В. В. ЛАГОВСЬКИЙ, А. А. ОМЕЛЬЧУК,
Л. В. СКАСКІВ

# ТЕОРІЯ АЛГОРИТМІВ ТА СТРУКТУРА ДАНИХ

**Навчальний посібник**

**ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ**

**В. В. Лаговський, А. А. Омельчук, Л. В. Скасків**

# ТЕОРІЯ АЛГОРИТМІВ ТА СТРУКТУРА ДАНИХ

*Навчальний посібник*

Ірпінь
2025

**УДК 004.021(075.8)**
**ББК 32.973.26-018.273**
**Л 14**

*Рекомендовано до друку Вченою радою*
*Державного податкового університету*
*(протокол № 5 від 27 листопада 2025 року)*

***Рецензенти:***

**В. М. Франчук,** доктор педагогічних наук, доцент, завідувач кафедри комп'ютерної та програмної інженерії УДУ ім. М. Драгоманова;

**А. Ю. Горбовий,** доктор технічних наук, професор, професор кафедри комп'ютерних та інформаційних технологій і систем.

**Автори:**

**В. В. Лаговський,** кандидат економічних наук, доцент, доцент кафедри комп'ютерних та інформаційних технологій і систем;

**А. А. Омельчук,** кандидат технічних наук, доцент кафедри комп'ютерних та інформаційних технологій і систем;

**Л. В. Скасків,** кандидат фізико-математичних наук, доцент, доцент кафедри вищої та прикладної математики.

**Лаговський В. В.**

**Л 14** **Теорія алгоритмів та структура даних** : навчальний посібник / В. В. Лаговський, А. А. Омельчук, Л. В. Скасків. – Ірпінь : Державний податковий університет, 2025. – 472 с.
ISBN 978-966-337-789-6

Навчальний посібник присвячено фундаментальним теоретичним і практичним засадам розробки та аналізу алгоритмів, а також організації структур даних. Особливістю видання є поєднання чіткого математичного обґрунтування із прикладною спрямованістю. Зміст охоплює шістнадцять логічно структурованих тем, що відображають еволюцію матеріалу від базових концепцій до складних алгоритмічних парадигм, теорії графів, просторових структур, потокових алгоритмів обробки значних обсягів інформації та теорії складності обчислень.

Посібник розрахований на здобувачів вищої освіти й науково-педагогічних працівників закладів вищої освіти.

**УДК 004.021(075.8)**
**ББК 32.973.26-018.273**

© Лаговський В. В., Омельчук А. А.,
Скасків Л. В., 2025

**ISBN 978-966-337-789-6**

© Державний податковий університет, 2025

# ЗМІСТ

**ПЕРЕДМОВА**.....10

**Тема 1. Основи аналізу алгоритмів**.....14
1.1. Поняття та властивості алгоритму.....15
1.2. Моделі обчислень для аналізу.....17
1.3. Оцінка ефективності: часова та просторова складність.....22
1.4. Асимптотичний аналіз.....23
1.5. Аналіз найкращого, середнього та найгіршого випадків.....25
1.6. Структури даних.....27
1.7. Практичний аналіз ітеративних алгоритмів.....28
1.8. Аналіз базових алгоритмів пошуку та сортування.....29
1.9. Приклади розв'язку задач.....34
*Контрольні питання*.....35
*Задачі для самостійного опрацювання*.....36

**Тема 2. Аналіз рекурсивних алгоритмів та математичні основи**.....38
2.1. Формальне визначення рекурсії.....38
2.2. Структура рекурсивної функції: базовий випадок та рекурсивний крок.....39
2.3. Класичний приклад: числа Фібоначчі.....41
2.4. Переваги та недоліки рекурсії.....43
2.5. Поняття рекурентного співвідношення.....44
2.6. Побудова рекурентних співвідношень на прикладах.....45
2.7. Метод підстановки (Substitution Method).....47
2.8. Метод дерева рекурсії (Recursion-Tree Method).....47
2.9. Основна теорема.....48
2.10. Амортизований аналіз.....50
2.11. Імовірнісний аналіз.....51
2.12. Моделі обчислень.....52
2.13. Поняття структури даних та її роль.....54
2.14. Приклади розв'язку задач.....56
*Контрольні питання*.....60
*Задачі для самостійного опрацювання*.....61

**Тема 3. Лінійні структури даних та абстрактні типи** ..... 62
3.1. Поняття абстрактного типу даних ..... 62
3.2. Основні лінійні структури даних: стек, черга, дек ..... 63
3.3. Послідовні структури даних ..... 66
3.4. Порівняльний аналіз статичних та динамічних масивів ..... 83
3.5. Приклади розв'язку задач ..... 84
3.6. Зв'язні списки та інші структури ..... 89
3.7. Оцінка складності операцій ..... 92
3.8. Приклади розв'язку задач ..... 94
*Контрольні питання* ..... 97
*Задачі для самостійного опрацювання* ..... 99

**Тема 4. Структури для словників** ..... 102
4.1. Словники та хеш-таблиці:
фундаментальні поняття ..... 102
4.2. Принципи хешування та хеш-функції ..... 105
4.3. Методи розв'язання колізій ..... 107
4.4. Кейс-стаді: кешування в сучасних вебдодатках ..... 110
4.5. Практична реалізація хеш-таблиці ..... 111
4.6. Аналіз складності операцій ..... 118
4.7. Приклади розв'язку задач ..... 119
*Контрольні питання* ..... 121
*Задачі для самостійного опрацювання* ..... 122

**Тема 5. Деревоподібні структури** ..... 124
5.1. Бінарні дерева та їх обходи ..... 125
5.2. Бінарні дерева пошуку (BST):
операції вставки, пошуку, видалення ..... 132
5.3. Проблема деградації дерева ..... 135
5.4. Збалансовані двійкові дерева пошуку
(AVL, червоно-чорні) ..... 137
5.5. Реалізація бінарного дерева пошуку
та основних операцій ..... 140
5.6. Вирішення проблем, пов'язаних
з незбалансованістю ..... 147
5.7. B-дерева ..... 155

5.8. Консистентне хешування ..... 159
5.9. Огляд B-дерев та їх варіантів ..... 161
5.10. Приклади розв'язку задач ..... 163
*Контрольні питання* ..... 166
*Задачі для самостійного опрацювання* ..... 167

**Тема 6. Купи та черги з пріоритетом** ..... 169
6.1. Бінарні купи та операції над ними ..... 170
6.2. Застосування куп в алгоритмах ..... 179
6.3. Біноміальні купи ..... 183
6.4. Фібоначчієві купи ..... 186
6.5. Лівосторонні купи ..... 188
6.6. Приклади розв'язку задач ..... 190
*Контрольні питання* ..... 193
*Задачі для самостійного опрацювання* ..... 194

**Тема 7. Алгоритми сортування** ..... 196
7.1. Ключові характеристики алгоритмів ..... 196
7.2. Класифікація алгоритмів ..... 197
7.3. Прості порівняльні алгоритми:
квадратична складність O($n^2$) ..... 198
7.4. Порівняльний аналіз алгоритмів з O($n^2$) складністю ..... 205
7.5. Ефективні порівняльні алгоритми: складність O(nlogn) ..... 206
7.6. Нижня межа складності для порівняльних сортувань ..... 211
7.7. Непорівняльні алгоритми: лінійна часова складність ..... 213
7.8. Практична реалізація та порівняльний аналіз ..... 215
7.9. Приклади розв'язку задач ..... 223
*Контрольні питання* ..... 225
*Задачі для самостійного опрацювання* ..... 226

**Тема 8. Стратегія «розділяй та володарюй»** ..... 228
8.1. Основні принципи парадигми ..... 228
8.2. Алгоритм Карацуби для множення ..... 232
8.3. Матричне множення: алгоритм Штрассена ..... 238
8.4. Пошук найближчої пари точок ..... 244

8.5. Розв’язання задач із застосуванням
стратегії «розділяй та володарюй» ..... 253
*Контрольні питання* ..... 257
*Задачі для самостійного опрацювання* ..... 258

**Тема 9. Динамічне програмування
та жадібні алгоритми** ..... 260
9.1. Динамічне програмування: принципи та приклади ..... 260
9.2. Класичні задачі динамічного програмування ..... 264
9.3. Жадібні алгоритми: принципи та приклади ..... 267
9.4. Коди Хаффмана ..... 269
9.5. Реалізація алгоритмів динамічного програмування ..... 270
9.6. Реалізація жадібних алгоритмів ..... 274
9.7. Редакційна відстань (відстань Левенштейна) ..... 279
9.8. Оптимальне множення матриць ..... 280
9.9. Задача про мінімальне покриття множини ..... 281
9.10. Приклади розв’язку задач ..... 282
*Контрольні питання* ..... 285
*Задачі для самостійного опрацювання* ..... 286

**Тема 10. Основи теорії графів та обходи** ..... 287
10.1. Вступ до теорії графів:
від абстракції до моделювання ..... 287
10.2. Класифікація та типологія графів ..... 289
10.3. Способи представлення графів у пам’яті комп’ютера ..... 290
10.4. Фундаментальні алгоритми обходу:
пошук у ширину (BFS) та в глибину (DFS) ..... 292
10.5. Програмна реалізація алгоритмів BFS та DFS ..... 294
10.6. Застосування обходів: пошук компонент зв’язності ..... 300
*Контрольні питання* ..... 303
*Задачі для самостійного опрацювання* ..... 303

**Тема 11. Найкоротші шляхи у графах** ..... 305
11.1. Вступ до задачі пошуку найкоротшого шляху ..... 305
11.2. Алгоритм Дейкстри ..... 306
11.3. Алгоритм Беллмана-Форда ..... 314

11.4. Алгоритм Флойда-Воршелла ..... 319
11.5. Кейс-стаді: оптимізація маршрутів
в Uber та Google Maps ..... 324
*Контрольні питання* ..... 327
*Задачі для самостійного опрацювання* ..... 327

**Тема 12. Мінімальні кістякові дерева** ..... 329
12.1. Проблема проєктування ефективних мереж ..... 329
12.2. Фундаментальні властивості
мінімальних кістякових дерев ..... 330
12.3. Алгоритм Крускала ..... 332
12.4. Структура даних
«Система неперетинних множин» (Union-Find) ..... 336
12.5. Реалізація алгоритму Крускала
з використанням Union-Find ..... 338
12.6. Алгоритм Пріма ..... 343
12.7. Підсумки та порівняльний аналіз ..... 349
*Контрольні питання* ..... 350
*Задачі для самостійного опрацювання* ..... 351

**Тема 13. Мережеві потоки та топологічне сортування** ..... 353
13.1. Вступ до орієнтованих графів
та лінійне впорядкування ..... 353
13.2. Алгоритми топологічного сортування:
DFS та Кана ..... 354
13.3. Структурна декомпозиція:
сильно зв’язані компоненти (SCC) ..... 360
13.4. Фундаментальні поняття мережевих потоків ..... 364
13.5. Теорема про максимальний потік
та мінімальний розріз ..... 365
13.6. Алгоритм Едмондса-Карпа
для знаходження максимального потоку ..... 366
13.7. Аналіз часової складності
алгоритму Едмондса-Карпа ..... 368
13.8. Алгоритм пошуку A*: оптимальний пошук шляху ..... 369

13.9. Поглиблений аналіз складності:
обернена функція Акермана (α(n)) ..... 370
13.10. Практичне застосування
мережевих потоків та графів ..... 371
13.11. Порівняльний аналіз ..... 372
*Контрольні питання* ..... 372
*Задачі для самостійного опрацювання* ..... 373

**Тема 14. Просторові та рядкові структури даних** ..... 375
14.1. Просторові структури даних
для запитів на відрізках ..... 375
14.2. Багатовимірні просторові структури: KD-дерева ..... 383
14.3. Алгоритми пошуку в рядках ..... 386
14.4. Реалізація алгоритму КМП ..... 390
*Контрольні питання* ..... 392
*Задачі для самостійного опрацювання* ..... 393

**Тема 15. Сучасні алгоритми** ..... 394
15.1. Алгоритми для машинного навчання та Big Data ..... 394
15.2. Потокові алгоритми (streaming algorithms)
та Count-Min Sketch ..... 409
15.3. Суфіксні масиви та дерева ..... 415
15.4. Алгоритм Ахо-Корасіка ..... 421
15.5. Зовнішні алгоритми
та системи Big Data (MapReduce) ..... 426
15.6. Приклади розв’язку задач ..... 430
*Контрольні питання* ..... 433
*Задачі для самостійного опрацювання* ..... 434

**Тема 16. Теорія складності** ..... 436
16.1. Межі обчислюваності.
Розв’язні, напіврозв’язні та нерозв’язні проблеми ..... 436
16.2. Проблема зупинки ..... 439
16.3. Класи часової складності: P та NP ..... 443
16.4. NP-повнота та NP-важкість ..... 446

16.5. Методи доведення NP-повноти ..... 447
16.6. Робота з NP-важкими задачами.
Апроксимаційні алгоритми ..... 454
16.7. Приклади розв’язку задач ..... 463
*Контрольні питання* ..... 467
*Задачі для самостійного опрацювання* ..... 468

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ** ..... 470

# ПЕРЕДМОВА

Запропонований навчальний посібник «Теорія алгоритмів та структура даних» створений для здобувачів вищої освіти спеціальності F3 «Комп'ютерні науки». Цей курс є одним із фундаментальних у підготовці сучасного фахівця в галузі інформаційних технологій, оскільки саме алгоритми та структури даних становлять теоретичний і практичний фундамент усієї комп'ютерної науки.

Вивчення алгоритмів – це не просто засвоєння набору готових рішень, а розвиток особливого типу мислення, що дає змогу декомпозувати складні проблеми на керовані компоненти, аналізувати ефективність різних підходів та обирати оптимальні стратегії розв'язання задач. У цьому навчальному посібнику було заплановано поєднати строгість математичного обґрунтування з практичною спрямованістю, надаючи як теоретичні основи, так і конкретні приклади реалізації на популярних мовах програмування – Python та C/C++.

Матеріал побудовано так, щоб забезпечити поступове зростання складності: від базових понять про складність алгоритмів і простих структур даних до сучасних методів обробки великих обсягів інформації та елементів теорії складності обчислень. Кожна темацього навчального посібника супроводжується детальним аналізом та прикладами розв'язання задач з реальних ситуацій.

Сучасний світ цифрових технологій ставить перед фахівцями з комп'ютерних наук значні виклики. Обсяги даних, що генеруються щосекунди, вимагають ефективних алгоритмів для їх обробки, аналізу та зберігання. Від швидкості пошуку в пошукових системах до оптимізації маршрутів у додатках для таксі, від захисту даних у розподілених системах до навчання моделей штучного інтелекту – всюди в основі лежать фундаментальні алгоритмічні концепції.

Актуальність цього курсу визначається декількома ключовими чинниками. По-перше, індустрія розробки програмного забезпечення активно використовує складні структури даних та алгоритми в повсякденній практиці – від хеш-таблиць у базах даних до графових алгоритмів у соціальних мережах. По-друге, технічні

співбесіди в провідних IT-компаніях світу (Google, Meta, Amazon, Microsoft) значною мірою зосереджені саме на знанні алгоритмів та вмінні аналізувати їх ефективність. По-третє, розвиток таких сфер, як машинне навчання, Big Data, блокчейн-технології та хмарні обчислення, неможливий без глибокого розуміння алгоритмічних основ. Нарешті, уміння правильно обирати структури даних та алгоритми безпосередньо впливає на продуктивність, масштабованість та надійність програмних систем, що створюються.

Курс «Теорія алгоритмів та структура даних» органічно інтегрований у систему професійної підготовки бакалаврів з комп'ютерних наук і тісно пов'язаний з рядом інших дисциплін навчального плану.

Фундаментальною основою курсу є дискретна математика, зокрема теорія графів, комбінаторика, математична логіка та теорія множин. Названі розділи математики надають формальний апарат для аналізу та доведення коректності алгоритмів. Лінійна алгебра та математичний аналіз використовуються для розуміння асимптотичної складності, аналізу рекурентних співвідношень та роботи з багатовимірними структурами даних.

Тісний зв'язок існує з курсами програмування, зокрема з курсом «Об'єктно-орієнтоване програмування». Знання, отримані під час вивчення алгоритмів, безпосередньо застосовуються в об'єктно-орієнтованому програмуванні, де структури даних реалізуються у вигляді класів та абстрактних типів даних. Курси з архітектури комп'ютерів та операційних систем доповнюють розуміння моделей пам'яті, кешування та оптимізації роботи з пам'яттю, що є критичним для ефективної реалізації алгоритмів.

У подальшому навчанні матеріал курсу стане основою для вивчення баз даних (індексування, оптимізація запитів, B-дерева), штучного інтелекту (алгоритми пошуку, графові методи, оптимізація), комп'ютерних мереж (алгоритми маршрутизації, протоколи), а також спеціалізованих курсів з машинного навчання, Big Data, розподілених систем та паралельних обчислень.

Глибоке розуміння алгоритмів і структур даних є однією з важливих компетенцій сучасного фахівця з комп'ютерних наук, програміста, що безпосередньо визначає його конкурентоспроможність на ринку праці та успішність у професійній кар'єрі.

У практиці розробки програмного забезпечення знання алгоритмів дає змогу приймати обґрунтовані рішення щодо вибору оптимальних структур даних та методів обробки інформації. Це критично важливо під час створення масштабованих систем, де неправильний вибір може призвести до деградації продуктивності в разі зростання навантаження. Наприклад, розуміння різниці між O(n) та O(n log n) може означати різницю між системою, що обробляє запити за мілісекунди, і системою, що працює хвилинами.

Для системних архітекторів і технічних лідерів цей курс надає інструментарій для проєктування ефективних рішень на рівні архітектури. Знання алгоритмів оптимізації, структур для розподілених систем (консистентне хешування, фільтри Блума) та методів обробки великих даних дає змогу створювати системи, здатні обслуговувати мільйони користувачів.

У галузі Data Science та машинного навчання алгоритми є основою роботи з великими обсягами даних. Ефективні структури для зберігання та пошуку, потокові алгоритми, методи оптимізації – усе це безпосередньо використовується під час побудови моделей та аналізуванні даних. Розуміння складності та можливості оптимізації дає змогу працювати з наборами даних, що містять мільярди записів.

Не менш важливим є значення курсу для дослідницької діяльності. Теорія складності та розуміння меж обчислюваності потрібні для визначення принципової можливості розв’язання задачі та пошуку оптимальних або приблизних методів. Знання класичних алгоритмів стає відправною точкою для розробки нових, більш ефективних підходів у специфічних предметних сферах.

Нарешті, володіння алгоритмами та структурами даних є обов’язковою умовою для успішного проходження технічних співбесід у провідних технологічних компаніях. Уміння аналізувати складність, оптимізувати рішення та обирати правильні структури даних демонструє зрілість інженерного мислення та готовність до розв’язання складних практичних задач.

Навчальний посібник складається з шістнадцяти тем, які логічно структуровані від базових концепцій до просунутих методів та теоретичних основ.

Теми 1–2 закладають фундамент курсу, вводячи базові поняття аналізу алгоритмів, асимптотичну нотацію, методи аналізу рекурсивних алгоритмів, охоплюючи рекурентні співвідношення та Майстер-теорему. Особлива увага приділяється розумінню різних моделей обчислень та взаємозв'язку між алгоритмами й структурами даних.

Теми 3–6 присвячені фундаментальним структурам даних. Розглядаються лінійні структури (масиви, списки, стеки, черги), хеш-таблиці з детальним аналізом методів розв'язання колізій, деревоподібні структури (бінарні дерева пошуку, AVL-дерева, червоно-чорні дерева, B-дерева) та купи різних типів. Кожна структура аналізується з точки зору часової та просторової складності операцій.

Теми 7–9 фокусуються на класичних алгоритмічних парадигмах. Детально розглядаються алгоритми сортування, стратегія «розділяй та володарюй» з такими прикладами, як алгоритми Карацуби та Штрассена, а також динамічне програмування та жадібні алгоритми з численними практичними задачами.

Теми 10–13 насамперед присвячені теорії графів та алгоритмам на графах. Від базових обходів (BFS, DFS) до алгоритмів пошуку найкоротших шляхів (Дейкстри, Беллмана-Форда, Флойда-Воршелла), від побудови мінімальних кістякових дерев (алгоритми Крускала та Пріма) до мережевих потоків та топологічного сортування. Окремі розділи містять кейс-стаді реальних застосувань у таких сервісах, як Uber та Google Maps.

Теми 14–15 охоплюють спеціалізовані та сучасні алгоритми: просторові структури даних (дерева відрізків, KD-дерева), алгоритми пошуку в рядках (КМП, Рабіна-Карпа, Ахо-Корасіка), а також сучасні методи для роботи з великими даними, охоплюючи потокові алгоритми, фільтри Блума та суфіксні структури.

Тема 16 присвячена теорії складності обчислень, де розглядаються класи P та NP, поняття NP-повноти та NP-важкості, межі обчислюваності та апроксимаційні алгоритми для розв'язання складних задач.

Кожна тема містить теоретичний матеріал з математичним обґрунтуванням, приклади реалізації на Python та C/C++, детальний аналіз складності та практичні задачі для закріплення матеріалу. Наскрізні кейс-стаді демонструють застосування розглянутих алгоритмів у реальних системах провідних технологічних компаній.

# ТЕМА 1
# ОСНОВИ АНАЛІЗУ АЛГОРИТМІВ

Будь-яка програма, від простого калькулятора на смартфоні до складної системи управління польотами, є реалізацією одного чи багатьох алгоритмів. Алгоритм – це, по суті, рецепт, правило, чітка послідовність дій для досягнення бажаного результату. Проте, як приклад, і в кулінарії, для однієї й тієї самої страви може існувати декілька рецептів, одні з них швидкі та елегантні, а інші вимагатимуть багато часу (повільні) та громіздкі.

У свій час Ніклаус Вірт[1] сформулював знамените рівняння: «Алгоритми + Структури даних = Програми». Ця формула підкреслює нерозривний зв'язок між послідовністю дій (алгоритмом) та способом організації даних (структурою даних), над якими ці дії виконуються. Ефективність програми залежить не лише від алгоритму, але й від того, наскільки вдало підібрана структура даних для конкретної задачі.

[1] Ніклаус Вірт (Niklaus Wirth) – швейцарський вчений у галузі інформатики, один із піонерів програмування та розробки мов програмування. Народився 15 лютого 1934 року в Вінтертурі, Швейцарія. Основні досягнення:

1. Мови програмування. Відомий як творець кількох важливих мов програмування, серед яких:

– Pascal (1970) – одна з найвідоміших і найвпливовіших мов програмування, особливо у сфері освіти.

– Modula (1975) та Modula-2 (1978) – мови, орієнтовані на модульність і системне програмування.

– Oberon (1986) - мова програмування та операційна система, створені для простоти, ефективності й елегантності.

2. Принципи проєктування. Вірт просував ідеї простоти, строгої типізації та модульності в програмуванні. Він вірив у те, що хороше програмне забезпечення має бути зрозумілим, надійним і ефективним.

Вірт більшу частину своєї наукової кар'єри провів у Швейцарській вищій технічній школі Цюриха (ETH Zurich), де займався викладанням, дослідженнями та розробкою нових мов і систем. Нагороди: Премія Тюрінга (1984) – найпрестижніша нагорода в галузі інформатики, яку Вірт отримав за «розробку послідовності інноваційних мов програмування».

## 1.1. Поняття та властивості алгоритму

Термін «алгоритм» походить від латинізованої форми імені видатного перського математика IX століття Мухаммада ібн Муси аль-Хорезмі[2] (Algorithmi), який у своїх працях сформулював правила виконання арифметичних операцій. У сучасному розумінні, *алгоритм* – це скінченна, впорядкована послідовність чітко визначених, виконуваних інструкцій, призначена для розв'язання певного класу задач. Важливо відмітити, що алгоритм є абстрактною сутністю, тобто такою ідеєю, яка не залежить від мови програмування, комп'ютера чи виконавця. Це формалізований план дій, який може бути реалізований у вигляді комп'ютерної програми, блок-схеми або словесного опису.

Щоб послідовність інструкцій могла вважатися алгоритмом, вона повинна задовольняти ряду фундаментальних властивостей, які є властивостями алгоритмів.

1. Дискретність (Discreteness). Алгоритм має складатися з окремих, чітко відокремлених кроків або команд. Виконання алгоритму – це послідовний перехід від одного кроку до наступного, причому кожен крок виконується за скінченний проміжок часу, і лише після його повного завершення виконавець переходить до наступного. Ця властивість протиставляє алгоритмічні процеси неперервним (аналоговим) процесам і забезпечує їхню структурованість та можливість покрокового аналізу.

---

[2] Мухаммад ібн Муса аль-Хорезмі бл. 780–850 н. е.) – видатний перський математик, астроном, географ і астролог, який жив і працював у Золотому віці ісламу, головним чином у Багдаді при Будинку мудрості (Байт аль-Хікма). Його внесок у науку став основою багатьох напрямів сучасної математики. Основні досягнення:

1. Засновник алгебри. Його найвідоміша праця – «Кітаб аль-джабр валь-мукабала», що в перекладі означає «Книга відновлення і протиставлення». Саме зі слова «аль-джабр» (al-jabr) походить сучасний термін «алгебра». У цій праці він описував методи розв'язання лінійних і квадратних рівнянь, використовуючи словесні описи, а не символи (символічна алгебра з'явиться пізніше).

2. Запровадження десяткової системи та арабських цифр. Аль-Хорезмі відіграв важливу роль у поширенні індійської цифрової системи, яка згодом стала відомою як арабські цифри в Європі. Його праця про обчислення з індійськими цифрами перекладена на латину в XII столітті як «Algoritmi de numero Indorum», звідки походить слово «алгоритм» (від латинізованого імені автора – Algoritmi).

2. Визначеність (Determinism/Definiteness). Кожна інструкція алгоритму повинна бути абсолютно точною, однозначною і не припускати жодних двозначностей у тлумаченні. Для будь-якого набору вхідних даних алгоритм повинен генерувати одну й ту саму послідовність дій і, відповідно, однаковий результат, скільки б разів не повторювались вхідні дані. Інструкції на кшталт «трохи підігріти молоко» або «взяти 2–3 ложки цукру» є неприпустимими в алгоритмах, оскільки різні виконавці можуть інтерпретувати їх по-різному.

3. Скінченність (Finiteness). Алгоритм повинен гарантовано завершувати свою роботу після виконання і мати скінченну кількість кроків для будь-яких допустимих вхідних даних. Ця властивість є критично важливою, оскільки вона забезпечує вимогу: розв'язок задачі буде отримано за реальний час. Послідовність дій, яка може виконуватися нескінченно (зациклюватися), не є алгоритмом. Наприклад, процедура, що постійно додає 7 до додатного числа, ніколи не завершиться і, отже, не є алгоритмом.

4. Масовість (Massiveness / Generality). Алгоритм розробляється для розв'язання не однієї конкретної задачі, а цілого класу однотипних задач, яківідрізняються лише вхідними даними. Наприклад, алгоритм сортування масиву повинен працювати для будь-якого масиву чисел. Ця властивість перетворює алгоритм на універсальний інструмент.

5. Результативність та коректність (Effectiveness and Correctness). Виконання алгоритму завжди повинно приводити до отримання конкретного і правильного результату. Кожен крок алгоритму має бути елементарним, тобто таким, що кожен виконавець може його здійснити. Кінцевий результат повинен бути коректним розв'язком поставленої задачі.

Описані вище властивості не є формальним переліком, вони логічно взаємопов'язані й формують цілісну концепцію. Наприклад, властивість скінченності втрачає сенс без визначеності, тобто якщо алгоритм може поводитися по-різному на однакових вхідних даних, то для одного шляху він може завершитися, а для іншого – ні, що робить поняття скінченності розмитим. Отже, визначеність є логічною передумовою для скінченності. Зі свого

боку результативність неможлива без скінченності, алгоритм, який ніколи не зупиняється, не може видати конкретний результат. Отже, вказані властивості утворюють логічний ланцюг, який гарантує, що алгоритм є надійним, передбачуваним та корисним інструментом для розв'язання задач.

## 1.2. Моделі обчислень для аналізу

Аналіз ефективності алгоритму[3] вимагає підрахунку виконаних операцій. Однак реальні комп'ютери надзвичайно складні – різні процесори мають різні набори інструкцій, різну швидкість виконання, а компілятори можуть по-різному оптимізувати код[4]. Пряме вимірювання часу виконання в секундах робить аналіз залежним від конкретного обладнання та програмного середовища. Щоб уникнути цього і мати можливість порівнювати алгоритми на абстрактному, математичному рівні, в теорії алгоритмів використовуються *моделі обчислень* – ідеалізовані математичні абстракції комп'ютера.

Найпоширенішою моделлю для аналізу алгоритмів є *RAM-машина* (машина з довільним доступом до пам'яті). Вона достатньо проста для аналізу, але водночас достатньо точно моделює роботу більшості сучасних комп'ютерів.

---

[3] Ефективність алгоритму - це міра того, наскільки добре алгоритм виконує свою задачу з точки зору часу і пам'яті (ресурсів), які він споживає. Ефективність важлива тому, що:

1. Великий обсяг даних = велике навантаження → потрібно знати, як алгоритм масштабується.
2. Обмеження ресурсів (пам'ять, час виконання, електроенергія тощо).
3. Оптимізація програм для швидкої роботи, особливо в системах реального часу, мобільних додатках, серверних рішеннях тощо.

[4] Оптимізувати код – це означає змінити його так, щоб він працював ефективніше, не змінюючи водночас його функціональності (тобто результат має залишитися тим самим).

1. Продуктивність (швидкодія). Зменшити час виконання - наприклад, замінити повільний алгоритм на швидший.
2. Економія пам'яті. Зменшити обсяг використаної оперативної пам'яті або уникнути дублювання даних.
3. Читабельність і підтримуваність. Зробити код чистішим, зрозумілішим, щоб інші програмісти (або ви в майбутньому) могли легше його змінювати.
4. Уникнення зайвих операцій. Прибрати дублікати обчислень, непотрібні цикли, неефективні умови тощо.

Компоненти RAM-машини.

1. Центральний процесор[5] (CPU) – виконує інструкції послідовно, одну за одною.

2. Пам’ять – розглядається як нескінченний масив комірок (регістрів), кожна з яких має унікальний числовий адрес (індекс). Процесор може звернутися до будь-якої комірки пам’яті за її адресом.

3. Програма – набір інструкцій, що зберігається окремо від даних (так звана Гарвардська архітектура) і не може модифікувати сама себе під час виконання.

Основні припущення (правила) RAM-моделі.

1. Кожна «проста» операція – арифметична, логічна, порівняння, присвоєння – виконується за один умовний крок, тобто за сталий час.

2. Доступ до будь-якої комірки пам’яті (читання або запис) також займає один крок, незалежно від її адреси. Саме ця властивість називається «довільним доступом» (random access).

3. Цикли та виклики функцій не є простими операціями. Їхня вартість обчислюється як сума вартостей усіх операцій, що виконуються в їх тілі впродовж усіх ітерацій або рекурсивних викликів.

---

[5] Центральний процесор (ЦП, англ. CPU – Central Processing Unit) – це основний компонент комп’ютера, який виконує всі обчислення і керує роботою інших частин системи. Його часто називають «мозком комп’ютера». ЦП виконує:

1. Обробку інструкцій з програм.
2. Обчислення (арифметичні, логічні операції).
3. Керування іншими пристроями (через шини і контролери).
4. Передачу даних між оперативною пам’яттю (RAM), відеокартою, жорстким диском тощо.

Основні частини ЦП:

| Компонент | Опис |
|---|---|
| ALU (Арифметико-логічний пристрій) | Виконує математичні й логічні операції |
| CU (Пристрій керування) | Керує порядком виконання інструкцій |
| Регістри | Дуже швидка пам’ять для тимчасового зберігання даних |
| Кеш-пам’ять (L1, L2, L3) | Проміжна пам’ять між RAM і процесором для пришвидшення доступу до даних |

Цикл роботи процесора (так званий машинний цикл):

1. Fetch (отримання) - процесор отримує інструкцію з пам’яті.
2. Decode (декодування) - розпізнає, що саме треба зробити.
3. Execute (виконання) - виконує команду.
4. Store (збереження) - результат зберігається або передається далі.

4. Комірки пам’яті можуть зберігати цілі числа довільного розміру. Це важливе спрощення, яке ігнорує обмеження розрядності реальних процесорів.

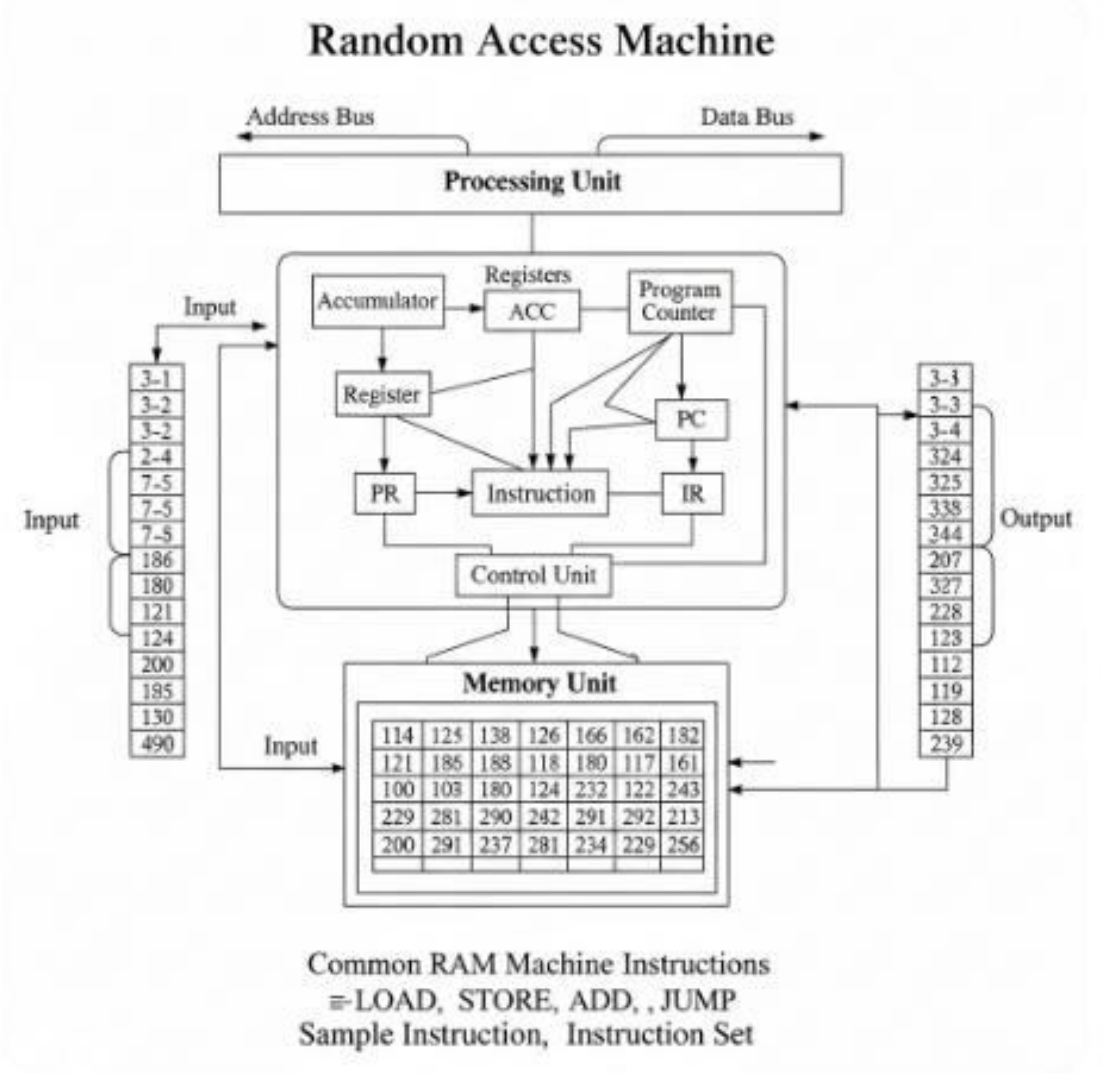



Рисунок 1.1 – Схема RAM-машини

Ця модель[6] робить свідомі спрощення. Наприклад, у реальному процесорі множення займає більше часу, ніж додавання, а доступ до віддалених комірок пам’яті може бути повільнішим. Однак ці відмінності зазвичай є константними чинниками, які не впливають на асимптотичну швидкість росту часу виконання, що є основним об’єктом аналізу.

[6] Модель – це спрощене уявлення про об’єкт, процес або явище**,** створене для того, щоб його легше було вивчати, пояснювати або прогнозувати. Модель – це уявлення реальності**,** яке ми використовуємо, щоб щось зрозуміти, проаналізувати або відтворити.

Хоча RAM-машина є стандартом для базового аналізу, існують більш складні моделі, що враховують специфічні аспекти сучасного обладнання.

1. Word RAM. Ця абстракція комп'ютера є уточненням RAM-машини і враховує, що реальні комп'ютери оперують даними фіксованого розміру, які називаються машинними словами[7] (наприклад, 64 біти). У цій моделі припускається, що операції над словами розміром $w$ біт займають константний час. Важливим припущенням є те, що розмір слова $w$ достатньо великий, щоб зберігати адресу будь-якої комірки пам'яті, тобто $w \geq \log n$, де $n$– розмір вхідних даних. Це дає змогу створювати більш ефективні алгоритми для певних задач, наприклад, для сортування цілих чисел.

2. Модель зовнішньої пам'яті (External Memory Model). Ця модель є незамінною для аналізу алгоритмів, що працюють з величезними обсягами даних, які не вміщуються в оперативну пам'ять (RAM) і зберігаються на повільних носіях, таких як жорсткі диски (HDD) або твердотільні накопичувачі (SSD). Модель вводить дворівневу ієрархію пам'яті. Перша – це швидка внутрішня пам'ять розміром $M$. Друга – повільна нескінченна зовнішня пам'ять. Дані між ними передаються блоками розміром $B$. Метою аналізу в цій моделі є не мінімізація кількості процесорних операцій, а мінімізація кількості операцій вводу-виводу (I/O), тобто читання та запису блоків, оскільки саме вони є основним вузьким місцем.

---

[7] Машинне слово – це основна одиниця даних, з якою безпосередньо працює процесор комп'ютера. Формально – це послідовність бітів (0 і 1), довжина якої визначається розрядністю (архітектурою) процесора. Наприклад: у 8-бітних процесорах машинне слово = 8 бітів, у 16-бітних – 16 бітів, у 32-бітних – 32 біти, у 64-бітних – 64 біти. Властивості машинного слова.

1. Довжина слова визначає максимальний розмір даних, які можна обробляти за одну операцію.
2. Адресація пам'яті часто ведеться в машинних словах або байтах.
3. У слові можуть зберігатися: число, адреса, команда, логічні дані.

Простіше кажучи, машинне слово – це «рідна» кількість бітів, яку процесор може обробити за один такт.

3. Кеш-моделі (Cache-aware / Cache-oblivious Models). Сучасні процесори мають багаторівневу ієрархію кеш-пам'яті[8] (L1, L2, L3), яка є значно швидшою за основну оперативну пам'ять. Кеш-моделі враховують цей факт.

Кеш-залежні (Cache-aware) алгоритми явно використовують параметри кешу (розмір $M$, довжину кеш-лінії $B$) для оптимізації своєї роботи. Вони можуть бути дуже ефективними, але втрачають портативність.

Кеш-незалежні (Cache-oblivious) алгоритми розробляються без урахування параметрів кешу. Вони зазвичай використовують рекурсивний підхід «розділяй та володарюй», розбиваючи задачу на підзадачі, доки вони не стануть настільки малими, що гарантовано вмістяться в кеш будь-якого розміру. Такі алгоритми є одночасно ефективними та портативними[9].

---

[8] Кеш-пам'ять (Cache memory) – це дуже швидка проміжна пам'ять між процесором і оперативною пам'яттю (RAM). Її завдання – зменшити час доступу до даних і команд, які процесор використовує найчастіше.

Рівні кешу:

1. L1 (Level 1 cache). Найшвидший і найменший кеш (зазвичай 16–128 КБ). Міститься безпосередньо всередині ядра процесора. Ділиться на інструкційний кеш (для команд) і кеш даних (для чисел/змінних). Час доступу: кілька тактів процесора.
2. L2 (Level 2 cache). Трохи повільніший, але більший за L1 (128 КБ – 2 МБ). Може бути як у кожного ядра окремо, так і спільним для кількох ядер. Використовується для збереження даних, що не вмістилися в L1.
3. L3 (Level 3 cache). Ще більший (зазвичай від 2 МБ до кількох десятків МБ). Повільніший за L1 і L2, але все ще швидший за оперативну пам'ять. Зазвичайспільний для всіх ядер процесора.

Принцип роботи. Коли процесору потрібні дані, він спочатку шукає їх у L1. Якщо немає – звертається до L2. Якщо і там немає - у L3. Якщо дані відсутні в усіх кешах (cache miss) – процесор отримує їх з оперативної пам'яті, що значно повільніше.

Інакше кажучи, кеш - це «надшвидкий буфер» між CPU і RAM, який мінімізує затримки.

[9] Портативні алгоритми – це алгоритми, які можна реалізувати та виконати на різних апаратних платформах і програмних середовищах без суттєвих змін у коді чи логіці. Основні риси:

1. Незалежність від конкретного обладнання – алгоритм можна реалізувати як на ПК, так і на мобільному пристрої чи вбудованій системі.
2. Мінімальна залежність від мови програмування – формулюється у вигляді логічних кроків, зрозумілих для реалізації на різних мовах.
3. Адаптивність - можна оптимізувати під різні архітектури, але базова ідея залишається спільною.

Приклади портативних алгоритмів: алгоритми сортування, алгоритми пошуку, криптографічні алгоритми – реалізуються на різних пристроях, алгоритми обробки графів (DFS, BFS). Простіше кажучи, портативний алгоритм – це універсальний рецепт, який можна виконати на будь-якій «кухні» (комп'ютерній системі), якщо є потрібні інгредієнти (дані та ресурси).

Еволюція моделей обчислень від простої RAM-машини до складних кеш-незалежних моделей не є випадковою. Вона безпосередньо відображає історію розвитку комп'ютерної архітектури та боротьбу з так званою «стіною пам'яті» (memory wall) – зростаючим розривом між швидкістю процесорів та швидкістю доступу до пам'яті. Спочатку, коли ці швидкості були співмірними, RAM-модель була адекватним наближенням. Згодом, коли процесори стали значно швидшими, затримка пам'яті перетворилася на основний обмежувальний чинник. Це призвело до створення ієрархії кешів. Теоретична інформатика відреагувала на цю апаратну реальність, розробивши більш точні моделі, які дають змогу аналізувати та проєктувати алгоритми, що ефективно використовують ієрархію пам'яті.

## 1.3. Оцінка ефективності: часова та просторова складність

Після вибору моделі обчислень ми можемо перейти до кількісної оцінки ефективності алгоритму. Основними метриками є часова та просторова складність.

Часова складність (Time Complexity) – це функція $T(n)$, яка визначає кількість елементарних операцій (кроків), що виконує алгоритм, як функцію від розміру вхідних даних $n$. Важливо наголосити, що це не реальний час у секундах, а абстрактна величина, яка дає змогу порівнювати алгоритми незалежно від швидкості комп'ютера. Розмір вхідних даних $n$ залежить від задачі. Так, для сортування масиву – це кількість елементів, для операцій з графами – кількість вершин та ребер, для обробки чисел – кількість бітів у їхньому представленні.

Просторова складність (Space Complexity) – це функція $S(n)$, яка визначає обсяг пам'яті, необхідний алгоритму для виконання, як функцію від розміру вхідних даних $n$. Під час аналізування просторової складності прийнято розрізняти:

– простір для вхідних даних – пам'ять, що займають самі вхідні дані;

– допоміжний простір (Auxiliary Space) – додаткова або тимчасова пам’ять, яку алгоритм використовує під час своєї роботи (наприклад, для зберігання проміжних результатів, змінних циклу, стеку рекурсивних викликів). Зазвичай, коли говорять про просторову складність, мають на увазі саме допоміжний простір. Алгоритм, що вимагає O(1) допоміжного простору, називається алгоритмом «на місці» (in-place).

Часто існує компроміс (trade-off) між часовою та просторовою складністю. Тобто можна розробити швидший алгоритм, який потребуватиме більше пам’яті, або навпаки, зекономити пам’ять за рахунок збільшення часу виконання.

## 1.4. Асимптотичний аналіз

Точний підрахунок кількості операцій для алгоритму може бути складним і часто непотрібним. Наприклад, два алгоритми з часом виконання $T1(n) = 100n^2 + 500n$ та $T2(n) = 5n^2 + 1\ 000n$ можуть мати різну кількість операцій для малих $n$, але за зростання $n$ поведінка обох функцій визначатиметься домінувальним членом $n^2$. Константи (100, 5) та члени нижчого порядку (500n, 1 000n) стають несуттєвими.

Асимптотичний аналіз – це метод опису поведінки функцій складності під час прямування розміру вхідних даних до нескінченності ($n \to \infty$). Він дає змогу класифікувати алгоритми за швидкістю росту їхніх ресурсних вимог, ігноруючи коефіцієнти вищих порядків та члени нижчого порядку. Для цього використовуються спеціальні асимптотичні нотації.

Нотація O (O-велике – Big O). Формальне визначення наступне. Кажуть, що функція $f(n)$ належить класу $O(g(n))$ (читається як «О велике від же від ен»), якщо існують такі додатні константи $c$ та $n_0$, що для всіх $n \geq n_0$ виконується нерівність $0 \leq f(n) \leq c \cdot g(n)$.

Інтуїтивне пояснення таке. Нотація $O(g(n))$ надає асимптотичну верхню межу для функції $f(n)$. Вона показує, що $f(n)$ зростає не швидше, ніж $g(n)$ з точністю до коефіцієнта вищого порядку. В аналізі алгоритмів O-нотація найчастіше використовується

для опису найгіршого випадку часової складності, гарантуючи, що час роботи алгоритму не перевищить певну межу для великих $n$.

Нотація Ω (омега-велике). Формальне визначення: кажуть, що функція $f(n)$ належить класу $\Omega(g(n))$, якщо існують такі додатні константи $c$ та $n_0$, що для всіх $n \geq n_0$ виконується нерівність $0 \leq c \cdot g(n) \leq f(n)$.

Інтуїтивне пояснення. Нотація $\Omega(g(n))$ надає асимптотичну нижню межу. Вона показує, що $f(n)$ зростає не повільніше, ніж $g(n)$. Ця нотація використовується для опису найкращого випадку або для доведення того, що будь-який алгоритм для розв'язання певної задачі вимагатиме щонайменше певного часу.

Нотація Θ (тета – велике). Формальне визначення: кажуть, що функція $f(n)$ належить класу Θ(g(n)), якщо $f(n) = O(g(n))$ і $f(n) = \Omega(g(n))$. Це еквівалентно тому, що існують такі додатні константи $c_1, c_2$ та $n_0$, що для всіх $n \geq n_0$ виконується нерівність

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

Інтуїтивне пояснення. Нотація $\Theta(g(n))$ надає асимптотично точну межу. Вона означає, що $f(n)$ зростає з такою самою швидкістю, як і $g(n)$. Функція $f(n)$«затиснута» між $c_1 \cdot g(n)$ та $c_2 \cdot g(n)$. Ця нотація дає найточнішу характеристику складності алгоритму, коли його час роботи у найкращому та найгіршому випадках зростає однаково.

Для кращого розуміння цих нотацій наведемо порівняльну таблицю.

Таблиця 1.1 – Порівняння асимптотичних нотацій

| Нотація | Математичне відношення | Аналогія | Опис |
|---|---|---|---|
| $f(n) = O(g(n))$ | $f(n) \leq c \cdot g(n)$ | $\leq$ | Верхня межа (не гірше ніж) |
| $f(n) = \Omega(g(n))$ | $f(n) \geq c \cdot g(n)$ | $\geq$ | Нижня межа (не краще ніж) |
| $f(n) = \Theta(g(n))$ | $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot (n)$ | == | Точна межа (саме так) |

На практиці алгоритми класифікують за швидкістю росту їхньої складності.

Таблиця 1.2 – Класи складності та приклади

| Клас складності | Нотація | Опис | Приклад алгоритму |
|---|---|---|---|
| Константна | O(1) | Час виконання не залежить від n | Доступ до елемента масиву за індексом |
| Логарифмічна | O(logn) | Час виконання зростає дуже повільно | Бінарний пошук |
| Лінійна | O(n) | Час виконання зростає пропорційно n | Лінійний пошук |
| Лінійно-логарифмічна | O(nlogn) | Ефективне сортування | Сортування злиттям |
| Квадратична | $O(n^2)$ | Час виконання зростає як $n^2$ | Сортування бульбашкою, вставками |
| Кубічна | $O(n^3)$ | Час виконання зростає як $n^3$ | Множення матриць (наївний метод) |
| Експоненційна | $O(2^n)$ | Дуже повільно, практично для малих n | Задача комівояжера (перебір) |
| Факторіальна | O(n!) | Надзвичайно повільно | Генерація всіх перестановок |

## 1.5. Аналіз найкращого, середнього та найгіршого випадків

Час роботи алгоритму може суттєво залежати не лише від розміру вхідних даних *n*, але і від їхньої структури. Наприклад, алгоритм сортування може працювати значно швидше на вже відсортованому масиві. Тому для повного розуміння поведінки алгоритму розглядають три сценарії.

