І. Р. Арсенюк, В. В. Колодний, А. А. Яровий

ТЕОРІЯ АЛГОРИТМІВ

Міністерство освіти і науки України
Вінницький національний технічний університет

І. Р. Арсенюк, В. В. Колодний, А. А. Яровий

ТЕОРІЯ АЛГОРИТМІВ

Затверджено Вченою радою Вінницького національного технічного університету як навчальний посібник для студентів спеціальностей “Інтелектуальні системи прийняття рішень” та „Програмне забезпечення автоматизованих систем”. Протокол № 4 від 27 жовтня 2005 р.

Вінниця ВНТУ 2006

УДК 519
А 85

Р е ц е н з е н т и:

О. В. Осадчук, доктор технічних наук, професор
А. М. Пєтух, доктор технічних наук, професор
Л. І. Тимченко, доктор технічних наук, професор

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України

Арсенюк І. Р., Колодний В. В., Яровий А. А.
А85 **Теорія алгоритмів.** Навчальний посібник. – Вінниця: ВНТУ, 2006. – 150 с.

В посібнику наведені основні підходи щодо уточнення поняття алгоритму та теоретичні основи аналізу ефективності алгоритмів. Розглянуто та проаналізовано ряд алгоритмів сортування, динамічного програмування, а також жадібні алгоритми.

Посібник розроблений у відповідності з планом кафедри та програмою дисципліни ”Теорія алгоритмів”.

УДК 519

ЗМІСТ

ВСТУП

Поняття алгоритму є центральним об'єктом спеціального розділу математики – *теорії алгоритмів*. Першим алгоритмом, що дійшов до нас, в його інтуїтивному розумінні, вважається запропонований Евклідом у III столітті до нашої ери алгоритм пошуку найбільшого загального дільника двох чисел. Протягом тривалого часу, аж до початку XX століття слово „алгоритм" вживалося у поєднанні „алгоритм Евкліда", а для опису покрокового розв'язання інших математичних задач використовувалося слово „метод".

Початковою точкою відліку сучасної теорії алгоритмів вважать роботу німецького математика К. Геделя (1931 р. – теорема про неповноту символічних логік), в якій було показано, що деякі математичні проблеми не можуть бути розв'язані алгоритмами з певного класу. Ця робота дала поштовх до пошуку і аналізу різних формалізацій алгоритму.

Перші фундаментальні роботи з теорії алгоритмів були опубліковані незалежно в 1936 році А. Тьюрінгом, А. Черчем та Е. Постом. Запропоновані ними машина Тьюрінга, машина Поста і лямбда-числення Черча були еквівалентними формалізаціями алгоритму. Сформульовані ними тези (Поста і Черча-Тьюрінга) постулювали еквівалентність запропонованих ними формальних систем та інтуїтивного поняття алгоритму. Важливим розвитком цих робіт стало формулювання і доведення алгоритмічно нерозв'язуваних проблем.

У 1950-і роки істотний внесок в теорію алгоритмів внесли Н. Колмогоров і А. Марков.

До 1970-х років сформувалися такі три напрями в теорії алгоритмів.

1. *Класична теорія алгоритмів* (формулювання завдань у термінах формальних мов, поняття задачі розв'язності, введення класів складності, формулювання проблеми P=NP, відкриття класу NP-повних задач та його дослідження).

2. *Теорія асимптотичного аналізу алгоритмів* (поняття складності і трудомісткості алгоритму, критерії оцінювання алгоритмів, методи отримання асимптотичних оцінок), у розвиток якої зробили істотний внесок Д. Кнут, А. Ахо, Д. Хопкрофт, Д. Ульман.

3. *Теорія практичного аналізу алгоритмів* (отримання явних функцій трудомісткості, інтервальний аналіз функцій, практичні критерії якості алгоритмів, методика вибору раціональних алгоритмів), найголовнішою роботою в цьому напрямку вважають фундаментальну працю Д. Кнута „Искусство программирования для ЭВМ".

В загальному випадку теорія алгоритмів розглядає такі питання як формалізація поняття „алгоритм” та дослідження формальних алгоритмічних систем; формальне доведення алгоритмічної нерозв’язуваності ряду завдань; класифікація завдань, визначення і дослідження класів складності; асимптотичний аналіз складності алгоритмів; дослідження та аналіз рекурсивних алгоритмів; отримання явних функцій трудомісткості для порівняльного аналізу алгоритмів; розробка критеріїв порівняльного оцінювання якості алгоритмів.

Одержані в теорії алгоритмів теоретичні результати знаходять достатньо широке практичне застосування. При цьому під час дослідження деякого завдання результати теорії алгоритмів дозволяють відповісти на ряд важливіших питань. Чи є це завдання принципово алгоритмічно розв’язним? І якщо так – то чи не належить воно до класу NP–повних завдань, при ствердній відповіді на яке можна говорити про істотні часові затрати для отримання точного розв’язку у випадку значних розмірностей вихідних даних. Крім цього, завдяки методам теорії алгоритмів стає можливим раціональний вибір алгоритму (з відомої множини алгоритмів) розв’язання даної задачі з урахуванням особливостей їх застосування; отримання часових оцінок розв’язання складних задач; отримання вірогідних оцінок неможливості розв’язання деякої задачі за певний час; розробку та вдосконалення ефективних алгоритмів на основі практичного аналізу тощо.

У посібнику розглянуто такі основні питання, як інтуїтивне поняття алгоритму та основні підходи до його уточнення; теоретичні та практичні основи оцінки ефективності алгоритмів; аналіз різних алгоритмів сортування; динамічне програмування; жадібні алгоритми, а також NP-повні алгоритми.

Кожен розділ містить теоретичний матеріал, який ілюструється прикладами розв’язання відповідних задач, контрольні питання і завдання для лабораторних занять. Такий підхід дозволяє досить ґрунтовно опанувати і засвоїти основи теорії алгоритмів та навчитись їх застосовувати під час розв’язання практичних задач. Крім того, в посібнику наведено список літератури, що дозволяє значно глибше вивчити теоретичні основи теорії алгоритмів, та її практичне застосування.

1 ІНТУЇТИВНЕ ПОНЯТТЯ АЛГОРИТМУ ТА ОСНОВНІ ПІДХОДИ ДО ЙОГО УТОЧНЕННЯ

1.1 Логічна та аналітична теорія алгоритмів

Теорія алгоритмів – це розділ математики, що вивчає загальні властивості алгоритмів. Відокремлюють 2 гілки теорії: *логічну теорію*, яка стосується питання конструктивного обґрунтування математики та вивчення феномену алгоритмічної нерозв'язуваності проблем, і *аналітичну теорію* алгоритмів, яка пов'язана з вивченням самих алгоритмів, аналізом їх структури, методами еквівалентних перетворень, способами побудови і оцінювання ефективності. Отже, оскільки центральним поняттям даної теорії є алгоритм, далі детальніше зупинемось на ньому.

Слово „алгоритм" походить від імені узбецького математика Хорезмі (арабскою аль-Хорезмі), який у IX столітті нашої ери розробив правила чотирьох арифметичних дій над числами в десятковій системі числення. Сукупність цих правил у Європі стали називати „алгорізм". У подальшому це слово перетворилось у „алгоритм" [7].

З алгоритмами, тобто процедурами, що однозначно приводять до результату, математика мала справу завжди. Шкільні методи множення „стовпчиком" і ділення „кутом", метод виключення невідомих під час розв'язання системи лінійних рівнянь, правило диференціювання складної функції, спосіб побудови трикутника за трьома заданими сторонами – все це алгоритми. Однак доки математика мала справу в основному з числами й обчисленнями, поняття алгоритму ототожнювалося з поняттям методу обчислення; потреби у вивченні самого цього поняття не виникало. Традиції організації обчислень формувалися століттями й стали складовою частиною загальної наукової культури так само, як і елементарні навички логічного мислення. Все різноманіття обчислень комбінувалося з 10-15 чітко визначених операцій арифметики, тригонометрії й аналізу. Тому поняття методу обчислення вважалося споконвічно зрозумілим і не мало потреби в спеціальних дослідженнях [6, 7].

До середини XIX століття єдиною областю математики, що працювала з нечисловими об'єктами, була геометрія, і саме вона, не маючи можливості опиратися на обчислювальну інтуїцію людини, значно відрізнялася від іншої математики підвищеними вимогами до чіткості своїх міркувань. Важке звикання до нових, підвищених вимог чіткості почалося у математиці в другій половині XIX століття. Воно стимулювалося в основному математикою нечислових об'єктів – відкриттям неевклідових геометрій, появою абстрактних алгебраїчних теорій типу теорії груп і т. д.

Однією з вирішальних обставин, які привели до перегляду основ математики, тобто принципів, що лежать в основі математичних міркувань,

було створення Кантором теорії множин. Досить швидко стало ясно, що поняття теорії множин лежить в основі всієї математики. Однак майже настільки ж швидко було показано, що деякі думки, які є цілком природними міркуваннями в рамках цієї теорії, приводять до нерозв'язуваних протиріч – парадоксів теорії множин (які до цього вважалися інтуїтивно зрозумілими) [6, 7]. Виникла особлива галузь математики – метаматематика.

Досвід парадоксів теорії множин навчив математика дуже обережно поводитися з нескінченністю і якщо можна навіть про нескінченність розмірковувати за допомогою фінітних методів [6, 7]. Сутність фінітного підходу полягає в тім, що він допускає тільки кінцеві послідовності дій над кінцевим числом об'єктів. З'ясування того, які об'єкти та дії над ними слід вважати точно визначеними, які властивості та можливості мають комбінації елементарних дій, що можна, а чого не можна зробити за їх допомогою – все це стало предметом теорії алгоритмів і формальних систем, що спочатку виникли в рамках метаматематики й стали найважливішою її частиною [7]. Головним внутрішньоматематичним додатком теорії алгоритмів вважалися докази неможливості алгоритмічного (тобто точного й однозначного) розв'язання деяких математичних проблем. Такі докази (та й точні формулювання тверджень, що доводяться) є нездійсненними без точного поняття алгоритму.

Поки техніка використовувала чисто обчислювальні методи, ці „високі" проблеми чистої математики її майже не цікавили. У техніку термін „алгоритм" прийшов разом з кібернетикою. Якщо поняття методу обчислення не мало потреби в поясненнях, то поняття процесу керування довелося виробляти практично заново. Знадобилося усвідомлювати, які вимоги повинна задовольняти послідовність дій (або її опис), щоб вважатися конструктивно заданою, тобто мати право називатися алгоритмом. У цьому усвідомленні величезну допомогу інженерній інтуїції зробила практика використання обчислювальних машин, що зробила поняття алгоритму відчутною реальністю.

З точки зору сучасної практики *алгоритм* – це програма, а критерієм алгоритмічності процесу є можливість його запрограмувати. Саме завдяки цій реальності алгоритму, а також тому, що підхід інженера до математичних методів завжди був конструктивним, поняття алгоритму в техніці за короткий термін стало надзвичайно популярним (можливо, навіть більше, ніж у самій математиці).

Однак у всякої популярності є свої „втрати". У повсякденній практиці слово „алгоритм" вживається занадто широко, найчастіше втрачаючи свій точний зміст. Приблизні описи поняття „алгоритм" часто приймаються за точні означення. В результаті за алгоритм найчастіше видається будь-яка інструкція, розбита на кроки. З'являються такі дикі словосполучення, як „алгоритм винаходу" (але ж наявність „алгоритму винаходу" означало б кінець винахідництва як творчої діяльності) [6, 7].

Чітке визначення алгоритму, важливе, звичайно, не тільки для правильного слововживання, а й під час розробки конкретних алгоритмів, особливо коли маємо на увазі їх наступне програмування. Однак воно ще важливіше при наведенні порядку в бурхливо зростаючому алгоритмічному світі. І для того, щоб орієнтуватися у величезній кількості алгоритмів, необхідно вміти порівнювати різні алгоритми розв'язання однакових задач, причому не лише за якістю розв'язку, а й за характеристиками самих алгоритмів (числом дій, обсягом пам'яті тощо). Таке порівняння неможливе без введення точної мови для обговорення усіх цих питань. Іншими словами, самі алгоритми повинні стати такими ж предметами точного дослідження, як і ті об'єкти, для роботи з якими вони призначені.

1.2 Властивості та форми подання алгоритмів

По-перше, що слід визначити в будь-якому алгоритмі – це те, що він застосовується до вхідних даних і видає результати. У звичних технічних термінах це означає, що алгоритм має *входи* й *виходи*. Крім того, в процесі роботи алгоритму з'являються проміжні результати, які використовуються надалі. Таким чином, кожний алгоритм має справи з *даними* – вхідними, проміжними та вихідними [6].

Зрозуміло, що ці об'єкти повинні бути чітко визначені і відрізнятися як один від одного, так і від „необ'єктів". У багатьох важливих випадках добре зрозуміло, що це означає: до таких алгоритмічних об'єктів відносяться числа, вектори, матриці суміжностей графів, формули тощо. Зображення (наприклад, рисунок графа) виглядають менш природними, ніж алгоритмічні об'єкти. Якщо говорити про граф, то справа навіть не в тому, що на рисунку більше несуттєвих деталей і різні люди той самий граф зобразять по-різному (зрештою, різні матриці суміжності теж можуть задавати той самий граф з точністю до ізоморфізму), а в тому, що матриця суміжності легко розбивається на елементи, причому з елементів всього двох видів (нулів і одиниць) складають матриці будь-яких графів, тоді як розбити на елементи рисунок набагато складніше. Нарешті, з такими об'єктами, як „гарна книга" або „осмислене твердження", з якими легко впорається будь-яка людина (але кожна по-своєму), алгоритм працювати відмовиться, поки вони не будуть описані як дані за допомогою інших, більш „підходящих" об'єктів.

Замість того, щоб намагатися дати загальне словарне означення чіткої визначеності об'єкта, у теорії алгоритмів фіксують конкретні кінцеві набори вихідних об'єктів (які називають елементарними) і кінцевий набір засобів побудови інших об'єктів з елементарних. Набір елементарних об'єктів утворює кінцевий *алфавіт* вихідних символів (цифр, букв і т.д.), з яких будуються інші об'єкти. Типовим засобом побудови є індуктивні означення, що вказують як будувати нові об'єкти з вже побудованих. Найпростіше індуктивне означення – це означення деякої множини слів, класичним

прикладом якого служить означення ідентифікатора: ідентифікатор – це або буква, або ідентифікатор, до якого приписана праворуч буква або цифра. Слова кінцевої довжини в кінцевих алфавітах (зокрема, числа) – найбільш звичайний тип алгоритмічних даних, а довжина слова – природна одиниця вимірювання обсягу оброблюваної інформації. Більш складний випадок алгоритмічних об'єктів – формули. Вони також визначаються індуктивно і також є словами в кінцевому алфавіті, однак, не кожне слово в цьому алфавіті є формулою. У такому випадку, зазвичай, основним алгоритмам передують допоміжні, які перевіряють, чи задовольняють вихідні дані потрібні вимоги. Така перевірка називається *синтаксичним аналізом*.

По-друге, дані для свого розміщення вимагають *пам'яті*. Пам'ять, зазвичай, вважається однорідною й дискретною, тобто складається з однакових комірок, причому кожна комірка може містити один символ алфавіту даних. Таким чином, одиниці вимірювання обсягу даних і пам'яті узгоджені. При цьому пам'ять може бути нескінченною. Питання про те, чи потрібна спільна або окрема пам'ять для кожного із трьох видів даних (вхідних, вихідних і проміжних), вирішується по-різному.

По-третє, алгоритм складається з окремих *елементарних кроків* або *дій*. І кожна така дія повинна бути виконаною ще до виконання наступної дії. Дана властивість алгоритму отримала назву ***дискретність***. Типовий приклад множини елементарних дій – система команд комп'ютера. Звичайно, елементарний крок має справу з фіксованим числом символів (це зручно, наприклад, для вимірювання часу роботи алгоритму числом виконаних кроків).

Властивість ***зрозумілість*** – означає, що алгоритм може бути виконаний, якщо цілком зрозуміла кожна дія (команда) і вона може бути виконана у повній відповідності до її призначення.

По-четверте, послідовність кроків алгоритму повинна бути такою, що після кожного кроку або вказується який крок робити далі, або дається команда зупинки, після чого робота алгоритму вважається завершеною. Така властивість алгоритму отримала назву ***точність***.

Часто під одночасним виконанням властивостей *зрозумілість* та *точність* розуміють таку властивість як ***визначеність*** або ***детермінованість***.

По-п'яте, логічно від алгоритму чекати *результативності*, тобто зупинки після кінцевого числа кроків (що залежить від даних) з отриманням результату. Дана властивість і отримала назву ***результативність***. Зокрема, будь-хто, хто розробляє алгоритм розв'язання деякого завдання, наприклад, обчислення функції *f(x)*, зобов'язаний показати, що алгоритм зупиняється після кінцевого числа кроків (як кажуть, з*бігається*) для будь-якого *x* з області завдання. Однак перевірити збіжність набагато складніше. Збіжність, звичайно, не вдається встановити простим переглядянням опису алгоритму, а загального методу перевірки збіжності, придатного для будь-якого алгоритму *A* і будь-яких даних *x*, взагалі не існує [6].

По-шосте, кожен алгоритм повинен мати властивість ***масовості***. Ця властивість передбачає, що даний алгоритм придатний для розв'язання будь-якої задачі з деякого класу задач. Проте дану властивість не слід розуміти як можливість розв'язання багатьох задач, оскільки у деяких випадках клас може складатися лише з однієї задачі.

По-сьоме, слід розрізняти: а) *опис алгоритму* (інструкції або програми); б) *механізм реалізації алгоритму* (наприклад, за допомогою комп'ютера), що включає засіб пуску, зупинки, реалізації елементарних кроків, видачі результатів і забезпечення управління ходом обчислення; в) *процес реалізації алгоритму*, тобто послідовність кроків, що буде породжена при застосуванні алгоритму до конкретних даних.

Припустимо, що опис алгоритму й механізм його реалізації скінченні (пам'ять, як зазначалося, може бути нескінченною, але вона не включається у механізм).

Приклад 1.1 Розглянемо таку задачу: дана послідовність P, що складається з n додатних чисел (n – довільне скінченне число); потрібно впорядкувати їх, тобто побудувати послідовність R, у якій ці ж числа розташовані у порядку зростання. Майже відразу спадає на думку такий простий спосіб її розв'язання: переглядаємо P і знаходимо в ній найменше число; викреслюємо його з P і виписуємо його як перше число R; знову повертаємося до P і знаходимо в ній найменше число; приписуємо його праворуч до отриманої частини R і так далі, доти, поки у P не будуть викреслені всі числа.

Виникає логічне питання: що означає „і так далі"?. Для більшої зрозумілості перепишемо опис способу розв'язання в більш чіткій формі, розбивши його на кроки й вказавши переходи між кроками.

Крок 1. Шукаємо у P найменше число.

Крок 2. Знайдене число приписуємо праворуч до R (у початковий момент R порожня) і викреслюємо його з P.

Крок 3. Якщо в P немає чисел, то переходимо до кроку 4. У протилежному випадку переходимо до кроку 1.

Крок 4. Завершення. Результатом вважати послідовність R, побудовану до даного моменту.

Більшість вважає такий опис досить зрозумілим (і навіть занадто формальним) для того, щоб користуючись ним, однозначно одержати потрібний результат. Однак це враження зрозумілості опирається на деякі неявні припущення, до правильності яких ми звикли, але які неважко порушити. Наприклад, що значить „дана" послідовність чисел? Чи є такою послідовність $\sqrt[7]{3}$, $\sqrt[5]{2}$, $(1/2)^{\pi}$? Очевидно, що є. Однак у нашому описі нічого не сказано, як знайти найменше число серед таких чисел. У ньому взагалі не говориться про те, як шукати найменші числа. Очевидно, передбачається, що мова йде про числа, подані у вигляді десяткових дробів, і що відомо як їх порівнювати.

Отже, необхідно уточнити форми подання даних. При цьому не можна просто заявити, що припустимо будь-яке подання чисел. Адже для кожного подання існує свій алфавіт (який крім цифр, може включати коми, дужки, знаки операцій і функцій тощо) і свій спосіб порівняння чисел (наприклад, спосіб переведення в десятковий дріб), тоді як кінцевість алфавіту вимагає фіксувати його заздалегідь, а кінцевість опису алгоритму дозволяє включити в нього лише заздалегідь фіксоване число способів порівняння. Фіксація подання чисел у вигляді десяткових дробів також не вирішує усіх проблем. Порівняння 200-, 300-розрядних чисел вже не може вважатися елементарною дією. В машинних алгоритмах саме подання числа вимагає подальшого уточнення: потрібно, по-перше, обмежити число розрядів у числі, тому що від цього залежить скільки комірок пам’яті воно буде займати, а по-друге, домовитися про спосіб розміщення десяткової коми в числі, тобто обрати подання у вигляді числа з фіксованою або рухомою комою, оскільки способи обробки цих двох варіантів подання різні. Нарешті, кожний хто мав справу з програмуванням, відзначить, що на першому кроці потрібно довідатися про дві речі: найменше число (щоб записати його в R) і його місце в P, тобто його адреси в тій частині пам'яті, де зберігається P (щоб викреслити його з P), а отже потрібно мати засіб вказання цієї адреси.

Таким чином, навіть у цьому простому прикладі нескладний аналіз показує, що опису, який виглядає цілком зрозумілим, ще далеко до алгоритму. Ми зіткнулися тут з необхідністю уточнити майже всі основні характеристики алгоритму, які відзначали раніше: алфавіт даних і форму їх подання, пам'ять і розміщення в ній елементів P і R, елементарні кроки (оскільки крок 1 явно неелементарний). Крім того, стає зрозумілим, що вибір механізму реалізації (скажімо, для людини або комп’ютера) буде впливати й на сам характер уточнення: у людини вимоги до пам’яті, подання даних і до елементарності кроків набагато узагальненіші і окремі незначні деталі вона може уточнити сама.

Мабуть, тільки дві вимоги до алгоритмів у наведеному вище описі виконані достатньою мірою (вони й створюють враження зрозумілості). Досить очевидна збіжність алгоритму: після виконання кроків 1 і 2 або робота закінчується, або з P викреслюється одне число; тому рівно після n виконань кроків 1 і 2 з P будуть викреслені всі числа й алгоритм зупиниться, а R буде результатом. Крім того, не викликає сумніву детермінованість: після кожного кроку ясно, що робити далі, якщо врахувати, що тут і надалі використовується загальноприйнята умова – якщо крок не містить вказівок про подальший перехід, то виконуємо крок, котрий розташований за ним в описі.

Форми подання алгоритмів

На практиці найбільш поширені такі форми подання алгоритмів:

- **словесна** (записи природною мовою);
- **графічна** (зображення у вигляді графічних символів);
- **у вигляді псевдокоду** (напівформалізовані описи алгоритмів умовною алгоритмічною мовою, що включає як елементи мови програмування, так і фрази природної мови, загальноприйняті математичні позначення тощо);
- **програмна** (тексти мовою програмування).

Словесний опис алгоритму

Розглянемо приклад на алгоритмі знаходження максимального з двох значень:

- визначимо формати змінних *x*, *y*, *m* (*x*, *y* – значення для порівняння, *m* – змінна для зберігання максимального значення);
- отримаємо значення чисел *x* та *y*;
- порівняємо *x* та *y*;
- якщо *x* менше ніж *y*, то більшим є *y;*
- запишемо у змінну *m* значення *y*;
- якщо *x* не менше (більше) *y*, то більшим є *x*.
- запишемо у змінну *m* значення *x*;

Словесний спосіб не отримав широкого розповсюдження, оскільки:

- описи строго не формалізуються;
- має місце багатослівність записів;
- може мати місце неоднозначність тлумачення окремих розпоряджень.

Структурна схема алгоритму

Даний спосіб виявився дуже зручним засобом зображення алгоритмів і набув широке розповсюдження у науковій та навчальній літературі. Структурна схема алгоритму (іноді у літературі зустрічається назва блок-та граф-схема) – графічне зображення алгоритму у вигляді зв'язаних між собою за допомогою стрілок (ліній, переходів) блоків – графічних символів, кожний з яких відповідає одному кроку алгоритму. Усередині блоку дається опис відповідної дії.

Графічне зображення алгоритму широко використовується перед програмуванням задачі внаслідок його наочності, оскільки зорове сприйняття, звичайно, полегшує процес написання програми, її корегування при можливих помилках, осмислення процесу обробки інформації.

Принцип програмування зверху вниз вимагає, щоб структурна схема поетапно конкретизувалася і кожен блок „розписувався" до елементарних операцій. Але такий підхід можна здійснити при розв'язанні нескладних задач.

При розв’язанні будь-якої серйозної задачі структурна схема „розростається” до такого ступеня, що її неможливо буде охопити одним поглядом. Структурні схеми алгоритмів зручно використовувати для пояснення роботи вже готового алгоритму, при цьому як блоки беруться дійсно блоки алгоритму, робота яких не вимагає пояснень. Блок-схема алгоритму повинна служити для спрощення зображення алгоритму, а не для ускладнення.

На структурній схемі добре видно різницю між описом алгоритму і процесом його реалізації. Опис – це граф, а процес реалізації – це шлях у графі. Різні шляхи в тому самому графі виникають при різних даних, які створюють різні логічні умови в точках розгалуження. Відсутність збіжності означає, що в процесі обчислення не виникає умов, що ведуть до кінця, і процес іде по нескінченному шляху (зациклюється).

При всій наочності мови блок-схем не треба переоцінювати їх можливості. Вона досить „груба” і відображає лише зв’язки керування (тобто якому блоку передати керування в наступний момент), а не зв’язки за інформацією (де цьому блоку брати вихідні дані). Тобто структурні схеми не містять відомостей ні про дані, ні про пам’ять, ні про використовуваний набір елементарних кроків. Зокрема, треба мати на увазі: якщо структурна схема не містить циклів, це ще не означає, що алгоритм їх немає, оскільки вони можуть бути у будь-якому неелементарному блоці. По суті структурні схеми – це дуже зручний засіб опису детермінованості алгоритму.

Псевдокод

Псевдокод є системою позначень і правил, призначеною для однозначного запису алгоритмів і займає проміжне місце між природною і формальною мовами. З одного боку, він близький до звичної природної мови, тому алгоритми можуть на ньому записуватися і читатися як звичайний текст. З іншого боку, у псевдокоді використовуються деякі формальні конструкції і математична символіка, що наближає запис алгоритму до загальноприйнятого математичного запису. У псевдокоді не прийняті строгі синтаксичні правила для запису команд, властиві формальним мовам, що полегшує запис алгоритму на стадії його проектування і дає можливість використовувати ширший набір команд, розрахований на абстрактного виконавця. Проте у псевдокоді, зазвичай, є деякі конструкції, властиві формальним мовам, що полегшує перехід від запису на псевдокоді до запису алгоритму формальною мовою. Зокрема, у псевдокоді так само, як і у формальних мовах, є службові слова, значення яких визначено раз і назавжди. Вони виділяються в друкарському тексті жирним шрифтом, а в рукописному тексті підкреслюються.

Єдиного або формального означення псевдокоду не існує, тому можливі різні псевдокоди, які відрізняються набором службових слів і основних (базових) конструкцій. Прикладом псевдокоду є шкільна алгоритмічна мова. Внаслідок вищеперерахованих переваг псевдокоду ми також будемо його застосовувати.

Програмне подання алгоритму

При записі алгоритму у словесній формі, у вигляді структурної схеми або на псевдокоді допускається певна свобода при зображенні команд. Разом з тим такий запис точний настільки, що дозволяє людині зрозуміти суть справи і виконати алгоритм. Проте на практиці як виконавці алгоритмів використовуються комп'ютери. Тому алгоритм, призначений для виконання на комп'ютері, повинен бути записаний „зрозумілою" йому мовою. І тут на перший план висувається необхідність точного запису команд, що не залишає місця для довільного тлумачення їх виконавцем. Отже, мова для запису алгоритмів повинна бути формалізованою. Таку мову прийнято називати мовою програмування, а запис алгоритму цією мовою – комп'ютерною програмою.

1.3 Основні підходи до уточнення поняття "алгоритм"

Вище були сформульовані основні вимоги до алгоритмів. Однак поняття, використані в цих формулюваннях (такі, як ясність, чіткість, елементарність), самі потребують уточнення. Очевидно, що їх словесні означення будуть містити нові поняття, які знову зажадають уточнення, і т. д. Тому в теорії алгоритмів приймається інший підхід: вибирається скінченний набір вихідних об'єктів, які оголошуються елементарними і кінцевий набір способів побудови з них нових об'єктів. Цей підхід був вже використаний під час обговорення питання про дані: уточненням поняття «дані» надалі будемо вважати безліч слів у кінцевих алфавітах. Для уточнення детермінованості будуть використовуватися або блок-схеми та еквівалентні їм словесні описи (типу того, котрий наведений у прикладі 1.1), або опис механізму реалізації алгоритму. Крім того, потрібно зафіксувати набір елементарних кроків і домовитися про організацію пам'яті. Після того як це буде зроблено і вийде конкретна алгоритмічна модель.

Розглянуті алгоритмічні моделі претендують на право вважатися формалізацією поняття „алгоритм". Це означає, що вони повинні бути універсальними, тобто допускати опис будь-яких алгоритмів. Тому може виникнути природне заперечення проти запропонованого підходу: чи не приведе вибір конкретних засобів до втрати спільності формалізації. Якщо враховувати основні цілі, що стояли при створенні теорії алгоритмів – універсальність і пов'язану з ними можливість говорити в рамках будь-якої моделі про властивості алгоритмів взагалі, то це заперечення знімається. По-перше, проводиться зведення одних моделей до інших, тобто показується, що будь-який алгоритм, описаний засобами однієї моделі, може бути описаний і засобами іншої. По-друге, завдяки взаємозведенню моделей у теорії алгоритмів вдалося виробити інваріантну щодо моделей систему понять, що дозволяє говорити про властивості алгоритмів незалежно від того, яка формалізація алгоритму обрана. Ця система понять заснована на понятті обчислюваної функції (функції, для обчислення якої існує алгоритм).

Проте хоча спільність формалізації в конкретній моделі не втрачається, різний вибір вихідних засобів приводить до моделей різного виду. Можна виділити три основних типи універсальних алгоритмічних моделей, що розрізняються вихідними евристичними міркуваннями відносно означення того, що таке алгоритм. Перший тип зв'язує поняття алгоритму з найбільш традиційними поняттями математики – обчисленнями й числовими функціями. Найбільш розвинена та вивчена модель цього типу – ***рекурсивні функції*** – є історично першою формалізацією поняття алгоритму [6, 7]. Другий тип заснований на уявленні про алгоритм як про деякий детермінований пристрій, здатний виконувати в кожний окремий момент лише досить примітивні операції. Таке подання не залишає сумнівів в однозначності алгоритму й елементарності його кроків. Крім того, евристика таких моделей близька до ЕОМ і, отже, до інженерної інтуїції. Основною теоретичною моделлю цього типу (створеною в 30-х роках – раніше ЕОМ) є ***машина Тьюрінга***. Третій тип алгоритмічних моделей – це перетворення слів у довільних алфавітах, у яких елементарними операціями є підстановки, тобто заміни частини слова (підслова) іншим словом. Переваги цього типу – у його максимальній абстрактності та можливості застосувати поняття алгоритму до об'єктів довільної (не обов'язково числової) природи. Втім, як буде ясно з подальшого, моделі другого й третього типу досить близькі (їх взаємозведення доводиться просто) і відрізняються в основному евристичними акцентами. Приклади моделей цього типу – ***канонічні системи Поста*** та ***нормальні алгоритми Маркова*** [6, 7].

1.3.1 Рекурсивні функції

Для вчених, яких цікавлять лише факти існування або неіснування алгоритмів, а не самі алгоритми, зовсім не обов’язково вивчати самі алгоритми. Досить знайти такі об’єкти, які існують або не існують тоді, коли існують або не існують алгоритми. Вважають, що такими об’єктами є рекурсивні функції. Покажемо як можуть функції бути пов’язаними з алгоритмами [7].

Як відомо, величину w називають функцією, а величини x_1, x_2, ..., x_n – аргументами або незалежний змінними, якщо відомий закон, який для різних наборів конкретних значень величин x_1, x_2, ..., x_n задає певні значення величини w. При цьому для кожного набору допускається лише єдине значення функції. В окремому випадку функція може мати один аргумент. Який же закон застосовується для визначення функції? Математика не накладає ніяких обмежень на нього і допускає будь-який мислимий закон. Таким законом може бути і деякий алгоритм. В цьому випадку функцію називають *обчислюваною*, оскільки є спосіб одержання її значень.

Рекурсивними функціями називають один окремий клас обчислюваних функцій. Алгоритми, які задають їх закони називають алгоритмами, що *сприяють рекурсивним функціям*.

Маючи на увазі деяку функцію розрізняють три сторони: 1) ім'я функції, 2) її (функціональний) запис, що має вигляд $w = f(x_1, x_2, ..., x_n)$, 3) її значення. Буква f називається *функціональним знаком*.

Побудова чітко виділеного класу обчислюваних функцій поєднана з рядом проблем; для досягнення такої мети потрібні деякі спрощення. Існує можливість арифметизувати математику, тобто виразити самі різні математичні поняття за допомогою цілих невід'ємних чисел. Тому цілком природно обмежитись випадком, у якому і незалежні змінні, і функції можуть приймати тільки цілі невід'ємні значення.

Сукупність рекурсивних функцій будується так. Безпосередньо опишемо найбільш прості з них і назвемо їх *базовими*. Алгоритми, що супутні цим функціям називають найпростішими або *однокроковими*. Потім опишемо три прийоми, котрі у теорії рекурсивних функцій називають *операторами.* За їх допомогою, виходячи з рекурсивних функцій можна одержати нові функції, які за означенням теж будемо вважати *рекурсивними*. Ці оператори, по суті, будуть алгоритмами. З'єднуючи їх, можна одержувати нові алгоритми. Отже, будуємо клас рекурсивних функцій.

Базові функції бувають трьох типів [7]:

а) функції будь-якого числа незалежних змінних, тотожно рівні нулю. Їх будемо позначати φ_n, де n – число аргументів. Наприклад, до числа таких функцій відносяться $y = \varphi_1(x)$, $w = \varphi_3(x, y, z)$, $w = \varphi_n(x_1, x_2, ..., x_n)$. Алгоритм для цих функцій такий: „якщо функціональний знак має вигляд φ_n, то будь-якій сукупності значень аргументів даної функції ставиться у відповідність значення 0". Наприклад, $\varphi_1(0) = 0;\ \varphi_n(7,8, ..., 125) = 0$.

Будемо вважати, що n може дорівнювати нулю, тобто, що можлива функція цього виду, *не залежна від жодного аргументу*. Ця функція дорівнює 0. Позначимо її $\varphi_0(\ )$ або φ_0;

б) тотожні функції будь-якого числа незалежних змінних. Позначимо їх $\psi_{n,i}$, де n – число аргументів, i – номер одного з аргументів ($1 \le i \le n$). Алгоритм для цих функцій такий: „якщо функціональний знак має вигляд $\psi_{n,i}$, то значенням функцій вважати значення i-го (рахуючи в функціональному записі зліва направо) незалежного змінного". Наприклад, для функцій $w = \psi_{3,2}(x, y, z)$, $\vartheta = \psi_{1,1}(u)$ маємо відповідно $\psi_{3,2}(5,8,0) = 8$, $\psi_{1,1}(6) = 6$. А запис $\psi_{3,4}(x, y, z)$ не має смислу, у ньому n=3, i=4 і, отже, не виконана умова $1 \le i \le n$. Для тотожних функцій ні n, ні i не може дорівнювати нулю;

в) функції слідування (інакше – одержання послідовника) одного незалежного змінного. Позначимо їх за допомогою функціонального знака λ. Наприклад, функціями слідування будуть $\lambda(x)$, $\lambda(w)$. Алгоритм в да-

ному випадку такий: „якщо функціональний знак має вигляд λ, то значенням функції вважати число, що безпосередньо випливає за числом, що є значенням аргументу”.

Відзначимо, що, створюючи рекурсивні функції, опираючись на які ми хочемо обґрунтувати математику, не можна користуватися нашими знаннями навіть арифметичних дій. Операція „одержання послідовника” простіша за додавання. Це цілком природно, тому що у давнину числа зображали як послідовності рисок і послідовник одержували, додаючи ще одну риску. Позначатимемо операцію одержання послідовника за допомогою штриха, наприклад, 0'=1, 1'=2, 10' = 11 і т. д. Отже, $\lambda(x) = x'$; $\lambda(5) = 6$.

Вважається, що базові функції мають значення тільки якщо дано значення їх аргументів.

Оператор суперпозиції або підстановки. Слова суперпозиція й підстановка в цьому випадку мають те саме значення. Цей оператор за функцією F від n аргументів і за функціями $f_1, f_2, ..., f_n$ будує нову функцію Φ для якої справедлива тотожність $\Phi \equiv F(f_1, f_2, ..., f_n)$.

Алгоритм такий: „значення функцій $f_1, f_2, ..., f_n$ прийняти за значення аргументів функції F і обчислити її значення”.

Наприклад, якщо за F прийняти $\lambda(x)$, а як f_1 взяти функцію $\lambda(y)$, то за допомогою операції суперпозиції одержимо функцію $\tau(y) = \lambda(\lambda(y)) = \lambda(y') = y''$. Зокрема, $\tau(5) = 7$.

Введемо знак „::=”, що читається „за визначенням є”. Оператор підстановки (суперпозиції) позначимо буквою S. Побудову функції Φ з функцій F і f_i; за допомогою оператора суперпозиції будемо записувати так:

$$\Phi ::= S[F; f_1, f_2, f_n].$$

Наприклад, для нашої функції $\tau(y)$ можна написати $\tau(y) ::= S[\lambda(x; \lambda(y))]$.

Оператор рекурсії. Цей оператор за двома функціями, одна з яких має $n-1$ незалежних змінних, а інша, крім зазначених незалежних змінних, має ще дві (тобто залежить від $n+1$ змінних), будує функцію n аргументів. Один з додаткових аргументів увійде разом з аргументами першої функції в число аргументів знову одержуваної функції, а інший відіграє допоміжну роль при виконанні оператора рекурсії. Оператор рекурсії будемо позначати буквою R; його застосування будемо записувати у вигляді рядка:

$$f ::= R[f_1, f_2; x, (y)],$$

де f – означає одержувану функцію, f_1 – функцію $n-1$ незалежних змінних, f_2 – функцію $n+1$ незалежних змінних, x – головний з додаткових аргументів, y – допоміжний аргумент.

Під час опису суті оператора рекурсії зручно не вказувати аргументів першої із заданих функцій ні в її функціональному записі, ні у записах двох інших функцій, маючи на увазі ці аргументи. Тоді можна сказати, що оператор рекурсії задає функцію за допомогою двох умов, в які входять функції f_1 та f_2:

$$f(0) = f_1, \quad f(i') = f_2(i, f(i)).$$

Алгоритм в цьому випадку такий: „значенням одержуваної функції для нульового значення головного додаткового аргументу вважати значення вихідної функції $n-1$ аргументів; значенням обумовленої функції для кожного наступного значення головного аргументу вважати значення другої із заданих функцій при попередньому значенні головного аргументу і при значенні допоміжного аргументу, що збігається з попереднім значенням визначеної функції".

Для прикладу розглянемо випадок, де як f_1 взята функція φ_0, а як f_2 – функція $\psi_{2,1}(x,y) = x$. Визначену функцію позначимо pp(x). Для неї маємо

$$pp(x)::=R[\varphi_0, \psi_{2,1}(x,y); x, (y)].$$

Ця функція визначається умовами $pp(0)=\varphi_0$, $pp(i')=\psi_{2,1}(i, pp(i))$, інакше кажучи, $pp(0)=0$, $pp(i')=i$.

Отримана функція називається функцією отримання попередника; її значеннями є $pp(0)=0$, $pp(1)=0$, $pp(2)=1$, $pp(3)=2$ і т. д. Оскільки рекурсивні функції можуть приймати тільки цілі невід'ємні значення, то попередником числа 0 ми вважаємо не -1, а 0.

Оператор побудови за першим нулем (його, зазвичай, називають оператором *найменшого числа*, а в деяких джерелах – *оператором мінімізації*). Цей оператор залежить від $n+1$ аргументів і будує нову функцію від n аргументів. Зникаючий аргумент є допоміжним і використовується при виконанні оператора. Позначають оператор побудови за першим нулем буквою μ. Його застосування позначають рядком вигляду

$$f ::= \mu[f_1; (x)],$$

де x – зникаючий аргумент. За допомогою оператора побудови за першим нулем одержують функцію f, значення якої визначаються при виконанні супутнього їй алгоритму, де вказується: „надавати допоміжному аргументу послідовні значення, починаючи з 0 доти, поки не виявиться, що функція f_1 стала (перший раз) дорівнювати нулю. Отримане значення допоміжного аргументу прийняти за значення визначеної функції, що відповідає тим значенням основних аргументів, при яких здійснювався описаний процес".

Зазначимо, що базові функції та функції, одержувані за допомогою операторів підстановки й рекурсії, але без застосування оператора побудо-

ви за першим нулем, визначені для будь-яких значень аргументів. Інакше – з функціями, отриманими за допомогою оператора побудови за першим нулем. Для деяких комбінацій значень аргументів (навіть для дуже багатьох) вони можуть бути не визначені, тому що вихідна функція не приймає нульових значень. Для прикладу як f_1 візьмемо базову функцію $\psi_{2,1}(x,y)$. Нехай $\rho(x) ::= \mu\left[\psi_{2,1}(x,y);\ (y)\right]$. Отримана функція $\rho(x)$ має такі властивості: $\rho(0)=0,\ \ \rho(i)$, *p(t)* не існує при $i \neq 0$; перевіримо це для випадку $i=2$. Отримаємо $\psi_{2,1}(2,0)=2;\ \ \psi_{2,1}(2,1)=2;\ \psi_{2,1}(2,2)=2$ і т. д. Зрозуміло, що оператор побудови за першим нулем не дозволить нам одержати $\rho(2)$, тому що функція $\psi_{2,1}(x,y)$ при $x=2$ ні для якого значення y не буде дорівнювати нулю.

Отже, *рекурсивними* називаються базові функції та будь-які функції, отримані з них за допомогою кінцевого числа операторів підстановки, рекурсії й побудови за першим нулем. При цьому функції, одержувані без використання оператора побудови за першим нулем, називаються *примітивно-рекурсивними*. Це найпростіші з рекурсивних функцій. Примітивно-рекурсивні функції разом з тими рекурсивними функціями, які неможливо отримати без застосування оператора побудови за першим нулем, але які визначені для будь-яких значень аргументів, називають *загальнорекурсивними*. Виявляється, що існують загальнорекурсивні функції, які не примітивно-рекурсивні. Рекурсивні функції, які визначені не для всіх можливих значень аргументів, називають *частково-рекурсивними*. Побудована нами вище функція $\rho(x)$ є частково-рекурсивною.

Тепер, коли клас рекурсивних функцій побудований, можна згадати всю математику, яку ми на деякий час забули, і подивитися, чи немає серед добре знайомих нам функцій рекурсивних. Виявляється є – і чимало. Наприклад, функція $y=x+1$ збігається з базовою функцією $\lambda(x)$ і, отже, рекурсивна.

За допомогою оператора підстановки функції $z+1$ замість z у функцію $\psi_{3,3}(x,y,z)$ ми можемо одержати функцію трьох змінних $f*(x,y,z)=z+1$, а за допомогою цієї функції і базової функції $\psi_{1,1}(x)$ визначити нову функцію *s*(*x, y*) шляхом рекурсії:

$$s(x,y) ::= R\left[\psi_{1,1}(x),\ f*(x,y,z);\ y,(z)\right].$$

Ця функція задовольняє умови

$$s(x,0)=\psi_{1,1}(x)=x,\ s(x,1)=s(x,0)+1=x+1;$$

$$s(x,2)=s(x,1)+1=x+2,\ \ldots,\ s(x,y)=x+y,$$

тобто є функцією *w=x+y*. Отже, і ця функція рекурсивна. Легко показати, що й функція *w=x-y* рекурсивна і т. д. Виявляється, що багато простіші відомі нам функції рекурсивні.

Американський математик А. Черч висловив думку про те, що поняттям рекурсивної функції вичерпується поняття обчислюваної функції. Ця думка є гіпотезою, котра підтверджується усім попереднім досвідом математиків. Вона відома за назвою тези Черча.

Теза Черча. Яка б не була обчислювана невід'ємна цілочислова функція від невід'ємних цілочислових аргументів, існує тотожно рівна їй рекурсивна функція.

Переносячи тезу Черча на алгоритми, що є супутніми рекурсивним функціям, ми можемо висловити для них таку гіпотезу [7].

Основна теза. Який би не був алгоритм, що перетворює набори цілих невід'ємних чисел у цілі невід'ємні числа, існує еквівалентний йому алгоритм супутній рекурсивній функції.

Вчені А. Н. Колмогоров і В. А. Успенський узагальнили цю гіпотезу. Припустимо, що нам задано деякий алгоритм в інтуїтивному смислі. На практиці завжди вдається знайти спосіб нумерації його припустимих вихідних даних, а також інший спосіб нумерації отримуваних ним результатів. Відповідність номерів вихідних даних номерам результатів, установлювана нашим алгоритмом, завжди збігається з відповідністю між значеннями аргументу й значеннями деякої рекурсивної функції від однієї незалежної змінної. Це означає, що виконання алгоритму еквівалентне обчисленню значення рекурсивної функції. А це означає, що неможливість рекурсивної функції означає й неможливість алгоритму (тому що для кожного алгоритму повинна бути рекурсивна функція, а її в даному випадку немає).

Зазначимо, що теза Черча та інші гіпотези, які з неї випливають не мають доказу і принципово не можуть бути доведені, оскільки в них мова йде про алгоритми в інтуїтивному смислі, що не є математичними об'єктами [7].

1.3.2 Машина Тьюрінга

Англійський математик Алан Тьюрінг у 1937 р. опублікував роботу, в якій уточнив поняття алгоритму, вдаючись до уявної обчислювальної машини, відомої тепер під назвою *машини Тьюрінга.* Ідея А. Тьюрінга виникла ще до появи електронних обчислювальних машин і тому, очевидно, зовсім не залежить від них [6].

А. Тьюрінг вирішив, що особливої уваги заслуговує поведінка виконавця алгоритму, тому її слід точно і ясно описати. При цьому виконавець повинен діяти повністю механічно, не вкладаючи у свою роботу ніякої ініціативи. Цілком зрозуміло, що така властивість може бути притаманна лише машині. Але ресурси реальної машини обмежені, а при виконанні алгоритмів ми їх вважаємо необмеженими. Тому машина повинна бути уявною. При бажанні можна буде створити реальну модель машини Тьюрінга, але ця модель вже буде мати обмежені ресурси.

Отже, машина повинна перетворити деякі об'єкти в шукані результати. Найпростішими об'єктами є рядки символів. Символи, з яких створюють рядки, називають у теорії алгоритмів буквами. Значення цього слова відрізняється від загальноприйнятого. Ми знаємо, що тільки деякі символи називають буквами. Саме символи, що служать для запису слів природної мови. Крім таких символів, відомі ще символи, що застосовують для запису чисел, їх називають цифрами. Нарешті, ще відомо багато знаків, котрі не називають ні буквами, ні цифрами. Такими є розділові знаки, знаки математичних дій тощо. У теорії алгоритмів будь-які знаки, що не діляться і не змінюють свого вигляду, називають *буквами.*

Рядки букв у теорії алгоритмів називають *словами.* Цей термін за своїм змістом теж відмінний від загальноприйнятого. Наприклад, рядок букв „аАлбмв" не вважають словом, проте у теорії алгоритмів – це слово. Аналогічно, словом є і набір символів „6;+ М-!?".

У житті прийнято вважати словами рядки букв, що є словами мови, зокрема, наділені змістом. У теорії алгоритмів поняття слова не пов'язують з наявністю в ньому змісту. Тут термін „слово" означає лише структуру об'єкта, побудованого із символів-букв – це рядок букв. Чим же обумовлений вибір слів в якості об'єктів для застосування до них машин? Для їх запису можна обмежитися найпростішим технічним пристроєм – стрічкою. Будемо вважати, що стрічка є нескінченною в обидва боки і розбита на однакові клітинки. У кожній такій клітинці може бути записана лише одна буква. Для зручності можна вважати, що пустота клітинки означає наявність в ній «пустої букви». Позначимо порожню букву значком a_0. Крім неї для даної машини повинні бути допустимі ще якісь (не порожні) букви. Припустимо, що кількість таких букв n. Позначимо їх $a_0, a_1, a_2, ..., a_n$. Отже, маємо алфавіт $a_0, a_1\ a_2, ..., a_n$, який називають *зовнішнім алфавітом* машини Тьюрінга.

Для того щоб пересувати стрічку, машина Тьюрінга (МТ) має простий стрічкопротяжний механізм, який може бути або нерухомим, або пересувати стрічку на одну клітинку вправо або вліво.

Для того щоб читати написане на стрічці або писати на ній, МТ має спеціальну *головку зчитування-записування*. При цьому вважають, що головка може за один прийом прочитати лише одну букву, що розташована під нею. За один крок головка може записувати тільки одну букву в тій клітинці стрічки, що стоїть під головкою.

Нарешті, основною частиною машини є *логічний блок* (ЛБ) на якому знаходиться кнопка, при натисканні котрої він приймає вихідний (для роботи) стан і починає працювати. Кнопка при цьому залишається натиснеою доти, поки машина не зупиниться сама. Тоді вона "вискакує", і машину можна використовувати для нової роботи. Одночасно з "вискакуванням" кнопки машина подає досить голосний і не дуже тривалий сигнал.

Працює ЛБ так.

1. Він змушує головку прочитати букву, що розташована під нею на стрічці.

2. В залежності від прочитаної букви і стану, в якому знаходиться ЛБ: а) змушує головку записати на стрічці в тій клітинці, що знаходиться під головкою, деяку букву; б) змушує стрічкопротяжний механізм пересунути стрічку вправо, вліво або залишитися на місці; в) змінює свій власний стан.

3. Якщо в результаті дій, зазначених у п. 2, буква, розташована під головкою, положення стрічки і стан ЛБ машини виявляться такими ж, якими були безпосередньо перед виконанням цих дій – машина зупиняється. Інакше ЛБ повертається до виконання п. 1.

ЛБ здатний знаходитися в кожен момент в одному з визначених станів. Позначимо його можливі стани символами $p_1, p_2, ..., p_m$. Послідовність цих символів називають *внутрішнім алфавітом* (станів) машини.

Введемо ще позначення d_{-1}, d_0, d_1 відповідно для позначення руху стрічки вліво, знаходження стрічки на місці і руху стрічки вправо. Тепер можна описати поведінку машини при виконанні логічним блоком п. 2 так: якщо головка читає букву a_i і блок знаходиться в стані p_j, то функціональна дія машини визначає запис $a_x d_y p_z$, що означає: „записати на стрічці замість a_i букву a_x, зробити переміщення стрічки d_y, логічному блоку перейти у стан p_z".

Очевидно, різні МТ відрізняються між собою реакціями на сполучення a_i, p_j для різних i та j. Найкомпактніше можна подати усі випадки сполучень a_i, p_j і усі види реакцій машини на них, якщо скласти так звану *функціональну таблицю* (таблиця 1) з двома входами, у клітинках якої стоїть трійка символів виду $a_x d_y p_z$

Таблиця 1.1 – Функціональна таблиця

	p_1	p_2	...	p_j	...	p_m
a_0						
a_1						
⋮						
a_i				$a_x d_y p_z$		
⋮						
a_n						

Отже, в МТ можна розрізнити дві частини: постійну – яка однакова для всіх машин (будова машини), і змінну, що відрізняє одну машину від іншої (внутрішній і зовнішній алфавіти і функціональна таблиця).

