_M\}.$$

Крок 2. Оцінити особини поточної популяції (evaluating) шляхом обчислення їх фітнес-функції $E(H_j)$, $i = 1, 2, ..., M$.

Крок 3. Перевірити умови закінчення пошуку (termination criteria). Як такі

умови можуть бути використані: досягнення максимально допустимого часу функціонування методу, числа ітерацій, значення функції пристосованості і т. ін. Якщо критерії закінчення пошуку задовільнено, тоді виконати перехід до кроку 12.

Крок 4. Збільшити лічильник ітерацій (часу): $t = t + 1$.

Крок 5. Вибрати частину популяції (батьківські особини) для схрещування (selection of parents) *P'*.

Крок 6. Сформувати батьківські пари (mating) з особин, що відібрані на попередньому кроці.

Крок 7. Схрестити (crossover) вибрані батьківські особини.

Крок 8. Застосувати оператор мутації (mutation) до особин *P'*.

Крок 9. Обчислити нову функцію пристосованості $E(H_j)$ особин, отриманих в результаті схрещування та мутації.

Крок 10. Сформувати нове покоління шляхом вибору особин, що вижили, виходячи з рівня їх пристосованості (replacing, selection of survivors).

Крок 11. Перейти до кроку 3.

Крок 12. Зупинення.

Таким чином, при реалізації генетичних методів необхідно:

– визначити параметри, що оптимізуються, залежно від вирішуваної задачі, вибрати спосіб кодування (подання в хромосомі) параметрів, що оптимізуються;

– задати цільову функцію;

– визначити правила ініціалізації початкової популяції;

– вибрати оператори відбору, схрещування і мутації, а також задати їх параметри;

– визначити критерії зупинення.

Оскільки популяції складаються з числових значень (час виконання робіт), ми можемо використовувати відповідно такі функції схрещування: «одноточковий» (англ. single point crossover), «двоточковий» (англ. two point crossover), «рівномірний» (англ. uniform crossover), «розсіяний» (англ. scattered crossover).

Приклад. Розглянемо застосування кожного з кросоверів на числовому прикладі двох батьківських розв'язків (H1=(18,21,24,18,24,16,21,29,17,25), H2=(19,20,22,21,21,18,20,30,14,25)).

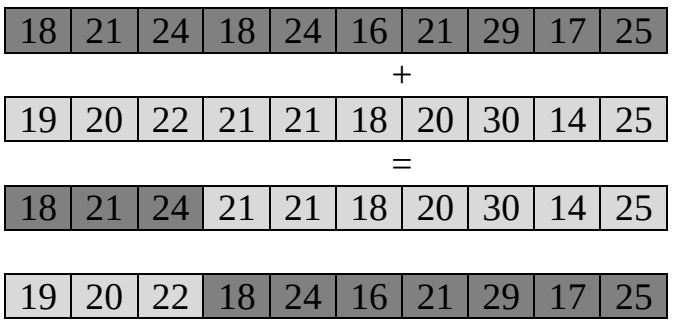


Рисунок 9.2 – «Одноточковий» (англ. single point crossover)

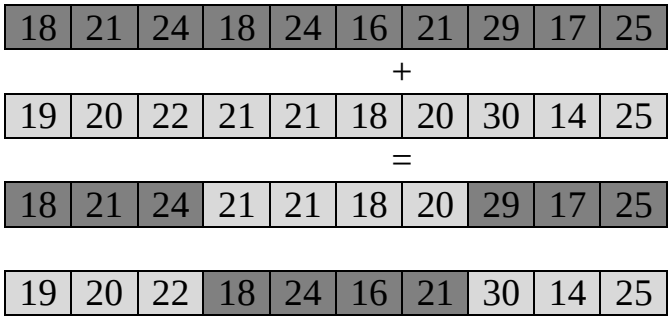


Рисунок 9.3 – «Двоточковий» (англ. two point crossover)

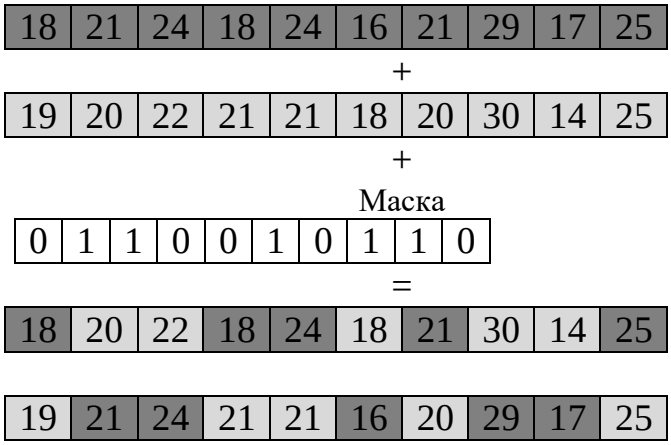


Рисунок 9.4 – «Рівномірний» (англ. uniform crossover) та «розсіяний» (англ. scattered crossover)