1. Найкращий випадок (Best Case). Це вхідні дані розміру *n*, для яких алгоритм виконує найменшу кількість операцій. Цей випадок показує «оптимістичну» межу продуктивності.

2. Найгірший випадок (Worst Case). Це вхідні дані розміру $n$, для яких алгоритм виконує найбільшу кількість операцій. Цей аналіз дає гарантовану верхню межу часу роботи.

3. Середній випадок (Average Case). Це усереднений час роботи по всіх можливих вхідних даних розміру $n$. Для цього аналізу потрібно мати уявлення про ймовірнісний розподіл вхідних даних, що часто є складною задачею.

На практиці найчастіше аналізують *найгірший випадок*. Причини цього такі:

1. Гарантія. Складність у найгіршому випадку – це гарантована верхня межа часу роботи, що є важливою характеристикою для критичних систем.

2. Частота. Для деяких алгоритмів (наприклад, пошук у базі даних) найгірший випадок (елемент не знайдено) може траплятися досить часто.

3. Простота аналізу. Аналіз середнього випадку часто вимагає складних математичних розрахунків та припущень щодо розподілу даних, тоді як найгірший випадок зазвичай легше визначити та проаналізувати.

Важливо чітко розрізняти поняття асимптотичних нотацій та аналізу випадків. Поширена хибна думка, що «Big O – це найгірший випадок». Це не так. Асимптотичні нотації ($O,\Omega,\Theta$) – це математичні інструменти для опису меж зростання функцій. Аналіз випадків (найкращий, середній, найгірший) – це спосіб визначення цих функцій для конкретного алгоритму. Ми можемо застосувати будь-яку нотацію до будь-якого випадку. Наприклад, для сортування вставками:

– час роботи в найкращому випадку точно дорівнює $\Theta(n)$;

– час роботи в найгіршому випадку точно дорівнює $\Theta(n^2)$;

– ми можемо сказати, що час роботи алгоритму загалом є $O(n^2)$, оскільки це верхня межа для будь-якого випадку.

Отже, O-нотація описує верхню межу, яка найчастіше застосовується до функції найгіршого випадку, але не є синонімом цього випадку.

## 1.6. Структури даних

*Структура даних* – це спосіб організації, зберігання та управління даними в комп'ютері, який дає змогу ефективно виконувати над ними різні операції. Це не просто набір даних, а набір, що має певну логічну будову та зв'язки між елементами.

Вибір правильної структури даних є критично важливим для ефективності програми. Один і той самий алгоритм може мати кардинально різну продуктивність залежно від того, як організовані дані, з якими він працює. Наприклад, пошук елемента у невпорядкованому списку вимагає перегляду всіх елементів, тоді як у збалансованому дереві пошуку та сама операція виконується значно швидше.

Алгоритми та структури даних нерозривно пов'язані між собою. Алгоритм визначає логіку обробки даних, а структура надає «каркас» для цих даних. Часто вибір алгоритму диктує використання певної структури даних, і навпаки, обрана структура даних обмежує набір ефективних алгоритмів, які можна до неї застосувати.

Існує велике різноманіття структур даних та алгоритмів, які можна класифікувати за різними критеріями.

Класифікація структур даних.

1. За логічною організацією.

1.1. Лінійні – елементи розташовані послідовно (масив, зв'язний список, стек, черга).

1.2. Нелінійні – елементи мають складніші, ієрархічні або мережеві зв'язки (дерева, графи, хеш-таблиці).

2. За фізичним зберіганням.

2.1. Статичні – розмір структури фіксується під час компіляції (статичний масив).

2.2. Динамічні – розмір може змінюватися під час виконання програми (зв'язний список, динамічний масив).

Класифікація алгоритмів (за парадигмами проєктування).

1. Розділяй та володарюй (Divide and Conquer). Задача розбивається на менші незалежні підзадачі, які розв'язуються рекурсивно, а потім їхні розв'язки об'єднуються.

2. Жадібні алгоритми (Greedy Algorithms). На кожному кроці робиться локально оптимальний вибір з надією, що це призведе до глобально оптимального розв'язку.

3. Динамічне програмування (Dynamic Programming). Задача розбивається на підзадачі, що перекриваються. Розв'язки підзадач зберігаються, щоб уникнути повторних обчислень.

## 1.7. Практичний аналіз ітеративних алгоритмів

Ітеративні алгоритми, що використовують цикли, є одними з найпоширеніших. Їх аналіз базується на кількох простих правилах. Якщо тіло циклу виконує константну кількість операцій (O(1)), то загальна часова складність дорівнює кількості ітерацій циклу.

```
// цикл, що виконується n разів
for i from 1 to n:
  // Дія, що займає O(1) часу
  print(i)
```

Тут цикл виконується n разів, а операція print займає константний час. Отже, загальна складність цього фрагмента коду становить O($n$).

Для вкладених циклів їхні складності перемножуються. Якщо зовнішній цикл виконується $n$ разів, а внутрішній для кожної ітерації зовнішнього виконується $m$ разів, то загальна складність буде O($n \cdot m$).

```
// вкладені цикли
for i from 1 to n:
  for j from 1 to n:
    // Дія, що займає O(1) часу
    print(i, j)
```

Тут обидва цикли виконуються по $n$ разів. Внутрішній цикл виконується $n$ разів для кожної з $n$ ітерацій зовнішнього циклу. Загальна кількість операцій *print* дорівнює $n \times n = n^2$. Отже, складність становить O($n^2$).

Загальні правила аналізу:

1. Послідовні блоки. Якщо алгоритм складається з кількох послідовних блоків коду, його загальна складність визначається складністю «найважчого» блоку (правило сум: $O(f(n)) + O(g(n)) = O(max(f(n),g(n)))$).

2. Вкладені блоки. Для вкладених структур (цикли або рекурсивні виклики) їхні складності перемножуються (правило добутків).

3. Спрощення. У кінцевому виразі складності відкидаються константні множники та члени нижчого порядку. Наприклад, функція $T(n) = 3n^2 + 10n + 5$ спрощується до $O(n^2)$.

## 1.8. Аналіз базових алгоритмів пошуку та сортування

Застосуємо розглянуті принципи для аналізу чотирьох фундаментальних алгоритмів.

Лінійний пошук (Linear Search). Це найпростіший алгоритм пошуку, який послідовно перевіряє кожен елемент масиву, доки не знайде шуканий елемент або не дійде до кінця масиву. Він не вимагає, щоб масив був відсортованим.

Приклад. Пошук числа 20 у масиві.

1. 10!= 20
2. 50!= 20
3. 30!= 20
4. 70!= 20
5. 20 == 20. Елемент знайдено на індексі 4.

Реалізації

```
Псевдокод:
function LinearSearch(array, target)
  for i from 0 to length(array) - 1
    if array[i] == target
      return i  // Повернути індекс знайденого елемента
  return -1     // Елемент не знайдено
```

```
Python
def linear_search(arr, target):
    """
    Виконує лінійний пошук елемента target у масиві arr.
    Повертає індекс елемента, якщо знайдено, інакше -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

```
C++
#include <vector>

int linearSearch(const std::vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) {
            return i; // Повернути індекс знайденого елемента
        }
    }
    return -1; // Елемент не знайдено
}
```

Часова складність:

1. Найкращий випадок $\Omega(1)$. Елемент міститься на першій позиції. Потрібна лише одна перевірка.

2. Найгірший випадок $O(n)$. Елемент міститься на останній позиції або відсутній у масиві. Потрібно $n$ перевірок.

3. Середній випадок $\Theta(n)$. У середньому потрібно перевірити $n/2$ елементів.

Просторова складність O(1). Алгоритм використовує лише кілька змінних для індексу та порівняння, незалежно від розміру масиву.

Бінарний пошук (Binary Search). Високоефективний алгоритм пошуку, який працює виключно на відсортованих масивах. Він працює за принципом «розділяй та володарюй» – порівнює шуканий елемент із середнім елементом масиву. Якщо вони рівні, пошук завершено. Якщо шуканий елемент менший, пошук продовжується в лівій половині масиву; якщо більший – у правій. На кожному кроці область пошуку зменшується вдвічі.

*Приклад.* Пошук числа 23 у відсортованому масиві.

1. Low = 0, high = 7. mid = 3. array = 15. 23 > 15, шукаємо у правій половині.

2. Low = 4, high = 7. mid = 5. array = 29  23 < 29, шукаємо у лівій половині нового діапазону.

3. Low = 4, high =4. mid = 4. array = 23. 23 == 23. Елемент знайдено на індексі 4.

Реалізації.

*Псевдокод (ітеративний):*

```
function BinarySearch(array, target)
  low = 0
  high = length(array) - 1

  while low <= high
    mid = floor((low + high) / 2)
    if array[mid] == target
      return mid
    else if array[mid] < target
      low = mid + 1
    else
      high = mid - 1

  return -1
```

```
Python
def binary_search(arr, target):
    """
    Виконує бінарний пошук елемента target у відсортованому масиві arr.
    Повертає індекс елемента, якщо знайдено, інакше -1.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

```
C++
#include <vector>

int binarySearch(const std::vector<int>& arr, int target) {
    int low = 0;
    int high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2; // Щоб уникнути переповнення для великих low, high

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

Аналіз часової складності. Найкращий випадок Ω(1). Елемент міститься посередині масиву. Найгірший випадок O(*logn*). На кожному кроці розмір масиву для пошуку ділиться на 2. Кількість таких поділів, доки не залишиться один елемент, є логарифмом за основою 2 від *n*. Середній випадок Θ(*logn*).

Просторова складність O(1) для ітеративної версії. Для рекурсивної версії складність буде O(*logn*) через глибину стеку рекурсивних викликів.

Таблиця 1.3 – Порівняльна таблиця складності алгоритмів пошуку та сортування

| Алгоритм | Часова складність (найкращий) | Часова складність (середній) | Часова складність (найгірший) | Просторова складність | Примітки |
|---|---|---|---|---|---|
| Лінійний пошук | Ω(1) | Θ(n) | O(n) | O(1) | Не потребує відсортованих даних |
| Бінарний пошук | Ω(1) | Θ(logn) | O(logn) | O(1) | Вимагає відсортованих даних |
| Сортування бульбашкою | Ω(n) | $\Theta(n^2)$ | $O(n^2)$ | O(1) | Неефективний, але простий. Стабільний. |
| Сортування вставками | Ω(n) | $\Theta(n^2)$ | $O(n^2)$ | O(1) | Ефективний для малих або майже відсортованих масивів. Стабільний. |

## 1.9. Приклади розв'язку задач

**Задача 1.** Визначити асимптотичну складність *T(n)* для фрагмента коду:

```
for i from 1 to n:
  for j from i to n:
    // operation that takes O(1)
```

*Розв'язок*

1. Зовнішній цикл for i виконується n разів.
2. Внутрішній цикл for j залежить від i. Проаналізуємо кількість його ітерацій:
   – Коли i = 1, j пробігає від 1 до n (n ітерацій).
   – Коли i = 2, j пробігає від 2 до n (n − 1 ітерацій).
   – ...
   – Коли i = n, j пробігає від n до n (1 ітерація).
3. Загальна кількість операцій – це сума арифметичної прогресії:

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

4. Розкриваємо дужки: $T(n) = 21n^2 + 21n$.
5. Згідно з правилами асимптотичного аналізу, ми відкидаємо константні множники (21) та члени нижчого порядку (21n).

Відповідь: Асимптотична складність становить $\Theta(n^2)$.

**Задача 2.** Довести, що $5n^2 + 100n + 20 = \Theta(n^2)$.

*Розв'язок*

Згідно з визначенням Θ-нотації, нам потрібно знайти такі додатні константи c1, c2, $n_0$, щоб для всіх $n \geq n_0$ виконувалась нерівність $c1n^2 \leq 5n^2 + 100n + 20 \leq c2n^2$.

1. Пошук верхньої межі (c2):

$5n^2 + 100n + 20 \leq 5n^2 + 100n^2 + 20n^2$ (для $n \geq 1$, оскільки $n \leq n^2 i\ 1 \leq n^2$);

$5n^2 + 100n + 20 \leq 125n^2$.

Отже, можемо взяти $c2 = 125$ та $n_0 = 1$. Нерівність виконується.

2. Пошук нижньої межі (c1):

$5n^2 + 100n + 20 \geq 5n^2$ (оскільки $100n + 20 > 0$ для $n \geq 1$).

Отже, можемо взяти c1 = 5. Ця нерівність також виконується для $n_0 = 1$.

Висновок. Ми знайшли $c1 = 5, c2 = 125, n_0 = 1$, для яких нерівність $5n^2 \leq 5n^2 + 100n + 20 \leq 125n^2$ є істинною для всіх $n \geq 1$. Отже, доведено, що $5n^2 + 100n + 20 = \Theta(n^2)$.

**Задача 3.** Проаналізувати часову та просторову складність алгоритму пошуку одночасно мінімального та максимального елементів у масиві за один прохід.

*Розв'язок*

Алгоритм:

1. Ініціалізувати min_val та max_val першим елементом масиву.

2. Пройти по решті елементів масиву (від другого до останнього).

3. Для кожного елемента порівняти його з min_val та max_val та оновити їх за потреби.

Часова складність: Цикл виконується n−1 разів. Усередині циклу виконуються дві операції порівняння. Загальна кількість порівнянь у найгіршому випадку становить $2(n - 1)$. Це лінійна функція від n. Отже, часова складність дорівнює $\Theta(n)$.

Просторова складність: Алгоритм використовує лише дві додаткові змінні (min_val, max_val) для зберігання поточних мінімуму та максимуму. Ця кількість не залежить від розміру масиву n. Отже, допоміжна просторова складність дорівнює O(1).

## Контрольні питання

1. Дайте формальне визначення алгоритму та перелічіть його п'ять основних властивостей.

2. Поясніть, чому властивість «скінченність» є критично важливою для алгоритму, і наведіть приклад послідовності дій, яка не є алгоритмом через порушення цієї властивості.

3. Що таке модель обчислень RAM-машина і які основні припущення вона робить для спрощення аналізу?

4. У чому полягає ключова відмінність між моделлю Word RAM та стандартною RAM-машиною?

5. Поясніть різницю між часовою та просторовою складністю. Що таке допоміжний простір (auxiliary space)?

6. Дайте формальне визначення нотації O-велике (Big O). Що вона описує з точки зору межі зростання функції?

7. Дайте формальне визначення нотацій Ω та Θ. Як вони співвідносяться з O-великим?

8. Чи є твердження «Часова складність сортування вставками − O(n)» коректним? Поясніть свою відповідь, враховуючи різні випадки аналізу.

9. Поясніть різницю між аналізом найгіршого та середнього випадків. Чому аналіз найгіршого випадку використовується найчастіше в практиці?

10. Сформулюйте принцип «Алгоритми + Структури даних = Програми» і поясніть його своїми словами.

11. Яка асимптотична складність алгоритму з трьома вкладеними циклами, кожен з яких виконується n разів?

12. Яка основна перевага бінарного пошуку над лінійним і яка його основна вимога до вхідних даних?

13. Поясніть, чому найкращий випадок для оптимізованого сортування бульбашкою має складність Ω(n), а не Ω(1).

14. Чому сортування вставками вважається «адаптивним» алгоритмом?

15. Для яких типів задач доцільно використовувати модель зовнішньої пам’яті замість RAM-машини?

## Задачі для самостійного опрацювання

1. Визначте часову складність фрагмента коду:

```
for i = 1 to n:
    for j = 1 to i*i:...
// O(1) operation.
```

2. Порівняйте функції nlogn та $n^{1.5}$ асимптотично. Яка з них зростає швидше? Доведіть свою відповідь.

3. Напишіть псевдокод для алгоритму, який перевіряє, чи є рядок паліндромом, та проаналізуйте його часову і просторову складність.

4. Алгоритм А має складність $T_A(n) = 1000n^2$, а алгоритм В − $T_B(n) = 2^n$. Знайдіть (приблизно) таке значення n, після якого алгоритм А завжди буде ефективнішим за алгоритм В.

5. Проаналізуйте складність наївного алгоритму множення двох матриць розміром $n \times n$.

6. Доведіть або спростуйте: $n! = O(n^n)$.

7. Доведіть або спростуйте: $\log(n!) = \Theta(n\log n)$.

8. Запропонуйте алгоритм для розміщення другого за величиною елемента в масиві та оцініть його складність у найгіршому випадку.

9. Проаналізуйте найкращий, середній та найгірший випадок для алгоритму, що знаходить перше входження елемента x у масиві, але починає пошук з кінця масиву.

10. Дано два відсортовані масиви розміром n та m. Напишіть алгоритм для їх злиття в один відсортований масив та проаналізуйте його часову і просторову складність.

11. Що зміниться у складності бінарного пошуку, якщо на кожному кроці ділити масив не на дві, а на три частини (тернарний пошук)?

12. Чи може алгоритм сортування, заснований на порівняннях, мати часову складність у найгіршому випадку кращу, ніж O(nlogn)? Поясніть чому.

13. Проаналізуйте просторову складність рекурсивної реалізації обчислення *n*-го числа Фібоначчі (без мемоізації[10]).

14. Алгоритм зі складністю $O(n^2)$ обробляє 104 елементи за 1 секунду. Скільки часу (приблизно) йому знадобиться для обробки 106 елементів?

15. Модифікуйте сортування бульбашкою так, щоб воно сортувало масив за спаданням. Як це вплине на його асимптотичну складність?

---

[10] Мемоізація – це метод оптимізації програм, за якого запам'ятовуються результати вже виконаних обчислень**,** щоб не рахувати їх повторно, якщо ті самі вхідні дані знову з'являються. Мемоізація = «запам'ятати результат, щоб не рахувати вдруге». Особливо корисна у повільних рекурсивних функціях.

# ТЕМА 2
# АНАЛІЗ РЕКУРСИВНИХ АЛГОРИТМІВ ТА МАТЕМАТИЧНІ ОСНОВИ

## 2.1. Формальне визначення рекурсії

*Рекурсія* (лат. *Recursio* – повернення) – це фундаментальний принцип, що описує об'єкти або процеси, які визначаються через самих себе, або об'єкти свого ж класу. Цей принцип самоподібності зустрічається в багатьох галузях: у математиці (визначення факторіала чи чисел Фібоначчі), у природі (структура сніжинок, суцвіття капусти-романеско, фрактальні форми берегових ліній), у мистецтві (ефект Дросте) та, що найважливіше для комп'ютерних наук, у програмуванні.

У контексті програмування, *рекурсія* – це метод розробки алгоритмів, за якого функція для розв'язання задачі викликає саму себе. Такий підхід дає змогу розбити складну проблему на простіші підзадачі, які є меншими за розміром, але ідентичними за структурою до вихідної задачі. Цей процес триває доти, доки підзадачі не стануть настільки простими, що їх можна розв'язати безпосередньо. Так, рекурсія є не просто синтаксичною особливістю мови програмування, а потужною парадигмою розв'язання задач, що лежить в основі стратегії «розділяй та володарюй» (*divide and conquer*).

Важливо розрізняти подвійну природу рекурсії. Вона використовується не лише для визначення процесів (алгоритмів), але й для визначення структур даних. *Рекурсивно визначена структура даних* – це структура, яка у своєму визначенні посилається на саму себе. Класичними прикладами є:

1. Зв'язний список – це або порожній список, або вузол, що містить дані та посилання на інший (менший) зв'язний список.

2. Дерево – це або порожня структура, або кореневий вузол, що містить дані та посилання на множину інших дерев (своїх піддерев).

Такий підхід до визначення даних природно веде до використання рекурсивних алгоритмів для їх обробки, наприклад, для обходу дерева чи пошуку елемента у зв'язному списку. Це підкреслює глибокий зв'язок між алгоритмами та структурами даних, який буде розглянуто далі в цьому розділі.

## 2.2. Структура рекурсивної функції: базовий випадок та рекурсивний крок

Будь-яка коректно реалізована рекурсивна функція повинна мати дві обов'язкові складові, які забезпечують її правильне та скінченне виконання.

1. Базовий випадок (base case). Це умова, за якої задача є настільки простою, що її можна розв'язати безпосередньо, не вдаючись до подальших рекурсивних викликів. Базовий випадок діє як умова завершення рекурсії. Без нього функція викликала б себе нескінченно, що призвело б до помилки переповнення стека викликів (*stack overflow*).

2. Рекурсивний крок (recursive step). Це частина функції, де вона викликає саму себе для розв'язання меншої за розміром підзадачі. Важливо, щоб кожен рекурсивний виклик змінював вхідні дані так, щоб вони поступово наближалися до базового випадку.

Розглянемо цю структуру на класичному прикладі обчислення факторіала числа n (позначається як n!), який визначається як добуток усіх цілих додатних чисел від 1 до n. Математично це можна записати рекурсивно:

$$n! = \begin{cases} 1, & \text{якщо } n = 0 \\ n \cdot (n-1)!\,, & \text{якщо } n > 0 \end{cases}$$

Тут $n = 0$ є базовим випадком, а $n \cdot (n-1)!$ – рекурсивним кроком.

Псевдокод:

```
FUNCTION Factorial(n)
   // Базовий випадок: факторіал 0 дорівнює 1
   IF n == 0 THEN
      RETURN 1
   // Рекурсивний крок: n помножити на факторіал (n-1)
   ELSE
      RETURN n * Factorial(n - 1)
   END IF
END FUNCTION
```

*Python*

```
# Функція для обчислення факторіала числа n
deffactorial(n: int) -> int:
"""
   Обчислює факторіал числа n за допомогою рекурсії.
   n! = n * (n-1)!
   """
# Базовий випадок: факторіал 0 дорівнює 1
if n == 0:
return1
# Рекурсивний крок: n * факторіал (n-1)
else:
return n * factorial(n - 1)

# Приклад виклику
print(f"5! = {factorial(5)}") # Виведе: 5! = 120
```

*C++*

```
#include<iostream>

// Функція для обчислення факторіала числа n
longlongfactorial(int n){
// Базовий випадок: факторіал 0 дорівнює 1
if (n == 0) {
return1;
   }
// Рекурсивний крок: n * факторіал (n-1)
else {
return n * factorial(n - 1);
   }
}

intmain(){
std::cout<<"5! = "<< factorial(5) <<std::endl; // Виведе: 5! = 120
return0;
}
```

## 2.3. Класичний приклад: числа Фібоначчі

Послідовність Фібоначчі є ще одним канонічним прикладом рекурсії, що часто використовується для демонстрації як її елегантності, так і потенційних недоліків. Послідовність починається з 0 та 1, а кожне наступне число є сумою двох попередніх: 0, 1, 1, 2, 3, 5, 8, 13,....

Математичне рекурсивне визначення для *n*-го числа Фібоначчі, *F(n),* виглядає так:

$$F(n) = \begin{cases} n & \text{якщо } n \leq 1 \\ F(n-1) + F(n-2) & \text{якщо } n > 1 \end{cases}$$

Тут маємо два базові випадки: *F(0) = 0* та *F(1) = 1*. Рекурсивний крок виражає F(n) через два попередні члени послідовності.

Однак наївна рекурсивна реалізація цього визначення є вкрай неефективною. Причина полягає у величезній кількості повторних обчислень одних і тих самих значень. Наприклад, для обчислення fib(5) буде викликано fib(4) та fib(3). Зі свого боку fib(4) викличе fib(3) та fib(2). Як наслідок, fib(3) буде обчислено двічі, fib(2) – тричі, і так далі. Це явище називається *проблемою підзадач, що перекриваються* (*overlapping subproblems*), і воно є ключовою ознакою того, що для розв’язання задачі можна застосувати більш ефективні методи, такі як динамічне програмування або мемоїзація. Часова складність такого наївного підходу є експоненційною, приблизно $O(1.618^n)$, що робить його непридатним для великих значень *n*.

```
Псевдокод:
FUNCTION Fibonacci(n)
   // Базові випадки F(0) = 0, F(1) = 1
   IF n <= 1 THEN
      RETURN n
   // Рекурсивний крок
   ELSE
      RETURN Fibonacci(n - 1) + Fibonacci(n - 2)
   END IF
END FUNCTION
```

```
Python
# Наївна рекурсивна функція для обчислення n-го числа Фібоначчі
deffibonacci_recursive(n: int) -> int:
"""
   Обчислює n-те число Фібоначчі за допомогою наївної рекурсії.
   """
# Базові випадки
if n <= 1:
return n
# Рекурсивний крок
else:
return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Приклад виклику
print(f"F(10) = {fibonacci_recursive(10)}") # Виведе: F(10) = 55
```

```
C++
#include<iostream>

// Наївна рекурсивна функція для обчислення n-го числа Фібоначчі
longlongfibonacci_recursive(int n){
// Базові випадки
if (n <= 1) {
return n;
   }
// Рекурсивний крок
else {
return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
   }
}

intmain(){
std::cout<<"F(10) = "<< fibonacci_recursive(10) <<std::endl; // Виведе:
F(10) = 55
return0;
}
```

## 2.4. Переваги та недоліки рекурсії

Вибір між рекурсивним та ітеративним (з використанням циклів) підходом до розв'язання задачі є важливим архітектурним рішенням, що впливає на читабельність, продуктивність та використання пам'яті. Кожен підхід має свої сильні та слабкі сторони.

Переваги рекурсії:

1. Стислість та читабельність. Для задач, що мають природну рекурсивну структуру (наприклад, обхід дерев, алгоритми «розділяй та володарюй», робота з фракталами), рекурсивний код часто є значно коротшим, елегантнішим та легшим для розуміння, оскільки він безпосередньо відображає математичне визначення задачі.

2. Простота реалізації. Рекурсія дає змогу уникнути складного ручного управління станом, яке може знадобитися в ітеративному рішенні (наприклад, використання явного стеку для імітації рекурсії під час обходу дерева).

Недоліки рекурсії:

1. Накладні витрати. Кожен виклик функції потребує додаткового часу та пам'яті для створення нового фрейму в стеку викликів (збереження адреси повернення, локальних змінних тощо). Для задач, які можна легко розв'язати ітеративно, рекурсія може бути повільнішою.

2. Ризик переповнення стека (Stack Overflow). Стек викликів має обмежений розмір. Якщо глибина рекурсії є занадто великою (наприклад, обчислення факторіала для дуже великого числа), це може призвести до вичерпання пам'яті стека і аварійного завершення програми. Це є основним обмеженням для застосування глибокої рекурсії.

3. Складність налагодження. Відстеження потоку виконання в глибокій рекурсії може бути складним, оскільки потрібно тримати в голові ланцюжок викликів.

Деякі мови програмування підтримують *оптимізацію хвостової рекурсії (tail-call optimization)*, яка дає змогу компілятору перетворити певний тип рекурсивних викликів (хвостові виклики) на ітерацію, усуваючи ризик переповнення стека. Однак не всі мови (наприклад, стандартні реалізації Python та C++) гарантують таку оптимізацію.

Таблиця 2.1 – Порівняння рекурсивного та ітеративного підходів

| Характеристика | Рекурсивний підхід | Ітеративний підхід |
|---|---|---|
| **Читабельність коду** | Часто вища для задач з природною рекурсивною структурою (дерева, фрактали) | Може бути складнішою для вкладених логічних конструкцій, але простішою для лінійних задач |
| **Швидкість виконання** | Може бути повільнішою через накладні витрати на виклики функцій | Зазвичай швидша через відсутність накладних витрат на виклики функцій |
| **Використання пам'яті** | Використовує стек викликів, ризик переповнення стека (*Stack Overflow*) за глибокої рекурсії | Використовує константну кількість пам’яті (для змінних циклу), не залежить від «глибини» |
| **Складність реалізації** | Елегантна та проста для рекурсивних задач | Може вимагати складнішого управління станом та додаткових структур даних (наприклад, явного стека) |
| **Приклади задач** | Обхід дерев, швидке сортування, числа Фібоначчі | Обробка елементів масиву, лінійний пошук, обчислення суми |

## 2.5. Поняття рекурентного співвідношення

Для того щоб формально проаналізувати ефективність (часову або просторову складність) рекурсивного алгоритму, потрібно перевести його структуру на мову математики. Інструментом для такого перекладу є рекурентне співвідношення (або рекурентне рівняння).

*Рекурентне співвідношення* – це рівняння або нерівність, що визначає функцію через її власні значення для менших аргументів. В аналізі алгоритмів функція, яку ми аналізуємо, це зазвичай $T(n)$ – час роботи алгоритму для вхідних даних розміром $n$. Співвідношення описує $T(n)$ через час, потрібний для розв’язання підзадач меншого розміру.

Так, рекурентне співвідношення є математичною моделлю виконання рекурсивного алгоритму. Воно слугує мостом між програмним кодом та його асимптотичною складністю. Процес аналізу виглядає так:

1. Написання рекурсивного алгоритму для розв'язання задачі.

2. Побудова рекурентного співвідношення, що моделює часову складність цього алгоритму.

3. Розв'язання рекурентного співвідношення для отримання асимптотичної оцінки (наприклад, у нотації O, Θ або Ω).

Зазвичай рекурентне співвідношення для алгоритму, що базується на принципі «розділяй та володарюй», має загальний вигляд:

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

де: $T(n)$ – час роботи для задачі розміром n;
$a$ – кількість рекурсивних викликів (підзадач);
$n/b$ – розмір кожної підзадачі (припускається, що всі підзадачі однакового розміру);
$f(n)$ – час, витрачений на роботу поза рекурсивними викликами на поточному кроці (розділення задачі та об'єднання результатів).

Під час аналізу зазвичай роблять два спрощення: по-перше, вважають, що $n$ є цілим числом; по-друге, для достатньо малих $n$ (базовий випадок) час роботи є константою, $T(n) = \Theta(1)$.

## 2.6. Побудова рекурентних співвідношень на прикладах

Розглянемо, як будуються рекурентні співвідношення для деяких відомих алгоритмів.

*Приклад 1.* Двійковий пошук (Binary Search)

Алгоритм працює на відсортованому масиві. На кожному кроці він порівнює шуканий елемент із середнім елементом масиву.

Якщо вони не збігаються, алгоритм відкидає половину масиву, в якій елемента точно немає, і рекурсивно продовжує пошук у половині, що залишилася.

Побудова співвідношення:

- задача розміром n зводиться до однієї підзадачі ($a = 1$);
- розмір підзадачі становить $n/2$ ($b = 2$);
- робота на поточному кроці (обчислення індексу середини та одне порівняння) займає константний час, тобто $f(n) = \theta(1)$.

Рекурентне співвідношення: $T(n) = T(n/2) + \Theta(1)$.

*Приклад 2.* Сортування злиттям (Merge Sort)

Алгоритм ділить масив на дві рівні частини, рекурсивно сортує кожну з них, а потім зливає дві відсортовані половини в один відсортований масив.

Побудова співвідношення:

- Задача розміром $n$ зводиться до двох підзадач ($a = 2$).
- Розмір кожної підзадачі − $n/2(b = 2)$.
- Робота на поточному кроці − це процедура злиття двох відсортованих масивів загальним розміром n. Ця процедура вимагає лінійного часу, отже, $f(n) = \Theta(n)$.

Рекурентне співвідношення: $T(n) = 2T(n/2) + \Theta(n)$.

*Приклад 3.* Наївне рекурсивне обчислення чисел Фібоначчі

Для обчислення $F(n)$ функція рекурсивно викликає себе для $F(n - 1)$ та $F(n - 2)$.

Побудова співвідношення:

- Задача розміром $n$ зводиться до двох підзадач.
- Розміри підзадач – $n - 1$ та $n - 2$. Це приклад рекурентних співвідношень, де розмір зменшується на константу, а не ділиться.
- Робота на поточному кроці – одне додавання, тобто $f(n) = \Theta(1)$.

Рекурентне співвідношення: $T(n) = T(n - 1) + T(n - 2) + \Theta(1)$.

## 2.7. Метод підстановки (Substitution Method)

Після побудови рекурентного співвідношення наступним кроком є його розв'язання, тобто знаходження замкненої форми для T(*n*), яка не містить рекурсивних членів. Існують декілька стандартних методів для цього.

Метод підстановки є потужним, але часто нетривіальним у застосуванні. Він складається з двох етапів:

1. Висунути припущення щодо асимптотичної форми розв'язку (наприклад, T(n) = O(nlogn)). Зробити правильне припущення − це найскладніша частина, яка вимагає досвіду, інтуїції або використання інших методів, як-от дерево рекурсії.

2. Довести припущення за допомогою методу математичної індукції. На цьому етапі припущена форма розв'язку підставляється в рекурентне співвідношення, щоб показати, що вона справджується за умови правильного вибору констант.

## 2.8. Метод дерева рекурсії (Recursion-Tree Method)

Метод дерева рекурсії − це інтуїтивно зрозумілий спосіб візуалізувати процес розгортання рекурсії та оцінити загальну вартість роботи алгоритму. Він особливо корисний для формування гіпотези, яку потім можна довести методом підстановки.

Процес складається з таких кроків:

1. Побудувати дерево, де кожен вузол представляє вартість роботи на певному рівні рекурсивного виклику.

2. Підсумувати вартість роботи на кожному рівні дерева.

3. Підсумувати вартість роботи на всіх рівнях, щоб отримати загальну вартість.

## 2.9. Основна теорема

Основна теорема, або «майстер-метод», є потужним інструментом, що надає готовий «рецепт» для розв'язання великого класу рекурентних співвідношень, які часто виникають при аналізі алгоритмів типу «розділяй та володарюй».

Теорема застосовується до рекурентних співвідношень виду:

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

де $a \geq 1$ та $b > 1$ є константами, а $f(n)$ – асимптотично додатна функція. У цьому рівнянні:

− $A$ – кількість підзадач, на які розбивається вихідна задача.

− $n/b$ – розмір кожної підзадачі.

− $f(n)$ – вартість роботи з розділення задачі та об'єднання результатів.

Таблиця 2.2 – Основна теорема (Master Theorem)

| Випадок | Умова порівняння f(n) та $n^{\log_b a}$ | Результат T(n) | Інтуїтивне пояснення |
|---|---|---|---|
| **Випадок 1** | $f(n) = O(n^{\log_b a - \varepsilon})$ для деякої константи $\varepsilon > 0$ | $\Theta(n^{\log_b a})$ | Функція $n^{\log_b a}$ зростає поліноміально швидше за f(n). Вартість роботи домінує на листках дерева рекурсії («bottom-heavy») |
| **Випадок 2** | $f(n) = \Theta(n^{\log_b a}\log^k n)$ для деякої константи $k \geq 0$ | $\Theta(n^{\log_b a}\log^{k+1} n)$ | Функції f(n) та $n^{\log_b a}$ зростають з однаковою швидкістю (з точністю до логарифмічного множника). Вартість роботи рівномірно розподілена по всіх рівнях дерева («balanced») |
| **Випадок 3** | $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для $\varepsilon > 0$, та виконується умова регулярності: $af(n/b) \leq cf(n)$ для деякої константи $c < 1$ та всіх достатньо великих n | $\Theta(f(n))$ | Функція f(n) зростає поліноміально швидше за $n^{\log_b a}$. Вартість роботи домінує в корені дерева рекурсії («top-heavy») |

Теорема порівнює швидкість росту функції f(n) з функцією $n\log_b a$. Ця функція являє собою асимптотичну кількість листків у дереві рекурсії і відображає, наскільки швидко зростає кількість підзадач на кожному рівні.

Залежно від результату порівняння *f(n)* та $n^{\log_b a}$, теорема виділяє три випадки, які визначають асимптотичний розв'язок для T(*n*).

Застосування Майстер-теореми

Приклад 1. $T(n) = 9T(n/3) + n$

1. Параметри: $a = 9, b = 3, f(n) = n$.
2. Обчислюємо $n^{\log_b a} = n^{\log_3 9} = n^2$.
3. Порівнюємо $f(n) = nзn^2$. Оскільки $n = O(n^{2-\varepsilon})$ для $\varepsilon = 1$, це випадок 1.
4. Результат: $T(n) = \Theta(n^2)$.

Приклад 2. $T(n) = T(2n/3) + 1$

1. Параметри: $a = 1, b = 3/2, f(n) = 1$.
2. Обчислюємо $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.
3. Порівнюємо $f(n) = 1$ з 1. Оскільки $f(n) = \Theta(1) = \Theta(n^0 \log^0 n)$, це випадок 2 з $k = 0$.
4. Результат: $T(n) = \Theta(n^0 \log^{0+1} n) = \Theta(\log n)$.

Приклад 3. $T(n) = 3T(n/4) + n\log n$

1. Параметри: $a = 3, b = 4, f(n) = n\log n$.
2. Обчислюємо $n^{\log_b a} = n^{\log_4 3} \approx n^{0.792}$.
3. Порівнюємо $f(n) = n\log nз\ n^{0.792}$. Очевидно, що $n\log n = \Omega(n^{0.792+\varepsilon})$ для, наприклад, $\varepsilon = 0.2$.
4. Перевіряємо умову регулярності: $af(n/b) = 3 \cdot (n/4)\log(n/4) = (3/4)n(\log n - \log 4)$. Ми повинні показати, що $(¾)n(\log n - \log 4) \le c \cdot n\log n$ для деякого $c < 1$. Це виконується для $c = 3/4$за достатньо великих n. Отже, це випадок 3.
5. Результат: $T(n) = \Theta(n\log n)$.

Основна теорема є дуже зручною, але вона не є універсальною. Існують «проміжки» між її випадками, коли її застосувати неможливо. Наприклад:

1. Неполіноміальна різниця. Якщо f(n) зростає швидше за $n^{\log_b a}$, але не поліноміально швидше. Приклад: $T(n) = 2T(n/2) + n/\log n$. Тут $n^{\log_b a} = n$, $af(n) = n/\log n$. $f(n)$ зростає повільніше за n, тому це не випадок 2 або 3. Але різниця не є поліноміальною, тому це і не випадок 1.

2. Невиконання умови регулярності. Для випадку 3 умова регулярності є обов'язковою. Існують функції, для яких вона не виконується.

3. Неконстантні параметри. Теорема вимагає, щоб a та b були константами. Співвідношення виду $T(n) = 2^n T(n/2) + n^n$ не можуть бути розв'язані за допомогою цієї теореми.

У таких випадках потрібно повертатися до більш фундаментальних методів, таких як метод підстановки або дерево рекурсії.

4. Розширені методи аналізу алгоритмів

Асимптотичний аналіз у найгіршому випадку є стандартом, але іноді він дає занадто песимістичну оцінку. Для більш тонкого аналізу продуктивності використовують амортизований та імовірнісний аналіз.

## 2.10. Амортизований аналіз

*Амортизований аналіз* – це метод оцінювання вартості послідовності операцій над структурою даних. Його мета – показати, що хоча окремі операції в послідовності можуть бути дуже «дорогими», середня вартість однієї операції залишається низькою. Важливо підкреслити, що це не аналіз середнього випадку, оскільки він не робить жодних припущень щодо ймовірності вхідних даних; він дає гарантовану середню вартість для будь-якої можливої послідовності операцій.

Приклад. Розглянемо динамічний масив (наприклад, std::vector в C++), який подвоює свій розмір, коли в ньому закінчується місце.

Операція push_back (додавання в кінець).

1. «Дешева» операція. Якщо в масиві є вільне місце, додавання елемента займає час O(1).

2. «Дорога» операція. Якщо масив заповнений, потрібно: 1) виділити новий масив удвічі більшого розміру; 2) скопіювати всі n елементів зі старого масиву в новий; 3) додати новий елемент. Вартість цієї операції становить O(n).

Аналіз у найгіршому випадку скаже, що push_back має складність O(n), що є правдою, але вводить в оману, оскільки ця ситуація виникає рідко.

Амортизований аналіз (сукупний метод). Розглянемо послідовність з N операцій push_back до порожнього масиву.

1. Загальна вартість усіх «дешевих» операцій становить $N \cdot O(1) = O(N)$.

2. «Дорогі» операції (реалокації) відбуваються, коли кількість елементів досягає степеня двійки: 1, 2, 4, 8,..., $2^k$, де $2^k < N$. Вартість копіювання на кожному з цих кроків становить 1, 2, 4, 8,...

3. Загальна вартість копіювання для N вставок буде сумою геометричної прогресії: $1 + 2 + 4 + \cdots + 2^{\lfloor \log_2(N-1) \rfloor} < \sum_{i=0}^{\log N} 2^i = 2^{\log N+1} - 1 < 2N$.

4. Отже, загальна вартість N операцій push_back складається з N (за самі вставки) + < 2N (за копіювання), що дорівнює < 3N. Так, загальна вартість становить O(N).

5. Амортизована вартість однієї операції push_back дорівнює $O(N)/N = O(1)$.

Існують також інші методи амортизованого аналізу, такі як обліковий метод (де кожна «дешева» операція «відкладає» трохи «кредиту» для оплати майбутніх «дорогих» операцій) та метод потенціалів (де «кредит» зберігається як «потенційна енергія» структури даних).

## 2.11. Імовірнісний аналіз

Імовірнісний аналіз використовується для оцінювання часу роботи алгоритму, який або використовує випадковість у своїй логіці (рандомізований алгоритм), або для оцінювання середнього часу роботи для всіх можливих вхідних даних.

Приклад (Швидке сортування (Quicksort)). Алгоритм швидкого сортування працює за принципом «розділяй та володарюй».

1. Вибрати опорний елемент (pivot) з масиву.

2. Перевпорядкувати масив так, щоб усі елементи, менші за опорний, містились ліворуч від нього, а всі більші – праворуч.

3. Рекурсивно застосувати алгоритм до двох підмасивів.

Продуктивність Quicksort критично залежить від вибору опорного елемента.

1. Найгірший випадок. Якщо опорним елементом щоразу обирається найменший або найбільший елемент (наприклад, під час сортування вже відсортованого масиву, якщо опорний – це завжди перший елемент), масив розбивається на підмасиви розміром 0 та $n - 1$. Рекурентне співвідношення буде $T(n) = T(n - 1) + \Theta(n)$, що дає часову складність $O(n^2)$.

2. Найкращий випадок. Якщо опорний елемент щоразу ділить масив навпіл, рекурентне співвідношення буде $T(n) = 2T(n/2) + \Theta(n)$, що дає O(nlogn).

Рандомізований Quicksort. Щоб уникнути найгіршого випадку на практиці, опорний елемент обирається випадковим чином. За такого підходу ймовірність постійно обирати погані опорні елементи досить мала. Імовірнісний аналіз показує, що очікуваний час роботи рандомізованого Quicksort становить O(nlogn) для будь-яких вхідних даних. На практиці цей алгоритм є одним з найшвидших для сортування загального призначення.

## 2.12. Моделі обчислень

Аналіз складності алгоритмів завжди базується на певній моделі обчислень – абстрактному уявленні про те, як працює комп'ютер, які операції є базовими та скільки вони «коштують». Вибір моделі впливає на результати аналізу. Розвиток моделей обчислень відображає еволюцію комп'ютерної архітектури: від простої моделі RAM, де основним обмежувальним чинником була швидкість процесора, до складних ієрархічних моделей, що враховують значну різницю у швидкості доступу до різних рівнів пам'яті.

*Word RAM (Random-Access Machine)* – це найпоширеніша модель для аналізу алгоритмів. Вона має такі ключові припущення:

1. Пам'ять представлена як великий масив «слів». Кожне слово має свій унікальний індекс (адресу).

2. Доступ до будь-якого слова в пам’яті (читання або запис) займає константний час, O(1).

3. Базові арифметичні (додавання, множення) та логічні операції над словами також займають час O(1).

4. Розмір слова (кількість бітів) є достатнім, щоб зберігати адресу будь-якої комірки пам’яті.

Ця модель добре відображає роботу сучасних процесорів з даними, що містяться в регістрах або кеші першого рівня, але ігнорує складну ієрархію пам’яті.

Модель зовнішньої пам’яті (External Memory Model). Ця модель була розроблена для аналізу алгоритмів, що працюють з величезними обсягами даних, які не вміщуються в основну пам’ять і зберігаються на повільних пристроях (наприклад, жорстких дисках). Вона враховує дворівневу ієрархію пам’яті: швидка внутрішня пам’ять (кеш) та повільна зовнішня пам’ять.

1. Параметри моделі:

− M – розмір внутрішньої пам’яті (кешу);

− B – розмір блоку, мінімальної одиниці даних, що передається між рівнями пам’яті.

2. Метрика складності. Основною метрикою є не кількість операцій процесора, а кількість операцій вводу-виводу (I/O), тобто кількість блоків, переміщених між зовнішньою та внутрішньою пам’яттю.

3. «Свідомість» алгоритму. Алгоритми, розроблені для цієї моделі, є «кеш-свідомими» (cache-aware), оскільки вони знають параметри M та B і можуть бути оптимізовані для їх конкретних значень.

*Кеш-несвідома модель* є узагальненням моделі зовнішньої пам’яті. Її мета – розробка алгоритмів, які є ефективними на будь-якій ієрархії пам’яті, не знаючи її параметрів (M та B).

Принцип роботи. Кеш-несвідомі алгоритми зазвичай використовують рекурсивну стратегію «розділяй та володарюй». Задача рекурсивно розбивається на менші підзадачі. Рано чи пізно розмір підзадачі стає настільки малим, що вона повністю вміщується в кеш будь-якого рівня. З цього моменту всі подальші операції з цими даними відбуваються швидко, без звернень до повільнішої пам’яті.

Перевага. Такий алгоритм є універсальним і не потребує ручного налаштування під конкретну апаратну конфігурацію, що робить його портативним та ефективним на різних машинах.

Таблиця 2.3 – Порівняльна таблиця моделей обчислень

| Модель | Структура пам’яті | Вартість операцій | Знання параметрів системи |
|---|---|---|---|
| **Word RAM** | Однорідний масив слів | O(1) для доступу до пам’яті та базових операцій | Немає параметрів ієрархії |
| **External Memory** | Дворівнева: кеш розміром M, зовнішня пам’ять. Передача даних блоками по B елементів | Аналізується кількість передач блоків (I/O) | Алгоритм знає M та B |
| **Cache-Oblivious** | Багаторівнева ієрархія (аналізується як дворівнева) | Аналізується кількість передач блоків (I/O) | Алгоритм не знає M та B |

## 2.13. Поняття структури даних та її роль

*Структура даних* – це спосіб організації, зберігання та управління даними в комп’ютері, який дає змогу ефективно виконувати над ними певні операції. Вибір правильної структури даних є критично важливим для створення ефективних програм. Наприклад, для швидкого пошуку даних можна використати збалансоване двійкове дерево пошуку, а для моделювання мережі доріг – граф.

Алгоритми та структури даних нерозривно пов’язані; вони утворюють симбіотичний зв’язок, який є основою програмування. Цей зв’язок можна описати відомим висловом Ніклауса Вірта: «Алгоритми + Структури даних = Програми».

Алгоритми визначають вибір структур даних.Тип задачі та логіка алгоритму диктують, яка структура даних буде найефективнішою. Наприклад, алгоритм Дейкстри для пошуку найкоротшого шляху в графі оптимально працює з використанням пріоритетної черги.

Структури даних впливають на ефективність алгоритмів. Ефективність алгоритму безпосередньо залежить від того, наскільки швидко обрана структура даних дає змогу виконувати потрібні операції (вставка, видалення, пошук). Використання хеш-таблиці може звести середній час пошуку до O(1), тоді як пошук у невпорядкованому масиві вимагатиме O(n).

Так, розробка ефективного програмного забезпечення – це процес знаходження оптимального поєднання алгоритму та структури даних для конкретної задачі.

Існує багато способів класифікувати алгоритми та структури даних. Нижче наведено узагальнену класифікацію, що підкреслює їх взаємозв'язок.

Класифікація структур даних:

1. Лінійні. Елементи організовані послідовно.

− *Масиви*: колекції елементів з доступом за індексом.

− *Списки*: зв'язані колекції елементів.

− *Стеки*: структура LIFO (Last-In, First-Out).

− *Черги*: структура FIFO (First-In, First-Out).

2. Ієрархічні. Елементи організовані у вигляді дерева.

− *Дерева*: загальна ієрархічна структура (двійкові дерева, B-дерева).

− *Купи (Heaps)*: деревоподібна структура для реалізації пріоритетних черг.

3. Мережеві. Елементи є вузлами та зв'язками між ними.

− *Графи*: множина вершин та ребер, що їх з'єднують.

4. Асоціативні. Зберігання даних у форматі «ключ-значення».

− *Хеш-таблиці (Словники)*: забезпечують швидкий доступ до даних за ключем.

− *Множини*: колекції унікальних елементів.

Класифікація парадигм алгоритмів:

1. Розділяй та володарюй. Розбиття задачі на менші підзадачі, їх рекурсивне розв'язання та об'єднання результатів.

2. Жадібні алгоритми. На кожному кроці робиться локально оптимальний вибір з надією, що він призведе до глобально оптимального розв'язку.

3. Динамічне програмування. Розв’язання задачі через розбиття її на підзадачі, що перекриваються, та збереження результатів для уникнення повторних обчислень.

4. Перебір з поверненням (Backtracking). Пошук розв’язку за рахунок систематичного перебору всіх можливих кандидатів, з відсіканням гілок пошуку, які гарантовано не ведуть до розв’язку.

Таблиця 2.4 – Класифікація та зв’язок алгоритмів і структур даних

| Парадигма алгоритму | Опис | Типові структури даних |
|---|---|---|
| **Розділяй та володарюй** | Рекурсивне розбиття задачі на менші незалежні підзадачі | Масиви, списки, дерева |
| **Жадібні алгоритми** | Прийняття локально оптимальних рішень на кожному кроці | Графи, пріоритетні черги (купи), множини |
| **Динамічне програмування** | Розв’язання підзадач, що перекриваються, та збереження їх результатів | Масиви, матриці (таблиці мемоїзації), хеш-таблиці |
| **Алгоритми пошуку на графах** | Систематичний обхід вершин та ребер графа (DFS, BFS) | Графи (списки суміжності, матриця суміжності), стеки, черги |
| **Перебір з поверненням** | Покрокова побудова кандидатів у розв’язки та відмова від них, якщо вони не відповідають умовам | Дерева (неявне дерево пошуку), стеки (неявний стек рекурсії) |

## 2.14. Приклади розв’язку задач

**Задача 1.** Розв’язати рекурентне співвідношення $T(n) = 4T(n/2) + n^2$ за допомогою дерева рекурсії та перевірити результат за допомогою Основної теореми.

*Розв’язок за допомогою дерева рекурсії*

1. Побудова дерева:

− Рівень 0 (корінь): вартість $n^2$. Має 4 нащадки.

− Рівень 1: 4 вузли, кожен з вартістю $(n/2)^2 = n^2/4$. Загальна вартість: $4 \cdot (n^2/4) = n^2$.

− Рівень 2: $4^2 = 16$ вузлів, кожен з вартістю $(n/4)^2 = n^2/16$. Загальна вартість: $16 \cdot (n^2/16) = n^2$.

− Рівень i: $4^i$ вузлів, кожен з вартістю $(n/2^i)^2 = n^2/4^i$. Загальна вартість: $4^i \cdot (n^2/4^i) = n^2$.

2. Аналіз глибини та загальної вартості:

− Глибина дерева $k = \log_2 n$. Кількість рівнів − $log_2 n + 1$.

− Вартість на кожному рівні однакова і дорівнює $n^2$.

− Загальна вартість = (вартість на рівень) × (кількість рівнів) = $n^2 \cdot (log_2 n + 1)$.

3. Висновок. Асимптотична складність $T(n) = \Theta(n^2 log n)$.

Перевірка за допомогою Основної теореми:

- Параметри: $a = 4, b = 2, f(n) = n^2$.
- Обчислюємо $n^{log_b a} = n^{log_2 4} = n^2$.
- Порівнюємо $f(n) = n^2 з n^2$. Оскільки $f(n) = \Theta(n^2) = \Theta(n^{log_b a} log^0 n)$, це випадок 2 з $k = 0$.
- Результат: $T(n) = \Theta(n^{log_b a} log^{0+1} n) = \Theta(n^2 log n)$.

Результати збігаються.

**Задача 2.** Написати рекурентну функцію для знаходження суми елементів у двійковому дереві. Проаналізувати її часову та просторову складність.

*Розв'язок*

Припустимо, що вузол дерева має структуру Node з полями value, left (лівий нащадок) та right (правий нащадок).