Якщо записати на стрічці МТ деяке слово в алфавіті A (зовнішньому), складене з букв $a_1, a_2, ..., a_n$, і встановити початкову букву цього слова під головкою зчитування, або залишити стрічку порожньою (тобто встановити порожнє слово), а потім натиснути пускову кнопку – машина почне працювати. При цьому можливі два випадки: 1) робота МТ після кінцевого числа кроків припиниться з видачею сигналу; 2) МТ ніколи не зупиниться і не дасть ніякого результату. У першому випадку говорять, що машина *застосовна* до вихідних даних, у другому випадку – що *незастосовна.*

Якщо машина застосовна до вихідних даних, то вона приводить до *результату:* а) порожнього слова, якщо під головкою знаходиться буква a_0; б) слова з букв $a_1, a_2, ..., a_n$, обмеженому з двох сторін порожніми буквами, якщо одна з букв (непорожня) цього слова знаходиться під головкою.

Відповідність, що встановлюється МТ між тими вихідними даними, до яких вона застосовна, і результатами її роботи є деякою функцією (у математичному смислі).

Якщо для функції *f(x)* є машина, що її реалізує, то говорять, що *f(x) обчислювана за Тьюрінгом.* Функцію, для обчислення якої існує алгоритм, називають *обчислюваною.* Тьюрінг висловив припущення (ми будемо його називати *основною тезою Тьюрінга),* що будь-яка обчислювана функція є обчислюваною за Тьюрінгом. Іншими словами, для будь-якої обчислюваної функції можна побудувати МТ, що її реалізує.

Для прикладу опишемо МТ, що обчислює функцію $\lambda(x)=x+1$ (це одна з базисних рекурсивних функцій). Дана функція має визначені значення тільки для цілих додатних значень *x.* Домовимося значення *x* записувати на стрічці машини у вигляді рядка, що складається з букв | (паличок). Крім цих букв, нам потрібна ще одна (порожня) буква. Позначимо її знаком □ (на стрічці вона буде зображуватися порожньою клітинкою). Число нуль будемо зображувати „порожнім" рядком паличок.

Записавши значення *x* (якщо воно не є нулем) у вигляді рядка паличок на стрічці машини і розташувавши стрічку так, щоб крайня ліва паличка була під головкою зчитування-записування, запустимо МТ.

ЛБ у початковий момент прийме стан p_1. Потрібно щоб він, якщо головка зчитує букву |, змусив її знову записати цю ж букву, стрічку – пересунутись вліво, а сам знову перейшов в той же стан (тобто зберіг його). Якщо ж головка зчитає букву □, а ЛБ при цьому буде знаходитися в стані p_1, то він повинен змусити головку написати |, стрічку – залишитись на місці, а сам – перейде у стан p_2.

Знаходячись у цьому стані, він при зчитуванні головкою букви | повинен змушувати її знову записати букву |, стрічку залишити на місці, а самому – зберігати свій стан p_2. При цьому МТ зупиниться і подасть сигнал. Яке б число ми не записали на стрічці, машина, рухаючи стрічку ліворуч, перегляне всі палички, потім просуне стрічку ще на один крок, припише до числа ще одну паличку і зупиниться. Функціональна таблиця МТ, що обчислює значення функції $\lambda(x)$, подана у табл. 1.2.

Таблиця 1.2

	p_1	p_2
□	$\|d_0 p_2$	
|	$\|d_{-1} p_1$	$\|d_0 p_2$

У порожній клітинці таблиці можна записати що завгодно, тому що її вміст не впливає на роботу МТ. З цієї таблиці видно, що машина буде давати правильний результат і при $x=0$. При цьому під головкою МТ потрібно встановити клітинку з буквою ڤ (а на стрічці не повинно бути жодної букви |). Після запуску МТ вона запише замість цієї букви букву | і зупиниться.

Машини Тьюрінга не тільки відповідають деяким алгоритмам спеціального виду, але і самі є ними. МТ – це тексти на певній мові, а зовсім не машини в звичайному розумінні слова. Основною частиною МТ, що визначає її дії виконавця є функціональна таблиця, оскільки саме вона визначає конкретний процес перетворення вихідних даних. Таким чином, МТ – це алгоритм, записом якого є функціональна таблиця, а правилом (алгоритмом) виконання – опис її дії.

1.3.3 Нормальні алгоритми Маркова

Радянський вчений А. А. Марков обрав інший шлях уточнення поняття алгоритму, розробивши строгу теорію класу алгоритмів, яку він назвав *нормальними алгоритмами* [7]. Вони, аналогічно машинам Тьюрінга, як вихідні дані і шукані результати також застосовують слова.

Припустимо, що заздалегідь визначено деякий алфавіт *А.* Букву, що входить в алфавіт *А,* називають *буквою в А.* Слово, що складається з букв в *А,* називають *словом в А.* При цьому для зручності міркувань допускають і порожні слова (які не мають у своєму складі жодної букви).

Якщо *А* і *В* – два алфавіти, причому кожна буква алфавіту *А* є буквою в *В,* але хоча б одна з букв алфавіту *В* не є буквою в *А,* то *В* називається *розширенням алфавіту А.*

Наприклад, якщо А ={а, б, у, м}, В={1, а, б, у, м, д}, то *В* є розширенням *А,* тому що містить дві букви («1» і «д»), що не є буквами в *А,* тоді як усі букви алфавіту *А* є буквами у *В.*

Розглянемо будь-яке конкретне слово для визначеності в алфавіті російських букв, наприклад, слово „студент”. Ми бачимо, що з нього можна виділити *підслова,* наприклад „студ”, „дент”, „уде”, „т” і т. д. Про такі підслова говорять, що вони *входять* у дане слово або є *входженнями* в нього. Відзначимо, що в наше слово входить і порожнє слово, причому кілька разів (перед першою буквою, між кожними двома буквами і, нарешті, після останньої букви).

Домовимося позначати слова заголовними латинськими буквами (якщо ці букви не є буквами в застосовуваному алфавіті). Якщо задано деяке слово і нами обрана буква, що є його позначенням (ім’ям), то будемо ставити між ними знак = (рівності). Повертаючись до нашого прикладу, ми можемо написати: для слова *R*=студент слово *P*=уде є входженням.

Зазначимо, що не тільки порожнє слово може багаторазово входити в інше слово. Наприклад, у слово *R*=тарарам слово *P*=ара входить двічі. Особливий інтерес для нас буде становити саме перше входження.

Марковською підстановкою називається операція над словами, що задається за допомогою пари слів *(P, Q)*. Так, якщо задано вихідне слово *R*, то в ньому знаходять перше входження слова *P* (якщо таке є) і, не змінюючи інших частин слова *R,* заміняють у ньому це входження словом *Q*. Отримане слово є результатом застосування марковської підстановки

(*P*, *Q*) до слова *R*. Якщо ж немає входження *P* у слово *R,* то вважається, що слово *R* не піддається марковській підстановці.

Окремими випадками марківських підстановок є (,*Q*), (*P*,) і (,). У першому з приведених прикладів *P,* у другому *Q,* а в третьому і *P*, і *Q* є порожніми. Приведемо деякі приклади перетворення слів за допомогою марковських підстановок (табл. 1.3).

Таблиця 1.3 – Перетворення слів за допомогою марковських підстановок

Перетворюване слово	Марковська підстановка	Результат
192375923	(923, 0000)	1000075923
функція	(,λ-)	λ -функція
паровоз	(овоз,)	пар
терек	(е,)	трек
слово	(,)	слово
слово	(ра, да)	(результату немає)

Будемо розглядати слова в деякому алфавіті *A*. Припустимо, що символи „→” і „→·” не є буквами в *A*. Записи

$$P \to Q \text{ і } P \to \cdot Q$$

будемо називати *записами* марковської підстановки (*P*, *Q*), причому першу з них будемо називати *підстановкою,* а другу – *заключною підстановкою.* Зміст цих назв стане ясним нижче.

Підстановки і заключні підстановки будемо називати *формулами,* розрізняючи в них ліву частину *P* и праву частину *Q*.

Записом нормального алгоритму в алфавіті *A* називають стовпець формул, ліві і праві частини яких є словами в *A*. Виконання нормального алгоритму стосовно вихідного даного *R,* що є словом в *A,* полягає в такому. Рухаючись по стовпцю формул, шукають першу формулу, ліва частина якої входить у перетворюване слово. Якщо такої формули не знайдено – процес закінчено, якщо ж вона є – виконують відповідну марковську підстановку. Якщо дана підстановка є заключною – процес закінчений, інакше процес повторюють із самого початку.

Нехай *B* є розширенням алфавіту *A*. Нормальний алгоритм у *B,* який слова в *A,*(якщо він до них застосовується), перетворює у результати, що є словами в *A,* називається *нормальним алгоритмом над A*. В деяких випадках побудова нормального алгоритма над *A* набагато легша, ніж побудова нормального алгоритму в *A* [7].

Приведемо приклади деяких нормальних алгоритмів. Ми вже показували як можна для $\lambda(x)=x+1$ побудувати машину Тьюрінга. Припустивши, що алфавіт *A* складається з єдиної букви |, шуканий нормальний алгоритм

набуде вигляду $\rightarrow\cdot|\cdot$

У випадку, коли *x* записано в звичайній десятковій системі числення нормальний алгоритм для обчислення $\lambda(x)$ дещо складніший. Як алфавіт *A* візьмемо перелік арабських цифр 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Нормальний алгоритм будемо будувати не в *A*, а над *A,* додавши до перерахованих букв ще дві букви *x* і *y.* Заради економії місця стовпець формул нашого алгоритму запишемо у вигляді чотирьох підстовбців.

$0y\rightarrow\cdot 1$	$8y\rightarrow\cdot 9$	$x5\rightarrow 5x$	$3x\rightarrow 3y$
$1y\rightarrow\cdot 2$	$9y\rightarrow y0$	$x6\rightarrow 6x$	$4x\rightarrow 4y$
$2y\rightarrow\cdot 3$	$y\rightarrow\cdot 1$	$x7\rightarrow 7x$	$5x\rightarrow 5y$
$3y\rightarrow\cdot 4$	$x0\rightarrow 0x$	$x8\rightarrow 8x$	$6x\rightarrow 6y$
$4y\rightarrow\cdot 5$	$x1\rightarrow 1x$	$x9\rightarrow 9x$	$7x\rightarrow 7y$
$5y\rightarrow\cdot 6$	$x2\rightarrow 2x$	$0x\rightarrow 0y$	$8x\rightarrow 8y$
$6y\rightarrow\cdot 7$	$x3\rightarrow 3x$	$1x\rightarrow 1y$	$9x\rightarrow 9y$
$7y\rightarrow\cdot 8$	$x4\rightarrow 4x$	$2x\rightarrow 2y$	$\rightarrow x$

Якби ми цей алгоритм застосували до вихідного даного *R* (порожнє слово), то одержали б нескінченний процес, проміжними результатами якого були б слова *x, xx, xxx* тощо. Це означає, що до порожнього слова наш алгоритм не застосовується. Його застосування до слова *R=299* дало б проміжні результати *x299* (див. останній рядок четвертого підстовпця) *2x99, 29x9, 299x (*див. рядки, що розташовані наприкінці другого і початку третього підстовпців), *299y* (див. передостанню формулу), *29y0, 2y00* (в результаті дворазового застосування другої формули другого підстовпця), і привело б до шуканого результату *300* (внаслідок застосування третьої формули алгоритму). Ми одержали б $\lambda(299)=300$, що і потрібно.

Розглядаючи нормальні алгоритми, можна побачити, що вони цілком відповідають нашому розумінню алгоритму, але мають певні особливості. Більшість алгоритмів, що зустрічаються на практиці, не є нормальними алгоритмами. Але, безумовно, нормальні алгоритми описані з повною математичною строгістю і точністю. Вони цілком придатні для тих цілей, для яких були розроблені, а саме – для цілей обґрунтування математики і дослідження нерозв’язуваності задач.

Інше питання, як це робити. Для них, як і для розглянутих нами інших сімейств обраних алгоритмів, висувається гіпотеза (яку не можна довести, але можна прийняти, опираючись на весь попередній досвід людства), відома за назвою принципу нормалізації.

Принцип нормалізації. Який би не був алгоритм, для якого припустимими вихідними даними і відповідними їм результатами є слова в *A,* існує еквівалентний йому нормальний алгоритм над *A*.

1.5 Контрольні питання

1. Що вивчає дисципліна „Теорія алгоритмів”?
2. Як і чому виникла теорія алгоритмів?
3. Наведіть походження та визначення терміна „алгоритм”.
4. Перерахуйте та поясніть властивості алгоритмів. Наведіть відповідні приклади.
5. Наведіть, охарактеризуйте та дайте порівняльну оцінку основним формам подання алгоритмів. Яка з цих форм Вам подобається найбільше і чому?
6. Поясніть, чому виникла потреба в уточненні поняття „алгоритм”.
7. Перерахуйте та стисло охарактеризуйте відомі Вам підходи до уточнення поняття алгоритму.
8. Які функції називають рекурсивними і з якою метою вони застосовуються?
9. Наведіть та охарактеризуйте базові функції. Наведіть відповідні приклади.
10. В чому сутність оператора суперпозиції або підстановки? Наведіть приклад оператора суперпозиції.
11. Дайте означення та наведіть приклад оператора рекурсії.
12. Охарактеризуйте та наведіть приклад оператора побудови за першим нулем.
13. Сформулюйте основну тезу А. Черча. В чому полягає значення цієї тези для теорії алгоритмів?
14. Наведіть означення та будову машини Тьюрінга (МТ).
15. Наведіть правила функціонування МТ.
16. Побудуйте МТ для будь-якого Вашого алгоритму.
17. З якою метою застосовують МТ?
18. Сформулюйте основну тезу А. Тьюрінга.
19. Які алгоритми називають нормальні алгоритми Маркова?
20. Наведіть власний приклад нормального алгоритму Маркова.

2 ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ АЛГОРИТМІВ

2.1 Критерії оцінювання складності алгоритмів

Для оцінювання алгоритмів існує багато критеріїв. Найчастіше нас буде цікавити порядок зростання необхідних для розв'язання задачі часу і ємності пам'яті при збільшенні вихідних даних. Потрібно зв'язати з кожним конкретним завданням деяке число, яке називається його розміром, і вказує на міру кількості вихідних даних. Наприклад, розміром задачі множення матриць може бути найбільший розмір матриць-співмножників. Розміром задачі про графи може бути кількість ребер даного графа тощо.

Час *T*, затрачений алгоритмом, як функція розміру задачі *f(n)*, називається *часовою складністю* цього алгоритму. Поведінка цієї складності в межах при збільшенні розміру задачі називається *асимптотичною часовою складністю*. Аналогічно можна визначити *ємнісну складність* і *асимптотичну ємнісну складність* [1]. Саме асимптотична складність алгоритму визначає розмір задач, які можна розв'язати за допомогою даного алгоритму.

Нижче ми розглянемо асимптотичні позначення, які застосовують для визначенні складності алгоритмів. Але спершу ще раз торкнемося питання, важливості розробки ефективних (з точки зору часу виконання) алгоритмів.

2.2 Про корисність швидких алгоритмів

В наш час – час стрімкого прогресу в галузі обчислювальної техніки може виникнути думка, що значне зростання швидкості обчислень, викликане появою нинішнього покоління комп'ютерів, знизить значення ефективних алгоритмів. Проте все навпаки. Оскільки комп'ютери працюють все швидше – то можна розв'язувати задачі все більшого розміру. Саме складність алгоритму і визначає те збільшення розміру задачі, яке можна досягти зі збільшенням швидкості комп'ютера.

Припустимо, що є п'ять алгоритмів A_1 – A_5 з деякими часовими складностями (див. табл. 2.1). Тут часова складність – це число одиниць часу, необхідного для оброблення входу розміром *n*. Нехай одиницею часу буде одна мілісекунда. Тоді алгоритм A_1 може обробити за одну секунду вхід розміром 1000, в той час як A_5 – вхід розміром не більше 9. У табл. 2.2 наведені розміри задач, які можна розв'язати за одну секунду, одну хвилину та одну годину за допомогою кожного з цих п'яти алгоритмів.

Таблиця 2.1

Алгоритм	Часова складність
A_1	n
A_2	$n \log n^1$
A_3	n^2
A_4	n^3
A_5	2^n

Таблиця 2.2 – Границі розмірів завдань, обумовлені швидкістю зростання складності

Алгоритм	Часова складність	Максимальний розмір входу задачі		
		1 секунда	1 хвилина	1 година
A_1	N	1000	$6 \cdot 10^4$	$3{,}6 \cdot 10^6$
A_2	$n \log n$	140	4893	$2 \cdot 10^5$
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Припустимо, що наступне покоління комп'ютерів буде у 10 разів швидше нинішнього. У табл. 2.3 показано як зростуть розміри задач, які можна розв'язати завдяки такому збільшенню швидкості. Зазначимо, що для алгоритма A_5 десятикратне збільшення швидкості збільшує розмір задачі, яку можна розв'язати, тільки на три, тоді як для алгоритму A_3 розмір задачі більш ніж потроюється.

Таблиця 2.3 – Ефект десятикратного прискорення

Алгоритм	Часова складність	Максимальний розмір задачі	
		до прискорення	після прискорення
A_1	n	S_1	$10\ S_1$
A_2	$n \log n$	S_2	$\approx 10\ S_2$ *для більших* S_2
A_3	n^2	S_3	$3{,}16\ S_3$
A_4	n^3	S_4	$2{,}15\ S_4$
A_5	2^n	S_5	$S_5+3{,}3$

Замість ефекту збільшення швидкості розглянемо тепер ефект застосування більш діючого алгоритму. Повернемося до таблиці 2.1. Якщо за основу для порівняння взяти 1 хвилину, то, замінюючи алгоритм A_4 алгоритмом A_3, можна розв'язати задачу, в 6 разів більшу, а замінюючи A_4 на A_2 – більшу в 125 разів. Ці результати набагато більше вражають, ніж дворазове поліпшення, що досягається за рахунок десятикратного збільшення швидкості. Якщо як основу для порівняння взяти 1 годину, то різниця буде ще більшою. Звідси маємо, що асимптотична складність алгоритму служить важливою мірою якості алгоритму, причому такою мірою, що обіцяє стати ще важливішою при наступному збільшенні швидкості обчислень.

Незважаючи на те, що основна увага тут приділяється порядку зростання величин, слід розуміти, що великий порядок зростання складності алгоритму може мати меншу мультиплікативну сталу, ніж малий порядок зростання складності іншого алгоритму. В такому випадку алгоритм із швидко зростаючою складністю може бути ефективнішим для задач з малим розміром – можливо, навіть для всіх задач, які нас цікавлять. Наприклад, припустимо, що часові складності алгоритмів A_1, A_2, A_3, A_4 і A_5

дорівнюють відповідно 1000n, l00$n \cdot log\ n$, 10n^2, n^3 і 2^n. Тоді A_5 буде найкращим для задач розміром $2 \le n \le 9$, A_3 – для задач розміром $10 \le n \le 58$, A_2 – при $59 \le n \le 1024$, а A_1 – при $n > 1024$.

2.3 Швидкість зростання функцій. Асимптотичні позначення

Аналізуючи алгоритми, можна намагатися знайти точну кількість виконуваних ним дій. Але у більшості випадків досить оцінити асимптотику зростання часу роботи алгоритму при наближенні розміру входу до нескінченності. Якщо в одного алгоритму асимптотика зростання менша, ніж в іншого, то у більшості випадків він буде ефективнішим для усіх входів, крім входів з малим розміром (хоча бувають і винятки).

Хоча в багатьох випадках ці позначення використовуються неформально, корисно почати з точних визначень [1].

Θ-позначення

Нехай час $T(n)$ роботи деякого алгоритму на входах довжини n є $\Theta(n^2)$. Точний зміст цього твердження такий: знайдуться такі константи $c_1, c_2 > 0$ і таке число n_0, що $c_1 n^2 \le T(n) \le c_2 n^2$ для всіх $n \ge n_0$. Взагалі, якщо $g(n)$ – деяка функція, то запис $f(n) = \Theta(g(n))$ означає, що знайдуться такі $c_1, c_2 > 0$ і таке n_0, що $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всіх $n \ge n_0$ (див. рис. 2.1.а).

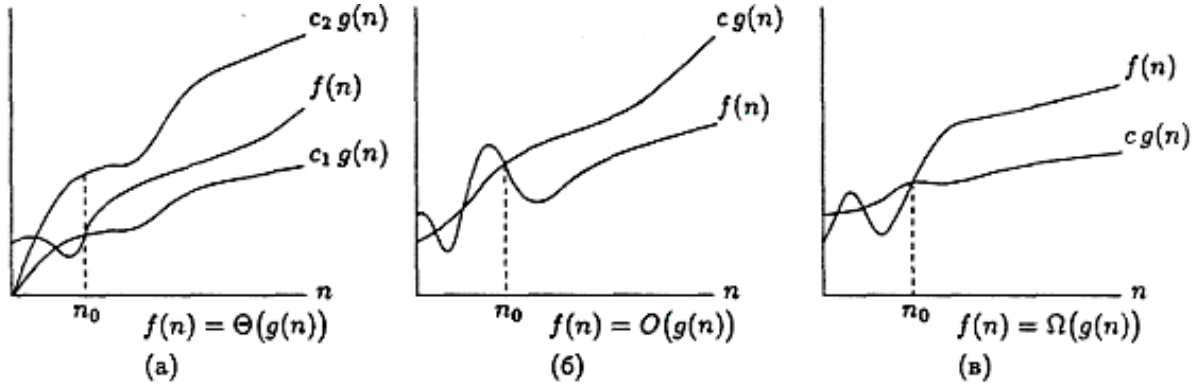


Рисунок 2.1

Слід пам’ятати що це позначення варто вживати з обережністю: встановивши, що $f_1(n) = \Theta(g(n))$ і $f_2(n) = \Theta(g(n))$, не слід робити висновок, що $f_1(n) = f_2(n)$.

Визначення $\Theta(g(n))$ припускає, що функції $f(n)$ і $g(n)$ асимптотично невід’ємні, тобто невід’ємні для досить великих значень n. Якщо функції f і g строго додатні, то можна виключити n_0 з визначення (змінивши константи c_1 і c_2 так, щоб для малих n нерівність також виконувалась).

Якщо $f(n) = \Theta(g(n))$, то кажуть, що $g(n)$ є асимптотично точною оцінкою для $f(n)$. Насправді це відношення симетричне: якщо $f(n) = \Theta(g(n))$, то $g(n) = \Theta(f(n))$.

Покажемо для прикладу, що $(1/2)n^2-3n=\Theta(n^2)$. Відповідно до визначення треба вказати додатні константи c_1,c_2 і число n_0 так, щоб нерівності

$$c_1n^2 \le 1/2 \cdot n^2 - 3n \le c_2n^2$$

виконувалися для всіх $n \ge n_0$. Розділимо дані нерівності на n^2:

$$c_1 \le 1/2 - 3/n \le c_2 .$$

Бачимо, що для виконання другої нерівності досить покласти $c_2 = 1/2$. Перше буде виконано, якщо, наприклад, $n_0 = 7$ і $c_1 = 1/14$.

Інший приклад використання формального визначення: покажемо, що $6n^3 \ne \Theta(n^2)$. Справді, нехай існують такі c_2 і n_0, що $6n^3 \le c_2n^2$ для всіх $n \ge n_0$. Але тоді $n \le c_2/6$ для всіх $n \ge n_0$ – що явно не так.

Визначаючи асимптотично точну оцінку для суми, можна відкидати члени меншого порядку, що при великих n стають малими порівняно з основним складовим. Зазначимо також, що коефіцієнт при старшому члені ролі не відіграє (він може вплинути тільки на вибір констант c_1 і c_2). Наприклад, розглянемо квадратичну функцію $f(n)=an^2+bn+c$, де a, b, c – деякі константи і $a>0$. Відкидаючи члени молодших порядків і коефіцієнт при старшому члені, знаходимо, що $f(n)=\Theta(n^2)$. Щоб переконатися в цьому формально, можна покласти $c_1=a/4$, $c_2=7a/4$ і $n_0=2\cdot max(|b|/a, (|c|/a)^{1/2})$. Взагалі, для будь-якого полінома $p(n)$ степеня d з додатним старшим коефіцієнтом маємо $p(n)=\Theta(n^d)$.

Згадаємо важливий окремий випадок використання Θ-позначень: $\Theta(1)$ – означає обмежену функцію, відокремлену від нуля деякою додатною константою при досить великих значеннях аргументу.

O- і *Ω*- позначення

Запис $f(n)=\Theta\, g(n)$ містить дві оцінки: верхню і нижню. Їх можна розділити. Говорять, що $f(n)=O(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0 \le f(n) \le cg(n)$ для всіх $n \ge n_0$ (див. рис. 2.1.б).

Кажуть, що $f(n)=\Omega(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0 \le cg(n) \le f(n)$ для всіх $n \ge n_0$ (див. рис 2.1.в).

Як і раніше ми припускаємо, що функції f і g додатні для досить великих значень аргументу. Легко побачити, що мають місце такі властивості:

Теорема 2.1. *Для будь-яких двох функцій $f(n)$ і $g(n)$ властивість $f(n)=\Theta(g(n))$ виконано тоді і тільки тоді, коли $f(n)=O(g(n))$ і $f(n)=\Omega(g(n))$.*

Для будь-яких двох функцій $f(n)$ і $g(n)$ властивості $f(n)=O(g(n))$ і $g(n)=\Omega(f(n))$ рівносильні.

Як ми бачили, $an^2+bn+c=\Theta(n^2)$ (при додатних a). Тому $an^2+bn+c=O(n^2)$. Інший приклад: при $a>0$ можна написати $an+b=O(n^2)$ (покладемо $c=a+|b|$ і $n_0=1$). Зауважимо, що в цьому випадку $an+b \ne \Omega(n^2)$ і $an+b \ne \Theta(n^2)$.

Асимптотичні позначення (Θ , O та Ω) часто вживаються усередині формул. Для $\Theta(n)$, наприклад, це означає деяку функцію, про яку нам важ-

ливо знати лише, що вона не менше $c_1 n$ і не більше $c_2 n$ для деяких додатних c_1 та c_2 і для всіх досить великих n. Аналогічно застосовують і позначення (O та Ω).

Часто асимптотичні позначення вживаються не цілком формально, хоча їх зміст, зазвичай, зрозуміло з контексту. Наприклад, ми можемо написати вираз

$$\sum_{i=1}^{n} O(i),$$

маючи на увазі суми $h(1) + h(2) + ... + h(n)$, де $h(i)$-деяка функція, для якої $h(i)=O(i)$. Легко побачити, що сама ця сума як функція від n є $O(n^2)$.

Типовий приклад неформального використання асимптотичних позначень – ланцюжок рівностей на зразок $2n^2+3n+1=2n^2+\Theta(n)=\Theta(n^2)$. Друга з цих рівностей ($2n^2+\Theta(n)=\Theta(n^2)$) розуміється при цьому так: яка б не була функція $h(n)=\Theta(n)$ у лівій частині, сума $2n^2+h(n)$ є $\Theta(n^2)$.

***o*- і ω-позначення**

Запис $f(n)=o(g(n))$ означає, що з ростом n відношення $f(n)/g(n)$ залишається обмеженим. Якщо до того ж

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, \qquad (2.1)$$

то ми пишемо $f(n) = o(g(n))$. Формально $f(n) = o(g(n))$, якщо для всякого додатного $\varepsilon > 0$ знайдеться таке n_0, що $0 \le f(n) \le \varepsilon g(n)$ при всіх $n \ge n_0$ (тим самим запис $f(n)=o(g(n))$ припускає, що $f(n)$ і $g(n)$ *невід'ємні* для досить великих n.)

Наприклад: $2n=o(n^2)$, але $2n^2 \neq o(n^2)$.

Аналогічно вводиться ω-позначення: кажуть, що $f(n)$ є $\omega(g(n))$, якщо для всякого додатного c найдеться таке n_0, що $0 \le cg(n) \le f(n)$ при всіх $n \ge n_0$. Очевидно, $f(n)=\omega(g(n))$ рівносильне $g(n)=o(f(n))$.

Наприклад: $n^2/2=\omega(n)$, але $n^2/2 \neq \omega(n^2)$.

Порівняння функцій

Введені нами означення мають деякі властивості транзитивності, рефлексивності та симетричності [1].

Транзитивність

$f(n)=\Theta(g(n))$ і $g(n)=\Theta(h(n))$ приводить $f(n)=\Theta(h(n))$,
$f(n)=O(g(n))$ і $g(n)=O(h(n))$ приводить $f(n)=O(h(n))$,
$f(n)=\Omega(g(n))$ і $g(n)=\Omega(h(n))$ приводить $f(n)=\Omega(h(n))$,
$f(n)=o(g(n))$ і $g(n)=o(h(n))$ приводить $f(n)=o(h(n))$,
$f(n)=\omega(g(n))$ і $g(n)=\omega(h(n))$ приводить $f(n)=\omega(h(n))$.

Рефлексивність

$f(n)=\Theta(f(n))$, $f(n)=O(f(n))$, $f(n)=\Omega(f(n))$.

Симетричність

$f(n)=\Theta(g(n))$ якщо і тільки якщо $g(n)=\Theta(f(n))$.

Звертання

$f(n)=O(g(n))$ якщо і тільки якщо $g(n)=\Omega(f(n))$,

$f(n)=o(g(n))$ якщо і тільки якщо $g(n)=\omega(f(n))$.

Можна провести таку паралель: відношення між функціями f і g подібні відношенню між числами a і b:

$$f(n)=O(g(n)) \approx a \le b,$$
$$f(n)=\Omega(g(n)) \approx a \ge b,$$
$$f(n)=\Theta(g(n)) \approx a=b,$$
$$f(n)=o(g(n)) \approx a<b,$$
$$f(n)=\omega(g(n)) \approx a>b.$$

Однак ця паралель досить умовна, оскільки властивості числових нерівностей не переносяться на функції. Наприклад, для будь-яких двох чисел a і b завжди або $a \le b$, або $a \ge b$, однак не можна стверджувати, що для будь-яких двох (додатних) функцій $f(n)$ та $g(n)$ або $f(n)=O(g(n))$, або $f(n)=\Omega(g(n))$. Справді, можна перевірити, що жодне з цих двох співвідношень не виконане для $f(n)=n$ та $g(n)=n^{1+\sin n}$ (показник ступеня у виразі для $g(n)$ змінюється у інтервалі від 0 до 2). Зазначимо також, що для чисел $a \le b$ приводить $a<b$ або $a=b$, у той час як для функцій $f(n)=O(g(n))$ не приводить $f(n)=o(g(n))$ або $f(n)=\Theta(g(n))$.

2.4 Рекурентні співвідношення

Багато задач, які доводиться розв'язувати програмістам, досить ефективно розв'язуються за допомогою так званих рекурсивних алгоритмів. Оцінюючи час роботи таких рекурсивних процедур, ми часто приходимо до співвідношення, що виражає цей час через час роботи тієї ж процедури на вхідних даних меншого розміру. Такого роду співвідношення називаються рекурентними [1].

Нижче наводяться три способи, що дозволяють розв'язати рекурентне співвідношення, тобто знайти асимптотичну оцінку для його розв'язання. По-перше, можна вгадати оцінку, а потім довести її за індукцією, підставляючи угадану формулу у праву частину співвідношення. Даний спосіб отримав назву *методу підстановки*. По-друге, можна «розгорнути» рекурентну формулу, одержавши при цьому суму, яку можна потім оцінювати. Цей спосіб називають *методом ітерацій*. Нарешті тут наводиться загальний рецепт для розв'язання рекурентних співвідношень вигляду

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

де $a \ge 1$, $b>1$ – деякі константи, а $f(n)$ – задана функція. Такий рецепт ґрунтується на *основній теоремі про рекуренті співвідношення*. Формулювання цієї теореми досить довге, зате в багатьох випадках вона відразу приводить до відповіді [1].

2.4.1 Метод підстановки

Ідея даного методу проста: відгадати відповідь і довести її за індукцією. Таке вгадування потребує певного досвіду. Наприклад, якщо Ви вже розв'язували подібне рекурентне співвідношення і отримали деяку відповідь – її можна прийняти і в даному випадку, а потім або довести правильність відповіді, або шукати іншу відповідь.

Часто відповідь містить коефіцієнти, які треба вибрати так, щоб міркування за індукцією проходило [1]. Індуктивний метод застосовують і до нижніх, і до верхніх оцінок.

Як приклад знайдемо верхню оцінку для функції, заданої співвідношенням

$$T(n) = 2T(\lfloor n/2 \rfloor) + n, \qquad (2.2)$$

де позначення $\lfloor n/2 \rfloor$ – найбільше ціле, що менше, або дорівнює n/2. (надалі ми будемо також застосовувати і позначення $\lceil n/2 \rceil$. Воно означає найменше ціле, що більше, або дорівнює n/2.)

Зауважимо, що співвідношення (2.2) має місце у випадку, коли алгоритм розбиває задачу розміром *n* на 2 підзадачі розміром *n/2,* ці підзадачі розв'язуються рекурсивно (кожна за час *T(n/2))* і результати об'єднуються. При цьому витрати на розбивання й об'єднання дорівнюють *n.*

Припустимо, що $T(n)=O(n\cdot\log n)$, тобто що $T(n)\le cn\cdot\log n$ для відповідного $c>0$. Доведемо це за індукцією. Нехай ця оцінка правильна для $\lceil n/2 \rceil$, тобто $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$. Підставивши її у співвідношення, одержимо

$$T(n) \le 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \le cn\log(n/2) + n =$$
$$= cn\log n - cn\log 2 + n = cn\log n - cn + n \le cn\log n.$$

Останній перехід законний при $c\ge 1$.

Залишається перевірити базис індукції, тобто довести оцінку для початкового значення *n.* Отут ми зіштовхуємося з тим, що при $n=1$ права частина нерівності перетворюється в нуль, яким би не взяти c (оскільки $\log 1 = 0$). В даному випадку слід пам'ятати, що асимптотичну оцінку досить довести для всіх *n*, починаючи з деякого. Підберемо c так, щоб оцінка $T(n) \le cn\cdot\log n$ була правильна при $n=2$ і $n=3$. Тоді для великих *n* можна міркувати за індукцією, і небезпечний випадок n = 1 нам не зустрінеться (оскільки $\lfloor n/2 \ge 2 \rfloor$ при $n > 3$).

Тепер виникає питання як вгадати відповідь? Для цього, як вже згадувалося вище потрібні певні навички і досвід. Наведемо приклади, які наштовхують на думку.

Аналогія. Розглянемо для прикладу співвідношення

$$T(n) = 2T(\lfloor n/2 \rfloor + 17) + n,$$

яке відрізняється від (2.2) додатковим доданком *17* у правій частині. Мож-

на очікувати, однак, що при великих n різниця між $\lfloor n/2 \rfloor + 17$ і $\lfloor n/2 \rfloor$ навряд чи така вже істотна. Природно припустити, що оцінка $T(n)=O(n \cdot \log n)$ залишається в силі, а потім довести це за індукцією.

Послідовні наближення. Можна почати з простих і грубих оцінок, а потім уточнювати їх. Наприклад, для співвідношення (2.2) є очевидна нижня оцінка $T(n)=\Omega(n)$ (оскільки праворуч є член n), і верхня оцінка $T(n)=O(n^2)$ (яку легко довести за індукцією). Далі можна поступово зближувати їх, прагнучи одержати асимптотично точні нижню і верхню оцінки, що відрізняються не більш ніж у константу раз.

Тонкощі. Іноді міркування за індукцією наштовхується на труднощі, хоча відповідь угадана правильно. Звичайно, це відбувається тому, що доказуване за індукцією твердження недостатньо вагоме. У цьому випадку може допомогти віднімання члена меншого порядку.

Розглянемо співвідношення $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$

Можна сподіватися, що в цьому випадку $T(n)=O(n)$. Це дійсно так. Спробуємо, однак, довести, що $T(n) \le cn$ при відповідному виборі константи c. Індуктивне припущення дає

$$T(n) \le c\lfloor n/2 \rfloor + c(\lceil n/2 \rceil) + 1 = cn + 1,$$

а звідси ні для якої константи c не випливає нерівність $T(n) \le cn$. У такій ситуації можна спробувати довести слабшу оцінку, наприклад $O(n^2)$, але насправді в цьому немає необхідності: наш здогад правильний, треба тільки не ослабити, а підсилити припущення індукції.

Нова гіпотеза виглядає так: $T(n) \le cn-b$ для деяких констант b і c. Підстановка в праву частину дає

$$T(n) \le (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = cn - 2b + 1 \le cn - b.$$

Останній перехід законний при $b \ge 1$. Залишається лише вибрати константу c з урахуванням початкових умов.

Це міркування викликає здивування. Якщо доведення не проходить, чи не варто послабити оцінку, а не підсилювати її? Але насправді нічого дивного тут немає – підсиливши оцінку, ми одержуємо можливість скористатися сильнішим індуктивним припущенням.

Як робити не треба. Асимптотичний запис небезпечний при неправильному застосуванні. Приклад неправильного доведення оцінки $T(n)=O(n)$ для співвідношення (2.2). Припустимо, що $T(n) \le cn$, тоді можна записати

$$T(n) \le 2(c\lfloor n/2 \rfloor) + n \le cn + n = O(n).$$

Це міркування, однак, нічого не доводить, тому що індуктивний перехід вимагає, щоб у правій частині було cn з тією ж самою константою c, а не абстрактне $O(n)$.

Заміна змінних

Часто нескладна заміна змінних дозволяє перетворити рекурентне співвідношення до звичного вигляду. Наприклад, співвідношення

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \ln n$$

здається досить складним, однак, заміною змінних його легко спростити. Зробивши заміну $m=\log n$, одержимо $T(2^m) = 2T(2^{m/2}) + m$.

Позначивши $T(2^m)$ через $S(m)$, приходимо до співвідношення

$$S(m) = 2S(m/2) + m,$$

яке вже зустрічалося (див. співвідношення(2.2)). Його розв'язок: $S(m)=O(m \cdot \log m)$. Тож, повертаючи до $T(n)$ замість $S(m)$, одержимо

$$T(n) = T(2^m) = S(m) = O(m \log m) = O(\log n \log\log n).$$

Приведене міркування вимагає, звичайно, уточнень, оскільки поки що ми довели оцінку на $T(n)$ лише для n, що є степенями двійки. Щоб вийти з положення, можна визначити $S(m)$ як максимум $T(n)$ для усіх n, що не перевищує 2^m.

2.4.2 Метод ітерацій

Як бути, якщо розв'язок вгадати не вдається? Тоді можна, ітеруючи це співвідношення (підставляючи його само в себе), одержати ряд, який можна оцінювати тим або іншим способом [1].

Для прикладу розглянемо співвідношення

$$T(n) = 3T(\lfloor n/4 \rfloor) + n$$

Підставляючи його в себе, одержимо:

$$T(n) = n + 3T(\lfloor n/4 \rfloor) = n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) = n + 3(\lfloor n/4 \rfloor +$$

$$+3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) = n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor).$$

Тут ми скористалися тим, що $\lfloor \lfloor n/4 \rfloor / 4 \rfloor = \lfloor n/16 \rfloor$ і $\lfloor \lfloor n/16 \rfloor / 4 \rfloor = \lfloor n/64 \rfloor$. Скільки кроків треба зробити, щоб дійти до початкової умови? Оскільки після і-ї ітерації праворуч виявиться $T(\lfloor n/4^i \rfloor)$, ми дійдемо до $T(1)$, коли $\lfloor n/4^i \rfloor = 1$, тобто коли $i \ge \log_4 n$. Враховуючи, що $\lfloor n/4^i \rfloor \le n/4^i$, ми можемо оцінити наш ряд спадною геометричною прогресією (плюс останній член, що відповідає $3^{\log_4 n}$ задачам обмеженого розміру):

$$T(n) \le n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \le$$

$$\le n\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i + \Theta(n^{\log_4 3}) = 4n + o(n) = O(n).$$

(Ми замінили кінцеву суму з не більш ніж $\log_4 n+1$ членів на суму нескінченного ряду, а також переписали $3^{\log_4 n}$ як $n^{\log_4 3}$, що є $o(n)$, оскільки $\log_4 3<1$.)

Часте перетворення рекурентного співвідношення в суму приводить до досить складних викладень. При цьому важливо стежити за двома речами: скільки кроків підстановки потрібно і яка сума членів, що виходять на даному кроці. Іноді декілька перших кроків дозволяють відгадати відповідь, що потім вдається довести за індукцією (з меншою кількістю обчислень).

Особливо багато турбот доставляють округлення (перехід до цілої частини). Для початку варто припустити, що значення параметра такі, що округлення не потрібні (у нашому прикладі при $n=4^k$ нічого округляти не треба). Взагалі, таке припущення не цілком законне, тому що оцінку треба довести для всіх досить великих цілих чисел, а не тільки для степенів четвірки. Нижче ми побачимо як можна обійти ці складності.

Дерева рекурсії

Процес підстановки співвідношення в себе можна зобразити у вигляді дерева рекурсії [1]. Як це робиться на прикладі співвідношення

$$T(n) = 2T(n/2) + n^2$$

показано на рис. 2.2.

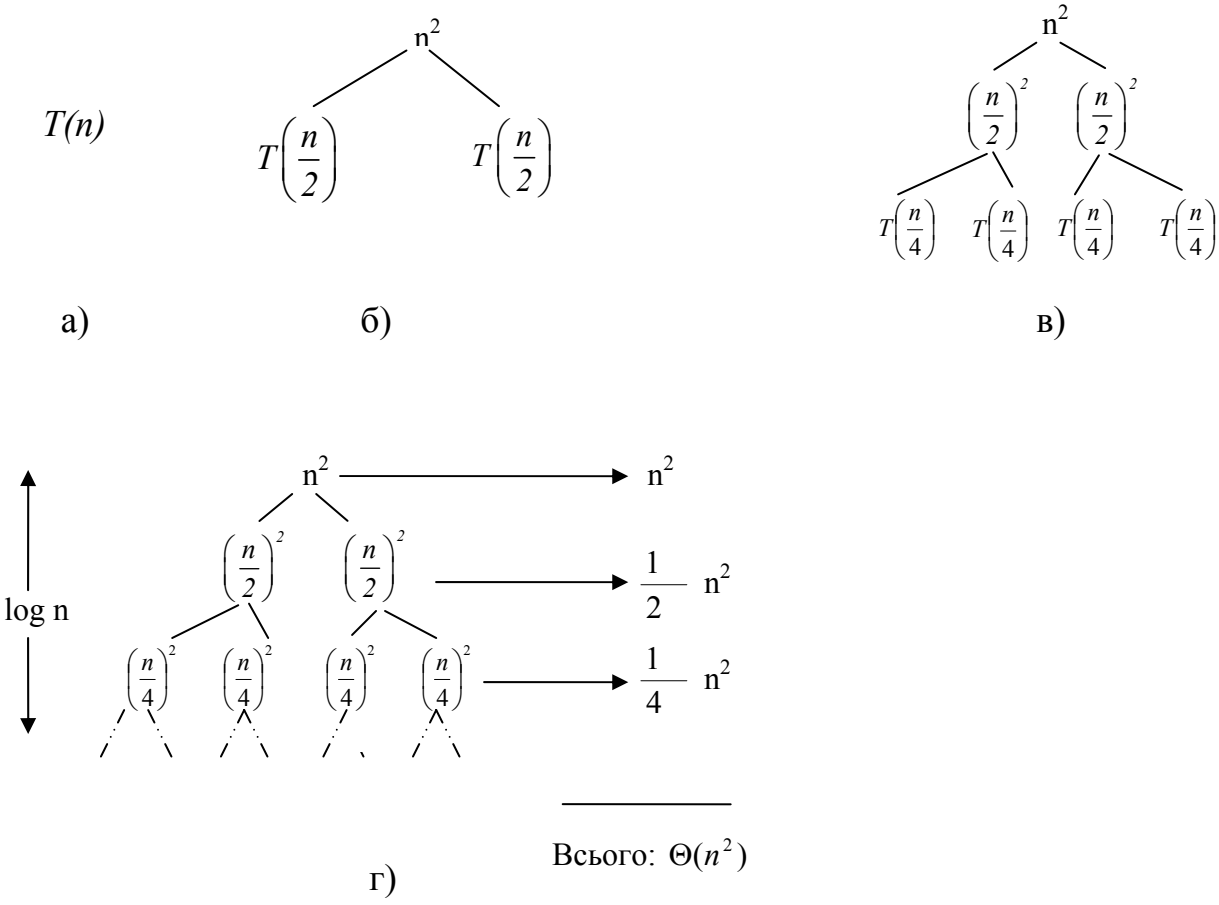


Рисунок 2.2 – Дерево рекурсії для співвідношення $T(n) = 2T(n/2) + n^2$

Для зручності припустимо, що *n* степінь двійки. Рухаючись від (а) до (г), ми поступово розвертаємо вираз для *T(n),* використовуючи вираз для *T(n), T(n/2), T(n/4)* і т. д. Тепер ми можемо обчислити *T(n) шляхом знаходження суми* значень вершин з кожного рівня. На верхньому рівні маємо n^2, на другому: $(n/2)^2+(n/2)^2=n^2/2$, на третьому: $(n/4)^2+(n/4)^2+(n/4)^2+(n/4)^2=n^2/4$. Виходить спадна геометрична прогресія, сума якої відрізняється від її першого члена не більше ніж на постійний множник. Отже, $T(n)=\Theta(n^2)$. На рис. 2.3 показано дещо складніший приклад – дерево для співвідношення

$$T(n)=T(n/3)+T(2n/3)+n$$

(для простоти ми знову ігноруємо округлення). Тут сума значень на кожному рівні дорівнює *n.* Дерево обривається, коли значення аргументу стають порівнянними з 1. Для різних галузей це відбувається на різних рівнях, і найдовшиий шлях $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$ вимагає біля $k=\log_{3/2} n$ кроків (при такому *k* ми маємо $(2/3)^k n=1$). Тому *T(n)* можна оцінити як $O(n \cdot \log n)$.

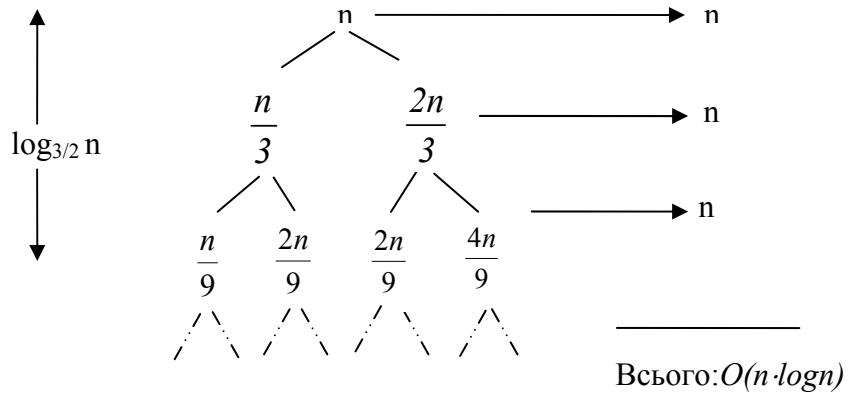


Рисунок 2.3 – Дерево рекурсії для співвідношення $T(n)=T(n/3)+T(2n/3)+n$

2.4.3 Загальний рецепт

Даний метод можна застосовувати для рекурентних співвідношень вигляду

$$T(n)=aT(n/b)+f(n), \qquad (2.3)$$

де $a \geq 1$ і $b>1$ – деякі константи, а *f* – додатна (принаймні для великих значень аргументу) функція. Він дає загальну формулу, запам'ятавши яку, можна швидко розв'язати різні рекурентні співвідношення [1].

Співвідношення (2.3) виникає, якщо алгоритм розбиває задачу розміром *n* на *a* підзадач розміром *n/b,* ці підзадачі розв'язуються рекурсивно (кожна за час *T(n/b))* і результати об'єднуються. При цьому витрати на розбиття й об'єднання описуються функцією *f(n)*

Зауважимо, що у формулі (2.3) виникає проблема з округленням, оскільки частка *n/b* може не бути цілою. Формально варто було б замінити

$T(n/b)$ на $T(\lfloor n/b \rfloor)$ або $T(\lceil n/b \rceil)$. Обидва варіанти приведуть до одної і тої ж відповіді, і для простоти ми будемо упускати округлення в наших формулах [1].

Основна теорема про рекурентне оцінювання

Нехай $a \geq 1$ і $b>1$ – константи, $f(n)$ – функція, $T(n)$ визначено при невід'ємних n формулою

$$T(n) = aT(n/b) + f(n), \tag{2.4}$$

де під n/b розуміється або $\lceil n/b \rceil$, або $\lfloor n/b \rfloor$. Тоді:

1) якщо $f(n) = O(n^{\log_b a - \varepsilon})$ для деякого $\varepsilon>0$, то $T(n) = \Theta(n^{\log_b a})$;

2) якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;

3) якщо $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для деякого $\varepsilon>0$ і якщо $af(n/b) \leq cf(n)$ для деякої константи $c<1$ і досить великих n, то $T(n) = \Theta(f(n))$.

Суть цієї теореми така. У кожному з трьох випадків ми порівнюємо $f(n)$ з $n^{\log_b a}$ і якщо одна з цих функцій зростає швидше інших, то вона і визначає порядок росту $T(n)$ (випадки 1 і 3). Якщо обидві функції одного порядку (випадок 2), то з'являється додатковий логарифмічний множник і відповіддю є формула $\Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.

Відзначимо важливі технічні деталі. У першому випадку недостатньо, щоб $f(n)$ була просто менша, ніж $n^{\log_b a}$ нам потрібен „зазор" розміром n^{ε} для деякого $\varepsilon>0$. Точно так само в третьому випадку $f(n)$ повинна бути більше $n^{\log_b a}$ з запасом і до того ж задовольняти умову регулярності: $af(n/b) \leq cf(n)$.

Зауважимо, що три зазначених випадки не вичерпують усіх можливостей: може виявитися, наприклад, що функція $f(n)$ менша, ніж $n^{\log_b a}$, але зазор недостатньо великий для того, щоб скористатися першим твердженням теореми. Аналогічна „щілина" є і між випадками 2 і 3. Нарешті, функція може не мати властивості регулярності.

Застосування основної теореми. Розглянемо кілька прикладів, де застосування теореми дозволяє відразу ж записати відповідь. Для початку розглянемо співвідношення

$$T(n) = 9T(n/3) + n.$$

У цьому випадку $a=9$, $b=3$, $f(n)=n$, а $n^{\log_b a} = \Theta(n^2)$. Оскільки $f(n) = O(n^{\log_3 9 - \varepsilon})$ для $\varepsilon=1$, ми застосовуємо перше твердження теореми й робимо висновок, що $T(n) = \Theta(n^2)$.

Тепер розглянемо співвідношення

$$T(n) = T(2n/3) + 1.$$

Тут $a=1$, $b=3/2$, $f(n)=1$ і $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Підходить випадок 2, оскільки $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, і ми одержуємо, що $T(n) = \Theta(\log n)$.

Для співвідношення

$$T(n) = 3T(n/4) + n \cdot \log n$$

ми маємо $a=3,\ b=4,\ f(n)= n \cdot \log n$; при цьому $n^{\log_b a} = n^{\log_3 4} = O(n^{0,793})$. Зазор (з $\varepsilon \approx 0,2$) є, залишається перевірити умову регулярності. Для досить великого n маємо $af(n/b) = 3(n/4)\log(n/4) \le (3/4)n \cdot \log n = cf(n)$ для $c=3/4$. Тим самим по третьому твердженню теореми $T(n) = \Theta(n \cdot \log n)$.