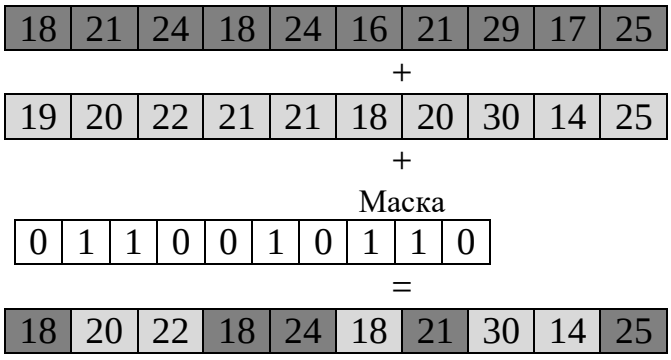


Рисунок 9.5 – «Розсіяний» (англ. scattered crossover)

У всіх кросоверів схожий принцип, однак, відрізняються деталі реалізації.

Перейдемо до восьмого кроку. Розглянемо оператори мутації, які реалізовано в системі («випадкова», «обмін», «скремблінг», «інверсія»).

Випадкова мутація, випадковим чином змінює значення деяких генів. Кількість генів, зазвичай визначається певною константою (наприклад, mutation_percent_genes).

Для кожного гена вибирається випадкове значення відповідно до діапазону, визначеного двома атрибутами random_mutation_min_val і random_mutation_max_val. Випадкове значення додається до вибраного гена (рис. 9.6).

Мутація обміну приводить до обміну значеннями двох генів, що підлягають мутації (рис. 9.7).

При інверсивній мутації вибирається підмножина генів та інвертується (рис. 9.8)

Скремблінг призводить до перемішування у випадковому порядку деякої підмножини генів (рис. 9.9).

До мутації

18	21	24	18	24	16	21	29	17	25

Після мутації (mutation_percent_genes=30%)

18	21	24	20	24	16	24	29	17	20

Рисунок 9.6 – Випадкова мутація

До мутації

18	21	24	18	24	16	21	29	17	25

Після мутації

18	21	24	21	24	16	18	29	17	25

Рисунок 9.7 – Мутація обміну

До мутації

18	21	24	18	24	16	21	29	17	25

Після мутації

18	21	24	21	16	24	18	29	17	25

Рисунок 9.8 – Мутація інверсії

До мутації

18	21	24	18	24	16	21	29	17	25

Після мутації

18	18	21	16	24	24	21	29	17	25

Рисунок 9.9 – Мутація скремблінгу

Переваги генетичних алгоритмів стають ще більш прозорими, якщо розглянути основні їх відмінності від традиційних методів:

1. Генетичні алгоритми працюють з кодами, в яких був представлений набір параметрів, які напряму залежать від аргументів цільової функції. Причому інтерпретація цих кодів відбувається тільки перед початком роботи алгоритму і після завершення його роботи для отримання результату. У процесі роботи маніпуляції з кодами відбуваються абсолютно незалежно від їх інтерпретації, код розглядається просто як бітовий рядок.

2. Для пошуку генетичний алгоритм використовує декілька точок пошукового простору одночасно, а не переходить від точки до точки, як це робиться в традиційних методах. Це дозволяє подолати один з їх недоліків –

небезпека попадання в локальний екстремум цільової функції, якщо вона має декілька екстремумів.

3. Генетичні алгоритми в процесі роботи не використовують ніякої додаткової інформації, що підвищує швидкість роботи. Єдиною інформацією, що використовується, може бути область допустимих значень параметрів і цільової функції в довільній точці.

4. Генетичний алгоритм використовує як правило ймовірності для породження нових точок аналізу, так і детерміновані правила для переходу від одних точок до інших. Одночасне використовування елементів випадковості і детермінованості дає значно більший ефект, ніж роздільне.

Після завершення роботи генетичного алгоритму з кінцевої популяції вибирається та особина, яка дає максимальне (або мінімальне) значення цільової функції і є, таким чином, результатом роботи генетичного алгоритму. За рахунок того, що кінцева популяція краща початкової, отриманий результат є поліпшеним рішенням.