```
Псевдокод:
FUNCTION TreeSum(node)
  // Базовий випадок: якщо вузол порожній, сума дорівнює 0
  IF node IS NULL THEN
    RETURN 0
  // Рекурсивний крок: значення вузла + сума лівого піддерева + сума правого піддерева
  ELSE
    RETURN node.value + TreeSum(node.left) + TreeSum(node.right)
  END IF
END FUNCTION
```

```
Python
classNode:
def__init__(self, value, left=None, right=None):
    self.value = value
    self.left = left
    self.right = right

deftree_sum(node: Node) -> int:
if node isNone:
return0
else:
return node.value + tree_sum(node.left) + tree_sum(node.right)
```

```
C++
structNode {
int value;
   Node *left;
   Node *right;
};

longlongtreeSum(Node* node){
if (node == nullptr) {
return0;
   } else {
return node->value + treeSum(node->left) + treeSum(node->right);
   }
}
```

Часова складність. Функція відвідує кожен вузол дерева рівно один раз. Якщо в дереві N вузлів, то часова складність становить Θ(N).

Просторова складність. Просторова складність визначається максимальною глибиною стека рекурсивних викликів. У найгіршому випадку (вироджене дерево, що схоже на список) глибина може дорівнювати N. У найкращому випадку (збалансоване дерево) глибина дорівнює logN. Отже, просторова складність становить O(H), де H − висота дерева. У найгіршому випадку O(N), у середньому для збалансованого дерева O(logN).

**Задача 3.** Провести амортизований аналіз для стека з операцією multipop(k), яка видаляє k елементів.

*Розв'язок*

Розглянемо послідовність з n операцій push, pop та multipop над порожнім стеком.

Вартість операцій:

- push: O(1).
- pop: O(1).
- multipop(k): O(min(k,s)), де s – поточний розмір стека. У найгіршому випадку O(n).

Аналіз у найгіршому випадку дає O(n) для multipop, що є песимістичною оцінкою.

Амортизований аналіз (обліковий метод). Призначимо кожній операції амортизовану вартість.

1. push. Призначимо амортизовану вартість 2. Одну одиницю «витрачаємо» на саму операцію push, а другу «кладемо на депозит» разом з елементом у стеку.

2. pop та multipop. Призначимо амортизовану вартість 0. Вартість цих операцій буде «оплачена» з «депозиту», що зберігається з елементами. Коли елемент видаляється, ми використовуємо його збережену одиницю вартості.

Оскільки кожен елемент може бути видалений зі стека лише один раз (або через pop, або як частина multipop), «депозиту» завжди вистачить для оплати видалення. Загальний «кредит» ніколи не стане від'ємним.

Отже, амортизована вартість кожної операції становить:

- push – O(1) (вартість 2).
- Pop – O(1) (вартість 0).
- Multipop – O(1) (вартість 0).

Будь-яка послідовність з n операцій матиме загальну вартість O(n).

## Контрольні питання

1. Що таке базовий випадок у рекурсивній функції і яка його роль?

2. Поясніть, чому наївна рекурсивна реалізація чисел Фібоначчі є неефективною.

3. Сформулюйте рекурентне співвідношення для алгоритму, який ділить задачу на 5 підзадач розміром n/3 і виконує лінійну роботу для об’єднання результатів.

4. У чому полягає ключова відмінність між методом підстановки та методом дерева рекурсії?

5. Опишіть третій випадок Основної теореми своїми словами. Який тип розподілу роботи в алгоритмі він характеризує?

6. Наведіть приклад рекурентного співвідношення, до якого не можна застосувати Основну теорему, і поясніть чому.

7. У чому принципова різниця між амортизованим та імовірнісним аналізом?

8. Що таке «переповнення стека» і за яких умов воно може виникнути під час використання рекурсії?

9. Поясніть, що означають параметри a, b та f(n) у формулюванні Основної теореми.

10. Яка модель обчислень є найбільш реалістичною для сучасних комп’ютерів і чому?

11. Наведіть приклад, де вибір структури даних кардинально впливає на складність алгоритму.

12. Що таке умова регулярності в Основній теоремі і для якого випадку вона є обов’язковою?

13. Чому рандомізація у виборі опорного елемента покращує продуктивність алгоритму Quicksort?

14. Поясніть концепцію «кеш-несвідомого» алгоритму. У чому його перевага?

15. Наведіть приклад рекурсивно визначеної структури даних та рекурсивного алгоритму для роботи з нею.

## Задачі для самостійного опрацювання

1. Розв’яжіть рекурентне співвідношення $T(n) = 2T(n/3) + 1$ усіма трьома методами (дерево, підстановка, Основна теорема).

2. Напишіть рекурсивну функцію на Python/C++ для обчислення $x^n$ за час $O(\log n)$ (швидке піднесення до степеня). Складіть та розв’яжіть для неї рекурентне співвідношення.

3. Проаналізуйте складність алгоритму сортування Stooge Sort, що описується рекурентним співвідношенням $T(n) = 3T(2n/3) + O(1)$.

4. Реалізуйте двійковий пошук (рекурсивний варіант) на псевдокоді, Python та C++.

5. Розв’яжіть рекурентне співвідношення $T(n) = T(n) + 1$. (Підказка: зробіть заміну змінних).

6. Складіть рекурентне співвідношення для алгоритму пошуку найбільшого елемента в масиві, який ділить масив на дві половини, знаходить максимум у кожній і повертає більший з них. Розв’яжіть його.

7. Напишіть рекурсивну функцію для знаходження глибини двійкового дерева.

8. Розв’яжіть $T(n) = T(n – 1) + n$. Якому відомому алгоритму сортування відповідає така складність у найгіршому випадку?

9. Напишіть рекурсивну функцію, яка перевіряє, чи є рядок паліндромом.

10. Розв’яжіть $T(n) = 2T(n/2) + n/\log n$. Чи можна тут застосувати Основну теорему?

11. Реалізуйте алгоритм сортування злиттям (Merge Sort) на псевдокоді, Python та C++.

12. Проведіть амортизований аналіз для двійкового лічильника, який інкрементується n разів.

13. Розв’яжіть рекурентне співвідношення $T(n) = 2T(n – 1) + 1$ з $T(0) = 0$. (Підказка: це пов’язано з задачею про Ханойські вежі).

14. Поясніть, чому просторова складність рекурсивного сортування злиттям становить $O(n)$, а не $O(\log n)$, як у випадку стека викликів.

# ТЕМА 3
# ЛІНІЙНІ СТРУКТУРИ ДАНИХ ТА АБСТРАКТНІ ТИПИ

## 3.1. Поняття абстрактного типу даних

*Абстрактний тип даних* (АТД) – це ключове поняття в теорії алгоритмів та структур даних, що являє собою математичну модель, яка визначає набір даних та операцій, які можна над ними виконувати. Основна мета АТД полягає в абстрагуванні від деталей реалізації, зосереджуючись виключно на поведінці та семантиці операцій. Це означає, що АТД описує, *що* можна робити з даними (наприклад, додавати, видаляти, шукати), але не пояснює, *як* це відбувається на рівні пам'яті чи програмного коду.

Фундаментом АТД є принцип інкапсуляції[11], який передбачає приховування внутрішньої структури даних від зовнішнього користувача. Користувач взаємодіє з типом даних виключно через публічний інтерфейс – заздалегідь визначений набір функцій. Цей підхід дає змогу досягти високої модульності програмного забезпечення та знизити залежність між різними компонентами системи. Розробник, що працює з АТД, не повинен знати, як саме елементи зберігаються в пам'яті, йому достатньо розуміти лише поведінку цих типів.

АТД часто реалізуються в мовах програмування за допомогою інтерфейсів або шаблонних класів. Наприклад, у мові програмування C++ для реалізації узагальненого підходу застосовуються

---

[11] Інкапсуляція – це один із базових принципів об'єктно-орієнтованого програмування (ООП), який означає приховування внутрішнього стану об'єкта та деталей його реалізації від зовнішнього світу. Доступ до даних здійснюється лише через спеціально визначені методи (інтерфейс), а не безпосередньо.

Основна ідея:

1. Клас «загортає» (інкапсулює) дані та методи, які з ними працюють.
2. Користувач класу може працювати тільки з публічними методами (API), не знаючи деталей внутрішньої реалізації.

Переваги інкапсуляції:

1. Безпека – обмежує прямий доступ до внутрішніх даних.
2. Гнучкість – можна змінювати реалізацію без впливу на зовнішній код.
3. Зручність – створюється чіткий і зрозумілий інтерфейс для роботи з об'єктом.

шаблони класів, що дає змогу створювати контейнери, такі як стек або черга, незалежно від конкретного типу даних, які вони містять. Це демонструє, що концепція АТД не є суто теоретичною, а активно застосовується в сучасній розробці для створення гнучкого та підтримуваного коду.

Варто зауважити, що часто поняття АТД та структури даних використовуються як синоніми, однак між ними існує принципова різниця. Ця різниця є фундаментальною для розуміння того, як проєктувати ефективні та надійні програмні системи.

*Структура даних* – це конкретна, фізична реалізація АТД, яка визначає спосіб організації та зберігання даних у пам'яті комп'ютера. Вона описує, *як* дані розташовуються та як операції над ними виконуються. Наприклад, стек (АТД) може бути реалізований як масив або як зв'язний список. Обидві реалізації виконують одні й ті самі операції (push, pop, peek), але використовують різні внутрішні механізми.

Ключовим наслідком такого розмежування є можливість мати кілька альтернативних реалізацій для одного й того самого АТД. Розробники можуть покращувати внутрішні алгоритми конкретної структури даних, не змінюючи її зовнішнього інтерфейсу. Це дає змогу оптимізувати продуктивність, надійність та використання пам'яті з часом, не порушуючи сумісності з програмами, що використовують цей АТД.

## 3.2. Основні лінійні структури даних: стек, черга, дек

*Стек* – це лінійна структура даних, яка реалізує принцип LIFO (Last-In, First-Out), що означає «останній прийшов – перший вийшов». У стеку додавання та видалення елементів відбувається лише з одного кінця, який називається *вершиною* (top). Найпопулярніші (найбільш вживані) операції над стеком:

1. push(item) – додавання нового елемента на вершину стека.
2. pop() – видалення та повернення елемента з вершини стека. Ця операція модифікує стек.

3. peek() або top() – повернення елемента з вершини стека без його видалення.

4. isEmpty() – перевірка, чи стек порожній.

5. size() – повернення кількості елементів у стеку.

Аналогією стеку в повсякденному житті є стопка тарілок або купа листів на столі. Останній лист, покладений зверху, буде прочитаний першим. Цей принцип використовується в багатьох алгоритмах, наприклад, для перевірки збалансованості дужок, обходу дерева в глибину, а також для роботи стека викликів функцій у комп'ютерній пам'яті.

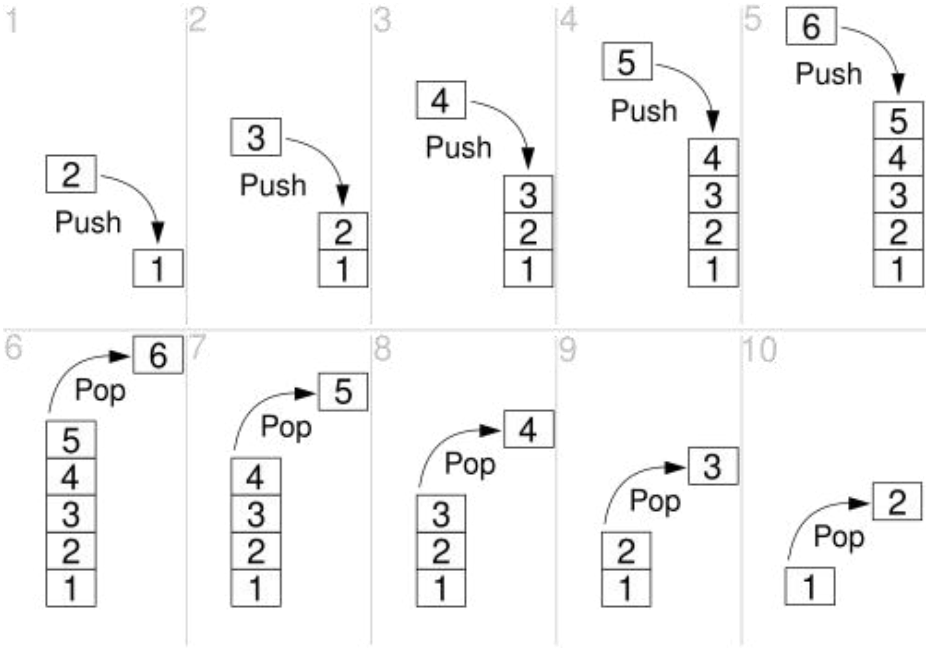



Рисунок 3.1 – Схема роботи стека (LIFO)

*Черга* – це лінійна структура даних, яка дотримується принципу *FIFO* (*First-In, First-Out*), тобто «перший прийшов – перший вийшов». На відміну від стека, додавання елементів відбувається в один кінець (який називають *хвостом* або rear), а видалення – з протилежного (який називають *головою* або front).

Основні операції над чергою:

1. enqueue(item) – додавання нового елемента у хвіст черги.

2. dequeue() – видалення та повернення елемента з голови черги.

3. front() – повернення елемента з голови черги без видалення.

4. isEmpty() – перевірка, чи черга порожня.
5. size() – повернення кількості елементів у черзі.

Класичним прикладом черги є черга в магазині: людина, яка прийшла першою, буде обслугована першою. Черги широко використовуються в програмуванні для моделювання потоків, обробки завдань у порядку їх надходження (наприклад, у планувальниках операційних систем) та в алгоритмах обходу графа в ширину.

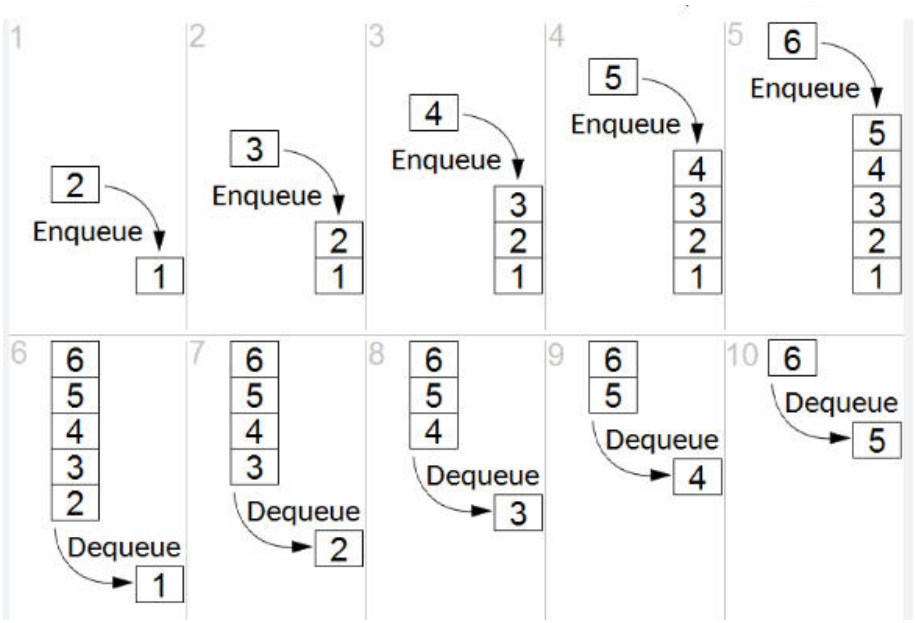



Рисунок 3.2 – Схема роботи черги (FIFO)

*Дек* (від англ. *Double-ended queue*) – це абстрактний тип даних, який є узагальненням як стека, так і черги. Він дає змогу додавати та видаляти елементи з обох кінців – як з голови (front), так і з хвоста (rear). Завдяки цій гнучкості дек може ефективно імітувати поведінку як стека, так і черги. Основні операції дека:

1. push_front(item) – додавання елемента на початок дека.
2. pop_front() – видалення елемента з початку дека.
3. push_back(item) – додавання елемента в кінець дека.
4. pop_back() – видалення елемента з кінця дека.

У Python для реалізації дека використовується оптимізована структура collections.deque, яка значно перевершує звичайний список list за продуктивністю для операцій додавання та видалення з початку, забезпечуючи сталу часову складність O(1). Це робить дек надзвичайно корисним інструментом, коли потрібна гнучкість у додаванні/видаленні елементів з обох кінців структури.

```
from collections import deque

d = deque()  # створення порожнього дека

# додавання елементів у кінець (праворуч)
d.append("A")
d.append("B")
print("Після додавання праворуч:", d)

# додавання елементів на початок (ліворуч)
d.appendleft("C")
d.appendleft("D")
print("Після додавання ліворуч:", d)

right = d.pop()  # видалення елементів із кінця
print("Видалено праворуч:", right)
print("Дек зараз:", d)

left = d.popleft()  # видалення елементів із початку
print("Видалено ліворуч:", left)
print("Дек зараз:", d)

# доступ до першого та останнього елемента
print("Перший елемент:", d[0])
print("Останній елемент:", d[-1])
```

## 3.3. Послідовні структури даних

У світі комп'ютерних наук структури даних є фундаментальними інструментами для організації, управління та зберігання даних у такий спосіб, що дає змогу ефективно виконувати над ними операції. Серед різноманіття структур даних особливе місце займають послідовні структури даних, ключовою характеристикою яких є зберігання елементів у неперервному (суміжному) блоці пам'яті. Цей принцип, відомий як *contiguous memory allocation*, є основою їхньої високої продуктивності та простоти.

Базовим і найбільш поширеним представником послідовних структур є *масив*. Масив – це не просто структура даних, це будівельний блок, на якому ґрунтуються багато інших, більш складних структур, таких як стеки, черги, хеш-таблиці й навіть деякі реалізації

дерев. Розуміння принципів роботи масивів, їхніх сильних та слабких сторін є абсолютно необхідним для будь-якого розробника програмного забезпечення.

*Статичний масив* – це лінійна структура даних, що являє собою колекцію елементів одного типу, які зберігаються в неперервному блоці пам'яті та мають фіксований розмір, визначений на етапі компіляції програми. «Статичний» у цьому контексті означає, що розмір масиву неможливо змінити під час виконання програми. Якщо масив створено для зберігання 10 елементів, він завжди матиме ємність рівно 10, неможливо додати 11-й елемент без створення нового, більшого масиву.

Спосіб виділення пам'яті для статичних масивів залежить від того, де вони оголошені в програмі.

Стекова алокація (Stack Allocation). Масиви, оголошені всередині функцій (локальні змінні), зазвичай розміщуються на стеку. Виділення пам'яті на стеку є надзвичайно швидкою операцією, оскільки зводиться до простого зміщення вказівника стека. Пам'ять, виділена на стеку, автоматично звільняється, коли виконання програми виходить за межі області видимості, в якій був оголошений масив (наприклад, під час завершення функції).

Алокація в сегменті даних (Data Segment Allocation). Глобальні масиви та масиви, оголошені з ключовим словом static, розміщуються в спеціальній області пам'яті, відомій як сегмент даних. Пам'ять для них виділяється один раз під час запуску програми та існує впродовж усього часу її виконання.

Ключова перевага масивів – це можливість доступу до будь-якого елемента за константний час, що позначається як O(1). Це стає можливим завдяки таким чинникам: елементи мають однаковий розмір, і вони зберігаються в пам'яті послідовно. Завдяки цьому адреса будь-якого елемента може бути обчислена математично.

Адреса *i*-го елемента масиву A розраховується за формулою:

$$\text{Address}(A[i]) = \text{BaseAddress} + i \times \text{sizeof}(\text{element\_type})$$

де:

- BaseAddress – це адреса першого елемента масиву (A).
- I – індекс елемента, до якого ми хочемо отримати доступ.
- sizeof(element_type) – розмір одного елемента масиву в байтах.

Наприклад, якщо масив цілих чисел (int, розміром 4 байти) починається за адресою 1000, то адреса елемента A буде 1000 + 3 × 4 = 1012. Оскільки ця операція складається лише з одного множення та одного додавання, вона виконується надзвичайно швидко, незалежно від розміру масиву.

На схемі нижче показано, як масив з 5 цілочисельних елементів (int arr) може бути розташований у пам’яті, якщо базова адреса дорівнює 2048, а розмір int − 4 байти.

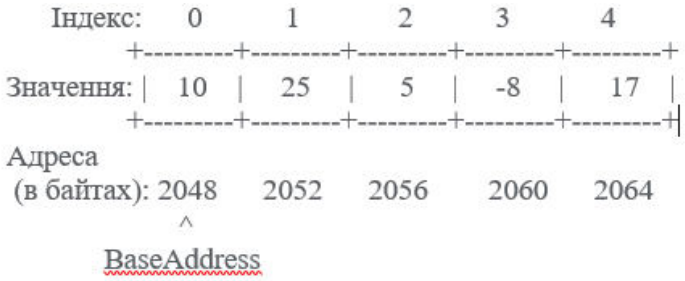



Рисунок 3.3 – Розташування в пам’яті цілочисельних елементів

Вибір статичного масиву є не просто вибором структури з фіксованим розміром; це свідомий вибір на користь максимальної продуктивності та передбачуваності. Це зумовлено глибоким зв’язком між архітектурою сучасних комп’ютерів та способом зберігання даних у статичних масивах.

1. Швидкість алокації. Як зазначалося вище, виділення пам’яті на стеку є миттєвою операцією, на відміну від виділення пам’яті в купі (heap), яке потребує складних алгоритмів для пошуку вільного блоку відповідного розміру.

2. Оптимізація на етапі компіляції. Оскільки розмір масиву відомий компілятору, він може генерувати високооптимізований машинний код. Адреси всіх елементів можуть бути обчислені статично відносно базової адреси, що усуває потребу в динамічних обчисленнях під час виконання.

3. Локальність посилань та кеш-ефективність. Це найважливіший аспект. Сучасні процесори мають багаторівневу систему кеш-пам’яті, яка є значно швидшою за основну оперативну пам’ять (RAM). Коли програма звертається до даних за певною адресою, процесор завантажує в кеш не лише ці дані, а й цілий блок сусідніх даних (так звану кеш-лінію). Оскільки елементи масиву зберігаються послідовно, під час доступу до arr[i], елементи arr[i + 1], arr[i + 2]тощо з високою ймовірністю вже будуть у

кеші. Це явище називається *просторовою локальністю посилань (spatial locality of reference)*. Як наслідок, послідовний обхід масиву (дуже поширена операція) виконується надзвичайно швидко, оскільки більшість звернень до пам'яті обслуговуються з надшвидкого кеша, а не з повільної RAM.

Ці чинники роблять статичні масиви ідеальним вибором для завдань, де продуктивність є критичною, наприклад, у системах реального часу, вбудованих системах, наукових обчисленнях та ігрових рушіях.

Ефективність статичного масиву сильно залежить від типу операції. Деякі операції є миттєвими, тоді як інші можуть бути дуже повільними.

1. Доступ за індексом (Access). Отримання значення елемента за відомим індексом (наприклад, arr) займає константний час O(1).

2. Модифікація за індексом (Update). Зміна значення елемента за відомим індексом займає O(1).

3. Обхід (Traversal). Послідовне відвідування кожного елемента масиву, наприклад, для виводу на екран або обчислення суми вимагає відвідування всіх n елементів, тому її складність є лінійною, O(n).

4. Пошук (Search):

− Лінійний пошук. У невпорядкованому масиві для знаходження елемента необхідно в найгіршому випадку перевірити всі n елементів. Тут складність становить O(n);

− Бінарний пошук. Якщо масив відсортований, можна використовувати значно ефективніший – бінарний пошук. Він працює за принципом «діли і володарюй», на кожному кроці відкидаючи половину елементів, що залишилися. Складність – O(*log*n).

5. Вставка (Insertion) – додавання нового елемента в масив. Оскільки масив має бути неперервним, для вставки елемента на позицію *i* потрібно зсунути всі елементи, починаючи з *i*, на одну позицію праворуч. У найгіршому випадку (вставка на початок, i = 0), потрібно зсунути всі n елементів. Складність становить O(n).

6. Видалення (Deletion) – видалення елемента з масиву. Аналогічно до вставки, для видалення елемента з позиції *i* потрібно зсунути всі наступні елементи на одну позицію ліворуч, щоб заповнити «прогалину». У найгіршому випадку (видалення першого елемента), потрібно зсунути n−1 елементів. Складність – O(n).

Таблиця 3.1 – Складність операцій над статичним масивом

| Операція | Опис | Часова складність (найгірший випадок) | Просторова складність |
|---|---|---|---|
| **Доступ** | Отримання елемента за індексом i | O(1) | O(1) |
| **Модифікація** | Зміна елемента за індексом i | O(1) | O(1) |
| **Пошук** | Пошук елемента v (лінійний) | O(n) | O(1) |
| **Пошук** | Пошук елемента v у відсортованому масиві | O(logn) | O(1) |
| **Вставка** | Вставка елемента на довільну позицію | O(n) | O(1) |
| **Видалення** | Видалення елемента з довільної позиції | O(n) | O(1) |
| **Обхід** | Відвідування всіх елементів | O(n) | O(1) |

Розглянемо, як оголошуються та використовуються статичні масиви різними мовами програмування.

```
Псевдокод
// Оголошення та ініціалізація масиву цілих чисел розміром 5
DECLARE numbers AS ARRAY OF INTEGER WITH SIZE 5
numbers = 10
numbers = 20
numbers = 30
numbers = 40
numbers = 50

// Доступ до елемента
value = numbers // value стане 30

// Модифікація елемента
numbers = 45

// Обхід масиву
FOR i FROM 0 TO 4
   PRINT numbers[i]
ENDFOR
```

У стандартній бібліотеці Python немає вбудованого типу «статичний масив»[12]. Вбудований тип list є динамічним. Однак поведінку статичного масиву можна симулювати, ініціалізувавши список фіксованого розміру і працюючи лише в його межах.

```
Python
# Симуляція статичного масиву розміром 5, заповненого None
static_array = [None] * 5

# Ініціалізація
static_array = 10
static_array = 20
static_array = 30
static_array = 40
static_array = 40
```

[12] У Python tuple часто плутають зі «статичним масивом», але це не зовсім те саме. Властивості tuple:

1. Незмінність (immutable): після створення кортежа не можна змінити його довжину чи значення елементів.
2. Фіксований розмір: кількість елементів визначається під час створення.
3. Різні типи: на відміну від класичних масивів, у tuple можна змішувати типи:
4. Призначення: кортежі більше використовуються для групування логічно пов'язаних даних, ніж для обчислень як масиви в C чи NumPy.

Порівняння зі «статичними масивами»: так само як у статичного масиву, довжина кортежа незмінна; але tuple не гарантує єдиного типу елементів і не має низькорівневих оптимізацій для швидких числових обчислень. Тобто можна сказати: tuple – це не статичний масив, а незмінна послідовність (immutable sequence). Для «справжніх» статичних масивів у Python краще використовувати numpy.ndarray.

| Характеристика | list (список) | tuple (кортеж) | numpy.ndarray (NumPy) |
|---|---|---|---|
| Змінність | ✔ Змінний (mutable) | ✘ Незмінний (immutable) | ✔ Змінний (mutable) |
| Довжина | Динамічна (можна змінювати) | Фіксована після створення | Фіксована (новий масив за зміни розміру) |
| Типи елементів | Будь-які, різні | Будь-які, різні | Зазвичай один тип (dtype) |
| Оптимізація для чисел | ✘ Немає | ✘ Немає | ✔ Так (ефективні обчислення, SIMD) |
| Призначення | Загальне зберігання даних | Групування незмінних даних | Числові обчислення, наукові задачі |
| Подібність до статичних масивів | ✘ Ні | ⚠ Частково (фіксований розмір, але не масив у класичному сенсі) | ✔ Так (найближчий аналог у Python) |

Висновок:

1. Якщо потрібна гнучкість → list.
2. Якщо потрібна незмінна структура даних → tuple.
3. Якщо потрібен аналог справжнього статичного масиву → numpy.ndarray.

```
# Доступ до елемента
value = static_array
print(f"Елемент за індексом 2: {value}")

# Модифікація елемента
static_array = 45
print(f"Модифікований масив: {static_array}")

# Обхід масиву
print("Елементи масиву:")
for item in static_array:
    print(item)
```

С та С++ надають пряму підтримку для статичних масивів.

```
C++
#include <iostream>
int main() {
    // Оголошення та ініціалізація масиву цілих чисел розміром 5
    int numbers = {10, 20, 30, 40, 50};
    // Доступ до елемента
    int value = numbers;
    std::cout << "Елемент за індексом 2: " << value << std::endl;

    // Модифікація елемента
    numbers = 45;
    // Обхід масиву
    std::cout << "Елементи масиву:" << std::endl;
    for (int i = 0; i < 5; ++i) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;
    // Визначення розміру масиву
    int size = sizeof(numbers) / sizeof(numbers);
    std::cout << "Розмір масиву: " << size << std::endl;

    return 0;
}
```

Статичні масиви мають чітко визначений набір характеристик, що робить їх ідеальними для одних завдань і абсолютно непридатними для інших.

Переваги:

1. Висока продуктивність – O(1) для доступу та модифікації є найшвидшим можливим.

2. Кеш-ефективність – послідовне зберігання даних максимізує використання процесорного кешу, що значно прискорює обхід.

3. Ефективне використання пам’яті – відсутні будь-які додаткові накладні витрати. Масив займає рівно стільки пам’яті, скільки потрібно для зберігання його елементів.

4. Простота та передбачуваність – легкі для розуміння та реалізації. Час виконання операцій є стабільним і передбачуваним.

Недоліки:

1. Фіксований розмір – основний недолік. Якщо реальна кількість даних виявиться більшою за розмір масиву, виникне помилка переповнення. Якщо меншою − буде марно витрачатися пам’ять.

2. Дорогі вставки та видалення – операції, що вимагають зсуву елементів, мають лінійну складність O(n), що робить їх дуже неефективними для великих масивів, у які часто вставляються або з яких видаляються елементи.

Статичні масиви є найкращим вибором, коли:

1. Кількість елементів відома наперед і не змінюється упродовж життя програми.

2. Продуктивність доступу є критично важливою.

3. Обмежені ресурси пам’яті не дозволяють додаткових накладних витрат.

Типові приклади використання охоплюють:

1. Вбудовані системи та мікроконтролери, де пам’ять дуже обмежена, а продуктивність є ключовою.

2. Обробку мультимедіа − буфери для аудіо- та відеоданих, де дані обробляються фіксованими блоками.

3. Ігрові рушії − для зберігання даних, розмір яких відомий (наприклад, матриці трансформацій, стан ігрових об’єктів).

4. Наукові та математичні обчислення − реалізація матриць та векторів для лінійної алгебри.

5. Низькорівневе програмування. Наприклад, під час реалізації драйверів або операційних систем.

Хоча статичні масиви є швидкими та ефективними, їхня головна вада − фіксований розмір − робить їх непрактичними для великої кількості завдань. У більшості реальних програм (вебсервери, текстові редактори, бази даних) кількість даних, які потрібно обробити, невідома на етапі написання коду і може динамічно змінюватися під час виконання. Спроба вгадати максимальний розмір наперед майже завжди призводить або до марної витрати пам’яті, або до неможливості обробити великі обсяги даних.

*Динамічний масив* (також відомий як масив, що росте, змінний масив або список-масив) − це структура даних, яка надає такий самий інтерфейс, як і звичайний масив (зокрема, швидкий доступ за індексом), але з можливістю змінювати свій розмір під час виконання програми. Він абстрагує від програміста ручне управління пам’яттю, автоматично розширюючись під час додавання нових елементів.

«Під капотом» динамічний масив − це обгортка над звичайним статичним масивом. Він керує цим внутрішнім масивом, замінюючи його на більший, коли це потрібно. Для цього динамічний масив відстежує два ключові параметри:

1. Логічний розмір (size) − кількість елементів, які наразі зберігаються в масиві. Це те, що бачить користувач структури.

2. Фізична ємність (capacity) − розмір внутрішнього статичного масиву, тобто максимальна кількість елементів, яку можна зберегти без необхідності виділення нової пам’яті. Завжди виконується умова *size <= capacity*.

На відміну від локальних статичних масивів, внутрішній масив для динамічної структури завжди виділяється в спеціальній

області пам’яті, що називається *купа*[13] *(heap)*. Купа − це великий пул пам’яті, доступний програмі для динамічного виділення блоків довільного розміру під час виконання. Управління пам’яттю в купі є складнішим і повільнішим, ніж на стеку, але саме воно надає потрібну гнучкість для зміни розміру структур даних.

Найцікавіший аспект динамічних масивів − це те, як вони керують своїм розміром. Коли користувач додає новий елемент, а масив вже заповнений (тобто size == capacity), відбувається процес розширення.

Процес розширення зазвичай складається з таких кроків:

1. Виділення нового блоку пам’яті. Створюється новий статичний масив у купі, розмір якого більший за поточний.

2. Копіювання даних. Усі елементи зі старого масиву копіюються в новий. Це найдорожча частина процесу, оскільки вона вимагає часу, пропорційного кількості елементів, тобто O(n).

3. Додавання нового елемента. Новий елемент додається в кінець нового масиву.

4. Звільнення старої пам’яті. Пам’ять, яку займав старий, менший масив, звільняється, щоб запобігти витокам пам’яті.

5. Оновлення вказівника. Внутрішній вказівник динамічного масиву тепер посилається на новий, більший блок пам’яті.

Розглянемо додавання елемента 50 до масиву A з size = 4 та capacity = 4.

Крок 1: Початковий стан

size = 4, capacity = 4

A -> [ 10 | 20 | 30 | 40 ]

---

[13] У програмуванні купа (англ. *heap*) може мати два різні значення залежно від контексту:

1. Купа як структура даних - це спеціалізоване дерево, зазвичай бінарне, яке задовольняє властивість купи: - у мін-купі кожен вузол має значення не більше від своїх нащадків (корінь – мінімум), - у макскупі – не менше (корінь -максимум).

Використовується для: реалізації черг з пріоритетами, алгоритму сортування купою (heapsort), ефективного пошуку найбільшого / найменшого елемента.

2. Купа як область пам’яті в системному програмуванні купа – це сфера оперативної пам’яті, що використовується для динамічного виділення (через malloc, new тощо).

Відміна від стека: стек працює за принципом LIFO і використовується для автоматичних змінних та викликів функцій; купа дає змогу виділяти пам’ять довільного розміру в будь-який момент і звільняти її вручну (або за допомогою *garbage collector*).

Крок 2: Спроба додати 50. Масив заповнений (size == capacity).

Крок 3: Виділення нового масиву B з подвоєною ємністю.

capacity нового масиву = 4 * 2 = 8

B -> [ | | | | | | | ]

Крок 4: Копіювання елементів з A в B.

B -> [ 10 | 20 | 30 | 40 | | | | ]

Крок 5: Додавання нового елемента 50 в B.

size стає 5

B -> [ 10 | 20 | 30 | 40 | 50 | | | ]

Крок 6: Звільнення пам’яті A та оновлення вказівника.

A (старий) звільняється.

Внутрішній вказівник тепер посилається на B.

size = 5, capacity = 8

На перший погляд, операція додавання елемента в кінець (push_back) виглядає неефективною. Її складність у найгіршому випадку становить O(n) через необхідність копіювання. Якби така вартість була в кожної операції, динамічні масиви були б занадто повільними. Однак дорогі операції перерозподілу трапляються рідко. Для аналізу таких ситуацій використовується амортизаційний аналіз. Його ідея полягає в тому, щоб розрахувати усереднену вартість операції[14] впродовж довгої послідовності операцій, гарантуючи продуктивність для *будь-якої* послідовності. Це відрізняється від аналізу середнього випадку, який робить припущення про ймовірнісний розподіл вхідних даних.

---

[14] Вартість операції в програмуванні – це умовний показник того, скільки ресурсів споживає виконання певної операції. Під «ресурсами» зазвичай мають на увазі:

1. Час виконання (скільки тактів процесора чи мілісекунд іде на виконання).
2. Пам’ять (скільки байтів потрібно зберігати чи переміщати).
3. Інші ресурси системи (виклики вводу / виводу, доступ до диска, мережі тощо).

У теорії алгоритмів «вартість операції» часто спрощують до кількості базових кроків, щоб можна було порівнювати алгоритми незалежно від конкретного комп’ютера.

Інші терміни, які вживають замість «вартість операції»:

1. Обчислювальна складність операції.
2. Ціна операції (рідше, як калька з англ. *cost*).
3. Ресурсомісткість операції.
4. Операційна складність.
5. Вартість (cost) у моделі обчислень.
6. У контексті Big-O ще кажуть: «часова складність» або «асимптотична вартість».

Ключовою для ефективності є стратегія розширення. Наївний підхід (збільшувати ємність на 1 за кожного переповнення) призвів би до того, що для додавання n елементів загальний час склав би 1 + 2 +...+ $n$ = O($n^2$), а амортизована вартість однієї операції − O($n$).

Ефективним підходом є стратегія геометричного розширення, зазвичай це подвоєння ємності.

Розглянемо вартість додавання n елементів до порожнього масиву, використовуючи агрегатний метод аналізу.

Нехай $Ci$ − вартість i-ї операції push_back. Якщо масив не заповнений, $Ci$ = 1 (проста вставка).Якщо масив заповнений (це відбувається, коли i − 1 є степенем двійки: 1, 2, 4, 8,...), то вартість складається з копіювання i − 1 елементів та вставки нового: $Ci$ = (i − 1) + 1 = i. Загальна вартість n операцій буде сумою вартостей всіх операцій:

$$\text{TotalCost(n)} = \sum_{i=1}^{n} Ci.$$

Ця сума складається з n простих вставок (вартість n) та вартостей копіювання під час переповнень. Переповнення відбуваються за розмірів 1, 2, 4,..., 2$k$, де 2$k$ < $n$.

$$\text{TotalCost(n)} = \underbrace{n}_{\text{базова вставка (1 за операцію)}} + \underbrace{\sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j}_{\text{копіювання при розширеннях}}.$$

Сума геометричної прогресії

$$\sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j = 2^{\lfloor \log_2 n \rfloor + 1} - 1 < 2n.$$

Оскільки 2$k$ < $n$, то 2$k$ + 1 < 2$n$. Отже, сума вартостей копіювання менша за 2n.

$$\text{TotalCost}(n) < n + 2n = 3n.$$

Загальна вартість n операцій є O($n$). Отже, амортизована вартість однієї операції push_back становить:

$$\text{AmortizedCost} = \text{TotalCost}(n)\ /n = O(n)\ /n = O(1).$$

Отже, незважаючи на те, що окремі операції можуть бути дорогими, в середньому по довгій послідовності кожна операція додавання в кінець є дуже дешевою, маючи константою амортизовану складність.

Цей результат є прикладом фундаментального інженерного компромісу. Розробники структур даних свідомо погоджуються на рідкісні, але передбачувані піки затримок (під час копіювання) в обмін на гарну загальну продуктивність. Кожна «дорога» операція вартістю O(n) ніби «купує» нам n«дешевих» операцій вартістю O(1) у майбутньому. Цей принцип проєктування ефективних систем, де аналізується не ізольована операція, а вся їхня послідовність, є ключовим не тільки для динамічних масивів, а й для інших просунутих структур, таких як хеш-таблиці. Саме завдяки цьому компромісу динамічні масиви (як-от std::vector, list) стали стандартною реалізацією списків у сучасних мовах програмування.

Враховуючи механізм зміни розміру, складність операцій для динамічного масиву така:

1. Доступ / модифікація за індексом − O(1). Оскільки внутрішньо використовується статичний масив, доступ залишається миттєвим.
2. Вставка в кінець (Append / push_back) − амортизована O(1).
3. Видалення з кінця (Pop / pop_back) − O(1). Ця операція ніколи не викликає перерозподілу пам’яті (хоча деякі реалізації можуть зменшувати ємність, якщо розмір стає значно меншим за неї), тому її вартість завжди константна.
4. Вставка / видалення в середині − O(n). Як і в статичному масиві, ці операції вимагають зсуву елементів, що робить їх лінійно залежними від кількості елементів після точки вставки / видалення.

Таблиця 3.2 – Складність операцій над динамічним масивом

| Операція | Опис | Часова складність (амортизована) | Просторова складність |
|---|---|---|---|
| **Доступ** | Отримання елемента за індексом i | O(1) | O(1) |
| **Модифікація** | Зміна елемента за індексом i | O(1) | O(1) |
| **Вставка в кінець** | Додавання елемента в кінець | O(1)* | O(1) |
| **Видалення з кінця** | Видалення останнього елемента | O(1) | O(1) |
| **Вставка в середину** | Вставка елемента на довільну позицію | O(n) | O(1) |
| **Видалення з середини** | Видалення елемента з довільної позиції | O(n) | O(1) |

** Найгірший випадок для однієї операції − O(n).*

Псевдокод (власний клас DynamicArray). Цей приклад ілюструє внутрішню логіку роботи динамічного масиву.

```
CLASS DynamicArray
  PRIVATE:
    data: POINTER TO INTEGER
    size: INTEGER
    capacity: INTEGER

  PRIVATE METHOD resize(new_capacity):
    new_data = ALLOCATE new_capacity OF INTEGER
    FOR i FROM 0 TO size - 1
      new_data[i] = data[i]
    ENDFOR
    DEALLOCATE data
    data = new_data
    capacity = new_capacity

  PUBLIC:
    CONSTRUCTOR():
      size = 0
      capacity = 0
      data = NULL
```

```
    DESTRUCTOR():
        DEALLOCATE data

    METHOD push_back(element):
        IF size == capacity THEN
            new_capacity = (capacity == 0)? 1 : capacity * 2
            resize(new_capacity)
        ENDIF
        data[size] = element
        size = size + 1

    METHOD get(index):
        IF index >= 0 AND index < size THEN
            RETURN data[index]
        ELSE
            ERROR "Index out of bounds"
        ENDIF
ENDCLASS
```

У Python тип list є вбудованим динамічним масивом, що надає багатий набір методів для роботи з ним. Важливою особливістю є те, що списки в Python є гетерогенними − вони можуть зберігати об’єкти різних типів, оскільки насправді зберігають не самі об’єкти, а посилання[15] на них.

[15] У програмуванні посилання – це спосіб звертання до об’єкта через його адресу в пам’яті, а не через сам об’єкт. Основна ідея:

1. Змінна може зберігати значення (наприклад, число 5) або посилання на місце в пам’яті, де лежить значення (наприклад, список [1, 2, 3]).
2. Якщо кілька змінних мають одне посилання, вони «вказують» на той самий об’єкт у пам’яті.

Важливі моменти:

1. Примітиви (int, float, bool, str, tuple) у Python поводяться як незмінні об’єкти, тому здається, що передаються «за значенням».
2. Змінні структури (list, dict, set, клас-об’єкти) передаються завжди через посилання.

Інші терміни для «посилання»:

1. Reference (англ.).
2. Вказівник (pointer) - у мовах на кшталт C/C++ (але там вказівник – більш низькорівневе поняття, ніж посилання в Python чи Java).
3. Handle (дескриптор, іноді) – у контексті системних ресурсів.

```
Python
# Створення порожнього списку (динамічного масиву)
dynamic_list =

# Додавання елементів в кінець (append)
dynamic_list.append(10)
dynamic_list.append(20)
dynamic_list.append("hello") # Можна додавати елементи різних типів
print(f"Список після append: {dynamic_list}")

# Видалення останнього елемента (pop)
last_element = dynamic_list.pop()
print(f"Видалений елемент: {last_element}, список: {dynamic_list}")

# Вставка елемента за індексом (insert)
dynamic_list.insert(1, 15) # Вставити 15 на позицію з індексом 1
print(f"Список після insert: {dynamic_list}")

# Видалення елемента за індексом (del)
del dynamic_list
print(f"Список після del: {dynamic_list}")

# Доступ та модифікація
print(f"Елемент за індексом 0: {dynamic_list}")
dynamic_list = 99
print(f"Модифікований список: {dynamic_list}")
```

У C++ стандартною реалізацією динамічного масиву є std::vector зі Standard Template Library (STL). На відміну від Python list, std::vector є гомогенним − він може зберігати елементи лише одного, наперед визначеного типу.