Наведемо приклад, коли теорему застосувати не вдається. Нехай

$$T(n) = 2T(n/2) + n \cdot \log n .$$

Тут $a=2,\ b=2,\ f(n)=n \cdot \log n,\ n^{\log_b a} = n$. Видно, що $f(n)=n \cdot \log n$ асимптотично більша, ніж $n^{\log_b a}$, але зазор недостатній: відношення

$$f(n) = (n \cdot \log n)/n = \log n$$

не оцінюється знизу величиною n^{ε} ні для якого $\varepsilon > 0$. Це співвідношення потрапляє у проміжок між випадками 2 та 3 і для нього можна одержати відповідь за формулою

$$T(n) = \Theta\left(\log^2 n\right).$$

І взагалі, якщо

$$f(n) = \Theta\left(n^{\log_b a} \log^k n\right),$$

де $k \ge 0$, то співвідношення (2.4), як вказується в [1] приводить до

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right).$$

Доведення основної теореми. Доведення складається з двох частин. Спочатку ми розглядаємо лише ті n, що є ступенями числа b. Усі основні ідеї видно вже для цього випадку. Потім отриманий результат поширюється на всі натуральні числа, при цьому ми акуратно стежимо за округленнями і т. і.

В даному розділі ми дозволимо собі не зовсім коректно застосовувати асимптотичний запис: будемо використовувати його для функцій, визначених тільки на степенях числа b, хоча визначення вимагає, щоб оцінки доводилися для всіх досить великих натуральних чисел.

З контексту буде зрозуміло, що мається на увазі, але потрібно бути уважним, щоб не заплутатися: якщо про функції $T(n)$ нічого не відомо, то оцінка $T(n)=O(n)$ для n, що є степенями двійки, нічого не гарантує для довільних n. Можливо, що $T(n)=n$ при $n=1, 2, 4, 8, \ldots$ і $T(n) = n^2$ при інших n.

Випадок натуральних степенів. Нехай $T(n)$ визначено для чисел, що є (натуральними) ступенями числа $b>1$ (не обов'язково цілого), і задовольняє співвідношення (2.4), тобто

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

Ми одержимо оцінку для $T(n)$ так: перейдемо від цього співвідношення до підсумовування (лема 2.1), потім оцінимо отриману суму (лема 2.2) і підведемо підсумки (лема 2.3) [1].

Лема 2.1 *Нехай* $a \ge 1$ *a, b>1* – константи, *f(n) – невід'ємна функція, що визначена на степенях b. Нехай T(n) – функція, визначена на степенях b співвідношенням* $T(n) = aT(n/b) + f(n)$ *при n>1, причому T(1)>0. Тоді*

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^i f(n/b^j). \qquad (2.5)$$

Доведення. Послідовно підставляючи співвідношення само в себе, одержуємо

$$\begin{aligned} T(n) &= f(n) + aT(n/b) = \\ &= f(n) + af(n/b) + a^2 T(n/b^2) = \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ ... \\ &... + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1). \end{aligned}$$

Оскільки $a^{\log_b n} = n^{\log_b a}$, останній член може бути записаний як $\Theta(n^{\log_b a})$. Члени, що залишилися, утворять суму, що фігурує у твердженні леми.

Дерево рекурсії. Доведення леми 2.1 можна пояснити в термінах дерева рекурсії (рис. 2.4), якщо число *a* ціле (хоча саме доведення цього не вимагає). У корені розташовано число *f(n),* у кожному з *a* його дітей міститься число *f(n/b),* у кожному з a^2 онуків міститься $f(n/b^2)$ і т. д. На рівні j є a^j вершин; вага кожної дорівнює $f(n/b^j)$. Листки знаходяться на відстані $\log_b n$ від кореня, і мають додатну вагу *T(1).* Всього на дереві міститься $a^{\log_b n} = n^{\log_b a}$ листків.

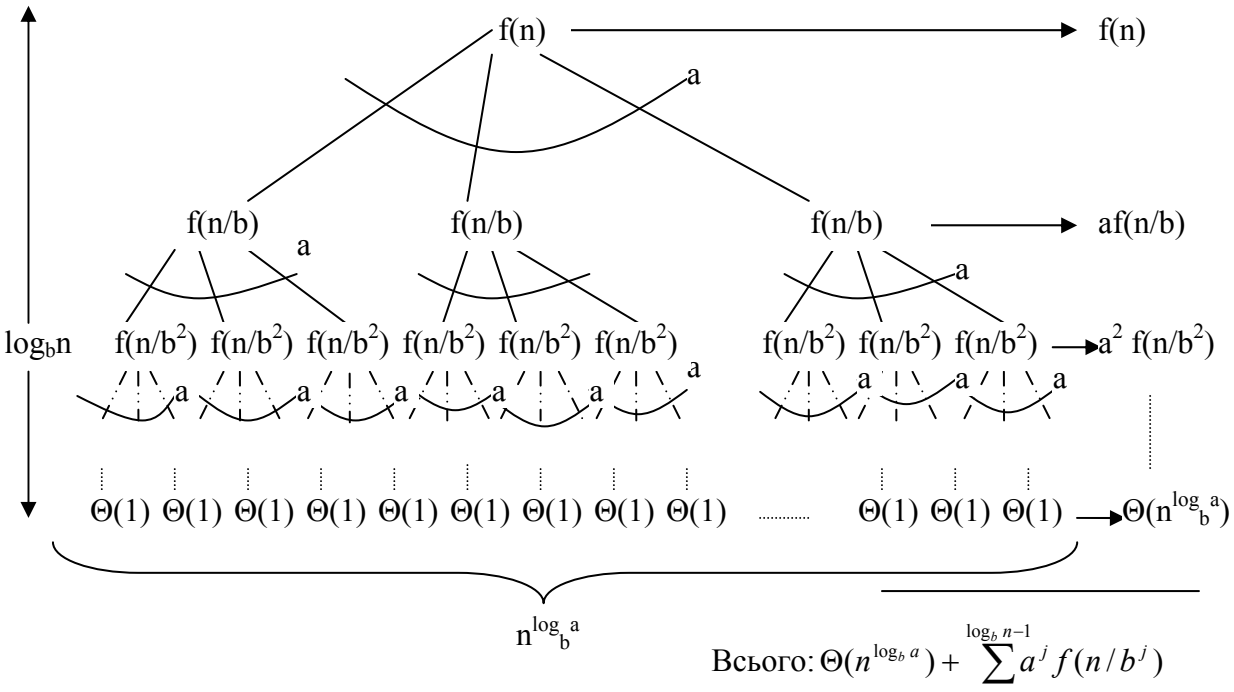


Рисунок 2.4 – Дерево рекурсії для співвідношення *T(n)=aT(n/b)+f(n)*

Рівняння (2.5) можна отримати, якщо знайти суму ваг на всіх рівнях: загальна вага на j-му рівні дорівнює $a^j f(n/b^j)$, а загальна вага внутрішньої частини дерева є

$$\sum_{j=0}^{\log_b n-1} a^j f\left(n/b^j\right).$$

Якщо рекурентне співвідношення описує алгоритм типу „розділяй і володарюй", ця сума відповідає вартості розбивання задачі на підзадачі та об'єднань розв'язків. Сумарна вага всіх листків є вартість розв'язання усіх $n^{\log_b a}$ задач розміром 1, що складає $\Theta(n^{\log_b a})$.

У термінах дерева рекурсії легко пояснити, чому відповідають три випадки у формулюванні основної теореми. У першому випадку основна частина ваги зосереджена у листках, в третьому – у корені, в другому вага рівномірно розподілена по рівнях дерева.

Тепер оцінимо величину суми у формулі (2.5).

Лема 2.2 *Нехай $a \geq 1$, $b>1$ – константи, $f(n)$ – невід'ємна функція, визначена на натуральних степенях b. Розглянемо функцію*

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \tag{2.6}$$

для n, що є степенями b. Тоді:

1) *якщо $f(n) = O(n^{\log_b a-\varepsilon})$ для деякої константи $\varepsilon>0$, то $g(n) = O(n^{\log_b a})$;*

2) *якщо $f(n) = \Theta(n^{\log_b a})$, то $g(n) = \Theta(n^{\log_b a} \log n)$;*

3) *якщо $af(n/b) \leq cf(n)$ для деякої константи $z < 1$* і для всіх $n \geq b$, то $g(n) = \Theta(f(n))$.

Доведення

1. У першому випадку досить довести твердження леми для функції $f(n) = n^\alpha$, де $\alpha = \log_b a - \varepsilon$. Для такої функції f рівність (2.6) може бути переписана так (тепер $\alpha = \log_b a$):

$$\begin{aligned} g(n) &= f(n) + af(n/b) + a^2 f(n/b^2) + \ \ldots \ + a^{k-1} f(n/b^{k-1}) = \\ &= n^\alpha + a(n/b)^\alpha + a^2(n/b^2)^\alpha + \ \ldots \ + a^{k-1}(n/b^{k-1})^\alpha \end{aligned} \tag{2.7}$$

права частина є геометричною прогресію довжини $k=\log_b n$, n зі знаменником a/b^α; цей знаменник більше 1, тому що $\alpha < \log_b a$ і $b^\alpha < a$. Для такої прогресії сума за порядком дорівнює останньому члену (відрізняється від нього не більш ніж у константу разів). Цей останній член є $O(a^k) = O(n^{\log_b a})$. Для першого випадку твердження леми доведене.

2. В другому випадку аналогічне міркування дає суму того ж вигляду:

$$g(n) = f(n) + af(n/b) + a^2 f(n/b^2) + \ldots = n^\alpha + a(n/b)^\alpha + a^2(n/b^2)^\alpha + \ldots,$$

але тепер $a = \log_b a$ і тому знаменник геометричної прогресії дорівнює 1 і всі її члени рівні. Число членів є $\log_b n$, і тому сума дорівнює

$$n^{\log_b a} \log_b n = \Theta(n^{\log_b a} \log n).$$

Для другого випадку твердження леми також доведено.

3. У цьому випадку умова регулярності функції f гарантує, що в нашій сумі кожен наступний член не перевищує попереднього, помноженого на $c<1$. Тим самим її можна оцінити зверху спадною геометричною прогресією зі знаменником c, і сума такої прогресії не більш ніж у константу (рівну $1/(1-c)$) разів перевершує перший член (але і не менше його, оскільки усі доданки невід'ємні). Таким чином, $g(n) = \Theta(f(n))$ для n, що є степенями b. Доведення леми завершено.

Тепер ми можемо довести основну теорему про рекурентні оцінки для випадку, коли n є натуральний степінь b.

Лема 2.3 *Нехай $a \geq 1$, $b > 1$ – константи, f(n) – невід'ємна функція, визначена на степенях b. Нехай T(n) – функція, визначена на ступенях b співвідношенням $T(n)=a \cdot T(n/b)+f(n)$ при $n > 1$ і $T(1)>0$. Тоді:*

1) *якщо $f(n) = O(n^{\log_b a-\varepsilon})$ для деякого $\varepsilon>0$, то $T(n) = \Theta(n^{\log_b a})$;*

2) *якщо $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;*

3) *якщо $f(n) = \Omega(n^{\log_b a+\varepsilon})$ для деякого $\varepsilon>0$ і якщо $af(n/b) \leq cf(n)$ для деякої константи $c<1$ і для досить великих n, то $T(n) = \Theta(f(n))$.*

Доведення складається в комбінації лем 2.1 і 2.2.

В першому випадку одержуємо $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$.

В другому випадку $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$.

В третьому випадку $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$.

Зауважимо, що в останньому випадку умову „$f(n) = \Omega(n^{\log_b a+\varepsilon})$ для деякого $\varepsilon>0$" можна було б опустити, тому що воно випливає з умови регулярності.

Цілі наближення зверху і знизу

Тепер нам залишилось вирішити питання з довільними n, що не є степенями числа b. У цьому випадку частка n/b підлягає округленню і співвідношення має вигляд

$$T(n) = aT(\lceil n/b \rceil) + f(n) \qquad (2.8)$$

або

$$T(n) = aT(\lfloor n/b \rfloor) + f(n). \qquad (2.9)$$

Треба переконатися, що оцінка залишається в силі і для цього випадку. Подивимось, які зміни треба внести в наші міркування. Замість послідовності $n, n/b, n/b^2, \ldots$ тепер треба розглядати послідовність n_i, визначену так:

$$n_i = \begin{cases} n, & \text{якщо } i = 0, \\ \lceil n_{i-1}/b \rceil, & \text{якщо } i > 0. \end{cases} \qquad (2.10)$$

(ми розглядаємо випадок округлення з надлишком). Ця послідовність – спадна, але не обов'язково наближається до одиниці. Однак ми можемо стверджувати, що після $log_b n$ ітерацій вийде число, обмежене не залежною від n (хоча залежною від b) константою.

Справді, з нерівності $\lceil x \rceil \le x + 1$ випливає, що

$$n_0 \le n;$$

$$n_1 \le \frac{n}{b} + 1;$$

$$n_2 \le \frac{n}{b^2} + \frac{1}{b} + 1;$$

$$n_3 \le \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1;$$

.....

Оскільки $1 + 1/b + 1/b^2 + \ ... \ \le b/(b-1)$, для $i = \lfloor log_b n \rfloor$, одержуємо $n_i \le n/b^i + b/(b-1) \le b + b/(b-1) = O(1).$

Ітеруючи співвідношення (2.8), одержуємо

$$\begin{aligned} T(n) &= f(n_0) + aT(n_1) = f(n_0) + af(n_1) + a^2 T(n_2) = \\ &= f(n_0) + af(n_1) + a^2 T(n_2) + \ ... \ + a^{\lfloor log_b n \rfloor - 1} f(n_{\lfloor log_b n \rfloor - 1}) + \\ &+ a^{\lfloor log_b n \rfloor} T(n_{\lfloor log_b n \rfloor}) = \Theta(n^{log_b a}) + \sum_{j=0}^{\lfloor log_b n \rfloor - 1} a^j f(n_j). \end{aligned} \qquad (2.11)$$

Це рівняння дуже схоже на (2.5), тільки тут n необов'язково повинно бути степенем числа b.

Строго кажучи, нам слід дещо уточнити. Справа в тому, що величина $T\left(n_{\lfloor log_b n \rfloor}\right)$ може виявитися рівною нулю. Але ми знаємо, що $f(n)$ асимптотично додатна, тобто додатна для всіх n, починаючи з деякого n_0. Тому в розгортанні суми потрібно зупинитися трохи не дійшовши до цього n_0 [1].

Тепер треба визначитися зі співвідношеннями між доданками в сумі

$$g(n) = \sum_{j=0}^{\lfloor log_b n \rfloor - 1} a^j f(n_j). \qquad (2.13)$$

Раніше ми порівнювали цю суму з геометричною прогресією, у якій знаменник був більше одиниці (перший випадок), одиничний (другий випадок) або менший за одиницю (третій випадок). Третій випадок не викликає додаткових проблем, тому що в умові регулярності також передбачається округлення (причому в ту ж сторону, що й в рекурентному співвідношенні). В другому випадку ми повинні оцінити, наскільки великі зміни внаслідок заміни n/b^2 на n_i. Як ми бачили, різниця не перевищує $b/(b-1)$, але треба ще від абсолютної помилки перейти до відносної. Нам треба перевірити,

що $f(n_j) = O(n^{\log_b a} / a^j) = O((n / b^j)^{\log_b a})$, тоді проходить доведення леми 2.2 (випадок 2). Зазначимо, що $b^j / n \le 1$ при $j \le \lfloor \log_b n \rfloor$. З оцінки $f(n) = O(n^{\log_b a})$ випливає, що для підходящих c і досить великих n_j

$$f\left(n_j\right) \le c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} = c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \le$$

$$\le c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} = O\left(\frac{n^{\log_b a}}{a^j}\right).$$

Під час останнього переходу ми враховували, що $c(1+b/(b-1))^{\log_b a}$ – константа. Отже, випадок 2 розглянуто.

Міркування для випадку 1 майже таке ж. Тут слід довести оцінку $f(n_j)=O(n^{\log_b a-\varepsilon})$, а це виконується приблизно так, як і у випадку 2, хоча перетворення будуть трохи складніші [1].

Отже, ми розглянули випадок довільного n для округлення з надлишком. Аналогічно можна розглянути випадок округлення з недостачею, при цьому потрібно буде доводити, що значення n_j не може бути набагато меншим ніж n/b^j.

2.5 Контрольні питання

1. Наведіть основні критерії оцінювання складності алгоритмів.
2. Поясніть, чому для нас найбільшу цікавість становлять саме асимптотичні оцінки?
3. Поясніть, чому у час стрімкого зростання потужностей обчислювальної техніки питання розробки ефективних алгоритмів залишається дуже важливим та актуальним.
4. Які асимптотичні позначення найчастіше застосовують для визначення складності алгоритмів? В чому суть цих позначень? Наведіть відповідні приклади.
5. Наведіть основні властивості асимптотичних позначень.
6. Є два алгоритма, що мають однакову асимптотичну складність. Можна стверджувати, що ефективність алгоритмів однакова? Відповідь мотивуйте.
7. У якому випадку оцінка складності алгоритму визначається рекурентним співвідношенням? Наведіть відповідний приклад.
8. Перерахуйте відомі Вам методи розв’язання рекурентних співвідношень.
9. Охарактеризуйте та наведіть приклад застосування методу підстановки.

10. Наведіть приклад застосування методу ітерацій. В яких випадках доцільно застосовувати даний метод?
11. Нарисуйте дерево рекурсії для довільного рекурентного співвідношення та дайте оцінку даному співвідношенню.
12. Сформулюйте основну теорему про рекурентне оцінювання.
13. Наведіть власні приклади застосування основної теореми про рекурентне оцінювання.
14. Наведіть порівняльний аналіз основних методів розв'язання рекурентних співвідношень.

2.6 Завдання

1. Яку ступінь зростання, використовуючи *O*- символіку має функція $T(n) = 3n^3 + 2n^2$? Відповідь проілюструйте відповідним доказом.

2. Покажіть, що:

а) з $T(n) = T\left(\lceil n/2 \rceil\right) + 1$ випливає $T(n)=O(\log n)$.

б) з $T(n) = 2T\left(\lfloor n/2 \rfloor\right) + n$ випливає $T(n) = \Omega(n \cdot \log n)$, і тим самим $T(n) = \Theta(n \cdot \log n)$.

в) з $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ випливає $T(n)=O(n \cdot \log n)$.

3. Розв'яжіть за допомогою заміни змінних співвідношення $T(n) = 2T\left(\sqrt{n}\right) + 1$ (без урахування того, що значення змінних – не цілі).

4. Ітеруючи співвідношення $T(n)=3T\left(\lfloor n/2 \rfloor\right)+1$ знайдіть верхню асимптотичну оцінку для $T(n)$.

5. Аналізуючи дерево рекурсії, покажіть, що з $T(n)=T(n/3)+T(2n/3)+n$ випливає $T(n)=\Omega(n \cdot \log n)$.

6. Нарисуйте дерево рекурсії для $T(n) = 4T(\lfloor n/2 \rfloor) + n$ і отримайте асимптотично точні оцінки для $T(n)$.

7. За ітераційним методом розв'яжіть співвідношення $T(n)=T(n-a)+T(a)+n$, де $a \geq 1$ – деяка константа.

8. За допомогою дерева рекурсії розв'яжіть співвідношення $T(n) = T(\alpha n) + T\left((1-\alpha)n\right) + n$, де α – константа в інтервалі $0 < \alpha < 1$.

9. Використовуючи основну теорему про рекурентне оцінювання,

знайдіть асимптотично точні оцінки для співвідношень:

а) $T(n)=4T(n/2)+n$;
б) $T(n)=4T(n/2)+n^2$;
в) $T(n)=4T(n/2)+n^3$;
г) $T(n)=T(n/2)+1$.

10. Час роботи алгоритму A описується співвідношенням $T(n)=7T(n/2)+n^2$, а час роботи алгоритму A' – співвідношенням $T'(n)=a\times T'(n/4)+n^2$. При якому найбільшому цілому значенні a алгоритм A' асимптотично швидший, ніж A?

11. Знайдіть якомога точніші асимптотичні верхні і нижні оцінки для нижченаведених співвідношень (вважаємо, що $T(n)$ – константа при $n\le 2$).

а) $T(n) = 2T(n/2) + n^3$;
б) $T(n) = 4T(n/3) + 1$;
в) $T(n) = T(9n/10) + n$;
г) $T(n) = 16T(n/4) + n^2$;
д) $T(n) = 3T(n/2) + 1$;
е) $T(n) = 7T(n/3) + n^2$;
ж) $T(n) = 7T(n/2) + n^2$;
и) $T(n) = 2T(n/4) + \sqrt{n}$;
к) $T(n) = T(n - 1) + n$;
л) $T(n) = T\left(\sqrt{n}\right) + 1$.

12. Вкажіть по можливості найбільш точні асимптотичні верхні і нижні оцінки для $T(n)$ у кожному із зазначених нижче прикладів. Передбачається, що $T(n)$ – константа при $n\le 8$.

а) $T(n) = 3T(n/2) + n \log n$;
б) $T(n) = 3T(n/3 + 5) + n/2$;
в) $T(n) = 2T(n/2) + n/\log n$;
г) $T(n) = T(n-1) + 1/n$;.
д) $T(n) = T(n-l)+\log n$;
е) $T(n) = \sqrt{n}\cdot T\left(\sqrt{n}\right) + n$.

3 АЛГОРИТМИ СОРТУВАННЯ ТА ЇХ АНАЛІЗ

Сортування є дуже показовим прикладом завдання, яке можна вирішувати за допомогою багатьох різних алгоритмів. Кожен з них має свої переваги та недоліки, і вибирати алгоритми потрібно, виходячи з конкретної постановки задачі.

У загальному випадку сортування слід розуміти як процес перегрупування заданої множини об'єктів у деякому певному порядку. Мета сортування – полегшити подальший пошук елементів у такій відсортованій множині. Це майже універсальна, фундаментальна діяльність [4]. Ми зустрічаємося з відсортованими об'єктами в телефонних довідниках, у списках прибуткових податків, у змістах книг, у бібліотеках, у словниках, на складах – майже скрізь, де потрібно шукати певні об'єкти.

Цікавість до сортування ґрунтується на тому, що під час побудови алгоритмів ми зустрічаємось з багатьма достатньо фундаментальними прийомами. Майже не існує методів, з якими не доводиться зустрічатися під час обговорення цього питання. Зокрема, сортування – це ідеальний об'єкт для демонстрації величезної розмаїтості алгоритмів, оскільки усі вони винайдені для однієї задачі і багато з них в деякому смислі оптимальні. Тому це ще й ідеальний об'єкт, який демонструє необхідність аналізу ефективності алгоритмів. До того ж на прикладах сортувань можна показати, як шляхом ускладнення алгоритму, хоча під рукою і є вже очевидні методи, можна домогтися значного виграшу в ефективності [1, 4].

3.1 Внутрішнє і зовнішнє сортування

Вибір алгоритму залежить від структури оброблюваних даних – це майже закон, але у випадку сортування така залежність настільки глибока, що відповідні методи були навіть поділені на два великих класи – *сортування масивів* і *сортування файлів (послідовностей)*. Іноді їх називають *внутрішнім* і *зовнішнім* сортуванням, оскільки масиви зберігаються у швидкій, оперативній, внутрішній пам'яті комп'ютера з випадковим доступом, а файли зазвичай розташовуються у більш повільній, але й більш ємній зовнішній пам'яті, на пристроях, заснованих на механічних переміщеннях (дисках або стрічках тощо).

Перш ніж розглядати конкретні алгоритми, введемо деякі поняття й позначення. Якщо в нас є елементи $a_1, a_2, \ldots, a_n$, то сортування є переставленням цих елементів у масив $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$, де при деякій функції впорядкування f виконується відношення

$$f\left(a_{k_1}\right) \le f\left(a_{k_2}\right) \le \ \ldots \ \le f\left(a_{k_n}\right). \qquad (3.1)$$

Зазвичай, функція впорядкування не обчислюється за будь-яким правилом, а зберігається як явний компонент (поле) кожного елементу.

Її значення називається ключем елементу.

Розглядаючи алгоритми сортування, ми будемо звертати увагу лише на компонент-ключ, інші ж компоненти можна навіть і не визначати.

Метод сортування називається *стійким,* якщо в процесі сортування відносне розташування елементів з рівними ключами не змінюється. Стійкість сортування часто буває бажаною, якщо мова йде про елементи, уже впорядкованих (відсортованих) за деякими вторинними ключами (тобто властивостями), що не впливають на основний ключ [1, 4].

3.2 Алгоритми сортування за квадратичний час

В даному розділі розглянуто та проаналізовано найпростіші алгоритми сортування: сортування вставками, за допомогою прямого обміну, а також методом бульбашкового сортування і його модифікаціями.

3.2.1 Сортування вставками

Сортування вставками зручне для сортування коротких послідовностей. Ідея процедури сортування вставками полягає в тому, що масив для сортування розбивається на 2 частини: ліву та праву. На початку частина, що розташована зліва містить лише один елемент, і отже є відсортованою; частина, що розташована справа містить усі інші, невідсортовані елементи. На кожному кроці з неї обирається наступний елемент і розташовується на потрібне місце у вже відсортованій послідовності. Коли останній елемент масиву займе відповідне місце сортування завершується.

Алгоритм сортування, що реалізує дану ідею наведений на рисунку 3.1. Індекс j вказує черговий елемент. Ділянку $A[i \ldots j-1]$ складають вже відсортовані елементи, а $A[j+1 \ldots n]$ – ще не переглянуті. У зовнішньому циклі алгоритму (блок 1) індекс j пробігає масив зліва направо. Ми беремо елемент $A[j]$ (блок 2) і пересуваємо елементи, що йдуть перед ним і є більшими ніж він (починаючи з $(j-1)$-го) справа, звільняючи місце для взятого елементу (блоки 3 – 6). У блоці 7 елемент $A[j]$ розміщується на звільнене місце.

Таким чином, послідовність сортується „на місці”, без додаткової пам’яті, тобто крім масиву ми використовуємо лише фіксоване число елементів пам'яті. Після виконання даного алгоритму масив A впорядкований за зростанням.

На рисунку 3.2 проілюстрована робота алгоритму при A=(5;2;4;6;1;3). Кружком обведено черговий елемент, що підлягає розміщенню, а стрілкою знайдене місце його розташування. Таким чином, масив з шести елементів буде відсортовано за п’ять кроків.

Для визначення складності алгоритму біля кожного блоку зліва у дужках наведена кількість разів його виконання (наприклад, блок 1 виконується n разів, блок 2 – $(n-1)$ разів).

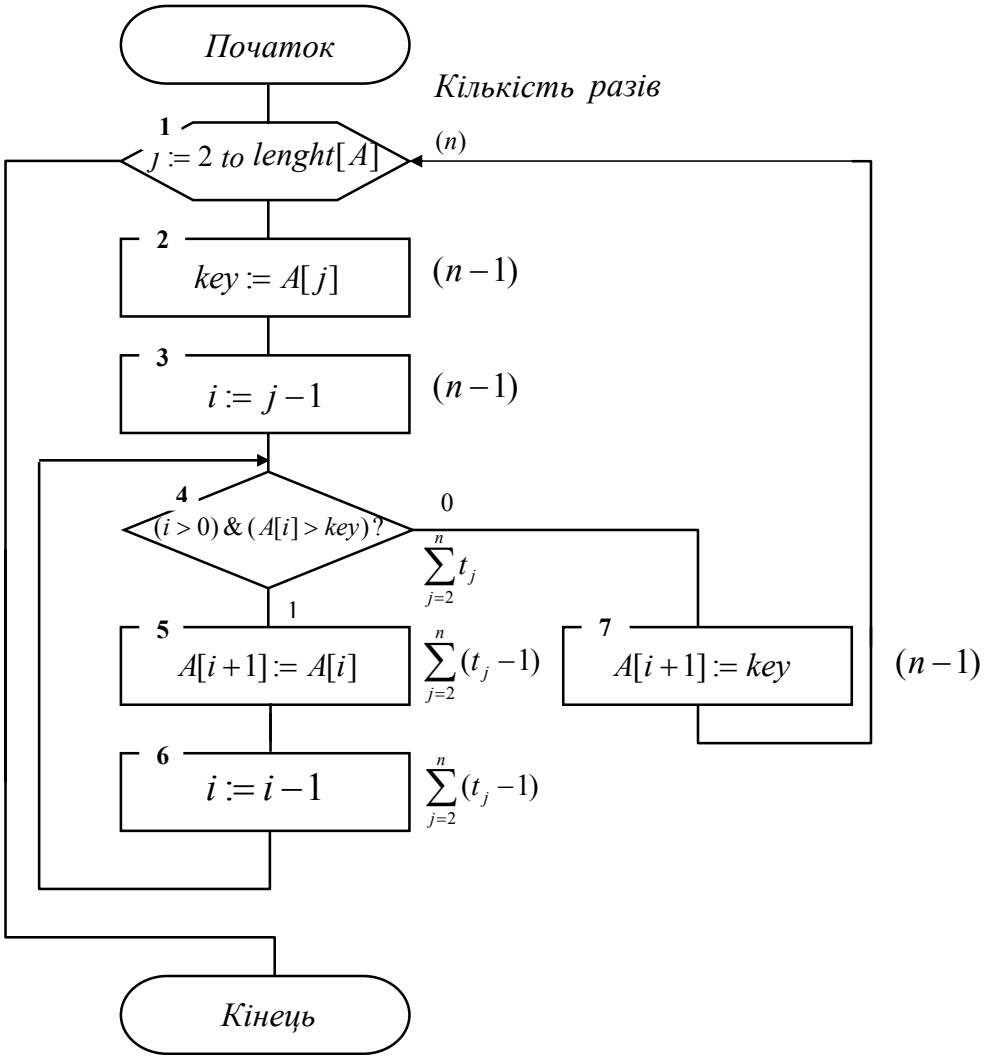


Рисунок 3.1

Аналіз складності алгоритму

Розглядаючи різні алгоритми рішення однієї і тієї ж задачі, корисно проаналізувати, скільки обчислювальних ресурсів вони вимагають (час роботи, пам'ять), і вибрати найефективніший. Крім того зазначимо що тут і нижче використовується звичайний однопроцесорний комп'ютер з довільним доступом, що не передбачає паралельного виконання операцій.

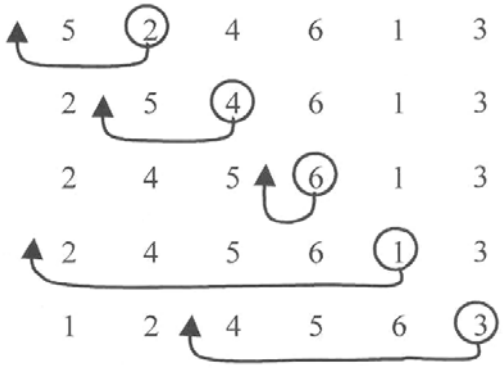


Рисунок 3.2

Очевидно, що час сортування вставками залежить від розміру масиву, що підлягає сортуванню. Крім того для нас важливий не лише розмір масиву, а й порядок його елементів (наприклад, якщо масив майже впорядкований, то часу потрібно менше.)

Час роботи алгоритму визначається кількістю елементарних кроків, які він виконує. Будемо вважати, що один блок алгоритму потребує не більше певного фіксованого числа операцій. Ми також розрізнятимемо виклик процедури (на який йде фіксоване число операцій) і її виконання, яке може бути досить довгим.

Отже, повернемося до нашого алгоритму і відзначимо біля кожного блоку його вартість c_i (i=1, ..., 7), що визначається кількістю операцій, а також число раз, яке цей блок виконується (на рисунку наведено зліва кожного блоку). Для кожного j від 2 до n (тут n=*length*[A] – розмір масиву) підрахуємо, скільки разів буде виконано блок 5, і позначимо це число через t_j. (Нагадаємо, що рядки усередині циклу виконуються на один раз менше, ніж перевірка, оскільки остання перевірка виводить з циклу.)

Блок вартістю c, що виконується m разів, дає внесок cm у загальне число операцій (для кількості використаної пам'яті цього сказати не можна). Склавши внески всіх рядків, одержимо

$$T(n)= c_1n+c_2(n-1)+c_3(n-1)+c_4\sum_{j=2}^{n}t_j + c_5\sum_{j=2}^{n}(t_j-1) + c_6\sum_{j=2}^{n}(t_j-1) + c_7(n-1).$$

Як ми говорили вище, час роботи алгоритму залежить не лише від n, а й від того, який, саме масив розміром n подано йому на вхід. Для нашого алгоритму найсприятливішим є випадок, коли масив вже відсортований. Тоді цикл у блоці 4 завершується після першої ж перевірки (оскільки $A[i] \le key$ при $i=j-1$), тому усі t_j дорівнюють 1, і загальний час

$$T(n)= c_1n+c_2(n-1)+c_3(n-1)+c_4(n-1) + c_7(n-1)=$$

$$=(c_1+c_2+c_3+c_4+c_7)n-(c_2+c_3+c_4+c_7).$$

Таким чином, в найсприятливішому випадку час $T(n)$, необхідний для сортування масиву розміром n, є лінійною функцією від n, тобто має вигляд $T(n)=an+b$ для деяких констант a і b. (Ці константи визначаються вибраними значеннями c_i ($i=1, ..., 8$))

Якщо ж масив відсортований у зворотному (в даному випадку, спадному) порядку – час роботи процедури буде максимальним: кожен елемент $A[j]$ доведеться порівняти зі всіма елементами A[1], ..., A[$j-1$]. При цьому $t_j=j$. Оскільки

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1; \tag{3.1}$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}, \tag{3.2}$$

одержуємо, що у гіршому випадку час роботи процедури дорівнює

$$T(n)= c_1n+c_2(n-1)+c_3(n-1)+c_4\left(\frac{n(n+1)}{2}-1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) +$$

$$+c_7(n-1)=\left(\frac{c_4}{2}+\frac{c_5}{2}+\frac{c_6}{2}\right)n^2+$$

$$+\left(c_1+c_2+c_3+\frac{c_4}{2}-\frac{c_5}{2}-\frac{c_6}{2}+c_7\right)n-(c_2+c_3+c_4+c_7).$$

Тепер функція $T(n)$ – квадратична, тобто має вигляд $T(n)=an^2+bn+c$ (константи a, b і c тут також визначаються значеннями c_1, …, c_7.)

Час роботи у гіршому разі і в середньому

Отже, ми бачимо, що час роботи у гіршому і кращому випадках можуть суттєво розрізнятися. Тому нас цікавитиме саме час роботи у гіршому випадку, яке визначається як максимальний час роботи для входів даного розміру. Ось три основні причини для цього [1].

1. Знаючи час роботи у гіршому разі, ми можемо гарантувати, що виконання алгоритму закінчиться за деякий час, навіть не знаючи, який саме вхід (даного розміру) буде задано.

2. На практиці „несприятливі" входи (для яких час роботи близький до максимуму) можуть часто зустрічатися. Наприклад, для бази даних поганим запитом може бути пошук відсутнього елементу (досить часта ситуація).

3. Час роботи в середньому часто близький до часу роботи у гіршому випадку. Нехай, наприклад, ми сортуємо випадково розташовані n чисел за допомогою алгоритму вставками. Скільки разів доведеться виконати цикл у блоках 4-7? В середньому біля половини елементів масиву $A[1..j-l]$ більші ніж $A[j]$, отже t_j в середньому можна вважати рівним $j/2$, і час $T(n)$ квадратично залежить від n .

Псевдокод

Домовимося надалі для простоти подавати алгоритми не у вигляді схеми, а у вигляді процедури, написаної псевдокодом [1]. Отже, процедура сортування INSERTION-SORT, що відповідає нашому алгоритму (див. рис. 3.1), наведена на рисунку 3.3.

```
INSERTION-SORT(A)
1 for  j← 2 to length[A]
2     do key← A[j]
3        ▷ додати A[j] до відсортованої частини A[l ... j-l]
4        i← j-l
5        while i > 0 and A[i]>key
6              do A[i+1] ← A[i]
7                 i← j-l
8        A[i+1]← key
```

Рисунок 3.3 – Процедура сортування вставками

Наведемо основні домовленості, які ми використовуватимемо [1].

1. Відступ від лівого поля указує на рівень вкладеності. Наприклад, тіло циклу for (рядок 1) складається з рядків 2-8, а тіло циклу while (рядок 5) містить рядки 6-7, але не 8. Це ж правило застосовується і для конструкції if-then-else**.** Це робить зайвим спеціальні команди початку і кінця блоку.

2. Цикли while, for, repeat і умовні конструкції if, then, else мають той же смисл, що й у Паскалі.

3. Символ ▷ починає коментар, що йде до кінця рядка.

4. Одночасне присвоювання r←j←*e.* (змінні r та *j* набувають значення *e*) замінює два присвоювання у порядку *j*←*e,* а потім *i*←*j*.

5. Змінні (в даному випадку *i, j, key)* локальні усередині процедури (якщо не сказано протилежне).

6. Індекс масиву записується у квадратних дужках: *A*[*i*] є *i*-й елемент у масиві *A*.

7. Часто використовуються об’єкти, що складаються з кількох полів (мають кілька атрибутів). Значення поля записується як *ім’я поля*[*ім’я-объекта*]. Наприклад, довжина масиву вважається його атрибутом і позначається *length* (наприклад, довжина масиву *A* позначається *length[A]*). З контексту, звичайно, ясно, що саме означають квадратні дужки (елемент масиву або поле об’єкта).

У псевдокоді може застосовуватися спеціальне значення *NIL*, не вказуючи ні на один об'єкт.

8. Параметри передаються за значенням: викликана процедура одержує власну копію параметрів; зміна параметра усередині процедури зовні є невидимою. При передаванні об’єктів копіюється покажчик на дані, що складають цей об’єкт, а самі поля об’єкту – ні. Наприклад, якщо *x* — параметр процедури, то присвоювання *x*←*y,* виконане усередині процедури, зовні помітити неможливо, а присвоювання *f*[*x*]←*4* – можна.

3.2.2 Сортування методом прямого вибирання

Ідея даного алгоритму полягає в такому.

1. Серед *n* елементів масиву вибирається найменший.

2. Він міняється місцями з першим елементом a_1.

3. Далі цей процес повторюється із рештою $n-1$ елементами, $n-2$ елементами і т. д. доти, поки не залишиться один, найбільший елемент.

Такий алгоритм називають сортуванням за допомогою *прямого вибору.* І він у певному смислі протилежний алгоритму сортування вставками. В останньому випадку на кожному кроці розглядається тільки один черговий елемент вихідної послідовності та всі елементи готової послідовності, серед яких відшуковується точка вставки; при прямому виборі для пошуку *одного* елементу з найменшим ключем проглядаються всі елементи вихідної послідовності і знайдений розташовується як черговий елемент у готову послідовність.

Псевдокод, що дозволяє реалізувати дану ідею, наведений на рисунку 3.4. Тут же наведено вартість кожної операції а також кількість разів її виконання.

Номер	Псевдокод	Вартість	Кількість разів
1	**for** $i \leftarrow 1$ **to** *length*[A]-1	c_1	n
2	**do** $min \leftarrow A[i]$	c_2	$n-1$
3	$N\_min \leftarrow i$	c_3	$n-1$
4	**for** $j \leftarrow i+1$ **to** *length*[A]	c_4	$\sum_{j=i+1}^{n} t_j = \sum_{j=2}^{n} t_j$
5	**do if** $A[i] < min$ **then**	c_5	$\sum_{j=i+1}^{n} (t_j - 1) = \sum_{j=2}^{n} (t_j - 1)$
6	$\min \leftarrow A[N\_min]$	c_6	$\sum_{j=i+1}^{n} (t_j - 1) = \sum_{j=2}^{n} (t_j - 1)$
7	$N\_min \leftarrow j$	c_7	$\sum_{j=i+1}^{n} (t_j - 1) = \sum_{j=2}^{n} (t_j - 1)$
8	$A[N\_min] \leftarrow A[i]$	c_8	$n-1$
9	$A[i] \leftarrow min$	c_9	$n-1$

Рисунок 3.4 – Псевдокод алгоритму сортування прямим вибором

Аналіз складності алгоритму

Як і у випадку сортування вставками, склавши внески всіх рядків отримаємо час виконання в загальному випадку

$$T(n) = c_1 n + (c_2 + c_3 + c_8 + c_9)(n-1) + c_4 \sum_{j=2}^{n} t_j +$$

$$+ c_5 \sum_{j=2}^{n} (t_j - 1) + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1).$$

Оцінимо час роботи в найбільш та найменш сприятливих випадках. Отже, найсприятливіший випадок буде у випадку, коли масив вже відсортовано і блоки 6 і 7 виконуватись не будуть. Тоді формула часу виконання з урахуванням формул (3.1 – 3.2) набуває вигляду

$$T(n) = c_1 n + (c_2 + c_3 + c_8 + c_9)(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + (c_6 + c_7) \cdot 0 =$$

$$= c_1 n + (c_2 + c_3 + c_8 + c_9)(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + c_5 \left(\frac{n(n-1)}{2} \right) =$$

$$= n^2 \left(\frac{c_4}{2} + \frac{c_5}{2} \right) + n \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} + c_8 + c_9 \right) - (c_1 + c_3 + c_4 + c_8 + c_9).$$

Найнесприятливіший випадок має місце коли масив відсортовано у зворотному порядку. Тоді формула визначення часу набуває вигляду

$$T(n)=c_1n+(c_2+c_3+c_8+c_9)(n-1)+c_4\left(\frac{n(n+1)}{2}-1\right)+$$

$$(c_5+c_6+c_7)\left(\frac{n(n-1)}{2}\right)=n^2\left(\frac{c_4+c_5+c_6+c_7}{2}\right)+$$

$$+n\left(c_1+c_2+c_3-\frac{c_4+c_5+c_6+c_7}{2}+c_8+c_9\right)-(c_2+c_3+c_4+c_8+c_9).$$

Отже, даний алгоритм в обох випадках має значний час сортування $O(n^2)$. Перевагами алгоритму є простота, стійкість та відсутність додаткової пам'яті.

3.2.3 Сортування методом бульбашки

У даному розділі ми наведемо метод, у якому обмін місцями двох елементів є характерною особливістю процесу. Викладений нижче алгоритм ґрунтується на порівнянні й зміні місць для пари сусідніх елементів і продовженні цього процесу доти, поки не будуть упорядковані всі елементи (тому даний алгоритм отримав ще назву „сортування прямим обміном"). І такі проходи по масиву слід повторювати, пересуваючи щоразу найменший елемент послідовності, яка залишилась, до лівого кінця масиву. Якщо розглядати масиви як вертикальні, а не горизонтальні, то елементи можна інтерпретувати як бульбашки у чані з водою, причому вага кожного відповідає його ключу [4]. Тут під час кожного проходу одна бульбашка „піднімається" до рівня, що відповідає її вазі (табл. 3.1).

Таблиця 3.1 – Сортування методом бульбашки

прохіду	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8
Елемент	44	6	6	6	6	6	6	6
	55	44	12	12	12	12	12	12
	32	55	44	18	18	18	18	18
	42	12	55	44	42	42	42	42
	94	42	18	55	44	44	44	44
	18	94	42	42	55	55	55	55
	6	18	94	67	67	67	67	67
	67	67	67	94	94	94	94	94

Псевдокод, що реалізує такий підхід у найпростішому вигляді, наведено на рисунку 3.5.

```
BUBBLESORT(A)
1.  for i←2 to n
2.     do  for j←n downto i
3.            do  if A[j-1] > A[j] then
4.                                  виконати обмін A[j]↔A[j-1]
```

Рисунок 3.5 – Псевдокод сортування методом бульбашки

Поліпшення цього алгоритму очевидне. На прикладі у табл. 3.1 бачимо, що три останніх проходи не впливають на порядок елементів, оскільки вони вже відсортовані. Отже, слід запам'ятовувати, мали місце перестановки під час деякого проходу або ні. І якщо у черговому проході перестановок не було – алгоритм можна завершувати.

Зазначимо, що це поліпшення можна знову ж поліпшити, якщо запам'ятовувати не лише факт обміну, а й індекс *k* останнього обміну. Цілком зрозуміло, що усі пари сусідніх елементів вище цього індексу вже відсортовані. Тому перегляд можна закінчувати на цьому індексі, а не йти до заздалегідь визначеної нижньої межі для *i*.

Крім того, тут можна помітити ще деяку своєрідну асиметрію: „спливає" бульбашка на потрібне місце за один прохід, а „тоне" на потрібне місце дуже повільно – лише на один крок за один прохід. Наприклад, масив

12 18 42 44 55 67 94 06

за допомогою удосконаленого бульбашкового сортування можна упорядкувати за один прохід, а для масиву

94 06 12 18 42 44 55 67

потрібно сім проходів. Така асиметрія наводить на думку про третє поліпшення: послідовно чергувати напрямок проходів. Алгоритм, що дозволяє реалізувати таке поліпшення називають *шейкерним сортуванням* (ShakerSort) [4].

Псевдокод, що дозволяє реалізувати алгоритм шейкерного сортування наведено на рисунку 3.6 (тут змінні *L, R, k* – позначають ліву та праву границі ділянки сортованого масиву, а також індекс останнього обміну, відповідно). А таблиця 3.2 ілюструє процес сортування тих же (табл. 3.1) восьми ключів.

Аналіз алгоритмів сортування методом бульбашки та його модифікацій ми пропонуємо виконати читачу самостійно, за аналогією з попередньо розглянутими алгоритмами. Проте, слід зауважити, усі алгоритми, що наведені у розділі 3.2 мають складність порядку $O(n^2)$, оскільки усі вищеперераховані вдосконалення не впливають на кількість переміщень, а лише скорочують кількість зайвих подвійних перевірок. Крім того, обмін місцями двох елементів – найчастіше більш трудомістка операція, ніж порівняння ключів, тому наведені поліпшення не дають значного виграшу [3, 4].

```
SHAKERSORT(A);
1. L←2
2. R←n
3. k←n
4.  repeat
5.    for j←R downto L
6.      do  if A[j-l]>A[j] then
7.                          виконати обмін A[j]↔A[j-1]
8.                          k←j
9.    L←k+l:
10.   for j← L to R
11.     do if A[j-l]>A[j] then
12.                         виконати обмін A[j]↔A[j-1]
13.                         k←j
14.   R←k-1
15. until L←R
```

Рисунок 3.6 – Псевдокод реалізації шейкерного сортування

Таблиця 3.2 – Приклад шейкерного сортування

L	2	3	3	4	4
R	8	8	7	7	4
Напрямок проходу	↑	↓	↑	↓	↑
Елементи	44	6	6	6	6
	55	44	44	12	12
	32	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	6	18	67	67	67
	67	67	94	94	94

Взагалі, шейкерне сортування рекомендується використовувати у випадках, коли відомо, що елементи майже впорядковані, що на практиці зустрічається досить рідко.

Насамкінець зазначимо, що усі вищерозглянуті прийоми сортування на кожному елементарному кроці переміщують кожен елемент на одну позицію і тому вимагають порядку n^2 таких кроків. Отже, в основу будь-яких серйозніших поліпшень повинен бути покладений принцип переміщення на кожному такті елементів на великі відстані.

Далі ми розглядаємо три поліпшених методи сортування, що працюють за псевдолінійний час.

3.3 Алгоритми сортування за псевдолінійий час

Нижче ми розглянемо алгоритми сортування злиттям, за допомогою купи і швидке сортування. Усі вони сортують за псевдологарифмічний час і спираються виключно на попарні порівняння елементів.

Будь-яке сортування такого типу в найгіршому випадку потребує часу $\Omega(n \cdot \log n)$. Тим самим ці алгоритми асимптотично оптимальні, тобто не існує алгоритму сортування порівнянням, який би перевищував вказані алгоритми більше ніж у кінцеву кількість разів [1].

3.3.1 Сортування методом злиття

Існує багато стандартних прийомів, які використовуються під час побудови алгоритмів. Наприклад, сортування вставками застосовує алгоритм, що діє покроково: ми додаємо елементи один за одним до відсортованої частини масиву.

Далі розглянемо інший підхід, який називають *„розділяй і володарюй”*, і побудуємо за його допомогою значно швидший алгоритм сортування.

Принцип „розділяй і володарюй”. Багато алгоритмів за своєю природі рекурсивні, тобто розв’язуючи деяку задачу, вони викликають самих себе для розв’язання її підзадач. Ідея принципу „*розділяй і володарюй*” полягає саме в цьому. Спочатку задача розбивається на декілька підзадач меншого розміру. Потім ці задачі розв’язуються (за допомогою рекурсивного виклику або безпосередньо, якщо розмір досить малий). Нарешті, їх розв’язки комбінуються і ми отримуємо шуканий розв’язок вихідної задачі [1 – 4].

Для задачі сортування ці три етапи виглядають так. Спочатку ми розбиваємо масив на дві половини меншого розміру. Потім ми сортуємо кожну з половин окремо. Після цього нам залишається з’єднати два впорядкованих масиви половинного розміру в один. Рекурсивне розбивання задачі на менші відбувається доти, поки розмір масиву не стане одиничним (будь-який масив розміром 1 можна вважати впорядкованим) [1].

Нетривіальним етапом є з’єднання двох впорядкованих масивів в один. Воно виконується за допомогою допоміжної процедури $\mathrm{MERGE}(A, p, q, r)$. Параметрами цієї процедури є масив A і числа p, r, q, що вказують границі ділянок, які зливаються. Процедура передбачає, що $p \le q < r$, а ділянки $A[p...q]$ та $A[q+1...r]$ вже відсортовані, і зливає їх в одну ділянку $A[p...r]$.

Зрозуміло, що час роботи процедури MERGE є $\theta(n)$, де n – загальна довжина ділянок, що зливаються $n = r - p + 1$. Це легко пояснити на картах. Нехай ми маємо дві стопки карт (сорочкою вниз), і в кожній карти йдуть зверху вниз у порядку зростання. Для об’єднання цих стопок в одну

на кожному кроці ми беремо меншу із двох верхніх карт і кладемо її (сорочкою вверх) у результуючу стопку. Коли одна з вихідних стопок стає порожньою – ми додаємо всі карти, що залишилися другої стопки до результуючої стопки. Отже, кожен крок вимагає обмеженого числа дій, і загальне число дій становить $\theta(n)$ [1].

Тепер напишемо процедуру сортування злиттям MERGE-SORT(A,p,r), яка сортує ділянку $A[p...r]$ масиву A, не змінюючи іншої його частини. При $p \geq r$ ділянка містить максимум один елемент, і, отже є відсортованою. У протилежному випадку ми шукаємо число q, що ділить ділянку на дві приблизно рівні частини $A[p...r]$ (містить $\lceil n/2 \rceil$ елементів) і $A[q+1...r]$ (містить $\lfloor n/2 \rfloor$ елементів). Нагадаємо, що через $\lfloor x \rfloor$ ми позначаємо цілу частину x (найбільше ціле число, яке менше або дорівнює x), а через $\lceil x \rceil$ – найменше ціле число, яке більше або дорівнює x.

MERGE-SORT (A,p,r)
1. **if** $p < r$
2. **then** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGE-SORT (A,p,q)
4. MERGE-SORT $(A,q+1,r)$
5. MERGE (A,p,q,r)

Весь масив тепер можна відсортувати за допомогою виклику MERGESORT $(A,1,length[A])$. Якщо довжина масиву *n=length[A]* є степенем двійки, то у процесі сортування відбудеться злиття пар елементів у відсортовані ділянки довжиною *2*, потім злиття пар таких ділянок у відсортовані ділянки довжиною *4* і так далі до *n* (на останньому кроці з'єднуються дві відсортованих ділянки довжиною *n/2*). Цей процес наведено на рисунку 3.7.

Аналіз алгоритмів типу „розділяй і володарюй". Для того, щоб оцінити час роботи рекурсивного алгоритму ми повинні враховувати час, який витрачається на рекурсивні виклики. Отже, отримаємо деяке *рекурентне співвідношення*, виходячи з якого й можемо оцінити час роботи алгоритму.