Генетичні алгоритми широко застосовуються для задач комбінаторної оптимізації, одним з прикладів, може бути розв'язання діофантових рівнянь. Розглянемо діофантове (тільки цілі розв'язки) рівняння: $a + 2b + 3c + 4d = 30$, де a, b, c і d – деякі додатні цілі числа. Застосування генетичного алгоритму за дуже короткий час знаходить шуканий розв'язок (a, b, c, d).

Звичайно, можна просто підставити всі можливі значення a, b, c і d (очевидно, $1 \le a,b,c,d \le 30$. Але архітектура систем генетичних алгоритмів дозволяє знайти розв'язок швидше за рахунок більш направленого перебору.

Практичні завдання

1. Реалізувати генетичний алгоритм розв'язання діофантових рівнянь.
2. Провести емпіричний та математичний аналіз алгоритму.
3. Реалізувати інтерфейс користувача. Функціональні можливості інтерфейсу:
   - виводити запрошення на введення коефіцієнтів рівняння;
   - виводити значення популяцій на кожній ітерації алгоритму та значення функції пристосованості;
   - виводити службову інформацію про параметри роботи алгоритму (час роботи, кількість ітерацій тощо).
4. Звіт з роботи повинен містити:
   - формулювання завдання;
   - лістинг програмного коду;
   - етапи проведення математичного та емпіричного аналізу алгоритму та висновок щодо отриманих результатів;
   - скриншоти роботи програми для різних вхідних даних;
   - відповіді на контрольні запитання.

Тестові завдання

1. Набір потенційних розв’язків задачі називається ____
   а) популяція;
   б) ген;
   в) хромосома;
   г) жодного з цих.
2. Для яких задач з перерахованих доцільно застосування генетичного алгоритму?
   а) задача комівояжера;
   б) задача про рюкзак;
   в) пошук перетину двох множин;
   г) обчислення біноміальних коефіцієнтів.
3. Оператор відбору генетичного алгоритму використовується для ___
   а) визначення батьківських елементів для генерації нащадків;
   б) випадкової зміни певних генів у хромосомах;
   в) генерації початкової популяції;
   г) жодне з наведених.
4. Нехай для батьківських елементів А=(2, 5, 9, 22) та Б=(0, 1, 6, 8) згенеровано новий розв’язок-нащадок В=(2, 5, 6, 8). Який тип оператора кросовера при цьому застосовувався?
   а) одноточковий;
   б) двоточковий;
   в) рівномірний;
   г) розсіяний.
5. В результаті мутації з розв’язку А=(1, 9, 2, 5, 3, 10, 0) отримано розв’язок Б=(1, 9, 5, 10, 3, 2, 0). Який тип оператора мутації використано?
   а) скремблінгу;
   б) інверсії;
   в) обміну;
   г) жодне з наведених.

Контрольні запитання

1. Що таке хромосома та популяція?
2. Які критерії зупинки зазвичай використовуються у генетичних алгоритмах?
3. Яка мета оператору кросовера?
4. Як впливає ступінь мутації на досягнення оптимального розв’язку.
5. Яка властивість класичного інтуїтивного визначення алгоритмів порушується у евристиках?

Приклади індивідуальних завдань

Змістовна частина завдання

Індивідуальна робота полягає у розробці програмної реалізації обраного алгоритму. Перелік алгоритмів наводиться нижче. Необхідно обрати один з алгоритмів, що не перетинається з опрацьованими в ході виконання лабораторних робіт даного курсу.

Індивідуальна робота оформлюється у вигляді звіту, який обов'язково має містити наступні елементи:

1. Теоретична частина (орієнтовний обсяг – 1-2 сторінки формату А4):
   - для яких типів задач використовується;
   - основні кроки реалізації алгоритму;
   - переваги та недоліки в порівнянні з іншими алгоритмами в межах одного блоку.
2. Практична частина:
   - блок-схема алгоритму;
   - лістинг програмного коду;
   - скриншоти роботи програми.

До звіту додається скомпільована програма.

Перелік алгоритмів для реалізації в індивідуальному завданні

1. Алгоритми сортування:
   - сортування вставками;
   - сортування перемішуванням;
   - блокове сортування;
   - сортування підрахунком;
   - сортування злиттям;
   - сортування за допомогою двійкового дерева;
   - сортування Шелла;
   - пірамідальне сортування;
   - порозрядне сортування;
   - алгоритми зовнішнього сортування.