```
C++
#include <iostream>
#include <vector>

int main() {
   // Створення порожнього вектора цілих чисел
   std::vector<int> dynamic_vector;
```

```
        // Додавання елементів в кінець (push_back)
        dynamic_vector.push_back(10);
        dynamic_vector.push_back(20);
        dynamic_vector.push_back(30);

        std::cout << "Вектор: ";
        for (int val : dynamic_vector) {
            std::cout << val << " ";
        }
        std::cout << std::endl;
        std::cout << "Розмір: " << dynamic_vector.size() << ", Ємність: "
<< dynamic_vector.capacity() << std::endl;

        // Додамо ще елементи, щоб спровокувати зміну розміру
        dynamic_vector.push_back(40);
        dynamic_vector.push_back(50);
        std::cout << "Після додавання нових елементів:" << std::endl;
        std::cout << "Розмір: " << dynamic_vector.size() << ", Ємність: "
<< dynamic_vector.capacity() << std::endl;

        // Видалення останнього елемента (pop_back)
        dynamic_vector.pop_back();

        // Вставка елемента за індексом (insert)
        dynamic_vector.insert(dynamic_vector.begin() + 1, 15);

        // Доступ та модифікація
        // at() − з перевіркою меж, operator − без
        dynamic_vector.at(0) = 99;
        dynamic_vector = 25;

        std::cout << "Фінальний вектор: ";
        for (int val : dynamic_vector) {
            std::cout << val << " ";
        }
        std::cout << std::endl;

        return 0;
    }
```

## 3.4. Порівняльний аналіз статичних та динамічних масивів

Вибір між статичним та динамічним масивом є одним з базових рішень під час проєктування алгоритмів та програм. Кожен тип має свої чіткі переваги та недоліки, що робить їх оптимальними для різних сценаріїв. Наведена нижче таблиця узагальнює ключові відмінності.

Таблиця 3.3 – Статичні і динамічні масиви

| Характеристика | Статичний масив | Динамічний масив |
|---|---|---|
| Виділення пам’яті | На етапі компіляції | Під час виконання (run-time) |
| Розташування пам’яті | Стек (локальні) або сегмент даних (глобальні) | Купа (heap) |
| Розмір | Фіксований, незмінний | Змінний, може рости і зменшуватися |
| Гнучкість | Низька. Потребує знання розміру наперед | Висока. Адаптується до невідомої кількості даних |
| Продуктивність (доступ) | O(1) − дуже висока | O(1) − дуже висока |
| Продуктивність (вставка / видалення в кінці) | Неможливо, якщо масив заповнений | O(1) амортизовано |
| Продуктивність (вставка / видалення в середині) | O(n) | O(n) |
| Накладні витрати пам’яті | Мінімальні (лише дані) | Невеликі (дані + змінні для розміру / ємності) |
| Кеш-ефективність | Дуже висока | Висока, але може тимчасово знижуватися під час перерозподілу |
| Типові приклади в мовах | int arr; (C/C++) | std::vector (C++), list (Python), ArrayList (Java) |
| Ідеальні сценарії використання | Вбудовані системи, ігрові рушії, буфери, алгоритми, де розмір відомий | Загальновживані програми, обробка вводу користувача, вебдодатки |

## 3.5. Приклади розв'язку задач

**Задача 1.** Знайти другий за величиною елемент у масиві.

Це проста задача на обхід масиву. Наївним рішенням було б відсортувати масив і взяти передостанній елемент, але це мало б складність O(nlogn). Ефективніше рішення можна знайти за один прохід (O(n)), відстежуючи два найбільші елементи. Потрібно враховувати граничні випадки: масив з менше ніж двома елементами та наявність дублікатів.

Алгоритм:

1. Ініціалізувати дві змінні: largest та secondLargest найменшим можливим значенням (наприклад, мінус нескінченністю).
2. Пройтися по кожному елементу масиву.
3. Якщо поточний елемент current більший за largest:

a. secondLargest присвоїти значення largest.

b. largest присвоїти значення current.

4. Інакше, якщо current більший за secondLargest і не дорівнює largest: a. secondLargest присвоїти значення current.

5. Після завершення циклу secondLargest буде містити шукане значення.

Реалізація:

```
Псевдокод
FUNCTION findSecondLargest(array)
   IF array.size < 2 THEN
      RETURN "Недостатньо елементів"
   ENDIF

   largest = -INFINITY
   secondLargest = -INFINITY

   FOR EACH number IN array
      IF number > largest THEN
         secondLargest = largest
         largest = number
      ELSE IF number > secondLargest AND number!= largest THEN
         secondLargest = number
      ENDIF
   ENDFOR
```

```
    IF secondLargest == -INFINITY THEN
        RETURN "Другого найбільшого елемента не існує"
    ELSE
        RETURN secondLargest
    ENDIF
ENDFUNCTION
```

```
Python
def find_second_largest(nums):
    if len(nums) < 2:
        return None

    largest = float('-inf')
    second_largest = float('-inf')

    for num in nums:
        if num > largest:
            second_largest = largest
            largest = num
        elif num > second_largest and num!= largest:
            second_largest = num

    return second_largest if second_largest!= float('-inf') else None

# Приклад
print(find_second_largest()) # Виведе: 34
print(find_second_largest()) # Виведе: None
```

```
C++
#include <iostream>
#include <vector>
#include <limits>

int findSecondLargest(const std::vector<int>& nums) {
    if (nums.size() < 2) {
        return std::numeric_limits<int>::min(); // Позначення помилки
    }

    int largest = std::numeric_limits<int>::min();
    int secondLargest = std::numeric_limits<int>::min();
```

```
        for (int num : nums) {
            if (num > largest) {
                secondLargest = largest;
                largest = num;
            } else if (num > secondLargest && num!= largest) {
                secondLargest = num;
            }
        }
        return secondLargest;
    }

    int main() {
        std::vector<int> data = {12, 35, 1, 10, 34, 1};
        int result = findSecondLargest(data);
        if (result!= std::numeric_limits<int>::min()) {
            std::cout << "Другий найбільший елемент: " << result << std::endl;
        } else {
            std::cout << "Не вдалося знайти другий найбільший елемент."
<< std::endl;
        }
        return 0;
    }
```

**Задача 2.** Знайти підмасив з максимальною сумою (Алгоритм Кадане).

Це класична задача, що часто зустрічається на співбесідах. Потрібно знайти неперервний підмасив, сума елементів якого є максимальною. Наївне рішення з перебором усіх можливих підмасивів має складність $O(n^2)$. Алгоритм Кадане дає змогу розв'язати цю задачу за один прохід, тобто за O(n), використовуючи принципи динамічного програмування.

Ключова ідея алгоритму полягає в тому, що якщо сума підмасиву, що закінчується на поточній позиції, стає від'ємною, то цей підмасив не може бути початком оптимального підмасиву в майбутньому, тому ми його «відкидаємо» і починаємо новий підмасив з наступного елемента.

1. Ініціалізувати дві змінні: max_so_far (глобальний максимум) та max_ending_here (максимальна сума підмасиву, що закінчується на поточній позиції). Обидві ініціалізуються першим елементом масиву.

2. Пройтися по масиву, починаючи з другого елемента.

3. На кожному кроці оновити max_ending_here: max_ending_here = max(current_element, max_ending_here + current_element).

4. Оновити max_so_far: max_so_far = max(max_so_far, max_ending_here).

5. Після циклу max_so_far міститиме максимальну суму.

*Псевдокод*

```
FUNCTION maxSubarraySum(array)
  IF array is empty THEN
    RETURN 0
  ENDIF

  max_so_far = array
  max_ending_here = array

  FOR i FROM 1 TO array.size - 1
    current_element = array[i]
    max_ending_here = MAX(current_element, max_ending_here + current_element)
    max_so_far = MAX(max_so_far, max_ending_here)
  ENDFOR

  RETURN max_so_far
ENDFUNCTION
```

*Python*

```
def max_subarray_sum(nums):
  if not nums:
    return 0

  max_so_far = nums
  max_ending_here = nums

  for i in range(1, len(nums)):
    max_ending_here = max(nums[i], max_ending_here + nums[i])
    max_so_far = max(max_so_far, max_ending_here)
```

```
    return max_so_far

# Приклад
print(max_subarray_sum([-2, 1, -3, 4, -1, 2, 1, -5, 4])) # Виведе: 6 (для підмасиву [4, -1, 2, 1])
```

```
C++
#include <iostream>
#include <vector>
#include <algorithm>

int maxSubarraySum(const std::vector<int>& nums) {
    if (nums.empty()) {
        return 0;
    }

    int max_so_far = nums;
    int max_ending_here = nums;

    for (size_t i = 1; i < nums.size(); ++i) {
        max_ending_here = std::max(nums[i], max_ending_here + nums[i]);
        max_so_far = std::max(max_so_far, max_ending_here);
    }

    return max_so_far;
}

int main() {
    std::vector<int> data = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    std::cout << "Максимальна сума підмасиву: " << maxSubarraySum(data) << std::endl;
    return 0;
}
```

## 3.6. Зв’язні списки та інші структури

*Зв’язний список* є фундаментальною лінійною структурою даних, яка складається з послідовності елементів, що називаються *вузлами* (nodes). На відміну від масивів, елементи зв’язного списку необов’язково зберігаються в суміжних ділянках пам’яті. Замість цього зв’язок між вузлами забезпечується за допомогою *покажчиків*[16] або посилань. Кожен вузол зазвичай містить два поля: поле для зберігання даних та покажчик на наступний вузол у списку.

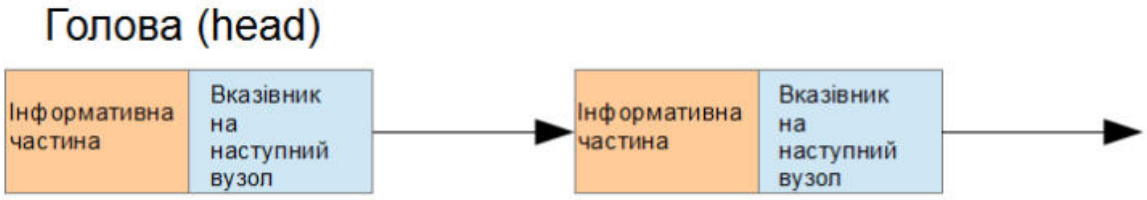



Рисунок 3.4 – Схема зв’язного списку

Ця архітектура надає зв’язним спискам значну перевагу над масивами − гнучкість у динамічному управлінні пам’яттю. Додавання або видалення елемента не вимагає зсуву всіх наступних

---

[16] Покажчик (вказівник) у програмуванні – це змінна, яка, замість того щоб зберігати безпосередньо дані (наприклад, число або символ), зберігає адресу пам’яті іншої змінної або об’єкта. По суті, покажчик – це «вказівник» на місце в оперативній пам’яті, де лежить потрібна вам інформація. Це дає змогу програмісту:

1. Отримувати прямий доступ до пам’яті: Це є важливим для низькорівневого програмування та оптимізації.
2. Динамічно виділяти пам’ять: За допомогою покажчиків можна виділяти та звільняти пам’ять під час виконання програми, що робить її більш гнучкою.
3. Ефективно передавати великі об’єкти: Замість копіювання всього об’єкта у функцію можна передати лише його адресу, що значно економить пам’ять та час.

Уявіть, що ваша комп’ютерна пам’ять – це велика шафа з безліччю пронумерованих комірок. Кожна комірка має свою унікальну адресу.

1. Звичайна змінна: якщо ви створюєте змінну int x = 10;, то в одній з комірок пам’яті зберігається число 10, а компілятор просто «запам’ятовує», що «x» – це назва для цієї комірки.
2. Змінна-покажчик: коли ви оголошуєте покажчик, наприклад int* p;, комп’ютер виділяє нову комірку пам’яті, але замість числа 10 туди записується адреса комірки, де лежить x. Тоді p «вказує» на x.

Операція розіменування (часто позначається символом *) дає змогу отримати доступ до даних, на які вказує покажчик. Наприклад, *p дасть вам значення 10, яке зберігається за адресою, на яку вказує p.

елементів, що є обов'язковим для масивів. Це робить операції вставки та видалення більш ефективними. Проте ця гнучкість має свою ціну: доступ до елемента за індексом у зв'язному списку є менш ефективним, ніж у масиві, оскільки для доступу до N-го елемента потрібно послідовно пройти N вузлів, починаючи з початку списку.

*Однозв'язний список* − це найпростіший тип зв'язного списку. Кожен його вузол містить два поля: одне для даних, а інше − покажчик (next), що вказує на наступний вузол у послідовності. Останній вузол списку має покажчик, що дорівнює NULL, що сигналізує про кінець структури.

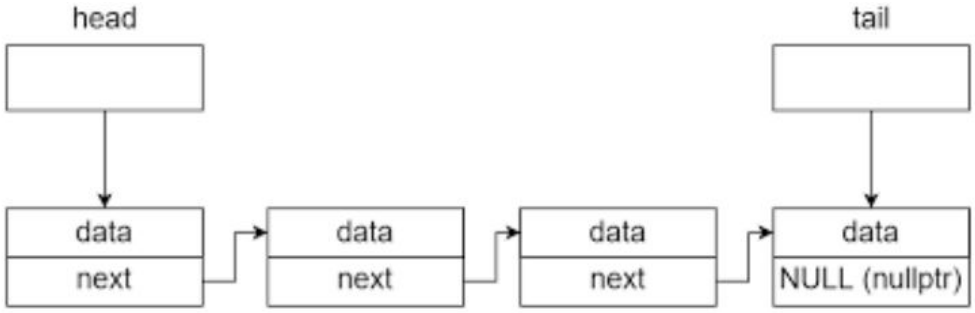



Рисунок 3.5 – Однозв'язний список

Робота з однозв'язним списком здійснюється через операції, які маніпулюють покажчиками.

Псевдокод, додавання елемента на початок:

1. Створити новий вузол newNode.
2. Встановити newNode.next = head.
3. Оновити покажчик голови списку: head = newNode.

Псевдокод − додавання елемента в кінець:

1. Створити новий вузол newNode.
2. Перевірити, чи список порожній. Якщо так, head = newNode.
3. Якщо ні, пройти по списку до останнього вузла, використовуючи тимчасовий покажчик current.
4. Встановити current.next = newNode.

Псевдокод, видалення елемента:

1. Знайти вузол, що передує тому, який потрібно видалити (prevNode).
2. Встановити prevNode.next = prevNode.next.next.

*Двобічнозв'язний* список є розширеною версією однозв'язного. Кожен вузол у цій структурі містить три поля: дані, покажчик на наступний вузол (next) та покажчик на попередній вузол (prev). Покажчик prev у першому вузлі та next в останньому вказують на NULL. Ця додаткова інформація дає змогу переміщуватися по списку в обох напрямах.

Основна перевага двобічнозв'язного списку полягає в ефективнішому виконанні деяких операцій, зокрема видалення. Маючи покажчик на вузол, який потрібно видалити, не потрібно обходити список, щоб знайти попередній елемент, оскільки покажчик на нього вже доступний через поле prev. Однак ця перевага супроводжується певними компромісами: кожен вузол потребує більше пам'яті (через додатковий покажчик prev), а операції додавання та видалення вимагають маніпуляцій з двома покажчиками замість одного, що збільшує обчислювальні витрати.

Крім класичних стека, черги та списків, вбудовані структури даних у Python, такі як кортежі, множини та словники, також відносяться до лінійних (з точки зору послідовного доступу) або хеш-орієнтованих структур.

1. Кортежі (tuple). Кортеж − це упорядкована, але *незмінна* послідовність об'єктів. На відміну від списків, їх елементи не можна змінювати, що робить їх безпечнішими для даних, які не повинні модифікуватися. Кортежі також займають менше пам'яті та працюють швидше за списки для операцій перебору. Завдяки своїй незмінності вони можуть використовуватися як ключі у словниках, що є неможливим для списків.

2. Множини (set). Множина − це неупорядкована колекція унікальних елементів. Вони реалізовані як хеш-таблиці, що забезпечує надзвичайно швидкі операції додавання, видалення та перевірки на входження. Середня часова складність цих операцій становить O(1).

3. Словники (dictionary). Словник є неупорядкованим набором пар «ключ-значення». Забезпечуючи доступ до значень за унікальними ключами, словники також реалізовані на базі хеш-таблиць. Це дає змогу виконувати операції додавання, видалення та пошуку за ключем із середньою часовою складністю O(1).

Важливо зазначити, що продуктивність цих операцій може знизитися до O(n) у найгіршому випадку, коли виникають колізії хешування.

Ці структури демонструють, як на рівні мови програмування інкапсулюються складні алгоритми та структури (наприклад, хешування), щоб надати розробнику простий та ефективний інструмент для вирішення типових завдань.

## 3.7. Оцінка складності операцій

Нотація Big O (О-велике) є стандартизованим математичним інструментом у комп'ютерній науці, що використовується для опису продуктивності алгоритму залежно від розміру вхідних даних n. Вона характеризує, скільки часу (кількість операцій) або пам'яті (просторова складність) знадобиться для виконання алгоритму, особливо в найгіршому випадку (worst-case scenario).

Основна мета нотації Big O − не вимірювання точного часу виконання, а оцінка порядку зростання складності. Це дає змогу порівнювати різні алгоритми та структури даних на предмет їх масштабованості, тобто як їх продуктивність змінюється зі збільшенням розміру вхідних даних.

Під час використання нотації Big O застосовуються два ключових принципи:

1. Ігнорування констант. Оскільки нотація описує асимптотичну поведінку, константні чинники відкидаються. Наприклад, алгоритм з $4n^2 + 2n$ операціями вважається $O(n^2)$.

2. Вибір домінантного терміна. З кількох термінів обирається той, що має найвищий порядок зростання. У прикладі $4n^2 + 2n$ домінантним є $n^2$, оскільки він зростає значно швидше за 2nза великих n.

Поширені типи складності охоплюють O(1) (постійна), O(log n) (логарифмічна), O(n) (лінійна), $O(n^2)$ (квадратична), O(n!) (факторіальна). Розуміння цих категорій є фундаментальним для вибору оптимальної структури даних та алгоритму для конкретного завдання.

Аналіз складності для стека і черги наочно демонструє, як вибір конкретної реалізації (структури даних) впливає на ефективність операцій, незважаючи на те, що вони відповідають одному й тому самому абстрактному типу.

Стек. Операції push, pop, peek завжди виконуються з вершиною стека. Це означає, що кількість операцій не залежить від кількості елементів у структурі. Так, як під час реалізації на масиві, так і на зв'язному списку, ці операції мають постійну часову складність O(1).

Черга:

1. Реалізація на масиві. Операція додавання елемента в кінець (enqueue) має складність O(1), але видалення елемента з початку (dequeue) є неефективним. Щоб видалити перший елемент, потрібно зсунути всі інші елементи в масиві, що вимагає n операцій. Отже, часова складність dequeue становить O(n). Для подолання цього недоліку використовується реалізація на базі циклічного масиву, яка дає змогу виконувати dequeue за O(1).

2. Реалізація на зв'язному списку. Завдяки тому, що вузли не є суміжними в пам'яті, додавання в кінець та видалення з початку (за умови наявності покажчиків на голову та хвіст) виконуються за постійний час O(1). Це робить реалізацію черги на зв'язаному списку набагато ефективнішою для частих операцій enqueue та dequeue, ніж реалізація на звичайному масиві.

Складність операцій для зв'язаних списків залежить від точки додавання, видалення або доступу до елементів.

Однозв'язний список:

1. Додавання на початок − O(1). Потрібно лише створити новий вузол та оновити покажчик голови.

2. Додавання в кінець − O(n). Потрібно пройти весь список від початку до кінця, щоб знайти останній вузол.

3. Видалення − O(n). Видалення елемента вимагає пошуку попереднього вузла, що знову ж таки передбачає обхід списку.

4. Пошук/Доступ − O(n). Доступ до елемента за індексом неможливий, тому потрібно послідовно обходити список від голови до необхідної позиції.

Двобічнозв’язний список:

1. Додавання / Видалення на початку та в кінці − O(1). Завдяки покажчикам на голову та хвіст, операції виконуються за постійний час.

2. Видалення в середині − O(1), якщо покажчик на вузол вже відомий. Якщо ж потрібно знайти вузол за значенням, складність залишається O(n).

3. Пошук / Доступ − O(n). Як і в однозв’язному списку, прямий доступ неможливий, що вимагає послідовного обходу.

Аналіз складності для вбудованих структур Python показує їх високу ефективність для певних сценаріїв.

1. Кортежі (tuple). Оскільки кортежі є незмінними, операції, що змінюють дані (додавання, видалення), не підтримуються. Доступ за індексом є постійною операцією O(1). Пошук елемента або ітерація по кортежу мають лінійну складність O(n).

2. Множини (set) та словники (dictionary). Ці структури, реалізовані як хеш-таблиці, є надзвичайно ефективними для операцій, що ґрунтуються на ключах або унікальних значеннях.

– Середня складність. Додавання, видалення та перевірка на входження мають середню часову складність O(1). Це робить їх ідеальними для завдань, де потрібен швидкий пошук.

– Найгірша складність. У рідкісних випадках, коли виникають колізії хешування (коли кілька елементів мають однаковий хеш), продуктивність може деградувати до лінійної складності O(n). Це підкреслює, що навіть під час використання оптимізованих вбудованих структур важливо розуміти їхні внутрішні механізми.

## 3.8. Приклади розв’язку задач

*Приклад 1.* Перевірка коректності дужкових послідовностей.

Дано послідовність дужок типу {, }, [, ], (, ). Використовуючи стек, перевірити, чи всі відкриваючі дужки мають відповідні закриваючі в правильному порядку.

Пояснення алгоритму. Алгоритм базується на принципі LIFO, який ідеально моделює вкладеність дужок. Проходячи по рядку зліва направо, виконуються такі кроки:

1. Створюється порожній стек.

2. Якщо зустрічається відкриваюча дужка ({, [, (), її додають на вершину стека (push).

3. Якщо зустрічається закриваюча дужка (}, ], )), перевіряється:

− Чи стек порожній. Якщо так, то послідовність некоректна (немає відповідної відкриваючої дужки).

− Чи елемент на вершині стека відповідає закриваючй дужці. Якщо так, пару вважають коректною і видаляють відкриваючу дужку зі стека (pop).

− Якщо умова відповідності не виконується, послідовність некоректна.

4. Після повного обходу рядка перевіряється, чи стек порожній. Якщо так, то всі відкриваючі дужки були успішно закриті, і послідовність є коректною. В іншому випадку, в стеку залишилися незакриті дужки, і послідовність вважається некоректною.

Реалізація на псевдокоді:

```
функція checkBrackets(рядок: string): boolean
 стек = СтворитиСтек()
 для кожного символу в рядку:
  якщо символ є відкриваючою дужкою ('{', '[', '('):
   стек.push(символ)
  інакше якщо символ є закриваючою дужкою ('}', ']', ')'):
   якщо стек.isEmpty() або не відповідає_парі(стек.pop(), символ):
    повернути false
 повернути стек.isEmpty()
```

```
Python
def check_brackets(s):
    stack =
    mapping = {")": "(", "}": "{", "]": "["}

    for char in s:
        if char in mapping.values():
            stack.append(char)
        elif char in mapping.keys():
            if not stack or mapping[char]! = stack.pop():
                return False

    return not stack
```

```
# Приклад використання
print(check_brackets("{[()]}"))  # Виведе: True
print(check_brackets("{[)]}"))   # Виведе: False
```

```
C++
#include<iostream>
#include<stack>
#include<string>

bool check_brackets(const std::string& s) {
   std::stack<char> st;
   for (char c : s) {
      if (c == '(' |

| c == '[' |
| c == '{') {
         st.push(c);
      } else if (c == ')' |

| c == ']' |
| c == '}') {
         if (st.empty()) {
            return false;
         }
         char top_char = st.top();
         st.pop();
         if ((c == ')' && top_char!= '(') ||
            (c == ']' && top_char!= '[') ||
            (c == '}' && top_char!= '{')) {
            return false;
         }
      }
   }
   return st.empty();
}

// Приклад використання
int main() {
   std::cout << std::boolalpha << check_brackets("{[()]}") << std::endl;
// Виведе: true
   std::cout << std::boolalpha << check_brackets("{[)]}") << std::endl;
// Виведе: false
   return 0;
}
```

*Приклад 2.* Застосування черги в алгоритмах обходу графа (BFS).

Продемонструвати принцип роботи черги на прикладі обходу графа в ширину (Breadth-First Search, BFS).

Пояснення алгоритму. Обхід в ширину − це алгоритм, який досліджує граф «шар за шаром», тобто спочатку відвідує всі вершини, що перебувають на відстані 1 від початкової, потім на відстані 2 і так далі. Черга ідеально підходить для цього, оскільки вона забезпечує обробку вершин у порядку їх виявлення (FIFO).

```
Псевдокод:
функція BFS(граф, початкова_вершина):
  створити_чергу Q
  створити_множину відвідані

  додати початкова_вершина в Q
  додати початкова_вершина в відвідані

  поки Q не пуста:
    поточна_вершина = Q.dequeue()
    виконати_обробку(поточна_вершина)

    для кожної сусідньої_вершини до поточної_вершини:
      якщо сусідня_вершина не в відвідані:
        додати сусідня_вершина в відвідані
        Q.enqueue(сусідня_вершина)
```

Цей алгоритм гарантує, що найближчі вершини будуть оброблені першими, оскільки вони були додані в чергу раніше.

## Контрольні питання

1. Поясніть, що таке абстрактний тип даних (АТД) і чим він відрізняється від структури даних.
2. Опишіть принцип роботи стека (LIFO) та наведіть реальний приклад.
3. Опишіть принцип роботи черги (FIFO) та наведіть реальний приклад.

4. Які основні операції підтримує стек? Назвіть їх.

5. Які основні операції підтримує черга? Назвіть їх.

6. Чому реалізація черги на звичайному масиві є неефективною? Яка її часова складність?

7. Чим двостороння черга (дек) відрізняється від стека і черги? Які її переваги?

8. Що таке зв'язний список і які його основні компоненти? Опишіть його структуру.

9. У чому полягає основна відмінність між однозв'язним та двобічнозв'язним списками?

10. Які переваги двобічнозв'язного списку над однозв'язним? Назвіть один з недоліків.

11. Поясніть поняття «часова складність» і що таке нотація Big O.

12. Чому доступ до елемента в масиві має складність O(1), а в зв'язному списку −O(n)?

13. Яка часова складність додавання елемента в кінець однозв'язного списку? А двобічнозв'язного?

14. Назвіть приклад використання стека в реальному програмуванні.

15. Назвіть приклад використання черги в реальному програмуванні.

16. Що таке неперервне (суміжне) розміщення в пам'яті і чому воно важливе для масивів?

17. Поясніть різницю між стековою та куповою алокацією пам'яті стосовно статичних та динамічних масивів.

18. Напишіть формулу для обчислення адреси елемента масиву за його індексом.

19. Чому операції вставки та видалення в статичному масиві мають складність O(n)?

20. Що таке «амортизаційний аналіз»? Чим він відрізняється від аналізу середнього випадку?

21. Поясніть, чому стратегія геометричного розширення (подвоєння ємності) є ключовою для досягнення амортизованої складності O(1) під час додавання елементів у динамічний масив.

22. У чому різниця між логічним розміром (size) та фізичною ємністю (capacity) динамічного масиву?

23. Чому доступ до елемента в динамічному масиві має складність O(1), незважаючи на те, що його розмір може змінюватися?

24. Порівняйте переваги та недоліки статичних і динамічних масивів.

25. Наведіть три приклади задач, де доцільніше використовувати статичний масив.

26. Наведіть три приклади задач, де доцільніше використовувати динамічний масив.

27. Що таке «локальність посилань» і як вона впливає на продуктивність операцій з масивами?

28. Чим реалізація listy Python відрізняється від std::vector в C++ з точки зору зберігання даних?

29. Що станеться, якщо спробувати звернутися до елемента за індексом, що виходить за межі масиву?

30. Чи може ємність динамічного масиву зменшуватися? Якщо так, то як і коли це може відбуватися?

## Задачі для самостійного опрацювання

1. Реалізувати стек на базі динамічного масиву та зв’язного списку.

2. Реалізувати чергу на базі масиву з циклічним буфером.

3. Написати функцію, що перевіряє, чи є рядок паліндромом, використовуючи стек.

4. Реалізувати чергу за допомогою двох стеків.

5. Написати програму, що імітує чергування клієнтів у банку з пріоритетом.

6. Реалізувати однозв’язний список, що підтримує операції додавання на початок, в кінець та видалення першого елемента.

7. Написати функцію для двобічнозв’язного списку, що змінює місцями сусідні елементи.

8. Реалізувати дек на основі двобічнозв’язного списку.

9. Написати функцію, що об’єднує два однозв’язних списки в один.

10. Здійснити інвертування зв’язного списку.

11. Написати функцію, що видаляє дублікати елементів з несортованого зв’язного списку.

12. Використовуючи стек, обчислити значення арифметичного виразу в постфіксній нотації.

13. Задана послідовність символів. Визначити, чи можна її перетворити на іншу послідовність, використовуючи стек (задача «перестановка»).

14. Знайти елемент, що міститься в k-ій позиції від кінця однозв’язного списку.

15. Реалізувати функцію, яка перевіряє, чи містить зв’язний список цикл.

16. Реалізувати функцію, що перевертає масив на місці (in-place), тобто без використання додаткового масиву.

17. Знайти мінімальний та максимальний елементи в масиві за один прохід.

18. Обчислити суму всіх елементів масиву.

19. Дано масив цілих чисел. Перемістити всі нульові елементи в кінець масиву, зберігаючи відносний порядок ненульових елементів.

20. Написати функцію, що перевіряє, чи є масив цілих чисел паліндромом (читається однаково зліва направо і справа наліво).

21. Реалізувати функцію, що виконує циклічний зсув масиву на k позицій праворуч. Наприклад, для і $k = 2$ результат має бути.

22. Дано масив, що може містити дублікати. Написати функцію, що повертає новий масив, який містить лише унікальні елементи з початкового.

23. Дано два відсортовані масиви. Об’єднати їх в один новий відсортований масив.

24. Знайти елемент, що зустрічається більше ніж n/2 разів у масиві розміром n (задача про мажоритарний елемент, алгоритм Боєра-Мура).

25. Дано масив цілих чисел nums. Повернути масив answer такий, що answer[i] дорівнює добутку всіх елементів nums, крім nums[i]. Задачу потрібно розв’язати за час O(n) і без використання операції ділення.

26. «Trapping Rain Water». Дано масив невід’ємних чисел, що представляють висоти стовпчиків однакової ширини. Обчислити, скільки води можна зібрати між ними після дощу.

27. Дано бінарний масив (складається з 0 та 1). Знайти довжину найдовшого неперервного підмасиву, що складається лише з одиниць.

28. Дано невпорядкований масив цілих чисел. Знайти найменший відсутній додатний цілий елемент. Наприклад, для [3, 4, -1, 1] відповідь − 2. Розв’язати за O(n) часу та O(1) додаткової пам’яті.

# ТЕМА 4
# СТРУКТУРИ ДЛЯ СЛОВНИКІВ

## 4.1. Словники та хеш-таблиці: фундаментальні поняття

Словник, також відомий як асоціативний масив або карта, є фундаментальним абстрактним типом даних (АТД) у комп'ютерних науках. Його основне призначення − зберігати колекцію елементів у вигляді пар «ключ-значення». Кожен ключ у словнику є унікальним і слугує ідентифікатором, який дає змогу швидко отримати доступ до асоційованого з ним значення. Інтерфейс словника, незалежно від його внутрішньої реалізації, зазвичай визначає три основні операції.

1. Додавання нової пари (ключ, значення).
2. Пошук значення за наданим ключем.
3. Видалення пари за ключем.

Принципова відмінність абстрактного типу даних від структури даних полягає саме в цій абстракції. АТД описує поведінку та семантику даних з точки зору користувача, не розкриваючи деталі внутрішньої реалізації. Наприклад, словник може бути реалізований за допомогою різних структур даних, таких як хеш-таблиця, збалансоване бінарне дерево, або навіть простий масив. Завдяки цій концепції розробник може працювати зі словником через його інтерфейс, не турбуючись про те, яка саме структура лежить в його основі. Це відповідає принципу інкапсуляції, який дає змогу в майбутньому змінювати реалізацію, наприклад, для покращення продуктивності або надійності, без необхідності змінювати код програм, що використовують словник. Така гнучкість і модульність є ключовими для розробки складних програмних систем.

*Хеш-таблиця* − це одна з найефективніших структур даних для реалізації словника. Вона дає змогу виконувати операції додавання, пошуку та видалення елементів у середньому за константний час O(1). Така виняткова швидкість робить хеш-таблиці ідеальним вибором для застосувань, де швидкий доступ до даних є критично важливим.

В основі хеш-таблиці лежить масив (або так званий «кошик»), де місце розташування елемента залежить від його ключа. Зв'язок між значенням ключа та його позицією в масиві встановлює спеціальна функція, що називається хеш-функцією. Вона перетворює довільний ключ на числовий індекс, який використовується для доступу до комірки масиву.

Перевага хеш-таблиці особливо помітна, коли множина ключів, що зберігаються, є набагато меншою за простір усіх можливих ключів. Уявімо, що нам потрібно зберігати дані для 10 000 користувачів, де ключами є їхні 12-значні ідентифікатори. Пряма адресація (використання масиву, де індекс відповідає ідентифікатору) вимагала б масив неймовірно великого розміру ($10^{12}$), що є практично неможливим. Хеш-таблиця вирішує цю проблему через відображення величезного простору ключів на набагато менший масив фіксованого розміру m. Це є яскравою ілюстрацією компромісу між часом і пам'яттю, де ми обмінюємо ефективність використання простору пам'яті на швидкість доступу.

Хоча хеш-таблиця є надзвичайно ефективною, вона не є універсальним рішенням. Існують інші структури даних, які також можуть реалізовувати словник, і вибір між ними залежить від конкретних вимог задачі.

1. Збалансовані бінарні дерева[17] пошуку (наприклад, червоно-чорні дерева). Ці структури гарантують логарифмічну складність O(log n) для всіх основних операцій (пошук, вставка, видалення)

---

[17] Бінарне дерево – це структура даних, у якій кожен вузол (node) має не більше ніж два нащадки:лівий (left child), правий (right child).Властивості бінарного дерева:

1. Є корінь (root) – початковий вузол.
2. Кожен вузол містить:
– значення (ключ, дані);
– посилання на лівого та правого нащадка (або null, якщо нащадка немає).
3. Дерево будується рекурсивно: кожен нащадок сам по собі є піддеревом.

Основні різновиди:

1. Повне бінарне дерево – усі рівні заповнені, крім, можливо, останнього.
2. Досконале (perfect) – усі рівні заповнені повністю.
3. Високобалансоване (balanced) – для кожного вузла висота лівого та правого піддерев відрізняється не більше ніж на 1.
4. Бінарне дерево пошуку (BST):
– усі ключі в лівому піддереві менші за ключ вузла;
– усі ключі в правому піддереві більші.

Застосування: організація пошуку (Binary Search Tree, AVL, Red-Black Tree), ефективне збереження та сортування даних, реалізація черг із пріоритетами (heap), парсери виразів у компіляторах (дерева розбору).

як у середньому, так і в найгіршому випадку. Їх перевага полягає в стабільності продуктивності та можливості обходу ключів у відсортованому порядку, що недоступно для хеш-таблиць.

2. Префіксні дерева (Trie). Ця структура даних особливо корисна для зберігання словників слів та пошуку за префіксом. Шлях від кореня до вузла в дереві визначає певний рядок. Хоча trie може бути дуже ефективним для певних типів ключів, воно може споживати значну кількість пам’яті.

Вибір між хеш-таблицею, збалансованим деревом та іншими структурами даних залежить від вимог до продуктивності, очікуваної кількості елементів та необхідності додаткових операцій, як-от обхід ключів у відсортованому порядку. Хеш-таблиця є оптимальним рішенням, коли основним пріоритетом є швидкий доступ до елементів за ключем, без додаткових вимог до впорядкованості.

Таблиця 4.1 – Порівняння структур даних для словників

| Тип структури даних | Часова складність (середній випадок) | Часова складність (найгірший випадок) | Використання пам’яті | Ключові переваги |
|---|---|---|---|---|
| **Хеш-таблиця** | Пошук: O(1) Вставка: O(1) | Пошук: O(n) Вставка: O(n) | O(n), з накладними витратами на покажчики / пусті комірки | Виняткова швидкість доступу (O(1) в середньому) |
| **Збалансоване дерево** | Пошук: O(logn) Вставка: O(logn) | Пошук: O(logn) Вставка: O(logn) | O(n) | Гарантована продуктивність, упорядкований обхід |
| **Префіксне дерево (Trie)** | Пошук: O(L) Вставка: O(L) | Пошук: O(L) Вставка: O(L) | O(n · Lavg) | Ефективний пошук за префіксом |
| **Сортований масив** | Пошук: O(logn) Вставка: O(n) | Пошук: O(logn) Вставка: O(n) | O(n) | Компактний, швидкий пошук, якщо масив статичний |
| *де L − довжина ключа, Lavg − середня довжина ключа, n − кількість елементів.* | | | | |

## 4.2. Принципи хешування та хеш-функції

*Хешування* – це фундаментальний процес перетворення вхідних даних довільного розміру у вихідний рядок (або число) фіксованої довжини, що називається хеш-значенням. Для хеш-таблиць основна мета хешування – відобразити великий і потенційно нескінченний простір ключів на обмежений діапазон індексів масиву. Завдяки цьому ми можемо оперувати ключами, що не є числовими або мають великі значення, і ефективно зберігати їх у звичайному масиві.

Процес хешування є одностороннім. Це означає, що з хеш-значення практично неможливо відновити початкові вхідні дані. Ця властивість, поряд з іншими, робить хешування надзвичайно корисним у багатьох галузях, крім хеш-таблиць, зокрема для:

– перевірки цілісності та автентифікації даних (наприклад, у цифрових підписах);

− пошуку дублікатів файлів;

– роботи систем контролю версій, таких як Git;

– забезпечення безпеки паролів (через зберігання їхніх хешів замість самих паролів).

Ефективність хеш-таблиці безпосередньо залежить від якості використовуваної хеш-функції. Хороша хеш-функція повинна відповідати кільком критичним вимогам.

1. Рівномірний розподіл (Uniformity). Функція має якомога рівномірніше розподіляти вхідні ключі по всьому діапазону індексів масиву. Це мінімізує ймовірність виникнення колізій, які є основною причиною деградації продуктивності хеш-таблиці.

2. Детермінізм. Для одного й того самого ключа хеш-функція завжди повинна повертати однакове хеш-значення. Це дає змогу послідовно знаходити елементи, які були додані раніше.

3. Ефективність. Обчислення хеш-функції має бути дуже швидким, щоб не ставати «вузьким місцем» в операціях хеш-таблиці.

Якщо хеш-функція не забезпечує рівномірного розподілу, це призводить до скупчення елементів в окремих комірках масиву.

У такому випадку середня складність операцій, яка в ідеалі дорівнює O(1), може деградувати до O(n), що перетворює хеш-таблицю на звичайний лінійний список. Це підкреслює, що швидкість хеш-таблиці є умовною і залежить від якості хеш-функції, яка по суті є її рушійною силою.

Існує багато методів для побудови хеш-функцій, але два з них є класичними та найпростішими для розуміння.

1. Метод ділення. Це найпоширеніший метод. Він обчислює хеш-значення як остачу від ділення ключа на розмір масиву. Формула[18]:

$$h(k) = k \bmod m,$$

де: $k$ − ключ, а $m$ − розмір масиву. Для мінімізації колізій рекомендовано обирати m простим числом, що не є близьким до степенів двійки.

2. Метод множення**.** Цей метод менш поширений, але ефективний, особливо для невипадкових ключів. Він обчислює хеш за формулою[19]:

$$h(k) = floor(m * (A*k \bmod 1)),$$

де: $A$ − константа в інтервалі $0 < A < 1$.

*Колізія* − це ситуація, коли хеш-функція повертає однаковий індекс для двох різних ключів. Навіть під час використання найкращої хеш-функції колізії є неминучими, особливо за збільшення кількості елементів.

Для оцінки «наповненості» хеш-таблиці використовується *коефіцієнт заповнення*(Load Factor), який визначається як відношення кількості елементів n до розміру масиву m: $\alpha = n/m$. Це є ключовим показником, що відображає ймовірність виникнення колізій. Зі збільшенням $\alpha$ зростає кількість колізій і, як наслідок, час, потрібний для виконання операцій. Коли коефіцієнт заповнення досягає певного критичного порогу (зазвичай 0.7−0.8),

[18] Оператор mod (остача від ділення) / Визначення: Вираз amodb означає остачу від ділення числа a на b.

[19] Функція floor (підлога, цілочисельне округлення вниз) / Визначення: floor(x) – це найбільше ціле число, яке не перевищує x.

хеш-таблицю потрібно *перебудувати (rehashing)*. Цей процес охоплює:

1. Створення нового, більшого масиву.
2. Повторне хешування та переміщення всіх існуючих елементів до нового масиву.
3. Звільнення пам’яті, зайнятої старим масивом.

Перебудова − це дорога операція з часовою складністю O(n), яка може тимчасово знизити продуктивність програми. Саме тому хеш-таблиця не гарантує O(1) для кожної окремої операції, а лише в середньому.

## 4.3. Методи розв’язання колізій

Для обробки колізій використовуються два основні підходи, які визначають, як будуть зберігатися елементи, що мають однакове хеш-значення.

*Метод ланцюжків*, або відкрите хешування, вирішує колізії за рахунок організації зв’язаних списків у кожній комірці масиву. Кожен елемент масиву є покажчиком на заголовок списку. Якщо кілька ключів хешуються в одну й ту саму комірку, вони просто додаються до відповідного списку. Так, колізії просто призводять до того, що довжина цих ланцюжків стає більшою за один елемент.

1. Вставка − новий елемент додається на початок або в кінець відповідного списку.
2. Пошук та видалення. Щоб знайти або видалити елемент, потрібно обійти весь зв’язаний список у відповідній комірці, порівнюючи ключі.

Переваги:

1. Простота реалізації. Цей метод є інтуїтивно зрозумілим і легким для реалізації.
2. Гнучкість. Він дає змогу коефіцієнту заповнення α бути більшим за 1, оскільки кількість елементів може перевищувати розмір масиву.
3. Ефективне використання пам’яті за рахунок того, що пам’ять для списків виділяється динамічно, лише за потреби.

Недоліки:

1. Накладні витрати пам’яті на зберігання покажчиків у кожному вузлі списку.

2. Низька продуктивність кешу, оскільки елементи, що належать до одного списку, можуть бути розкидані по різних ділянках пам’яті.

3. Деградація в найгіршому випадку. Якщо всі елементи хешуються в одну комірку, хеш-таблиця стає звичайним зв’язаним списком, і час доступу деградує до O(n).

На відміну від ланцюжків, *відкрита адресація* (або закрите хешування) зберігає всі елементи безпосередньо в основному масиві хеш-таблиці. Коли виникає колізія, алгоритм шукає наступну вільну комірку в масиві, дотримуючись певної «послідовності зондування». Основною відмінністю цього методу є те, що коефіцієнт заповнення α не може перевищувати 1, оскільки в кожній комірці може міститися лише один елемент.

Існує кілька стратегій зондування:

1. Лінійне зондування (Linear Probing). Це найпростіший метод. Він послідовно переглядає комірки з фіксованим кроком (зазвичай 1), доки не знайде вільну. Формула:

$$h(k,i) = (h_1(k) + i)(modm)$$, де $i$ − номер спроби.

Його основний недолік − «первинна кластеризація», коли елементи, що хешуються близько, утворюють довгі суцільні блоки, що уповільнює пошук.

2. Квадратичне зондування (Quadratic Probing). Цей метод використовує квадратичний приріст кроку зондування:

$$h(k, i) = (h_1(k) + i + i^2)(modm).$$

Це допомагає уникнути первинної кластеризації, але може призвести до «вторинної кластеризації», коли елементи з однаковим початковим хешем проходять однакову послідовність зондування.

3. Подвійне хешування (Double Hashing). Це один з найкращих методів відкритої адресації. Крок зондування визначається другою, допоміжною хеш-функцією: $h(k, i) = (h_1(k) + i \cdot h_2(k))(modm)$. Це забезпечує унікальну послідовність зондування для кожного

ключа, що дає змогу мінімізувати кластеризацію та досягти рівномірнішого розподілу.

Переваги відкритої адресації:

1. Краща продуктивність кешу. Елементи зберігаються послідовно в масиві, що підвищує ймовірність їх розміщення в кеші процесора.

2. Відсутність накладних витрат на покажчики.

Недоліки:

1. Складність реалізації операції видалення, оскільки просто видаливши елемент, можна порушити послідовність пошуку для інших елементів. Для вирішення цієї проблеми використовуються спеціальні маркери «видалено».

2. Чутливість до коефіцієнта заповнення. Ефективність різко падає під час наближенняα до 1.

Таблиця 4.2 – Порівняння методів розв’язання колізій

| Метод | Принцип роботи | Використання пам’яті | Продуктивність кешу | Макс. коефіцієнт заповнення (α) | Складність видалення |
|---|---|---|---|---|---|
| **Ланцюжки** | Кожна комірка – це покажчик на зв’язаний список | Динамічне, з накладними витратами на покажчики | Низька | > 1 | Пряме видалення зі списку, просте |
| **Відкрита адресація** | Зондування для пошуку вільної комірки в масиві | Статичне, з пустими комірками | Висока | ≤ 1 | Складне, вимагає використання маркерів «видалено» |
| **Лінійне зондування** | Послідовний перебір з кроком 1 | | | | |
| **Квадратичне зондування** | Послідовний перебір з квадратичним кроком | | | | |
| **Подвійне хешування** | Крок зондування визначається другою хеш-функцією | | | | |

## 4.4. Кейс-стаді: кешування в сучасних вебдодатках

*Кешування* − це стратегія збереження копій даних у швидкому, тимчасовому сховищі (кеші) для прискорення доступу до них у майбутньому. Основна ідея полягає в тому, щоб уникнути повторного виконання дорогих операцій, таких як звернення до бази даних або зовнішнього API, через зберігання їхніх результатів у оперативній пам'яті.

Хеш-таблиці є ідеальною структурою для реалізації кешу. Їхня здатність забезпечувати майже миттєвий доступ до даних за ключем є прямою відповіддю на основну вимогу кешування: надзвичайно швидке отримання інформації. Коли веб-серверу потрібно отримати дані (наприклад, частину вебсторінки або інформацію про користувача), він спочатку перевіряє кеш. Завдяки хеш-таблиці, ця перевірка відбувається за O(1) час, і якщо дані знайдено, це дає змогу уникнути повільного звернення до основного сховища даних. Це значно зменшує навантаження на сервери та покращує час відгуку.

*LRU-кеш* − це одна з найпоширеніших політик заміщення в кеші. Принцип роботи полягає в тому, що коли кеш заповнюється, з нього видаляється найменш нещодавно використаний елемент, щоб звільнити місце для нових даних. На перший погляд, може здатися, що звичайна хеш-таблиця є достатньою для реалізації LRU-кешу, адже вона забезпечує швидкий доступ до елементів. Однак проблема полягає в тому, як ефективно визначити «найменш нещодавно використаний» елемент, який потрібно видалити. У простій хеш-таблиці для пошуку цього елемента довелося б перебрати всі записи, що має часову складність O($n$). Це нівелює переваги швидкого доступу і робить таку реалізацію неефективною для високопродуктивних систем. Ця проблема вирішується через комбінування двох структур даних: хеш-таблиці та двонаправленого зв'язаного списку. Хеш-таблиця зберігає пари «ключ-вузол списку», забезпечуючи O(1) доступ до будь-якого елемента за ключем. Двонаправлений список зі свого боку підтримує порядок доступу до елементів:

1. Коли до елемента відбувається звернення, його відповідний вузол швидко переміщується на початок списку (голову).

2. Елемент наприкінці списку (хвості) завжди є найменш нещодавно використаним.

Отже, коли кеш заповнюється, видалення елемента з хвоста списку та відповідного запису з хеш-таблиці відбувається за O(1) час. Це є класичним прикладом того, як дві структури даних (кожна з яких має свої обмеження) можуть бути об'єднані для створення потужнішої гібридної структури, що відповідає складним вимогам.

Концепції кешування на основі хеш-таблиць широко використовуються в реальних застосунках. Одним з найпопулярніших інструментів для цього є Redis − in-memory сховище даних, яке працює за принципом «ключ-значення» і часто використовується як кеш, брокер повідомлень та база даних. Redis підтримує різні типи даних, зокрема хеші, і його висока продуктивність досягається саме завдяки зберіганню даних в оперативній пам'яті. Такі хмарні сервіси, як Azure Cache for Redis, надають кешування як керовану послугу, що демонструє, як технологія, заснована на хеш-таблицях, масштабується для великих корпоративних рішень, що вимагають низької затримки та високої пропускної здатності.

## 4.5. Практична реалізація хеш-таблиці

Для практичної реалізації хеш-таблиці потрібні такі компоненти:

1. Масив фіксованого розміру, який буде основним сховищем.
2. Структура для зберігання пар «ключ-значення». Для методу ланцюжків вона також повинна містити покажчик на наступний елемент.
3. Хеш-функція, що перетворює ключ на індекс масиву.
4. Методи для виконання операцій вставки, пошуку та видалення, що враховують обраний спосіб вирішення колізій.

Далі буде розглянута реалізація хеш-таблиці, що використовує метод ланцюжків, оскільки він є одним з найпростіших для навчальних цілей.

Реалізація на псевдокоді

```
// Структура вузла списку
СТРУКТУРА Node {
  ключ;
  значення;
  покажчик на наступний елемент (next);
}

// Структура хеш-таблиці
СТРУКТУРА HashTable {
  масив Node* table;
  розмір_таблиці (size);
}

// Хеш-функція
ФУНКЦІЯ hash(ключ, розмір_таблиці) {
  // Приклад: метод ділення
  ПОВЕРНУТИ ключ mod розмір_таблиці;
}

// Операція вставки
ФУНКЦІЯ INSERT(таблиця, ключ, значення) {
  // Обчислення індексу
  індекс = hash(ключ, таблиця.size);

  // Створення нового вузла
  новий_вузол = new Node(ключ, значення, NULL);

  // Вставка на початок списку
  новий_вузол.next = таблиця.table[індекс];
  таблиця.table[індекс] = новий_вузол;
}

// Операція пошуку
ФУНКЦІЯ SEARCH(таблиця, ключ) {
  індекс = hash(ключ, таблиця.size);
  поточний_вузол = таблиця.table[індекс];
```

```
  // Послідовний пошук у списку
  ПОКИ поточний_вузол НЕ NULL {
    ЯКЩО поточний_вузол.ключ == ключ {
      ПОВЕРНУТИ поточний_вузол.значення;
    }
    поточний_вузол = поточний_вузол.next;
  }
  // Елемент не знайдено
  ПОВЕРНУТИ NULL;
}

// Операція видалення
ФУНКЦІЯ DELETE(таблиця, ключ) {
  індекс = hash(ключ, таблиця.size);
  поточний_вузол = таблиця.table[індекс];
  попередній_вузол = NULL;

  ПОКИ поточний_вузол НЕ NULL {
    ЯКЩО поточний_вузол.ключ == ключ {
      ЯКЩО попередній_вузол == NULL {
        // Елемент - перший у списку
        таблиця.table[індекс] = поточний_вузол.next;
      } ІНАКШЕ {
        попередній_вузол.next = поточний_вузол.next;
      }
      звільнити(поточний_вузол);
      ПОВЕРНУТИ ПРАВДА; // Видалення успішне
    }
    попередній_вузол = поточний_вузол;
    поточний_вузол = поточний_вузол.next;
  }
  ПОВЕРНУТИ НЕПРАВДА; // Елемент не знайдено
}
```

У Python хеш-таблиці реалізовані вбудованим типом даних dict. Це один з найпотужніших інструментів мови, який забезпечує швидкий доступ до даних. Усі основні операції − вставка, пошук, видалення − виконуються через синтаксис квадратних дужок.