Припустимо, що алгоритм розбиває задачу розміром *n* на *a* підзадач, кожна з яких має в *b* разів менший розмір. Будемо вважати, що розбиття потребує часу $D(n)$, а з'єднання отриманих рішень – часу $C(n)$. Тоді одержуємо співвідношення для часу роботи *T(n)* на задачах розміром *n* (у найгіршому випадку):

$$T(n) = aT(n/b) + D(n) + C(n).$$

Це співвідношення виконується для достатньо великих n, коли є смисл розбивати задачу на підзадачі. Для малих n, коли таке розбивання неможливе або не потрібне, застосовується деякий прямий метод розв'язання задачі. Оскільки n обмежене – час роботи також не перевищує деякої константи.

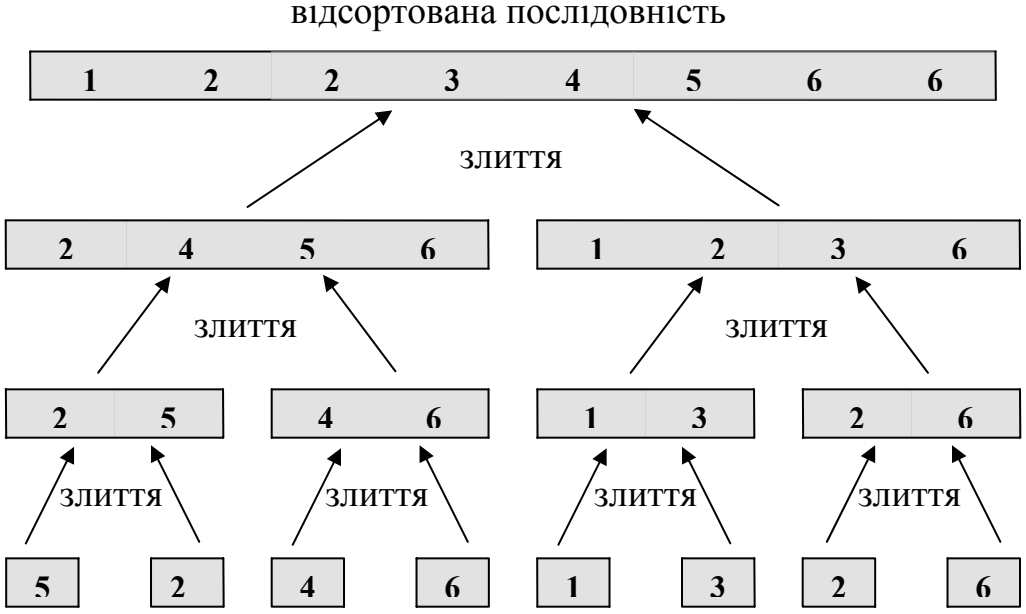


Рисунок 3.7 – Сортування злиттям для масиву $A\{5,2,4,6,1,3,2,6\}$

Аналіз алгоритму сортування злиттям. Для простоти припустимо, що розмір масиву є степінь двійки. Тоді на кожному кроці ділянка, що підлягає сортуванню ділиться на дві рівні половини. Розбиття на частини (обчислення границі) вимагає часу $\theta(1)$, а злиття – часу $\theta(n)$. Таким чином, одержуємо співвідношення

$$T(n)\begin{cases}\theta(1), & якщо\ n=1;\\ 2T(n/2)+\theta(n), & якщо\ n>1.\end{cases} \qquad (3.3)$$

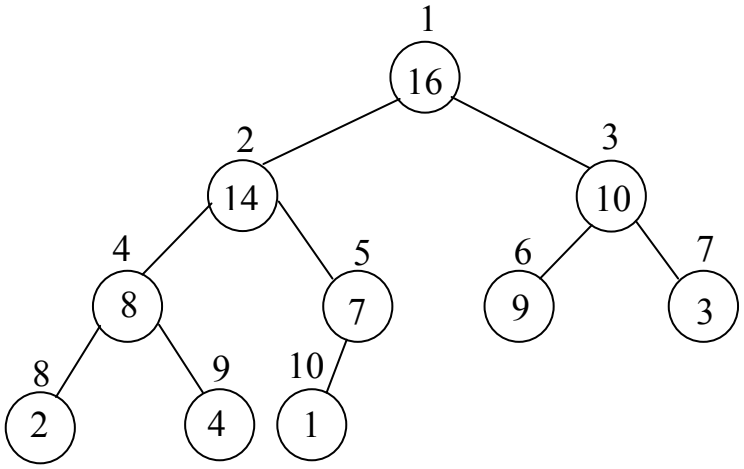
Отже, розв'язуючи дане співвідношення приходимо до такої оцінки: $T(n) = \theta(n \cdot log\, n)$, де через *log* ми позначаємо двійковий логарифм (основа логарифмів не відіграє ролі, тому що приводить лише до зміни константи). Тому для більших n сортування злиттям ефективніше сортування вставками, яке вимагає часу $\theta(n^2)$.

3.3.2 Сортування за допомогою купи

Як і алгоритм сортування злиттям, алгоритм сортування за допомогою купи потребує часу $\theta(n \cdot log\, n)$ для сортування n об'єктів, але потребує додаткову пам'ять розміром $\theta(1)$ замість $\theta(n)$ для сортування злиттям. Таким чином, цей алгоритм містить переваги двох раніше розглянутих алгоритмів – сортування злиттям і сортування вставками [1].

Структура даних, яку використовує алгоритм (вона називається двійковою купою) буває корисною і в інших ситуаціях. Особливо ефективно на її основі можна організовувати чергу з пріоритетами тощо [1].

Двійковою купою називається масив з визначеними властивостями впорядкованості. Щоб сформулювати ці властивості, будемо розглядати масив як двійкове дерево (рис.3.8).



а)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

б)

Рисунок 3.8

Кожна вершина дерева відповідає елементу масива. Якщо вершина має індекс i, то її батько має індекс $\lfloor i/2 \rfloor$, вершина з індексом *1* є коренем, а її дочірні вершини мають індекси $2i$ та $2i+1$. Будемо вважати, що купа може не займати всього масиву, і тому будемо зберігати не тільки масив A і його довжину *length[A]*, а й спеціальний параметр *heap-size[A]* (розмір купи), причому *heap-size[A]*$\leq$ *length[A]*. Купа складається з елементів *A[1], ..., A[heap-size[A]]*.

Рух по дереву здійснюється за допомогою процедур:

```
PARENT(i)          LEFT(i)          RIGHT(i)
    return ⌊i/2⌋;      return 2i;       return 2i + 1.
```

Купу можна розглядати як дерево (рис. 3.8 а) або масив (рис. 3.8 б). Всередині кожної вершини наведено її значення. Біля вершини наведено її індекс у масиві. Елемент *A[1]* є коренем дерева.

В більшості комп'ютерів для виконання процедур LEFT і PARENT можна використовувати команди лівого і правого зсуву, відповідно. Процедура RIGHT потребує лівого зсуву, після якого у молодший розряд записується одиниця.

Елементи, що зберігаються в купі повинні характеризуватися головною властивістю купи: для кожної вершини *i*, крім кореня (при *2≤i≤heap-size[A]},*

$$A\left[PARENT(i)\right] \geq A[i] \qquad (3.4)$$

Звідси випливає, що значення нащадка не перевищує значення батька. Таким чином, найбільший елемент дерева (або будь-якого піддерева) знаходиться у кореневій вершині цього дерева (або піддерева).

Висотою вершини дерева є висота піддерева з коренем в цій вершині (число ребер в найдовшому шляху збігається з початком у цій вершині вниз по дереву до листка). Висота дерева, таким чином, збігається з висотою його кореня. У дереві, що утворює купу, всі рівні (крім останнього) заповнені повністю. Тому висота цього дерева дорівнює $\theta(\log n)$, де n – число елементів купи. Як побачимо нижче, час роботи основних операцій над купою пропорційний висоті дерева і, відповідно, складає $\theta(\log n)$.

Перерахуємо основні операції над купою [1]:

• Процедура HEAPIFY дозволяє підтримувати головні властивості купи. Час її роботи складає $\theta(\log n)$.

• Процедура BUILD-HEAP будує купу з вихідного (невідсортованого) масиву. Час її роботи $\theta(n)$.

• Процедура HEAPSORT сортує масив, не використовуючи допоміжної пам'яті. Час її роботи $\theta(n \cdot \log n)$.

• Процедури EXTRACT-MAX (отримання найбільшого) і INSERT (доповнення елементу) використовується при моделюванні черги з пріоритетами на базі купи. Час роботи обох процедур складає $\theta(\log n)$.

Збереження головної властивості купи. Процедура HEAPIFY – важливий метод роботи з купою [1]. Її параметрами є масив *A* та індекс *i*. Вважається, що піддерева з коренями LEFT(*i*) і RIGHT(*i*) вже мають головні властивості. Процедура переміщує елементи піддерева з вершиною *i*, після чого воно буде мати головну властивість. Ідея проста: якщо головна властивість не виконана для вершини *i*, то її слід поміняти із старшим з її дочірніх вершин і т. д., доти, поки елемент A[*i*] не „завантажиться" до потрібного місця.

Процедура HEAPIFY наведена на рисунку 3.9. У рядках 3 – 7 змінна *largest* набуває значення, що дорівнює індексу найбільшого з елементів

$A[i]$, A[LEFT(i)] та A[RIGHT(i)]. Якщо *largest=i*, то елемент $A[i]$ вже „завантажився" до потрібного місця, і робота процедури закінчена. Інакше процедура міняє місцями $A[i]$ та A[*largest*] (що забезпечує виконання властивості (3.4) у вершині i, але, можливо, порушує цю властивість у вершині *largest*) і рекурсивно викликає себе для вершини *largest* (рядок 10), щоб виправити можливі порушення.

```
HEAPIFY(A, i)
1.  l←LEFT(i)
2.  r←RIGHT(i)
3. if l ≤ (heap-size[A] and  A[l] > A[i])
4.      then largest ← l
5.      else largest ← i
6. if r ≤ (heap-size[A] and A[r] > A[largest])
7.      then largest ← r
8. if largest ≠ i
9.      then  виконати обмін A[i] ↔ A[largest]
10.           HEAPIFY(A, largest)
```

Рисунок 3.9 – Процедура збереження головної властивості купи

Приклад виконання процедури HEAPIFY наведений на рис. 3.10. Робота процедури HEAPIFY(*A,2*) при *heap-size*[*A*]=10 (а). У вершині *i=2* головна властивість порушена. Щоб відновити її, необхідно поміняти місцями $A[2]$ і $A[4]$. Після цього (б) головна властивість порушується у вершині з індексом 4. Рекурсивний виклик процедури HEAPIFY(*A*,*4*) відновлює головну властивість у вершині з індексом *4* шляхом перестановки $A[4] \leftrightarrow A[9]$ (в). Після цього головна властивість виконана для усіх вершин, тому процедура HEAPIFY(*A*,*9*) вже нічого не виконує.

Оцінимо час роботи процедури HEAPIFY. На кожному кроці потрібно провести $\theta(1)$ дій, не враховуючи рекурсивного виклику. Нехай *T(n)* – час роботи для піддерева, що містить n елементів. Якщо піддерево з коренем i складається з n елементів, то піддерева з коренями LEFT(i) та RIGHT(i) містять не більше ніж по $2n/3$ елементів кожен (найгірший випадок – коли останній рівень в піддереві заповнений наполовину). Таким чином,

$$T(n) \le T(2n/3) + \theta(1).$$

З основної теореми про рекурентні оцінки (випадок 2) отримуємо, що $T(n) = \theta(\log n)$. Цю ж оцінку можна отримати так: на кожному кроці ми опускаємось по дереву на один рівень, враховуючи, що висота дерева це $\theta(\log n)$.

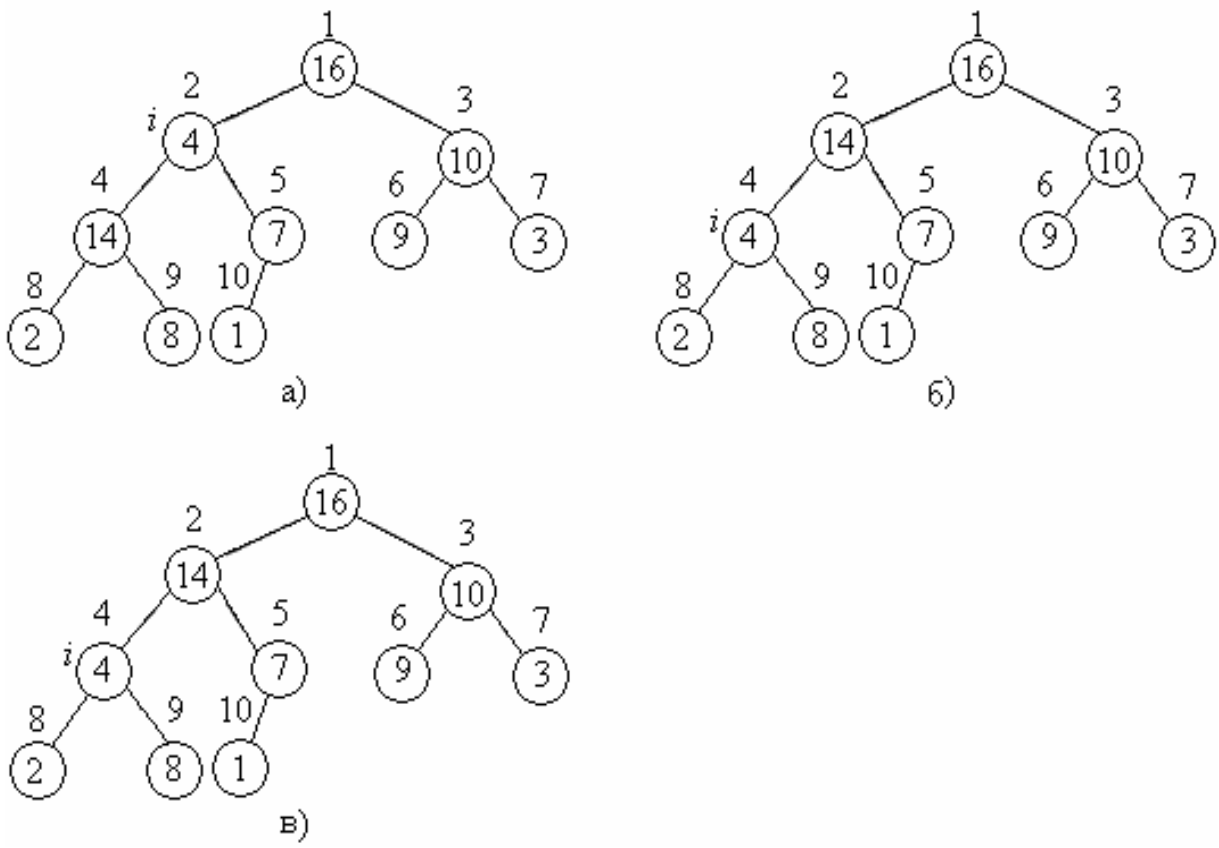


Рисунок 3.10 – Приклад виконання процедури HEAPIFY

Побудова купи. Нехай дано масив $A[1. . n]$, який ми хочемо перетворити на купу, переставивши його елементи. Для цього можна використати процедуру HEAPIFY, застосовуючи її по черзі до усіх вершин, починаючи з нижніх. Оскільки вершини з номерами $\lfloor n/2 \rfloor + 1$, ..., n є листям, піддерева з цими вершинами задовольняють головну властивість. Для кожної з вершин, що залишились, у порядку зменшення індексів, використовуємо процедуру HEAPIFY. Порядок обробки вершин гарантує, що кожен раз умови виклику процедури (виконання головної властивості для піддерев) будуть виконані.

Процедура BUILD-HEAP(A) наведена на рисунку 3.11

BUILD-HEAP (A)
1. *heap-size[A]* ← *length[A]*
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. **do** HEAPIFY (A, i)

Рисунок 3.11 – Процедура побудови купи

Приклад роботи даної процедури наведено на рис. 3.12, де показано стан даних перед кожним викликом процедури HEAPIFY у рядку 3.

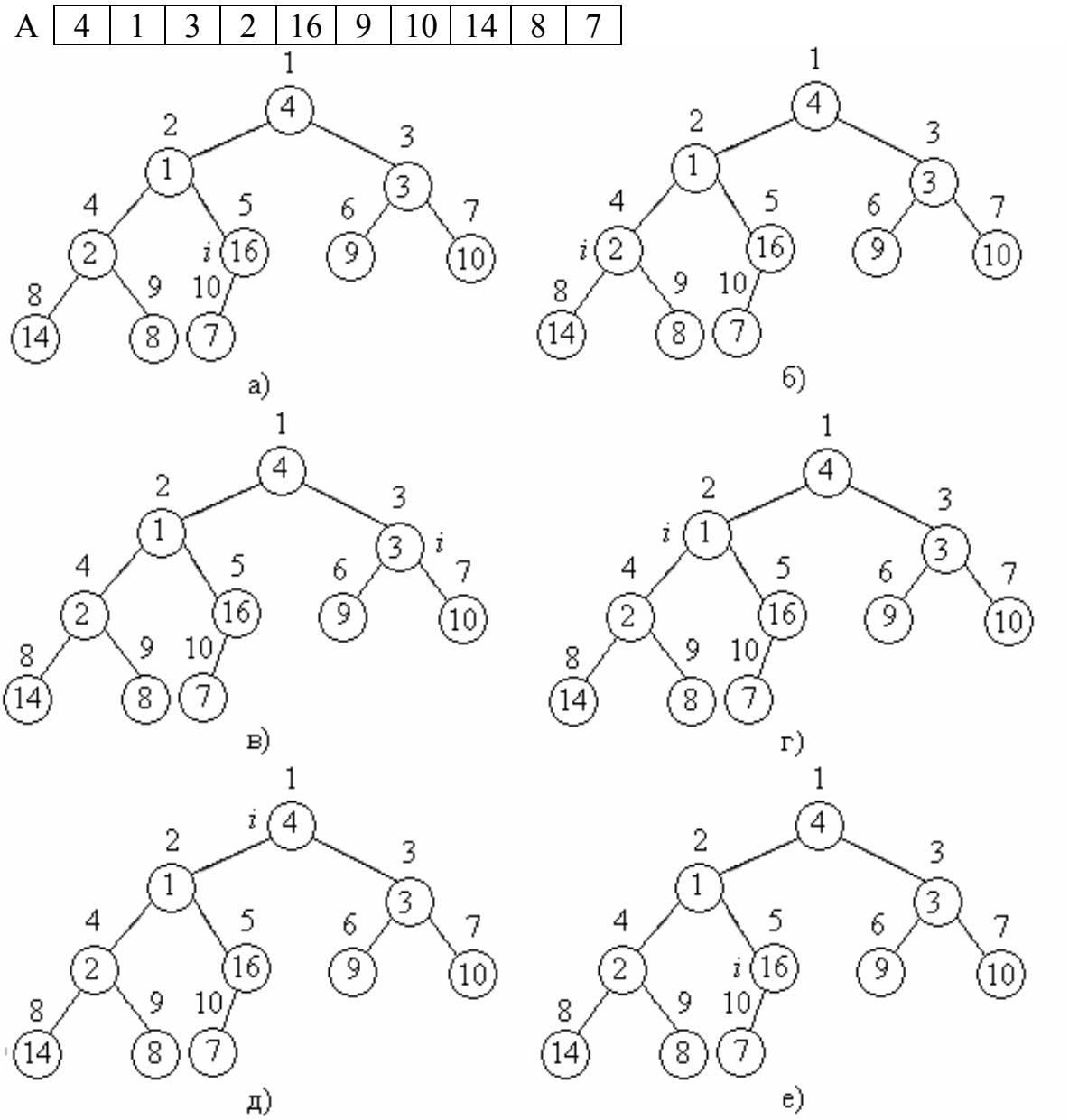


Рисунок 3.12 Робота процедури BUILD-HEAP

Зрозуміло, що час роботи процедури BUILD-HEAP не перевищує $\theta(n \cdot log\, n)$. Дійсно, процедура HEAPIFY викликається $\theta(n)$ раз, а кожне її виконання потребує часу $\theta(log\, n)$. Однак цю оцінку можна покращити.

Справа в тому, що час роботи процедури HEAPIFY залежить від висоти вершини, для якої вона викликається (*i* пропорційна цій висоті). Оскільки число вершин висотою *h* в купі з *n* елементів не перевищує $\lceil n / 2^{h+1} \rceil$, а висота всієї купи не перевищує $\lfloor logn \rfloor$, час роботи процедури BUILD-HEAP не перевищує

$$\sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{n+1}} \right\rceil \theta(h) = \theta\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right). \qquad (3.5)$$

Припускаючи, що x=1/2, отримуємо верхню оцінку для суми у правій частині

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

Таким чином, час роботи процедури BUILD-HEAP складає:

$$\theta\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = \theta(n).$$

Алгоритм сортування за допомогою купи. Алгоритм сортування за допомогою купи складається з двох частин. Спочатку викликається процедура BUILD-HEAP, після виконання якої масив стає купою. Ідея другої частини така: максимальний елемент масиву тепер знаходиться у корені дерева $A[1]$. Його слід поміняти з елементом $A[n]$, зменшити розмір купи на 1 і відновити головну властивість у кореневій вершині (оскільки піддерева з коренями LEFT(1) і RIGHT(1) не втратили головної властивості купи, це можна зробити з допомогою процедури HEAPIFY. Після цього в корені буде знаходитися максимальний з елементів, що залишився. Так робиться до тих пір, поки в купі не залишиться всього один елемент [1].

HEAPSORT(*A*)
1. BUILD-HEAP (*A*)
2. **for** *i* ← *length*[*A*] downto 2
3. **do** замінити *A*[*1*]↔ *A*[*i*]
4. *heap-size*[*A*] ←*heap-size*[*A*] - 1
5. HEAPIFY(*A*, 1)

Робота другої частини алгоритму показана на рис. 3.13. Тут зображенні стани купи перед кожним виконанням циклу for (рядок 2), а заштриховані елементи вже не входять до купи. Час роботи процедури HEAPSORT складає $\theta(n \cdot \log n)$. Дійсно, перша частина (побудова купи) потребує часу $\theta(n)$, а кожне з *n-1* виконань циклу for потребує час $\theta(\log n)$.

Насамкінець доцільно зазначити, що на практиці алгоритм сортування за допомогою купи не є самим швидким – як правило, швидке сортування, яке ми розглянемо нижче, працює швидше. Однак сама купа як структура даних часто виявляється корисною. [1].

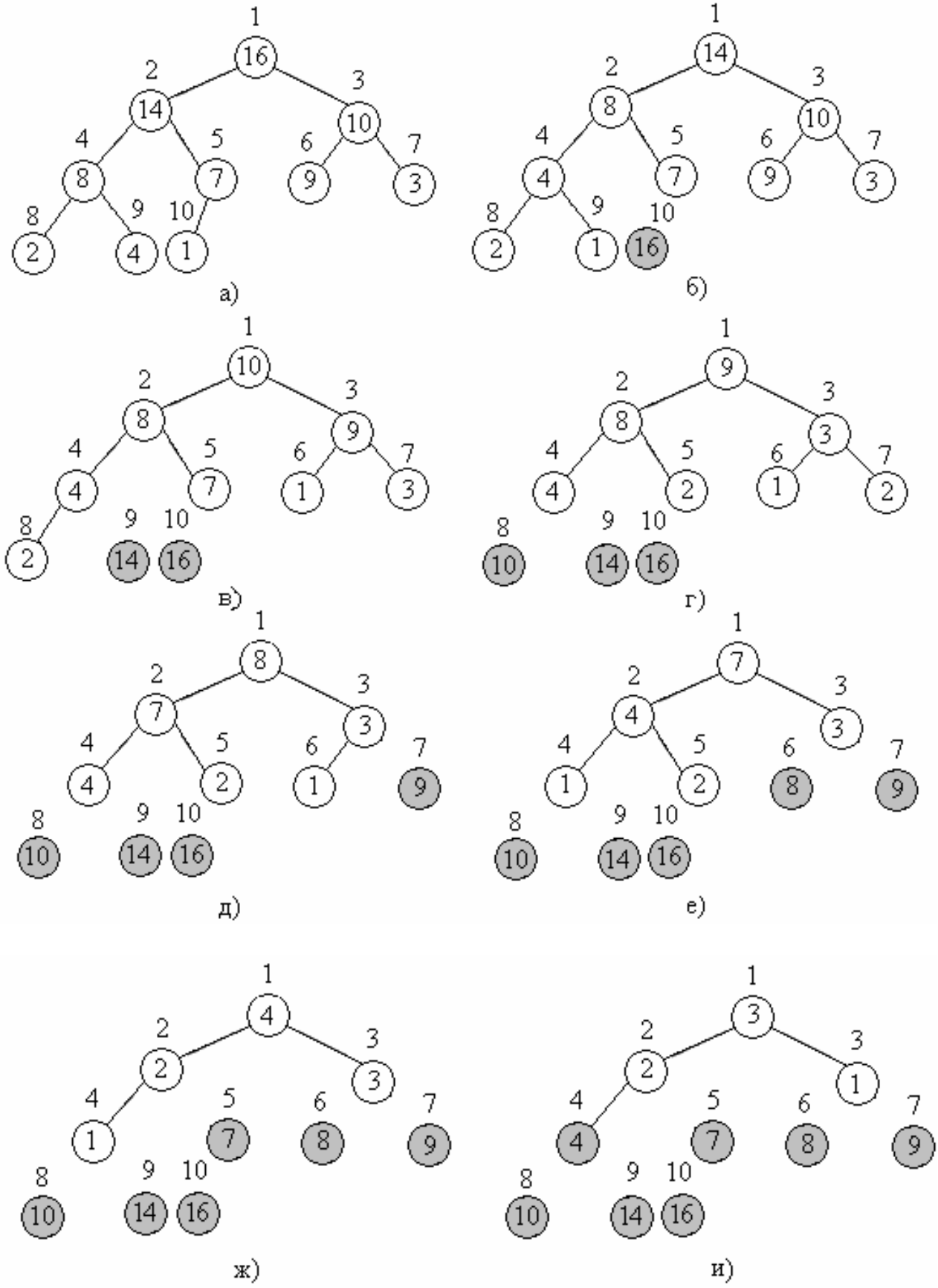
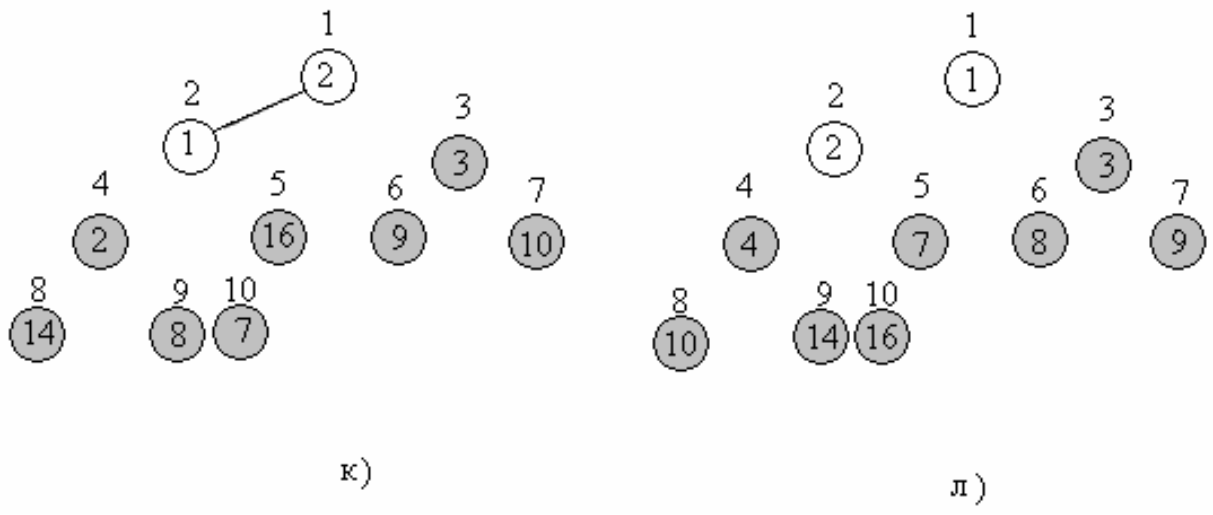


Рисунок 3.13 – Приклад сортування за допомогою купи



A	1	2	3	4	7	8	9	10	14	16

м)

Рисунок 3.13 – (продовження)

3.3.3 Швидке сортування

Даний алгоритм ще називають алгоритмом Хоора. Хоча час його роботи для масиву з n чисел у найгіршому разі дорівнює $\theta(n^2)$, на практиці він є одним з найшвидших: математичне очікування часу роботи становить $\theta(n \cdot log\, n)$. При цьому множник при $n \cdot log\, n$ досить малий. Крім того, швидке сортування не вимагає додаткової пам'яті та зберігає ефективність для систем з віртуальною пам'яттю.

Швидке сортування, як і сортування злиттям, ґрунтується на принципі „розділяй і володарюй". Сортування ділянки $A[p...r]$ здійснюється так [1]:

а) елементи масиву A переставляються так, щоб кожний з елементів $A[p]$, ..., $A[q]$ був не більшим за кожний з елементів $A[q+1]$, ..., $A[r]$, де q – деяке число в інтервалі $p \le q < r$. Цю операцію ми будемо називати поділом;

б) процедура сортування рекурсивно викликається для масивів $A[p...q]$ і $A[q+1...r]$;

в) після цього масив $A[p...r]$ відсортований.

Процедура сортування QUICKSORT псевдокодом наведена на рисунку 3.14. Для сортування всього масиву слід виконати процедуру QUICKSORT$(A,1,length[A])$.

Поділ масиву

Основний крок алгоритму – процедура PARTITION, що переставляє

елементи масиву $A[p...r]$ потрібним чином. Псевдокод цієї процедури наведено на рисунку 3.15.

QUICKSORT (A, p, r)
1. **if** $p < r$
2. **then** q←PARTITION(A, p, r)
3. QUICKSORT (A, p, q)
4. QUICKSORT $(A, q+1, r)$

Рисунок 3.14 – Швидке сортування

PARTITION (A, p, r)
1. $x \leftarrow A[p]$
2. $i \leftarrow p-1$
3. $j \leftarrow p-1$
4. **while** TRUE
5. **do repeat** $j \leftarrow j-1$
6. **until** $A[j] \le x$
7. **repeat** $i \leftarrow i-1$
8. **until** $A[j] \le x$
9. **if** $i < j$
10. **then** виконати заміну $A[i] \leftrightarrow A[j]$
11. **else return** j

Рисунок 3.15 – процедура PARTITION

Приклад роботи процедури PARTITION показаний на рис. 3.16. Елемент $x = A[p]$ вибирається як „граничний”; масив $A[p...q]$ буде містити елементи, не більші ніж x, а масив $A[q+1...r]$ – елементи, не менші ніж x. Ідея полягає в тому, щоб накопичувати елементи, не більші x, у початковому відрізку масиву $(A[p...i])$, а елементи, не менші x – у кінці $(A[j...r])$. На початку обидва „накопичувачі” порожні: $i = p-1, \quad j = r+1$.

Всередині циклу **while,** у рядках 5-8 (рис. 3.15), до початкової й кінцевої ділянок приєднуються елементи (як мінімум по одному). Після виконання цих рядків $A[i] \ge x \ge A[j]$. Якщо ми поміняємо $A[i]$ і $A[j]$ місцями, то їх можна буде приєднати до початкової й кінцевої ділянок.

У момент виходу із циклу виконана нерівність $i \ge j$. При цьому масив розбитий на частини $A[p], ..., A[j]$ і $A[j+1], ..., A[r]$; будь-який елемент першої частини не є більшим будь-якого елемента другої частини.

Процедура повертає значення j.

Повертаючись до рисунку 3.16 зазначимо, що незафарбовані елементи відносяться до вже сформованих кусків, а зафарбовані ще не розподілені. На рис. 3.16,а наведено вихідний стан масиву, початковий і кінцевий куски порожні. Елемент $x = A[p] = 5$ використовується як граничний. На рис. 3.16,б ілюструється результат першого проходу циклу while (рядки 4 – 8). На рис. 3.16,в – елементи $A[i]$ і $A[j]$ міняються місцями (рядок 10). На рис. 3.16,г показано результат другого проходу циклу while, а на рис. 3.16,д – результат третього (останнього) проходу циклу while. Оскільки $i \geq j$, процедура зупиняється й повертає значення $q=j$. Елементи зліва від $A[j]$ (включаючи і сам цей елемент) не більші, ніж x=5, а елементи праворуч від $A[j]$ не менші, ніж x=5.

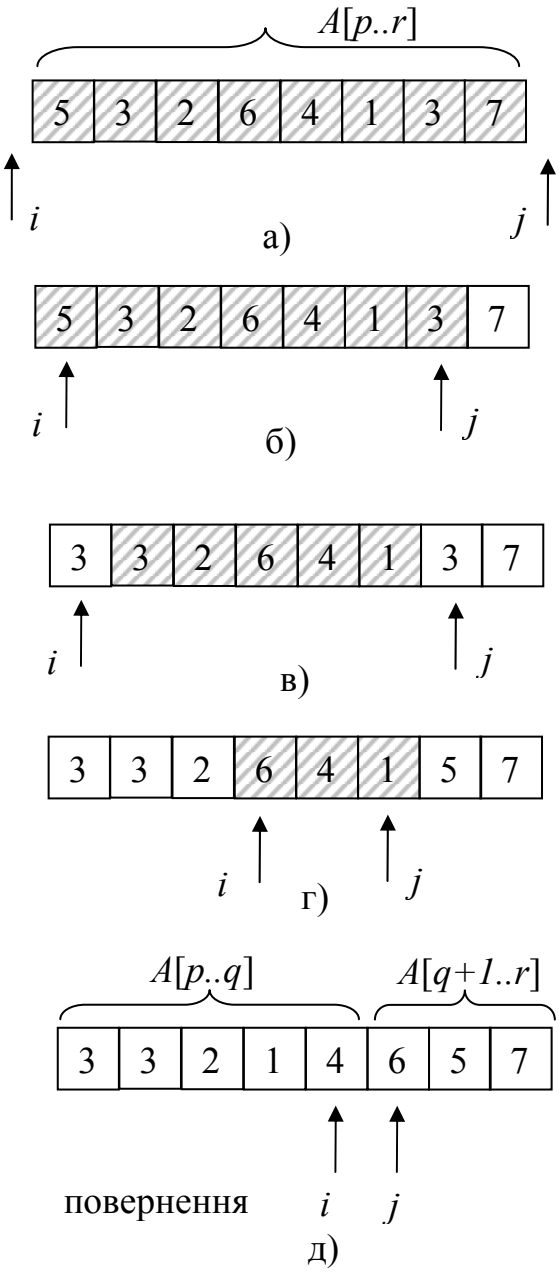


Рисунок 3.16 – Приклад роботи процедури PARTITION

Хоча ідея процедури дуже проста, алгоритм має ряд проблем. Наприклад, не можна чітко визначити, що індекси i та j не виходять за межі проміжку від p до r у процесі роботи. Інший приклад: важливо, що як граничне значення вибирається $A[p]$, а не скажімо, $A[r]$. В останньому випадку може виявитися, що $A[r]$ – найбільший елемент масиву, і наприкінці виконання процедури буде $i = j = r$, так що повертати $q = j$ буде неможливо – порушиться вимога $q < r$, і процедура QUICKSORT зациклиться.

Час роботи процедури PARTITION становить $\theta(n)$, де $n = r - p + 1$ [1].

Робота швидкого сортування

Час роботи алгоритму швидкого сортування залежить від того, як розбивається масив на кожному кроці. Якщо такий поділ відбувається на приблизно рівні частини, час роботи становить $\theta(n \cdot \log n)$, як і для сортування злиттям. Якщо ж розміри частин суттєво відрізняються, сортування може

потребувати час $\theta(n^2)$, як при сортуванні вставками.

Найгірший поділ. „Найбільш нерівні частини" отримаються, якщо одна частина містить $n-1$ елемент, а друга – лише один. Припустимо, що на кожному кроці відбувається саме так. Оскільки процедура поділу потребує часу $\theta(n)$, для часу роботи $T(n)$ одержуємо відношення

$$T(n) = T(n-1) + \theta(n).$$

Оскільки $T(1) = \theta(1)$, маємо

$$T(n) = T(n-1) + \theta(n) = \sum_{k=1}^{n} \theta(k) = \theta\left(\sum_{k=1}^{n} k\right) = \theta(n^2),$$

де остання сума – арифметична прогресія. Дерево рекурсії для цього випадку показане на рис. 3.17.

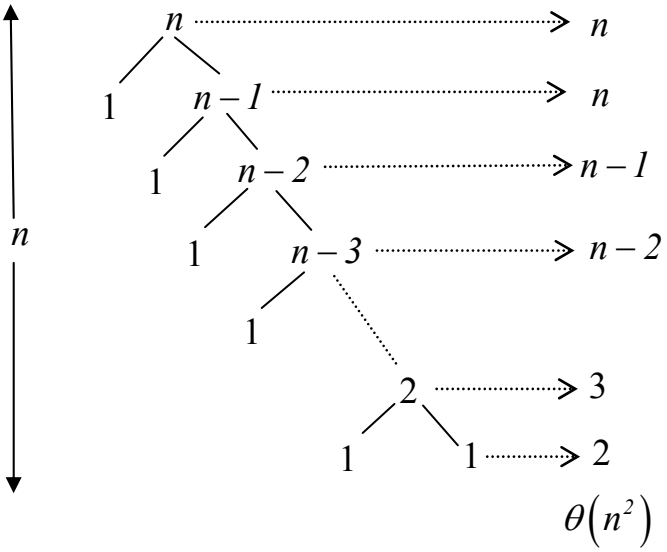


Рисунок 3.17 – Дерево рекурсії для співвідношення $T(n) = T(n-1) + \theta(n)$

Ми бачимо, що при максимально незбалансованому поділі час роботи становить $\theta(n^2)$, як і для сортування вставками. Зокрема, це відбувається, якщо масив спочатку відсортований (у цьому випадку сортування вставками виробляється за час $\theta(n)$).

Найкращий поділ. Якщо на кожному кроці масив розбивається рівно навпіл, швидке сортування працює набагато швидше. Дійсно, у цьому випадку рекурентне співвідношення має вигляд $T(n) = 2T(n/2) + \theta(n)$
і відповідно основній теоремі про рекурентні співвідношення (випадок 2) $T(n) = \theta(n \cdot \log n)$. Дерево рекурсії для цього випадку показано на рисунку 3.18.

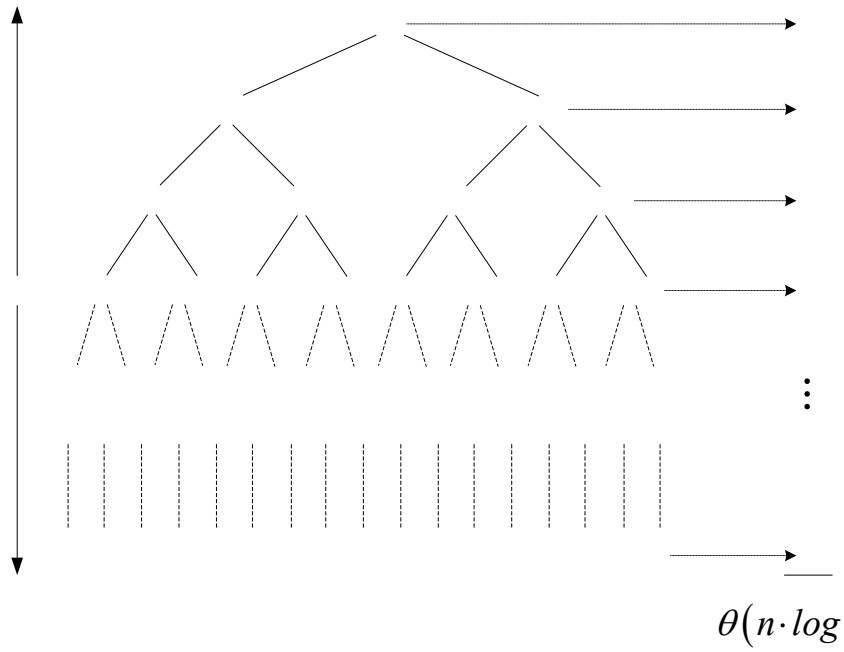


Рисунок 3.18 – Дерево рекурсії для співвідношення $T(n) = 2T(n/2) + \theta(n)$

Проміжний випадок. Середній час роботи (з точністю до множника) збігається із часом роботи в найкращому випадку. Щоб пояснити це, подивимося, як змінюється рекурентне співвідношення залежно від ступеня збалансованості поділу.

Нехай, наприклад, на кожному кроці масив розбивається на дві частини з відношенням розмірів 9:1. Тоді $T(n) = T(9n/10) + T(n/10) + n$ (для зручності ми замінили $\theta(n)$ на n). Дерево рекурсії наведено на рис. 3.19.

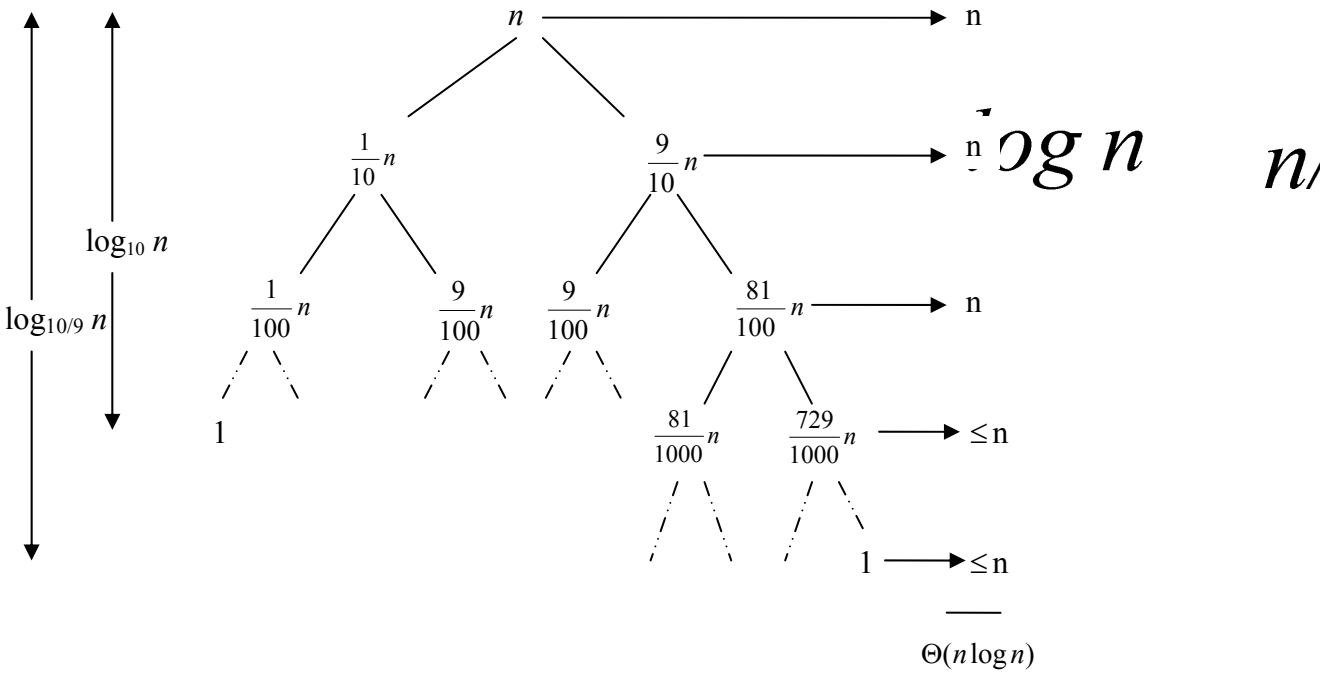


Рисунок 3.19 – Дерево рекурсії для співвідношення $T(n) = T(9n/10) + T(n/10) + n$

На кожному рівні ми робимо не більше n дій, так що час роботи визначається глибиною рекурсії. У даному випадку ця глибина дорівнює $\log_{10/9} n = \Theta(\log n)$, так що час роботи як і раніше складає $\Theta(n \cdot \log n)$, хоча константа і більша. Зрозуміло, що для будь-якого фіксованого відношення розмірів частин (яке б велике воно не було) глибина дерева рекурсії як і раніше буде логарифмічною, а час роботи буде дорівнювати $\Theta(n \cdot \log n)$.

Середній час: інтуїтивні міркування

Щоб питання про середній час роботи мало смисл, слід уточнити, з якою частотою з'являються різні вхідні значення. Як правило, передбачається, що всі перестановки вхідних значень рівноймовірні.

Для довільно взятого масиву поділ навряд чи буде завжди відбуватися в однаковому відношенні – скоріше частина поділів буде добре збалансована, а частина – ні. Як стверджують автори роботи [1] приблизно 80% поділів здійснюються у відношенні не більше ніж 9:1.

Припустимо для простоти, що на кожному другому рівні всі поділи найгірші, а на половині рівнів, що залишилися – найкращі (приклад показано на рис. 3.20,а). Оскільки після кожного „гарного" поділу розмір частин зменшується вдвічі – кількість „гарних" рівнів дорівнює $\Theta(\log n)$, а оскільки кожен другий рівень „гарний" – загальне число рівнів дорівнює $\Theta(\log n)$, а час роботи – $\Theta(n \cdot \log n)$,. Таким чином, „погані" рівні не „зіпсували" асимптотику часу роботи (а лише збільшили константу, сховану в асимптотичному позначенні).

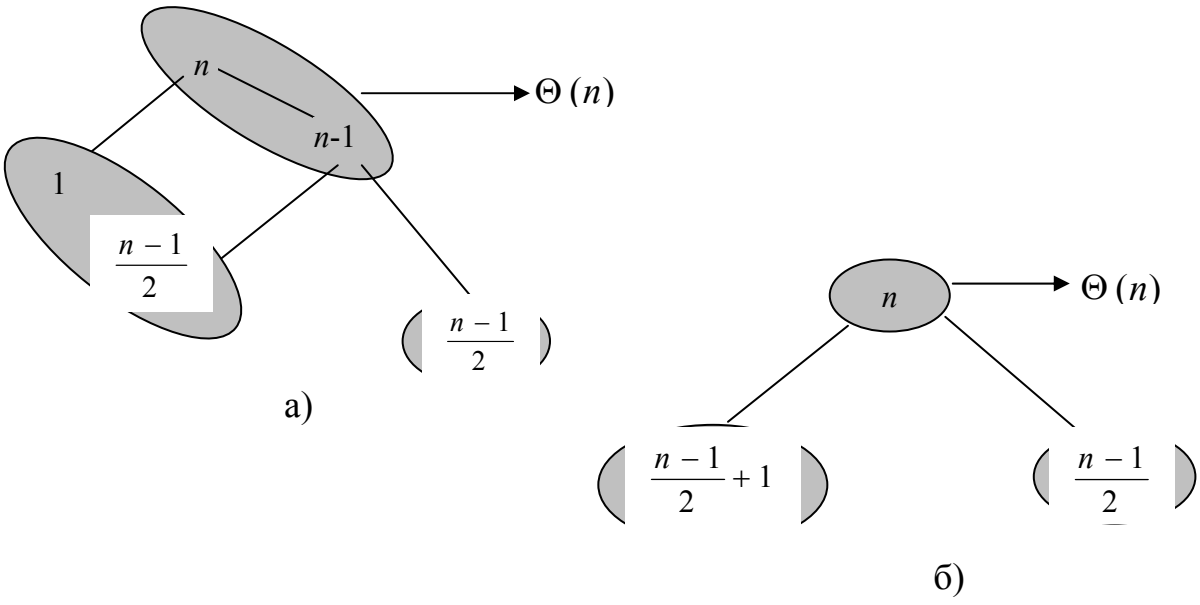


Рисунок 3.20 – а) два рівні: „поганий" *(n* розбивається на *n-1* і *1*) та „гарний" (*n-1* розбивається на дві рівні частини), б) якщо ці два рівні замінити одним, вийде поділ на майже рівні за величиною частини

3.3.4 Ймовірнісні алгоритми швидкого сортування

У пункті 3.3.3 ми припускали, що всі перестановки вхідних значень рівноймовірні. Якщо це так, а розмір масиву досить великий, швидке сортування – один з найефективніших алгоритмів. На практиці, однак, це припущення (рівної ймовірності всіх перестановок на вході) не завжди виправдане [1]. У цьому розділі ми введемо поняття ймовірнісного алгоритму і розглянемо два ймовірнісні алгоритми швидкого сортування, які дозволяють відмовитися від припущення про рівну ймовірність усіх перестановок.

Ідея полягає у привнесенні випадковості, що забезпечує потрібний розподіл. Наприклад, перед початком сортування можна випадково переставити елементи, після чого вже всі перестановки стануть рівноймовірними (це можна зробити за час $O(n)$ [1]). Така модифікація не істотно збільшує час роботи, але тепер математичне сподівання часу роботи не залежить від порядку елементів у вхідному масиві (вони все рівно випадково переставляються).

Алгоритм називається ймовірнісним, якщо він використовує генератор випадкових чисел. Ми будемо вважати, що генератор випадкових чисел RANDOM працює так: RANDOM(a, b) повертає з рівною ймовірністю будь-яке ціле число в інтервалі від a до b. Наприклад, RANDOM(*0,1*) повертає *0* або *1* з ймовірністю *1/2*. При цьому різні виклики процедури незалежні в смислі теорії ймовірностей. Можна вважати, що ми кожного разу кидаємо гральну кістку з (b-a+1) гранями і повідомляємо номер грані, що випала.

Для такого ймовірнісного варіанта алгоритму швидкого сортування немає „незручних входів”, тобто перестановок, при яких час роботи великий, зовсім мало – тому ймовірність того, що алгоритм буде працювати довго (для будь-якого конкретного входу) невелика [1].

Аналогічний підхід застосуємо й в інших ситуаціях, коли у процесі виконання алгоритму ми повинні вибрати один з багатьох варіантів його продовження, причому ми не знаємо, які з них „гарні”, а які „погані”, але знаємо, що „гарних” варіантів досить багато. Потрібно тільки, щоб „погані” вибори не руйнували досягнутого при попередніх „гарних”.

Замість того, щоб попередньо переставляти елементи масиву, ми можемо внести елемент випадковості у процедуру PARTITION. Саме, перед поділом масиву $A[p..r]$ будемо змінювати елемент $A[p]$ з випадково обраним елементом масиву. Тоді кожен елемент із рівною ймовірністю може виявитися граничним, і в середньому поділи будуть виходити досить збалансованими.

Цей підхід заміняє разову випадкову перестановку входів на початку використання випадкових виборів на всьому протязі роботи алгоритму. По суті, це те ж саме, і обидва алгоритми мають математичне сподівання часу

роботи $O(n \cdot logn)$, але невеликі технічні розходження роблять аналіз нового варіанта простішим.

Зміни, які потрібно внести в процедури, зовсім невеликі:

RANDOMIZED-PARTITION(*A,p,r*)
1. *i*←RANDOM (*p,r*)
2. виконати заміну *A*[*p*]↔*A*[*i*]
3. **return** PARTITION(*A,p,r*)

В основній процедурі тепер буде використовуватися RANDOMIZED-PARTITION замість PARTITION:

RANDOMIZED-QUICKSORT(*A,p,r*)
1. **if** *p*<*r*
2. **then** *q*←RANDOMIZED-PARTITION(*A,p,r*)
3. RANDOMIZED-QUICKSORT(*A,p,q)*
4. RANDOMIZED-QuiCKSORT(*A,q+1,r*)

Аналіз швидкого сортування

У цьому розділі ми перетворимо „інтуїтивні" міркування, наведені у пункті 3.3.3 у строге міркування. Спочатку ми розглянемо найгірший випадок (міркування будуть однакові і для алгоритму QUICKSORT, і для алгоритму RANDOMIZED-QUICKSORT), а потім знайдемо середній час роботи алгоритму RANDOMIZED-QUICKSORT.