2. Алгоритми стиснення:
   - алгоритм Лемпеля-Зіва;
   - арифметичне кодування;
   - алгоритм Хаффмана;
   - алгоритм Шеннона-Фано.

3. Алгоритми шифрування:
   - блокове шифрування;
   - потокове шифрування (гамування);
   - асиметричне шифрування (RSA).

4. Алгоритми на графах:
   – алгоритм Прима;
   – алгоритм Крускала;
   – алгоритм Борувки;
   – алгоритм Дейкстри;
   – алгоритм Флойда;
   – хвильовий алгоритм пошуку шляху в лабіринті.

5. Теоретико-числові алгоритми:
   – задача розкладання заданого числа на прості множники (факторизація цілого числа);
   – задача перевірки, чи є дане число простим (тест простоти).

6. Математичні об’єкти:
   – реалізувати абстрактний тип даних – матриця. Реалізувати дії над матрицями:
      – додавання, множення матриць;
      – знаходження оберненої матриці;
      – обчислення визначника;
   – реалізувати абстрактний тип даних – комплексні числа. Реалізувати арифметичні дії над комплексними числами та графічне подання комплексних чисел.

7. Пошук підрядка в рядку:
   – алгоритм послідовного (прямого) пошуку;
   – алгоритм Рабина-Карпа;
   – алгоритм Кнута-Морріса-Пратта;
   – алгоритм Бойєра-Мура.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна:

1. Васильєв О. Алгоритми. Київ: Видавництво «Ліра-К», 2022. 424 с.
2. Ковалюк Т.В. Алгоритмізація та програмування. Львів: «Магнолія 2006», 2013. 400 с.
3. Кормен Т.Г. Алгоритми доступно / пер. з англ. К. Яценка. Київ: К.І.С., 2021. 194 с.
4. Креневич А.П. Алгоритми і структури даних. Київ: ВПЦ «Київський Університет», 2021. 200 с.
5. Матвієнко М.П. Теорія алгоритмів. Київ: Видавництво «Ліра-К», 2017. 340 с.
6. Матвієнко М.П., Шаповалов С.П. Математична логіка та теорія алгоритмів. Київ: Видавництво «Ліра-К», 2017. 212 с.
7. Ришковець Ю.В., Висоцька В.А. Алгоритмізація та програмування. Частина І. Львів: Видавництво «Новий світ-2000», 2018. 337 с.
8. Ришковець Ю.В., Висоцька В.А. Алгоритмізація та програмування. Частина ІІ. Львів: Видавництво «Новий світ-2000», 2018. 316 с.
9. Шаховська Н.Б., Голощук Р.О. Алгоритми та структури даних. Львів: «Магнолія 2006», 2020. 216 с.
10. Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms. The MIT Press, third edition, 2009. 1292 p.
11. Levitin A. Introduction the Design & Analysis of Algorithms. Pearson, third edition, 2012. 565 p.

Додаткова:

1. Глибовець М.М., Гулаєва Н.М. Еволюційні алгоритми. Київ: НаУКМА, 2013. 826 с.
2. Agarwal B., Baka B. Hands-On Data Structures and Algorithms with Python: Write complex and powerful code using the latest features of Python 3.7. Ingram short title, 2nd edition, 2018. 398 p.
3. Stephens R. Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#. Wiley, 2nd edition, 2019. 800 p.

Інформаційні джерела:

1. Наукова бібліотека Запорізького національного університету. URL: http://library.znu.edu.ua/
2. Система електронного забезпечення навчання ЗНУ. URL: https://moodle.znu.edu.ua/
3. Національна бібліотека України імені В.І. Вернадського. URL: http://www.nbuv.gov.ua/

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Ковалюк Т.В. Алгоритмізація та програмування. Львів: «Магнолія 2006», 2018. 400 с.

2. Кормен Т.Г. Алгоритми доступно / пер. з англ. К. Яценка. Київ: К.І.С., 2021. 194 с.
3. Levitin A. Introduction the Design & Analysis of Algorithms. Pearson, third edition, 2012. 565 p.

Навчальне видання
(українською мовою)

Гребенюк Сергій Миколайович
Кудін Олексій Володимирович
Лісняк Андрій Олександрович
Столярова Анастасія Валеріївна

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмна інженерія»

Рецензент *С.І. Гоменюк*
Відповідальний за випуск *А.О. Лісняк*
Коректор *С.М. Гребенюк*