```
Python
# Приклад використання вбудованого словника (dict) в Python

# Створення словника
my_dict = {}

# Додавання елементів
my_dict['apple'] = 10
my_dict['banana'] = 20
my_dict['orange'] = 15

print(f"Словник після додавання: {my_dict}")
# Результат: {'apple': 10, 'banana': 20, 'orange': 15}

# Пошук елемента
value_apple = my_dict.get('apple')
print(f"Значення за ключем 'apple': {value_apple}")
# Результат: 10

# Оновлення значення
my_dict['apple'] = 12
print(f"Словник після оновлення: {my_dict}")
# Результат: {'apple': 12, 'banana': 20, 'orange': 15}

# Видалення елемента
del my_dict['banana']
print(f"Словник після видалення 'banana': {my_dict}")
# Результат: {'apple': 12, 'orange': 15}
```

Важливою особливістю Python є те, що лише *незмінювані (immutable)* об’єкти можуть бути ключами словника, оскільки їхнє хеш-значення є стабільним. До них належать рядки, числа, кортежі. Змінювані типи, такі як списки або множини, не можуть бути ключами, оскільки їхнє хеш-значення може змінюватися.

Реалізація на C/C++ вимагає ручного керування пам’яттю, що робить її більш складною, але дає змогу глибоко зрозуміти внутрішні механізми роботи хеш-таблиць.

```
C++
#include<iostream>
#include<string>
#include<vector>

// Структура для вузла списку
struct Node {
    std::string key;
    std::string value;
    Node* next;
};

// Структура для хеш-таблиці
struct HashTable {
    std::vector<Node*> table;
    int size;
    int count;
};

// Проста хеш-функція (метод ділення)
unsigned long hash_function(const std::string& key, int size) {
    unsigned long hash_val = 0;
    for (char c : key) {
        hash_val += c;
    }
    return hash_val % size;
}

// Ініціалізація хеш-таблиці
HashTable* create_table(int size) {
    HashTable* new_table = new HashTable();
    new_table->size = size;
    new_table->count = 0;
    new_table->table.resize(size, nullptr);
    return new_table;
}

// Вставка елемента
void insert(HashTable* table, const std::string& key, const std::string& value) {
    unsigned long index = hash_function(key, table->size);
    Node* new_node = new Node{key, value, nullptr};
```

```
    // Перевірка на дублікати та оновлення значення
    Node* current = table->table[index];
    while (current!= nullptr) {
        if (current->key == key) {
            current->value = value;
            delete new_node;
            return;
        }
        current = current->next;
    }

    // Вставка нового елемента на початок списку
    new_node->next = table->table[index];
    table->table[index] = new_node;
    table->count++;
}

// Пошук елемента
std::string* search(HashTable* table, const std::string& key) {
    unsigned long index = hash_function(key, table->size);
    Node* current = table->table[index];

    while (current!= nullptr) {
        if (current->key == key) {
            return &current->value;
        }
        current = current->next;
    }
    return nullptr;
}

// Видалення елемента
void remove(HashTable* table, const std::string& key) {
    unsigned long index = hash_function(key, table->size);
    Node* current = table->table[index];
    Node* prev = nullptr;

    while (current!= nullptr && current->key!= key) {
        prev = current;
        current = current->next;
    }
```

```
    if (current == nullptr) {
        return; // Елемент не знайдено
    }

    if (prev == nullptr) {
        table->table[index] = current->next;
    } else {
        prev->next = current->next;
    }
    delete current;
    table->count--;
}

// Функція для очищення пам’яті
void free_table(HashTable* table) {
    for (int i = 0; i < table->size; ++i) {
        Node* current = table->table[i];
        while (current!= nullptr) {
            Node* to_delete = current;
            current = current->next;
            delete to_delete;
        }
    }
    delete table;
}

int main() {
    HashTable* my_hash_table = create_table(10);
    insert(my_hash_table, "hello", "world");
    insert(my_hash_table, "key", "value");
    insert(my_hash_table, "test", "data");

    std::string* result = search(my_hash_table, "key");
    if (result) {
        std::cout << "Знайдено значення: " << *result << std::endl;
    } else {
        std::cout << "Ключ не знайдено." << std::endl;
    }
```

```
    remove(my_hash_table, "key");
    result = search(my_hash_table, "key");
    if (result) {
        std::cout << "Знайдено значення: " << *result << std::endl;
    } else {
        std::cout << "Ключ не знайдено." << std::endl;
    }

    free_table(my_hash_table);
    return 0;
}
```

## 4.6. Аналіз складності операцій

Середній випадок. У середньому, за добре спроєктованої хеш-функції та низького коефіцієнта заповнення, всі основні операції − вставка, пошук та видалення − виконуються за константний час, тобто за O(1). Це пояснюється тим, що обчислення хеш-значення займає постійний час, а колізії виникають рідко, або списки колізій є дуже короткими. У цьому і полягає основна перевага хеш-таблиць. Для методу ланцюжків середній час пошуку прямо залежить від коефіцієнта заповнення α і становить Θ(1 + α). Це означає, що зі збільшенням наповнення таблиці час доступу зростає, але лінійно.

Найгірший випадок. У найгіршому випадку, коли всі n ключів хешуються в одну комірку, хеш-таблиця перетворюється на звичайний лінійний список або один довгий кластер. У такому разі час для виконання операцій пошуку, вставки та видалення зростає лінійно з кількістю елементів, і часова складність дорівнює O(n). Така ситуація є вкрай небажаною і свідчить про неефективність хеш-функції або невдалий вибір її параметрів.

Просторова складність хеш-таблиці зазвичай становить O(n). Це означає, що пам’ять, необхідна для зберігання даних, зростає лінійно з кількістю елементів. До цього варто додати деякі накладні витрати:

1. У методі ланцюжків − додаткова пам’ять для покажчиків у кожному вузлі.

2. У відкритій адресації − пам’ять для незаповнених комірок масиву.

Таблиця 4.3 – Зведена таблиця складності операцій

| Операція | Часова складність (середній випадок) | Часова складність (найгірший випадок) | Просторова складність |
|---|---|---|---|
| **Вставка** | O(1) | O(n) | O(n) |
| **Пошук** | O(1) | O(n) | O(n) |
| **Видалення** | O(1) | O(n) | O(n) |

## 4.7. Приклади розв’язку задач

**Задача 1.** Підрахунок частоти слів у тексті. Потрібно підраховувати, скільки разів кожне слово зустрічається в заданому тексті.

Для розв’язання цієї задачі можна використати хеш-таблицю, де ключем буде слово, а значенням − його кількість.

1. Створюємо порожній словник frequency_map.
2. Розбиваємо вхідний текст на окремі слова.
3. Для кожного слова в тексті:

− Перевіряємо, чи існує воно як ключ у frequency_map.

− Якщо слово є, збільшуємо асоційоване з ним значення на 1.

− Якщо слова немає, додаємо його до словника з початковим значенням 1.

4. Після обробки всіх слів frequency_map буде містити частоту кожного слова.

Цей підхід має часову складність O(N), де N − загальна кількість слів у тексті, оскільки кожна операція додавання / оновлення в словнику виконується в середньому за O(1) час.

**Задача 2.** Пошук першого неповторюваного символу в рядку. Знайти перший символ у рядку, який зустрічається лише один раз.

Цю задачу можна вирішити за два проходи по рядку, використовуючи хеш-таблицю.

1. Створюємо словник char_counts.

2. Перший прохід. Проходимо по рядку від початку до кінця і підраховуємо частоту кожного символу, зберігаючи її в char_counts.

3. Другий прохід. Знову проходимо по рядку від початку. Для кожного символу перевіряємо його частоту в char_counts. Якщо частота дорівнює 1, повертаємо цей символ.

4. Якщо після другого проходу жоден символ не знайдено, це означає, що всі символи повторюються.

Цей алгоритм має часову складність O(n), де $n$ − довжина рядка, і просторову складність O($k$), де $k$ − кількість унікальних символів.

**Задача 3.** Знайти два елементи, сума яких дорівнює заданому значенню. Дано масив цілих чисел та цільове значення. Потрібно знайти два елементи в масиві, сума яких дорівнює цьому значенню.

Використовуємо хеш-таблицю для зберігання вже перевірених елементів та їхніх індексів.

1. Створюємо порожній словник seen_numbers.

2. Проходимо по масиву. Для кожного елемента num на поточному індексі i:

− Обчислюємо «доповнення» (complement): complement = target − num.

− Перевіряємо, чи існує complement як ключ у seen_numbers.

− Якщо complement знайдено, це означає, що ми знайшли пару. Повертаємо поточний індекс i та індекс, який зберігається в seen_numbers[complement].

− Якщо complement не знайдено, додаємо поточний елемент num та його індекс i до словника: seen_numbers[num] = i.

3. Якщо після обходу масиву пару не знайдено, повертаємо порожній результат.

Цей метод має часову складність O(n) та просторову O(n), що значно ефективніше, ніж наївне рішення з двома вкладеними циклами (O($n^2$)).

## Контрольні питання

1. Що таке абстрактний тип даних «Словник»? Чим він відрізняється від структури даних «Хеш-таблиця»?

2. Поясніть, що таке хешування. Яку роль відіграє хеш-функція в роботі хеш-таблиці?

3. Опишіть три ключові властивості «хорошої» хеш-функції.

4. Що таке колізія і чому вона є неминучою під час роботи з хеш-таблицями?

5. Поясніть принцип роботи методу ланцюжків для розв'язання колізій. Назвіть його переваги та недоліки.

6. Поясніть принцип роботи відкритої адресації. Чому для цього методу коефіцієнт заповнення не може бути більшим за 1?

7. Опишіть, у чому різниця між лінійним, квадратичним та подвійним зондуванням. Який з цих методів вважається найефективнішим і чому?

8. Поясніть, що таке «первинна кластеризація» і який метод зондування її спричиняє.

9. У чому полягає компроміс «пам'ять-час» під час використання хеш-таблиць?

10. Що таке коефіцієнт заповнення і як він впливає на продуктивність хеш-таблиці?

11. Опишіть процес перебудови (rehashing) хеш-таблиці. Коли він виконується і яка його часова складність?

12. Поясніть, як хеш-таблиця використовується для реалізації LRU-кешу. Яку додаткову структуру даних вона комбінує і чому?

13. Чому Python дає змогу використовувати лише незмінювані типи даних як ключі для словника?

14. Назвіть два приклади практичного застосування хешування за межами хеш-таблиць.

15. Поясніть, чому хеш-таблиця має середню часову складність O(1) для операцій, але в найгіршому випадку вона може деградувати до O(n).

## Задачі для самостійного опрацювання

1. Реалізуйте хеш-таблицю, що використовує метод ланцюжків, на C++ без використання стандартних бібліотек std::unordered_map.

2. Напишіть Python-функцію, яка перевіряє, чи є два рядки анаграмами. Використайте словник для ефективного розв'язання.

3. Змоделюйте покрокову вставку 8 елементів з ключами: 25, 12, 35, 42, 57, 65, 71, 80 у хеш-таблицю з 10 комірками. Використайте метод лінійного зондування та хеш-функцію h(k) = k mod 10.

4. Напишіть програму, яка знаходить найдовшу підпослідовність унікальних символів у заданому рядку, використовуючи хеш-таблицю.

5. Змоделюйте вставку тих самих 8 елементів з ключами: 25, 12, 35, 42, 57, 65, 71, 80 у хеш-таблицю з 10 комірками, але цього разу використайте метод ланцюжків.

6. Створіть власний клас Point (x, y) у Python та зробіть його хешованим, щоб він міг бути ключем у словнику.

7. Розробіть алгоритм для знаходження кількості унікальних елементів у масиві, де деякі елементи можуть бути дублікатами.

8. Реалізуйте LRU-кеш для 10 елементів на Python.

9. Напишіть функцію, яка ефективно перевіряє, чи містить масив дублікати.

10. Розробіть алгоритм, який знаходить усі пари чисел у масиві, сума яких дорівнює заданому значенню, використовуючи хеш-таблицю.

11. Напишіть програму, яка симулює хеш-таблицю з відкритою адресацією та лінійним зондуванням, що автоматично перебудовується за досягненні коефіцієнта заповнення 0.7.

12. Порівняйте на практиці часову ефективність пошуку в std::unordered_map (C++) та std::map (C++) для 100 000 випадково згенерованих чисел.

13. Реалізуйте операцію видалення для хеш-таблиці з відкритою адресацією, що використовує маркери «видалено» для коректної роботи алгоритму пошуку.

14. Спроєктуйте алгоритм, який використовує хешування для перевірки, чи є один масив перестановкою іншого.

15. Напишіть функцію, яка обчислює хеш-значення для рядка, використовуючи метод множення.

# ТЕМА 5
# ДЕРЕВОПОДІБНІ СТРУКТУРИ

У комп'ютерних науках структури даних є фундаментальним інструментом для організації та зберігання інформації. Досі ми переважно розглядали лінійні структури даних, такі як масиви та зв'язні списки, де елементи розташовані послідовно, один за одним. Однак багато реальних задач вимагають представлення ієрархічних зв'язків, які неможливо ефективно змоделювати за допомогою лінійних підходів. Для таких завдань використовуються деревоподібні структури.

*Дерево* − це нелінійна структура даних, що складається з набору пов'язаних вузлів. З точки зору теорії графів, дерево є зв'язним ациклічним графом. Ця структура природно моделює ієрархії, такі як файлова система на комп'ютері, організаційна структура компанії, генеалогічне дерево або синтаксична структура речення в мові.

Основна термінологія. Для ефективного вивчення дерев потрібно засвоїти ключову термінологію, яка описує їхні компоненти та властивості:

- Вузол (Node) − основний елемент дерева, який містить дані та посилання на інші вузли.
- Ребро (Edge) − зв'язок між двома вузлами.
- Корінь (Root) − найвищий вузол у дереві, який не має батьківського вузла. Кожне не порожнє дерево має рівно один корінь.
- Батьківський вузол (Parent − вузол, який має посилання на інший вузол. У дереві кожен вузол (крім кореня) має рівно одного батька.
- Нащадок (Child) − вузол, на який посилається батьківський вузол.
- Листок (Leaf) − вузол, який не має нащадків. Також називається кінцевим або зовнішнім вузлом.
- Внутрішній вузол (Internal Node) − будь-який вузол, що не є листком, тобто має хоча б одного нащадка.

- Піддерево (Subtree) − дерево, що складається з вузла та всіх його нащадків.
- Шлях (Path) − послідовність вузлів та ребер, що з'єднують вузол з його нащадком.
- Глибина (Depth) вузла − довжина шляху від кореня до цього вузла. Глибина кореня дорівнює 0.
- Висота (Height) дерева − максимальна глибина будь-якого вузла в дереві. Висота порожнього дерева зазвичай визначається як -1, а дерева з одним вузлом − як 0.

## 5.1. Бінарні дерева та їх обходи

Серед усіх деревоподібних структур бінарні дерева є одними з найпоширеніших та найважливіших для вивчення. Вони слугують основою для багатьох складніших структур, охоплюючи бінарні дерева пошуку, купи та дерева виразів.

*Бінарне дерево* − це скінченна множина вузлів, яка або є порожньою, або складається з кореневого вузла та двох окремих бінарних дерев, що називаються лівим та правим піддеревами.

Ключова відмінність бінарного дерева від загального дерева полягає в тому, що кожен вузол може мати *не більше двох* нащадків, і ці нащадки є впорядкованими: існує чітке розрізнення між лівим та правим нащадком. Навіть якщо вузол має лише одного нащадка, важливо, чи є він лівим, чи правим.

Обхід дерева − це процес систематичного відвідування кожного вузла рівно один раз. Для бінарних дерев найпоширенішими є алгоритми обходу в глибину (DFS), які досліджують гілки дерева настільки глибоко, наскільки це можливо, перш ніж повернутися назад (backtracking). Існує три основних типи обходу в глибину, що відрізняються порядком відвідування кореня відносно його піддерев.

Розглянемо наступне бінарне дерево як приклад для всіх трьох обходів:

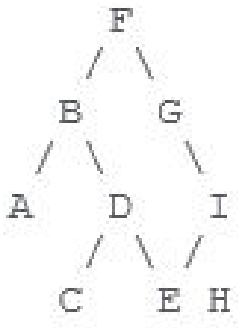



Рисунок 5.1 – Структура бінарного дерева

Прямий обхід (Pre-order, NLR). Цей алгоритм відповідає порядку: Корінь → Ліве піддерево → Праве піддерево.

Алгоритм:

1. Відвідати кореневий вузол.
2. Рекурсивно обійти ліве піддерево.
3. Рекурсивно обійти праве піддерево.

Прямий обхід часто використовується для створення копії дерева, оскільки він відтворює структуру, починаючи з кореня. Також він дає змогу отримати префіксну форму математичного виразу (польський запис) з дерева виразів.

Приклад. Для дерева вище порядок прямого обходу буде: F, B, A, D, C, E, G, I, H.

Симетричний (центрований) обхід (In-order, LNR). Цей алгоритм слідує порядку: Ліве піддерево → Корінь → Праве піддерево.

Алгоритм:

1. Рекурсивно обійти ліве піддерево.
2. Відвідати кореневий вузол.
3. Рекурсивно обійти праве піддерево.

Симетричний обхід є надзвичайно важливим для бінарних дерев пошуку (BST), оскільки він дає змогу отримати ключі у відсортованому (зростаючому) порядку. Це є однією з ключових властивостей, яка робить BST ефективними для операцій, що вимагають впорядкованості.

Приклад. Для нашого дерева порядок симетричного обходу буде: A, B, C, D, E, F, G, H, I.

Зворотний обхід (Post-order, LRN). Цей алгоритм слідує порядку: Ліве піддерево → Праве піддерево → Корінь.

Алгоритм:

1. Рекурсивно обійти ліве піддерево.
2. Рекурсивно обійти праве піддерево.
3. Відвідати кореневий вузол.

Зворотний обхід є незамінним, коли операції над вузлом вимагають, щоб його нащадки були оброблені першими. Класичний приклад − видалення дерева з пам'яті: ми повинні спочатку видалити всіх нащадків, перш ніж видаляти сам батьківський вузол, щоб уникнути «висячих» вказівників. Також цей обхід використовується для отримання постфіксної форми виразу (зворотний польський запис).

*Приклад.* Для нашого дерева порядок зворотного обходу буде: A, C, E, D, B, H, I, G, F.

Порядок відвідування вузла відносно його піддерев не є довільним вибором. Це фундаментальне рішення, яке визначає, для яких обчислювальних завдань підходить той чи інший алгоритм обходу. Прямий обхід (корінь-спочатку) ідеальний для завдань, де батьківський вузол має бути оброблений перед нащадками, наприклад, під час серіалізації дерева. Зворотний обхід (корінь-вкінці) використовується у протилежних ситуаціях, коли нащадки мають бути оброблені до батька, як за звільнення пам'яті. Симетричний обхід зі свого боку є основою для роботи з упорядкованими даними в бінарних деревах пошуку.

Обходи дерев можна реалізувати двома основними способами: рекурсивно та ітеративно.

*Рекурсивний підхід.* Рекурсивна реалізація є найбільш природною та елегантною, оскільки структура дерева сама по собі є рекурсивною. Кожне піддерево є меншим деревом, що дає змогу застосовувати той самий алгоритм до менших частин проблеми.

*Псевдокод:*

```
// Прямий обхід
procedure PreOrder(node)
  if node is not null
    visit(node)
    PreOrder(node.left)
    PreOrder(node.right)

// Симетричний обхід
procedure InOrder(node)
  if node is not null
    InOrder(node.left)
    visit(node)
    InOrder(node.right)

// Зворотний обхід
procedure PostOrder(node)
  if node is not null
    PostOrder(node.left)
    PostOrder(node.right)
    visit(node)
```

*Python*

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

def print_preorder(root):
    if root:
        print(root.val, end=" ")
        print_preorder(root.left)
        print_preorder(root.right)

def print_inorder(root):
    if root:
        print_inorder(root.left)
        print(root.val, end=" ")
        print_inorder(root.right)
```

```
def print_postorder(root):
    if root:
        print_postorder(root.left)
        print_postorder(root.right)
        print(root.val, end=" ")
```

```
C++
#include<iostream>

struct Node {
    int data;
    struct Node *left, *right;
};

Node* newNode(int data) {
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = nullptr;
    return temp;
}

void printPreorder(struct Node* node) {
    if (node == nullptr)
        return;
    std::cout << node->data << " ";
    printPreorder(node->left);
    printPreorder(node->right);
}

void printInorder(struct Node* node) {
    if (node == nullptr)
        return;
    printInorder(node->left);
    std::cout << node->data << " ";
    printInorder(node->right);
}

void printPostorder(struct Node* node) {
    if (node == nullptr)
        return;
    printPostorder(node->left);
    printPostorder(node->right);
    std::cout << node->data << " ";
}
```

*Ітеративний підхід.* Хоча рекурсивний підхід є інтуїтивно зрозумілим, він має недолік: для глибоких дерев він може призвести до переповнення стека викликів (stack overflow), оскільки кожен рекурсивний виклик займає місце в системному стеку[20]. У найгіршому випадку, коли дерево вироджується у зв'язний список, глибина рекурсії може дорівнювати кількості вузлів n, що призводить до просторової складності O(*n*).

Ітеративний підхід вирішує цю проблему, використовуючи явний стек (зазвичай реалізований на купі), що дає змогу уникнути переповнення системного стека і надає більший контроль над виконанням.

```
Псевдокод (прямий обхід):
procedure IterativePreOrder(root)
  if root is null
    return

  stack s
  s.push(root)

  while s is not empty
    node = s.pop()
    visit(node)

    // Спочатку додаємо правого нащадка, щоб лівий був оброблений першим (LIFO)
    if node.right is not null
      s.push(node.right)
    if node.left is not null
      s.push(node.left)
```

[20] Системний стек (call stack, стек викликів) – це спеціальна ділянка оперативної пам'яті, яку використовує процесор та операційна система для збереження проміжних даних під час виконання програм. Основні функції системного стека:

1. Зберігання адрес повернення – коли викликається функція, у стек заноситься адреса інструкції, куди потрібно повернутися після її виконання.
2. Зберігання параметрів функцій – аргументи функції часто передаються через стек.
3. Зберігання локальних змінних – кожна функція має власну *рамку стека* (stack frame), де зберігаються її локальні змінні.
4. Керування рекурсією – кожен рекурсивний виклик створює нову рамку в стеку.

Важливі властивості:

1. Організований за принципом LIFO (Last In, First Out) – останній виклик знімається першим.
2. Має обмежений розмір (stack size). Якщо перевищити його через нескінченну рекурсію → виникає stackoverflow.

```
Python
def iterative_inorder(root):
    stack =
    current = root
    while current or stack:
        while current:
            stack.append(current)
            current = current.left

        current = stack.pop()
        print(current.val, end=" ")
        current = current.right
```

```
C++
#include<stack>
#include<vector>
#include<algorithm>

void printPostorderIterative(Node* root) {
    if (root == nullptr) return;

    std::stack<Node*> s1, s2;
    s1.push(root);
    Node* node;

    while (!s1.empty()) {
        node = s1.top();
        s1.pop();
        s2.push(node);

        if (node->left)
            s1.push(node->left);
        if (node->right)
            s1.push(node->right);
    }

    while (!s2.empty()) {
        node = s2.top();
        s2.pop();
        std::cout << node->data << " ";
    }
}
```

Вибір між рекурсивною та ітеративною реалізацією є класичним компромісом у програмуванні. Рекурсія пропонує чистоту коду та близькість до математичного визначення, але покладається на системний стек, що може бути ризиковано для дерев великої глибини. Ітеративний підхід, хоч і є складнішим для написання та розуміння, забезпечує надійність та стійкість до переповнення стека, переносячи управління пам'яттю з неявного системного рівня на явний, контрольований програмістом.

Таблиця 5.1 – Порівняння реалізацій обходів бінарного дерева

| Метод | Часова складність | Просторова складність | Переваги | Недоліки |
|---|---|---|---|---|
| Рекурсивний DFS | O(n) | O(h) (у найгіршому випадку O(n)) | Простий та інтуїтивно зрозумілий код, що відповідає рекурсивній природі дерева | Ризик переповнення стека для глибоких або вироджених дерев. Неявне використання системного стека |
| Ітеративний DFS | O(n) | O(h) (у найгіршому випадку O(n)) | Надійний, не залежить від глибини рекурсії, відсутній ризик переповнення стека. Явний контроль над стеком | Складніший для реалізації та розуміння, особливо для симетричного та зворотного обходів |

*Примітка: n − кількість вузлів у дереві, h − висота дерева.*

## 5.2. Бінарні дерева пошуку (BST): операції вставки, пошуку, видалення

*Бінарне дерево пошуку* (Binary Search Tree, BST) − це спеціалізований тип бінарного дерева, який значно прискорює операції пошуку, вставки та видалення елементів. Його ефективність базується на строгій властивості впорядкування, яка дає змогу відкидати половину дерева на кожному кроці пошуку, подібно до алгоритму двійкового пошуку в масиві.

BST визначається наступною ключовою властивістю, яка повинна виконуватися для кожного вузла в дереві рекурсивно.

Для будь-якого вузла x:

1. Усі ключі в його лівому піддереві є меншими або рівними ключу вузла x (key(y) ≤ key(x) для будь-якого y в лівому піддереві).
2. Усі ключі в його правому піддереві є більшими або рівними ключу вузла x (key(y) ≥ key(x) для будь-якого y в правому піддереві).

Ця властивість гарантує, що симетричний обхід (in-order) BST завжди повертає елементи у відсортованому порядку.

Основні операції в BST виконуються за час, пропорційний висоті дерева, h.

Пошук (Search). Алгоритм пошуку використовує властивість BST для швидкого знаходження ключа.

1. Почати з кореня.
2. Порівняти шуканий ключ з ключем поточного вузла.
3. Якщо ключі збігаються, елемент знайдено.
4. Якщо шуканий ключ менший, перейти до лівого нащадка.
5. Якщо шуканий ключ більший, перейти до правого нащадка.
6. Повторювати кроки 2−5, доки елемент не буде знайдено або доки не буде досягнуто кінця гілки (нульовий вказівник), що означає відсутність елемента в дереві.

Часова складність цієї операції становить O(h).

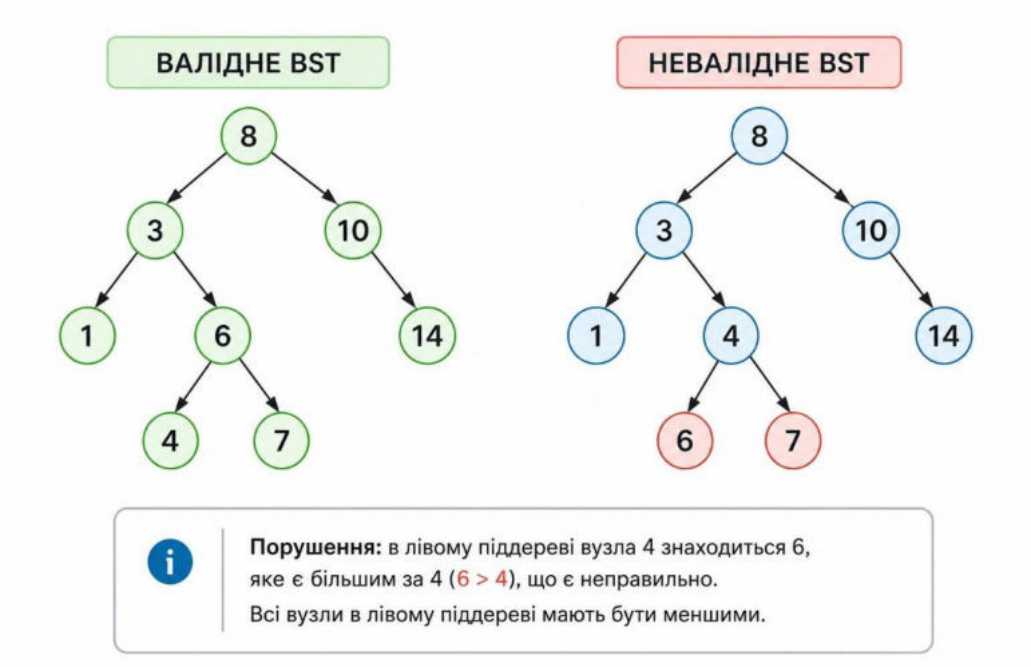



Рисунок 5.2 – Приклад валідного та невалідного BST (верстальнику, виправити на рисунку зверху помилку)

Вставка (Insertion). Вставка нового елемента також використовує властивість BST для знаходження правильного місця.

1. Виконати пошук ключа, який потрібно вставити.

2. Пошук завершиться на вузлі, у якого відсутній відповідний нащадок (лівий або правий).

3. Створити новий вузол і приєднати його як відповідного нащадка до цього вузла.

Як і пошук, вставка має часову складність O($h$).

Видалення (Deletion). Видалення є найскладнішою операцією, оскільки потрібно зберегти властивість BST після видалення вузла. Розглядають три випадки:

1. Вузол для видалення є листком. Це найпростіший випадок. Вузол просто видаляється, а посилання на нього в батьківського вузла встановлюється в null.

2. Вузол для видалення має одного нащадка. Вузол замінюється своїм єдиним нащадком. Посилання в батьківського вузла тепер вказує на нащадка видаленого вузла.

3. Вузол для видалення має двох нащадків. Це найскладніший випадок, оскільки просте видалення вузла розділить дерево на дві частини.

3.1. Знайти in-order наступника (in-order successor) вузла − це найменший елемент у його правому піддереві. Альтернативно, можна знайти in-order попередника (in-order predecessor) − найбільший елемент у лівому піддереві.

3.2. Скопіювати значення ключа наступника (або попередника) у вузол, який потрібно видалити.

3.3. Рекурсивно видалити наступника (або попередника) з його початкового місця. Оскільки наступник є найменшим елементом у своєму піддереві, він не може мати лівого нащадка, що зводить задачу видалення до простіших випадків 1 або 2.

Часова складність видалення також становить O(h).

Таблиця 5.2 – Часова складність операцій в BST

| Операція | Найкращий випадок | Середній випадок | Найгірший випадок |
|---|---|---|---|
| Пошук | O(logn) | O(logn) | O(n) |
| Вставка | O(logn) | O(logn) | O(n) |
| Видалення | O(logn) | O(logn) | O(n) |

*Примітка: Найкращий та середній випадки відповідають збалансованому або випадково побудованому дереву, де h ≈ logn. Найгірший випадок відповідає виродженому (незбалансованому) дереву, де h ≈ n.*

## 5.3. Проблема деградації дерева

Ефективність бінарного дерева пошуку, як показано в таблиці 5.2, критично залежить від його висоти *h*. В ідеальному випадку, коли дерево є збалансованим, його висота є логарифмічною відносно кількості вузлів ($h \approx \log_2 n$), що забезпечує чудову продуктивність. Однак BST не має вбудованих механізмів для підтримки балансу, що може призвести до серйозної проблеми, відомої як деградація дерева.

Деградація відбувається, коли послідовність операцій вставки або видалення призводить до створення сильно незбалансованої, або «викривленої» (skewed), структури. Найпоширенішою причиною є вставка елементів у вже відсортованому або майже відсортованому порядку (зростаючому або спадному).

Розглянемо вставку послідовності ключів: 10, 20, 30, 40, 50.

- Вставляємо 10: 10 стає коренем.
- Вставляємо 20: 20 > 10, тому 20 стає правим нащадком 10.
- Вставляємо 30: 30 > 10, 30 > 20, тому 30 стає правим нащадком 20.
- Вставляємо 40: 40 > 10, 40 > 20, 40 > 30, тому 40 стає правим нащадком 30.
- Вставляємо 50: 50 > 10, 50 > 20, 50 > 30, 50 > 40, тому 50 стає правим нащадком 40.

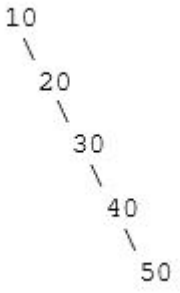



Рисунок 5.3 – Деградація BST у зв’язний список

Отримана структура функціонально еквівалентна звичайному зв’язному списку.

Коли дерево деградує, його висота h стає рівною кількості вузлів n. Це має катастрофічні наслідки для продуктивності. Часова складність операцій (пошук, вставка, видалення) зростає з O(logn) до O(n).У такому стані BST втрачає всі свої переваги і працює не ефективніше, ніж простий масив або зв’язний список, для яких пошук вимагає лінійного часу.

Проблема деградації виникає через фундаментальне обмеження простого BST. Його інваріант[21] (властивість впорядкування) забезпечує лише коректність, але нічого не говорить про структуру чи баланс. Алгоритм вставки є «жадібним»: він розміщує новий вузол у першому ж доступному місці, не враховуючи

[21] Інваріант (від лат. *invarians* – «незмінний») – це властивість, вираз або умова, яка залишається істинною (незмінною) упродовж виконання алгоритму, програми чи під час трансформацій даних. У програмуванні виділяють кілька типів інваріантів:

1. Цикловий інваріант:

– умова, яка є істинною до початку, після кожної ітерації та після завершення циклу.

– Використовується для доведення коректності алгоритмів.

– Приклад: у алгоритмі сортування вставками – «підмасив зліва від поточної позиції вже відсортований».

2. Класовий інваріант (в ООП):

– умова, яка завжди виконується для кожного екземпляра класу після створення та після будь-якої операції над ним.

– Наприклад: у класі «Рахунок у банку» баланс не може бути від’ємним.

3. Алгоритмічний інваріант:

– величина або властивість, що не змінюється під час виконання алгоритму.

– Приклад: у алгоритмі Евкліда для знаходження НСД завжди виконується НСД(a, b) = НСД(b, a mod b).

4. Математичний інваріант – властивість об’єкта, яка не змінюється за певних перетворень (наприклад, довжина вектора під час поворотів у просторі).

Синоніми та близькі терміни: незмінна умова, стала властивість, умова правильності.

глобального впливу на висоту дерева. Ця «короткозорість» і є першопричиною деградації. Це не помилка в алгоритмі, а невід'ємна властивість базового визначення BST, яка мотивувала створення цілого класу досконаліших структур − дерев, що самобалансуються.

## 5.4. Збалансовані двійкові дерева пошуку (AVL, червоно-чорні)

Щоб гарантувати логарифмічну продуктивність у найгіршому випадку, були розроблені *самобалансувальні бінарні дерева пошуку*. Ці структури є розширеннями BST, які автоматично підтримують свою висоту близькою до логарифмічної через виконання операцій реструктуризації (балансування) після кожної модифікації (вставки або видалення).

Основна ідея самобалансування полягає у введенні додаткових, суворіших інваріантів (правил), що стосуються структури дерева, на додаток до основної властивості BST. Коли операція вставки або видалення порушує один із цих інваріантів, дерево виконує локальні перетворення, які називаються *поворотами* (rotations), щоб відновити баланс.

Назване на честь своїх винахідників, Адельсона-Вельського та Ландіса, AVL-дерево є першим винайденим самобалансувальним бінарним деревом пошуку.

AVL-дерево − це BST, в якому для кожного вузла висоти його лівого та правого піддерев відрізняються не більше ніж на одиницю.

Для підтримки цього інваріанту кожен вузол зберігає коефіцієнт балансу, який визначається як:

$$\text{BalanceFactor} = \text{height}(\text{left\_subtree}) - \text{height}(\text{right\_subtree}).$$

У збалансованому AVL-дереві коефіцієнт балансу кожного вузла може приймати лише значення -1 (праве піддерево вище), 0 (піддерева однакової висоти) або +1 (ліве піддерево вище).

Якщо після вставки або видалення коефіцієнт балансу якогось вузла стає -2 або +2, дерево вважається незбалансованим, і для цього вузла виконується поворот.

Повороти (Rotations) − це локальні операції, які змінюють структуру дерева, переміщуючи вузли, але зберігаючи властивість BST. Існує чотири основні сценарії дисбалансу, які виправляються за допомогою одного або двох поворотів.

1. Left-Left (LL) випадок. Дисбаланс виникає через вставку в ліве піддерево лівого нащадка. Виправляється одним правим поворотом.

2. Right-Right (RR) випадок. Дисбаланс виникає через вставку в праве піддерево правого нащадка. Виправляється одним лівим поворотом.

3. Left-Right (LR) випадок. Дисбаланс виникає через вставку в праве піддерево лівого нащадка. Виправляється лівим поворотом навколо лівого нащадка, за яким слідує правий поворот навколо початкового незбалансованого вузла.

4. Right-Left (RL) випадок. Дисбаланс виникає через вставку в ліве піддерево правого нащадка. Виправляється правим поворотом навколо правого нащадка, за яким слідує лівий поворот навколо початкового незбалансованого вузла.

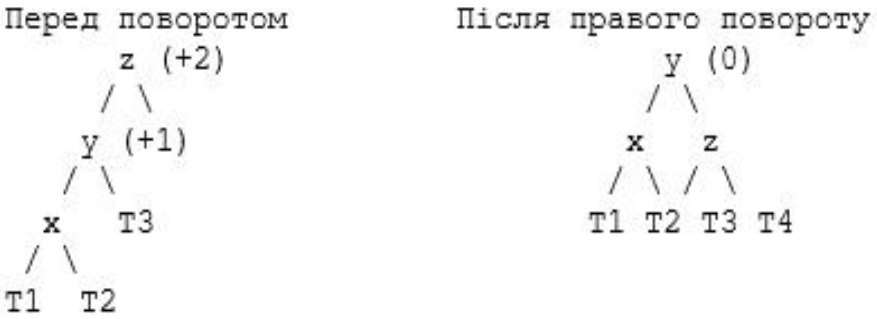



Рисунок 5.4 – Одиночний правий поворот (для LL випадку)

Червоно-чорне дерево (R-B tree) − це інший тип самобалансувального BST, який забезпечує баланс за допомогою менш суворих правил, ніж AVL-дерево. Замість зберігання висоти кожен вузол має додатковий біт для зберігання кольору − *червоного* або *чорного*.

Властивості червоно-чорного дерева:

Щоб бути валідним, червоно-чорне дерево повинно відповідати таким п’яти властивостям.

1. Кожен вузол є або червоним, або чорним.
2. Корінь завжди чорний.
3. Кожен листок (уявні NIL вузли) є чорним.
4. Якщо вузол червоний, то обидва його нащадки є чорними (не може бути двох червоних вузлів поспіль на одному шляху).
5. Для кожного вузла всі прості шляхи від цього вузла до його нащадків-листків містять однакову кількість чорних вузлів. Ця кількість називається чорною висотою вузла.

Ці властивості гарантують, що найдовший шлях від кореня до листка не більше ніж удвічі довший за найкоротший шлях, що забезпечує висоту дерева $h \leq 2log_2(n + 1)$.

Після вставки або видалення властивості дерева можуть бути порушені. Відновлення балансу відбувається за допомогою комбінації поворотів та перефарбовування (recoloring) вузлів. Нові вузли завжди вставляються як червоні. Якщо це порушує властивість 4 (червоний батько має червоного нащадка), виконується процедура виправлення, яка аналізує колір «дядька» (брата батьківського вузла) нового вузла і, залежно від цього, виконує повороти або перефарбовування.

Існування двох популярних типів збалансованих дерев, AVL та червоно-чорних, не є випадковістю. Вони являють собою компроміс між швидкістю пошуку та швидкістю модифікації. AVL-дерева є більш строго збалансованими, оскільки різниця висот піддерев не може перевищувати 1. Це призводить до того, що дерева в середньому є нижчими, що робить операції пошуку швидшими. Однак підтримка такого суворого балансу вимагає частіших та потенційно складніших поворотів під час вставки та видалення.

З іншого боку, червоно-чорні дерева мають менш суворі правила балансування (найдовший шлях не більше ніж удвічі довший за найкоротший). Це дає змогу дереву бути дещо «незбалансованим», що значно зменшує кількість поворотів, потрібних під час вставки та видалення. Часто для відновлення властивостей

достатньо лише перефарбувати кілька вузлів, що є значно швидшою операцією, ніж поворот. Цей компроміс робить AVL-дерева кращим вибором для систем, де операції читання (пошуку) значно переважають операції запису (наприклад, деякі типи баз даних), тоді як червоно-чорні дерева є стандартом де-факто для застосунків з інтенсивними операціями вставки та видалення, і саме тому вони використовуються в стандартних бібліотеках мов програмування, таких як std::map в C++ та TreeMap в Java.

Таблиця 5.3 – Порівняльний аналіз збалансованих дерев

| Характеристика | Звичайне BST | AVL-дерево | Червоно-чорне дерево |
|---|---|---|---|
| Гарантована висота | O(n) | $\approx 1.44\log_2 n$ | $\leq 2\log_2(n+1)$ |
| Складність пошуку (найгірший випадок) | O(n) | O(logn) | O(logn) |
| Складність вставки / видалення (найгірший випадок) | O(n) | O(logn) | O(logn) |
| Максимальна кількість поворотів на операцію | 0 | 2 (для LR/RL) | 2 |
| Строгість балансування | Відсутня | Дуже сувора | Менш сувора |
| Типове застосування | Прості навчальні реалізації | Бази даних та інші системи, де пошук є домінуючою операцією | Стандартні бібліотеки (C++ std::map, Java TreeMap), планувальники ОС |

## 5.5. Реалізація бінарного дерева пошуку та основних операцій

Після розгляду теоретичних аспектів BST та проблем, що виникають через їх незбалансованість, важливо зрозуміти, як ці структури реалізуються на практиці. У цьому розділі наведено повну реалізацію стандартного (незбалансованого) бінарного дерева пошуку.

Основою будь-якого дерева є його вузол. Для BST вузол повинен містити ключ (дані) та вказівники на лівого та правого нащадків.

```
Python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

```
C++
struct Node {
    int key;
    Node *left;
    Node *right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};
```

Клас дерева інкапсулює логіку роботи з вузлами, починаючи з кореня, та надає інтерфейс для основних операцій.

Нижче наведено повні реалізації операцій пошуку, вставки та видалення.

```
Псевдокод
Пошук:
function Search(root, key)
  if root is null or root.key == key
    return root
  if key < root.key
    return Search(root.left, key)
  else
    return Search(root.right, key)
Вставка:
function Insert(root, key)
  if root is null
    return new Node(key)
  if key < root.key
    root.left = Insert(root.left, key)
  else if key > root.key
```

```
    root.right = Insert(root.right, key)
  return root
Видалення:
function Delete(root, key)
  if root is null
    return root

  if key < root.key
    root.left = Delete(root.left, key)
  else if key > root.key
    root.right = Delete(root.right, key)
  else
    // Вузол з одним або без нащадків
    if root.left is null
      temp = root.right
      free(root)
      return temp
    else if root.right is null
      temp = root.left
      free(root)
      return temp

    // Вузол з двома нащадками
    temp = MinValueNode(root.right)
    root.key = temp.key
    root.right = Delete(root.right, temp.key)

  return root
```

```
Python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert_recursive(self.root, key)
```

```
def _insert_recursive(self, root, key):
    if root is None:
        return Node(key)
    if key < root.key:
        root.left = self._insert_recursive(root.left, key)
    else:
        root.right = self._insert_recursive(root.right, key)
    return root

def search(self, key):
    return self._search_recursive(self.root, key)

def _search_recursive(self, root, key):
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self._search_recursive(root.left, key)
    return self._search_recursive(root.right, key)

def delete(self, key):
    self.root = self._delete_recursive(self.root, key)

def _min_value_node(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current

def _delete_recursive(self, root, key):
    if root is None:
        return root

    if key < root.key:
        root.left = self._delete_recursive(root.left, key)
    elif key > root.key:
        root.right = self._delete_recursive(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
```

```
            temp = self._min_value_node(root.right)
            root.key = temp.key
            root.right = self._delete_recursive(root.right, temp.key)

        return root

    def inorder(self):
        self._inorder_recursive(self.root)
        print()

    def _inorder_recursive(self, root):
        if root:
            self._inorder_recursive(root.left)
            print(root.key, end=" ")
            self._inorder_recursive(root.right)

# Приклад використання
bst = BST()
keys =
for key in keys:
    bst.insert(key)

print("Inorder traversal:")
bst.inorder()

print("\nDeleting 20")
bst.delete(20)
bst.inorder()

print("\nDeleting 70")
bst.delete(70)
bst.inorder()
```

```
C++
#include<iostream>

struct Node {
    int key;
    Node *left, *right;
};
```

```
Node* createNode(int key) {
    Node* newNode = new Node();
    newNode->key = key;
    newNode->left = newNode->right = nullptr;
    return newNode;
}

Node* insert(Node* root, int key) {
    if (root == nullptr) {
        return createNode(key);
    }
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}

Node* search(Node* root, int key) {
    if (root == nullptr |

| root->key == key) {
        return root;
    }
    if (key < root->key) {
        return search(root->left, key);
    }
    return search(root->right, key);
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left!= nullptr) {
        current = current->left;
    }
    return current;
}
```

```
Node* deleteNode(Node* root, int key) {
    if (root == nullptr) {
        return root;
    }
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

void inorder(Node* root) {
    if (root!= nullptr) {
        inorder(root->left);
        std::cout << root->key << " ";
        inorder(root->right);
    }
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```

```
    std::cout << "Inorder traversal: ";
    inorder(root);
    std::cout << std::endl;

    std::cout << "Deleting 20" << std::endl;
    root = deleteNode(root, 20);
    inorder(root);
    std::cout << std::endl;

    std::cout << "Deleting 70" << std::endl;
    root = deleteNode(root, 70);
    inorder(root);
    std::cout << std::endl;

    return 0;
}
```

## 5.6. Вирішення проблем, пов'язаних з незбалансованістю

Щоб продемонструвати практичне застосування концепцій самобалансування, розглянемо повну реалізацію AVL-дерева. Ця реалізація розширює базовий BST, додаючи механізми для відстеження висоти, виявлення дисбалансу та виконання поворотів.