Аналіз найгіршого випадку

У розділі 3.3.3 ми бачили: якщо поділ на кожному кроці максимально незбалансований, то час роботи складає $\Theta(n^2)$. Інтуїтивно зрозуміло, що це найгірший за витраченим часом випадок. Для доведення того, що час роботи має квадратичну залежність використовуємо метод підстановки [1]. Нехай $T(n)$ – найбільший час роботи алгоритму для масиву довжиною n. Тоді,

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) \qquad (3.6)$$

(ми розглядаємо всі можливі поділи на першому кроці). Припустимо, що $T(q) \le cq^2$ для деякої константи c і для всіх q, менших деякого n. Тоді

$$T(n) = \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) = c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n).$$

Квадратний тричлен $q^2+(n-q)^2$ досягає максимуму на відрізку $1 \le q \le n-1$ у його кінцях (друга похідна q додатна, тому функція опукла вниз). Цей максимум дорівнює $1^2+(n-1)^2=n^2-2(n-1)$. Звідси одержуємо

$$T(n) = \max_{1 \le q \le n-1} (cn^2 - 2c(n-1) + \Theta(n) \le cn^2,$$

якщо константа c обрана так, щоб останній доданок був меншим ніж

передостанній. Отже, час роботи в гіршому випадку складає $\Theta(n^2)$.

Аналіз середнього часу роботи

Як ми вже бачили в пункті 3.3.3, якщо поділи виконуються так, що відношення розмірів частин обмежені, то глибина дерева рекурсії дорівнює $\Theta(\log n)$, а час роботи – $\Theta(n \cdot \log n)$. Щоб одержати оцінку середнього часу роботи алгоритму RANDOMIZED-QUICKSORT, спочатку проаналізуємо роботу процедури PARTITION, а потім одержимо рекурентне співвідношення, що виражає середній час роботи і розв'яжемо його.

Аналіз поділів

Нагадаємо, що перед тим як у рядку 3 процедури RANDOMIZED-PARTITION викликається процедура PARTITION, елемент A[p] переставляється з випадково обраним елементом масиву $A[p..r]$ Для простоти будемо припускати, що всі числа масиву різні (хоча оцінка середнього часу зберігається й у тому випадку, коли у масиві є однакові елементи, одержати її складніше і ми цього робити не будемо).

Насамперед, зазначимо, що значення q, яке поверне процедура PARTITION, залежить тільки від того, скільки у масиві елементів, не більших, ніж $x=A[p]$ (кількість таких елементів будемо називати рангом елемента x і позначати rank(x)). Якщо $n=r-p+1$ – кількість елементів у масиві, то, оскільки всі елементи мають рівні шанси потрапити на місце $A[p]$ – усі значення rank(x), від 1 до n, рівноймовірнісні (мають імовірність $1/n$).

Якщо rank(x)>1, то при виконанні поділу ліва частина буде містити rank(x)-1 елементів і в ній будуть усі елементи менші x. Якщо ж rank(x)=1, то ліва частина буде містити один елемент (після першого ж виконання циклу буде $i=j=p$). Звідси випливає, що з ймовірністю $1/n$ ліва частина буде містити $2, 3, ..., n-1$ елементів, а з ймовірністю $2/n$— один елемент.

Рекурентне співвідношення для середнього часу роботи

Позначимо середній час роботи алгоритму RANDOMIZED-QUICKSORT для масиву з n елементів через $T(n)$. Зрозуміло, що $T(1)=\Theta(1)$. Час роботи складається з часу роботи процедури PARTITION, який складає $\Theta(n)$, і часу роботи для двох масивів розміром q та $n-q$, причому q з ймовірністю $2/n$ приймає значення 1 і з ймовірністю $1/n$ – значення 2, ..., $n-1$. Тому

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} \left(T(q) + T(n-q) \right) \right) + \Theta(n) \qquad (3.7)$$

(доданок, що відповідає q=1, входить двічі). Оскільки $T(1)=\Theta(1)$ і $T(n)=O(n^2)$, маємо

$$\frac{1}{n}\left(T(1) + T(n-1) \right) = \frac{1}{n}\left(\Theta(1) + O(n^2) \right) = O(n).$$

Тому $T(1)$ і $T(n-1)$ у перших дужках рівняння (3.7) можна включити в $\Theta(n)$.

З урахуванням цього одержуємо

$$T(n)=\frac{1}{n}\sum_{q=1}^{n-1}\big(T(q)+T(n-q)\big)+\Theta(n). \tag{3.8}$$

Оскільки кожен доданок $T(k)$, де $k = 1, ..., n-1$, зустрічається в сумі двічі, її можна переписати так:

$$T(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n). \tag{3.9}$$

Розв'язання рекурентного співвідношення

Співвідношення (3.9) можна розв'язати, використовуючи метод підстановки. Припустимо, що $T(n)\le an\cdot\log n+b$, де константи $a>0$ і $b>0$ поки невідомі, і спробуємо довести це за індукцією. При $n=1$ це правильно, якщо взяти досить великі a і b. При $n>1$ маємо

$$T(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n)\le\frac{2}{n}\sum_{k=1}^{n-1}(ak\cdot\log k+b)+\Theta(n)=$$

$$=\frac{2a}{n}\sum_{k=1}^{n-1}k\log k+\frac{2b}{n}(n-1)+\Theta(n).$$

Нижче ми покажемо, що першу суму можна оцінити так [1]:

$$\sum_{k=1}^{n-1}k\log k\le\frac{1}{2}n^2\log n-\frac{1}{8}n^2. \tag{3.10}$$

Використовуючи це, одержимо

$$T(n)=\frac{2a}{n}\left(\frac{1}{2}n^2\log n-\frac{1}{8}n^2\right)+\frac{2b}{n}(n-1)+\Theta(n)\le an\cdot\log n-\frac{a}{4}n+2b+\Theta(n)=$$

$$=an\cdot\log n+b+\left(\Theta(n)+b-\frac{a}{4}n\right)\le an\cdot\log n+b,$$

якщо обрати a так, щоб $\frac{a}{4}n$ було більше $\Theta(n)+b$. Отже, середній час роботи є $O(n\cdot\log n)$.

Доведення оцінки для суми

Залишилося довести оцінку (3.10). Оскільки кожен доданок не перевищує $n\cdot\log n$, одержуємо оцінку

$$\sum_{k=1}^{n-1}k\log k\le n^2\log n$$

Але нам необхідна більш точна оцінка $\frac{1}{2}n^2\log n-\Omega(n^2)$.

Якщо у попередній оцінці заміняти лише $\log k$ на $\log n$, не торкаючись k, одержимо оцінку

$$\sum_{k=1}^{n-1} k \cdot \log k \le \log n \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} \log n \le \frac{1}{2} n^2 \cdot \log n.$$

Зазначимо також, що заміняючи *logk* на *logn*, ми додали принаймні по *k*-1 до кожної складової першої половини суми (де $k \le n/2$), усього приблизно $(n/2)^{2/2}=n^{2/8}$. Більш формально,

$$\sum_{k=1}^{n-1} k \cdot \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \cdot \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \cdot \log k.$$

При $k < \lceil n/2 \rceil$ маємо $\log k \le \log(n/2) = \log n - 1$. Тому

$$\sum_{k=1}^{n-1} k \log k \le (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \le$$

$$\le \frac{1}{2} n(n-1) \log n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2} \le \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

при $n \ge 2$.

Оцінка (3.10) доведена.

Нижченаведене просте виведення оцінки для середнього часу роботи ймовірносного алгоритму швидкого сортування (для трохи іншого варіанта алгоритму) запропонував Л. А. Лєвін [1].

1. Уявимо сортування так: є N каменів різної ваги і шалькові ваги для їх порівняння. Беремо випадковий камінь і ділимо всі камені, що залишилися на три частини: легші цього каменя, тяжчі і він сам, після чого (рекурсивний виклик) сортуємо першу і другу частини.

2. Як вибрати випадковий камінь? Можна вважати, що спочатку всім каменям випадково привласнюються різні ранги (числа від *1* до *N),* і як границю беремо камінь мінімального рангу (з тих, що підлягають сортуванню в даний момент). (Можна перевірити, що це рівносильно незалежним виборам каменів на кожному кроці: на першому кроці кожний з каменів може бути обраний з рівною ймовірністю, після такого вибору в кожній із груп усі камені також рівноймовірнісні і т. д.).

3. Таким чином, кожен камінь характеризується двома числами від *1* до *N* – порядковим *номером* (у порядку зростання ваг) і рангом. Відповідність між номерами і рангами визначає число операцій у процесі сортування.

4. Для кожних двох номерів *i*, *j* з {*1*, ..., *N*} через *p(i,j)* позначимо ймовірність того, що камені з цими номерами будуть порівнюватися один з одним. Наприклад, $p(i, i+1)=1$, оскільки сусідні за вагою камені повинні бути порівняні обов'язково (порівняння з іншими каменями їх не розрізняють).

5. Зазначимо, що $p(i, i+2)=2/3$. Справді, порівняння не відбудеться в тому і тільки тому випадку, коли з трьох каменів з номерами *i, i+1* та *i+2* камінь *i+1* має найменший ранг. Аналогічно, $p(i, i+k)=2/(k+1)$ (камені з номерами *i* та $i + k$ порівнюються, якщо серед $k + 1$ каменів $i, i + 1, .. , i + k$

із двох крайніх має найменший ранг).

6. Математичне сподівання числа порівнянь можна розбити на суму $\sum m(i, j)$ математичних сподівань числа порівнянь між каменями з номерами i та j. Але оскільки два даних камені порівнюються не більш одного разу, виконана рівність $m(i,j)=p(i,j)$. Таким чином, одержуємо точний вираз для математичного сподівання числа порівнянь:

$$\sum_{1 \le i < j \le n} \frac{2}{j - i + 1}.$$

7. Групуючи в цій сумі рівні члени і згадуючи, що 1+1/2+1/3+ ... + +1/k=O($logk$), одержуємо оцінку $O(NlogN)$ для математичного сподівання часу роботи швидкого сортування.

Аналіз полегшується тим, що алгоритм поділу (на три частини) більш симетричний. Реалізація такого способу розбивання також проста: масив ділиться на чотири ділянки (перелічуємо їх зліва направо). Ці ділянки такі: менші границі, рівні границі, непереглянуті і більші границі.

3.4 Алгоритми сортування за лінійний час

В попередніх розділах були розглянуті різні алгоритми, які можуть відсортувати n чисел за час $O(n \log n)$. Алгоритми сортування злиттям (merge sort) і сортування за допомогою купи (heap sort) працюють за такий час у найгіршому випадку, а у алгоритму швидкого сортування – це є середній час роботи. Оцінка $O(n \log n)$ достатньо точна: для кожного із цих алгоритмів можна подати послідовність із n чисел, час обробки якої буде $\Omega(n \log n)$.

Всі згадані алгоритми здійснюють сортування, ґрунтуючись винятково на попарних порівняннях елементів, тому їх іноді називають сортуваннями порівнянням (comparison sort). Проте будь-який алгоритм такого типу сортує n елементів за час не менше $\Omega(n \log n)$ у найгіршому випадку. Таким чином, алгоритми сортування злиттям і за допомогою купи асимптотично оптимальні: не існує алгоритму сортування порівнянням, що перевищував би зазначені алгоритми більш ніж у скінченне число раз.

У наступних параграфах розглядаються три алгоритми сортування (сортування підраховуванням, цифрове сортування і сортування вичерпуванням), що працюють за лінійний час. Зрозуміло, вони поліпшують оцінку $\Omega(n \log n)$ за рахунок того, що використовують не лише попарні порівняння, але й внутрішню структуру об'єктів, що відсортовуються.

3.4.1 Сортування підраховуванням

Алгоритм сортування підраховуванням (counting sort) застосовують, якщо кожний з n елементів послідовності, що відсортовується, – ціле додатне число у відомому діапазоні (що не переважає заздалегідь відоме k). Якщо $k=O(n)$, то алгоритм сортування підраховуванням працює за час $O(n)$.

Ідея цього алгоритму в тому, щоб для кожного елементу x попередньо підрахувати, скільки елементів вхідної послідовності менше x, після чого записати x прямо у вихідний масив відповідно до цього числа (якщо, скажімо, 17 елементів вхідного масиву менше x, то у вихідному масиві x повинен бути записаний на місце номер 18). Якщо в послідовності, що відсортовується, присутні рівні числа, цю схему треба дещо модифікувати, щоб не записати кілька чисел на одне місце.

У псевдокоді, що приводиться нижче, використовується допоміжний масив *C[1... k]* із k елементів. Вхідна послідовність записана в масиві *A[1...n]*, відсортована послідовність записується в масив *B[1...n]*.

```
Counting-Sort(A, B, k)
1     for i ← 1  to k
2           do C[i] ← 0
3     for j ←1  to length [A]
4           do C[A[j]] ← C [A[j]] +1
5     ⇒ C[i] дорівнює кількості елементів рівних i.
6     for i ←2 to k
7           do C[i] ←C[i] ←C[i -1]
8     ⇒ C[i] дорівнює кількості елементів, що не більші i.
9     for j ←length [A] downto 1
10          do B[C[A[j]]] ←A[j]
11                C[A[j]] ←C[A[j]] - 1
```

Роботу алгоритму сортування підраховуванням проілюстровано на рис. 3.21. Після ініціалізації (рядки 1-2) ми спочатку поміщаємо в *C[i]* кількість елементів масиву *A*, рівних *i* (рядки 3-4), а потім, знаходячи часткові суми послідовності *C[1], C[2],..., C[k]* – кількість елементів, що не перевищують *i* (рядки 6-7). Нарешті, у рядках 9-11 кожний з елементів масиву *A* розміщується на потрібне місце в масиві *B*. Справді, якщо всі n елементів різні, то у відсортованому масиві число *A[j]* повинно стояти на місці номер *C[A[j]]*, тому що саме стільки елементів масиву *A* не перевищують *A[j]*; якщо в масиві *A* зустрічаються повторення, то після кожного записування числа *A[j]* у масив *B* число *C[A[j]]* зменшується на одиницю (рядок 11), так що при наступній зустрічі із числом, рівним *A[j]*, воно буде записано на одну позицію лівіше.

Оцінимо час роботи алгоритму сортування підраховуванням. Цикли в рядках 1-2 і 6-7 працюють за час *O(k)*, цикли в рядках 3-4 і 10-11 – за час *O(n)*, а весь алгоритм працює за час *O(k+ n)*. Якщо *k=O(n)*, то час роботи є *O(n)*.

Варто відзначити, що алгоритм сортування підраховуванням не порівнює числа, що відсортовуються, між собою, а використовує їх як індекси масиву.

Алгоритм сортування підраховуванням має важливу властивість, яку називають стійкістю (is stable). А саме, якщо у вхідному масиві присутні декілька рівних чисел, то у вихідному масиві вони розташовуються у тому ж порядку, що й у вхідному. Ця властивість має сенс, тільки якщо в масиві разом із числами записані додаткові дані.

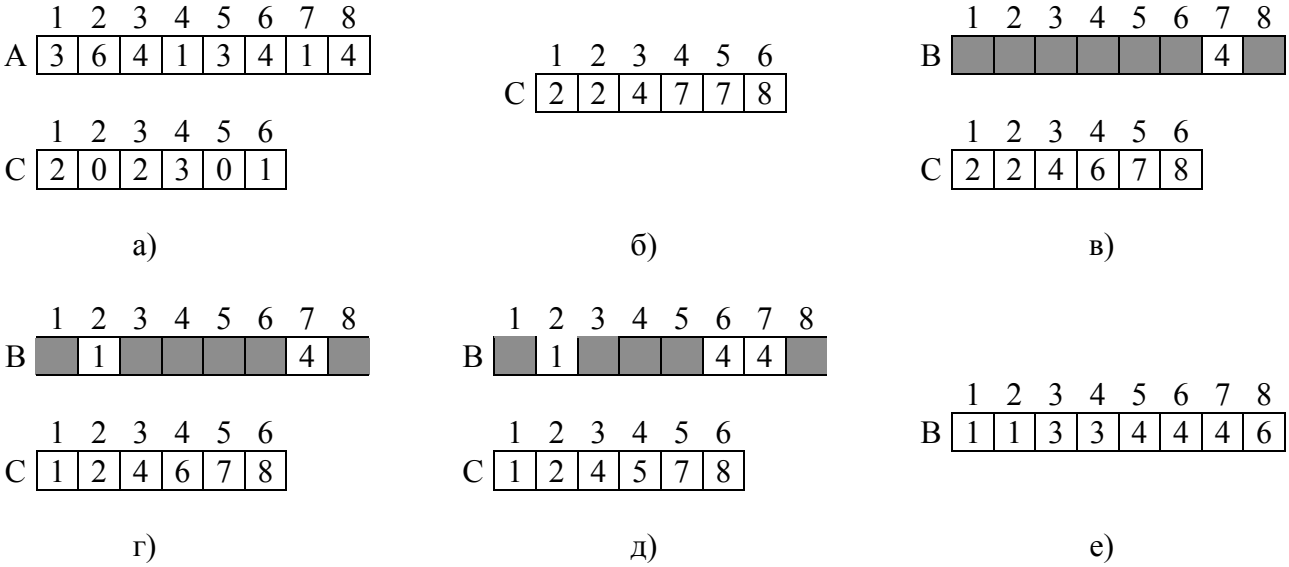


Рисунок 3.21 – Робота алгоритму Counting-Sort, застосованого до масиву A[1..8], що складається з натуральних чисел, які не перевищують *k=6.*

а – масив *A* та допоміжний масив *C* після виконання циклу в рядках 3-4;
б – масив *C* після виконання циклу в рядках 6-7;
в, г, д – вихідний масив *B* і допоміжний масив *C* після одного, двох і трьох повторень циклу в рядках 9-11. Закреслені клітинки відповідають елементам масиву, значення для яких ще не присвоєні;
е – масив *B* після закінчення роботи алгоритму

Більш детально уявимо собі, що ми сортуємо не лише числа, а пари *(t,x)*, де *t* – число від 1 до *k*, а *x* – довільний об'єкт, і хочемо переставити їх так, щоб перші компоненти пар йшли в неспадному порядку. Описаний алгоритм дозволяє це зробити, причому відносне розташування пар з рівними першими компонентами не змінюється. Ця властивість і називається стійкістю.

3.4.2 Цифрове сортування

Алгоритм цифрового сортування (radix sort) використовувся в машинах для сортування перфокарт (зараз такі машини є історією комп'ютерних систем). У картонних перфокартах спеціальний перфоратор пробивав дірки. У кожному з 80 стовпців були місця для 12 прямокутних дірок. Взагалі ж в одній колонці можна було пробити кілька дірок, їхня комбінація відповідала символу (таким чином на карті було місце для 80 символів), але цифри 0-9 кодувалися одиночними дірками в рядках 0-9 відповідного стовпця.

Сортувальній машині вказували стовпець, за яким потрібно зробити сортування, і вона розкладала колоду перфокарт на 10 стовпців залежно від того, яка з дірок 0-9 була пробита в зазначеному стовпці.

Як відсортувати колоду перфокарт із багатозначними числами (розряд одиниць в одному стовпці, десятків – у попередньому і т.д.)? Перше, що спадає на думку, – почати сортування зі старшого розряду. При цьому вийде 10 стовпців, кожен з яких на наступному кроці необхідно буде розбивати на 10 стовпців і так далі – вийде багато стовпців перфокарт, у яких легко заплутатися.

Як не дивно, виявляється зручніше почати з молодшого розряду, розклавши колоду на 10 стовпців залежно від того, де пробитий отвір в «молодшому» стовпці. Отримані 10 стовпців треба після цього скласти в один в такому порядку: спочатку карти з 0, потім карти з 1 і т.д. Отриману колоду знову розсортуємо на 10 стовпців, але вже відповідно до розряду десятків, складемо отримані стовпців в одну колоду і т.д.; якщо на перфокартах були записані *d*-значні числа, то знадобиться *d* раз скористатися сортувальною машиною. На рис. 3.22 зображено, як діє даний алгоритм, який застосовано до семи тризначних чисел.

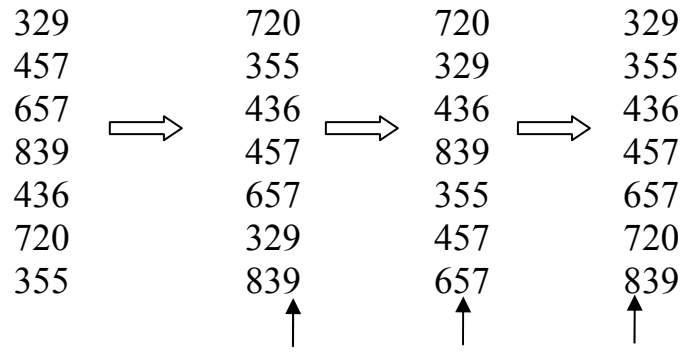


Рисунок 3.22 – Цифрове сортування послідовності із 7 тризначних чисел. На вхід подаються числа першого стовпця. Далі показано порядок чисел після сортування за третьою, другою і першою цифрами (вертикальні стрілки вказують за якою цифрою здійснювалось сортування)

Важливо, щоб алгоритм, за допомогою якого відбувається сортування за даним розрядом, був стійким: картки, в яких у даному стовпці знаходиться одна і та ж цифра, повинні вийти із сортувальної машини в тій же послідовності, в якій вони туди подавалися (зрозуміло, що при складанні 10 стовпців в один змінювати порядок карт у стовпцях теж не потрібно).

У комп'ютерах цифрове сортування іноді використовується для впорядкування даних, що містять декілька полів. Нехай, наприклад, нам потрібно відсортувати послідовність дат. Це можна зробити за допомогою будь-якого алгоритму сортування, порівнюючи дати в такий спосіб: порівняти роки, якщо роки збігаються – порівняти місяці, якщо збігаються й місяці – порівняти числа. Проте замість цього можна просто тричі відсортувати масив дат за допомогою стійкого алгоритму: спочатку за днями, потім за місяцями, потім за роками.

Програму для цифрового сортування написати нескладно. Ми припускаємо, що кожний елемент n-елементного масиву A складається з d цифр, причому цифра номер 1 – молодший розряд, а цифра номер d – старший.

```
RADIX-SORT(A, d)
1     for i ← 1 to d
2           do відсортувати масив A стійким
                  алгоритмом за значенням цифри номер i
```

Правильність алгоритму цифрового сортування доводиться індукцією за номером розряду. Час роботи залежить від часу роботи обраного стійкого алгоритму. Якщо цифри можуть приймати значення від 1 до k, де k не занадто велике число, то очевидний вибір – сортування підраховуванням. Для n чисел з d знаками від 0 до $k - 1$ кожний прохід забирає час $\theta(n+\kappa)$; оскільки ми робимо d проходів, час роботи цифрового сортування дорівнює $\theta(dn + kd)$. Якщо d стале та $k=O(n)$, то цифрове сортування здійснюється за лінійний час.

При цифровому сортуванні важливо правильно вибрати основу системи числення, оскільки від неї залежить розмір необхідної додаткової пам'яті та час роботи. Конкретний приклад: нехай треба відсортувати мільйон 64-бітових чисел. Якщо розглядати їх як чотиризначні числа в системі числення з основою 2^{16}, то при цифровому сортуванні ми впораємося з задачею за чотири проходи, використовуючи в процедурі ***Counting-Sort*** масив B розміром 2^{16} (це невелике значення в порівнянні з розміром відсортованого масиву). Це вигідно відрізняється від сортування порівнянням, коли на кожне число витрачається по $\log n \approx 20$ операцій. На жаль, цифрове сортування, що базується на сорту-

ванні підраховуванням, потребує ще одного масиву (того ж розміру що й масив сортуванння) для зберігання проміжних результатів, у той час як багато алгоритмів сортування порівнянням обходяться без цього. Тому, якщо треба заощаджувати пам'ять, алгоритм швидкого сортування може виявитися більш вигідним для застосування.

3.4.3 Сортування вичерпуванням

Алгоритм сортування вичерпуванням (bucket sort) працює за лінійний середній час. Як і сортування підраховуванням, сортування вичерпуванням застосовується не для будь-яких вихідних даних: вказуючи на лінійний середній час, ми припускаємо, що на вхід подається послідовність незалежних випадкових чисел, рівномірно розподілених на проміжку [0,1].

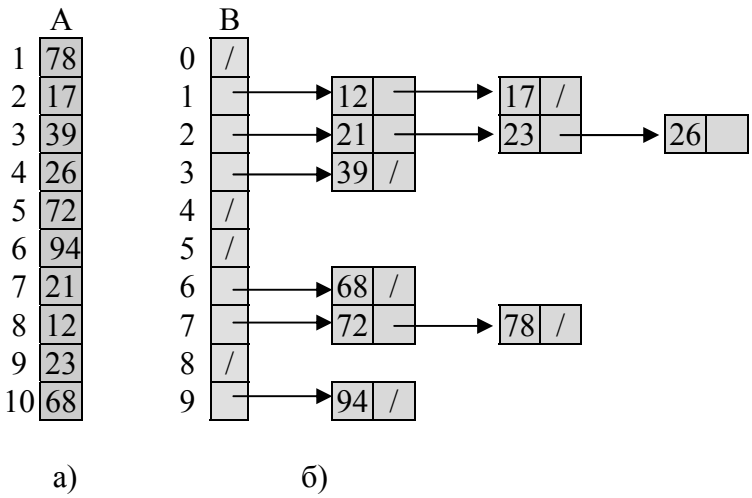


Рисунок 3.23 – Робота алгоритму Bucket-Sort

а – на вхід подано масив *A[1.. 10]*;

б – масив списків *B[0.. 9]* після виконання рядка 5.

Список з індексом *i* містить числа, у яких перший знак після коми є *i*.

Відсортований масив буде тоді, якщо послідовно виписати списки *B[0],..., B[9]*

Зауважимо, що цей алгоритм – детермінований (не використовує генератора випадкових чисел); поняття випадковості виникає лише при аналізі часу його роботи.

Ідея алгоритму полягає в тому, що проміжок [0,1] поділяється на *n* рівних частин, після чого для чисел з кожної частини виділяється свій ящик-черпак (bucket), і *n* чисел, що підлягають сортуванню, розкладаються по цих ящиках. Оскільки числа рівномірно розподілені на проміжку [0,1], варто очікувати, що в кожному ящику їх буде небагато. Тепер

відсортуємо числа в кожному ящику окремо й пройдемося по ящиках у порядку зростання, виписуючи числа, що потрапили в кожний з них, також у порядку зростання.

Будемо вважати, що на вхід подається *n*-елементний масив *A*, причому *0≤A[i]<1* для всіх *i*. Використовується також допоміжний масив *B[0.. n-1]*, що складається зі списків, що відповідають ящикам. Алгоритм використовує операції зі списками.

```
Bucket-Sort(A)
1    n ← length[A]
2    for i ← l to n
3         do додати A[i] до списку B[nA[i]]
4    for i ← 0 to n - 1
5         do відсортувати список B[i] (сортування вставками)
6    з'єднати списки B[0], B[1],..., B[n - 1] (в зазначеному порядку)
```

На рис. 3.23 показано роботу цього алгоритму на прикладі масиву з 10 чисел.

Щоб довести, що алгоритм сортування вичерпуванням правильний, розглянемо два числа *A[i]* та *A[j]*. Якщо вони потрапили в різні ящики, то менше з них потрапило в ящик з меншим номером, і у вихідній послідовності воно виявиться раніше; якщо вони потрапили в один ящик, то після сортування вмісту ящика менше число буде також передувати більшому.

Проаналізуємо час роботи алгоритму. Операції у всіх рядках, крім п'ятого, потребують загального часу *O(n)*. Перегляд всіх ящиків також забирає час *O(n)*. Таким чином, нам залишається тільки оцінити час сортування вставками всередині ящиків.

Нехай у ящик *B[i]* потрапило n_i чисел (n_i – випадкова величина). Оскільки сортування вставками працює за квадратичний час, математичне очікування часу сортування чисел у ящику номер *i* має значення $O(M[n_i^2])$, а математичне очікування сумарного часу сортування у всіх ящиках має значення

$$\sum_{i=0}^{n-1} O(M[n_i^{\,2}]) = O\left(\sum_{i=1}^{n-1} M[n_i^{\,2}]\right). \qquad (3.11)$$

Знайдемо функцію розподілу випадкових величин n_i. Оскільки числа розподілені рівномірно, а довжини всіх відрізків рівні, ймовірність того, що дане число потрапить у ящик номер *i*, дорівнює *1/n*. Отже, ми перебуваємо в відомій ситуації з кулями й урнами: у нас *n* куль-чисел, *n* урн-ящиків, і ймовірність влучення даної кулі в дану урну дорівнює

$p = 1/n$. Тому числа n_i розподілені біноміально: ймовірність того, що $n_i=k$, дорівнює $C_n^k p^k (1-p)^{n-k}$, математичне очікування дорівнює $M[n_i]=np=1$, і дисперсія дорівнює $D[n_i]=np(1-p)=1-1/n$. Отже, можемо записати:

$$M[n_i^2] = M[n_i] + M^2[n_i] = 2 - \frac{1}{n} = \Theta(1).$$

Підставляючи цю оцінку в (3.11), одержуємо, що математичне очікування сумарного часу сортування вмісту всіх ящиків має значення $O(n)$, так що математичне очікування часу роботи алгоритму сортування вичерпуванням справді лінійно залежить від кількості чисел.

3.5 Контрольні питання

1. Поясніть чому задача сортування елементів є однією з найцікавіших та показових задач для курсу теорії алгоритмів.
2. За якими критеріями на Ваш погляд можна класифікувати алгоритми сортування?
3. Наведіть класифікацію алгоритмів сортування.
4. Перерахуйте та порівняйте відомі Вам алгоритми сортування за квадратичний час.
5. Які алгоритми сортування за псевдолінійний час Ви знаєте? Порівняйте між собою ці алгоритми.
6. Перерахуйте основні алгоритми сортування за лінійний час, та наведіть їх основні перевали та недоліки.
7. Можна стверджувати, що алгоритми сортування за квадратичний час працюють завжди довше для будь-яких входів? Відповідь поясніть.
8. Поясніть чому при оцінці складності алгоритму нас частіше за все цікавить робота у найгіршому випадку.
9. На прикладі алгоритму бульбашкового сортування продемонструйте оцінку складності у найкращому та найгіршому випадках, а також у середньому.
10. Сформулюйте принцип „розділяй і володарюй” та покажіть як його можна застосувати до задачі сортування.
11. Яку структуру даних називають купою? Наведіть відповідний приклад.
12. Наведіть основні операції над купою.
13. Наведіть та поясніть як працює процедура збереження головної властивості купи.
14. Поясніть яким чином з довільного масиву побудувати купу. Скільки часу потребує така процедура?
15. Наведіть власний приклад сортування за допомогою купи.
16. У чому полягає ідея алгоритму швидкого сортування?
17. Оцініть час роботи алгоритму швидкого сортування.

18. З якою метою застосовують ймовірнісні алгоритми швидкого сортування?

19. Чому для ймовірнісного алгоритму важливий не максимальний час роботи (для даного входу), а математичне сподівання цього часу?

20. Як правило, у банках чеки обробляють у порядку їх надходження. Клієнти ж надають перевагу, щоб у звіті платежі були зазначені у порядку номерів чеків. Власник чекової книжки, зазвичай, виписує чеки підряд, а одержувачі чеків пред'являють їх у банк незабаром після виписування. Таким чином, порядок номерів порушується незначною мірою. Отже, банку потрібно відсортувати майже відсортований масив. Поясніть чому сортування вставками в таких випадках працює швидше, ніж швидке сортування.

21. Охарактеризуйте особливості алгоритмів сортування, що працюють за лінійний час.

22. Наведіть та поясніть основні властивості алгоритму сортування підраховуванням.

23. Сфери застосування алгоритму цифрового сортування.

24. Наведіть власний приклад алгоритму цифрового сортування.

25. Який час роботи алгоритму сортування вичерпуванням у найгіршому випадку? Наведіть його просту модифікацію, що зберігає лінійний середній час роботи та зменшує час роботи в найгіршому випадку до O(n log n).

3.6 Завдання

1. а) наведіть алгоритм сортування, заданий згідно з варіантом;

Варіант	Алгоритм сортування	Критерій сортування
1	сортування вставками	за зростанням
2	сортування вставками	за спаданням
3	сортування прямим вибором	за зростанням
4	сортування прямим вибором	за спаданням
5	бульбашкове сортування	за зростанням
6	бульбашкове сортування	за спаданням
7	модифіковане бульбашкове сортування	за зростанням
8	модифіковане бульбашкове сортування	за спаданням
9	шейкерне сортування	за зростанням
10	шейкерне сортування	за спаданням

б) оцініть складність алгоритму та наведіть графік, побудований за теоретично отриманою залежністю часу розв'язання задачі $\boldsymbol{T}$ від розміру входу $\boldsymbol{N}$;

в) запрограмуйте розроблений алгоритм сортування та визначте час його виконання для $N = 10^3, 10^4, 10^5, 10^6$. (З цією метою згенеруйте файл випадкових чисел) Наведіть практично отримані графіки залежності ***T(N)*** для трьох випадків:

– вихідний масив не відсортований;

– вихідний масив відсортований у зворотному порядку;

– вихідний масив відсортований у потрібному порядку;

г) порівняйте теоретично та практично отримані графіки залежності;

д) наведіть переваги і недоліки даного алгоритму та зробіть відповідні висновки.

2. Розв’яжіть попередню задачу для алгоритмів сортування заданих згідно з нижченаведеними варіантами.

Варіант	Алгоритм сортування	Критерій сортування
1	злиттям	за зростанням
2	злиттям	за спаданням
3	за допомогою купи	за зростанням
4	за допомогою купи	за спаданням
5	швидке сортування	за зростанням
6	швидке сортування	за спаданням
7	ймовірнісне швидке сортування	за зростанням
8	ймовірнісне швидке сортування	за спаданням
9	методом підраховування	за зростанням
10	методом підраховування	за спаданням
11	цифрове сортування	за зростанням
12	цифрове сортування	за спаданням
13	сортування вичерпуванням	за зростанням
14	сортування вичерпуванням	за спаданням

3. Дотримуючись зразка (рис. 3.22), покажіть як відбувається цифрове сортування (за алфавітом) англійських слів COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

4. Які з вказаних алгоритмів сортування є стійкими: сортування вставками, сортування злиттям, сортування за допомогою купи, швидке сортування? Поясніть яким чином можна будь-який алгоритм сортування перетворити в стійкий. Скільки при цьому буде необхідно додаткового часу й пам'яті?

4 ДИНАМІЧНЕ ПРОГРАМУВАННЯ

Динамічне програмування застосовується до задач, у яких відповідь складається із частин, кожна з яких у свою чергу дає оптимальне

розв’язання деякої підзадачі. Динамічне програмування корисне в тому випадку, якщо на різних шляхах багаторазово зустрічаються ті самі підзадачі. Основний технічний прийом – запам'ятовувати розв’язання підзадач, що зустрічаються, на випадок, якщо та ж підзадача зустрінеться знову.

Подібно методу «розділяй і володарюй», динамічне програмування вирішує задачу, розбиваючи її на підзадачі та поєднуючи їх розв’язання. Алгоритми типу «розділяй і володарюй» ділять задачу на незалежні підзадачі, ці підзадачі – на більш дрібні підзадачі й так далі, а потім збирають розв’язання основної задачі «знизу вгору». Динамічне програмування ефективно застосовувати тоді, коли підзадачі не є незалежними, іншими словами, коли в підзадачі є загальні «під-підзадачі». У цьому випадку алгоритм типу «розділяй і володарюй» буде робити зайву роботу, вирішуючи ті самі «під-підзадачі» декілька раз. Алгоритм, заснований на динамічному програмуванні, вирішує кожну з підзадач одноразово та запам'ятовує відповіді в спеціальній таблиці. Це дозволяє не обчислювати заново відповідь для підзадачі, що вже зустрічалася.

У типовому випадку динамічне програмування застосовується до задач оптимізації (optimization problems). У таких задач може бути багато можливих рішень; їх «якість» визначається значенням певного параметра, і потрібно вибрати оптимальне розв’язання, при якому значення параметра буде мінімальним або максимальним (залежно від постановки задачі). Загалом кажучи, оптимум може досягатися для декількох різних рішень.

Як будується алгоритм, заснований на динамічному програмуванні? Для цього необхідно:

а) описати структуру оптимальних рішень;

б) виписати рекурентне співвідношення, що зв'язує оптимальне значення параметра для підзадач;

в) рухаючись знизу вгору, обчислити оптимальне значення параметра для підзадач;

г) користуючись отриманою інформацією, побудувати оптимальне розв’язання.

Основну частину роботи становлять кроки 1-3. Якщо нас цікавить тільки оптимальне значення параметра, крок 4 не потрібний. Якщо ж крок 4 необхідний, для побудови оптимального розв’язання іноді доводиться одержувати й зберігати додаткову інформацію в процесі виконання кроку 3.

У даному розділі ми розглянемо та вирішимо деякі оптимізаційні задачі за допомогою динамічного програмування. У підрозділі 4.1 ми з'ясуємо як знайти добуток декількох матриць, зробивши якнайменше операцій множень. У підрозділі 4.2 ми обговоримо, які властивості задачі уможливлюють застосування динамічного програмування. Після цього, у

підрозділі 4.3, ми розглянемо як знайти найбільшу загальну підпослідовність двох послідовностей.

4.1 Задача про пошук добутку кількох матриць

Ми хочемо знайти добуток

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n \qquad (4.1)$$

послідовності n матриць $(A_1,A_2,\ldots,A_n)$. Ми будемо користуватися стандартним алгоритмом перемножування двох матриць як підпрограмою. Але спочатку необхідно розставити дужки в (4.1), щоб вказати порядок операцій множення. Зауважимо, що в добутку матриць повністю розставлені дужки (product is fully parenthesized), якщо цей добуток або складається з єдиної матриці, або є взятим у дужки добутком двох добутків з повністю розставленими дужками. Оскільки множення матриць асоціативне, кінцевий результат обчислень не залежить від розміщення дужок. Наприклад, у добутку $A_1 \cdot A_2 \cdot A_3 \cdot A_4$ можна повністю розставити дужки п'ятьма різними способами:

$(A_1(A_2(A_3\ A_4)))$, $(A_1((A_2\ A_3)A_4))$, $((A_1\ A_2)(A_3A_4))$, $((A_1(A_2\ A_3))A_4)$, $(((A_1A_2)A_3)A_4)$; у всіх випадках відповідь буде однакова.

Не впливаючи на відповідь, спосіб розміщення дужок може сильно вплинути на вартість перемножування матриць. Подивимося спочатку скільки операцій вимагає перемножування двох матриць. Ось стандартний алгоритм (rows і columns позначають кількість рядків і стовпців матриці відповідно):

Matrix-Multiply (A, B)

```
1  if columns [A] ≠ rows [B]
2     then error "помножити не можна"
3     else for i←1 to rows [A]
4             do for j←1 to columns [B]
5                    do C[i, j] ← 0
6                           for k←1 to columns [A]
7                                  do C[i ,j] ← C[i ,j] + A[i ,k]· B[k ,j]
8           return C
```

Матриці A і B можна перемножувати, тільки якщо число стовпців в A дорівнює числу рядків у B. Якщо A – це $(p\times q)$-матриця, а B – це $(q\times r)$-матриця, то і їхній добуток C є $(p\times r)$-матрицею. При виконанні цього алгоритму робиться pqr множень (рядок 7) і приблизно стільки ж додавань. Для простоти ми будемо враховувати тільки множення.

Щоб побачити, як розміщення дужок може впливати на вартість, розглянемо послідовність із трьох матриць (A_1,A_2,A_3) розмірністю 10×100, 100×5 і 5×50, відповідно. При обчисленні $((A_1A_2)A_3)$ потрібно 10·100·5 = = 5000 множень, щоб знайти (10×5)-матрицю A_1A_2, а потім 10·5·50 = = 2500 множень, щоб помножити цю матрицю на A_3. Усього 7500 множень. При розміщенні дужок $(A_1(A_2A_3))$ ми робимо 100·5·50 = 25 000 множень для знаходження (100×50)- матриці A_2A_3, плюс ще 10·100·50 = = 50 000 множень (множення A_1 на A_2A_3), разом 75 000 множень. Тим самим перший спосіб розміщення дужок в 10 разів вигідніше.

Задача про множення послідовності матриць (matrix-chain multiplication problem) може бути сформульована в такий спосіб: дано послідовність із n матриць $\{A_1, A_2, ..., A_n\}$ заданої розмірності (матриця A_i має розмірність $p_{i-1} \times p_i$); потрібно знайти таке (повне) розміщення дужок у добутку $A_1A_2....A_n$, щоб обчислення добутку потребувало найменшого числа множень.

Кількість розміщень дужок

Перш ніж застосовувати динамічне програмування до задачі про множення послідовності матриць, варто переконатися, що простий перебір всіх можливих розміщень дужок не дасть ефективного алгоритму. Позначимо $P(n)$ кількість повних розміщень дужок у добутку n матриць. Останнє множення може відбуватися на границі між k-ю і $(k+1)$-ю матрицями. До цього ми окремо обчислюємо добуток перших k та інших n-k матриць. Тому

$$P(n) = \begin{cases} 1, & \text{якщо } n = 1, \\ \sum_{k-1}^{n-1} P(k)P(n-k), & \end{cases}$$

$$C(n) = \frac{1}{n+1} C^n{}_{2n} = \Omega(4^n / n^{3/2}), \qquad \text{якщо } n \geq 2.$$

Це співвідношення задає послідовність чисел Каталана

$$P(n) = C(n-1),$$

де $C(n) = \frac{1}{n+1} C^n_{2n} = \Omega(4^n / n^{3/2})$.

Виходить число розміщень експоненціально залежить від n, так що повний перебір неефективний. Інший спосіб визначити, що число варіантів експоненціальне – розіб'ємо матриці на групи по три; добуток для ко-

жної групи можна обчислити двома способами, так що для *3n* матриць є не менше *2n* варіантів.

Крок 1: структура оптимального розміщення дужок

Якщо ми збираємося скористатися динамічним програмуванням, то для початку повинні описати склад оптимальних рішень. Для задачі про множення послідовності матриць це виглядає в такий спосіб. Позначимо для зручності через $A_{i..j}$ матрицю, що є добутком $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$. Оптимальне розміщення дужок у добутку $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ розриває послідовність між A_k і A_{k+1} для якогось k, що задовольняє нерівності $1 \le k < n$. Іншими словами, при обчисленні добутку, диктованого цим розміщенням дужок, ми спочатку обчислюємо добуток $A_{1...k}$ і $A_{k+1...n}$, потім перемножуємо їх і одержуємо остаточну відповідь $A_{1..n}$. Отже, вартість цього оптимального розміщення дорівнює вартості обчислення матриці $A_{1...k}$ плюс вартість обчислення матриці $A_{k+1...n}$ плюс вартість перемножування цих двох матриць.

Чим менше множень нам буде потрібно для обчислення $A_{1...k}$ і $A_{k+1...n}$, тим менше буде загальне число множень. Виходить, оптимальне розв'язання задачі про перемножування послідовності матриць містить оптимальні розв'язання задач про перемножування її частин.

Крок 2: рекурентне співвідношення

Тепер необхідно виразити вартість оптимального розв'язання задачі через вартості оптимальних рішень її підзадач. Такими підзадачами будуть задачі про оптимальне розміщення дужок у добутках

$A_{i..j} = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ для

$1 \le i \le j \le n$.

Позначимо через *m[i,j]* мінімальну кількість множень, необхідну для обчислення матриці $A_{i..j}$; зокрема, вартість обчислення всього добутку $A_{1..n}$ є *m[l,n]*.

Числа *m[i,j]* можна обчислити так. Якщо *i=j*, то послідовність складається з однієї матриці $A_{i..i} = A_i$ і множення взагалі не потрібні. Виходить *m[i, i] = 0* для *i = 1,2,... ,n*. Щоб підрахувати *m[i, j]* для $i < j$, ми скористаємося інформацією про будову оптимального розв'язання, отриманого на кроці 1. Нехай при оптимальному розміщенні дужок у добутку $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ останнім буде множення $A_i \cdot \ldots \cdot A_k$ на $A_{k+i} \cdot \ldots \cdot A_j$, де $i \le k < j$. Тоді *m[i, j]* дорівнює сумі мінімальних вартостей обчислення добутків $A_{i...k}$ і $A_{k+1...j}$ плюс вартість перемножування цих двох матриць. Оскільки для обчислення добутку $A_{i...k} A_{k+1...j}$ потрібно $p_{i-1} p_k p_j$ множень,

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

У цьому співвідношенні мається на увазі, що оптимальне значення k відомо, насправді це не так. Однак число k може приймати всього лише *j - i* різних значень: *i,i + 1,..., j - 1*. Оскільки одне з них оптимальне, досить перебрати ці значення *k* і обрати найкраще. Одержуємо рекурентну формулу:

$$m[i,j]=\begin{cases}0 & \text{при } i=j,\\ \min\limits_{i\le k<j}\{m[i,k]+m[k+1,j]+p_{i-1}p_k p_j\} & \text{при } i<j.\end{cases} \quad (4.2)$$

Числа *m[i,j]* – вартості оптимальних рішень підзадач. Щоб простежити за тим як виходить оптимальне розв'язання, позначимо через *s[i,j]* оптимальне місце останнього множення, тобто таке *k*, що при оптимальному обчисленні добутку $A_i \cdot A_{i+1} \ldots \cdot A_j$ останнім іде множення $A_i \cdot \ldots \cdot A_k$ на $A_{k+i} \cdot \ldots \cdot A_j$. Іншими словами, *s[i,j]* дорівнює числу *k*, для якого

$$m[i,j]=m[i,k]+m[k+1,j]+p_{i-1}p_k p_j .$$

Крок 3: обчислення оптимальної вартості

Користуючись співвідношеннями (4.2), тепер легко написати рекурсивний алгоритм, що визначає мінімальну вартість обчислення добутку $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ (тобто число *m[1,n]*). Однак час роботи такого алгоритму експоненціально залежить від n, так що цей алгоритм не краще повного перебору. Дійсний виграш у часі ми одержимо, якщо скористаємося тим, що підзадач відносно небагато: по одній задачі для кожної пари *(i,j)*, для якої $1 \le i \le j \le n$, а всього $C_n^2 + n = \theta(n^2)$. Експонентний час роботи виникає тому, що рекурсивний алгоритм вирішує кожну з підзадач багато раз на різних розгалуженнях дерева рекурсії. Таке «перекриття підзадач» – характерна ознака задач, розв'язуваних методом динамічного програмування.

Замість рекурсії ми обчислимо оптимальну вартість «знизу вгору». У нижченаведеній програмі передбачається, що матриця A_i має розмірність $p_{i-1} \cdot p_i$ при *i=1,2,...,n*. На вхід подається послідовність $p = (p_0, p_1, \ldots, p_n)$, де *length [p] = n+1*. Програма використовує допоміжні таблиці *m[1..n, 1.. n]* (для зберігання вартостей *m[i,j]*) і *s[1... n, 1.. n]* (у ній відзначається при якому *k* досягається оптимальна вартість при обчисленні *m[i,j]*).

Заповнюючи таблицю *m*, цей алгоритм послідовно вирішує задачі про оптимальне розміщення дужок для *1, 2, ..., n* співмножників. Справді, співвідношення (4.2) показує, що число *m[i,j]* – вартість перемножування *j-i+1* матриць – залежить тільки від вартостей перемножування меншого (ніж *j-i+1*) числа матриць. Саме для

$k = i,i + 1,... ,j - 1$ виходить, що $A_{i...k}$ – добуток $k - i + l<j - i + 1$ матриць, а $A_{k+1...j}$ – добуток $j - k < j - i + 1$ матриць.

```
Matrix – Chain – Order (p)
1 n←length[p] -1
2 for i←1 to n
3       do m[i,i]←0
4 for l ←2 to n
5       do for i←1 to n-l+1
6             do j←i+l – 1
7                  m[I,j]←∞
8                  for k←i to j - 1
9                       do q←m[i,k] + m[k+1,j]+p_{i-1}p_kp_j
10                           if q < m [i,j]
11                              then m [i,j]← q
12                                   s [i,j] ←k
13 return m, s
```

Спочатку (у рядках 2-3) алгоритм виконує присвоювання $m[i, i] \leftarrow 0$ для $i = 1, 2,..., n$; вартість перемножування послідовності з однієї матриці дорівнює нулю. При першому виконанні циклу (рядки 4-12) обчислюються (за допомогою співвідношень (4.2)) значення $m[i,i + 1]$ для $i=1,2,...,n - 1$ – це мінімальні вартості для послідовностей довжиною 2. При другому проході обчислюються $m[i, i + 2]$ для i = 1, 2,..., n - 2 – мінімальні вартості перемножування послідовностей довжиною 3 і т. д. На кожному кроці значення $m[i,j]$, що обчислює в рядках 9-12, залежить тільки від обчислених раніше значень $m[i, k]$ і $m[k + 1,j]$.

На рис. 4.1 показано як це відбувається при $n = 6$. Оскільки ми визначаємо $m[i, j]$ тільки для $i \le j$, використовується частина таблиці, що знаходиться над головною діагоналлю. На рисунку таблиця повернута (головна діагональ горизонтальна).

Внизу виписана послідовність матриць. Число $m[i,j]$ – мінімальна вартість перемножування $A_i \cdot A_{i+1} \cdot \ ... \ \cdot A_j$ – перебуває на перетині діагоналей, що йдуть вгору від матриці A_i і вгору від матриці A_j. У кожному горизонтальному ряді зібрані вартості перемножування підпослідовностей фіксованої довжини. Для заповнення комірки $m[i,j]$ потрібно знати добуток $p_{i-1} \cdot p_k \cdot p_j$ для $k = i, i + 1,..., j - 1$ і вміст комірок, що лежать зліва внизу й справа внизу від $m[i,j]$.

Просте оцінювання показує, що час роботи алгоритму MATRIX-CHAINORDER є $O(n^3)$. Справді, число вкладених циклів дорівнює трьом, і кожний з індексів l, i, k приймає не більше n значень.

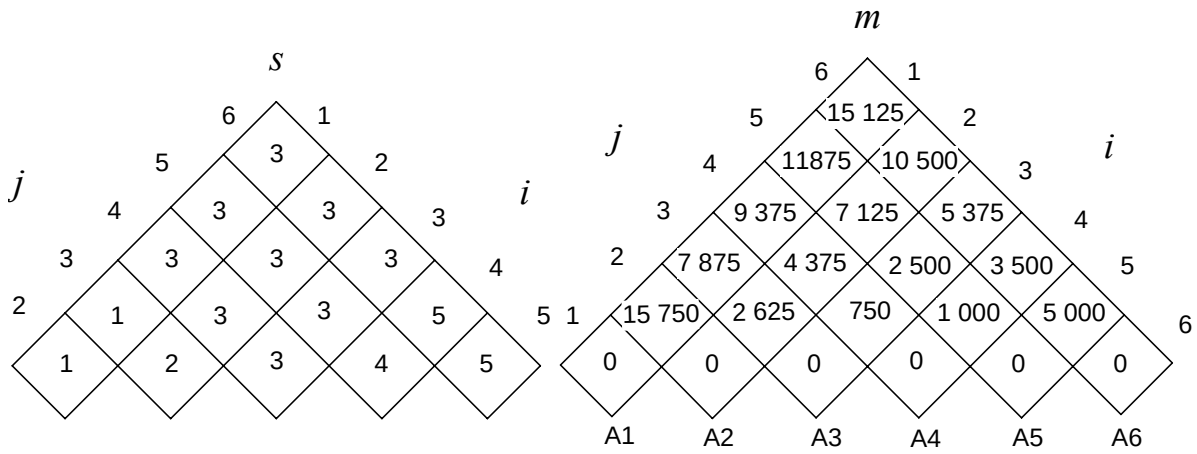


Рисунок 4.1 – Таблиці *m* та *s*, що обчислюються процедурою Matrix-Chain-Order для $n = 6$ і матриць такої розмірності:

матриця	розмірність	матриця	розмірність
A_1	30×35	A_4	5×10
A_2	35×15	A_5	10×20
A_3	15×5	A_6	20×25

Таблиці повернуті так, що головна діагональ горизонтальна. У таблиці m використовуються тільки комірки, що лежать не нижче головної діагоналі, у таблиці s – тільки комірки, що лежать строго вище. Мінімальна кількість множень, необхідних для перемножування всіх шести матриць, дорівнює *m[1,6] = 15 125*. Пари комірок, заштрихованих однаковим світлим штрихуванням, спільно входять у праву частину формули в процесі обчислення *m[2,5]* (рядок 9 процедури Matrix-Chain-Order):

$$m[2,5] = \min\begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375. \end{cases}$$

Час роботи цього алгоритму – $\theta(n^3)$. Обсяг пам'яті, необхідний для зберігання таблиць *m* і *s* – $\theta(n^2)$. Тим самим цей алгоритм значно ефективніший, ніж перебір всіх розміщень, що потребує експонентного часу.

Крок 4: побудова оптимального розв'язання

Алгоритм Matrix-Chain-Order знаходить мінімальне число множень, необхідних для перемножування послідовності матриць. Залишилося знайти розміщення дужок, що приводить до такого числа множень.

Для цього ми використаємо таблицю *s[l.. n, 1.. n]*. У комірці *s[l,n]* записане місце останнього множення при оптимальному розміщенні дужок; інакше кажучи, при оптимальному способі обчислення $A_{1\ldots n}$ остан-

нім іде множення $A_{1...s[1,n]}$ на $A_{s[1,n]+1...n}$. Попередні множення можна знайти рекурсивно: значення *s[1,s[1,n]]* визначає останнє множення при знаходженні $A_{1..s}$,*[1,n]* , а *s[s[l, n] + 1, n]* визначає останнє множення при обчисленні $A_{s[1,n]+1...n}$. Наведена нижче рекурсивна процедура обчислює добуток $A_{i...j}$, маючи такі дані: послідовність матриць $A = (A_1, A_2, ..., A_n)$, таблицю *s*, знайдену процедурою Matrix-Chain-Order, а також індекси *i* , *j*. Добуток $A_1 \cdot A_2 \cdot \ ... \ \cdot A_n$ дорівнює Matrix-Chain-Multiply(A, s, 1, n).

```
Matrix-Chain-Multiply(A, s, i, j)
1   if j > i
2       then X ← Matrix-Chain-Multiply(A, s, i, s[i,j])
3            Y ← Matrix-Chain-Multiply(A, s, s[i,j] + 1, j)
4            return Matrix-Multiply(X, Y)
5       else return A_i
```

У прикладі на рис. 4.1 виклик Matrix-Chain-Multiply (A,s,1,6) обчислить добуток шести матриць відповідно до розміщення дужок

$$((A_1(A_2A_3))((A_4A_5)A_6)). \qquad (4.3)$$

Технічне зауваження: варто подбати, щоб при передаванні масиву s у процедуру не відбувалося копіювання.

4.2 Випадки застосування динамічного програмування

У цьому розділі ми вкажемо дві ознаки, характерні для задач, що розв'язуються методом динамічного програмування.

Оптимальність для підзадач

При рішенні оптимізаційної задачі за допомогою динамічного програмування необхідно спочатку описати структуру оптимального розв’язання. Будемо говорити, що задача має властивість оптимальності для підзадач (has optimal substructure), якщо оптимальне розв’язання задачі містить оптимальні розв’язання її підзадач. Якщо задача має цю властивість, то динамічне програмування може виявитися корисним для її розв’язання (а можливо, застосуємо й жадібний алгоритм).

У підрозділі 4.1 ми відзначили, що задача перемножування матриць має властивість оптимальності для підзадач: кожна дужка в оптимальному добутку вказує оптимальний спосіб перемножування вхідних у неї матриць. Щоб переконатися, що задача має цю властивість, треба (як у підрозділі 4.1) показати, що, поліпшуючи розв’язання підзадачі, ми поліпшимо й розв’язання вихідної задачі.

Як тільки властивість оптимальності для підзадач встановлено, звичайно, стає ясно, з якою саме множиною підзадач буде мати справу алгоритм. Наприклад, для задачі про перемножування послідовності матриць підзадачами будуть задачі про перемножування частин цієї послідовності.

Підзадачі, що перекриваються

Друга властивість задач – істотна при використанні динамічного програмування, невелике число різних підзадач. Завдяки цьому при рекурсивному рішенні задачі ми багаторазово виходимо на ті самі підзадачі. У такому випадку говорять, що в оптимізаційної задачі є **підзадачі, що перекриваються** (overlapping subproblems). У типових випадках кількість підзадач поліноміально залежить від розмірності вихідних даних.

У задачах, розв'язуваних методом «розділяй і володарюй», так не буває: для них рекурсивний алгоритм, як правило, на кожному кроці породжує зовсім нові підзадачі. Алгоритми, засновані на динамічному програмуванні, використовують перекриття підзадач у такий спосіб: кожна з підзадач вирішується тільки один раз, і відповідь заноситься в спеціальну таблицю; коли ж ця підзадача зустрічається знову, програма не витрачає час на її розв’язання, а бере готову відповідь із таблиці.

Повернемося до прикладу для задачі про перемножування послідовності матриць. З рисунка 4.1 видно, що розв’язання кожної з підзадач, записане в даній комірці таблиці, багаторазово використовується процедурою Matrix-Chain-Order при рішенні підзадач із розташованих вище клітинок. Наприклад, *m[3,4]* використовується чотириразово: при обчисленні *m[2,4]*, *m[1,4], m[3,5]* і *m[3,6]*. Було б вкрай неефективно обчислювати *m[3,4]* щораз заново.

Справді, розглянемо такий (неефективний) рекурсивний алгоритм, заснований безпосередньо на співвідношеннях (4.2) і що обчислює *m[i,j]* – мінімальну кількість множень, необхідну для обчислення $A_{i...j} = A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.

На рис. 4.2 зображено дерево рекурсії для Recursive-Matrix-Chain (p,1,4). У кожній вершині записані значення i та j. Зверніть увагу, що деякі пари (i,j) зустрічаються багаторазово.

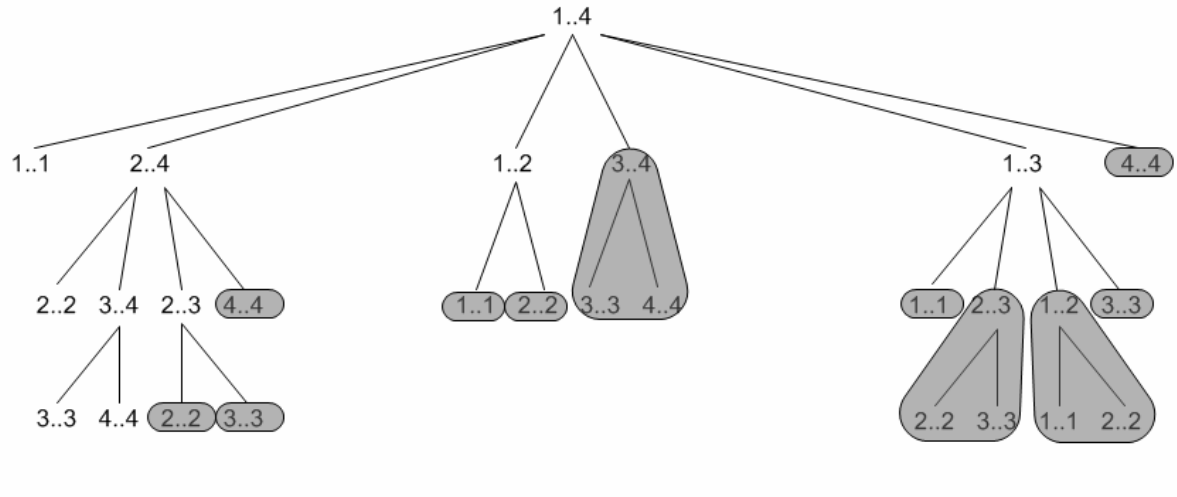


Рисунок 4.2 – Дерево рекурсії для Recursive-Matrix-Chain(p, 1,4). У кожній вершині записані значення і та j. Заштриховані «зайві» вершини (обчислення, які повторюють, уже здійснені)

```
Recursive-Matrix-Chain(p,i,j)
1 if i=j
2       then return 0
3 m[i,j] ←∞
4 for k←i  to j-1
5         do q ← Recursive-Matrix-Chain(p,i,k) +
              + Recursive-Matrix-Chain(p,k+1,j)+p_{i-1}p_kp_j
6            if q<m[i,j]
7                 then m[i,j] ←q
8 return m[i,j]
```

Легко побачити, що час роботи Recursive-Matrix-Chain(p,1,n) залежить від n щонайменше експоненціально. Справді, позначимо його T(n) і приймемо, що час виконання рядків 1-2, а також 6-7, дорівнює одиниці. Тоді:

$$T\ (1)\ \geq\ 1\ ,$$

$$T\ (n)\ \geq\ 1\ +\ \sum_{k\ =\ 1}^{n\ -\ 1}\ (T\ (k)\ +\ T\ (n\ -\ k)\ +\ 1),\ \textit{if}\ \ n\ >\ 1\ .$$

У сумі по *k* кожне *T(i)*

(при *i = 1,2,..., n - 1*) зустрічається двічі, і ще існує *n - 1* одиниць. Виходить,

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n. \qquad (4.4)$$

Доведемо за індукцією, що $T(n) \le 2n-1$ для всіх $n \ge 1$. При $n = 1$ нерівність виконана, тому що $T(1) \ge 1 = 2°$. Крок індукції:

$$T(n) \ge 2\sum_{i=1}^{n-1} 2^{i-1} + n = 2\sum_{i=0}^{n-2} 2^{i} + n = 2(2^{n-1} - 1) + n = 2^{n} - 2 + n \ge 2^{n-1}$$

Ми бачимо, що алгоритм Recursive-Matrix-Chain вимагає експонентного часу. Причина в тому, що цей алгоритм багаторазово зустрічає однакові підзадачі й щораз вирішує їх заново. Різних підзадач усього лише $O(n^2)$, і маса часу губиться на зайву роботу. Метод динамічного програмування дозволяє цієї зайвої роботи уникнути.

Динамічне програмування «зверху вниз»

Алгоритм підрозділу 4.1 діяв «знизу вгору». Але той же прийом (виключення повторного розв'язання підзадач) можна реалізувати й для алгоритмів, що працюють «зверху вниз». Для цього потрібно запам'ятовувати відповіді до вже вирішених підзадач у спеціальній таблиці. Спочатку вся таблиця порожня (тобто заповнена спеціальними записами, що вказують на те, що відповідне значення ще не обчислене). Коли в процесі виконання алгоритму підзадача зустрічається в перший раз, її розв'язання заноситься в таблицю. Надалі розв'язання цієї підзадачі береться прямо з таблиці (в нашому прикладі таблицю відповідей завести легко, тому що підзадачі нумеруються парами (i,j), в більш складних випадках можна використати хешування.) Англійською мовою, цей прийом поліпшення рекурсивних алгоритмів називається memorization.

Застосуємо це вдосконалення до алгоритму Recursive-Matrix-Chain:

Memoized-Matrix-Chain(p,i,j)

```
1 n←length[p]-1
2 for i←1 to n
3       do for j←I to n
4             do m[i,j] ←∞
5 return Lookup-Chain(p,1,n)
```

Lookup-Chain(p,i,j)

```
1 if m[i,j]<∞
2       then return m[i,j]
3 if i=j
4       then m[i,j] ← 0
5       else for k ← i to j-1
6             do q ← Lookup-Chain(p,i,k) +
                + Lookup-Chain(p,k+1,j)+p_{i-1}p_k p_j
7                if q<m[i,j]
8                     then m[i,j] ← q
9 return m[i,j] ← q
```

Процедура Memoized-Matrix-Chain, подібно Matrix-Chain-Order, заповнює таблицю *m[1.. n, 1.. n]*, де *m[i,j]* – мінімальна кількість множень, необхідна для обчислення $A_{i..j}$. Спочатку $m[i,j] = \infty$ – це означає, що відповідне місце в таблиці не заповнено. Якщо при виконанні Lookup-Chain(p,i j) виявляється, що $m[i,j] < \infty$, то процедура відразу видає це значення *m[i,j]*. У протилежному випадку *m[i,j]* обчислюється як у процедурі Recursive-Matrix-Chain, записується в таблицю й видається як відповідь. Тим самим виклик Lookup-Chain(p,i,j) завжди повертає *m[i,j],* але обчислення проводяться тільки при першому такому виклику.

Рис. 4.2 ілюструє економію, що досягається заміною Recursive-Matrix-Chain на Memoized-Matrix-Chain. Заштриховані вершини дерева рекурсії відповідають тим випадкам, коли значення *m[i,j]* не обчислюється, а беруться прямо з таблиці.

Алгоритм Memoized-Matrix-Chain вимагає часу $O(n^3)$, як і алгоритм Matrix-Chain-Order. Справді, кожна із $\Theta(n^2)$ позицій таблиці один раз ініціалізується (рядок 4 процедури Memoized-Matrix-Chain) і один-єдиний раз заповнюється – при першому виклику *Lookup-Chain(p, i,j)* для даних *i, j*. Всі виклики *LookuP-Chain (p,i,j)* поділяються на перші й повторні. Кожний з $\Theta(n^2)$ перших викликів вимагає часу $O(n)$ (не включаючи часу роботи рекурсивних викликів LOOKUP-CHAIN для менших ділянок); загальний час роботи є $O(n^3)$. Кожний з повторних викликів вимагає часу *O(1);* їхнє число є $O(n^3)$ (обчислення для кожної із $O(n^2)$ комірок таблиці породжують $O(n)$ викликів). Тим самим рекурсивний алгоритм, що вимагає часу $\Omega(2^n)$, перетворився в поліноміальний, що потребує час $O(n^3)$.

Підведемо підсумки: задача про оптимальний порядок множення n матриць може бути вирішена за час $O(n^3)$ або «зверху вниз» (рекурсивний алгоритм із запам'ятовуванням відповідей), або «знизу вгору» (динамічне програмування). Обидва алгоритми засновані на перекритті підзадач; число підзадач є $\Theta(n^2)$, і обидва алгоритми вирішують кожну з підзадач лише одноразово.

Загалом кажучи, якщо кожна з підзадач повинна бути вирішена хоч раз, метод динамічного програмування («знизу вгору»), звичайно, ефективніше, ніж рекурсія із запам'ятовуванням відповідей, оскільки реалізація рекурсії (а також перевірка, є відповідь у таблиці або ще немає) вимагає додаткового часу. Але якщо для знаходження оптимуму не обов'язково вирішувати всі підзадачі, підхід «зверху вниз» має ту перевагу, що вирішуються лише ті підзадачі, які дійсно потрібні.

4.3 Найбільша загальна послідовність

Підпослідовність виходить із даної послідовності, якщо видалити деякі її елементи (сама послідовність також вважається своєю підпослідовністю). Формально: послідовність $Z = (z_1, z_2,...,z_k)$ називається підпос-

лідовністю (subsequence) послідовності $X = (X_1, X_2,...,X_n)$, якщо існує строго зростаюча послідовність індексів $(i_1,i_2,...., i_k)$, для якої $z_j = x_{ij}$ при всіх $j – l,2,...,k$. Наприклад, $Z=(B,C,D,B)$ є підпослідовністю послідовності $X = (A, B, C, B, D, A, B)$; відповідна послідовність індексів є (2,3,5,7). Відзначимо, що говорячи про послідовності, ми – на відміну від курсів математичного аналізу – маємо на увазі кінцеві послідовності.

Будемо говорити, що послідовність Z є загальною підпослідовністю (common subsequence) послідовностей X і B, якщо Z є підпослідовністю як X, так і Y. Приклад: $X = (A, B, C, B, D, A, B)$, $Y = \{B,D,C,A,B,A)$, $Z = (B,C,A)$. Послідовність Z у цьому прикладі – не найдовша із загальних підпослідовностей X і B (послідовність (B,C, B, A) довша). Послідовність (B,C,B,A) буде найбільшою загальною підпослідовністю для X і B, оскільки загальних підпослідовностей довжиною 5 у них немає. Найбільших загальних підпослідовностей може бути кілька. Наприклад, (B, D, A, B) – інша найбільша загальна підпослідовність X і Y.

Задача про найбільшу загальну підпослідовність (скорочено НЗП; англійською LCS = longest-common-subsequence) полягає в тому, щоб знайти загальну підпослідовність найбільшої довжини для двох даних послідовностей X і B. У цьому розділі ми покажемо як вирішити цю задачу за допомогою динамічного програмування.

Будова найбільшої загальної підпослідовності

Якщо вирішувати задачу про НЗП «в лоб», перебираючи всі підпослідовності послідовності X і перевіряючи для кожної з них, чи не буде вона підпослідовністю послідовності B, то алгоритм буде працювати експонентний час, оскільки послідовність довжиною m має 2^m підпослідовностей (стільки ж, скільки підмножин у множини $\{1,2,... ,m\}$).

Однак задача про НЗП має властивість оптимальності для підзадач, як показує теорема 4.1 (див. нижче). Підходяща множина підзадач – множина пар префіксів двох даних послідовностей. Нехай $X – (x_1,x_2,... ,x_m)$ – деяка послідовність. Її префікс (prefix) довжини i – це послідовність $X_i = (X_1,X_2,... ,X_i)$ *(при і від 0 до m)*. Наприклад, якщо $X = (A,B, C,B, D, A,B)$, те $X_4 = (A,B,C, B)$, а X_0 – порожня послідовність.

Теорема 4.1 (про будову НЗП). Нехай $Z = (z_1, z_2,...., z_k)$ – одна з найбільших загальних підпослідовностей для $X = (x_1,x_2,... ,x_m)$ і $Y = (y_1,y_2,...,y_n)$. Тоді:

а) якщо $x_m = y_n$, то $z_k = x_m = y_n$ і Z_{k-i} є НЗП для X_{m-1} і Y_{n-1};

б) якщо $x_m \neq y_n$ і $z_k \neq x_m$, то Z є НЗП для X_{m-1} і Y;

в) якщо $x_m \neq y_n$ і $z_k \neq y_n$, те Z є НЗП для X_m і Y_{n-1} .

Доведення

1. Якщо $z_k \neq x_m$, то ми можемо дописати $x_m = y_n$ у кінець послідовності Z і одержати загальну підпослідовність довжини $k + 1$, що суперечить умові. Виходить $z_k = x_m = y_n$. Якщо в послідовностей X_{m-1} і

Y_{n-1} є довша (ніж Z_{k-i}) загальна підпослідовність, то ми можемо дописати до неї $x_m = y_n$ і одержати загальну підпослідовність для X і B, більш довгу, ніж Z, чого бути не може.

2. Як тільки $z_k \neq x_m$, послідовність Z є загальною підпослідовністю для X_{m-1} і Y. Оскільки Z – НЗП для X і B, то вона тим паче є НЗП для X_{m-1} і Y.

3. Аналогічно 2.

Ми бачимо, що НЗП двох послідовностей містить у собі найбільшу загальну підпослідовність їхніх префіксів. Отже, задача про НЗП має властивість оптимальності для підзадач. Зараз ми переконаємося, що перекриття підзадач також має місце.

Рекурентна формула

Теорема 4.1 показує, що знаходження НЗП послідовностей $X = (x_1,x_2, ..., x_m)$ і $Y = (y_1, y_2, ..., y_n)$ зводиться до розв'язання або однієї, або двох підзадач. Якщо $x_m = y_n$, то досить знайти НЗП послідовностей X_{m-1} і Y_{n-1} і дописати до неї наприкінці $x_m = y_n$. Якщо ж $x_m \neq y_n$, то треба вирішити дві підзадачі: знайти НЗП для X_{m-1} і Y, а потім знайти НЗП для X і Y_{n-1}. Більш довга з них і буде служити НЗП для X і Y.

Тепер відразу видно, що виникає перекриття підзадач. Дійсно, щоб знайти НЗП X і B, нам може знадобитися знайти НЗП X_{m-1} і B, а також НЗП X і Y_{n-1}; кожна із цих задач містить підзадачу знаходження НЗП для X_{m-1} і Y_{n-1}. Аналогічні перекриття будуть зустрічатися й далі.

Як і в задачі перемножування послідовності матриць, ми почнемо з рекурентного співвідношення для вартості оптимального розв'язання. Нехай $c[i,j]$ позначає довжину НЗП для послідовностей X_i і Y_j. Якщо i або j дорівнює нулю, то одна із двох послідовностей порожня, отже $c[i,j] = 0$. Сказане вище можна записати так:

$$c[i,j] = \begin{cases} 0, & \text{якщо } i = 0 \text{ або } j = 0, \\ c[i-1, j-1] + 1, & \text{якщо } i, j > 0 \text{ або } x_i \neq y_j, \\ \max(c[i, j-1], c[i-1, j]), & i, j > 0 \text{ або } x_i \neq y_j. \end{cases} \quad (4.5)$$

Обчислення довжини НЗП

Виходячи зі співвідношення (4.5), легко написати рекурсивний алгоритм, що працює експонентний час і обчислює довжину НЗП двох даних послідовностей. Але оскільки різних підзадач усього $\Theta(mn)$, краще скористатися динамічним програмуванням.

Вихідними даними для алгоритму LCS-LENGTH служать послідовності $X = (x_1,x_2, ... ,x_m)$ і $Y = (y_1,y_2, ...,y_n)$. Числа $c[i,j]$ записуються в таблицю $c[0 .. m, 0 .. n]$ у такому порядку: спочатку заповнюється зліва направо перший рядок, потім другий, і т.д. Крім того, алгоритм запам'ятовує в таблиці $b[1.. m, 1.. n]$ «походження» $c[i,j]$: у клітинку $b[i,j]$ зано-

ситься стрілка, що вказує на клітинку з координатами *(i - 1, j - 1), (i - 1,j)* або *(i,j - 1),* залежно від того, чи дорівнює *c[i,j]* числу *c[i - 1, j - 1] + 1, c[i - 1, j]* або *c[i,j - 1]* (див. (16.5)). Результатами роботи алгоритму є таблиці *c* та *b*.

```
LCS-Length(X,Y)
1 m← length[X]
2 n← length[Y]
3 for i←1 to m
4       do c[i,0] ←0
5 for j←0 to n
6       do c[0,j] ←0
7 for i←1 to m
8       do for j←1 to n
9             do if xi=yi
10                  then c[i,j] ←c[i-1, j-1]+1
11                       b[i,j] ← "↖"
12                  else if c[i-1,j] ≥c[i, j-1]
13                        then c[i,j] ←c[i-1, j]
14                             b[i,j] ← "↑"
15                        else c[i,j] ←c[i, j-1]
16                             b[i,j] ← "←"
17 return c, b
```

На рис. 4.3 показана робота LCS-Length для *X = {A,B,C,B,D, A,B}* і *Y = {B,D,C,A,B,A}*.

Алгоритм LCS-LENGTH вимагає часу *O(mn)*: на кожну клітинку потрібно *O(1)* кроків.

У клітинці з координатами *(i,j)* записане число *c[i,j]* і стрілка *b[i,j]*. Число 4 у правій нижній клітинці є довжина НОП. При *i,j > 0* значення *c[i,j]* визначається тим, чи рівні x_i і y_j, і обчисленими раніше значеннями *c[i-l,j], c[i,j - 1] і c[i - 1, j - 1]*. Шлях по стрілках, детальніше розглянутий в [7,6], заштрихований. Кожна коса стрілка на цьому шляху відповідає елементу НЗП (ці елементи виділені).

Побудова НЗП

Таблиця *b*, створена процедурою LCS-Length, дозволяє швидко знайти НЗП послідовностей $X = (x_1,x_2, ...,x_m)$ і $Y = (y_1,y_2, .. ,y_n)$. Для цього треба пройти по шляху, зазначеному стрілками, починаючи з *b[m,n]*. Пройдена стрілка ↑ у клітинці (i,j) означає, що $x_i = y_j$ входить у найбільшу загальну підпослідовність. От як це реалізовано в рекурсивній процедурі Print-LCS (НЗП для *X* і *Y* друкується при виклику *Print-LCS(b, X, length[X], length[Y])*):

```
Print-LCS(b,X,i,j)
1 if i=0 або j=0
2       then return
3 if b [i,j] = "↖"
4       then Print-LCS(b, X, i-1, j-1)
5               надрукувати x_i
6 else if b[i,j] = "↑"
7       then Print-LCS(b, X, i-1, j)
8 else Print-LCS(b, X, i, j-1)
```

Будучи застосованою до таблиці рис. 4.3, ця процедура надрукує BCBA. Час роботи процедури є *O(m + n)*, оскільки на кожному кроці хоча б одне із чисел m і n зменшується.

	j	0	1	2	3	4	5	6
i		y_i	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	←1	↖ 1
2	B	0	↖ 1	←1	←1	↑ 1	↖ 1	←2
3	C	0	↑ 1	↑ 1	↖ 2	←2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Рисунок 4.3 Таблиці c і b, створені алгоритмом LCS-Length при *X = (A, B, C, B, D, A, B)* і *Y = (B, D, C, A, B, A)*

Поліпшення алгоритму

Після того, як алгоритм розроблений, нерідко вдається зробити його більш ощадливим. У нашому прикладі можна обійтися без таблиці b. Справді, кожне із чисел *c[i,j]* залежить від *c[i-l,j], c[i,j-1] і c[i-1, j-1]*. Знаючи *c[i,j]*, ми можемо за час *O(1)* з'ясувати, який із цих трьох записів був використаний. Тим самим можна знайти НЗП за час *O(m+n)* за допо-

могою однієї тільки таблиці *c*. При цьому ми заощаджуємо *O(mn)* пам'яті. (Втім, асимптотика не змінюється: об'єм таблиці є також *O(mn)*.)

Якщо нас цікавить тільки довжина найбільшої загальної підпослідовності, то стільки пам'яті не потрібно: обчислення *c[i,j]* стосується тільки двох рядків з номерами *i* та *i - 1* (це не межа економії: можна обійтися пам'яттю на один рядок таблиці *c* плюс ще трохи). Однак, при цьому саму підпослідовність знайти (за час *O(m+n)*) не вдається.

4.4 Контрольні питання

1. Сфери застосування динамічного програмування.
2. Як будується алгоритм, заснований на динамічному програмуванні?
3. Як застосувати динамічне програмування до задачі про множення послідовності матриць?
4. Сформулюйте задачу про множення послідовності матриць.
5. Оптимальне розміщення дужок: суть задачі та алгоритм її реалізації.
6. Структура оптимального розміщення дужок.
7. Поняття вартості оптимального розв’язання задачі оптимального розміщення дужок.
8. Алгоритм обчислення оптимальної вартості.
9. Поняття «перекриття підзадач».
10. Обчислення оптимальної вартості методом «знизу вгору».

4.5 Завдання

1. Знайдіть оптимальне розміщення дужок в задачі про перемножування матриць, якщо *p= (5,10,3,12,5,50,6)*.

2. Розробіть алгоритм Print-Optimal-Parens, що відображає оптимальне розміщення дужок. (Таблицю *s* уже обчислено алгоритмом Matrix-Chain-Order).

3. Як краще шукати оптимальний порядок перемножування матриць: перебираючи всі розміщення дужок і обчислюючи кількість множень для кожної з них або ж за допомогою алгоритму Recursive-Matrix-Chain? Відповідь обґрунтуйте.

4. Знайдіть найбільшу загальну підпослідовність (НЗП) послідовностей *(1,0,0,1,0,1,0,1)* і *(0,1, 0,1,1,0,1,1,0)*.

5. Розробіть алгоритм, що будує НЗП для $X = (x_1,x_2,\ldots ,x_m)$ і $Y = (y_1,y_2, \ldots, y_n)$ за час $O(m+n)$.

5 ЖАДІБНІ АЛГОРИТМИ

Для багатьох оптимізаційних задач є більш прості й швидкі алгоритми, ніж динамічне програмування. У цьому розділі ми розглядаємо задачі, які можна вирішувати за допомогою **жадібних алгоритмів** (greedy algorithms). Такий алгоритм робить на кожному кроці локально оптимальний вибір, з розрахунком, що підсумкове розв'язання також виявиться оптимальним. Це не завжди так, але для багатьох задач такі алгоритми дійсно дають оптимум. Як приклад – проста, але не цілком тривіальна задача про вибір заявок (підрозділ 5.1). Далі (підрозділ 5.2) ми обговорюємо, для яких задач ефективно застосовувати жадібні алгоритми. У подальшому ми не раз зустрінемося з жадібними алгоритмами (задача про мінімальне покриваюче дерево, алгоритм Дейкстри для пошуку найкоротших шляхів з даної вершини, жадібна евристика, задачі про покриття множин та інші). Мінімальні покриваючі дерева – класичний приклад використання жадібного алгоритму.

5.1 Задача про вибір заявок

Нехай дані n **заявок** на проведення занять в одній і тій же аудиторії. Два різних заняття не можуть перетинатися у часі. У кожній заявці зазначені початок і кінець заняття (s_i і f_i для і-ї заявки). Різні заявки можуть перетинатися, і тоді можна задовольнити тільки одну з них. Ми ототожнюємо кожну заявку із проміжком (s_i,f_i), так що кінець одного заняття може збігатися з початком іншого, і це не вважається перетином.

Формально кажучи, заявки з номерами i і j сумісні (compatible), якщо інтервали (s_i,f_j) і $[sj,fj)$ не перетинаються (іншими словами, якщо $f_j \le s_j$ або $f_j \le s_i$). **Задача про вибір заявок** (activity-selection problem) полягає в тому, щоб набрати максимальну кількість сумісних одна з одною заявок.

Жадібний алгоритм працює в такий спосіб. Ми припускаємо, що заявки впорядковані в порядку зростання часу закінчення:

$$f_i \le f_2 \le \ldots \le f_n. \quad (5.1)$$

Якщо це не так, то можна відсортувати їх за час $O(nlogn)$; заявки з однаковим часом кінця розташовуємо в довільному порядку. Тоді алгоритм виглядає так (f і s - масиви):

```
Greedy-Activity-Selector (s, f)
1 n ← legth[s]
2 A← {1}
3 j←1
4 for i←2 to n
```

```
5       do if s_i≥f_j
6               then A←A∪{i}
7                       j←i
8 return A
```

Робота цього алгоритму показана на рис. 5.1. Множина A складається з номерів обраних заявок, j – номер останньої з них; при цьому

$$f_j = \max\{fk : k \in A\}, \tag{5.2}$$

оскільки заявки відсортовані за зростанням часу закінчення.

Спочатку A містить заявку номер 1, і $j = 1$ (рядка 2-3). Далі (цикл у рядках 4-7) шукається заявка, що починається не раніше закінчення заявки номер j. Якщо така знайдена, вона включається в множину A і змінній j присвоюється її номер (рядка 6-7).

Алгоритм GREEDY-ACTIVITY-SELECTOR вимагає всього лише $\Theta(n)$ кроків (не вважаючи попереднього сортування). Як і личить жадібному алгоритму, на кожному кроці він робить вибір так, щоб вільний час, що залишається, був максимальним.

Правильність алгоритму

Не для всіх задач жадібний алгоритм дає оптимальне розв'язання, але для нашої дає. Переконаємося в цьому.

Теорема 5.1. *Алгоритм* GREEDY-ACTIVITY-SELECTOR *дає набір з найбільшої можливої кількості спільних заявок.*

Доведення. Нагадаємо, що заявки відсортовані за зростанням часу закінчення. Насамперед доведемо, що існує оптимальне розв'язання задачі про вибір заявок, що містить заявку номер 1 (із самим раннім часом закінчення).

Справді, якщо в якійсь оптимальній множині заявок заявка номер 1 не втримується, то можна замінити в ньому заявку із самим раннім часом закінчення на заявку номер 1, що не зашкодить спільності заявок (тому що заявка номер 1 закінчується раніше, ніж попередня, і ні з чим перетнутися не може) і не змінить їхньої загальної кількості. Отже, можна шукати оптимальну множину заявок A серед утримуючих заявку номер 1: існує оптимальне розв'язання, що починається з жадібного вибору.

Після того як ми домовилися розглядати тільки набори, що містять заявку номер 1, всі неспільні з нею заявки можна викинути, і задача зводиться до вибору оптимального набору заявок із множини заявок, що залишилися (спільних із заявкою номер 1). Інакше кажучи, ми звели задачу до аналогічної задачі з меншим числом заявок. Розмірковуючи за індукцією, одержуємо, що, роблячи на кожному кроці жадібний вибір, ми прийдемо до оптимального розв'язання.

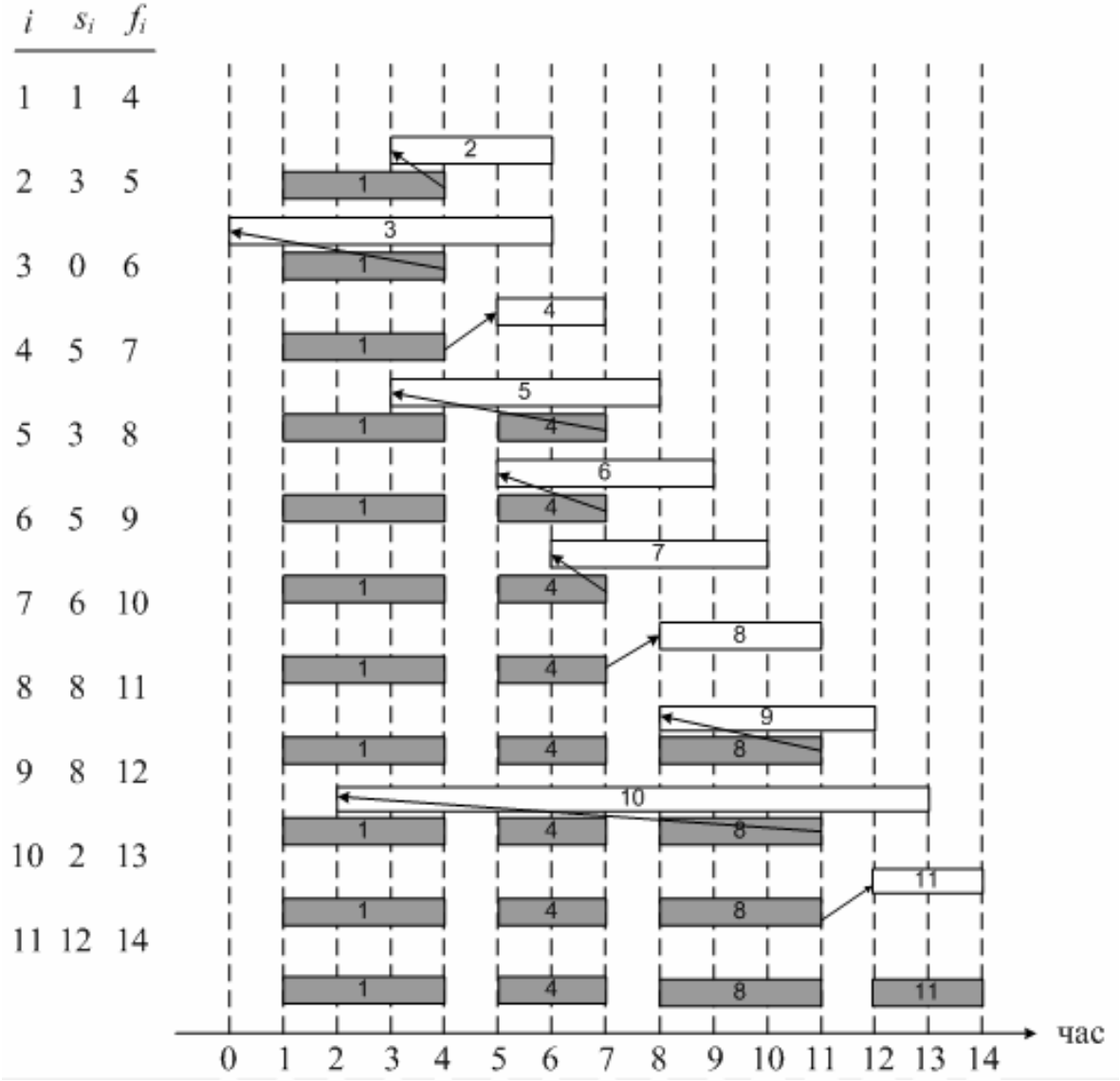


Рисунок 5.1 – Робота алгоритму GREEDY-ACTIVITY-SELECTOR для 11 заявок (таблиця зліва).

Кожний рядок на рисунку відповідає одному проходу циклу в рядках 4-7. Сірі заявки вже включені в *A,* біла зараз розглядається. Якщо лівий кінець білого прямокутника лівіше правого кінця правого сірого (стрілка йде вліво), то заявка відкидається, у протилежному випадку вона додається до *A*

5.2 Випадки застосування жадібних алгоритмів

Як довідатися чи дасть жадібний алгоритм оптимум стосовно до даної задачі? Загальних рецептів тут немає, але існують дві особливості, характерні для задач, розв'язуваних жадібними алгоритмами. Це принцип жадібного вибору й властивість оптимальності для підзадач.

Принцип жадібного вибору

Говорять, що до оптимізаційної задачі застосуємо **принцип жадібного вибору** (greedy-choice property), якщо послідовність локально оптимальних (жадібних) виборів дає глобально оптимальне розв'язання. Розходження між жадібними алгоритмами й динамічним програмуванням можна пояснити так: на кожному кроці жадібний алгоритм бере «самий жирний шматок», а потім уже намагається зробити найкращий вибір серед тих, що залишилися, які б вони не були; алгоритм динамічного програмування приймає розв'язання, прорахувавши заздалегідь наслідки для всіх варіантів.

Як довести, що жадібний алгоритм дає оптимальне розв'язання? Це не завжди тривіально, але в типовому випадку таке доведення потрібне схемі, використаній в доведенні теореми 5.1. Спочатку ми доводимо, що жадібний вибір на першому кроці не закриває шляхи до оптимального розв'язання: для всякого розв'язання є інше, погоджене з жадібним вибором і не гірше першого. Потім показується, що підзадача, що виникає після жадібного вибору на першому кроці, аналогічна вихідній, і міркування завершується за індукцією.

Оптимальність для підзадач

Інакше кажучи, розв'язувані за допомогою жадібних алгоритмів задачі мають властивість **оптимальності для підзадач** (have optimal substructure): оптимальне розв'язання всієї задачі містить у собі оптимальні розв'язання підзадач. (Із цією властивістю ми вже зустрічалися, кажучи про динамічне програмування.) Наприклад, при доведенні теореми 5.1 ми бачили, що якщо A – оптимальний набір заявок, що містить заявку номер 1, то $A' = A \setminus \{1\}$ – оптимальний набір заявок для меншої множини заявок S', що складається з тих заявок, для яких $s_i \geq f_1$.

Жадібний алгоритм або динамічне програмування?

І жадібні алгоритми, і динамічне програмування ґрунтуються на властивості оптимальності для підзадач, тому може виникнути спокуса застосувати динамічне програмування в ситуації, де вистачило б жадібного алгоритму, або навпаки, застосувати жадібний алгоритм до задачі, у якій він не дасть оптимуму. Ми проілюструємо можливі пастки на прикладі двох варіантів класичної оптимізаційної задачі.

Дискретна задача про рюкзак (0-1 knapsack problem) полягає в такому. Нехай злодій пробрався на склад, на якому зберігається n речей. Річ з номером i коштує v_i доларів і важить w_i кілограмів (v_i і w_i – цілі числа). Злодій хоче вкрасти товару на максимальну суму, причому максимальна вага, що він може винести в рюкзаку, дорівнює W (число W теж ціле). Що він повинен покласти в рюкзак?

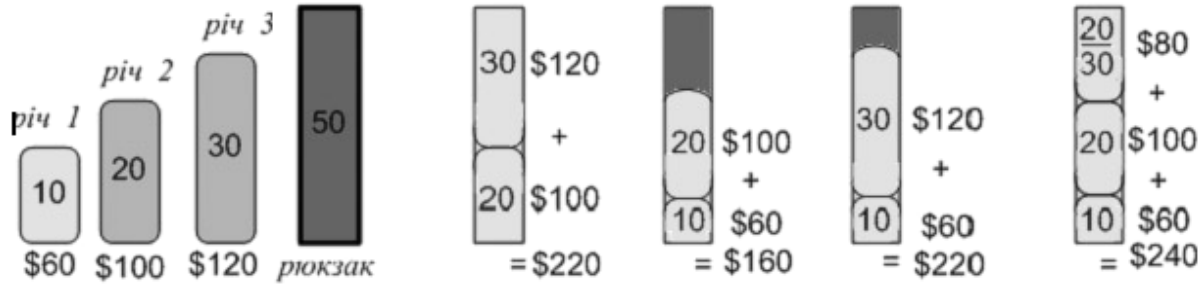
Неперервна задача про рюкзак (fractional knapsack problem) відрізняється від дискретної тем, що злодій може дробити крадені товари на частини й укладати в рюкзак ці частини, а не обов'язково брати речі цілими (якщо в дискретній задачі злодій має справу із золотими злитками, то в неперервній – із золотим піском).

Обидві задачі про рюкзак мають властивість оптимальності для підзадач. Справді, розглянемо дискретну задачу. Вийнявши річ номер *j* з оптимально завантаженого рюкзака, отримаємо розв'язання задачі про рюкзак з максимальною вагою $W - w_j$ і набором з $n - 1$ речей (всі речі, крім *j*-ї). Аналогічне міркування проходить і для неперервної задачі: вийнявши з оптимально завантаженого рюкзака, у якому лежить *w* кілограмів товару номер *j,* одержимо оптимальне розв'язання неперервної задачі, у якій максимальна вага дорівнює $W - w$ (замість *W)*, а кількість j-го товару дорівнює $w_j - w$ (замість w_j*)*.

Хоча дві задачі про рюкзак і схожі, жадібний алгоритм дає оптимум у неперервній задачі про рюкзак і не дає в дискретній. Справді, розв'язання неперервної задачі про рюкзак за допомогою жадібного алгоритму виглядає так. Обчислимо ціни (розраховуючи на кілограм) всіх товарів (ціна товару номер *г* дорівнює v_i/w_i*)*. Спочатку злодій бере максимум найдорожчого товару; якщо весь цей товар скінчився, а рюкзак не заповнений, злодій бере наступний за ціною товар, і так далі, поки не набере ваги *W*. Оскільки товари треба спочатку відсортувати за цінами, на що піде час *O(n logn)*, час роботи описаного алгоритму буде *O(n logn),* що дійсно дає оптимум.

Щоб переконатися в тім, що аналогічний жадібний алгоритм не зобов'язаний давати оптимум у дискретній задачі про рюкзак, подивіться на рис. 5.2,а. Вантажопідйомність рюкзака 50 кг, на складі є три речі, що важать 10, 20 і 30 кг і вартістю 60, 100 і 120 доларів, відповідно. Ціна їх розраховуючи на одиницю ваги дорівнює 6, 5 і 4. Жадібний алгоритм для початку покладе в рюкзак річ номер 1; однак оптимальне розв'язання включає предмети номер 2 і 3.

Для неперервної задачі з тими ж вихідними даними жадібний алгоритм, що пропонує почати з товару номер 1, дає оптимальне розв'язання (рис. 5.2, в). У дискретній задачі така стратегія не спрацьовує: поклавши в рюкзак предмет номер 1, злодій втрачає можливість заповнити рюкзак «під зав'язку», а порожнє місце в рюкзаку знижує ціну накраденого розраховуючи на одиницю ваги. Тут, щоб вирішити, чи класти дану річ у рюкзак, треба зрівняти розв'язання двох підзадач: коли дана річ свідомо лежить у рюкзаку, і коли цієї речі в рюкзаку свідомо немає. Тим самим дискретна задача про рюкзак породжує множину підзадач, що перекриваються – типову ознаку задач, для яких ефективно застосовувати динамічне програмування.



а) б) в)

Рисунок 5.2 – У дискретній задачі про рюкзак жадібна стратегія може не спрацювати (а); злодій повинен вибрати дві речі із трьох для того, щоб їхня сумарна вага не перевищила 50 кг (б)

Оптимальний вибір – друга й третя речі; якщо покласти в рюкзак першу, то вибір оптимальним не буде, хоча саме вона дорожча всіх розраховуючи на одиницю ваги, (в). Для неперервної задачі про рюкзак з тими ж вихідними даними вибір товарів в порядку зменшення ціни на одиницю ваги буде оптимальний.

5.3 Контрольні питання

1. Жадібні алгоритми: основні поняття та означення.
2. Сфери застосування жадібних алгоритмів.
3. Сформулюйте задачу про вибір заявок.
4. Сформулюйте теорему про правильність розглянутого алгоритму GREEDY-ACTIVITY-SELECTOR.
5. Приведіть доведення теореми про правильність алгоритму GREEDY-ACTIVITY-SELECTOR.
6. Приведіть приклади технічного застосування задачі про вибір заявок.
7. Порівняйте можливості застосування жадібних алгоритмів та динамічного програмування для оптимізаційних задач.
8. Сформулюйте дискретну задача про рюкзак та охарактеризуйте її особливості.
9. Сформулюйте неперервну задачу про рюкзак та охарактеризуйте її особливості.
10. Застосування жадібних алгоритмів для дискретних задач.

5.4 Завдання

1. Задачу про вибір заявок можна вирішувати за допомогою динамічного програмування, обчислюючи послідовно
   (для $i = 1, 2,..., n$) число m_i – максимально можливе число спільних заявок серед заявок з номерами $1,2,...,i$. Порівняєте час роботи такого алгоритму й алгоритму GREEDY-ACTIVITY-SELECTOR.

2. Нехай як і раніше є множина заявок на проведення занять (кожна заявка вказує початок і кінець), але аудиторій скільки завгодно. Потрібно розподілити заявки по аудиторіях, використовуючи якнайменше аудиторій. Розробіть ефективний жадібний алгоритм, що розв'язує цю задачу.
3. Для задачі про вибір заявок є кілька способів жадібного вибору, і не всі вони годяться. Покажіть на прикладі, що правила «на кожному кроці вибирати заявку найменшої тривалості, спільну із уже обраними», а також «на кожному кроці вибирати заявку, спільну з найбільшою кількістю залишків», не підходять.
4. Доведіть, що для неперервної задачі про рюкзак виконаний принцип жадібного вибору.
5. Розробіть заснований на динамічному програмуванні алгоритм для розв'язання дискретної задачі про рюкзак. Алгоритм повинен працювати за час $O(n)$ (n – кількість речей, W – максимальна вага рюкзака).
6. Нехай у дискретній задачі про рюкзак речі можна впорядкувати таким чином, щоб одночасно виконувалися нерівності $w_1 \le w_2 \le \le \ldots\ldots \le w_n$ і $v_1 \ge v_2 \ge \ldots \ge v_n$. Розробіть ефективний алгоритм для знаходження оптимального набору й доведіть, що він правильний.

6 NP-ПОВНІ АЛГОРИТМИ

Всі розглянуті нами раніше алгоритми були поліноміальними (працювали за поліноміальний час). Це значить, що час роботи алгоритму на вході довжини n становив не більше $O(n^k)$ для деякої константи k (не залежної від довжини входу). Природно поцікавитися чи всяка задача може бути розв'язана за поліноміальний час.

Відповідь на це питання негативна – деякі задачі взагалі не можуть бути розв'язані ніяким алгоритмом. Класичний приклад такої задачі– «проблема зупинки» (з'ясувати чи зупиняється дана програма на даному вході). Крім того, бувають задачі, для яких існує вирішальний їхній алгоритм, але такий алгоритм працює довго – час його роботи не є $O(n^k)$ ні для якого фіксованого числа k.

Якщо ми хочемо провести нехай грубу, але формальну границю між «практичними» алгоритмами й алгоритмами, що становлять лише теоретичний інтерес, то клас алгоритмів, що працюють за поліноміальний час, є розумним першим наближенням.

У цьому розділі ми розглянемо клас задач, які називають «NP-повними». Для цих задач не знайдені поліноміальні алгоритми, однак, і не доведено, що таких алгоритмів не існує. Вивчення NP-повних задач

пов'язане з так званим питанням $P \neq NP$. Це питання було поставлено в 1971 році і є зараз однією з найцікавіших і складних проблем теорії обчислень.

Більшість фахівців вважають, що NP-повні задачі не можна розв’язати за поліноміальний час. Справа в тому, що якщо хоча б для однієї NP-повної задачі існує вирішальний її поліноміальний алгоритм, то й для всіх NP-повних задач такі алгоритми існують. На сьогоднішній день відомо дуже багато NP-повних задач. Всі спроби знайти для них поліноміальні алгоритми виявилися невдалими. Очевидно, таких алгоритмів немає зовсім.

Навіщо програмістові знати про NP-повні задачі? Якщо для деякої задачі вдається довести її NP-повноту, є підстави вважати її практично такою, що не має розв’язку. У цьому випадку краще витратити час на побудову наближеного алгоритму, ніж продовжувати шукати швидкий алгоритм, що розв’язує її точно.

Багато цікавих й практично важливих задач є NP-повними, хоча на перший погляд нітрохи не складніші, ніж задача сортування, задача про найкоротший шлях у графі або про максимальний потік у мережі (для яких поліноміальні алгоритми існують).

6.1 Поліноміальний час розв’язування

Як ми вже казали, поняття задачі, що розв'язується поліноміально, прийнято вважати уточненням ідеї «практично розв'язної» задачі. Чим це пояснюється?

По-перше, поліноміальні алгоритми, які використовують на практиці, звичайно, дійсно працюють досить швидко. Звичайно, важко назвати практично розв'язною задачу, що вимагає часу n^{100}. Однак поліноми такого ступеня в реальних задачах майже не зустрічаються.

Другий аргумент на користь розгляду класу поліноміальних алгоритмів – той факт, що об'єм цього класу не залежить від вибору конкретної моделі обчислень (для досить широкого класу моделей). Наприклад, клас задач, які можуть бути розв’язані за поліноміальний час на послідовній машині з довільним доступом (RAM), збігається із класом задач, поліноміально розв'язних на машинах Тьюрінга (означення яких можна знайти в книзі Хопкрофта й Ульмана [2,5]). Клас буде тим же й для моделей паралельних обчислень, якщо, звичайно, число процесорів обмежене поліномом від довжини входу.

По-третє, клас поліноміально розв'язних задач має природні властивості замкнутості. Наприклад, композиція двох поліноміальних алгоритмів (вихід першого алгоритму подається на вхід другого) також працює за поліноміальний час. Пояснюється це тим, що сума, добуток і композиція багаточленів знову є багаточлен.

Абстрактні задачі

Ми приймаємо таку абстрактну модель обчислювальної задачі. Будемо називати **абстрактною задачею** (abstract problem) довільне бінарне відношення Q між елементами двох множин – множини умов (instances) I і множини рішень (solutions) S. Наприклад, у задачі SHORTEST-PATH (пошук найкоротшого шляху між двома заданими вершинами деякого неорієнтованого графу $G=(V, E)$) умовою (елементом I) є трійка, що складається із графу й двох вершин, а розв'язанням (елементом S) – послідовність вершин, що становлять необхідний шлях у графі. При цьому один елемент множини I може перебувати у відношенні Q з декількома елементами множини S (якщо найкоротших шляхів між даними вершинами декілька).

У теорії NP-повноти розглядаються тільки **задачі розв'язування** (decision problems) – задачі, у яких потрібно дати відповідь «так» або «ні» на деяке питання. Формально задачу розв'язування можна розглядати як функцію, що відображає множину умов I у множині {0,1} (1 = «так», 0 = =«ні»). Багато задач можна тим або іншим способом перетворити до такого вигляду. Наприклад, із задачею пошуку найкоротшого шляху в графі зв'язана задача розв'язування PATH: «за заданим графом $G=(V,E)$, парою вершин i, $v \in V$ і натуральним числом k визначити, чи існує в G шлях між вершинами u і v, довжина якого не перевищує k». Нехай четвірка $i = (G,u,v,k)$ є умовою задачі PATH. Тоді PATH(i) = 1, якщо довжина найкоротшого шляху між вершинами u і v не перевищує k, і PATH(i) = 0 у гіршому випадку.