Вузол AVL-дерева є розширенням вузла BST. Він додатково зберігає висоту піддерева, коренем якого він є. Це дає змогу ефективно обчислювати коефіцієнт балансу.

```
Python
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1 # Висота нового вузла (листка) - 1
```

```
C++
class Node {
public:
    int key;
    Node *left;
    Node *right;
    int height;
};
```

Операції insert та delete починаються так само, як і в звичайному BST. Однак після модифікації дерева вони проходять шлях від зміненого вузла до кореня, оновлюючи висоти та перевіряючи коефіцієнти балансу. Якщо виявлено дисбаланс (коефіцієнт балансу –2 або +2), виконуються відповідні повороти.

```
Python
import sys

class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def getHeight(self, root):
        if not root:
            return 0
        return root.height

    def getBalance(self, root):
        if not root:
            return 0
        return self.getHeight(root.left) - self.getHeight(root.right)

    def leftRotate(self, z):
        y = z.right
        T2 = y.left
```

```
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def rightRotate(self, z):
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def insert(self, root, key):
        if not root:
            return AVLNode(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

       root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))
        balance = self.getBalance(root)

        # LL Case
        if balance > 1 and key < root.left.key:
            return self.rightRotate(root)
        # RR Case
        if balance < -1 and key > root.right.key:
            return self.leftRotate(root)
        # LR Case
        if balance > 1 and key > root.left.key:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
        # RL Case
        if balance < -1 and key < root.right.key:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)
```

```
        return root

    def getMinValueNode(self, root):
        if root is None or root.left is None:
            return root
        return self.getMinValueNode(root.left)

    def delete(self, root, key):
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            temp = self.getMinValueNode(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

        if root is None:
            return root

        root.height = 1 + max(self.getHeight(root.left), self.getHeight(root.right))
        balance = self.getBalance(root)

        # LL Case
        if balance > 1 and self.getBalance(root.left) >= 0:
            return self.rightRotate(root)
        # RR Case
        if balance < -1 and self.getBalance(root.right) <= 0:
```

```
            return self.leftRotate(root)
        # LR Case
        if balance > 1 and self.getBalance(root.left) < 0:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
        # RL Case
        if balance < -1 and self.getBalance(root.right) > 0:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    def preOrder(self, root):
        if not root:
            return
        print("{0} ".format(root.key), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)

# Приклад використання
myTree = AVLTree()
root = None
keys =
for key in keys:
    root = myTree.insert(root, key)

print("Preorder traversal of the constructed AVL tree is")
myTree.preOrder(root)
print()

root = myTree.delete(root, 10)

print("Preorder traversal after deletion of 10")
myTree.preOrder(root)
print()
```

```
C++
#include<iostream>
#include<algorithm>

class Node {
public:
    int key;
    Node *left;
    Node *right;
    int height;
};

int height(Node *n) {
    if (n == nullptr)
        return 0;
    return n->height;
}

Node* newNode(int key) {
    Node* node = new Node();
    node->key = key;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}

Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = std::max(height(y->left), height(y->right)) + 1;
    x->height = std::max(height(x->left), height(x->right)) + 1;
    return x;
}

Node *leftRotate(Node *x) {
    Node *y = x->right;
```

```
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = std::max(height(x->left), height(x->right)) + 1;
    y->height = std::max(height(y->left), height(y->right)) + 1;
    return y;
}

int getBalance(Node *n) {
    if (n == nullptr)
        return 0;
    return height(n->left) - height(n->right);
}

Node* insert(Node* node, int key) {
    if (node == nullptr)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node;

    node->height = 1 + std::max(height(node->left), height(node->right));
    int balance = getBalance(node);

    // LL Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    // RR Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    // LR Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```
    // RL Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

void preOrder(Node *root) {
    if(root!= nullptr) {
        std::cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main() {
    Node *root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    std::cout << "Preorder traversal of the constructed AVL tree is \n";
    preOrder(root);
    std::cout << std::endl;

    return 0;
}
```

# 5.7. B-дерева

Досі ми розглядали дерева, оптимізовані для роботи в основній пам'яті (RAM), де вартість доступу до будь-якого елемента є приблизно однаковою. Однак, коли обсяг даних перевищує розмір RAM і їх доводиться зберігати на зовнішніх носіях, таких як жорсткі диски[22] (HDD) або твердотільні накопичувачі[23] (SSD), правила гри змінюються.

---

[22] Жорсткий диск (HDD, Hard Disk Drive) – це енергозалежний накопичувач даних на магнітних дисках, який використовується для довготривалого зберігання інформації у комп'ютерах та серверах.

Принцип роботи. Всередині HDD містяться жорсткі магнітні пластини (покриті феромагнітним шаром), що обертаються з високою швидкістю (зазвичай 5400–7200 об/хв у ПК, до 15 000 об/хв у серверів). Голівки зчитування/запису розташовані над поверхнею пластин і зчитують або змінюють намагніченість ділянок поверхні (бітів). Інформація кодується у вигляді 0 та 1.

Сучасні HDD мають обсяг від сотень гігабайтів до десятків терабайтів (10–20 ТБ і більше).Час доступу: ~ 5–15 мс. Швидкість передачі даних: 100–250 МБ/с. Значно повільніші за SSD через механічні рухомі частини. Інтерфейси підключення: SATA (найпоширеніший у ПК), SAS (у серверах). Переваги HDD: висока ємність за відносно низьку ціну; довготривале зберігання великих обсягів даних; добре підходять для архівів і резервних копій. Недоліки: механічні частини → чутливі до ударів і вібрацій; повільніший доступ до даних порівняно з SSD; більше енергоспоживання та шум. Використання: персональні комп'ютери та ноутбуки (як додаткове сховище); сервери й датацентри (архіви, великі бази даних); зовнішні HDD для резервного копіювання.

[23] Твердотільні накопичувачі (Solid State Drives, SSD) – це тип пристроїв зберігання даних, які зберігають інформацію на інтегрованих мікросхемах флеш-пам'яті, на відміну від традиційних жорстких дисків, що використовують магнітні пластини, що обертаються. SSD значно швидші за HDD. Операційні системи та програми завантажуються набагато швидше. Передача файлів відбувається значно швидше. Це пов'язано з тим, що SSD не мають рухомих частин і можуть миттєво отримувати доступ до будь-якої ділянки пам'яті.

Відсутність рухомих частин дає такі переваги: менш схильні до фізичних пошкоджень від ударів або падінь; працюють абсолютно безшумно; споживають менше енергії, що важливо для ноутбуків (збільшується час роботи від батареї) та серверів.

Можуть бути дуже компактними, що дає змогу використовувати їх у тонких ноутбуках та інших мініатюрних пристроях. Існують різні форм-фактори, такі як 2.5-дюймові (схожі на ноутбучні HDD), M.2 (маленькі плати, що вставляються безпосередньо в материнську плату) та NVMe (ще швидші, використовують інтерфейс PCIe).

Хоча ціна за гігабайт все ще вища, ніж у HDD, розрив постійно зменшується, і SSD стають все доступнішими.

SSD стали стандартом для сучасних комп'ютерів завдяки їхній високій продуктивності, надійності та низькому енергоспоживанню. Вони значно покращують загальний досвід використання комп'ютера, особливо під час роботи з операційною системою та програмами.

Ієрархія пам’яті комп’ютера складається з кількох рівнів: від швидких, але малих кешів процесора, до повільної, але великої дискової пам’яті. Доступ до даних на диску є на порядки повільнішим за доступ до RAM. Дані з диска читаються не побайтно, а цілими блоками (або сторінками), розмір яких зазвичай становить кілька кілобайт (наприклад, 4 KB).

Бінарні дерева, з їхньою низькою розгалуженістю (максимум два нащадки), є неефективними для дискових структур. Високе і «худе» бінарне дерево вимагатиме великої кількості дискових операцій читання для проходження шляху від кореня до листка. Кожен перехід до нащадка може потенційно вимагати читання нового блоку з диска.

B-дерева були розроблені спеціально для вирішення цієї проблеми. Вони є структурами, оптимізованими для роботи з великими обсягами даних на блокових пристроях зберігання, мінімізуючи кількість дорогих дискових операцій вводу-виводу.

B-дерево є узагальненням бінарного дерева пошуку, де кожен вузол може мати значно більше двох нащадків. Це робить дерево «низьким і широким», що є ключем до його ефективності.

Порядок (Order) $m$ B-дерева визначає максимальну кількість нащадків, яку може мати вузол.Ключові властивості B-дерева порядку $m$:

1. Кожен вузол має не більше m нащадків.
2. Кожен внутрішній вузол (крім кореня) має щонайменше $\lceil m/2 \rceil$ нащадків.
3. Корінь має щонайменше 2 нащадки, якщо він не є листком.
4. Внутрішній вузол з k нащадками містить $k$-1 ключів. Ключі у вузлі впорядковані.
5. Усі листки містяться на одному рівні. Це означає, що B-дерево завжди є ідеально збалансованим за висотою.

Ці властивості гарантують, що вузли заповнені щонайменше наполовину, що запобігає марнуванню простору і підтримує баланс дерева.

Пошук у B-дереві є узагальненням пошуку в BST.

1. Починаючи з кореня, виконується пошук (лінійний або двійковий) серед ключів поточного вузла, щоб знайти шуканий ключ або визначити, в яке піддерево спускатися далі.

2. Якщо ключ знайдено, пошук завершується.

3. Якщо ключ не знайдено, алгоритм переходить до відповідного дочірнього вузла (вказівник між двома ключами, що обмежують шукане значення) і повторює крок 1.

4. Якщо досягнуто листка і ключ не знайдено, його немає в дереві.

Вставка завжди відбувається в листковому вузлі.

1. Виконується пошук, щоб знайти відповідний листковий вузол для нового ключа.

2. Якщо у вузлі є вільне місце (кількість ключів $< m - 1$), ключ просто вставляється у вузол зі збереженням порядку.

3. Якщо вузол заповнений, відбувається операція розділення (split):

3.1. Вузол розділяється на два.

3.2. Середній (медіанний) ключ «просувається» (promoted) до батьківського вузла.

3.3. Ключі, менші за медіанний, залишаються в старому вузлі, а більші − переміщуються в новий.

3.4. Цей процес може поширюватися рекурсивно вгору до кореня. Якщо розділяється корінь, створюється новий корінь, і висота дерева збільшується на одиницю.

Псевдокод для розділення вузла (splitChild):

```
procedure splitChild(Node parent, int index, Node child_to_split)
    // child_to_split - повний дочірній вузол parent на індексі index
    // t - мінімальний ступінь (кількість ключів у вузлі від t-1 до 2t-1)

    Node new_sibling = new Node(child_to_split.is_leaf)
    new_sibling.num_keys = t - 1

    // Копіюємо t-1 "верхніх" ключів зі старого вузла в новий
    for j from 0 to t - 2
        new_sibling.keys[j] = child_to_split.keys[j + t]

    // Якщо вузол не є листком, копіюємо відповідних нащадків
    if not child_to_split.is_leaf
        for j from 0 to t - 1
            new_sibling.children[j] = child_to_split.children[j + t]
```

```
        child_to_split.num_keys = t - 1

        // Зсуваємо нащадків у батьківському вузлі, щоб звільнити місце для нового
        for j from parent.num_keys down to index + 1
            parent.children[j + 1] = parent.children[j]

        parent.children[index + 1] = new_sibling

        // Зсуваємо ключі у батьківському вузлі
        for j from parent.num_keys - 1 down to index
            parent.keys[j + 1] = parent.keys[j]

        // Просуваємо медіанний ключ до батька
        parent.keys[index] = child_to_split.keys[t - 1]
        parent.num_keys = parent.num_keys + 1
```

Видалення є складнішим, оскільки може порушити властивість мінімальної кількості ключів у вузлі.

1. Знайти ключ для видалення.

2. Якщо ключ у листку:

2.1. Якщо видалення не порушує мінімальну кількість ключів, просто видалити його.

2.2. Якщо вузол стає недостатньо заповненим (underflow), виконати запозичення (borrowing) у сусіднього вузла (якщо у нього є «зайві» ключі) або злиття (merging) з сусіднім вузлом.

3. Якщо ключ у внутрішньому вузлі:

3.1. Замінити ключ його in-order наступником або попередником (який завжди міститься в листку).

3.2. Рекурсивно видалити цього наступника / попередника з листка, що зводить задачу до випадку 2.

Процеси запозичення та злиття можуть поширюватися вгору до кореня.

Властивості B-дерева не є випадковими; вони є прямим результатом адаптації до фізичних обмежень апаратного забезпечення. «Вузол» у B-дереві концептуально відповідає одній «сторінці» або «блоку» на диску. Високий порядок m дає змогу вмістити

багато ключів в один блок. Коли система читає вузол з диска, вона за одну дорогу операцію вводу-виводу отримує великий набір ключів та вказівників. Подальший пошук серед цих ключів відбувається в швидкій оперативній пам'яті. Так, «низька і широка» структура B-дерева мінімізує кількість переходів між вузлами, що безпосередньо транслюється в мінімізацію кількості дискових звернень. Це є яскравим прикладом того, як дизайн структури даних повинен враховувати реалії апаратної архітектури для досягнення оптимальної продуктивності.

## 5.8. Консистентне хешування

Переходячи від структур даних на одному комп'ютері до розподілених систем, ми стикаємося з новою проблемою: як ефективно розподілити дані між багатьма серверами (вузлами) та підтримувати цей розподіл за зміни їх кількості. Консистентне хешування є алгоритмічним рішенням цієї проблеми.

У розподілених системах, таких як кеші (Memcached, Redis) або бази даних NoSQL (Cassandra, DynamoDB), дані потрібно розподілити (шардувати) по кластеру серверів. Найпростіший підхід − використовувати стандартну хеш-функцію з операцією по модулю: server_index = hash(key) % N, де N − кількість серверів.

Цей підхід добре працює, поки N є сталим. Однак, якщо додати новий сервер (N→N + 1) або видалити існуючий (N→N − 1), значення N змінюється, і hash(key) % N дасть зовсім інший результат для майже всіх ключів. Це призведе до масового переміщення даних по всьому кластеру, що є надзвичайно ресурсоємною та руйнівною операцією, відомою як «каскадне перехешування».

Консистентне хешування вирішує цю проблему, відокремлюючи хешування від кількості серверів. Алгоритм працює так:

1. Створення хеш-простору. Визначається великий діапазон цілих чисел, який уявляється у вигляді кільця (наприклад, від 0 до 232 − 1).

2. Розміщення серверів. Кожен сервер отримує одне або кілька місць на цьому кільці. Його ідентифікатор (наприклад, IP-адреса) хешується за допомогою тієї самої хеш-функції, і отримане значення визначає його позицію на кільці.

3. Розміщення даних. Щоб визначити, на якому сервері зберігати дані з певним ключем, ключ також хешується, отримуючи позицію на кільці.

4. Пошук сервера. Від позиції ключа на кільці починається рух за годинниковою стрілкою, і перший зустрінутий сервер є тим, що відповідає за цей ключ. Отже, кожен сервер відповідає за дугу кільця від свого попередника до власної позиції.

Елегантність цього підходу проявляється за зміни кількості серверів.

Коли новий сервер додається на кільце, він «перехоплює» відповідальність за частину ключів у свого наступника за годинниковою стрілкою. Тільки дані, що належать до нової дуги (від нового сервера до його попередника), потрібно перемістити. Дані на всіх інших серверах залишаються недоторканими.

Коли сервер видаляється з кільця, всі ключі, за які він відповідав, передаються його наступнику за годинниковою стрілкою. Знову ж таки, зміна є локальною і не зачіпає більшість серверів у системі.

Так, консистентне хешування гарантує, що за зміни кількості серверів на k лише приблизно 1/k ключів потребують перерозподілу.

Просте розміщення серверів на кільці може призвести до нерівномірного розподілу даних, якщо сервери випадково згрупуються в одній частині кільця. Щоб вирішити цю проблему, використовується техніка *віртуальних вузлів.* Кожен фізичний сервер представляється не однією, а багатьма (сотнями) «віртуальними» точками на кільці. Це значно згладжує розподіл даних і гарантує, що навантаження на фізичні сервери буде більш рівномірним.

Хоча «кільце» є зручною концептуальною моделлю, на практиці виникає питання: як ефективно знайти «наступний сервер за годинниковою стрілкою»? Маючи набір хеш-значень серверів, для даного хешу ключа потрібно знайти найменше хеш-значення сервера, яке є більшим за хеш ключа. Це класична задача пошуку. Її можна вирішити, зберігаючи відсортований масив хешів серверів і використовуючи двійковий пошук. Ще ефективнішим рішенням є зберігання хеш-значень серверів у збалансованому бінарному дереві пошуку, наприклад, у червоно-чорному дереві. Пошук наступника (successor) у такому дереві дає потрібний сервер за час O(logS), де S − кількість (віртуальних) серверів. Це демонструє, як фундаментальні принципи, розроблені для дерев, знаходять застосування у вирішенні складних проблем у розподілених системах.

## 5.9. Огляд B-дерев та їх варіантів

B-дерево є основою для цілого сімейства структур даних, оптимізованих для дискового зберігання. Найважливішими його варіантами є B+ дерево та B* дерево, кожне з яких пропонує свої вдосконалення.

B+ дерево є найпоширенішим варіантом B-дерева і стандартом де-факто для індексів у сучасних реляційних базах даних (наприклад, MySQL, PostgreSQL) та файлових системах.

1. Зберігання даних. У B+ дереві всі дані (або вказівники на них) зберігаються виключно в листкових вузлах. Внутрішні вузли містять лише копії ключів, які використовуються як «дороговкази» для навігації по дереву. У звичайному B-дереві дані можуть зберігатися як у внутрішніх, так і в листкових вузлах.

2. Зв’язані листки. Усі листкові вузли в B+ дереві з’єднані між собою у двобічно зв’язаний список. Це дає змогу здійснювати послідовний обхід усіх даних без необхідності повертатися до батьківських вузлів.

Переваги B+ дерева:

1. Вища розгалуженість. Оскільки внутрішні вузли не зберігають даних, вони можуть вмістити більше ключів-вказівників.

Це збільшує коефіцієнт розгалуження (fanout) і, як наслідок, зменшує висоту дерева, що призводить до меншої кількості дискових операцій для пошуку.

2. Ефективність діапазонних запитів. Зв’язана структура листків робить B+ дерева надзвичайно ефективними для діапазонних запитів (наприклад, SELECT * FROM users WHERE age BETWEEN 25 AND 35). Після знаходження першого елемента діапазону решту можна швидко отримати, просто рухаючись по зв’язаному списку листків. У B-дереві такий запит вимагав би багаторазових спусків по дереву.

3. Спрощені операції. Операції вставки та видалення стають дещо простішими, оскільки всі модифікації даних відбуваються на рівні листків.

B* дерево − це варіант B-дерева, що фокусується на кращому використанні простору за рахунок підтримки вищої щільності заповнення вузлів.

Структура та ключові відмінності:

1. Мінімальне заповнення. B* дерево вимагає, щоб кожен внутрішній вузол (крім кореня) був заповнений щонайменше на 2/3, на відміну від 1/2 у B-дереві.

2. Процес вставки. Щоб підтримувати вищу щільність, коли вузол стає повним, він не одразу розділяється. Спочатку він намагається перерозподілити (spill) свої ключі з сусіднім вузлом. Лише якщо сусідній вузол також повний, відбувається розділення. Водночас два повні вузли розділяються на три вузли, а не на два, як у B-дереві.

Переваги B-дерева:*

1. Краще використання простору. Вимоги до щільнішого заповнення зменшують кількість невикористаного простору в вузлах.

2. Менша висота. Більш щільні вузли можуть призвести до ще меншої висоти дерева порівняно зі звичайним B-деревом.

Таблиця 5.4 – Порівняння B-дерев, B+ дерев та B* дерев

| Характеристика | B-дерево | B+ дерево | B-дерево* |
|---|---|---|---|
| Зберігання даних | Внутрішні та листкові вузли | Тільки листкові вузли | Внутрішні та листкові вузли |
| Структура листкових вузлів | Не зв’язані | Зв’язані у двобічний список | Не зв’язані |
| Ефективність для точкових запитів | Висока (можливе раннє завершення у внутрішньому вузлі) | Висока (завжди доходить до листка) | Висока |
| Ефективність для діапазонних запитів | Низька (вимагає обходу дерева) | Дуже висока (лінійний прохід по листках) | Низька |
| Мінімальне заповнення вузла | 1/2 | 1/2 | 2/3 |
| Процес вставки (під час переповнення) | Розділення 1 вузла на 2 | Розділення 1 вузла на 2 | Перерозподіл із сусідом; якщо неможливо, розділення 2 вузлів на 3 |

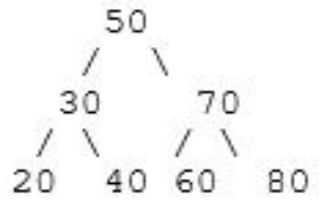
## 5.10. Приклади розв’язку задач

**Задача 1.** Побудова BST та обходи. Дано послідовність чисел. Побудуйте бінарне дерево пошуку та виведіть результати прямого, симетричного та зворотного обходів.

*Розв’язок*

1. Побудова дерева:

- 50 стає коренем.
- 30 < 50 → лівий нащадок 50.
- 70 > 50 → правий нащадок 50.
- 20 < 50, 20 < 30 → лівий нащадок 30.
- 40 < 50, 40 > 30 → правий нащадок 30.
- 60 > 50, 60 < 70 → лівий нащадок 70.
- 80 > 50, 80 > 70 → правий нащадок 70.




2. Обходи:

− прямий (pre-order): 50, 30, 20, 40, 70, 60, 80.

− симетричний (in-order): 20, 30, 40, 50, 60, 70, 80.

− зворотний (post-order): 20, 40, 30, 60, 80, 70, 50.

**Задача 2.** Видалення з BST. З дерева, побудованого в Задачі 1, видаліть вузол з ключем 30.

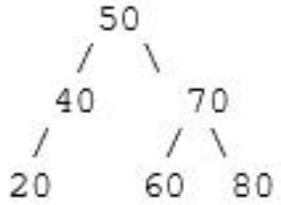
*Розв'язок*

1. Вузол 30 має двох нащадків (20 і 40).

2. Знаходимо його in-order наступника (найменший елемент у правому піддереві). Праве піддерево 30 − це вузол 40. Найменший елемент у цьому піддереві −40.

3. Копіюємо ключ 40 у вузол, який ми видаляємо.

4. Видаляємо вузол 40 з його початкового місця. Оскільки 40 є листком, він просто видаляється.




**Задача 3.** Балансування AVL-дерева. Послідовно вставте ключі в порожнє AVL-дерево. Покажіть кроки балансування.

*Розв'язок*

Вставка 10:10 стає коренем. Дерево збалансоване.

10 (BF = 0)

Вставка 20:20 > 10, стає правим нащадком.

10 (BF = -1)

\

20 (BF = 0)

Дерево все ще збалансоване.

Вставка 30:30 > 20, стає правим нащадком 20.

```
10 (BF = -2) <-- Дисбаланс!
  \
 20 (BF = -1)
   \
    30 (BF = 0)
```

Вузол 10 має коефіцієнт балансу -2. Це випадок Right-Right (RR).

Балансування. Виконуємо один лівий поворот навколо незбалансованого вузла 10.

```
  20 (BF = 0)
 /  \
 10   30
 (BF = 0) (BF = 0)
```

Дерево знову збалансоване.

**Задача 4.** Вставка в B-дерево. Дано B-дерево порядку m = 3 (мінімальний ступінь t = 2, макс. ключів 2t-1 = 3, мін. ключів t-1 = 1). Вставте ключ 9.

Початкове дерево:

```
    [ 6 ]
   /     \
```

*Розв'язок*

1. Починаємо пошук місця для 9 з кореня . `9 > 6`, тому спускаємося в праве піддерево.
2. Вузол є листком. Намагаємося вставити `9`. Вузол стає.
3. Кількість ключів (3) дорівнює максимальній m-1 = 2 (тут помилка в умові, для m = 3, max_keys = 2). Припустимо m = 4, t = 2, max_keys = 3. Тоді вузол `` заповнений.
4. Розділення (Split):

- Вузол `` розділяється.
- Медіанний ключ 8 просувається до батьківського вузла ``.
- Батьківський вузол стає ``.
- Ключ 7 залишається в лівому вузлі, а 9 − у новому правому.

Фінальне дерево:

```
[ 6, 8 ]
/|\
```

## Контрольні питання

1. Що таке дерево в контексті структур даних і чим воно відрізняється від лінійних структур?

2. Назвіть та опишіть три основні типи обходу бінарного дерева в глибину. Для яких завдань кожен з них є найбільш прийнятним?

3. Сформулюйте основну властивість бінарного дерева пошуку (BST). Чому симетричний обхід BST повертає елементи у відсортованому порядку?

4. Опишіть три випадки видалення вузла з бінарного дерева пошуку. Який випадок є найскладнішим і як він вирішується?

5. Що таке деградація бінарного дерева пошуку і за яких умов вона виникає? Як це впливає на асимптотичну складність операцій?

6. Яку проблему вирішують дерева, що самобалансуються? Назвіть два приклади таких дерев.

7. Що таке коефіцієнт балансу в AVL-дереві і які значення він може приймати? Що відбувається, коли його значення виходить за межі допустимого діапазону?

8. Назвіть п’ять основних властивостей червоно-чорного дерева. Як ці властивості гарантують, що дерево залишається збалансованим?

9. Порівняйте AVL-дерева та червоно-чорні дерева. У яких сценаріях доцільніше використовувати кожне з них і чому?

10. Яка основна мотивація для використання B-дерев замість бінарних дерев у системах баз даних та файлових системах?

11. Опишіть процес вставки ключа в B-дерево, охоплюючи випадок, коли вузол переповнюється і потребує розділення.

12. Чим B+ дерево відрізняється від B-дерева? Які переваги надає структура B+ дерева для діапазонних запитів?

13. Що таке консистентне хешування і яку проблему воно вирішує в розподілених системах?

14. Поясніть концепцію «хеш-кільця» і як вона допомагає мінімізувати перерозподіл даних під час додавання або видалення серверів.

15. Для чого використовуються «віртуальні вузли» в консистентному хешуванні?

## Задачі для самостійного опрацювання

1. Напишіть функцію, яка перевіряє, чи є два бінарні дерева ідентичними (мають однакову структуру та однакові значення у відповідних вузлах).

2. Реалізуйте ітеративну версію зворотного обходу (post-order) бінарного дерева, використовуючи один стек.

3. Напишіть функцію для знаходження висоти бінарного дерева.

4. Напишіть функцію, яка перевіряє, чи є задане бінарне дерево валідним бінарним деревом пошуку.

5. Реалізуйте функцію для знаходження k-го найменшого елемента в бінарному дереві пошуку.

6. Напишіть функцію, яка перетворює відсортований масив на збалансоване бінарне дерево пошуку.

7. Реалізуйте функцію для знаходження найнижчого спільного предка (Lowest Common Ancestor, LCA) для двох вузлів у бінарному дереві пошуку.

8. Напишіть програму, яка демонструє деградацію BST за вставки відсортованої послідовності та порівнює час пошуку елемента в ньому з часом пошуку в збалансованому дереві (наприклад, AVL), побудованому з тих самих даних.

9. Реалізуйте функцію видалення для червоно-чорного дерева, охоплюючи всі випадки виправлення порушених властивостей.

10. Напишіть функцію, яка серіалізує бінарне дерево в рядок та десеріалізує рядок назад у дерево.

11. Реалізуйте B-дерево порядку 5 та операції вставки й пошуку. Продемонструйте роботу на прикладі, що охоплює щонайменше одне розділення вузла.

12. Напишіть функцію, яка знаходить всі ключі в B-дереві, що містяться в заданому діапазоні [min_key, max_key].

13. Спроєктуйте та реалізуйте простий симулятор консистентного хешування. Програма повинна дозволяти додавати та видаляти сервери з кільця та показувати, як ключі перерозподіляються між серверами.

14. Теоретична задача: доведіть, що висота червоно-чорного дерева з n внутрішніми вузлами не перевищує $2\log_2(n + 1)$.

15. Порівняйте теоретичну продуктивність пошуку в B-дереві та B+ дереві для точкових та діапазонних запитів. Поясніть, чому B+ дерева є кращими для індексів баз даних.

# ТЕМА 6
# КУПИ ТА ЧЕРГИ З ПРІОРИТЕТОМ

Абстрактні типи даних слугують фундаментальними концептуальними моделями для організації та маніпулювання даними. На відміну від конкретних структур даних, які визначають *як* дані зберігаються та обробляються, АТД фокусуються на *що* − на семантиці операцій, які можна виконувати. Серед цих концептуальних інструментів особливе місце займає черга з пріоритетом (priority queue).

На відміну від простіших структур, таких як звичайна черга (що працює за принципом «перший прийшов − перший вийшов», FIFO) або стек («останній прийшов − перший вийшов», LIFO), черга з пріоритетом упорядковує елементи на основі їхньої внутрішньої ваги або пріоритету. Кожен елемент, що додається до черги, має асоційований з ним пріоритет. Коли надходить запит на вилучення елемента, структура завжди повертає той, що має найвищий пріоритет.

Формально, черга з пріоритетом − це АТД, що підтримує щонайменше такі операції:

− insert(element, priority) − додає новий елемент до колекції з відповідним пріоритетом;

− find-min() (або find-max()) − повертає елемент з найвищим пріоритетом, не видаляючи його з черги, за домовленістю, «найвищий» пріоритет може означати або найменше числове значення (мін-черга), або найбільше (макс-черга).

− extract-min() (або extract-max()) − видаляє та повертає елемент з найвищим пріоритетом.

Щоб зрозуміти потребу в спеціалізованих структурах для реалізації цього АТД, розглянемо кілька простих підходів:

1. Невпорядкований масив або список. Операція insert виконується за константний час O(1), просто додаючи елемент в кінець. Однак, щоб знайти та видалити елемент з найвищим пріоритетом, потрібно переглянути всю колекцію, що вимагає лінійного часу O($n$).

2. Впорядкований масив або список. Операції find-min та extract-min виконуються за час O(1), оскільки елемент з найвищим пріоритетом завжди міститься на початку або в кінці. Проте операція insert вимагає знаходження правильної позиції та зсуву елементів, що займає час O(*n*).

Очевидно, що для застосувань, де операції вставки та вилучення виконуються часто, ці прості реалізації є неефективними. Саме тут на сцену виходить купа (heap) − спеціалізована деревоподібна структура даних, що є канонічною та найефективнішою реалізацією черги з пріоритетом.

Важливо чітко розмежовувати ці два поняття. Черга з пріоритетом − це *інтерфейс*, набір правил та операцій. Купа − це *конкретна структура даних*, що реалізує цей інтерфейс. Подібно до того, як абстрактний тип «список» може бути реалізований за допомогою масиву або зв'язаного списку, так і «черга з пріоритетом» може бути реалізована різними способами, але саме купа забезпечує оптимальний баланс, гарантуючи логарифмічну часову складність O(*logn*) для ключових операцій вставки та вилучення. У цьому параграфі ми детально дослідимо різні види куп, починаючи з фундаментальної бінарної купи і закінчуючи більш складними структурами, що оптимізовані для специфічних операцій, таких як злиття.

## 6.1. Бінарні купи та операції над ними

Бінарна купа є найпростішою та найпоширенішою формою купи. Вона є основою для багатьох алгоритмів, охоплюючи сортування купою, та слугує ефективною реалізацією черги з пріоритетом.

Формально, бінарна купа − це бінарне дерево, що задовольняє дві фундаментальні властивості: структурну та порядкову.

1. Структурна властивість (Shape Property):купа повинна бути повним бінарним деревом. Це означає, що всі рівні дерева повністю заповнені, за винятком, можливо, останнього, який заповнюється зліва направо без пропусків. Ця строга структурна вимога є ключовою, оскільки вона гарантує, що висота дерева з n

вузлами завжди буде O(*logn*), і дає змогу ефективно представляти дерево у вигляді масиву.

2. Порядкова властивість (Heap Property): значення (ключ) у кожному вузлі повинно відповідати певному порядку відносно значень у його дочірніх вузлах. Існує два типи бінарних куп:

− мін-купа (min-heap) − ключ кожного вузла є *меншим або рівним* ключам його дочірніх вузлів. як наслідок, найменший елемент у всій купі завжди міститься в корені дерева;

− макс-купа (max-heap) − ключ кожного вузла є *більшим або рівним* ключам його дочірніх вузлів. відповідно, найбільший елемент завжди міститься в корені.

Якщо не вказано інше, у цьому параграфі для прикладів використовуватиметься мін-купа.

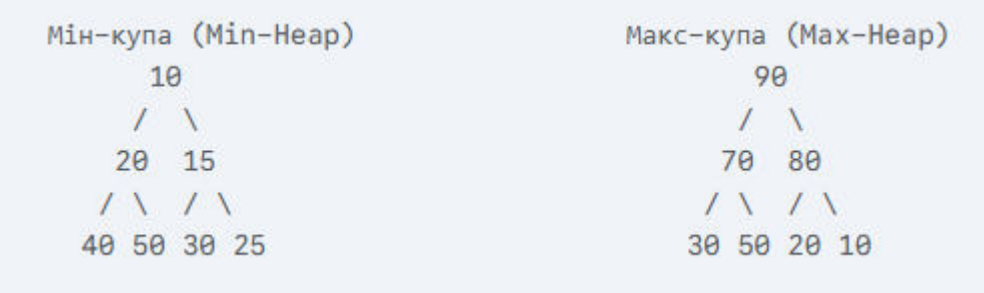



Рисунок 6.1 – Приклади мін-купи та макс-купи

У мін-купі кожен батьківський вузол менший за своїх нащадків. У макс-купі − більший.

Завдяки властивості повноти, бінарну купу можна дуже ефективно представити у вигляді звичайного масиву (або вектора) без використання вказівників чи інших складних структур для зв'язків між вузлами. Елементи дерева просто записуються в масив рівень за рівнем, зліва направо. За такого представлення (з 0-індексацією) зв'язки між батьківськими та дочірніми вузлами легко обчислюються за допомогою простих арифметичних формул. Для вузла, що міститься за індексом i в масиві, отримаємо:

− індекс його батька: $\lfloor (i - 1)/2 \rfloor$;

− індекс його лівого нащадка: $2 \cdot i + 1$;

− індекс його правого нащадка: $2 \cdot i + 2$.

Ці формули є основою для всіх алгоритмів, що маніпулюють купою, оскільки вони дають змогу здійснювати переміщення по дереву з часовою складністю O(1).

Таблиця 6.1 – Навігація в купі на основі масиву (0-індексація)

| Операція | Формула (для вузла за індексом i) |
|---|---|
| Знайти батька | $\lfloor (i-1)/2 \rfloor$ |
| Знайти лівого нащадка | $2 \cdot i + 1$ |
| Знайти правого нащадка | $2 \cdot i + 2$ |

Для підтримки порядкової властивості купи після вставки або видалення елементів використовуються дві допоміжні процедури: sift-up та sift-down.

1. Процедура sift-up (просіювання вгору). Ця операція використовується для відновлення властивості купи, коли ключ вузла стає *меншим* за ключ його батька (у мін-купі). Процедура полягає в послідовному обміні місцями цього вузла з його батьком, доки він не займе правильну позицію, тобто доки його ключ не стане більшим або рівним ключу батька, або доки вузол не стане коренем. Ця операція є ядром операції insert.

2. Процедура sift-down (просіювання вниз, також відома як heapify). Ця операція застосовується, коли ключ вузла стає *більшим* за ключ одного або обох його нащадків. Процедура полягає в послідовному обміні місцями цього вузла з його *найменшим* нащадком. Цей процес триває доти, доки вузол не стане меншим або рівним за обох своїх нащадків, або доки не досягне рівня листка. sift-down є ключовою для операцій extract-min та build-heap.

На основі sift-up та sift-down реалізуються основні операції черги з пріоритетом.

1. Операція insert(key) (вставка елемента).Новий елемент додається в першу вільну позицію в кінці масиву. Це зберігає структурну властивість повного бінарного дерева.До новододаного елемента застосовується процедура sift-up для відновлення порядкової властивості купи.Часова складність цієї операції визначається висотою дерева і становить O(logn).

2. Операція extract-min() (вилучення мінімального елемента):

− значення кореневого елемента (за індексом 0) зберігається для повернення;

− останній елемент масиву переміщується на місце кореня;

− розмір купи зменшується на одиницю (наприклад, видаленням останнього елемента з масиву);

− до нового кореня (за індексом 0) застосовується процедура sift-down для відновлення порядкової властивості.

Часова складність становить O(logn).

3. Операція get-min() (отримання мінімального елемента) − просто повертає значення елемента за індексом 0,її складність − O(1).

*Псевдокод*

```
class MinHeap:
    heap_array =
    size = 0

    function parent(i):
        return floor((i - 1) / 2)

    function left_child(i):
        return 2 * i + 1

    function right_child(i):
        return 2 * i + 2

    procedure sift_up(i):
        while i > 0 and heap_array[parent(i)] > heap_array[i]:
            swap(heap_array[parent(i)], heap_array[i])
            i = parent(i)

    procedure sift_down(i):
        minIndex = i
        l = left_child(i)
        if l < size and heap_array[l] < heap_array[minIndex]:
            minIndex = l
        r = right_child(i)
        if r < size and heap_array[r] < heap_array[minIndex]:
            minIndex = r
        if i!= minIndex:
            swap(heap_array[i], heap_array[minIndex])
            sift_down(minIndex)
```

```
procedure insert(key):
    add key to end of heap_array
    size = size + 1
    sift_up(size - 1)

function extract_min():
    if size == 0:
        return ERROR
    result = heap_array
    heap_array = heap_array[size - 1]
    size = size - 1
    remove last element from heap_array
    sift_down(0)
    return result

procedure build_heap(source_array):
    heap_array = copy of source_array
    size = length of source_array
    start_index = floor(size / 2) - 1
    for i from start_index down to 0:
        sift_down(i)
```

У Python для роботи з купами існує вбудований модуль heapq, який реалізує мін-купу на основі звичайного списку. Однак для повного розуміння структури корисно розглянути її реалізацію «з нуля» у вигляді класу.

```
Python
class MinHeap:
    def __init__(self):
        """
        Ініціалізує порожню купу. Використовується 0-індексація.
        """
        self.heap_list =

    def get_min(self):
        """Повертає мінімальний елемент без видалення."""
        return self.heap_list if self.heap_list else None
```

```
        def _sift_up(self, i):
            """Відновлює властивість купи, рухаючись вгору від індексу i."""
            parent_index = (i - 1) // 2
            while i > 0 and self.heap_list[i] < self.heap_list[parent_index]:
                self.heap_list[i], self.heap_list[parent_index] =
self.heap_list[parent_index], self.heap_list[i]
                i = parent_index
                parent_index = (i - 1) // 2

        def _sift_down(self, i):
            """Відновлює властивість купи, рухаючись вниз від індексу i."""
            max_index = len(self.heap_list) - 1
            while True:
                left_child_index = 2 * i + 1
                right_child_index = 2 * i + 2
                smallest = i

                if left_child_index <= max_index and self.heap_list[left_child_index] <
self.heap_list[smallest]:
                    smallest = left_child_index

                if right_child_index <= max_index and self.heap_list[right_child_index]
< self.heap_list[smallest]:
                    smallest = right_child_index

                if smallest!= i:
                    self.heap_list[i], self.heap_list[smallest] = self.heap_list[smallest],
self.heap_list[i]
                    i = smallest
                else:
                    break

        def insert(self, key):
            """Додає новий елемент до купи."""
            self.heap_list.append(key)
            self._sift_up(len(self.heap_list) - 1)

        def extract_min(self):
            """Видаляє та повертає мінімальний елемент."""
            if not self.heap_list:
                return None
            if len(self.heap_list) == 1:
                return self.heap_list.pop()
```

```
        min_val = self.heap_list
        self.heap_list = self.heap_list.pop()
        self._sift_down(0)
        return min_val

    def build_heap(self, arr):
        """Будує купу з існуючого масиву за лінійний час."""
        self.heap_list = list(arr)
        start_index = (len(self.heap_list) // 2) - 1
        for i in range(start_index, -1, -1):
            self._sift_down(i)
```

У C++ стандартна бібліотека (STL) надає std::priority_queue, що за замовчуванням є макс-купою. Для створення мін-купи потрібно вказати додаткові шаблонні параметри. Також доступні алгоритми std::make_heap, std::push_heap, std::pop_heap для маніпуляції будь-яким контейнером з довільним доступом, наприклад std::vector. Нижче наведено реалізацію мін-купи «з нуля» у вигляді шаблонного класу.

```
C++
#include<iostream>
#include<vector>
#include<stdexcept>
#include<algorithm> // Для std::swap

template <typename T>
class MinHeap {
private:
    std::vector<T> heap_vec;

    int parent(int i) { return (i - 1) / 2; }
    int left_child(int i) { return 2 * i + 1; }
    int right_child(int i) { return 2 * i + 2; }

    void sift_up(int i) {
        while (i > 0 && heap_vec[i] < heap_vec[parent(i)]) {
            std::swap(heap_vec[i], heap_vec[parent(i)]);
            i = parent(i);
        }
    }
```

```
    void sift_down(int i) {
        int min_index = i;
        int l = left_child(i);
        if (l < size() && heap_vec[l] < heap_vec[min_index]) {
            min_index = l;
        }
        int r = right_child(i);
        if (r < size() && heap_vec[r] < heap_vec[min_index]) {
            min_index = r;
        }
        if (i!= min_index) {
            std::swap(heap_vec[i], heap_vec[min_index]);
            sift_down(min_index);
        }
    }

public:
    MinHeap() = default;

    bool is_empty() const {
        return heap_vec.empty();
    }

    int size() const {
        return heap_vec.size();
    }

    void insert(const T& key) {
        heap_vec.push_back(key);
        sift_up(heap_vec.size() - 1);
    }

    T get_min() const {
        if (is_empty()) {
            throw std::out_of_range("Heap is empty");
        }
        return heap_vec.at(0);
    }
```

```
    T extract_min() {
        if (is_empty()) {
            throw std::out_of_range("Heap is empty");
        }
        T min_val = heap_vec;
        heap_vec = heap_vec.back();
        heap_vec.pop_back();
        if (!is_empty()) {
            sift_down(0);
        }
        return min_val;
    }

    void build_heap(const std::vector<T>& arr) {
        heap_vec = arr;
        int start_index = (heap_vec.size() / 2) - 1;
        for (int i = start_index; i >= 0; --i) {
            sift_down(i);
        }
    }
};
```

Одним з найбільш елегантних та контрінтуїтивних результатів в аналізі алгоритмів є доведення того, що бінарну купу з n елементів можна побудувати за лінійний час, тобто O($n$). Наївний підхід полягає у створенні порожньої купи та послідовній вставці n елементів. Оскільки кожна вставка коштує O($logn$), загальна складність такого методу становить O($nlogn$).

Однак існує значно ефективніший метод, відомий як алгоритм Флойда. Він працює «знизу-вгору» за таким принципом.

1. Вхідний невпорядкований масив розглядається як повне бінарне дерево, яке, ймовірно, порушує порядкову властивість купи.

2. Алгоритм ітерує у зворотному порядку, починаючи з останнього вузла, що не є листком (його індекс $\lfloor n/2 \rfloor - 1$), і рухаючись до кореня (індекс 0).

3. Для кожного з цих вузлів викликається процедура sift-down, яка відновлює властивість купи для піддерева з коренем у цьому вузлі.

Причина, чому цей метод працює за O(n), полягає в тому, що вартість операції sift-down залежить від висоти вузла, а не від загальної висоти дерева. У повному бінарному дереві переважна більшість вузлів міститься на нижніх рівнях, де їх висота мала.

Приблизно $n/2$ вузлів є листками (висота 0), для них sift-down не виконує жодної роботи. Приблизно $n/4$ вузлів містяться на висоті 1, і sift-down для них виконає щонайбільше 1 обмін. Приблизно n/8 вузлів містяться на висоті 2, і sift-down для них виконає щонайбільше 2 обміни. І так далі, аж до кореня, який єдиний має максимальну висоту ≈ logn.

Загальний обсяг роботи можна виразити як суму робіт, виконаних на кожному рівні. Кількість вузлів на висоті h від листя не перевищує $\lceil n/2^{h+1} \rceil$, а вартість sift-down для них пропорційна h. Так, загальна вартість T(n) обмежена сумою:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) = O\left( n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} \right)$$

Ряд

$$\sum_{h=0}^{\infty} \frac{h}{2^{h+1}}$$

є збіжним і дорівнює константі (зокрема, 1). Отже, загальна складність становить $O(n) \cdot 1 = O(n)$. Цей аналіз демонструє потужний принцип: амортизація вартості по всій структурі, замість припущення найгіршого випадку для кожного елемента, може виявити значно точнішу та ефективнішу межу складності. Вибір sift-down та обхід знизу-вгору є не довільним, а саме тим механізмом, що забезпечує лінійну продуктивність.

## 6.2. Застосування куп в алгоритмах

Завдяки своїй ефективності, купи та черги з пріоритетом є фундаментальними компонентами багатьох класичних алгоритмів, від сортування до пошуку шляхів у графах та обробки потокових даних.

Сортування купою − це елегантний та ефективний алгоритм сортування на основі порівнянь, який використовує властивості макс-купи для сортування масиву на місці (in-place). Алгоритм складається з двох основних фаз:

1. Фаза побудови (Build Phase). Вхідний невпорядкований масив перетворюється на макс-купу. Це робиться за допомогою ефективного алгоритму build-heap за час O(n). Після цього кроку найбільший елемент масиву гарантовано міститься в корені купи (за індексом 0).

2. Фаза сортування (Sortdown Phase). Ця фаза виконується ітеративно й охоплює чотири кроки:

2.1. Кореневий елемент (найбільший у поточній купі) міняється місцями з останнім елементом купи. Тепер найбільший елемент міститься на своєму фінальному відсортованому місці в кінці масиву.

2.2. Розмір купи віртуально зменшується на одиницю, вилучаючи щойно відсортований елемент.

2.3. Новий елемент у корені, ймовірно, порушує властивість макс-купи. Для її відновлення викликається sift-down для кореня.

2.4. Кроки a-c повторюються, доки розмір купи не стане рівним 1.9.

Хоча фаза побудови купи займає лише O(n), загальна складність алгоритму Heapsort становить O(nlogn). Це пов’язано з тим, що фаза сортування домінує в асимптотиці. Вона складається з n − 1 ітерацій, на кожній з яких виконується операція sift-down. Вартість sift-down пропорційна висоті купи, яка зменшується з logn до 1. На відміну від фази побудови, де sift-down викликається для вузлів з переважно малою висотою, у фазі сортування sift-down завжди викликається для кореня, де вартість є максимальною для поточної висоти купи. Сумарна вартість цих n − 1 операцій є O(nlogn), що і визначає загальну складність.

Heapsort має перевагу в тому, що він вимагає лише O(1) додаткової пам’яті (in-place), але він не є стабільним алгоритмом сортування.

Алгоритм Дейкстри знаходить найкоротші шляхи від однієї початкової вершини до всіх інших вершин у зваженому графі з невід'ємними вагами ребер. Його основна ідея − жадібно розширювати множину «відвіданих» вершин, на кожному кроці обираючи ще не відвідану вершину з найменшою відомою відстанню від початкової.

У базовій реалізації пошук вершини з мінімальною відстанню серед усіх невідвіданих вимагає лінійного сканування масиву відстаней, що займає час O(V), де V − кількість вершин. Оскільки цей пошук виконується V разів, загальна складність алгоритму становить $O(V^2)$.

Використання мін-черги з пріоритетом (реалізованої на основі мін-купи) кардинально оптимізує цей процес. Черга зберігає пари (відстань, вершина) для всіх вершин, до яких вже знайдено шлях, але які ще не були «фіналізовані».

Алгоритм з чергою пріоритетів.

1. Створюється черга пріоритетів. До неї додається початкова вершина з пріоритетом (відстанню) 0. Відстані до всіх інших вершин ініціалізуються нескінченністю.

2. Доки черга не порожня, виконується таке:

a) З черги вилучається вершина u з найменшою відстанню (extract-min). Ця операція тепер займає O(logV) часу.

b) Для кожного сусіда v вершини u виконується «релаксація»: якщо шлях до v через u коротший за поточну відому відстань до v, то відстань до v оновлюється.

c) Після оновлення відстані до v її пріоритет у черзі також оновлюється. Це може бути реалізовано через операцію decrease-key або, якщо вона не підтримується (як у std::priority_queue C++), через додавання нової пари (нова_відстань, v) до черги.

Аналіз складності.

1. Кожна з V вершин додається до черги та вилучається з неї один раз. Це дає V операцій extract-min, загальною вартістю O(VlogV).

2. Кожне з E ребер розглядається один раз. У найгіршому випадку для кожного ребра може відбутися релаксація, що призведе до операції decrease-key (або insert). Вартість кожної такої операції − O(logV). Загальна вартість для всіх ребер − O(ElogV).

3. Сумарна часова складність становить O(VlogV + ElogV) = O((V + E)logV). Для зв’язних графів, де E ≥ V − 1, це зазвичай спрощується до O(ElogV).[37]

Алгоритм Прима − це ще один жадібний алгоритм, який знаходить мінімальне кістякове дерево (Minimum Spanning Tree, MST) для зв’язного неорієнтованого графа. Алгоритм будує MST, починаючи з довільної вершини і поступово додаючи до дерева найдешевше ребро, яке з’єднує вершину, що вже є в дереві, з вершиною, якої ще немає.

Як і в алгоритмі Дейкстри, ключовим кроком є ефективний пошук цього «найдешевшого ребра». Мін-черга з пріоритетом ідеально підходить для цього завдання. Вона зберігає вершини, які ще не долучені до MST, а їхніми пріоритетами є ваги найлегших ребер, що з’єднують їх з вершинами, які вже є в MST.