Часто зустрічаються **задачі оптимізації** (optimization problems), в яких потрібно мінімізувати або максимізувати значення певної величини. Перш ніж порушувати питання про NP-повноту таких задач, їх варто перетворити в задачу розв'язування. Звичайно, як таку задачу розв'язування розглядають задачу перевірки, чи є деяке число верхньою (або нижньою) границею для оптимізованої величини. Так, наприклад, ми перейшли від задачі оптимізації SHORTEST-PATH до задачі розв'язування PATH, додавши в умову задачі границю довжини шляху k.

Якщо після цього виходить NP-повна задача, то тим самим встановлена складність вихідної задачі. Справді, якщо для оптимізаційної задачі є швидкий алгоритм, то й отриману з неї задачу розв'язування можна розв'язати швидко (треба просто зрівняти відповідь цього алгоритму із заданою границею). Таким чином, теорію NP-повноти можна використати й для дослідження задач оптимізації.

Подання даних

Перш ніж подавати на вхід алгоритму вихідні дані (тобто елемент множини f), треба домовитися про те, як вони подаються в «зрозумілому для комп'ютера вигляді»; ми будемо вважати, що вихідні дані закодовані

послідовністю бітів. Формально кажучи, поданням (encoding) елементів деякої множини *S* називається відображення *e* з *S* у множину бітових рядків. (Зрозуміло, замість бітових рядків можна було б розглядати послідовності символів будь-якого іншого кінцевого алфавіту, що має не менше двох символів.) Наприклад, натуральні числа 0,1,2,3,... звичайно, подають бітовими рядками 0,1,10,11,100,... (при цьому, наприклад, *e(17) = 10001*). У комп'ютерах для подання букв і інших символів використовують код ASCII (рідше – EBCDIC). Так, *e(A)* = 1000001 у коді ASCII.

Фіксуючи подання даних, ми перетворюємо абстрактну задачу в рядкову**,** для якої вхідними даними є бітовий рядок, що є вихідними даними абстрактної задачі. Будемо говорити, що алгоритм **вирішує** (solves) рядкову задачу за час *O(T(n)),* якщо на вхідних даних (бітовому рядку) *i* довжини *n* алгоритм працює час *O(T(n)).* Рядкова задача називається **поліноміальною** (polynomial-time solvable), якщо існує константа *k* й алгоритм, що вирішує цю задачу за час $O(n^k)$.

Складним класом (complexity class) P називається множина всіх рядкових задач розв'язування, які можуть бути вирішені за поліноміальний час.

Бажаючи визначити поняття поліноміальної абстрактної задачі, ми стикаємося з тим, що можливі різні подання даних. Для кожного подання *e* множини *I* входів абстрактної задачі *Q* ми отримуємо свою рядкову задачу, яку ми надалі позначаємо *e(Q).* Загалом кажучи, при одному поданні може вийти поліноміальна рядкова задача, а при іншому – ні. (Тут є ще одна – чисто технічна деталь: деякі бітові рядки можуть не мати ніяких вихідних даних; що потрібно в цьому випадку від алгоритму, який вирішує рядкову задачу? Будемо вважати, що він повинен давати в цьому випадку відповідь 0, тобто що властивість *e(Q)* хибна для таких входів).

Повернемося до питання про те, як залежить поліноміальність задачі від обраного подання. Нехай, наприклад, входом задачі є натуральне число *k,* і час роботи алгоритму є $\Theta(k)$. Якщо число *k* подати в **системі числення з основою 1** (unary representation), тобто у вигляді послідовності *k* одиниць, то час роботи алгоритму на вході довжини *n* буде дорівнювати *O(n),* і алгоритм буде поліноміальним. При більш природному двійковому поданні числа *k* час роботи на вході довжини *n* (тобто на *n*-бітовому числі) буде дорівнювати $\Theta(2^n)$, і алгоритм не буде поліноміальним.

Однак на практиці (якщо виключити такі «дорогі» способи подання, як система числення з основою 1) природні способи подання виявляються, звичайно, еквівалентними, оскільки вони можуть бути швидко (поліноміально) перетворені один в одного. Наприклад, двійковий запис числа можна перетворити в трійковий і навпаки за поліноміальний час.

Будемо казати, що функція *f: {0,1}* ==> {0,1}** **обчислюється** за **поліноміальний час** (is polynomial-time computable), якщо існує поліноміальний алгоритм *A,* що для будь-якого $x \in \{0,1\}^*$ видає результат *f(x).*

Розглянемо тепер множину *I* умов довільної абстрактної задачі розв'язування. Два подання e_1 і e_2 цієї множини називаються **поліноміально зв'язаними** (polynomially related), якщо існують дві обчислювані за поліноміальний час функції f_{12} і f_{21}, для яких $f_{12}(e_1(i)) = e_2(i)$ і $f_{21}(e_2(i)) = e_i(i)$ для будь-якого $i \in I$. Це значить, що e_1-подання входу може бути за поліноміальний час отримано з e_2-подання й навпаки. У цьому випадку не має значення, яке із двох поліноміально зв'язаних подань вибрати, як показує наступна лема.

Лема 6.1. *Нехай Q – абстрактна задача розв'язування із множиною умов I, а, e_1 і e_2-поліноміально зв'язані подання для елементів множини I. Припустимо, що множина всіх рядків, які є e_1-поданнями елементів Q, розв'язується за поліноміальний час, і що аналогічна властивість виконана для подання e_2. Тоді властивості $e_1(Q) \in$ P і $e_2(Q) \in P$ рівносильні.*

Доведення. Твердження симетричне, так що досить довести його в одну сторону. Припустимо, що задача $e_1(Q)$ розв'язується за час $O(n^k)$ для деякого фіксованого числа *k.* За припущенням для будь-якої умови $i \in I$ подання $e_1(i)$ може бути отримане з подання $e_2(i)$ за час $O(n^c)$ (де c – деяка константа, $n = |e_2(i)|$). Для розв'язання задачі $e_2(Q)$, одержавши на вхід $e_2(i)$, ми спершу обчислимо $e_1\{i)$, а потім застосуємо алгоритм, що дозволяє $e_1(Q)$, до рядка $e_1(i)$. Скільки часу займе наше обчислення? Перетворення $e_2(i)$ в $e_1(i)$ вимагає поліноміального часу. Отже, $|e_i(i)| = O(n^c)$, оскільки довжина виходу алгоритму не перевищує часу його роботи. Розв'язання задачі з умовою $e_1(i)$ займає $O(|e_1(i)|^k) = O(n^{ck})$ часу. Отже, час обчислення виявився поліноміальним. (Ми пропустили важливий момент: одержавши на вході деякий рядок, ми повинні спочатку перевірити, що він є e_2-поданням деякого входу; за припущенням це можна зробити за поліноміальний час.)

Ми не будемо докладно описувати використовуване подання в конкретних задачах, вважаючи, що воно обрано досить розумно й ощадливо (цілі числа задаються двійковим записом, кінцеві множини – списком елементів і т.п.). Подання об'єкта будемо позначати кутовими дужками: *(G)* – це стандартне подання об'єкта *G.* При цьому множина всіх рядків, що є поданнями, виявляється поліноміально розв'язною (існує поліноміальний алгоритм, що перевіряє по рядку, чи є він яким-небудь об'єктом), а різні «розумні» способи подання даних виявляються поліноміально зв'язаними, так що можна скористатися лемою 6.1 і не описувати подання детально, якщо нас цікавить лише питання про поліноміальність задачі. Таким чином, надалі ми не будемо робити різниці між абстрактною зада-

чею і її рядковим поданням, як це, звичайно, й роблять (крім тих рідкісних задач, у яких стандартне подання не очевидно – для них вибір подання може сильно вплинути на складність розв'язання задачі).

Формальні мови

Для задач розв'язування зручно використати термінологію теорії формальних мов. **Алфавітом** (alphabet) Σ називається будь-який скінченний набір символів. **Мовою *L* над алфавітом** Σ (language L over Σ) називається довільна множина рядків символів з алфавіту Σ (такі рядки називають **словами** в алфавіті Е). Наприклад, можна розглянути $\Sigma = \{0,1\}$ і мова $L = \{10,11,101, 111, 1011,1101,10001,... \}$, що складається із двійкових записів простих чисел. Ми будемо позначати символом Е **порожнє слово** (empty string), яке не має символів, а символом 0 – **порожню мову** (empty language), що не містить слів. Мова, що складається із всіх рядків в алфавіті Е, позначається Σ^*. Наприклад, при $S = \{0,1\}$ маємо $\Sigma^* = \{e, 0,1, 00, 01,10,11, 000,. .. \}$. Таким чином, усяка мова L над Σ є підмножиною множини Σ^*.

Є кілька стандартних операцій над мовами. Операції **об'єднання** (union) і **перерізу** (intersection) мов визначаються як звичайні операції об'єднання й перерізу множин. **Доповненням** (complement) мови L називають мову $L = \Sigma^* \setminus L$. **Конкатенацією** (concatenation) або **об'єднанням** двох мов L_1 і L_2 називається мова

$$L = \{x_1x_2: x_1 \in L_1, x_2 \in L_2\}.$$

Замиканням (closure) мови L називається мова

$$L^* = \{E\} \cup L \cup L^2 \cup L^3 \cup,$$

де L – мова, отримана k-кратною конкатенацією мови L сама з собою. Операція замикання називається також ***– операцією Кліні** (Kleene star).

Тепер можна сказати, що задача розв'язування (точніше, що відповідає їй строкова задача розв'язування) є мовою над алфавітом Е = {0,1}. Наприклад, задачі PATH відповідає мова PATH = {$(G, u, v, k) : G = (V,E)$ – неорієнтовний граф, $i, v \in V; k \geq 0$ – ціле число, і в графі G існує шлях з u в v, довжина якого не перевищує k}.

(Ми будемо використовувати ту саму назву – у цьому випадку PATH – для позначення задачі й відповідної мови.)

Продовжимо знайомство з термінологією теорії формальних мов. Кажуть, що алгоритм A **приймає слово** $x \in \{0,1\}^*$ (accepts a string x), якщо на вході x алгоритм видає результат 1 $(A(x) = 1)$. Алгоритм A **відкидає** (rejects) **слово** x, якщо $A\{x) = 0$. (Зауважимо, що алгоритм може не зупи-

нитися на вході x або дати відповідь, відрізнити від 0 і 1. У цьому випадку він і не приймає, і не відкидає слово x). Алгоритм A **приймає мову** L (language L is accepted by A), якщо алгоритм приймає ті й тільки ті слова, які належать мові L.

Алгоритм A, який приймає деяку мову L, не зобов'язана відкидати буль-яке слово $x \notin L$. Якщо алгоритм приймає всі слова з L, а всі інші слова відкидає, говорять, що A **розпізнає мову** L (language L is decided by A). Мова L **приймається за поліноміальний час** (is accepted in polynomial time), якщо є алгоритм A, що приймає дану мову, причому всяке слово $x \in L$ приймається алгоритмом за час $O(n^k)$, де n – довжина слова x, a, k – деяке не залежне від x число. Мова L **розпізнається за поліноміальний час** (is decided in polynomial time), якщо деякий алгоритм A розпізнає дану мову, причому час роботи алгоритму на кожному слові довжини n не більше $O(n^k)$. Розглянута нами мова PATH приймається за поліноміальний час. Неважко побудувати алгоритм, що методом пошуку в ширину за поліноміальний час знаходить найкоротший шлях між вершинами u і v у графі G, а потім порівнює довжину знайденого шляху з даним в умові числом k. Якщо довжина шляху не перевищує k, алгоритм видає 1 і зупиняється. У протилежному випадку алгоритм зациклюється, не видаючи ніякої відповіді. Ясно, що такий алгоритм приймає, але не розпізнає мови PATH. Однак легко виправити описаний алгоритм таким чином, щоб слова, що не належать мові, відкидалися (якщо довжина найкоротшого шляху перевищує k, треба відкидати вхідне слово). Такий алгоритм приймає й розпізнає мову PATH.

Відмітимо, що для деяких мов (наприклад, для множини всіх програм, що закінчують свою роботу) є алгоритм, що приймає, але немає алгоритму, що розпізнає (маються на увазі алгоритми без якого-небудь обмеження на час роботи).

Ми вже дали означення класу задач P. Нижче ми дамо означення також класу NP. У теорії складності обчислень розглядаються багато інших **складних класів** (complexity classes). Наприклад, є важливий клас PSPAOE, що складається із задач, розв'язуваних алгоритмами з використанням пам'яті поліноміального розміру, або клас EXP, що складається із задач, які можна вирішити за час $O(2^n)$. Неформально, складний клас можна визначити як сімейство мов, для яких алгоритми, що розпізнають, мають **задану ступінь** складності (complexity measure), наприклад, заданий час роботи.

Тепер можна переформулювати означення складного класу P так: P = $\{L \subset \{0,1\}^*$: існує алгоритм A, який розпізнає мову L за поліноміальний час$\}$.

Насправді в даній ситуації немає різниці між мовами, які прийнятні і розпізнаються за поліноміальний час.

Теорема 6.1. P = *{L : L приймається за поліноміальний час}.*

Доведення. Якщо мова розпізнається деяким алгоритмом, то він і приймається тим же алгоритмом. Залишається довести, що якщо мова *L* приймається поліноміальним алгоритмом *A*, то він розпізнається деяким (можливо, іншим) поліноміальним алгоритмом *A′.* Нехай алгоритм *A* приймає мову *L* за час $O(n^k)$. Це значить, що існує константа *c*, для якої *A* приймає будь-яке слово довжиною $n \subset L$, зробивши не більше $T = cn^k$ кроків. (Формально кажучи, це правильно для досить довгих слів *x*; ми опускаємо очевидні деталі.) З іншого боку, слова не з *L* алгоритм не приймає (ні за який час). Новий алгоритм *A′* моделює роботу алгоритму *A* і враховує число кроків цього алгоритму, порівнюючи його з відомою границею *T*. Якщо за час *T* алгоритм *A* приймає слово *x,* алгоритм *A′* також приймає це слово й видає 1. Якщо ж *A* не приймає *x* за зазначений час, то алгоритм *A′* припиняє моделювання й відкидає слово (видає 0). Уповільнення роботи за рахунок моделювання й підрахунку кроків не таке й велике і залишає час роботи поліноміальним.

Зауважимо, що доведення теореми 6.1 неконструктивне: якщо ми маємо поліноміальний алгоритм *A,* який приймає мову *L,* але не знаємо, яким саме поліномом обмежений час його роботи, ми не можемо побудувати алгоритм *A′,* а можемо лише стверджувати, що такий алгоритм існує.

6.2 Перевірка належності класу NP

У попередньому розділі ми розглядали задачу розв’язування PATH (з'ясувати, чи є в даному графі *G* між даними двома вершинами *u* і *v* шлях довжиною не більше *k).* Ця задача виявилася поліноміальною (і вирішується за допомогою алгоритму пошуку в ширину). Уявимо собі, однак, що ми нічого не знаємо про пошук в ширину. Тоді задача PATH буде для нас важкою: бачачи граф *G*, вершини *u* і *v,* і знаючи число *k,* ми не будемо знати чи є шуканий шлях поки нам хто-небудь його не покаже. Але якщо нам його покажуть, то ми можемо просто перевірити, що шлях цей – шуканий.

Саме така ситуація має місце для задач із класу NP, про які ми будемо говорити в цьому розділі.

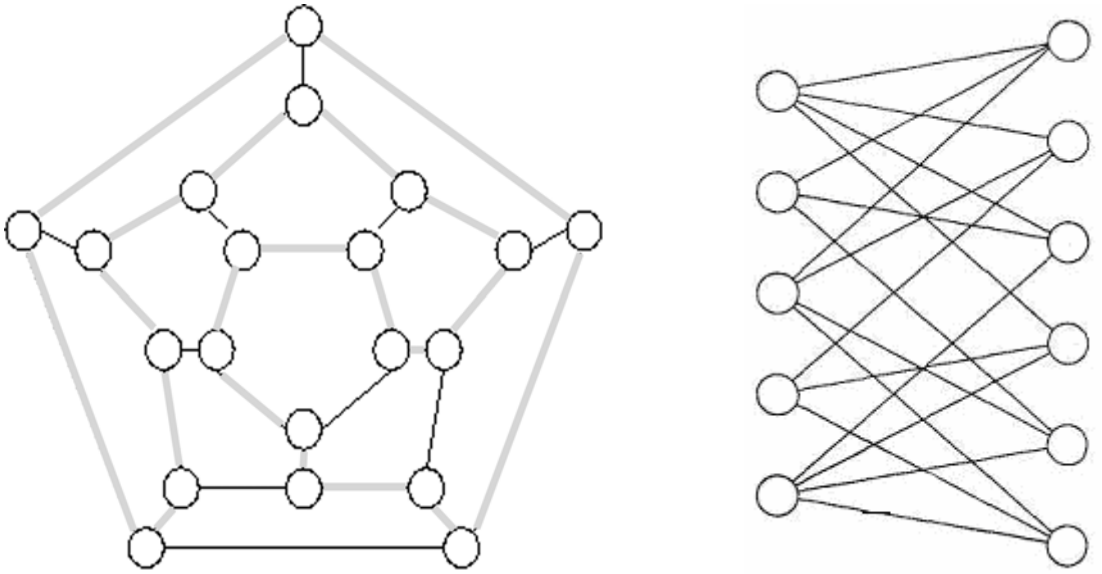
Гамільтонів цикл

Задача пошуку гамільтонового циклу в графі вивчається вже більше ста років. Бонді й Мерті цитують лист Гамільтона (W.R.Hamilton), у якому описана така гра: перший гравець відзначає в додекаедрі (рис. 6.1(а)) шлях з п'яти вершин, які йдуть одна за одною, а другий гра-

вець повинен доповнити цей шлях до простого циклу, що проходить через всі вершини додекаедра.

Взагалі **гамільтоновим циклом** (hamiltonian cycle) у неорієнтовному графі *G = {V, E)* називається простий цикл, що проходить через всі вершини графу. Графи, у яких є гамільтонів цикл, називають гамільтоновими (hamiltonian graph).

Додекаедр (проекція якого зображена на рис. 6.1(а)) – гамільтонів граф; сірими кольорами показаний один з гамільтонових циклів. Не всі графи гамільтонові: на рис. 6.1(б) показаний двочастковий граф з непарним числом вершин; зразу видно, що такий граф не може мати гамільтонового циклу.



а) б)

Рисунок 6.1 – а – додекаедр (вигляд із точки, розташованої недалеко від центра однієї із граней); сірі ребра утворять гамільтонів цикл; б – двочастковий граф з непарним числом вершин – такий граф не гамільтонів

Задача про гамільтоновий цикл (hamiltonian-cycle problem) складається з з'ясування чи має даний граф *G* гамільтонів цикл. Формально кажучи,

HAM-CYCLE = {(G) : G – гамільтонів граф}.

Як вирішувати таку задачу? Можна перебрати всі перестановки вершин даного графу й перевірити чи є хоча б одна з них гамільтоновим циклом. Оцінимо час роботи такого алгоритму. Якщо ми використаємо подання графу за допомогою матриці інцидентності, то число вершин ***m*** у графі буде $\Omega(\sqrt{n})$, де $n = |(G)|$ – довжина подання графу *G*. Є *m!* різних

перестановок вершин графу, і час роботи алгоритму дорівнює $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, тобто не є поліноміальним. Таким чином, вказаний алгоритм не дає ефективного розв'язання задачі. Насправді, задача про гамільтоновий цикл є NP-повною, і тому можна припускати, що поліноміального алгоритму для неї взагалі не існує.

Перевірка належності мові

Нехай ви заперечуєте колезі, який стверджує, що (нарисований перед вами на дошці) граф є гамільтоновим. При цьому ви не можете швидко перевірити, так це чи ні. Проте приятель може виграти парі, якщо якимось чином відгадає гамільтонів цикл і покаже його вам. Перевірка того, що даний цикл є гамільтоновим, проста. Потрібно лише перевірити, що поданий цикл проходить через всі вершини графу, і що він дійсно йде по ребрах. Отже, доведення існування гамільтонового циклу в графі (що полягає в поданні гамільтонового шляху) можна перевірити за поліноміальний час.

Назвемо **перевірним алгоритмом** (verification algorithm) *алгоритм A* з двома аргументами; перший аргумент ми будемо називати (як і раніше) вхідним рядком, а другий – сертифікатом (certificate). Ми говоримо, що алгоритм *A* з двома аргументами приймає вхід *x {A* verifies an input string *x),* якщо існує сертифікат *y,* для якого *A(x,y)* = 1. **Мовою, що перевіряє алгоритм** *A* (language verified by *A),* ми назвемо мову

$$L - \{x \in \{0,1\}^* : \text{існує } y \in \{0,1\}^*, \text{ для якого } A(x,y) = 1\}.$$

Інакше кажучи, алгоритм *A* перевіряє мову *L,* якщо для будь-якого рядка $x \in L$ знайдеться сертифікат *y*, за допомогою якого *A* може перевірити належність *x* до мови *L,* а для $x \notin L$ такого сертифіката немає. Наприклад, у задачі HAM-CYCLE сертифікатом була послідовність вершин, що утворить гамільтонів цикл.

Складний клас NP

Складний клас NP (complexity class *NP*) – це клас мов, для яких існують перевірні алгоритми, що працюють за поліноміальний час, причому довжина сертифіката також обмежена деяким поліномом. Більш точно, мова *L* належить класу *NP*, якщо існує такий поліноміальний алгоритм *A* з двома аргументами й такий багаточлен *p(x)* із цілими коефіцієнтами, що

$$L = \{x E| \{0,1\}^* : \text{існує сертифікат } y, \text{ для якого } |y| <= p(|x|) \text{ і } A(x, y) = 1\}.$$

У цьому випадку ми говоримо, що алгоритм *A* перевіряє мову *L* за поліноміальний час *(A* verifies language *L* in polynomial time). Зауважимо,

що, формально кажучи, із цього не випливає, що A перевіряє мову L у сенсі раніше даного означення, тому що для $x \notin L$ можуть існувати довгі сертифікати, які відкидаються при новому означенні, але суперечать старому.

Кілька слів про назву: скорочення *NP* походить від англійських слів Nondeterministic Polynomial (time), що перекладається як недетермінований поліноміальний час. Спочатку клас *NP* був визначений в термінах так званих недетермінованих обчислень (див. книгу Хопкрофта й Ульмана [5]).

Ми вже знаємо одну задачу із класу *NP* – це задача HAM-CYCLE. Крім того, будь-яка задача з *P* належить також і *NP*. Дійсно, якщо є поліноміальний алгоритм, що розпізнає мову, то легко побудувати алгоритм, що перевіряє, для тієї ж мови – алгоритм, що перевіряє, може просто ігнорувати свій другий аргумент (сертифікат). Таким чином, P з NP.

Тепер невідомо чи збігаються класи *P* і *NP*, але більшість фахівців думає, що ні. Інтуїтивно клас *P* можна уявляти собі як клас задач, які можна швидко вирішити, а клас NP – як клас задач, розв'язання яких може бути швидко перевірене. На практиці вирішити самому задачу часто набагато складніше, ніж перевірити вже готове розв'язання, особливо якщо час роботи обмежений. За аналогією можна думати, що в класі NP є задачі, які не можна вирішити за поліноміальний час.

Є й більше серйозні причини думати, що $P \neq NP$ – насамперед це існування «NP-повних» задач.

Крім проблеми *P=NP*, залишається відкритими й багато інших питань про клас NP. Так, незважаючи на величезні зусилля дослідників, не відомо чи замкнуть клас NP щодо доповнення, тобто чи правильно, що з $L \subseteq NP$ треба $L \subseteq NP$. Ми можемо визначити складний клас co-NP як множина мов L, для котрих $L \subseteq NP$. Питання про замкнутість класу NP щодо доповнення можна переформулювати так: чи рівні класи NP і co-NP?

Оскільки клас P замкнутий щодо переходу до доповнення, $P \subseteq NP \cap co\text{-}NP$. У той же час залишається невідомим чи правильна рівність $P = NP \cap co\text{-}NP$. На рис. 6.2 показані чотири можливих ситуації.

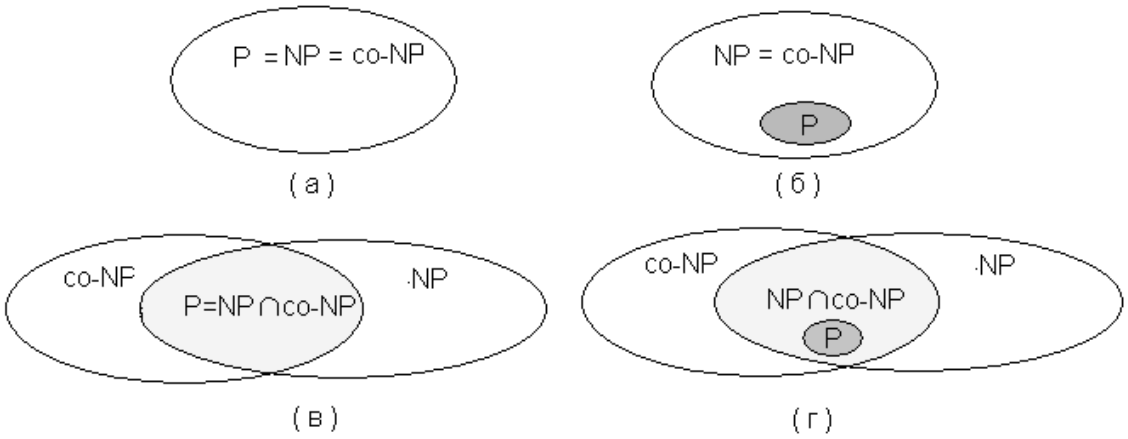


Рисунок 6.2 – Чотири можливих ситуації (мається на увазі, що всі показані на рисунку області не порожні)

а – класи *P, NP* і *co-NP* збігаються (більшість експертів вважає це найменш імовірним); б – клас *NP* замкнутий щодо доповнення (*NP = co-NP*), але не збігається з *P*; в – клас *NP* не замкнутий щодо доповнення й у перерізі з *co-NP* дає *P*; г – клас *NP* не замкнутий щодо доповнення й перерізу *NP* ∩ *co- NP* більше, ніж *P*

На жаль, наші знання про співвідношення класів *P* і *NP* далеко не повні. Але вже поняття NP-повноти є важливим засобом класифікації задач; як ми побачимо, воно зводить питання про складності даної задачі до загального (нехай і не вирішеного) питання про співвідношення класів *P* і *NP***.**

6.3 NP-повнота і зведення

Очевидно, найбільш переконливим аргументом на користь того, що класи *P* и *NP* різні, є існування так званих NP-повних задач. Цей клас має дивну властивість: якщо яка-небудь NP-повна задача розв'язна за поліноміальний час, то і всі задачі із класу *NP* розв'язні за поліноміальний час, тобто *P = NP*. Незважаючи на багаторічні дослідження, ні для одної NP-повної задачі не знайдений поліноміальний розв'язний алгоритм.

Зокрема, задача HAM-PATH (про гамільтоновий цикл) є NP-повною. Таким чином, навчившись вирішувати її за поліноміальний час, ми одержимо поліноміальні алгоритми для всіх задач класу *NP*. Переформулювання: якщо *NP* \ *P* не порожні, то *HAM-PATH* ∈ *NP* \ *P*.

Неформально кажучи, NP-повні мови є самими «важкими» у класі *NP*. При цьому труднощі мов можна порівнювати, зводячи одну мову до іншої. У цьому розділі ми даємо означення звідності, потім визначаємо клас NP-повних мов і встановлюємо повноту однієї конкретної мови, CIRCUIT-SAT.

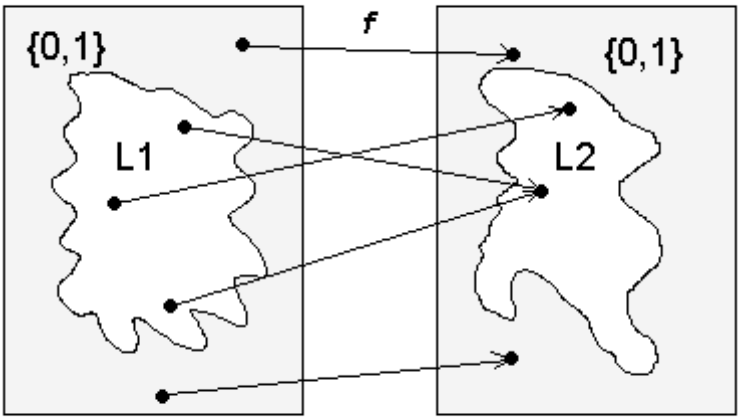


Рисунок 6.3 – Мова L_1 зводиться за поліноміальний час до мови L_2: властивості $x \notin L_1$ і $f(x) \notin L_2$ рівносильні

Звідність

Кажучи неформально, задача Q зводиться до задачі Q', якщо задачу Q можна вирішити для будь-якого входу, вважаючи відомим розв'язання задачі Q' для якогось іншого входу. Наприклад, задача розв'язання лінійного рівняння зводиться до задачі розв'язання квадратного рівняння (лінійне рівняння можна перетворити у квадратне, додавши фіктивний старший член). Якщо задача Q зводиться до задачі Q', то будь-який алгоритм, що вирішує Q^1, можна використати для розв'язання задачі Q, тобто Q «не складніше» Q'.

Дамо формальне означення. Говорять, що мова L_1 **зводиться за поліноміальний час** (is polynomial-time reducible) до мови L_2 (запис: $L_1 \le_p L_2$), якщо існує така функція f: $\{0,1\}^* \Longrightarrow \{0,1\}^*$, обчислювана за поліноміальний час, що для будь-якого $x \in \{0,1\}^*$,

$$x \in L_x \Longleftrightarrow f(x) \in L_2. \qquad (6.1)$$

Функцію f називають **звідною функцією** (reduction function) , а поліноміальний алгоритм F, що обчислює функцію f, **– звідним алгоритмом** (reduction algorithm).

Рис. 6.3 ілюструє означення звідність мови L_1 до мови L_2 за поліноміальний час. Кожна мова є підмножина множини $\{0,1\}^*$. Звідна функція перетворює будь-яке *слово* $x \in L_1$ у *слово* $f(x) \in L_2$, а *слово* $x \in L_1$ – в *слово* $f(x) \notin L_2$– тому, якщо ми довідаємося, чи належить слово $f(x)$ мові L_2, ми тим самим одержимо відповідь на питання про належність слова x мові L_1.

Лема 6.2. *Якщо мова* $L_1 \subseteq \{0,1\}^*$ *зводиться за поліноміальний час до мови* $L_2 \subseteq \{0,1\}^*$ *і* $L_2 \in P$, *то* $L_1 \in \boldsymbol{P}$.

Доведення. Нехай A_2 – поліноміальний алгоритм, що розпізнає мову L_2, а F – поліноміальний алгоритм, що зводить мову L_1 до мови L_2. Побудуємо алгоритм A_1, що буде за поліноміальний час приймати мову L_1.

Рис. 6.4 ілюструє побудову. Одержавши вхід $x \in \{0,1\}$, алгоритм A_1 (за допомогою алгоритму F) одержує $f(x)$ і за допомогою алгоритму A_2 перевіряє чи належить слово $f(x)$ мові L_2. Результат роботи алгоритму A_2 на слові $f(x)$ і видається алгоритмом A_1 як відповідь.

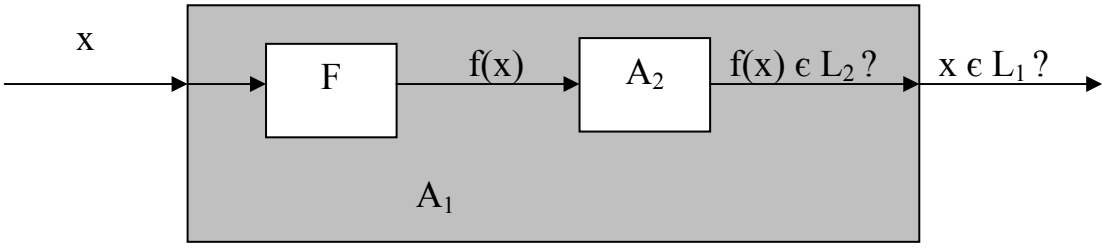


Рисунок 6.4 – Належність слова x до мови L_1 можна перевірити, використавши звідний алгоритм F і алгоритм A_2, що приймає *мову* L_1

Означення (6.1) гарантує, що алгоритм A_1 дає правильну відповідь; він поліноміальний, оскільки поліноміальні алгоритми F і A_1.

NP-повнота

Поняття зведення дозволяє додати точне розуміння твердження про те, що одна мова не менш важка ніж інша (з точністю до полінома). Запис $L_1 \le_p L_2$ можна інтерпретувати так: складність мови L_1 не більш ніж поліноміально перевищує складність мови L_2. Найбільш важкі в цьому розумінні NP-повні задачі.

Мова $L \subseteq \{0,1\}^*$ називається **NP-повною** (NP-complete), якщо:

а) $L \in NP$;

б) $L' \le_p L$ для будь-якого $L' \in NP$.

Клас NP-повних мов будемо позначати NPC. Мови, які мають властивість 2 (але не обов'язково мають властивість 1), називають **NP-важкими** (NP-hard).

Основна властивість NP-повних мов полягає в такому.

Теорема 6.2. *Якщо деяка NP-повна задача розв'язна за поліноміальний час, то P = NP. Інакше кажучи, якщо в класі NP існує задача, що не розглянута за поліноміальний час, то всі NP-повні задачі такі.*

Доведення. Нехай L – NP-повна мова, що одночасно виявилася розв'язною за поліноміальний час ($L \in P$ і $L \in NPC$). Тоді для будь-якої мови $L' \in NP$ за властивістю 2 означення NP-повної мови маємо $L' \le_p L$. Отже, $L' \in P$ (лема 36.3), і перше твердження теореми доведене. Друге твердження теореми є переформулюванням першого.

Таким чином, гіпотеза $P \neq NP$ означає, що NP-повні задачі не можуть бути вирішені за поліноміальний час. Більшість експертів думає, що це дійсно так; передбачуване співвідношення між класами *P, NP* і *NPC* показано на рис. 6.5.

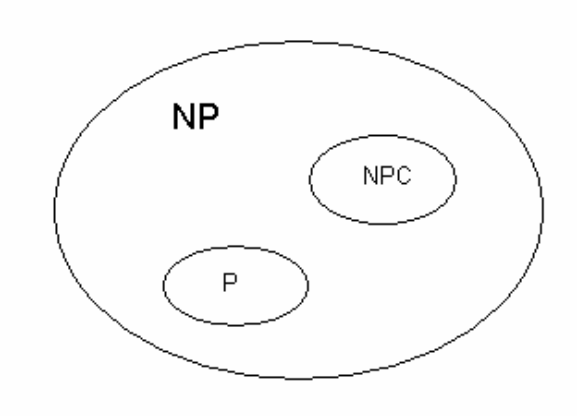


Рисунок 6.5 – Передбачуване співвідношення між класами *P, NP* і *NPC*. Класи *P* и *NPC* містяться в *NP* (що очевидно), не перетинаються й не покривають усього *NP*

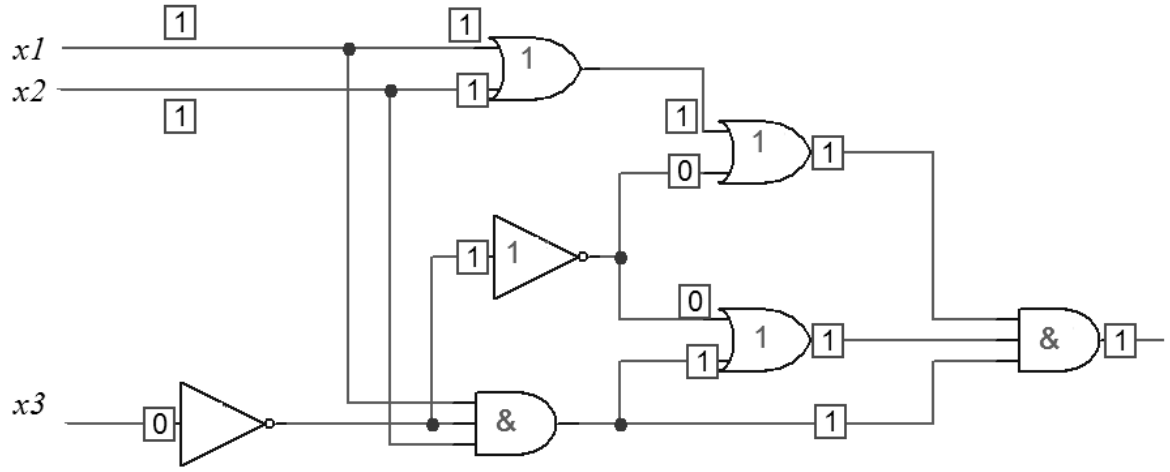
Звичайно, ми не можемо бути впевнені, що колись хто-небудь не надасть поліноміальний алгоритм для розв’язання NP-повної задачі й не доведе тим самим, що $P = NP$. Але поки що цього нікому не вдалося, і тому доведення NP-повноти деякої задачі є переконливим аргументом на користь того, що вона є практично нерозв'язною.

Задача про виконуваність для схем

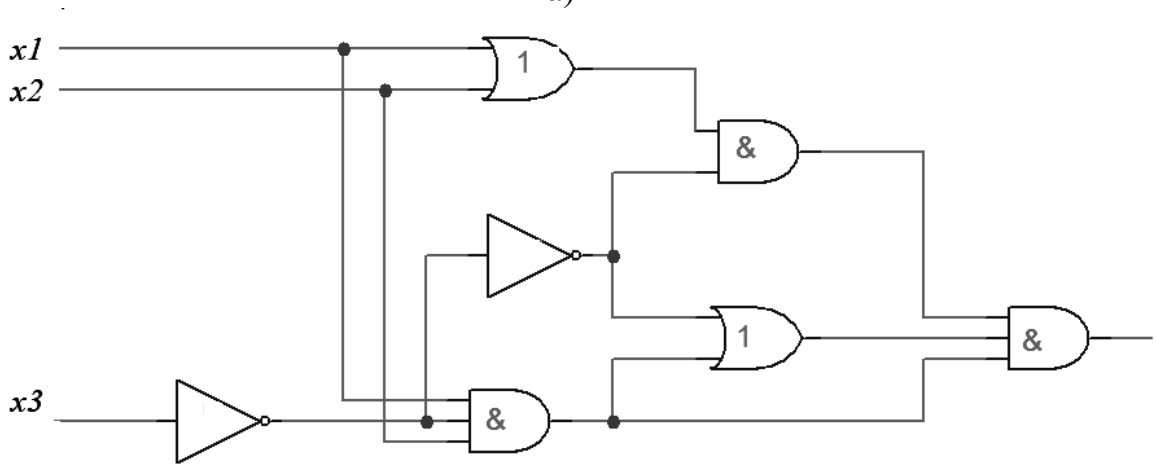
Як ми вже говорили, ми почнемо з однієї конкретної задачі; встановивши її NP-повноту, можна доводити NP-повноту інших задач. Як приклад ми розглянемо **задачу про виконуваність схеми** (circuit-satisfiability problem або CIRCUIT-SAT). На жаль, докладне доведення NP-повноти цієї задачі вимагає розгляду технічних деталей, що виходить за рамки даного посібника. Тому ми обмежимося неформальним припущенням доведення, вважаючи, що читач має уявлення про схеми з функціональних елементів.

На рис. 6.6 показані дві схеми з функціональних елементів. Обидві мають по три входи й по одному виходу.

Будемо розглядати **набори значень** булевих змінних, відповідним входам схем (truth assignments). Схема з функціональних елементів з одним виходом називається **здійсненною** (satisfiable), якщо існує **виконуючий набір** (satisfying assignment), тобто такий набір значень входів, при якому на виході схеми з'являється одиниця.



а)



б)

Рисунок 6.6 – Два набори вхідних даних для задачі CIRCUIT-SAT.
а – набір вхідних значень *($x_1 = 1, x_2 = 1, x_3 = 0$)* дає значення 1 на виході, так що ця схема здійсненна;
б – для цієї схеми ніякий набір вхідних значень не приводить до одиниці на виході, так що ця схема не здійсненна

Наприклад, схема рис. 6.6, а має виконуючий набір *{$x_1 = 1, x_2 = 1, x_3 = 0$)* і тому є здійсненною. У той же час ніякі значення змінних x_1, x_2, x_3 для схеми рис. 6.6, б не приводять до появи 1 на виході. Отже, ця схема нездійсненна.

Задача про виконуваність схеми (circuit-satisfiability problem) вимагає з'ясувати чи є дана схема, складена з елементів І, АБО й НІ, виконуваною. Звичайно, потрібно домовитися про спосіб подання булевих схем за допомогою рядків бітів – це робиться природно (подібно тому, як це робиться для графів); при цьому розмір отриманого рядка бітів не

більш ніж поліноміально залежить від розміру схеми. Зафіксувавши таке подання, розглянемо мову *CIRCUIT-SAT = {(C) :C* – виконувана схема з функціональних елементів.

Задачу про виконуваність можна сформулювати так: чи можна дану схему замінити на еквівалентну, у якій вихід з'єднаний прямо з нульовим проводом.

Зрозуміло, можна довідатися чи здійсненна дана схема, перебравши всі можливі комбінації значень входів. На жаль, їх багато: для схеми з k входами прийдеться перебрати 2^k наборів – час роботи такого переборного алгоритму не обмежено поліномом (від k або, також від розміру схеми). Як ми вже відзначали, більшість фахівців впевнені, що задача про виконуваність схеми не може бути вирішена поліноміальним алгоритмом (оскільки NP-повна).

Для початку переконаємося, що задача CIRCUIT-SAT належить класу NP.

Лема 6.3. *Задача CIRCUIT-SAT належить класу NP.*

Доведення. Сертифікатом є набір вхідних значень, при яких вихідне значення дорівнює 1, ясно що цей факт легко перевірити за поліноміальний час.

Для доведення NP-повноти задачі CIRCUIT-SAT нам залишилося довести, що дана задача NP-важка, тобто що будь-яка задача із класу NP зводиться до задачі CIRCUIT-SAT за поліноміальний час. Повне доведення цього твердження досить громіздке, так що ми дамо лише припущення доведення.

Коротко нагадаємо як влаштований комп'ютер. Програма зберігається в пам'яті комп'ютера у вигляді послідовності інструкцій (команд). Інструкції містять код операції, яку потрібно виконати, адреси аргументів (операндів) команди й адреси, за якими варто записати результат операції. У спеціальному місці пам'яті, що називається **лічильником команд** (program counter), зберігається адреса поточної команди. Ця адреса в процесі виконання програми змінюється (поступово збільшуючись при виконанні звичайних команд або перескакуючи в інше місце програми в умовних операторах і циклах).

Поточний стан програми, що визначає подальший хід її виконання, записано в кожний момент у пам'яті (ми включаємо в це поняття регістри процесора, лічильник команд і т.п.). Кожний стан пам'яті комп'ютера будемо називати **конфігурацією** (configuration). Виконання інструкцій може розглядатися як послідовне перетворення поточної конфігурації в наступну за нею відповідно до деяких правил. Ці правила визначаються електронікою комп'ютера й можуть бути подані у вигляді схеми з функціональних елементів (якщо конфігурація подається набором n бітів, то

ця схема має n входів і n виходів). У доведенні наступної леми ми будемо позначати цю схему через M.

Лема 6.4. *Задача CURCUIT-SAT є NP-важкою.*

Доведення. Нехай L – довільна мова із класу NP. Ми побудуємо функцію f, що буде відображати кожне двійкове слово в схему $C = f(x)$, для якої $x \in L$ рівносильне $C \in$ CIRCUIT-SAT. (Функція f буде обчислюватися поліноміальним алгоритмом).

Оскільки $L \in NP$, існує алгоритм A, який перевіряє дану мову за поліноміальний час. Ми використаємо цей алгоритм для побудови звідного алгоритму F. Нехай $T(n)$ – найбільший час роботи алгоритму A на входах довжини n (нагадаємо, що ми називаємо входом перший аргумент алгоритму A; другий називається сертифікатом). Виберемо таке число k, що $T(n) = O(n^k)$ і довжина сертифікатів для входів довжини n теж є $O(n^k)$. (Час роботи алгоритму поліноміально залежить від загальної довжини умови й сертифіката. Але оскільки довжина сертифіката обмежена поліномом від довжини умови, час роботи алгоритму також обмежено поліномом від n.)

Ідея доведення складається в поданні роботи алгоритму A у вигляді послідовності конфігурацій. Як показано на рис. 6.7, кожна конфігурація складається із частини, що містить програму, лічильника команд, стану регістрів процесора, входу x, сертифіката y і робочої пам'яті.

Початковий стан пам'яті комп'ютера утворить конфігурацію C_0. Потім кожна конфігурація C_i перетвориться в наступну конфігурацію C_{i+1}, причому це перетворення виконується за допомогою схеми з функціональних елементів M, реалізованої в електроніці комп'ютера. Наприкінці роботи алгоритм одержує відповідь (нуль або одиницю). Будемо вважати, що ця відповідь записується в певне місце пам'яті. Після одержання відповіді програма зупиняється, і стан пам'яті надалі не змінюється. Отже, якщо алгоритм робить не більше $T(n)$ кроків, результат роботи алгоритму перебуває у фіксованому біті конфігурації $C_{T(n)}$.

Таким чином, можна побудувати схему, що послідовно обчислює конфігурації c , c_{2} ,..., $c_{T(N)}$. Ця схема складається з послідовно з'єднаних $T(n)$ копій схеми M. Вихід i-ї копії схеми M буде конфігурацією c_i. Вихідними значеннями схеми будуть біти конфігурації $c_{T(n)}$. Описану схему назвемо C'.

Згадаємо, що повинен робити звідний алгоритм F. Отримавши вхідне слово x, він повинен побудувати схему $C = f(x)$, яка виконується тоді і тільки тоді, якщо існує такий сертифікат y, що $A(x,y) = 1$. Для цього знайдемо довжину n слова x і побудуємо схему C описаним способом. Входом схеми C буде початкова конфігурація обчислення $A(x,y)$, а виходом – конфігурація $c_{(n)}$.

Потім отриману схему C' варто трішки перетворити. По-перше, потрібно фіксувати біти, що відповідають програмі *A,* початковому положенню лічильника команд, входу *x,* вихідному стану службової інформації й робочої частини пам'яті. (Для цього ми з'єднуємо відповідні входи схеми з постійними сигналами 0 або 1.) Невизначеними залишаються тільки ті вхідні змінні, які відповідають за значення сертифіката *y.* По-друге, ігноруються всі виходи схеми *C',* крім одного – того біта конфігурації $c_{T(n)}$, у якому зберігається результат роботи алгоритму *A.* Нова, перетворена схема і є потрібна нам схема *C.* Звідний алгоритм, який отримавши вхід *x,* будує схему *C* и видає її подання у вигляді рядка бітів.

Залишається довести, що звідний алгоритм має дві необхідні властивості. По-перше, потрібно показати, що схема *C* виконується тоді й тільки тоді, коли існує сертифікат y, для якого $A(x, y) = 1$. По-друге, потрібно показати, що алгоритм *F* працює поліноміальний час.

Доведемо першу властивість. Нехай існує сертифікат *y* довжини $O(n^к)$, для якого $A(x, y) = 1$. Підставимо біти сертифіката *y* у вхідні змінні схеми *C.* Тоді вихід схеми $C(y)$ буде дорівнювати $A(x, y) = 1$. Таким чином, якщо існує сертифікат, то схема $C = f(x)$ здійсненна. Навпаки, якщо схема *C* здійсненна, то для деякого набору *y* значень вхідних змінних маємо $C(y) = 1$ і тому $A(x, y) = 1$. Коректність алгоритму *F* доведена.

Залишилося довести, що розмір схеми *C* і час роботи звідного алгоритму поліноміально залежать від розміру входу $n=|x|$. Насамперед зауважимо, що кожна конфігурація містить поліноміальне число бітів.

Кожна конфігурація відповідає стану обчислення в якийсь момент часу й містить у собі (крім *A, x* і *y)* також значення лічильника команд (PC), а також вміст регістрів процесора й робочої пам'яті. Кожна конфігурація переводиться в наступну за допомогою схеми *M* з функціональних елементів. Вихідний біт записаний у виділеному місці робочої пам'яті.

Дійсно, довжина програми *A* фіксована й не залежить від входу, довжина входу *x* дорівнює *n*, а довжина сертифіката *y* поліноміально залежить від *n.* Оскільки алгоритм *A* робить тільки поліноміальне число кроків, обсяг використовуваної ним пам'яті теж поліноміальний. (Ми вважаємо, що не тільки число фактично використаних комірок пам'яті, але й загальне число комірок пам'яті поліноміальне). Таким чином, і число рівнів на рис. 6.7, і розмірність кожного рівня поліноміальна; конструкція схеми також регулярна й побудова її за поліноміальний час не викликає труднощів.

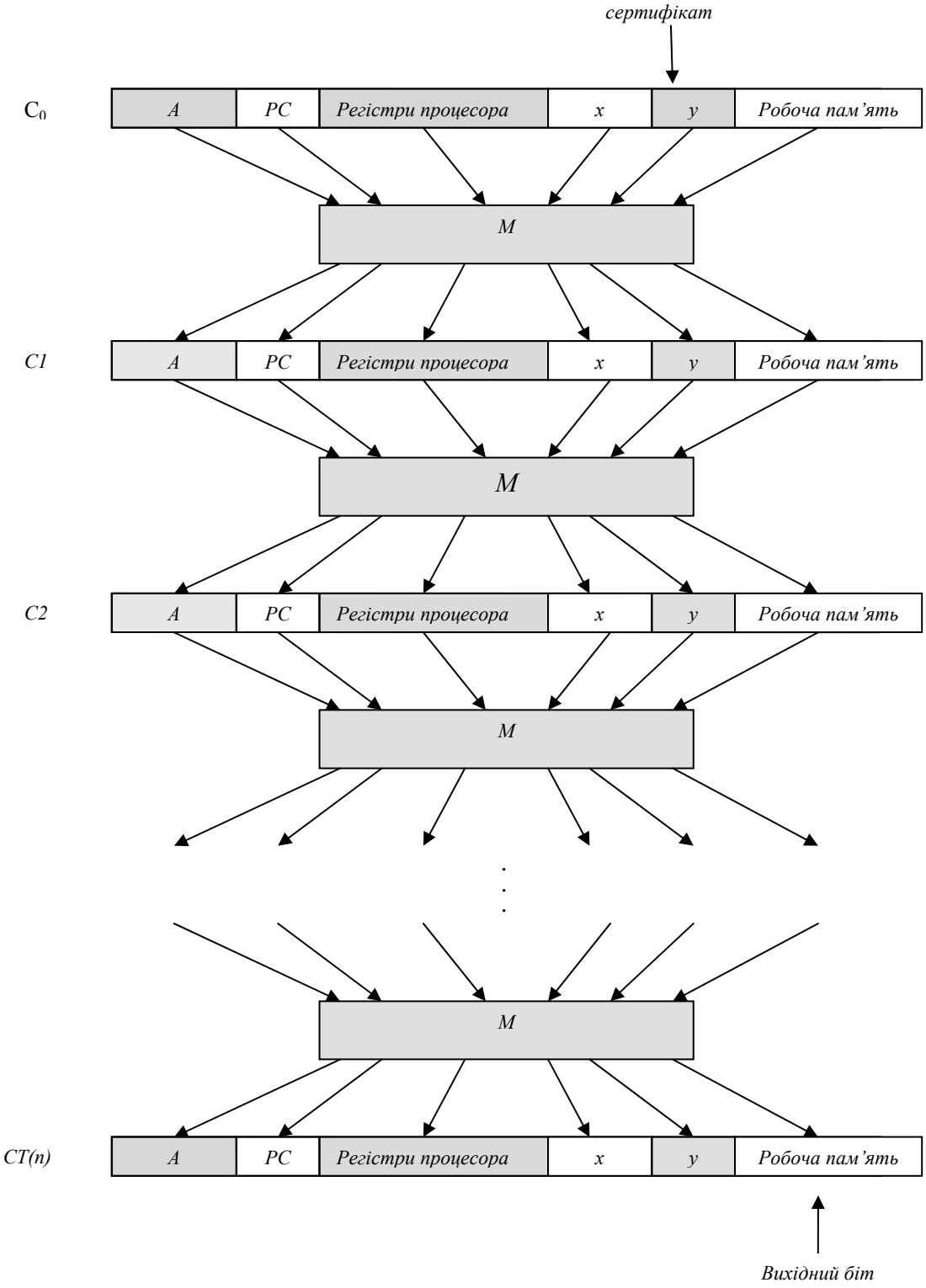


Рисунок 6.7 – Послідовність конфігурацій, що відповідає роботі алгоритму *A* для входу *x* і сертифіката *y*

Отже, ми довели, що мова CIRCUIT-SAT лежить у класі NP і що будь-яка мова з NP до нього зводиться, тобто що задача CIRCUIT-SAT є NP-повною.

Теорема 6.3. *Задача про виконуваність схеми NP – повна.*

6.4 Приклади NP-повних задач

На сьогоднішній день відомо багато NP-повних задач, пов'язаних із різними областями математики й інформатики: логікою, теорією графів, комп'ютерними мережами, множинами і розбивками, розкладаннями, математичним програмуванням, алгеброю і теорією чисел, іграми і головоломками, оптимізацією програм і т.д.

У цьому розділі ми доведемо NP-повноту декількох задач про графи і множини за допомогою метода поліноміального зведення.

Зв'язок між цими задачами показано на рис. 6.8; стрілки означають звідність за поліноміальний час (CIRCUIT-SAT зводиться до SAT і т.д.). Довівши (у теоремі 6.3) NP-повноту задачі CIRCUIT-SAT, за допомогою цієї звідності ми переконуємося в повноті всіх перерахованих на рисунку задач.

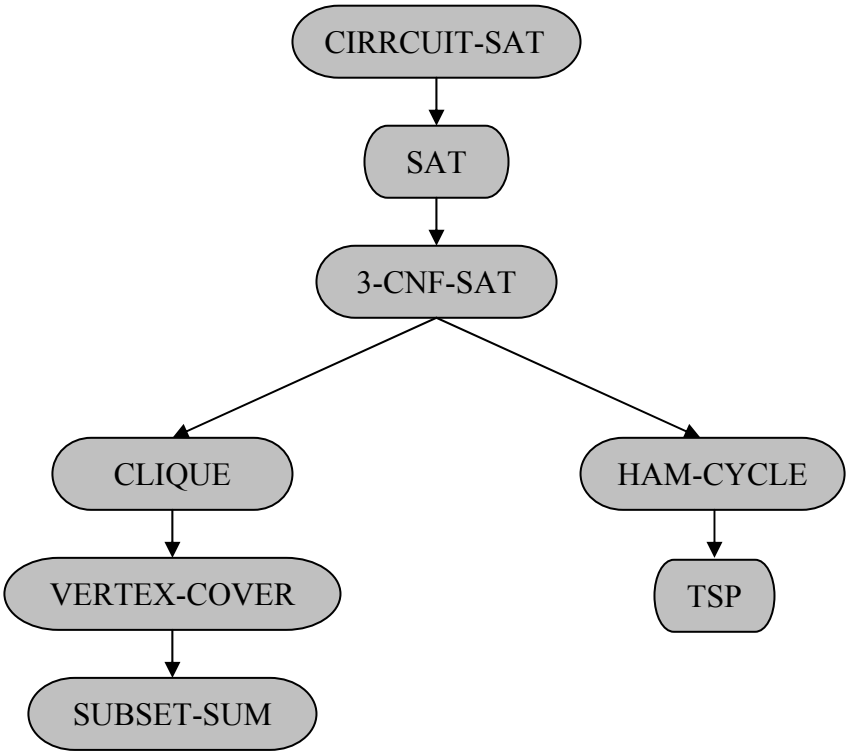


Рисунок 6.8 – Звідність, яка використовується при доведеннях NP-повноти

Задача про кліку

Клікою (clique) у неорієнтовному графі $G = (V, E)$ називається підмножина вершин $V \subset V$, кожні дві з яких з'єднані ребром графу. Іншими словами, кліка – це повний підграф графу G. **Розміром** (size) кліки називається число вершин, що містяться в ній. Розглянемо оптимізаційну за-

дачу: визначити максимальний розмір кліки в даному графі. Її називають **задачею про кліку** (clique problem). Відповідна задача дозволу формулюється так: дані граф *G* і число *k;* потрібно встановити чи є в графі *G* клі-ка розміром *k.* Формально кажучи,

$$CLIQUE=\{\langle G,k\rangle : \text{в графі } G \text{ є кліка розміром } k\}.$$

Як завжди, ми можемо перебрати всі підмножини розміру *k* в *V* і перевірити чи є серед них кліка. Для цього потрібно $\Omega\left(k^2 \cdot C_V^k\right)$ дій *(V* – число вершин у графі). При будь-якому фіксованому *k* ця величина поліноміально залежить від розміру графу *G.* Однак у загальній постановці задачі *k* може бути будь-яким числом, що не перевищує |*V*|, і алгоритм не є поліноміальним.

Поліноміального алгоритму швидше за все немає, оскільки має місце така теорема.

Теорема 6.4. *Задача CLIQUE є NP-повною.*

Доведення. Спочатку переконаємося, що CLIQUE ∈ NP. Насправді як сертифікат можна взяти список усіх вершин, що утворять кліку (маючи цей список, наявність усіх з'єднувальних ребер можна перевірити за поліноміальний час).

Для доведення NP-труднощів задачі CLIQUE покажемо, що *3-CNF-SAT* $\leq_p$ CLIQUE. На перший погляд це здається дивним – пропозиційні формули ніяк не зв'язані з графами і кліками – але насправді зведення побудувати легко.

Звідний алгоритм одержує формулу з класу 3-CNF, виконуваність якої потрібно перевірити (тобто звести до задачі про кліку). Нехай дана формула

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k,$$

де кожна підформула C_r є диз'юнкція трьох літералів $l_1^r, l_2^r mal\ {}_3^r$. Побудуємо граф *G = (V,E),* що містить кліку розміру *k,* якщо і тільки якщо формула φ здійсненна.

Для кожної диз'юнкції $C_r = \left(l_1^r \vee l_2^r \vee l_3^r\right)$ з формули φ нарисуємо три вершини $\upsilon_1^r, \upsilon_2^r, \upsilon_3^r$. Таким чином, граф *G* буде містити 3*k* вершин. (Забігаючи вперед, можна сказати, що майбутня кліка буде утворена дійсними членами диз'юнкцій.) Поки що опишемо ребра графу: дві вершини υ_i^r і υ_j^s з'єднані ребром у графі *G,* якщо виконані такі умови:

– υ_i^r і υ_j^s належать різним трійкам ($r \neq s$);

– літерали l_i^r і l_j^s, що відповідають даним вершинам, сумісні (are consistent), тобто не є запереченнями один одного.

Граф G легко побудувати за формулою φ за поліноміальний час. На рис. 6.9 показаний граф, що відповідає формулі

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3).$$

Покажемо, що описане перетворення дійсно є зведенням. Спочатку припустимо, що формула φ має виконуючий набір. Тоді кожна диз'юнкція d містить хоча б один дійсний літерал; виберемо один з таких (для кожної диз'юнкції). Відзначимо відповідні обраним літералам вершини графу G. Ми затверджуємо, що k відзначених вершин утворять кліку. Дійсно, два відзначених літерала спільні, тому що обидва дійсні на тому самому виконуваному наборі (і тому не можуть бути запереченнями один одного).

Навпаки, нехай у графі G є кліка V розміру k. У кожній трійці вершини не з'єднані ребрами одна з одною, тому кліка V' містить рівно по одній вершині з кожної трійки. Розглянемо відповідні літерали й оголосимо їх дійсними. Спільність літералів гарантує, що для цього не прийдеться повідомляти змінну одночасно дійсною і помилковою. Якщо після цього значення деяких змінних ще не визначені, виберемо їх невимушено. Одержимо набір значень змінних, який буде виконуючим, тому що в кожній з диз'юнкцій є хоча б один дійсний член.

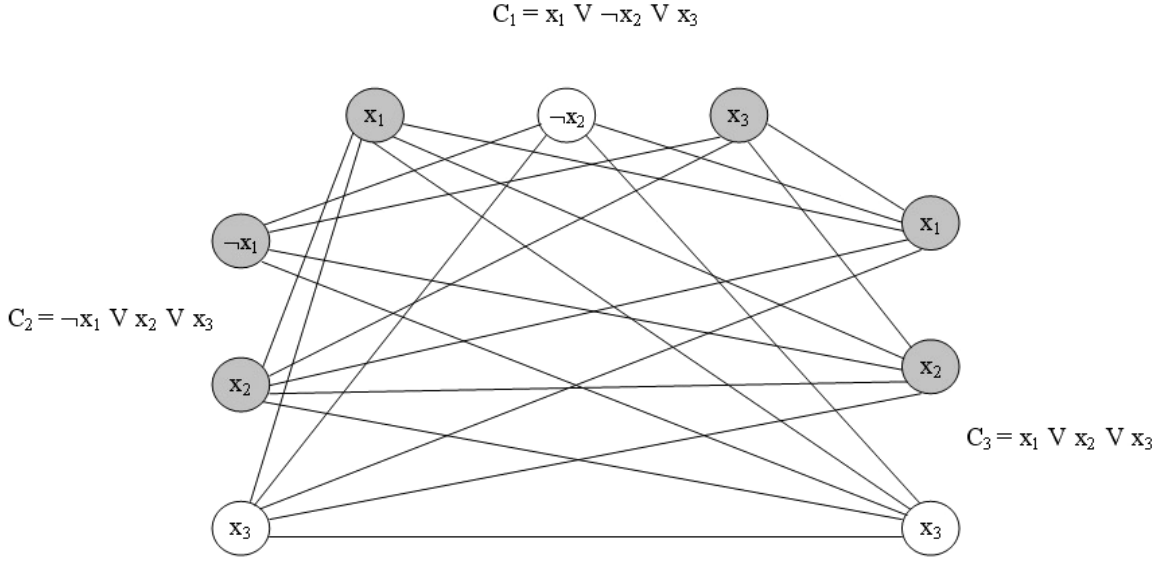


Рисунок 6.9 – Граф, що відповідає формулі $\varphi = C_1 \wedge C_2 \wedge C_3$, що виконує набір $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$ виконує C_1 за рахунок $\neg x_2$, а C_2 і C_3 – за рахунок x_3

Відповідні вершини показані світло-сірим кольором й утворюють кліку.

Задача про вершинне покриття

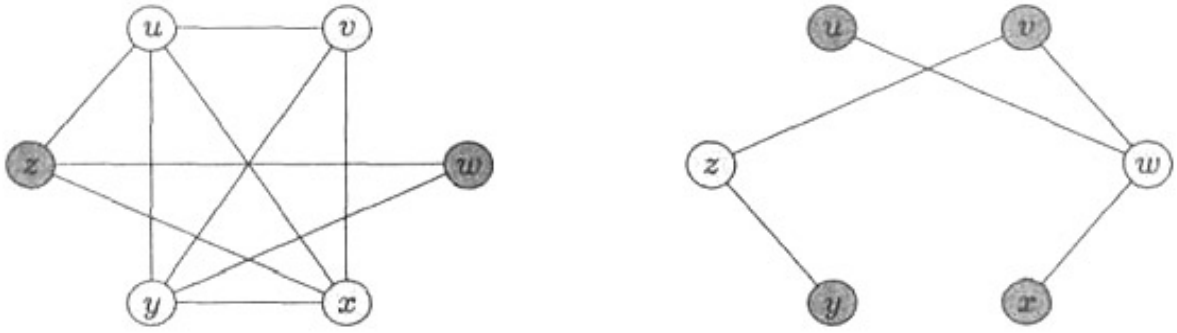
Множина вершин $V' \subseteq V$ графу $G = (V, E)$ називається **вершинним покриттям** (vertex cover) графу, якщо в будь-якого ребра графу хоча б один з кінців входить у V'. Якщо вважати, що вершина «покриває» інцидентні їй ребра, то вершинне покриття графу G – це множина вершин, що покривають всі його ребра. **Розміром** (size) вершинного покриття називається число вхідних у нього вершин. Наприклад, граф рис. 6.10, б має вершинне покриття $\{w, z\}$ (кількість вершин рівна 2).

Задача про вершинне покриття (vertex-cover problem) вимагає вказати мінімально можливий розмір вершинного покриття для заданого графу. Як звичайно, ми перейдемо від задачі оптимізації до задачі дозволу і будемо запитувати чи має даний граф вершинне покриття даного розміру. Відповідна мова така:

VERTEX-COVER = $\{\langle G, k\rangle$ *:* граф G має вершинне покриття розміру $k\}$.

Теорема 6.5. *Задача VERTEX-COVER є NP-повною.*

Доведення. Спочатку переконаємося, що дана задача належить класові NP. Справді, в якості сертифіката підходить саме вершинне покриття.



а) б)

Рисунок 6.10 – Зведення задачі CLIQUE до задачі VERTEX-COVER.
а – неорієнтовний граф $G = (V, E)$ і кліка $V' = \{u, v, x, y\}$;
б – його доповнення $\overline{G}$, побудоване звідним алгоритмом, має вершинне покриття $V \setminus V' = \{w, z\}$

Щоб довести, що задача VERTEX-COVER є NP-важкою, зведемо до неї задачу про кліку. У доведенні використовується поняття доповнення графу. Нехай даний неорієнтовний граф $G = (V, E)$. Його доповненням (complement) назвемо граф $G = (V, E)$, де $\overline{E} = \{(u, \upsilon) : (u, \upsilon) \notin E\}$. Іншими

словами, граф $\overline{G}$ має ті ж вершини, що і граф *G,* а ребра з'єднують ті пари вершин, що не були з'єднані ребром у графі *G.* На рис. 6.10 показані приклад графу і його доповнення, що з'являються при зведенні задачі CLIQUE до задачі VERTEX-COVER.

Звідний алгоритм, повинен довідатися чи є в даному графі $\overline{G}$ кліка заданого розміру *k.* Алгоритм будує граф *G* (доповнення графу *G)* і дає на виході пару $\langle \overline{G}, |V| - k \rangle$. Залишається помітити, що граф $\overline{G}$ має вершинне покриття розміру $|V| - k$ тоді і тільки тоді, коли граф *G* має кліку розміру *k.*

Справді, якщо є кліка розміром *k,* то її доповнення утворить вершинне покриття графу $\overline{G}$ *, що* має розмір $|V|$ - *k* (будь-яке ребро графу $\overline{G}$ відсутнє у графі *G,* тому один з кінців цього ребра повинен бути поза клікою).

Навпаки, доповнення до вершинного покриття графу $\overline{G}$ є клікою в *G:* якщо якісь дві вершини цього доповнення не зв'язані ребром у графі *G,* то вони зв'язані ребром у $\overline{G}$, і це ребро не покрите.

Оскільки задача VERTEX-CLIQUE є NP-повною, навряд чи для неї існує ефективний алгоритм. Однак існує ефективний алгоритм, що дає «наближене» розв'язання цієї задачі – можна знайти вершинне покриття, у якому число вершин не більш ніж удвічі перевищує мінімально можливе.

Таким чином, NP-повнота задачі ще не означає, що треба відмовитися від ідеї її вирішити – можливо, що є поліноміальний алгоритм, що дає розв'язання, близьке до оптимального.

Задача про суми підмножин

Розглянемо ще одну NP-повну задачу, цього разу – арифметичну. Нехай дані скінченна множина натуральних чисел $S \subset \mathrm{N}$ і число $t \in \mathrm{N}$. У **задачі про суми підмножин** (subset-sum problem) потрібно з'ясувати, чи існує така підмножина $S' \subseteq S$, сума елементів якого дорівнює *t.* Наприклад, якщо *S*={1,4,16,64,256,1040,1093,1284,1344} і *t*=3754, то відповідь буде позитивною – можна взяти *S′* = {1,16,64,256,1040,1093,1284}.

Відповідна мова така:

SUBSET-SUM = $\{\langle S, t \rangle$:існує така підмножина $S' \subseteq S$,

що $t = \sum_{s \in S'} s \}$.

Нагадаємо, що ми записуємо числа в двійковій системі (подаючи вхід задачі у вигляді бітового рядка).

Теорема 6.6. *Задача SUBSET-SUM є NP-повною.*

Доведення. Очевидно, ця задача належить класові NP (сертифікатом можна вважати саму підмножину *S′).*

Тепер покажемо, що VERTEX-COVER≤pSUBSET-SUM. Звідний алгоритм перетворить вхід $\langle G,k\rangle$ задачі про вершинне покриття в парі $\langle S,t\rangle$ з такою властивістю: у графі *G* існує вершинне покриття розміру *k* тоді і тільки тоді, коли в *S* найдеться підмножина із сумою *t.*

Будемо використовувати подання графу *G* матрицею інцидентності. Нехай $G=(V,E)$– неорієнтовний граф; ми вважаємо, що його вершинами є числа 0,1, 2,...,$|V|-1$, а ребрами – числа 0,1,2,...,$|E|-1$. Тоді **матрицею інцидентності** (incidence matrix) графу *G* буде $|V|\times|E|$– матриця *B,* обумовлена так:

$$b_{ij}=\begin{cases} 1, \text{ якщо ребро } j \text{ інцидентне вершині } i, \\ 0 \text{ в протилежному випадку.} \end{cases}$$

Наприклад, матриця інцидентності рис. 6.11, б відповідає графові рис. 6.11, а.

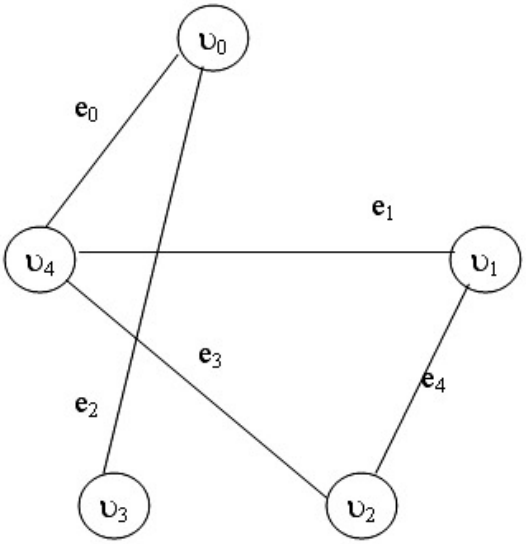
Звідний алгоритм, одержує на вхід матрицю інцидентності *B* і число *k.* Потрібно побудувати множину *S* і число *t.* Усі числа будемо записувати в системі числення за основою 4.

Множина *S* буде складатися з чисел двох типів: одні відповідають вершинам графу, інші – його ребрам. Кожній вершині $i\in V$ ставиться у відповідність число, що записується (у системі з основою 4) так: спочатку йде одиниця, а потім *i*-й рядок матриці інцидентності $B=(b_{ij})$ (рис.6.11, в), тобто рядок, що відповідає цій вершині. Кожному ребру $j\in E$ ставиться у відповідність число y_i, запис якого містить єдину одиницю в *j-й* позиції (тобто число 4^j).

Залишається вказати число *t.* Старші розряди числа *t* збігаються з записом числа *k* з основою 4, а наступні $|E|$розрядів заповнені двійками. Більш точно, $t=k\cdot 4^{|E|}+\sum_{j=0}^{|E|-1}2\cdot 4^j$.

Усі побудовані числа мають двійкове подання поліноміального розміру і будуються за поліноміальний час.

Тепер потрібно перевірити, що граф *G* має вершинне покриття розміру *k* тоді і тільки тоді, коли в *S* існує підмножина *S′* із сумою *t.* Нехай дане вершинне покриття $V'\subseteq V$ необхідного розміру, що містить вершини $i_1,i_2,...,i_k$.



а)

	e_4	e_3	e_2	e_1	e_0
υ_0	0	0	1	0	1
υ_1	1	0	0	1	0
υ_2	1	1	0	0	0
υ_3	0	0	1	0	0
υ_4	0	1	0	1	1

б)

			основа 4						десятковий
			e_4	e_3	e_2	e_1	e_0		запис
x_0	=	1	0	0	1	0	1	=	1041
x_1	=	1	1	0	0	1	0	=	1284
x_2	=	1	1	1	0	0	0	=	1344
x_3	=	1	0	0	1	0	0	=	1040
x_4	=	1	0	1	0	1	1	=	1093
y_0	=	0	0	0	0	0	1	=	1
y_1	=	0	0	0	0	1	0	=	4
y_2	=	0	0	0	1	0	0	=	16
y_3	=	0	0	1	0	0	0	=	64
y_4	=	0	1	0	0	0	0	=	256
t	=	3	2	2	2	2	2	=	3754

в)

Рисунок 6.11 – Зведення задачі про вершинне покриття до задачі про суми підмножин

а – неорієнтовний граф *G*. Світло-сірі вершини утворять вершинне покриття $\{\upsilon_1,\upsilon_3,\upsilon_4\}$ розміром 3;

б – матриця інцидентності цього графу. Світло-сірі рядки відповідають вершинам покриття;

в – відповідний вхід задачі про суми підмножин. Всередині рамки знаходиться матриця інцидентності. Вершинне покриття $\{\upsilon_1,\upsilon_3,\upsilon_4\}$ розміру *k=3* відповідає підмножині з світло-сірих елементів {1,16,64,256,1040,1093, 1284}, сума якого є 3754

Включимо в множину S' числа, що відповідають цим вершинам. Тоді сума елементів множини S', записана з основою 4, буде виглядати так: у старших розрядах стоїть число k, а у всіх молодших розрядах стоять цифри 1 або 2 (у залежності від того, увійшли у вершинне покриття обидва кінці ребра чи тільки один). Узявши ті ребра, у яких тільки один кінець увійшов у вершинне покриття, і додавши відповідні їм числа в множину S', ми перетворимо одиниці в двійки, тобто одержимо в сумі число t (на рис.6.10, в) треба було додати чотири рядки y_0, y_2, y_3, y_4, щоб забезпечити відсутні двійки в молодших розрядах.)

Зворотне міркування аналогічне. Нехай є множина S', сума елементів якої дорівнює t. Це значить, що в молодших розрядах суми стоять двійки. Оскільки рядки, що відповідають ребрам, можуть дати в кожному розряді максимум одиницю, але ніяк не двійку, це значить, що відсутня одиниця приходить від «вершинних» рядків. Отже, вершини, що відповідають вхідним в S' рядкам, утворять вершинне покриття. А старші розряди гарантують, що число вершин у цьому покритті дорівнює k.

6.5 Задача пошуку гамільтонового циклу у графі

Повернемося до задачі про гамільтонів цикл (HAM-CYCLE; про неї згадувалось в підрозділі 6.2).

Теорема 6.7. *Задача HAM-CYCLE є NP-повною.*

Доведення. Ми вже пояснювали, чому ця задача належить класові NP (гамільтонів цикл можна вважати сертифікатом).

Щоб довести NP-повноту задачі HAM-CYCLE, ми покажемо, що *3-CNF-SAT* $\leq_p$ *HAM-CYCLE*. Іншими словами, ми опишемо алгоритм, що перетворить формулу φ із класу 3-CNF з змінними $x_1, x_2, ..., x_n$ у граф $G = (V, E)$, що має гамільтонів цикл у тому і тільки тому випадку, якщо вихідна формула була здійсненною. У нашій конструкції ми будемо використовувати **блоки** – частини графу, що мають деякі спеціальні властивості.

Перший з використовуваних блоків *(A)* показаний на рис. 6.12, а. Припустимо, що A входить у деякий граф G, причому лише вершини a, a', b, b' можуть бути з'єднані з іншими вершинами графу. Тоді гамільтонів цикл у графі G (якщо такий існує) повинен проходити через вершини $z_1 - z_4$, і це може відбуватися лише двома способами (рис. 6.12, б і в). Тому можна символічно зображувати блок A як на рис. 6.12, г, маючи на увазі, що він замінює два ребра (a, a') і (b, b') з такою додатковою умовою: гамільтонів цикл повинен містити рівно одне з двох цих ребер.

Другий використовуваний блок *(B)* показаний на рис. 6.13. Припустимо, що блок *B* входить у деякий граф *G,* причому *B* може бути зв'язаний з іншою частиною графу тільки через вершини b_1, b_2, b_3, b_4. Можна перевірити, що гамільтонів цикл графу *G* (якщо він існує) не може проходити одночасно через три ребра $(b_1, b_2), (b_2, b_3)$ і (b_3, b_4), оскільки тоді цикл не зміг би пройти через усі вершини *B.* Однак гамільтонів цикл може проходити через будь-яку власну підмножину цієї трійки ребер, як видно з рис.6.13, а-д (ще два симетричних варіанти вийдуть, якщо перевернути (б) і (д)). Блок *B* будемо символічно зображувати як на рис.6.13, е (стрілки на рисунку означають, що хоча б один із трьох шляхів, на які вони вказують, повинен увійти у гамільтонів цикл).

Тепер у нас усе готово для побудови графу *G,* що відповідає формулі з класу 3-CNF. Цей граф *G* буде складатися з декількох блоків типу *A* і *B* (рис. 6.14).

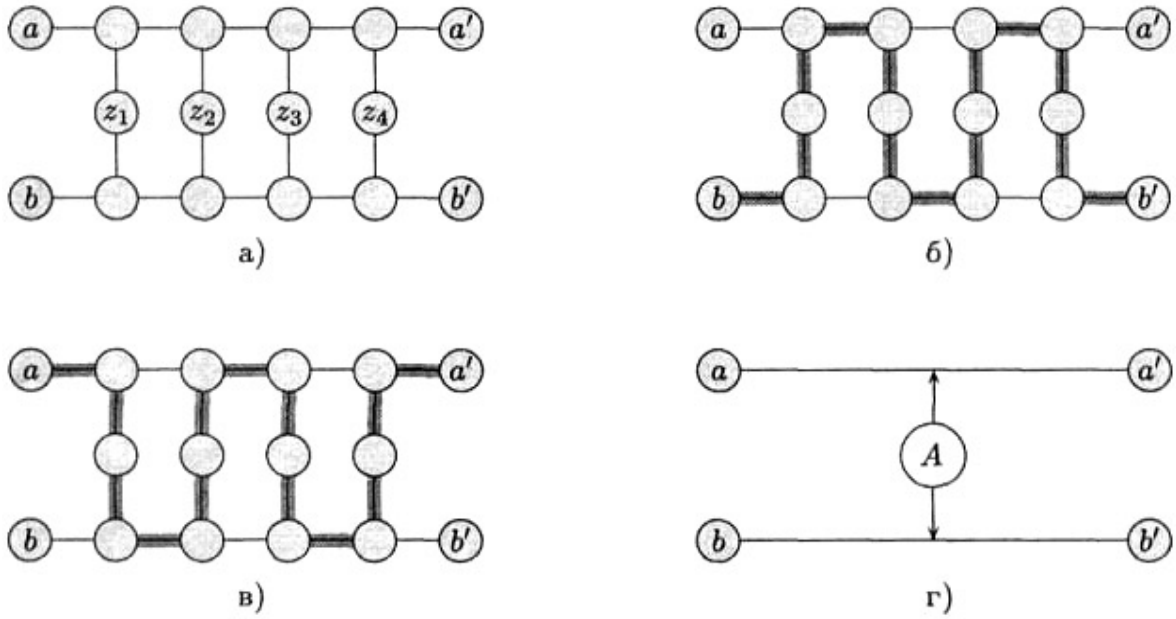


Рисунок 6.12 – Блоки графу

а – блок *A,* використовуваний при зведенні задачі 3-SAT до задачі HAM-CYCLE;

б, в – якщо блок *A* входить у граф *G* і з'єднується з ним лише в кутових вершинах, то будь-який гамільтонів цикл у графі *G* проходить через *A* одним із двох способів;

г – символічне зображення блоку *A*

Припустимо, що формула φ складена з *k* диз'юнкцій $C_1, C_2, ..., C_k$, кожна з яких містить рівно 3 літерали. Для кожної диз'юнкції C_i виготовимо блок типу *B;* позначимо через $b_{i,j}$ відповідні копії вершин b_j. З'єднаємо вершини $b_{i,4}$ і $b_{i+1,1}$ для $i = 1,2,...,k-1$.

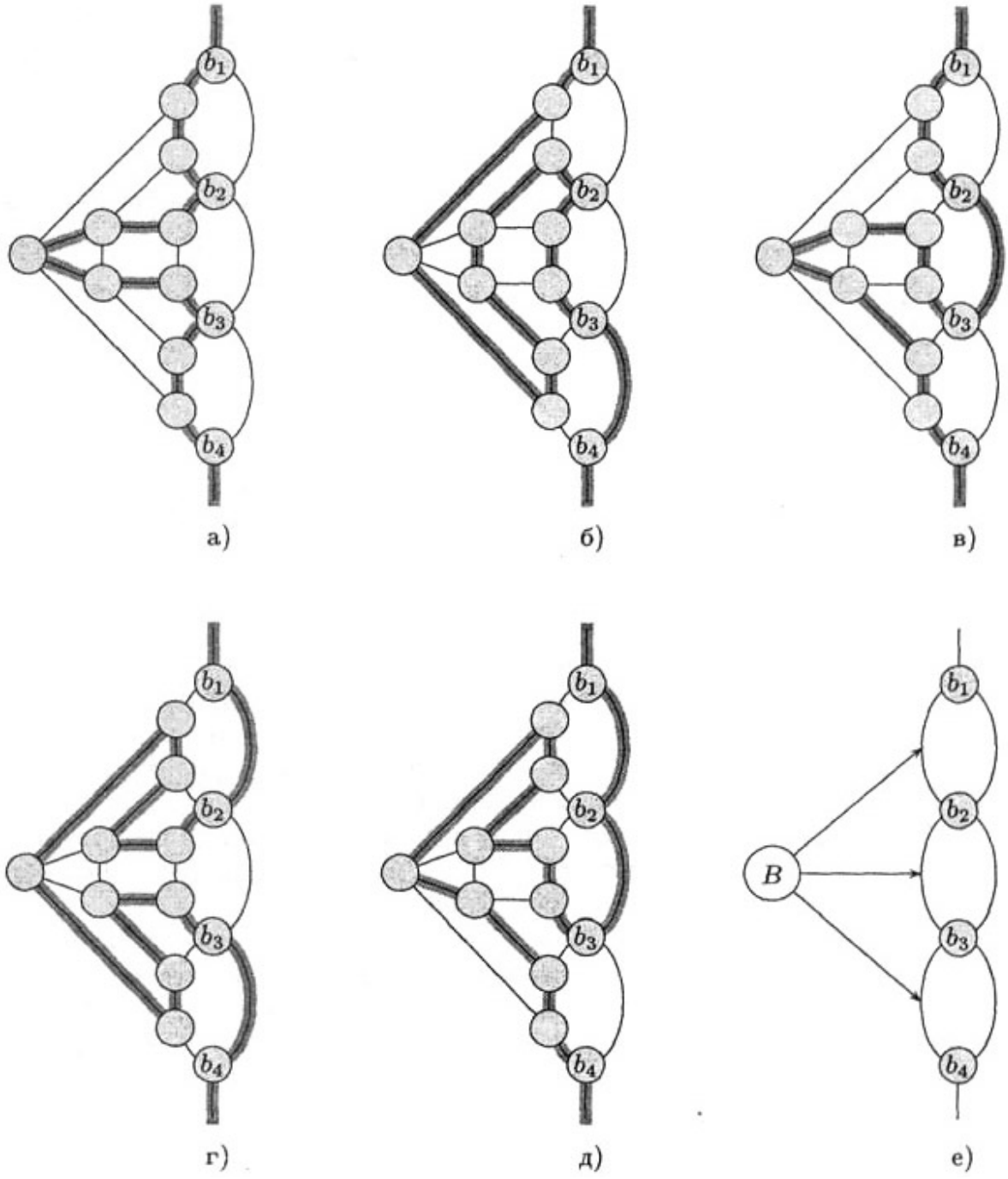


Рисунок 6.13 – Блок *B,* використовуваний при зведенні задачі 3-CNF-SAT до задачі HAM-CYCLE.

Шлях з вершини b_1 в b_4 не може проходити по всіх трьох ребрах, $(b_1, b_2), (b_2, b_3), (b_3,b_4)$, але може проходити по будь-якій власній підмножині цієї множини, як показано в (а-д). Символічне зображення (е) підкреслює цю властивість: принаймні один із трьох шляхів, на які вказують стрілки, повинен увійти в гамільтоновий шлях

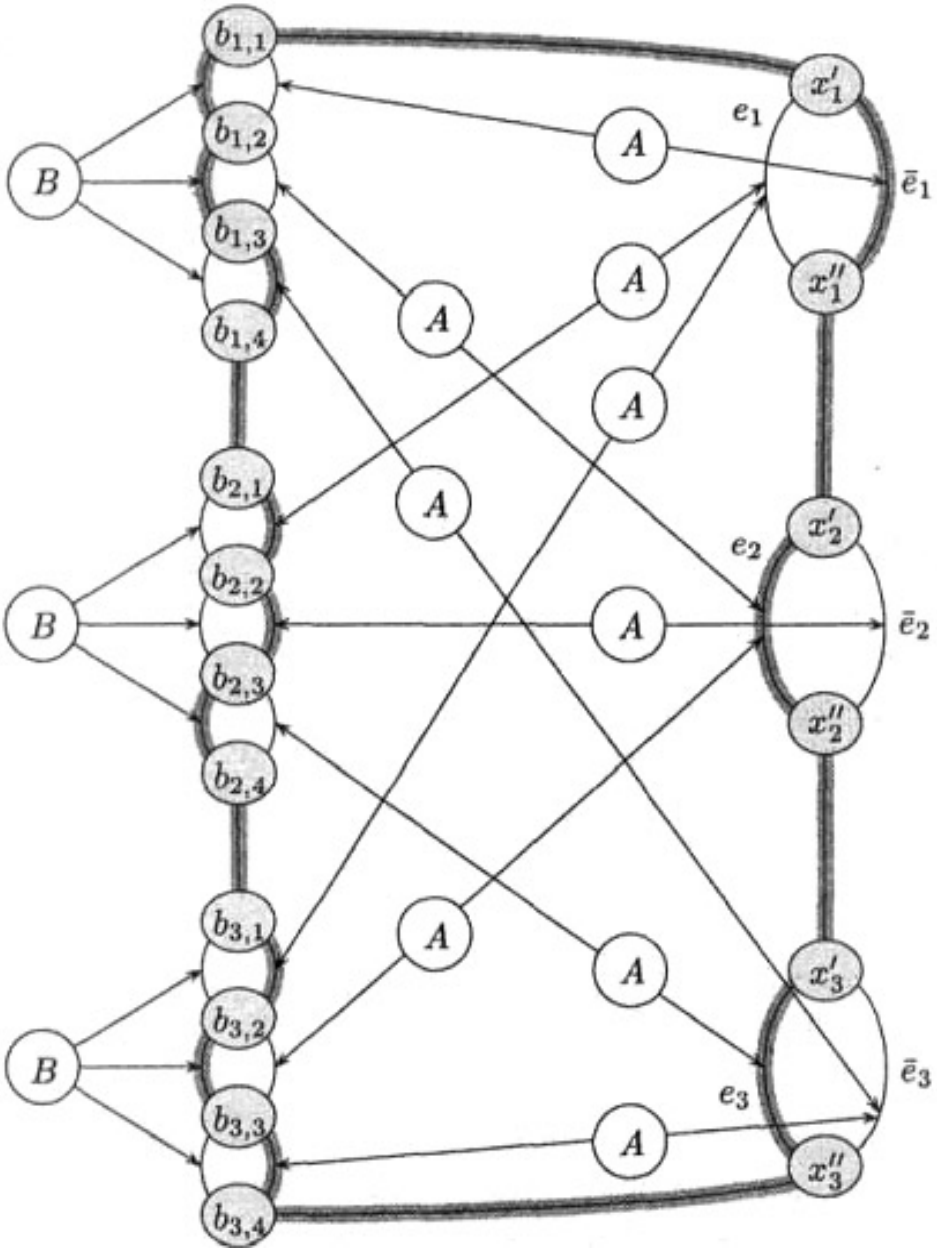


Рисунок 6.14 – Граф G побудований за формулою

$$\varphi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3).$$

Для цієї формули є виконуючий набір $x_1=0$, $x_2=1$. Відповідний гамільтоновий цикл показано сірим. Відзначимо, що якщо в наборі $x_m=1$, то в цикл входить ребро e_m, а якщо $x_m = 0$, то $\bar{e}_m$

Далі, для кожної змінної x_m формули φ ми зіставимо пари вершин x'_m, x''_m. Ці вершини ми з'єднаємо двома ребрами e_m і e'_m. (Насправді, як ми побачимо, це будуть не ребра, а більш складні конструкції, так що кратних ребер не буде). Ідея тут у тім, що гамільтонів цикл буде проходити

через ребро e_m, якщо у виконуючому наборі змінна x_m приймає значення 1, і через $\bar{e}_m$, якщо ця змінна приймає значення 0.

Щоб дати гамільтоновому циклу можливість пройти по e_m або $\bar{e}_m$, додамо в граф ребра (x''_m, x'_{m+1}) для m = 1,2,...,*n-1*, а також ще два ребра: $(b_{1,1}, x'_1)$ і $(b_{4,4}, x''_n)$ (верхнє і нижнє ребра, рис. 6.14).

Побудову графу ще не закінчено; ми повинні зв'язати блоки графу, що відповідають змінним формули, із блоками, що відповідають її диз'юнкціям. Якщо *j*-й літерал диз'юнкції C_i є x_m, з'єднаємо ребро $(b_{i,j}, b_{i,j+1})$ з ребром e_m за допомогою *A*-блоку. Якщо ж *j*-м літералом диз'юнкції C_i є $\neg x_m$, ми з'єднаємо за допомогою *A*-блоку ребра $(b_{i,j}, b_{i,j+1})$ і $\bar{e}_m$. Так, у прикладі рис. 6.14 маємо $C_2 = x_1 \vee \neg x_2 \vee x_3$, тому ми розміщуємо три *A*-блоки між ребрами:

– $(b_{2,1}, b_{2,2})$ і e_1;

– $(b_{2,2}, b_{2,3})$ і $\bar{e}_2$;

– $(b_{2,3}, b_{2,4})$ і e_3.

Відзначимо, що слова «з'єднати два ребра за допомогою *A*» означають, що кожне з них замінюється на ланцюжок з п'яти нових ребер і додаються зв'язуючі їх ребра і вершини, як це передбачено конструкцією *A*-блоку (рис. 6.12).

Один літерал l_m може зустрічатися в декількох диз'юнкціях (наприклад, $\neg x_3$ на рис. 6.14). У цих випадках потрібно «підключити» до відповідного ребра кілька А-блоків; це можна зробити, якщо вхідні в А-блоки ланцюжка з п'яти ребер з'єднати послідовно як показано на рис. 6.15.

Ми стверджуємо, що формула φ здійсненна, якщо і тільки якщо побудований граф *G* має гамільтоновий цикл. Припустимо спочатку, що в графі *G* є гамільтоновий цикл *h,* і доведемо виконання формули φ.

Цикл *h* повинен бути улаштований так:

– спочатку проходимо ребро $(b_{1,1,} x'_1)$ (будемо рахувати, зліва направо);

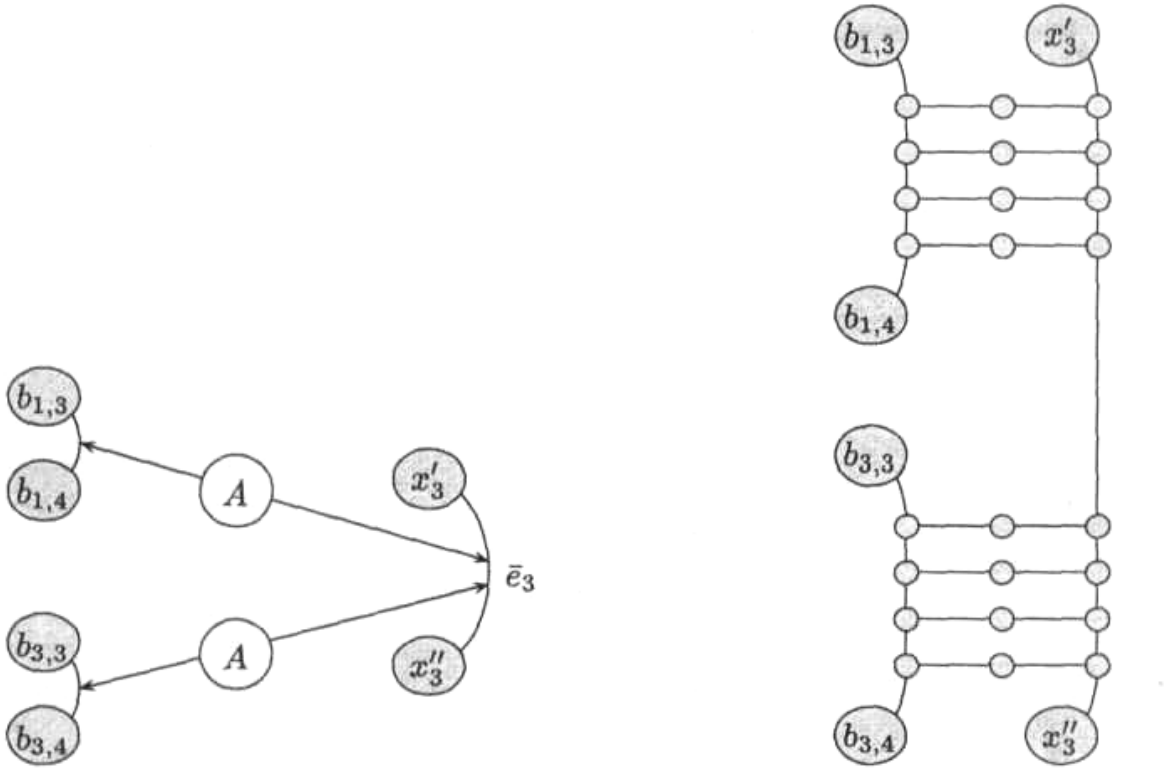
– потім для кожного *m* проходимо по одному (і тільки одному) з ребер e_m, e'_m

– проходимо по ребру $(b_{k,4}, x''_n)$ справа наліво;

– проходимо усі В-блоки знизу вгору.

Відзначимо, що в дійсності ми проходимо не по самих ребрах e_m і $\bar{e}_m$, а по А-блоках, що до них приєднані (якщо такі є). Відзначимо також, що якщо ні до e_m ні до $\bar{e}_m$ не приєднано жодного А-блоку, то

змінна x_m не входить у формулу і її можна взагалі видалити, оскільки кратних ребер дійсно немає.



а) б)

Рисунок 6.15 – Подробиці конструкції для випадку, коли ребро e_m (або $\overline{e_m}$) входить у кілька А-блоків
а – фрагмент рис. 6.14; б – розгорнутий варіант цього фрагмента

Тепер можна указати виконуючий набір для формули φ якщо ребро e_m належить гамільтоновому циклу h, покладемо $x_m = 1$. У протилежному випадку цикл h містить ребро $\bar{e}_m$, і ми думаємо $x_m = 0$.

Доведемо, що побудований набір є виконуючим для формули φ. Розглянемо будь-яку диз'юнкцію C_i і відповідний їй В-блок. Кожне ребро $(b_{i,j,}b_{j+1})$ зв'язано за допомогою А-блоку або з ребром e_m, або з ребром $\bar{e}_m$ (у залежності від того, чи є *j-м* літералом у диз'юнкції C_i змінна x_m або її заперечення $\neg x_m$). Цикл h проходить через ребро $(b_{i,j,}b_{j+1})$, якщо і тільки якщо відповідний литерал дорівнює 0. Згадаємо, що h не може проходити через усі три ребра $(b_{i,1,}b_{i,2}),(b_{i,2},b_{i,3}),(b_{i,3,}b_{i,4})$ у В-блоці, тому хоча б одному з цих ребер відповідає справжній літерал диз'юнкції і диз'юнкція C_i істинна. Це можна сказати про будь-яку диз'юнкцію C_i, так що усі

вони істинні і формула φ істинна для побудованого набору значень змінних.

Навпаки, нехай φ істинна для деякого набору значень змінних. Побудуємо цикл h за описаними вище правилами: цикл містить ребро e_m при $x_m = 1$ і ребро $\bar{e}_m$ при $x_m = 0$; цикл проходить через ребро $(b_{i,j}, b_{j+1})$, якщо і тільки якщо *j-й* літерал диз'юнкції C_i дорівнює 0 на даному наборі. Очевидно, описані правила дозволяють побудувати гамільтоновий цикл за виконуючим набором.

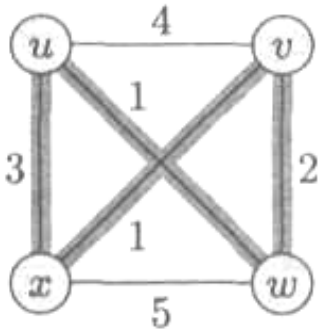


Рисунок 6.16 – Задача комівояжера

У цьому прикладі оптимальний маршрут проходить по сірих ребрах і має вартість 7.

Відзначимо, нарешті, що граф G має поліноміальний розмір (тобто його розмір обмежений поліномом від розміру формули φ) і що побудова графу G за формулою φ проводиться за поліноміальний час. Таким чином, задача 3-CNF-SAT зводиться за поліноміальний час до задачі HAM-CYCLE, що і потрібно було довести.

Задача комівояжера

З задачею про гамільтоновий цикл тісно зв'язана **задача комівояжера** (traveling-salesman problem, TSP). У цій задачі потрібно знайти оптимальний маршрут відвідування n міст.

Комівояжер хоче об'їхати всі міста, побувавши в кожному один раз, і повернутися в місто, з якого почата подорож. Відомо, що переліт з міста i у місто j коштує $c(i,j)$ карбованців (вважаємо ціни цілими числами). У термінах теорії графів задачу можна сформулювати так: потрібно знайти в даному графі гамільтоновий цикл найменшої вартості (вартість циклу є сума вартостей усіх його ребер). Відповідна задача дозволу є мовою $TSP = (G, c, k) : G = (V,E)$ – повний граф, с: $V \times V \to \to Z$ – функція вартості, $k \in Z$ в G є гамільтоновий цикл вартості не більш $k\}$.

Теорема 6.8. *Задача комівояжера (мова TSP) є NP-повною.*

Доведення. Очевидно, мова TSP належить класові NP (як сертифікат можна взяти гамільтоновий цикл вартості не вище k*)*.

Щоб переконатися, що задача комівояжера є NP-важкою, зведемо до неї задачу HAM-CYCLE. Щоб довідатися, чи є в графі G гамільтоновий цикл, побудуємо повний граф G' з тими ж вершинами; ребра з G будуть мати ціну 0, а всі інші ребра – ціну 1. Очевидно, що в графі G' існує маршрут комівояжера вартістю 0 у тому і тільки в тому випадку, коли граф G має гамільтонів цикл.

6.6 Контрольні питання

1. Поняття NP-повноти.
2. Поняття поліноміального часу.
3. Задачі та сфери застосування NP-повноти.
4. Особливості термінології теорії формальних мов.
5. Поняття гамільтонового циклу.
6. NP-повні задачі і зведення.
7. Задача про виконуваність схеми.
8. Задача про кліку.
9. Задача про вершинне покриття.
10. Задача про сумму підмножин.
11. Задача про гамільтонів цикл.
12. Задача комівояжера.

6.7 Завдання

1. Визначити формальну задачу знаходження найбільш простого циклу в неорієнтовному графі. Сформулюйте відповідну строкову задачу розв’язування (мова).

2. Розгляньте два подання для графу – за допомогою матриці інцидентності (закодованою послідовністю бітів) і за допомогою списків ребер (також належним чином закодованих). Поясніть, чому два дані подання поліноміально зв'язані.

3. Покажіть, що алгоритм, який містить фіксоване (не залежне від входу) число викликів процедури, що працює за поліноміальний час. Як-

що ж алгоритм робить поліноміальне число викликів такої процедури, то загальний час роботи може бути експоненціальним. Чому?

4. Розглянемо мову *GRAPH-ISOMORPHISM = {(G_1,G_2) :* графи G_1 і G_2 ізоморфні}. Доведіть, що дана мова належить класу NP (побудуйте поліноміальний алгоритм, що перевіряє цю мову).

5. Доведіть, що неорієнтовний двочастковий граф з непарним числом вершин не гамільтонів.

6. Покажіть, що для орієнтовних графів без циклів можна за поліноміальний час з'ясувати чи існує гамільтонів шлях.

7. У доведенні леми 6.4 ми припускали, що комп'ютер використає ділянку пам'яті поліноміального розміру (а не поліноміальне число осередків у різних місцях пам'яті експонентного розміру). Чому це суттєво? Як обійтися без цього припущення?

8. Доведіть NP-повноту задачі про гамільтоновий шлях.

9. Задача про найдовший простий цикл (longest-simple-cycle problem) полягає у знаходженні в даному графі простого (без повторюваних вершин) циклу найбільшої довжини. Сформулюйте відповідну задачу дозволу і доведіть її NP-повноту.

ЛІТЕРАТУРА

1. Т. Кармен, Ч. Лейзерсон, Р. Ривест Алгоритмы: построение и анализ. – М.: МЦНМО, 2000. – 960 с.
2. А. Ахо, Д. Хопкрофт, Д. Ульман. Структуры данных и алгоритмы: Пер. с англ.: Учебное пособие – М.: Издательский дом “Вильямс”, 2000. – 384 с.
3. Д. Э. Кнут. Сортировка и поиск. 2-е изд: Пер. с англ. Учебное пособие – М.: изд. дом “Вильямс”, 2000. – 832 с.
4. Н. Вирт. Алгоритмы и структуры данных: Пер с англ. – М.: Мир, 1989. – 360 с.
5. А. Ахо, Д. Хопкрофт, Д. Ульман. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – 536 с.
6. О.П. Кузнєцов, Г.М. Адельсон-Вельский. Дискретная математика для инженера. 2-е изд. – М.: Энергоатомиздат. – 1988. – 480 с.
7. Н.А. Криницкий. Алгоритмы вокруг нас. – М.: Наука, 1984. – 224 с.
8. А.А. Марков, Н.М. Нагорный. Теория алгоритмов. – М.: Наука, 1984. – 432 с.
9. І.Ф. Следзінський, А.М. Ломакович, Ю.С. Рамський, Р.І. Зароський. Техніка обчислень і алгоритмізація. – К.: Вища шк., 1991. –199 с.
10. В.М. Бондарев, В.И. Рублинецкий, Е.Г. Качко. Основы программирования. – Харьков, 1997. – 368 с.

Навчальне видання

Ігор Ростиславович Арсенюк
Володимир Володимирович Колодний
Андрій Анатолійович Яровий

ТЕОРІЯ АЛГОРИТМІВ

Навчальний посібник

Оригінал-макет підготовлено І. Р. Арсенюком

Редактор В. О. Дружиніна
Коректор З. В. Поліщук

Науково-методичний відділ ВНТУ
Свідоцтво Держкомінформу України
серія ДК № 746 від 15.12.2001
21021, м. Вінниця, Хмельницьке шосе, 95, ВНТУ

Підписано до друку
Формат 29,7х42 1/4
Друк різографічний
Тираж ____ прим.
Зам. №

Гарнітура Times New Roman
Папір офсетний
Ум. друк. арк.

Віддруковано в комп'ютерному інформаційно-видавничому центрі
Вінницького національного технічного університету
Свідоцтво Держкомінформу України
серія ДК № 746 від 15.12.2001