Логіка роботи та аналіз складності дуже схожі на алгоритм Дейкстри: виконується V операцій extract-min та до E операцій decrease-key, що призводить до такої самої загальної часової складності O(ElogV) під час використання бінарної купи.

Купи є незамінними для задач, де дані надходять послідовно (потоком), і потрібно підтримувати певну статистику в реальному часі.

Задача пошуку медіани в потоці. Використовуються дві купи:

1. Макс-купа lower_half − зберігає меншу половину всіх отриманих чисел. Корінь цієї купи є найбільшим елементом серед меншої половини.

2. Мін-купа upper_half − зберігає більшу половину всіх отриманих чисел. Корінь цієї купи є найменшим елементом серед більшої половини.

Підтримується інваріант: розміри куп відрізняються не більше ніж на 1. Під час додавання нового числа воно спочатку додається до однієї з куп, після чого виконується балансування: якщо розміри стали незбалансованими, елемент переміщується з однієї купи в іншу.

Обчислення медіани:

1. Якщо загальна кількість елементів парна (купи однакового розміру), медіана − це середнє арифметичне коренів обох куп.

2. Якщо кількість елементів непарна, медіана − це корінь тієї купи, яка містить більше елементів (зазвичай lower_half).

Цей підхід дає змогу додавати елемент за O(*logn*) і знаходити медіану за O(1).

Задача злиття *k* відсортованих списків полягає в такому: об'єднати *k* відсортованих списків (або масивів) в один великий відсортований список.

Використовується мін-купа розміром k. Спочатку купа заповнюється першими елементами з кожного з k списків(ініціалізація). Кожен елемент у купі зберігається як кортеж (значення, індекс_списку). Далі відбувається процес злиття:

1. З купи вилучається мінімальний елемент (extract-min). Це і є наступний найменший елемент у загальному відсортованому списку. Він додається до результуючого списку.

2. З того списку, звідки було взято цей елемент, береться наступний елемент і додається до купи (insert).

3. Процес повторюється, доки купа не стане порожньою.

Аналіз складності. Нехай N − загальна кількість елементів у всіх списках. Кожен з N елементів буде доданий до купи та вилучений з неї рівно один раз. Оскільки розмір купи ніколи не перевищує k, кожна операція з купою коштує O(*logk*). Загальна часова складність становить O(*Nlogk*), а просторова − O(*k*) для зберігання купи.

## 6.3. Біноміальні купи

Хоча бінарна купа є надзвичайно корисною, вона має один суттєвий недолік: операція злиття (merge) двох куп вимагає лінійного часу O(n), що фактично зводиться до створення нової купи з елементів обох вихідних. Для застосувань, де злиття є частою операцією, потрібні ефективніші структури. Однією з таких структур є біноміальна купа, яка підтримує злиття за логарифмічний час.

Основою біноміальної купи є *біноміальне дерево*. Біноміальне дерево порядку k, що позначається як $B_k$, визначається рекурсивно: $B_0$ − це дерево, що складається з одного вузла; $B_k$ складається з двох дерев $B_{k-1}$, де корінь одного дерева стає найлівішим нащадком кореня іншого дерева.

Біноміальні дерева мають унікальні властивості, які доводяться індуктивно:

1. Дерево $B_k$ має рівно $2^k$ вузлів.
2. Висота дерева $B_k$ дорівнює k.
3. Корінь дерева $B_k$ має ступінь (кількість нащадків) k.
4. Нащадками кореня $B_k$ є корені дерев $B_{k-1}$, $B_{k-2}$,…, $B_0$ (зліва направо).

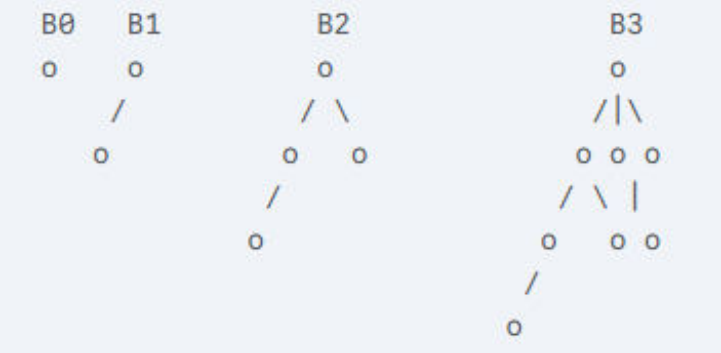



Рисунок 6.2 – Побудова біноміальних дерев

Дерево $B_k$ утворюється за рахунок з’єднання двох дерев $B_{k-1}$.

Біноміальна купа − це не одне дерево, а колекція (ліс) біноміальних дерев, що задовольняє такі властивості:

1. Порядкова властивість. Кожне біноміальне дерево в купі є мін-впорядкованим (або макс-впорядкованим).
2. Структурна властивість. Для будь-якого цілого невід’ємного k у купі існує *не більше одного* біноміального дерева порядку k.

Ця структурна властивість призводить до елегантного зв’язку між структурою купи та двійковим представленням кількості її елементів. Якщо біноміальна купа містить n вузлів, її структура однозначно визначається двійковим записом числа n. Для кожної одиниці в k-й позиції двійкового запису n купа містить одне біноміальне дерево $B_k$.

Наприклад, якщо купа містить n = 13 елементів, двійковий запис 13 − це $1101_2 = 2^3 + 2^2 + 2^0$. Отже, ця біноміальна купа буде

складатися з трьох дерев: $B_3$, $B_2$ та $B_0$. Корені цих дерев зазвичай зберігаються у зв'язаному списку, впорядкованому за зростанням їхнього порядку.

Операція злиття є центральною для біноміальних куп, оскільки на ній базуються інші операції. Процес злиття двох біноміальних куп $H_1$ та $H_2$ дуже схожий на двійкове додавання чисел, що відповідають розмірам цих куп.

Алгоритм одночасно проходить по списках коренів обох куп, впорядкованих за ступенем, подібно до алгоритму злиття для сортування:

1. Розглядаються дерева однакового порядку k з обох куп.
2. Випадок 1 (0 + 1 = 1). Якщо тільки одна купа має дерево $B_k$, воно просто переноситься до результуючої купи.
3. Випадок 2 (1 + 1 = 10). Якщо обидві купи мають дерево $B_k$, вони зливаються в одне дерево $B_{k+1}$. Корінь з меншим ключем стає новим коренем, а інший корінь − його нащадком. Це нове дерево $B_{k+1}$ стає «переносом» (carry) на наступний крок.
4. Випадок 3 (1 + 1 + 1 = 11). Якщо обидві купи мають дерево $B_k$ і є ще дерево $B_k$ від попереднього «переносу», одне з дерев $B_k$ залишається в результуючій купі, а два інших зливаються в $B_{k+1}$, що стає «переносом».

Оскільки в купі з n елементів є щонайбільше $\lfloor \log_2 n \rfloor + 1$ дерев, обхід списків коренів та виконання злиттів займає час O(logn).

Операція insert(key). Створюється нова біноміальна купа, що складається з одного вузла (дерево $B_0$), і зливається з існуючою купою. Складність − O(logn).

Операція find-min(). Мінімальний елемент є одним із коренів дерев. Його потрібно знайти, переглянувши список коренів. Оскільки дерев O(logn), складність операції − O(logn).

Операція extract-min():

1. Знаходиться корінь з мінімальним ключем (час O(logn)).
2. Це дерево $B_k$ видаляється з купи.
3. Нащадки видаленого кореня утворюють новий ліс дерев $B_{k-1}, \ldots, B_0$.
4. Ця нова колекція дерев зливається з рештою купи.

Загальна складність операції − O(logn).

## 6.4. Фібоначчієві купи

Фібоначчієві купи є подальшим розвитком ідеї зливних куп, що досягають ще кращої теоретичної продуктивності для певних операцій завдяки ще більш «лінивій» стратегії. Їх розробка була мотивована потребою в оптимізації алгоритмів на графах, таких як алгоритм Дейкстри, де операція decrease-key (зменшення ключа) виконується значно частіше, ніж extract-min. Мета полягала в тому, щоб звести амортизовану вартість decrease-key до O(1).

Основна ідея фібоначчієвої купи − відкладати якомога більше роботи на майбутнє. Структурні обмеження, наявні в біноміальній купі, тут послаблюються.

Операції insert та merge виконуються за константний час O(1). Вставка нового елемента − це просто створення нового дерева з одного вузла і додавання його до списку коренів. Злиття двох куп − це конкатенація їхніх списків коренів.

Операція extract-min виконує всю відкладену роботу з «прибирання» та реструктуризації купи.

Фібоначчієва купа − це колекція дерев, що задовольняють властивість мін-купи. На відміну від біноміальної купи, ці дерева необов'язково є біноміальними.Корені всіх дерев зберігаються вциклічному двобічнозв'язаному списку, що дає змогу швидко об'єднувати списки за O(1). Існує вказівник min на вузол-корінь з мінімальним ключем.Кожен вузол має поле mark (позначка), яке може бути true або false. Некореневий вузол позначається (mark = true), якщо він втратив одного зі своїх нащадків. Корені завжди не позначені.[65] Цей механізм є ключовим для обмеження «розростання» дерев і забезпечення логарифмічної складності extract-min.

Операція extract-min() та консолідація − це найскладніша операція, яка виконує відкладену роботу. Вузол з мінімальним ключем (на який вказує min) видаляється зі списку коренів. Усі його нащадки додаються до списку коренів як окремі дерева.

Виконується *консолідація*[24] − процес, що зменшує кількість дерев у списку коренів. Алгоритм використовує допоміжний

[24] Консолідація – це процес об'єднання, зміцнення або злиття чогось в одне ціле. Це слово використовується в різних сферах, і його точне значення залежить від контексту.

масив, індексований за ступенем коренів. Він проходить по всіх коренях і послідовно з'єднує дерева однакового ступеня, доки в списку не залишиться не більше одного дерева кожного ступеня. Процес з'єднання двох дерев полягає в тому, що корінь з більшим ключем стає нащадком кореня з меншим ключем. Після консолідації список коренів оновлюється і знаходиться новий вказівник min.

Амортизований аналіз показує, що хоча фактична вартість може бути великою, значне зменшення кількості дерев (і, відповідно, потенціалу) «оплачує» цю роботу, і амортизована вартість становить O(logn).

Операція decrease-key() та каскадні розрізиє основною перевагою фібоначчієвої купи.

1. Ключ вузла x зменшується до нового значення.

2. Якщо властивість купи порушена (ключ x став меншим за ключ його батька y), вузол x«відрізається» від свого батька. Він стає новим коренем у списку коренів, а його позначка mark встановлюється в false.

3. Далі запускається процес каскадних розрізів (cascading cuts):

3.1. Розглядається батько y. Якщо y не є коренем і він не був позначений, він позначається (mark = true), щоб указати, що він втратив нащадка. Процес зупиняється.

3.2. Якщо y не є коренем, але вже був позначений, це означає, що він втратив другого нащадка. У цьому випадку y також відрізається від свого батька, стає новим коренем (і його позначка знімається), а процес каскадного розрізу рекурсивно застосовується до батька y.

3.3. Каскад продовжується вгору по дереву, доки не буде досягнуто кореня або не позначеного вузла.

Фактична вартість пропорційна кількості розрізів. Однак амортизований аналіз показує, що кожен каскадний розріз (крім першого) «оплачується» потенціалом, що зберігався в позначеному вузлі. У результаті амортизована вартість decrease-key становить O(1).

## 6.5. Лівосторонні купи

Лівосторонні купи (leftist heaps) − це ще один варіант зливних куп, який, подібно до біноміальних, забезпечує логарифмічний час для операції злиття. На відміну від бінарних куп, які прагнуть до збалансованості, лівосторонні купи є навмисно незбалансованими певним чином, щоб оптимізувати злиття.

Лівостороння купа − це бінарне дерево, яке задовольняє властивість мін-купи, а також додаткову властивість лівосторонності. Ця властивість базується на концепції null path length (npl).Null Path Length (npl) для вузла x − це довжина найкоротшого шляху від x до нащадка, у якого відсутній хоча б один дочірній вузол (тобто до вузла з null лівим або правим вказівником). Формально, npl визначається рекурсивно:

- npl(null) = -1 (або 0, за домовленістю);
- npl(x) = 1 + min(npl(x.left), npl(x.right)).

Властивість лівосторонності: для кожного вузла x у купі виконується умовanpl(x.left) ≥ npl(x.right). Ця властивість гарантує, що дерево «важче» зліва. Прямим наслідком є те, що найкоротший шлях від будь-якого вузла до null завжди пролягає через правого нащадка. Цей шлях називається правим хребтом (right spine). Доведено, що для лівосторонньої купи з n вузлами довжина правого хребта не перевищує O(logn). Саме на цій властивості базується ефективність операцій.

Операція merge є фундаментальною для лівосторонніх куп, оскільки всі інші операції реалізуються через неї. Злиття двох лівосторонніх куп $H_1$ та $H_2$ виконується за рахунок злиття їхніх правих хребтів.

Алгоритм merge($H_1$, $H_2$):

1. Базовий випадок. Якщо одна з куп порожня, повертається інша.
2. Порівнюються корені $H_1$ та $H_2$. Купа з меншим коренем (нехай це буде $H_1$) стає коренем результуючої купи.
3. Правий нащадок $H_1$ рекурсивно зливається з усією купою $H_2$. Результат цього злиття стає новим правим піддеревом для кореня $H_1$.

4. Відновлення властивості. Після повернення з рекурсивного виклику перевіряється властивість лівосторонності для кореня $H_1$. Якщо npl(H1.left) < npl(H1.right), лівий та правий нащадки міняються місцями.

5. Оновлюється значення npl для кореня $H_1$ і повертається результат.

Оскільки всі рекурсивні виклики відбуваються вздовж коротких правих хребтів, загальна складність операції злиття становить O(logn).

Операція insert(key) − створюється нова лівостороння купа з одного вузла і зливається з існуючою. Складність становить O(logn).

Операція extract-min() − корінь видаляється, а його ліве та праве піддерева зливаються між собою. Складність − O(logn).

Вибір конкретної реалізації черги з пріоритетом залежить від специфіки задачі, зокрема від частоти виконання різних операцій. Кожна з розглянутих структур даних пропонує унікальний набір компромісів між складністю реалізації та асимптотичною продуктивністю.

Таблиця 6.2 – Асимптотична складність операцій над купами

| Операція | Бінарна купа | Біноміальна купа | Фібоначчієва купа (амортизована) | Лівостороння купа |
|---|---|---|---|---|
| **insert** | O(logn) | O(logn) | O(1) | O(logn) |
| **find-min** | O(1) | O(logn) | O(1) | O(1) |
| **extract-min** | O(logn) | O(logn) | O(logn) | O(logn) |
| **merge** | O(n) | O(logn) | O(1) | O(logn) |
| **decrease-key** | O(logn) | O(logn) | O(1) | O(logn) |
| **build-heap** | O(n) | O(n) | O(n) | O(n) |

Аналіз компромісів:

1. Бінарна купа є чудовим вибором для загального використання. Вона проста в реалізації, має низькі константні множники та ефективно використовує пам’ять завдяки представленню у вигляді масиву. Її продуктивність є оптимальною для застосувань, де не потрібні часті операції злиття, наприклад, для Heapsort.

2. Біноміальні та лівосторонні купи є структурами вибору, коли операція merge є критично важливою. Вони забезпечують логарифмічний час злиття, що є значним покращенням порівняно з лінійним часом бінарної купи.

3. Фібоначчієва купа пропонує найкращу теоретичну асимптотичну продуктивність, особливо для алгоритмів, де домінують операції decrease-key, як-от алгоритм Дейкстри на розріджених графах. Амортизована вартість O(1) для insert, merge та decrease-key робить її теоретично найшвидшою. Однак ця продуктивність досягається ціною значно вищої складності реалізації та більших константних множників у часі виконання, що може робити її менш практичною для багатьох реальних сценаріїв порівняно з простішою бінарною купою.

Отже, вибір структури даних для реалізації черги з пріоритетом − це завжди інженерне рішення, що вимагає аналізу очікуваного навантаження та співвідношення між теоретичною ефективністю та практичною складністю.

## 6.6. Приклади розв’язку задач

**Задача 1.** Злиття k відсортованих списків. Дано масив з k відсортованих за зростанням зв’язаних списків. Потрібно об’єднати їх в один відсортований зв’язаний список.

Ця задача ефективно вирішується за допомогою мін-купи. Купа буде зберігати до k елементів − по одному з кожного списку.

Алгоритм:

1. Створити мін-купу. Для кожного з k списків додати до купи його перший вузол (якщо список не порожній). Щоб уникнути проблем з порівнянням об’єктів вузлів, у купу варто додавати кортежі (значення, індекс_списку, вузол).

2. Створити фіктивний початковий вузол dummy для результуючого списку та вказівник current, що вказує на останній доданий вузол.

3. Поки купа не порожня:

3.1. Вилучити з купи вузол з мінімальним значенням (extract-min).

3.2. Приєднати цей вузол до результуючого списку (current.next = вилучений_вузол).

3.3. Пересунути вказівник current на щойно доданий вузол.

3.4. Якщо у вилученого вузла є наступний елемент у його вихідному списку, додати цей наступний елемент до купи.

4. Повернути dummy.next − початок відсортованого списку.

```
Python
import heapq

class ListNode:
    def __init__(self, val = 0, next = None):
        self.val = val
        self.next = next

def mergeKLists(lists):
    min_heap =
    #Додаємо унікальний лічильник для стабільного порівняння кортежів
    counter = 0
    for i, l in enumerate(lists):
        if l:
            heapq.heappush(min_heap, (l.val, counter, l))
            counter += 1

    dummy = ListNode()
    current = dummy

    while min_heap:
        val, _, node = heapq.heappop(min_heap)
        current.next = node
        current = current.next
        if node.next:
            heapq.heappush(min_heap, (node.next.val, counter, node.next))
            counter += 1

    return dummy.next
```

Загальна кількість вузлів N. Розмір купи не перевищує k. Кожен вузол додається і видаляється з купи один раз. Складність операцій з купою − O(logk). Загальна часова складність − O(Nlogk). Простір − O(k).

**Задача 2.** Пошук медіани в потоці даних. Розробити структуру даних, яка підтримує дві операції: addNum(num) − додає число з потоку, та findMedian() − повертає медіану всіх доданих на даний момент чисел.

Використовуємо дві купи: макс-купу lower_half для зберігання меншої половини чисел і мін-купу upper_half для більшої половини.

Алгоритм:

1. addNum(num):

1.1. Додати num до lower_half. Оскільки в Python heapq − це мін-купа, для реалізації макс-купи зберігаємо від'ємні значення.

1.2. Щоб підтримувати властивість, що всі елементи в upper_half більші за елементи в lower_half, перемістити найбільший елемент з lower_half до upper_half.

1.3. Збалансувати розміри куп: якщо upper_half стала більшою за lower_half, перемістити найменший елемент з upper_half назад до lower_half.

2. findMedian():

2.1. Якщо lower_half має більше елементів, медіана − це її корінь (змінений знак).

2.2. Інакше (розміри рівні), медіана − середнє арифметичне коренів обох куп.

```
Python
import heapq

class MedianFinder:
    def __init__(self):
        self.lower_half =  # Макс-купа (зберігаємо від'ємні значення)
        self.upper_half =  # Мін-купа

    def addNum(self, num: int) -> None:
        heapq.heappush(self.lower_half, -num)
```

```
        # Переміщення найбільшого з меншої половини до більшої
        if self.lower_half and self.upper_half and (-self.lower_half > self.upper_half):
            val = -heapq.heappop(self.lower_half)
            heapq.heappush(self.upper_half, val)

        # Балансування
        if len(self.lower_half) > len(self.upper_half) + 1:
            val = -heapq.heappop(self.lower_half)
            heapq.heappush(self.upper_half, val)

        if len(self.upper_half) > len(self.lower_half):
            val = heapq.heappop(self.upper_half)
            heapq.heappush(self.lower_half, -val)

    def findMedian(self) -> float:
        if len(self.lower_half) > len(self.upper_half):
            return -self.lower_half
        else:
            return (-self.lower_half + self.upper_half) / 2.0
```

Операція addNumохоплює кілька операцій з купами, кожна з яких коштує O(logn). Загальна складність addNum − O(logn). findMedian − O(1).

## Контрольні питання

1. Що таке черга з пріоритетом і чим вона відрізняється від звичайної черги та стека?

2. Які дві основні властивості визначають бінарну купу? Поясніть кожну.

3. Чому представлення повної бінарної купи у вигляді масиву є ефективним? Напишіть формули для знаходження батька та дочірніх елементів для вузла з індексом i.

4. Опишіть алгоритми sift-up та sift-down. В яких операціях над купою вони використовуються?

5. Доведіть, що побудова бінарної купи з n елементів може бути виконана за час O(n). Чому цей підхід ефективніший за n послідовних вставок?

6. Поясніть, як використання мін-купи оптимізує алгоритм Дейкстри. Яка результуюча часова складність?

7. Опишіть два основні етапи алгоритму сортування купою (Heapsort). Чому його загальна складність O(nlogn)?

8. Що таке біноміальне дерево Bk? Як воно рекурсивно визначається?

9. Поясніть аналогію між операцією злиття двох біноміальних куп та двійковим додаванням.

10. Яка основна мотивація для створення фібоначчієвих куп? Яку операцію вони оптимізують і чому це важливо?

11. Поясніть концепцію «лінивих» операцій у фібоначчієвих купах. Яка операція виконує основну «роботу з прибирання»?

12. Опишіть процес консолідації дерев під час операції extract-min у фібоначчієвій купі.

13. Що таке «каскадний розріз» (cascading cut) у фібоначчієвій купі і коли він відбувається?

14. Що таке «null path length» (npl) у лівосторонній купі? Яку властивість дерева вона допомагає підтримувати?

15. Порівняйте часову складність операції merge для бінарної, біноміальної та фібоначчієвої куп. Поясніть причини таких відмінностей.

## Задачі для самостійного опрацювання

1. Реалізуйте Макс-Купу (Max-Heap) на C++ у вигляді шаблонного класу.

2. Напишіть функцію, яка перевіряє, чи є заданий масив бінарною мін-купою.

3. Реалізуйте алгоритм Heapsort, який сортує масив у порядку спадання.

4. Задача «K найчастіших елементів»: маючи масив чисел, знайдіть k елементів, що зустрічаються найчастіше. Використайте мін-купу.

5. Задача «K точок, найближчих до початку координат»: маючи список точок на площині, знайдіть k точок, найближчих до (0,0).

6. Реалізуйте операцію delete(i) для бінарної купи, яка видаляє елемент за довільним індексом i за час O(logn).

7. Доведіть, що біноміальне дерево $B_k$ має ($j_k$) вузлів на глибині j.

8. Намалюйте біноміальну купу, що містить 19 елементів. Які біноміальні дерева вона міститиме?

9. Виконайте покроково операцію extract-min на біноміальній купі з 13 елементами {1, 2,..., 13}.

10. Поясніть, чому в фібоначчієвій купі максимальний ступінь будь-якого вузла обмежений O(logn).

11. Виконайте послідовність операцій decrease-key, що призведе до каскадного розрізу щонайменше 2 рівнів у фібоначчієвій купі.

12. Доведіть, що лівостороння купа з n вузлами має довжину правого шляху не більше $\lfloor \log_2(n + 1) \rfloor$.

13. Реалізуйте лівосторонню купу на Python або C++.

14. Модифікуйте алгоритм Дейкстри для роботи з std::priority_queue в C++, враховуючи, що вона не підтримує decrease-key.

15. Дослідіть та опишіть структуру даних «Skew Heap». Порівняйте її з лівосторонньою купою. У чому її переваги та недоліки?

# ТЕМА 7
# АЛГОРИТМИ СОРТУВАННЯ

Задача сортування є однією з найбільш фундаментальних та досліджуваних проблем у комп'ютерних науках. Її розв'язання є ключовим для оптимізації багатьох інших алгоритмів, зокрема алгоритмів пошуку, злиття даних та обчислювальної геометрії. Формально, задача сортування визначається так:

Вхід: послідовність з n елементів $\langle a1, a2, ..., an \rangle$.

Вихід: перестановка (перевпорядкування) вхідної послідовності $\langle a1', a2', ..., an' \rangle$ така, що для будь-яких двох елементів виконується умова $a1' \leq a2' \leq \cdots \leq an'$.

Елементи, що сортуються, називаються ключами. Вони можуть бути як простими типами даних (цілі числа, символи), так і складними структурами (об'єкти), для яких визначено відношення порядку.

## 7.1. Ключові характеристики алгоритмів

Для аналізу та порівняння ефективності різних алгоритмів сортування використовуються декілька ключових метрик.

Часова складність (Time Complexity). Це оцінка кількості операцій, які виконує алгоритм, як функція від розміру вхідних даних (n). Оскільки поведінка алгоритму може залежати від початкового стану даних, аналіз проводять для трьох сценаріїв.

1. Найкращий випадок (Best Case). Мінімальний час роботи для входу розміром n. Використовується Ω-нотація для позначення нижньої межі.

2. Найгірший випадок (Worst Case). Максимальний час роботи для входу розміром n. Це найважливіша характеристика, оскільки вона гарантує верхню межу продуктивності. Використовується O-нотація.

3. Середній випадок (Average Case). Очікуваний час роботи для «середньостатистичного» входу розміром n. Використовується Θ-нотація, коли найкращий та найгірший випадки асимптотично збігаються.

Просторова складність (Space Complexity). Це оцінка об'єму додаткової пам'яті, яку алгоритм використовує під час своєї роботи, крім пам'яті, що займають вхідні дані. Алгоритми, що вимагають мінімальний, константний об'єм додаткової пам'яті (зазвичай O(1) або O(logn) для стека рекурсії), називаються алгоритмами сортування «на місці» (in-place). Алгоритми, що потребують додаткової пам'яті, пропорційної розміру вхідних даних (O(n)), називаються «out-of-place».

Стабільність (Stability). Алгоритм сортування називається стабільним, якщо він не змінює відносний порядок елементів з однаковими ключами. Іншими словами, якщо два елементи R і S мають однакові ключі, і в початковому масиві R стоїть перед S, то у відсортованому масиві R також буде стояти перед S.

Стабільність є критично важливою властивістю в реальних системах обробки даних. Наприклад, якщо користувач сортує таблицю в електронному редакторі спочатку за стовпцем «Прізвище», а потім за стовпцем «Місто», він очікує, що записи для одного міста залишаться впорядкованими за прізвищем. Нестабільний алгоритм сортування може порушити цей порядок, що призведе до непередбачуваних та некоректних результатів з точки зору користувача. Отже, стабільність є не просто абстрактною властивістю, а фундаментальною вимогою для збереження цілісності даних у багатоетапних процесах сортування.

## 7.2. Класифікація алгоритмів

Усі алгоритми сортування можна розділити на дві великі категорії, що визначає їх фундаментальні можливості та обмеження:

1. Порівняльні сортування (Comparison Sorts). Це алгоритми, які визначають порядок елементів виключно на основі операцій порівняння (<, =, >). Більшість класичних алгоритмів, таких як сортування бульбашкою, вставками, швидке сортування, сортування злиттям та пірамідальне сортування, належать до цієї категорії. Вони є універсальними, оскільки можуть працювати з будь-яким типом даних, для якого визначена операція порівняння.

Однак ця універсальність має свою ціну: існує теоретична нижня межа часової складності для таких алгоритмів, яка дорівнює Ω(nlogn).

2. Непорівняльні сортування (Non-comparison Sorts). Ці алгоритми використовують специфічні властивості даних для їх впорядкування без прямих порівнянь. Наприклад, вони можуть припускати, що ключі є цілими числами з певного невеликого діапазону. До цієї категорії належать сортування підрахунком, розрядне сортування та кошикове сортування. Використовуючи додаткову інформацію про дані, ці алгоритми можуть подолати межу в Ω(nlogn) і досягти лінійної часової складності за певних умов.

Фундаментальна відмінність між цими двома класами полягає в обсязі інформації, яку алгоритм використовує. Порівняльні алгоритми отримують лише один біт інформації з кожного порівняння (більше / менше), що обмежує їхню швидкість. Непорівняльні алгоритми використовують самі значення ключів для визначення їхньої позиції (наприклад, як індекс у допоміжному масиві), що є значно потужнішою операцією, але вимагає сильних припущень щодо природи даних. Це ілюструє ключовий компроміс в алгоритмічному дизайні: між загальністю та ефективністю.

## 7.3. Прості порівняльні алгоритми: квадратична складність O($n^2$)

Ця група алгоритмів є простою для розуміння та реалізації, що робить їх чудовим інструментом для вивчення базових принципів сортування. Однак їхня квадратична часова складність робить їх неефективними для обробки великих наборів даних.

Сортування бульбашкою − це один з найпростіших алгоритмів, який багаторазово проходить по масиву, порівнюючи сусідні елементи і міняючи їх місцями, якщо вони містяться в неправильному порядку. Після кожного повного проходу найбільший (або найменший, залежно від напрямку сортування) елемент серед невідсортованих «спливає» на свою фінальну позицію в кінці масиву,

подібно до бульбашки повітря у воді. Процес повторюється доти, доки весь масив не буде відсортовано.

Розглянемо сортування масиву за зростанням.

Перший прохід:

[**5, 1**, 4, 2, 8] → [**1, 5**, 4, 2, 8] (обмін, бо 5 > 1)

[1, **5, 4**, 2, 8] → [1, **4, 5**, 2, 8] (обмін, бо 5 > 4)

[1, 4, **5, 2**, 8] → [1, 4, **2, 5**, 8] (обмін, бо 5 > 2)

[1, 4, 2, **5, 8**] → `` (без обміну, бо 5 < 8). Найбільший елемент 8 на своєму місці.

Другий прохід:

[**1, 4**, 2, 5, 8] → `` (без обміну)

[1, **4, 2**, 5, 8] → [1, **2, 4**, 5, 8] (обмін, бо 4 > 2)

[1, 2, **4, 5**, 8] → `` (без обміну). Другий найбільший елемент 5 на своєму місці.

Третій прохід:

[**1, 2**, 4, 5, 8] → `` (без обміну)

[1, **2, 4**, 5, 8] → `` (без обміну).

Четвертий прохід:

[**1, 2**, 4, 5, 8] → `` (без обміну).

Оптимізована версія алгоритму може завершити роботу раніше, якщо під час повного проходу не відбулося жодного обміну, що свідчить про те, що масив уже відсортовано.

*Псевдокод (оптимізований)*

```
procedure bubbleSort(A : list of sortable items)
  n := length(A)
  repeat
    swapped := false
    for i := 1 to n-1 inclusive do
      if A[i-1] > A[i] then
        swap(A[i-1], A[i])
        swapped := true
      end if
    end for
    n := n - 1
  until not swapped
end procedure
```

```
Python
def bubbleSort(arr):
    n = len(arr)
    # Проходимо по всіх елементах масиву
    for i in range(n):
        swapped = False
        # Останні і елементів вже на своїх місцях
        for j in range(0, n-i-1):
            # Проходимо масив від 0 до n-i-1
             # Міняємо місцями, якщо знайдений елемент більший за наступний
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        # Якщо на внутрішньому циклі не було обмінів, масив відсортовано
        if not swapped:
            break
C++
#include<iostream>
#include<vector>
#include<utility> // for std::swap

void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // Якщо на внутрішньому циклі не було обмінів, масив відсортовано
        if (!swapped) {
            break;
        }
    }
}
```

Часова складність − O($n^2$) у середньому та найгіршому випадках. Найгірший випадок − обернено відсортований масив. Найкращий випадок для оптимізованої версії − O($n$), коли масив уже відсортований.

Просторова складність − O(1), оскільки сортування відбувається на місці.

Стабільність -алгоритм є стабільним, оскільки обмін відбувається лише між сусідніми елементами, і елементи з однаковими ключами ніколи не «перестрибують» один через одного.

Ідея сортування вибором полягає в тому, щоб розділити масив на дві частини: відсортовану (спочатку порожню) та невідсортовану. На кожній ітерації алгоритм знаходить найменший елемент у невідсортованій частині та міняє його місцями з першим елементом цієї частини. Так, відсортована частина поступово зростає зліва направо, доки весь масив не буде впорядковано.

Розглянемо сортування масиву. Символ | позначає межу між відсортованою та невідсортованою частинами.

Початковий стан:|

Ітерація 1: Найменший елемент − 11. Міняємо його з першим елементом 64.

[**11** | 25, 12, 22, **64**]

Ітерація 2: Шукаємо найменший у ``. Це 12. Міняємо його з 25.

[11, **12** | **25**, 22, 64]

Ітерація 3: Шукаємо найменший у ``. Це 22. Міняємо його з 25.

[11, 12, **22** | **25**, 64]

Ітерація 4: Шукаємо найменший у ``. Це 25. Він уже на першій позиції, обмін не потрібен.

[11, 12, 22, **25** | 64]

Масив відсортовано.

*Псевдокод*

```
procedure selectionSort(A : list of sortable items)
  n := length(A)
  for i from 0 to n-2 do
    minIndex := i
    for j from i+1 to n-1 do
      if A[j] < A[minIndex] then
        minIndex := j
```

```
        end if
      end for
      swap(A[i], A[minIndex])
    end for
  end procedure
```

```
Python
def selectionSort(arr):
    n = len(arr)
    # Проходимо по всіх елементах масиву
    for i in range(n):
        # Знаходимо індекс мінімального елемента в невідсортованій частині
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Міняємо знайдений мінімальний елемент з першим елементом невідсортованої частини
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

```
C++
#include <iostream>
#include <vector>
#include <utility> // for std::swap

void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        // Знаходимо індекс мінімального елемента в невідсортованій частині arr[i..n-1]
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Міняємо знайдений мінімальний елемент з першим елементом
        if (min_idx!= i) {
            std::swap(arr[min_idx], arr[i]);
        }
    }
}
```

Часова складність − O($n^2$) у всіх випадках (найкращому, середньому та найгіршому). Алгоритм завжди виконує два вкладені цикли, оскільки для знаходження мінімального елемента потрібно переглянути всю невідсортовану частину, незалежно від початкового стану масиву.

Просторова складність становить O(1), оскільки сортування відбувається на місці.

Стабільність. Алгоритм є нестабільним. Обмін елемента з першим елементом невідсортованої частини може порушити відносний порядок елементів з однаковими ключами. Наприклад, у масиві [5a, 3, 5b, 2], на першому кроці 2 буде обміняно з 5a, і відносний порядок 5a та 5b зміниться.

Сортування вставками працює подібно до того, як людина сортує гральні карти в руках. Масив віртуально ділиться на відсортовану та невідсортовану частини. Елементи з невідсортованої частини по черзі виймаються і вставляються на свою правильну позицію у відсортованій частині. Для цього кожен новий елемент порівнюється з елементами у відсортованій частині (справа наліво), доки не буде знайдено правильне місце для вставки.

Розглянемо сортування масиву.

Початковий стан:[12 | 11, 13, 5, 6] (вважаємо, що 12 вже відсортовано)

Ітерація 1: Беремо 11. Порівнюємо з 12. 11 < 12, тому зсуваємо 12 праворуч і вставляємо 11 на його місце.

[**11, 12** | 13, 5, 6]

Ітерація 2: Беремо 13. Порівнюємо з 12. 13 > 12, тому 13 залишається на місці.

[11, 12, **13** | 5, 6]

Ітерація 3: Беремо 5. Порівнюємо з 13 (зсув), з 12 (зсув), з 11 (зсув). Вставляємо 5 на початок.

[**5, 11, 12, 13** | 6]

Ітерація 4: Беремо 6. Порівнюємо з 13 (зсув), з 12 (зсув), з 11 (зсув). 6 > 5, тому вставляємо 6 після 5.

[5, **6, 11, 12, 13** |]

Масив відсортовано.

*Псевдокод*

```
procedure insertionSort(A : list of sortable items)
  for i from 1 to length(A)-1 do
    key := A[i]
    j := i - 1
    while j >= 0 and A[j] > key do
      A[j + 1] := A[j]
      j := j - 1
    end while
    A[j + 1] := key
  end for
end procedure
```

*Python*

```
def insertionSort(arr):
    # Проходимо від 1-го до останнього елемента
    for i in range(1, len(arr)):
        key = arr[i]
        # Переміщуємо елементи arr[0..i-1], які більші за key,
        # на одну позицію вперед
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

*C++*

```
#include<iostream>
#include<vector>

void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Переміщуємо елементи arr[0..i-1], які більші за key,
        // на одну позицію вперед від їх поточної позиції
        while (j > = 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Часова складність − O($n^2$) у середньому та найгіршому випадках. Найкращий випадок − O(n), коли масив уже відсортований, оскільки внутрішній цикл while не виконається жодного разу. Це робить його дуже ефективним для майже відсортованих даних.

Просторова складність − O(1), оскільки сортування відбувається на місці.

Алгоритм є стабільним. Елементи зсуваються лише тоді, коли новий елемент строго менший (<), тому елементи з однаковими ключами не змінюють свого відносного порядку.

## 7.4. Порівняльний аналіз алгоритмів з O($n^2$) складністю

Хоча всі три алгоритми мають однакову середню часову складність, їхні практичні характеристики суттєво відрізняються. Ці відмінності зумовлені кількістю та типом операцій, які вони виконують.

Сортування вибором виконує найменшу можливу кількість обмінів − рівно $n - 1$ у найгіршому випадку. Це робить його корисним у сценаріях, де операції запису в пам'ять є дуже «дорогими» (наприклад, під час роботи з флеш-пам'яттю). Однак кількість порівнянь завжди залишається високою, O($n^2$), незалежно від вхідних даних.

Сортування вставками є адаптивним. На майже відсортованих даних воно виконує значно менше порівнянь та зсувів, наближаючись до лінійної складності. Це робить його найшвидшим серед цих трьох для таких випадків. Через низькі постійні множники та простоту реалізації сортування вставками часто використовується як частина більш складних гібридних алгоритмів (наприклад, Timsort, що використовується в Python) для сортування невеликих підмасивів, де воно перевершує рекурсивні алгоритми з вищими накладними витратами.

Сортування бульбашкою зазвичай є найповільнішим на практиці через велику кількість як порівнянь, так і обмінів. Його основна цінність − педагогічна, як простий приклад сортування.

Таблиця 7.1 – Порівняння алгоритмів сортування з квадратичною складністю

| Алгоритм | Часова складність (найкращий випадок) | Часова складність (середній випадок) | Часова складність (найгірший випадок) | Просторова складність | Стабільність | Ключова особливість |
|---|---|---|---|---|---|---|
| Сортування бульбашкою | O(n) | $O(n^2)$ | $O(n^2)$ | O(1) | Так | Простота реалізації, порівняння сусідніх елементів |
| Сортування вибором | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | O(1) | Ні | Мінімальна кількість обмінів (O(n)) |
| Сортування вставками | O(n) | $O(n^2)$ | $O(n^2)$ | O(1) | Так | Ефективний для малих та майже відсортованих масивів |

## 7.5. Ефективні порівняльні алгоритми: складність O(nlogn)

Для роботи з великими наборами даних алгоритми з квадратичною складністю стають неприйнятно повільними. Наступна категорія алгоритмів використовує більш витончені підходи, зокрема парадигму «Розділяй і володарюй», щоб досягти значно кращої асимптотичної складності O(nlogn).

Швидке сортування, розроблене Тоні Хоаром, є одним з найпопулярніших та найшвидших на практиці алгоритмів сортування. Він базується на принципі «Розділяй і володарюй» (Divide and Conquer). Алгоритм працює так:

1. Розділення (Partition). Вибирається елемент з масиву, який називається опорним (pivot). Масив перевпорядковується

так, що всі елементи, менші за опорний, розміщуються ліворуч від нього, а всі елементи, більші або рівні − праворуч. Після цього кроку опорний елемент міститься на своїй фінальній відсортованій позиції.

2. Володарюй (Conquer). Рекурсивно застосувати швидке сортування до двох підмасивів, що утворилися ліворуч і праворуч від опорного елемента.

3. Комбінування (Combine). Цей крок є тривіальним, оскільки сортування відбувається на місці, і після сортування підмасивів весь масив автоматично стає відсортованим.

Основна робота в Quicksort виконується на етапі розділення, на відміну від сортування злиттям, де ключовим є етап комбінування. Ця структурна відмінність пояснює їхні різні профілі продуктивності: Quicksort працює на місці, але його ефективність сильно залежить від якості розділення, тоді як Mergesort гарантує збалансований поділ, але потребує додаткової пам’яті.

Процес розділення (Partition). Існує кілька схем реалізації цього кроку. Дві найпоширеніші − це схеми Ломуто та Хоара. Схема Ломуто (Lomuto Partition Scheme) − ця схема простіша для розуміння. Зазвичай останій елемент масиву обирається як опорний. Вводиться індекс i, який указує на межу між елементами, меншими за опорний та більшими. Масив сканується іншим індексом j. Якщо A[j] менший за опорний і інкрементується, і A[i] міняється місцями з A[j]. У кінці опорний елемент міняється місцями з A[i + 1].

Схема Хоара (Hoare Partition Scheme) − це оригінальна і на практиці більш ефективна схема. Використовуються два індекси (вказівники), що рухаються з протилежних кінців масиву назустріч один одному. Лівий індекс шукає елемент, більший або рівний опорному, а правий − менший або рівний. Коли такі елементи знайдено, вони міняються місцями. Процес триває, доки індекси не перетнуться. Схема Хоара виконує в середньому менше обмінів, ніж схема Ломуто.

Вибір опорного елемента (Pivot). Продуктивність Quicksort критично залежить від вибору опорного елемента. Ідеальний опорний елемент − це медіана масиву, яка ділить його на дві рівні частини.

Поганий вибір − вибір першого або останнього елемента у вже відсортованому або обернено відсортованому масиві призводить до виродженого розділення (один підмасив порожній, інший розміром n − 1), що веде до найгіршої часової складності $O(n^2)$.

Хороші стратегії:

1. Вибір випадкового елемента як опорного з високою ймовірністю дає змогу уникнути найгіршого випадку.

2. Медіана трьох (Median-of-three) − вибирається медіана з першого, середнього та останнього елементів масиву. Ця стратегія значно зменшує ймовірність вибору поганого опорного елемента і є поширеною практичною оптимізацією.

Часова складність. Найкращий та середній випадок: $\Theta(n\log n)$. Це досягається, коли опорний елемент щоразу ділить масив приблизно навпіл. Найгірший випадок -$O(n^2)$. Виникає за систематично невдалого вибору опорного елемента.

Просторова складність − $O(\log n)$ у середньому випадку для зберігання стека рекурсивних викликів. У найгіршому випадку може сягати $O(n)$.

Класичні реалізації Quicksort не є стабільними.

Сортування злиттям − це класичний приклад алгоритму «Розділяй і володарюй», який, на відміну від Quicksort, гарантує високу продуктивність у найгіршому випадку. Його логіка така:

1. Розділення (Divide). Вхідний масив рекурсивно ділиться на дві рівні (або майже рівні) половини, доки не залишаться підмасиви, що складаються з одного елемента. Підмасив з одного елемента за визначенням є відсортованим.

2. Володарюй (Conquer). Цей крок по суті є базовим випадком рекурсії – сортування підмасивів з одного елемента.

3. Комбінування (Combine). Відсортовані підмасиви послідовно зливаються (merge) в один більший відсортований масив. Цей крок є ключовим і містить основну логіку алгоритму.

Процес злиття (Merge). Процедура злиття приймає два вже відсортовані підмасиви та об’єднує їх в один. Для цього створюється тимчасовий масив. Два вказівники встановлюються на початки підмасивів. На кожному кроці менший з двох елементів, на які вказують вказівники, копіюється в тимчасовий масив, і відповідний вказівник зсувається. Процес триває, доки один з підмасивів

не вичерпається. Після цього залишок іншого підмасиву копіюється в кінець тимчасового масиву. Нарешті, відсортовані елементи з тимчасового масиву копіюються назад у вихідний.

Часова складність − O(nlogn) у всіх випадках (найкращому, середньому та найгіршому). Поділ масиву завжди збалансований, що гарантує логарифмічну глибину рекурсії, а на кожному рівні рекурсії виконується O(n) роботи для злиття.

Просторова складність становить O(n), оскільки для процедури злиття потрібен допоміжний масив розміром, пропорційним розміру вхідних даних. Це робить його «out-of-place» алгоритмом.

Сортування злиттям є стабільним, якщо реалізація процедури злиття є коректною (за рівності ключів перевага надається елементу з лівого підмасиву).

Пірамідальне Сортування (Heapsort) − це ефективний алгоритм сортування на місці, який використовує структуру даних, відому як бінарна купа (Binary Heap). Алгоритм можна розглядати як удосконалену версію сортування вибором: на кожному кроці він знаходить максимальний елемент і переміщує його в кінець, але робить це значно ефективніше, за O(logn) часу, завдяки властивостям купи.

Бінарна купа (Binary Heap) − це майже повне бінарне дерево, яке можна зручно представити у вигляді масиву. Вона задовольняє властивості купи: для кожного вузла, крім кореня, ключ батьківського вузла не менший (для max-heap) або не більший (для min-heap) за ключ самого вузла. Так, у max-heap найбільший елемент завжди міститься в корені дерева.

Для елемента з індексом i в масиві:

- його батько міститься за індексом ⌊(i − 1)/2⌋;
- його лівий нащадок − за індексом 2i + 1;
- його правий нащадок − за індексом 2i + 2.

Процедури:

1. Просіювання вниз (siftDown / heapify) − це ключова операція для підтримки властивості купи. Вона приймає вузол, властивість купи для якого може бути порушена (але для його нащадків вона виконується), і «просіює» його вниз по дереву, міняючи місцями з найбільшим нащадком, доки властивість купи не буде

відновлено. Час роботи цієї процедури пропорційний висоті дерева, тобто O(logn).

2. Побудова купи (buildMaxHeap). Для перетворення невпорядкованого масиву на max-heap процедура siftDown застосовується до всіх вузлів, що не є листками, починаючи з останнього такого вузла і рухаючись до кореня. Цей процес, на диво, має лінійну часову складність O(n).

Алгоритм сортування:

1. Побудова купи. Перетворити вхідний масив на max-heap за допомогою процедури buildMaxHeap.

2. Сортування. Повторити n − 1 раз:

2.1. Обміняти кореневий елемент (найбільший у купі) з останнім елементом купи.

2.2. «Відрізати» останній елемент від купи (зменшити її розмір на 1). Тепер цей елемент міститься на своїй фінальній позиції.

2.3. Викликати siftDown для нового кореня, щоб відновити властивість max-heap для зменшеної купи.

Часова складність становитьO(nlogn) у всіх випадках. Побудова купи займає O(n), а далі виконується n − 1 операцій вилучення максимуму, кожна з яких займає O(logn).

Просторова складність − O(1), оскільки сортування відбувається на місці.

Heapsort не є стабільним.

Heapsort поєднує в собі найкращі риси Quicksort та Mergesort: гарантовану часову складність O(nlogn) та сортування на місці. Однак на практиці він часто повільніший за добре реалізований Quicksort. Причина полягає у слабкій локальності посилань. Операція siftDown вимагає доступ до елементів, що розміщені далеко один від одного в масиві (батько і та нащадки 2i + 1, 2i + 2), що призводить до частих промахів кешу процесора. Натомість, Quicksort переважно виконує послідовні сканування масиву, що є значно більш дружнім до кешу і пояснює його вищу швидкість на реальному обладнанні.

## 7.6. Нижня межа складності для порівняльних сортувань

Ми розглянули кілька алгоритмів сортування з часовою складністю O(nlogn). Виникає природне питання: чи можна створити порівняльний алгоритм сортування, який буде асимптотично швидшим? Відповідь − ні. Існує фундаментальне доведення, що будь-який алгоритм сортування, що базується на порівняннях, вимагає щонайменше Ω(nlogn) порівнянь у найгіршому випадку.

Для доведення цієї нижньої межі використовується абстрактна модель, що називається деревом рішень. Будь-який порівняльний алгоритм сортування можна представити у вигляді бінарного дерева.

1. Внутрішні вузли. Кожен внутрішній вузол дерева представляє одну операцію порівняння між двома елементами, наприклад, $a_i : a_j$.

2. Гілки. Кожен вузол має дві гілки, що відповідають можливим результатам порівняння. Наприклад, ліва гілка відповідає $a_i \leq a_j$, а права − $a_i > a_j$.

3. Листки. Кожен листовий вузол представляє один з можливих результатів сортування − унікальну перестановку вхідних елементів.

4. Шлях виконання. Робота алгоритму для конкретного вхідного масиву відповідає унікальному шляху від кореня до одного з листків. Довжина цього шляху (кількість ребер) дорівнює кількості порівнянь, виконаних алгоритмом.

Так, кількість порівнянь у найгіршому випадку для будь-якого алгоритму дорівнює висоті його дерева рішень.

Доведення нижньої межі Ω(nlogn). Доведення базується на властивостях цього дерева.

1. Кількість листків. Для масиву з n унікальних елементів існує n! (n факторіал) можливих початкових перестановок. Коректний алгоритм сортування повинен бути здатним обробити кожну з них і видати правильний відсортований порядок. Це означає, що дерево рішень повинно мати щонайменше n! листків, оскільки кожен листок відповідає унікальному результату.

2. Властивість бінарного дерева. Бінарне дерево з висотою h може мати не більше ніж 2h листків. Це випливає з того, що на кожному рівні кількість вузлів щонайбільше подвоюється.

3. Формулювання нерівності. Поєднуючи ці два факти, ми отримуємо фундаментальну нерівність:

$$n! \leq \text{кількість листків} \leq 2^h$$

4. Щоб знайти нижню межу для висоти h, візьмемо логарифм за основою 2 від обох частин нерівності:

$$\log_2(n!) \leq \log_2(2^h)$$
$$h \geq \log_2(n!)$$

5. Для оцінки $\log_2(n!)$ використовується формула Стірлінга, яка дає наближення для факторіала$n!$ $\approx$ $\sqrt{2\pi n}(\frac{n}{e})^n$. Логарифмуючи це, отримуємо:

$$\log_2(n!) \approx n\log_2 n - n\log_2 e = \Theta(n\log n)$$

Більш строгий аналіз показує, що $\log_2(n!) = \Omega(n\log n)$.

Оскільки висота дерева h представляє кількість порівнянь у найгіршому випадку, будь-який порівняльний алгоритм сортування повинен виконати щонайменше $\Omega(n\log n)$ порівнянь.

Сила цього доведення полягає в його загальності. Воно не аналізує конкретний алгоритм, а натомість аналізує саму проблему сортування в рамках моделі, що базується на порівняннях. Це встановлює фундаментальну межу для цілого класу алгоритмів, навіть для тих, які ще не були винайдені. Алгоритми, такі як Mergesort та Heapsort, які досягають складності $O(n\log n)$ у найгіршому випадку, є асимптотично оптимальними в межах цієї моделі.

Ця теоретична межа є основною мотивацією для дослідження непорівняльних алгоритмів. Щоб досягти лінійної часової складності, потрібно «порушити правила» моделі порівнянь, тобто використати додаткову інформацію про дані, крім результатів порівнянь.

## 7.7. Непорівняльні алгоритми: лінійна часова складність

Непорівняльні алгоритми сортування досягають вищої продуктивності, використовуючи специфічні властивості даних, що дає змогу їм уникнути операцій порівняння між елементами. Вони можуть досягти лінійної часової складності, але ціною цього є втрата універсальності.

Сортування підрахунком (Counting Sort) − це ефективний алгоритм для сортування колекції об’єктів, ключі яких є цілими числами в невеликому діапазоні. Основна ідея полягає в тому, щоб підрахувати кількість входжень кожного унікального елемента і використати цю інформацію для обчислення позиції кожного елемента у відсортованому вихідному масиві.

Покроковий алгоритм:

Нехай вхідний масив A містить n елементів, значення яких лежать у діапазоні від 0 до k.

1. Ініціалізація. Створити допоміжний масив count розміром k + 1 і заповнити його нулями. Цей масив буде зберігати частоти елементів.

2. Підрахунок частот. Пройти по вхідному масиву A. Для кожного елемента A[i] інкрементувати лічильник count[A[i]]. Після цього кроку count[j] містить кількість елементів, що дорівнюють j.

3. Обчислення кумулятивних сум**.** Модифікувати масив count так, щоб count[j] містив кількість елементів, менших або рівних j. Це робиться через додавання до кожного елемента значення попереднього: count[j] = count[j] + count[j-1]. Тепер count[j] вказує на кінцеву позицію елемента j у відсортованому масиві.

4. Формування вихідного масиву: Створити вихідний масив B розміром n. Пройти по вхідному масиву A у зворотному порядку (від n − 1 до 0). Для кожного елемента A[i]:

4.1. Помістити A[i] у вихідний масив B на позицію count[A[i]] - 1.

4.2. Декрементувати count[A[i]].

Прохід у зворотному порядку забезпечує стабільність алгоритму.

Часова складність − O(n + k). Крок 1 займає O(k), крок 2 − O(n), крок 3 − O(k), крок 4 − O(n). Загальна складність − O(n + k). Якщо k = O(n), то складність стає лінійною, O(n).

Просторова складність − O(n + k) для зберігання масивів count та B.

Обмеження. Ефективність алгоритму різко падає, коли діапазон значень k значно перевищує кількість елементів n. Наприклад, сортування мільйона 32-бітних чисел (k ≈ 4 · 109) за допомогою цього методу є непрактичним через величезний розмір масиву count.

Розрядне Сортування (Radix Sort) − це алгоритм, який сортує числа, обробляючи їх по розрядах (цифрах), від найменш значущого (LSD − Least Significant Digit) до найбільш значущого (MSD − Most Significant Digit). Для сортування за кожним розрядом він використовує будь-який стабільний алгоритм сортування, найчастіше − сортування підрахунком.

Принцип роботи (LSD Radix Sort):

1. Визначити максимальну кількість розрядів (d) у найбільшому числі масиву.

2. Для кожного розряду i від 1 до d (від найменш значущого до найбільш значущого):відсортувати весь масив за і-м розрядом, використовуючи стабільний алгоритм сортування.

Ключовим моментом є стабільність підпрограми сортування. Коли масив сортується за наступним, більш значущим розрядом, відносний порядок елементів, що мають однакову цифру в цьому розряді, зберігається. Цей порядок був установлений на попередніх кроках сортування за менш значущими розрядами. Так, після сортування за останнім, найбільш значущим розрядом, весь масив стає повністю відсортованим.

Відсортуємо масив. Максимальне число 802 має 3 розряди.

Сортування за одиницями (1-й розряд):

Ключі: 0, 5, 5, 0, 2, 4, 2, 6

Результат (після стабільного сортування): [**170, 90**], [80**2**, **2**], [2**4**], [4**5**, 7**5**], [6**6**]

Сортування за десятками (2-й розряд):

Ключі: 7, 9, 0, 0, 2, 4, 7, 6

Сортуємо попередній результат: [8**0**2, **0**2], [**2**4], [**4**5], [**6**6], [1**7**0, **7**5], [**9**0]

Сортування за сотнями (3-й розряд):

Ключі: 1, 0, 0, 0, 8, 0, 0, 0 (для чисел < 100 розряд сотень дорівнює 0)

Сортуємо попередній результат: [**0**02, **0**24, **0**45, **0**66, **0**75, **0**90], [**1**70], [**8**02]

Масив відсортовано.

Часова складність − O(d · (n + b)), де d − кількість розрядів, n − кількість елементів, b − основа системи числення (radix, зазвичай 10 для десяткових чисел). Якщо d є константою, а b не є значно більшим за n (наприклад, b = O(n)), то складність є лінійною відносно n.

Просторова складність − O(n + b), оскільки вона визначається просторовою складністю стабільного сортування, що використовується як підпрограма (наприклад, сортування підрахунком).

## 7.8. Практична реалізація та порівняльний аналіз

Теоретичний аналіз дає важливе розуміння асимптотичної поведінки алгоритмів, але на практиці вибір оптимального алгоритму залежить від багатьох чинників: характеристик даних, об’єму пам’яті, вимог до стабільності та особливостей апаратної платформи.

QuickSort та MergeSort є двома найважливішими алгоритмами сортування, що базуються на принципі «Розділяй і володарюй». Хоча обидва мають середню часову складність O(nlogn), їхні практичні характеристики роблять їх придатними для різних завдань.

Таблиця 7.2 – QuickSort і MergeSort – Практичне порівняння

| Критерій | QuickSort | MergeSort |
|---|---|---|
| Часова складність (найкращий випадок) | O(nlogn) | O(nlogn) |
| Часова складність (середній випадок) | O(nlogn) | O(nlogn) |
| Часова складність (найгірший випадок) | $O(n^2)$ | O(nlogn) |
| Просторова складність | O(logn) (для стека рекурсії) | O(n) (для допоміжного масиву) |
| Сортування «на місці» (In-Place) | Так | Ні |
| Стабільність | Ні (у типових реалізаціях) | Так |
| Локальність посилань (Cache Efficiency) | Висока (послідовний доступ до пам’яті) | Низька (копіювання в допоміжний масив) |
| Переважна структура даних | Масиви (вимагає довільного доступу) | Зв’язні списки (не вимагає довільного доступу) |
| Основні сценарії використання | Сортування масивів у пам’яті, де швидкість у середньому є пріоритетом | Зовнішнє сортування, сортування зв’язних списків, коли потрібна стабільність або гарантія продуктивності |

На практиці, для сортування масивів у оперативній пам’яті, добре реалізований QuickSort (з рандомізованим вибором опорного елемента або медіаною трьох) часто виявляється швидшим за MergeSort. Це пов’язано з його кращою локальністю посилань та відсутністю потреби в копіюванні даних у допоміжний масив, що зменшує накладні витрати.

З іншого боку, MergeSort є незамінним, коли:

1. Потрібна гарантія продуктивності. Його складність O(*nlogn*) у найгіршому випадку робить його надійним вибором для критичних систем.

2. Потрібна стабільність − це одна з ключових переваг MergeSort.

3. Сортуються зв’язні списки. MergeSort не вимагає довільного доступу до елементів, а операція злиття для зв’язних списків може бути реалізована дуже ефективно, без потреби в додаткових O(n) пам’яті.

4. Виконується зовнішнє сортування. Коли дані не вміщуються в оперативну пам’ять і зберігаються на диску, MergeSort є ідеальним вибором, оскільки він працює з послідовними блоками даних, мінімізуючи кількість дорогих операцій читання / запису з диска.

Реалізація QuickSort

```
Псевдокод (схема Ломуто)
procedure quickSort(A, low, high)
  if low < high then
    pi := partition(A, low, high)
    quickSort(A, low, pi - 1)
    quickSort(A, pi + 1, high)
  end if
end procedure

procedure partition(A, low, high)
  pivot := A[high]
  i := low - 1
  for j := low to high - 1 do
    if A[j] < pivot then
      i := i + 1
      swap(A[i], A[j])
    end if
  end for
  swap(A[i + 1], A[high])
  return (i + 1)
end procedure
```

```
Python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            (arr[i], arr[j]) = (arr[j], arr[i])
    (arr[i + 1], arr[high]) = (arr[high], arr[i + 1])
    return i + 1

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

```
C++
#include<vector>
#include<utility>

int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Реалізація MergeSort

```
Псевдокод
procedure mergeSort(A, left, right)
  if left < right then
    mid := floor((left + right) / 2)
    mergeSort(A, left, mid)
    mergeSort(A, mid + 1, right)
    merge(A, left, mid, right)
  end if
end procedure

procedure merge(A, left, mid, right)
  //... логіка злиття з використанням допоміжних масивів...
end procedure
```

```
Python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

```
C++
#include<vector>

void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<int> L(n1), R(n2);
```

```
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Реалізація сортування підрахунком

*Псевдокод*

```
procedure countingSort(A, B, k)
  let C[0..k] be a new array
  for i := 0 to k do
    C[i] := 0
  end for
  for j := 1 to length(A) do
    C[A[j]] := C[A[j]] + 1
  end for
  // C[i] now contains the number of elements equal to i
  for i := 1 to k do
    C[i] := C[i] + C[i-1]
  end for
  // C[i] now contains the number of elements less than or equal to i
  for j := length(A) downto 1 do
    B[C[A[j]]] := A[j]
    C[A[j]] := C[A[j]] - 1
  end for
end procedure
```

*Python*

```
def countingSort(arr):
    max_val = max(arr)
    min_val = min(arr)
    range_of_elements = max_val - min_val + 1

    count =  * range_of_elements
    output =  * len(arr)

    for i in range(len(arr)):
        count[arr[i] - min_val] += 1

    for i in range(1, len(count)):
        count[i] += count[i - 1]

    for i in range(len(arr) - 1, -1, -1):
        output[count[arr[i] - min_val] = arr[i]
        count[arr[i] - min_val] -= 1

    for i in range(len(arr)):
        arr[i] = output[i]
    return arr
```

```
C++
#include <vector>
#include <algorithm>

void countingSort(std::vector<int>& arr) {
    if (arr.empty()) return;

    int max_val = *std::max_element(arr.begin(), arr.end());
    int min_val = *std::min_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    std::vector<int> count(range, 0);
    std::vector<int> output(arr.size());

    for (int i = 0; i < arr.size(); i++) {
        count[arr[i] - min_val]++;
    }

    for (int i = 1; i < count.size(); i++) {
        count[i] += count[i - 1];
    }

    for (int i = arr.size() - 1; i >= 0; i--) {
        output[count[arr[i] - min_val] = arr[i];
        count[arr[i] - min_val]--;
    }

    for (int i = 0; i < arr.size(); i++) {
        arr[i] = output[i];
    }
}
```

Таблиця 7.3 – Комплексна зведена таблиця
алгоритмів сортування

| Алгоритм | Часова складність (найкращий випадок) | Часова складність (середній випадок) | Часова складність (найгірший випадок) | Просторова складність | Стабільність | «На місці» |
|---|---|---|---|---|---|---|
| Bubble Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | Так | Так |
| Selection Sort | $\Omega(n^2)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | Ні | Так |
| Insertion Sort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | Так | Так |
| Merge Sort | $\Omega(n\log n)$ | $\Theta(n\log n)$ | $O(n\log n)$ | $O(n)$ | Так | Ні |
| Quick Sort | $\Omega(n\log n)$ | $\Theta(n\log n)$ | $O(n^2)$ | $O(\log n)$ | Ні | Так |
| Heap Sort | $\Omega(n\log n)$ | $\Theta(n\log n)$ | $O(n\log n)$ | $O(1)$ | Ні | Так |
| Counting Sort | $\Omega(n + k)$ | $\Theta(n + k)$ | $O(n + k)$ | $O(n + k)$ | Так | Ні |
| Radix Sort | $\Omega(d(n + b))$ | $\Theta(d(n + b))$ | $O(d(n + b))$ | $O(n + b)$ | Так | Ні |

## 7.9. Приклади розв’язку задач

**Задача 1.** Сортування майже відсортованих даних. Вам надано масив з 1 мільйона записів, де менше ніж 100 елементів містяться не на своїх місцях. Який алгоритм сортування буде найефективнішим для цього завдання і чому?

*Розв’язок*

Найефективнішим алгоритмом буде сортування вставками (Insertion Sort). Його часова складність у загальному випадку залежить від кількості інверсій у масиві (пар елементів, що стоять у неправильному порядку). Для майже відсортованого масиву кількість інверсій є дуже малою. У найкращому випадку (повністю відсортований масив) складність Insertion Sort становить O(n). У

цьому сценарії, оскільки кількість «неправильних» елементів мала, час роботи буде близьким до лінійного, що значно швидше за O(nlogn) алгоритми, які виконають повний цикл поділу та злиття незалежно від початкового стану масиву.

**Задача 2.** Сортування з вимогою стабільності. У вас є список об'єктів-студентів, який вже відсортовано за прізвищем в алфавітному порядку. Тепер потрібно відсортувати цей список за середнім балом (за спаданням), але водночас зберегти алфавітний порядок для студентів, які мають однаковий середній бал. Який алгоритм потрібно обрати?

*Розв'язок*

Для цього завдання потрібен стабільний алгоритм сортування. Ідеальними кандидатами є сортування злиттям (Merge Sort) або сортування вставками (Insertion Sort). Merge Sort є кращим вибором для великих наборів даних, оскільки він гарантує часову складність O(nlogn) і є стабільним. QuickSort або Heapsort використовувати не можна, оскільки вони не є стабільними і можуть порушити початковий алфавітний порядок студентів з однаковими балами.

**Задача 3.** Сортування великого масиву цілих чисел. Потрібно відсортувати масив, що містить 10 мільйонів 32-бітних цілих чисел (значення від 0 до 232−1). Який алгоритм буде найшвидшим?

*Розв'язок*

Найшвидшим алгоритмом буде розрядне сортування (Radix **Sort)**. Оскільки ми маємо справу з цілими числами фіксованої довжини (32 біти), кількість розрядів d є константою. Можна обрати основу системи числення b (наприклад, b = 216), щоб обробляти по 16 бітів за прохід (d = 2). Часова складність буде $O(d \cdot (n + b))$, що є лінійною відносно n. Це буде значно швидше, ніж будь-який порівняльний алгоритм зі складністю O(nlogn) на такому великому обсязі даних.

## Контрольні питання

1. Поясніть різницю між стабільним та нестабільним алгоритмом сортування. Наведіть приклад, де стабільність є критично важливою.

2. Опишіть принцип «Розділяй і володарюй» на прикладі алгоритму Merge Sort, детально пояснивши кроки поділу та злиття.

3. Чому найгірший випадок часової складності для QuickSort становить $O(n^2)$ і як цього можна уникнути на практиці?

4. Що таке модель дерева рішень для порівняльних сортувань і який фундаментальний висновок про їхню часову складність можна з неї зробити?

5. Яка основна теоретична перевага Heapsort над Quicksort, і чому Quicksort часто швидший на практиці?

6. За яких умов сортування підрахунком (Counting Sort) буде асимптотично ефективнішим за MergeSort?

7. Який з алгоритмів сортування з квадратичною складністю ($O(n^2)$) виконує мінімальну кількість операцій обміну (swaps)?

8. Поясніть, чому для коректної роботи Radix Sort потрібно використовувати стабільний алгоритм сортування як підпрограму.

9. Порівняйте просторову складність алгоритмів MergeSort, QuickSort та Heapsort. Який з них є найбільш ефективним з точки зору використання пам’яті?

10. Який з $O(n^2)$ алгоритмів найкраще підходить для сортування масиву, який вже є майже відсортованим? Поясніть чому.

11. Опишіть роботу процедури partition у схемі Ломуто для QuickSort.

12. Чому MergeSort вважається добрим вибором для сортування зв’язних списків, на відміну від QuickSort?

13. Поясніть два основні етапи алгоритму Heapsort: побудова купи та вилучення елементів.

14. Які основні обмеження має сортування підрахунком, що роблять його непридатним для деяких задач?

15. Що означає, що алгоритм сортування є «in-place» (сортує на місці)? Наведіть приклади таких алгоритмів.

## Задачі для самостійного опрацювання

1. Реалізуйте алгоритм QuickSort, використовуючи стратегію вибору опорного елемента «медіана трьох» для покращення продуктивності в середньому випадку.

2. Модифікуйте стандартну реалізацію сортування вибором (Selection Sort) так, щоб вона стала стабільною. Проаналізуйте, як змінилася кількість операцій.

3. Напишіть функцію, що сортує масив, який містить лише числа 0, 1 та 2, за лінійний час O(n) та з константною додатковою пам'яттю O(1) (задача «голландського національного прапора»).

4. Реалізуйте Radix Sort для сортування масиву рядків однакової довжини в алфавітному порядку.

5. Напишіть функцію для знаходження k-го найбільшого елемента в масиві за середній час O(n), не сортуючи весь масив. (Підказка: використайте ідею процедури partition з QuickSort).

6. Реалізуйте алгоритм сортування злиттям (Merge Sort) для двобічно зв'язного списку.

7. Напишіть програму, яка ефективно визначає кількість інверсій у масиві (пар індексів $i < j$ таких, що $A[i] > A[j]$). (Підказка: модифікуйте алгоритм MergeSort).

8. Реалізуйте сортування бульбашкою з двонаправленими проходами, відоме як Cocktail Shaker Sort. Проаналізуйте його ефективність порівняно зі звичайним Bubble Sort.

9. Проведіть експериментальне порівняння часу роботи QuickSort та MergeSort на трьох типах масивів різного розміру (103, 105, 107 елементів): а) випадково згенерованих; б) вже відсортованих; в) обернено відсортованих. Поясніть отримані результати.

10. Реалізуйте Heapsort з використанням min-heap для сортування масиву за спаданням.

11. Напишіть функцію, що ефективно зливає k відсортованих масивів в один великий відсортований масив.

12. Надайте математичне доведення того, що процедура побудови купи (buildHeap) в алгоритмі Heapsort має лінійну часову складність O(n).

13. Реалізуйте кошикове сортування (Bucket Sort) для сортування масиву дійсних чисел, які рівномірно розподілені в діапазоні [0, 1).

14. Дослідіть та реалізуйте стабільну версію алгоритму QuickSort.

15. Проаналізуйте, який алгоритм сортування (або комбінація алгоритмів) найкраще підійде для сортування дуже великого файлу (наприклад, 100 ГБ), який не вміщується в оперативну пам’ять. Обґрунтуйте свій вибір, враховуючи повільні операції читання / запису з диска.

# ТЕМА 8
# СТРАТЕГІЯ «РОЗДІЛЯЙ ТА ВОЛОДАРЮЙ»

Парадигма «розділяй та володарюй» (англ. Divide and Conquer) є однією з трьох фундаментальних стратегій розробки алгоритмів[25], поряд з динамічним програмуванням та жадібними алгоритмами. Її суть полягає в декомпозиції складної задачі на простіші, однотипні підзадачі, розв'язання цих підзадач та комбінування їх результатів для отримання остаточного розв'язку. Цей підхід вирізняється своєю елегантністю та потужністю, даючи можливість розв'язувати широкий клас обчислювальних задач зі значно вищою ефективністю порівняно з прямолінійними, або «наївними», методами.

Історично, сам принцип «розділяй та володарюй» (лат. *divide et impera*) є давньою політичною та військовою стратегією, що передбачала поділ великої концентрації влади на менші, менш потужні групи для полегшення керування ними. В інформатиці ця концепція знайшла своє потужне втілення, але її метою стало керування не людьми, а обчислювальною складністю. Замість того, щоб розв'язувати велику, громіздку задачу «в лоб», ми розбиваємо її на частини, доки вони не стануть тривіальними, а потім збираємо з цих простих рішень загальну відповідь.

## 8.1. Основні принципи парадигми

Будь-який алгоритм, що базується на стратегії «розділяй та володарюй», складається з трьох послідовних кроків, які застосовуються рекурсивно.

1. Розділення (Divide). На цьому етапі вихідна задача розмірності n розбивається на a незалежних підзадач, кожна з яких має розмірність n/b. Критично важливо, щоб кожна підзадача була меншою версією вихідної задачі, тобто мала ту саму структуру, що дає змогу застосовувати до неї той самий алгоритм рекурсивно.

[25] Три фундаментальні стратегії розробки алгоритмів – це метод «розділяй і володарюй», динамічне програмування та жадібний підхід. Кожна з них застосовується для вирішення різних типів обчислювальних задач.

2. Володарювання (Conquer). Кожна з а підзадач розв'язується. Якщо підзадача все ще є достатньо великою, до неї рекурсивно застосовується той самий алгоритм. Цей процес триває доти, доки розмір підзадачі не досягне так званого базового випадку − межі рекурсії, коли задача стає настільки простою, що її можна розв'язати безпосередньо, без подальшого розбиття. Наприклад, для задачі сортування базовим випадком може бути масив з одного елемента, який за визначенням є відсортованим.

3. Комбінування (Combine). Розв'язки, отримані для кожної з підзадач на етапі володарювання, об'єднуються для формування розв'язку вихідної, більшої задачі. Складність цього етапу може варіюватися від тривіальної (як у бінарному пошуку) до досить обчислювально місткої (як у сортуванні злиттям) і є ключовим чинником, що впливає на загальну ефективність алгоритму.

Цей процес можна візуалізувати у вигляді дерева рекурсії, де корінь представляє вихідну задачу, внутрішні вузли − підзадачі, що утворюються в процесі поділу, а листки − базові випадки рекурсії. Процес розв'язання рухається від кореня до листків (розділення), а потім результати збираються від листків до кореня (комбінування).

Загальна схема парадигми «Розділяй та володарюй»

Фрагмент коду

```
graph TD
    A[Задача (розмір n)] --> B1[Підзадача 1 (розмір n/b)];
    A --> B2[Підзадача 2 (розмір n/b)];
    A --> B_dots[...]
    A --> Ba[Підзадача a (розмір n/b)];

    B1 --> C1[...];
    B2 --> C2[...];
    Ba --> Ca[...];

    C1 -- Розв'язок --> D1[Комбінування];
    C2 -- Розв'язок --> D1;
    Ca -- Розв'язок --> D1;

    D1 -- Розв'язок --> A;
```

```
subgraph "Етап 1: Розділення"
   A
end

subgraph "Етап 2: Володарювання (рекурсивне розв'язання)"
   B1
   B2
   B_dots
   Ba
   C1
   C2
   Ca
end

subgraph "Етап 3: Комбінування"
   D1
end
```

Оскільки алгоритми типу «розділяй та володарюй» за своєю природою є рекурсивними, їх часова складність T(n) найзручніше описується за допомогою рекурентного співвідношення. Загальна форма такого співвідношення виглядає так:

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

де: $T(n)$ − час виконання алгоритму для задачі розміром n;
$a$ − кількість підзадач, на які розбивається вихідна задача на кожному кроці рекурсії ($a \geq 1$);
$n/b$ − розмір кожної підзадачі (передбачається, що всі підзадачі мають однаковий розмір, $b > 1$);
$f(n)$ − час, що витрачається на етапи розділення та комбінування на одному рівні рекурсії.

Саме функція f(n) часто визначає кінцеву асимптотичну складність алгоритму. Наприклад, в алгоритмі сортування злиттям масив ділиться на дві частини ($a = 2, b = 2$), а етап злиття двох відсортованих половин вимагає лінійного часу, тому f(n) = O(n). Як буде показано далі, це призводить до загальної складності O($nlogn$).

Розв’язання рекурентних співвідношень за рахунок прямого аналізу дерева рекурсії може бути громіздким. На щастя, для великого класу співвідношень виду $T(n) = aT(n/b) + f(n)$ існує потужний інструмент − *Майстер-теорема* (також відома як основна теорема про рекуренції), яка дає змогу отримати асимптотичну оцінку T(n) майже механічно.

Теорема розглядає співвідношення між зростанням функції f(n) (вартість розділення / комбінування) та величиною $n^{\log_b a}$ (кількість листків у дереві рекурсії). Залежно від того, яка з цих функцій домінує, виділяють три випадки.

1. Випадок 1. Якщо $f(n) = O(n^{\log_b a - \epsilon})$ для деякої сталої $\epsilon > 0$. Це означає, що вартість роботи на етапі розділення / комбінування є поліноміально меншою, ніж кількість листків. У цьому випадку обчислювальна вартість алгоритму концентрується в листках дерева рекурсії, і загальний час визначається їх кількістю.

Розв’язок: $T(n) = \Theta(n^{\log_b a})$.

2. Випадок 2. Якщо $f(n) = \Theta(n^{\log_b a}\log^k n)$ для деякої сталої $k \geq 0$.

Це означає, що вартість роботи приблизно однакова на всіх рівнях дерева рекурсії. Загальний час є добутком вартості роботи на одному рівні на кількість рівнів (яка дорівнює logn).

Розв’язок: $T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$.

3. Випадок 3. Якщо $f(n) = \Omega(n^{\log_b a + \epsilon})$ для деякої сталої $\epsilon > 0$, і додатково виконується умова регулярності: $af(n/b) \leq cf(n)$ для деякої сталої $c < 1$ та достатньо великих n. Це означає, що вартість роботи на етапі розділення / комбінування поліноміально більша, ніж кількість листків. У цьому випадку обчислювальна вартість концентрується в корені дерева (тобто на першому кроці розділення / комбінування), і вона домінує над усією рештою роботи.

Розв’язок: $T(n) = \Theta(f(n))$.

Важливо зазначити, що Майстер-теорема не є всеохоплювальною. Існують «проміжки» між випадками, а також ситуації, коли її умови не виконуються (наприклад, якщо a не є константою або f(n) не є поліноміально порівнянною з $n^{\log_b a}$).

Ефективність алгоритму, розробленого за парадигмою «Розділяй та володарюй», залежить від тонкого балансу між вартістю рекурсивного розбиття та вартістю подальшого «склеювання»

результатів. Майстер-теорема є формальним вираженням цього балансу. Функція f(n) у рекурентному співвідношенні є математичною моделлю кроків «розділяй» та «комбінуй», тоді як *aT(n/b)* моделює крок «володарюй». Так, Майстер-теорема не просто розв'язує рівняння, а відповідає на фундаментальне питання: «Що є обчислювальним вузьким місцем в даному алгоритмі?». Якщо домінує випадок 1, вузьке місце − це величезна кількість базових операцій у листках. Якщо домінує випадок 3 − вузьке місце у вартості самого процесу злиття. Випадок 2 представляє ідеально збалансований алгоритм, де робота рівномірно розподілена по всьому дереву рекурсії.

Таблиця 8.1 – Зведена таблиця Майстер-теореми

| Випадок | Умова на f(n) | Результат T(n) | Інтуїтивне пояснення |
|---|---|---|---|
| 1 | $f(n) = O(n^{\log_b a - \epsilon})$ для $\epsilon > 0$ | $\Theta(n^{\log_b a})$ | Робота домінує в листках дерева рекурсії |
| 2 | $f(n) = \Theta(n^{\log_b a} \log^k n)$ для $k \geq 0$ | $\Theta(n^{\log_b a} \log^{k+1} n)$ | Робота рівномірно розподілена по рівнях |
| 3 | $f(n) = \Omega(n^{\log_b a + \epsilon})$ та умова регулярності | $\Theta(f(n))$ | Робота домінує в корені (розділення / комбінування) |

## 8.2. Алгоритм Карацуби для множення

Класичний метод множення двох n-розрядних чисел, відомий зі школи як множення «у стовпчик», вимагає виконання $n^2$ операцій множення однорозрядних чисел. Наприклад, для множення двох 4-значних чисел потрібно 4 × 4 = 16 множень. Це призводить до часової складності алгоритму $O(n^2)$. Для невеликих чисел це цілком прийнятно, але під час роботи з багаторозрядними числами, наприклад, у криптографії, така складність стає обмежувальним чинником.

1960 року радянський математик Анатолій Карацуба запропонував революційний підхід, що дав змогу знизити кількість множень. Ідея полягає в тому, щоб замінити одну дорогу операцію множення на декілька дешевших операцій додавання та віднімання.

Розглянемо два n-розрядні числа X та Y. Припустимо, що n є степенем двійки. Ми можемо розбити кожне число на дві половини по n/2 розрядів:

$X = a \cdot 10^{n/2} + b.$

$Y = c \cdot 10^{n/2} + d.$

Тоді їх добуток можна записати так:

$$X \cdot Y = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) = ac \cdot 10^{n} + (ad + bc) \cdot 10^{n/2} + bd.$$

На перший погляд, для обчислення цього виразу потрібно виконати чотири множення чисел розміром n/2: ac, ad, bc та bd. Це не дає виграшу в асимптотиці.

Ключова ідея Карацуби полягає в тому, щоб обчислити середній член (ad + bc) хитрішим способом. Замість чотирьох множень, він запропонував обчислювати лише три:

1. $P1 = a \cdot c.$
2. $P2 = b \cdot d.$
3. $P3 = (a + b) \cdot (c + d).$

Тепер розкриємо P3: $P3 = ac + ad + bc + bd = P1 + (ad + bc) + P2$. Звідси можна виразити потрібний нам середній член:

$$Ad + bc = P3 - P1 - P2.$$

Так, кінцева формула для добутку X·Y виражається через три множення P1, P2, P3 та кілька операцій додавання / віднімання:

$$X \cdot Y = P1 \cdot 10^{n} + (P3 - P1 - P2) \cdot 10^{n/2} + P2.$$

Алгоритм Карацуби застосовує цю формулу рекурсивно. Для обчислення добутків P1, P2 та P3 (які є множенням чисел розміром n/2) знову викликається алгоритм Карацуби. Рекурсія зупиняється, коли числа стають достатньо малими (наприклад, однорозрядними), і їх добуток обчислюється напряму.

Часову складність цього алгоритму можна описати рекурентним співвідношенням:

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

Тут $3T(n/2)$ відповідає трьом рекурсивним викликам для множення чисел удвічі меншого розміру, а $O(n)$ представляє вартість операцій додавання, віднімання та зсувів (множення на степені 10, що є швидкою операцією) на кожному кроці рекурсії.

Для аналізу цього співвідношення застосуємо Майстер-теорему. Маємо $a = 3$, $b = 2$, $f(n) = O(n)$. Обчислимо критичний показник степеня: $\log_b a = \log_2 3 \approx 1.585$.

Оскільки $f(n) = O(n) = O(n^1)$, що є поліноміально меншим за $n^{1.585}$ (тобто $f(n) = O(n^{\log_2 3 - \epsilon})$ для $\epsilon \approx 0.585$), ми потрапляємо у Випадок 1 Майстер-теореми.

Отже, часова складність алгоритму Карацуби становить:

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

Цей результат є значно кращим за $O(n^2)$ для наївного методу, особливо для великих значень n. Алгоритм Карацуби є яскравим прикладом того, як зміна математичного підходу до проблеми перед застосуванням обчислювальної техніки (рекурсії) може призвести до прориву в ефективності. Зменшення кількості рекурсивних викликів з 4 до 3 є прямою причиною зміни основи логарифма в кінцевій складності з $\log_2 4 = 2$ до $\log_2 3 \approx 1.585$. Це демонструє, наскільки потужно коефіцієнт a у Майстер-теоремі впливає на асимптотику.

Таблиця 8.2 – Порівняння наївного множення та алгоритму Карацуби

| Характеристика | Наївне множення «у стовпчик» | Алгоритм Карацуби |
|---|---|---|
| Кількість множень | $n^2$ | 3 рекурсивні виклики на кожному кроці |
| Часова складність | $O(n^2)$ | $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$ |
| Парадигма | Ітеративна | Розділяй та володарюй |
| Переваги | Проста реалізація | Асимптотично швидший для великих чисел |
| Недоліки | Неефективний для великих n | Більші накладні витрати для малих n |

Під час реалізації важливо враховувати обробку чисел непарної довжини та чисел різної довжини, що зазвичай вирішується доповненням коротшого числа провідними нулями.

*Псевдокод*

```
FUNCTION karatsuba(num1, num2)
   IF num1 < 10 OR num2 < 10 THEN
      RETURN num1 * num2
   END IF

   // Визначення довжини та середини
   m = max(length(num1), length(num2))
   m2 = floor(m / 2)

   // Розбиття чисел на частини
   high1 = num1 / 10^m2
   low1 = num1 % 10^m2
   high2 = num2 / 10^m2
   low2 = num2 % 10^m2

   // 3 рекурсивні виклики
   z0 = karatsuba(low1, low2)
   z2 = karatsuba(high1, high2)
   z1 = karatsuba(low1 + high1, low2 + high2)

   // Комбінування результатів за формулою
   RETURN (z2 * 10^(m2 * 2)) + ((z1 - z2 - z0) * 10^m2) + z0
END FUNCTION
```

*Python*

```
def karatsuba(x, y):
   """Множення двох чисел за алгоритмом Карацуби."""
   if len(str(x)) == 1 or len(str(y)) == 1:
      return x * y
   else:
      n = max(len(str(x)), len(str(y)))
      n2 = n // 2

      # Розбиття чисел на частини
      a = x // 10**(n2)
      b = x % 10**(n2)
      c = y // 10**(n2)
      d = y % 10**(n2)
```

```
    # Рекурсивні виклики
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    ad_plus_bc = karatsuba(a + b, c + d) - ac - bd

    # Комбінування результату
    prod = (ac * 10**(2 * n2)) + (ad_plus_bc * 10**n2) + bd
    return prod

# Приклад
print(karatsuba(12345678, 87654321))
```

```
C++
#include <iostream>
#include <string>
#include <algorithm>
#include <cmath>

// Функція для додавання двох великих чисел, представлених рядками
std::string add(std::string s1, std::string s2) {
   std::string result = "";
   int i = s1.length() - 1, j = s2.length() - 1, carry = 0;
   while (i > = 0 |

| j > = 0 |
| carry) {
      int sum = carry + (i > = 0? s1[i--] - '0' : 0) + (j >= 0? s2[j--] - '0' : 0);
      result = std::to_string(sum % 10) + result;
      carry = sum / 10;
   }
   return result;
}

// Функція для віднімання двох великих чисел (s1 > = s2)
std::string subtract(std::string s1, std::string s2) {
   std::string result = "";
   int n1 = s1.length(), n2 = s2.length();
   std::reverse(s1.begin(), s1.end());
   std::reverse(s2.begin(), s2.end());
   int carry = 0;
```

```
        for (int i = 0; i < n2; i++) {
            int sub = ((s1[i] - '0') - (s2[i] - '0') - carry);
            if (sub < 0) {
                sub = sub + 10;
                carry = 1;
            } else {
                carry = 0;
            }
            result.push_back(sub + '0');
        }
        for (int i = n2; i < n1; i++) {
            int sub = ((s1[i] - '0') - carry);
            if (sub < 0) {
                sub = sub + 10;
                carry = 1;
            } else {
                carry = 0;
            }
            result.push_back(sub + '0');
        }
        std::reverse(result.begin(), result.end());
        // Видалити провідні нулі
        size_t first_digit = result.find_first_not_of('0');
        if (std::string::npos!= first_digit) {
            return result.substr(first_digit);
        }
        return "0";
    }

    std::string karatsuba(std::string x, std::string y) {
        int n = std::max(x.length(), y.length());

        // Доповнення нулями до однакової довжини, яка є степенем двійки
(для простоти)
        while (x.length() < n) x = "0" + x;
        while (y.length() < n) y = "0" + y;

        if (n == 1) return std::to_string((x - '0') * (y - '0'));
        if (n == 0) return "0";

        int n2 = n / 2;
        int n1 = n - n2;
```

```
    std::string a = x.substr(0, n1);
    std::string b = x.substr(n1);
    std::string c = y.substr(0, n1);
    std::string d = y.substr(n1);

    std::string ac = karatsuba(a, c);
    std::string bd = karatsuba(b, d);
    std::string ab_cd = karatsuba(add(a, b), add(c, d));

    std::string ad_plus_bc = subtract(ab_cd, add(ac, bd));

    for (int i = 0; i < 2 * n2; i++) ac += "0";
    for (int i = 0; i < n2; i++) ad_plus_bc += "0";

    return add(add(ac, ad_plus_bc), bd);
}

int main() {
    std::string num1 = "12345678";
    std::string num2 = "87654321";
    std::cout << "Product is: " << karatsuba(num1, num2) << std::endl;
    return 0;
}
```

## 8.3. Матричне множення: алгоритм Штрассена

Множення двох квадратних матриць розміром n × n є однією з фундаментальних операцій у лінійній алгебрі та багатьох галузях обчислювальної науки. Стандартний алгоритм, що випливає безпосередньо з визначення, вимагає трьох вкладених циклів для обчислення кожного з n2 елементів результуючої матриці. Це призводить до часової складності $O(n^3)$. Подібно до множення чисел, виникає питання: чи можна робити це швидше?

1969 року Фолькер Штрассен запропонував алгоритм, який застосовує філософію, подібну до ідеї Карацуби, але в контексті матриць. Алгоритм базується на методі «розділяй та володарюй».

Розглянемо дві матриці A та B розміром n × n, де n є степенем двійки. Розділимо кожну з них на чотири підматриці (блоки) розміром n/2 × n/2:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \; C = A \cdot B = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Згідно зі стандартним блоковим множенням, блоки результуючої матриці C обчислюються так:

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$.

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$.

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$.

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$.

Це вимагає 8 рекурсивних множень матриць розміром n/2 × n/2 та 4 додавання. Рекурентне співвідношення для такого підходу $T(n) = 8T(n/2) + O(n^2)$ за Майстер-теоремою дає розв'язок $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$, що не покращує асимптотику.

Штрассен знайшов спосіб обчислити блоки $C_{ij}$ за допомогою лише семи множень підматриць, але ціною збільшення кількості додавань та віднімань.

Алгоритм Штрассена вводить 7 проміжних матриць $P_1,\ldots, P_7$, які обчислюються за допомогою одного множення кожна. Потім блоки результуючої матриці C конструюються за допомогою лінійних комбінацій цих проміжних матриць.

Таблиця 8.3 – Формули алгоритму Штрассена

| Проміжні матриці (7 множень) | Результуючі блоки (18 додавань / віднімань) |
|---|---|
| $P_1 = A_{11} \cdot (B_{12} - B_{22})$ | $C_{11} = P_5 + P_4 - P_2 + P_6$ |
| $P_2 = (A_{11} + A_{12}) \cdot B_{22}$ | $C_{12} = P_1 + P_2$ |
| $P_3 = (A_{21} + A_{22}) \cdot B_{11}$ | $C_{21} = P_3 + P_4$ |
| $P_4 = A_{22} \cdot (B_{21} - B_{11})$ | $C_{22} = P_5 + P_1 - P_3 - P_7$ |
| $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$ | |
| $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$ | |
| $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$ | |

Примітка. Існує кілька еквівалентних наборів формул. Наведений варіант є одним з найпоширеніших.

Рекурентне співвідношення для часової складності алгоритму Штрассена має вигляд:

$$T(n) = 7T(n/2) + O(n^2).$$

Тут $7T(n/2)$ відповідає семи рекурсивним викликам для множення матриць розміром $n/2 \times n/2$, а $O(n^2)$ − вартості 18 операцій додавання / віднімання матриць такого самого розміру (кожна з яких вимагає $(n/2)^2$ операцій).

Застосуємо Майстер-теорему: $a = 7$, $b = 2$, $f(n) = O(n^2)$. Критичний показник: $\log_b a = \log_2 7 \approx 2.807$.

Оскільки $f(n) = O(n^2)$ є поліноміально меншим за $n^{\log_2 7}$ (тобто $f(n) = O(n^{\log_2 7 - \epsilon})$ для $\epsilon \approx 0.807$), ми знову потрапляємо у Випадок 1.

Отже, часова складність алгоритму Штрассена:

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807}).$$

Цей результат є асимптотично кращим за $O(n^3)$. Однак на практиці алгоритм Штрассена має більші приховані константи через значну кількість операцій додавання. Тому він стає ефективнішим за стандартний метод лише для матриць досить великого розміру. Ця межа, відома як точка перетину (crossover point), залежить від реалізації та апаратної платформи, але зазвичай перебуває в діапазоні розмірів матриць від $32 \times 32$ до $128 \times 128$. Ефективні реалізації алгоритму Штрассена є гібридними: вони використовують рекурсію до досягнення цієї точки, а потім переходять на стандартний алгоритм $O(n^3)$ для множення малих підматриць.

Існування алгоритму Штрассена стало проривом, який показав, що складність $O(n^3)$ не є фундаментальною межею для множення матриць. Це стимулювало подальші дослідження, що призвели до створення ще складніших, але асимптотично швидших алгоритмів (наприклад, Копперсміта-Винограда). Це демонструє, що навіть для фундаментальних операцій, які здаються давно «вирішеними», можуть існувати неінтуїтивні шляхи до підвищення ефективності, і що теоретична інформатика є живою дисципліною з відкритими питаннями.

Реалізація алгоритму вимагає рекурсивної функції, яка розбиває матриці на квадранти, виконує 7 рекурсивних викликів та комбінує результати. Для матриць, розмір яких не є степенем двійки, застосовується доповнення нулями (padding) до найближчого степеня двійки.

*Псевдокод*

```
FUNCTION strassen(A, B)
   n = A.rows
   IF n <= LEAF_SIZE THEN
      RETURN standard_multiply(A, B)
   END IF

   // Розбиття матриць на 4 підматриці n/2 x n/2
   A11, A12, A21, A22 = partition(A)
   B11, B12, B21, B22 = partition(B)

   // 7 рекурсивних викликів за формулами Штрассена
   P1 = strassen(A11, B12 - B22)
   P2 = strassen(A11 + A12, B22)
   //... і так далі для P3..P7

   // Обчислення результуючих блоків Cij
   C11 = P5 + P4 - P2 + P6
   C12 = P1 + P2
   //... і так далі для C21, C22

   // Комбінування блоків у результуючу матрицю C
   C = combine(C11, C12, C21, C22)
   RETURN C
END FUNCTION
```

*Python*

```
import numpy as np

def strassen(A, B):
   """Множення матриць за алгоритмом Штрассена."""
   n = len(A)

   # Базовий випадок для рекурсії
   if n <= 64: # LEAF_SIZE
      return np.dot(A, B)
```

```
    # Розбиття матриць на 4 підматриці
    mid = n // 2
    A11, A12 = A[:mid, :mid], A[:mid, mid:]
    A21, A22 = A[mid:, :mid], A[mid:, mid:]
    B11, B12 = B[:mid, :mid], B[:mid, mid:]
    B21, B22 = B[mid:, :mid], B[mid:, mid:]

    # 7 рекурсивних викликів за формулами Штрассена
    P1 = strassen(A11, B12 - B22)
    P2 = strassen(A11 + A12, B22)
    P3 = strassen(A21 + A22, B11)
    P4 = strassen(A22, B21 - B11)
    P5 = strassen(A11 + A22, B11 + B22)
    P6 = strassen(A12 - A22, B21 + B22)
    P7 = strassen(A11 - A21, B11 + B12)

    # Обчислення результуючих блоків
    C11 = P5 + P4 - P2 + P6
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P5 + P1 - P3 - P7

    # Комбінування блоків у результуючу матрицю
    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
    return C

# Приклад використання
# n = 128
# A = np.random.rand(n, n)
# B = np.random.rand(n, n)
# C = strassen(A, B)
```

```
C++
#include<vector>
#include<iostream>

using Matrix = std::vector<std::vector<int>>;

// Функції для додавання та віднімання матриць
Matrix add(const Matrix& A, const Matrix& B) {
    int n = A.size();
```

```
    Matrix C(n, std::vector<int>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}

Matrix subtract(const Matrix& A, const Matrix& B) {
    int n = A.size();
    Matrix C(n, std::vector<int>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            C[i][j] = A[i][j] - B[i][j];
    return C;
}

Matrix strassen(const Matrix& A, const Matrix& B) {
    int n = A.size();
    if (n <= 64) { // LEAF_SIZE
        Matrix C(n, std::vector<int>(n, 0));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                for (int k = 0; k < n; ++k)
                    C[i][j] += A[i][k] * B[k][j];
        return C;
    }

    int mid = n / 2;
    Matrix A11(mid, std::vector<int>(mid)), A12(mid, std::vector<int>(mid)),
           A21(mid, std::vector<int>(mid)), A22(mid, std::vector<int>(mid));
    Matrix B11(mid, std::vector<int>(mid)), B12(mid, std::vector<int>(mid)),
           B21(mid, std::vector<int>(mid)), B22(mid, std::vector<int>(mid));

    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
            A11[i][j] = A[i][j]; A12[i][j] = A[i][j + mid];
            A21[i][j] = A[i + mid][j]; A22[i][j] = A[i + mid][j + mid];
            B11[i][j] = B[i][j]; B12[i][j] = B[i][j + mid];
            B21[i][j] = B[i + mid][j]; B22[i][j] = B[i + mid][j + mid];
        }
    }
```

```
    Matrix P1 = strassen(A11, subtract(B12, B22));
    Matrix P2 = strassen(add(A11, A12), B22);
    Matrix P3 = strassen(add(A21, A22), B11);
    Matrix P4 = strassen(A22, subtract(B21, B11));
    Matrix P5 = strassen(add(A11, A22), add(B11, B22));
    Matrix P6 = strassen(subtract(A12, A22), add(B21, B22));
    Matrix P7 = strassen(subtract(A11, A21), add(B11, B12));

    Matrix C11 = add(subtract(add(P5, P4), P2), P6);
    Matrix C12 = add(P1, P2);
    Matrix C21 = add(P3, P4);
    Matrix C22 = subtract(add(P5, P1), add(P3, P7));

    Matrix C(n, std::vector<int>(n));
    for (int i = 0; i < mid; ++i) {
        for (int j = 0; j < mid; ++j) {
            C[i][j] = C11[i][j]; C[i][j + mid] = C12[i][j];
            C[i + mid][j] = C21[i][j]; C[i + mid][j + mid] = C22[i][j];
        }
    }
    return C;
}
```

## 8.4. Пошук найближчої пари точок

Задача пошуку найближчої пари точок є класичною задачею обчислювальної геометрії. Формально вона звучить так: для заданої множини з n точок на евклідовій площині знайти таку пару точок (p, q), відстань між якими є мінімальною серед усіх можливих пар. Відстань між точками

$p_1 = (x_1, y_1)$ та $p_2 = (x_2, y_2)$ обчислюється за формулою:

$$d(p_1,p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Наївний алгоритм (brute-force) розв’язує цю задачу шляхом повного перебору: він обчислює відстані між кожною парою точок і знаходить мінімальну. Оскільки кількість унікальних пар

точок у множині з n елементів дорівнює $n(n-1)/2$, часова складність такого підходу становить $O(n^2)$. Для великих наборів даних такий підхід є неефективним.

Ефективніший алгоритм зі складністю O(nlogn) базується на парадигмі «розділяй та володарюй». Основні кроки алгоритму такі:

1. Попередня підготовка. Створюємо два масиви, що містять усі точки: Px, відсортований за x-координатою, та Py, відсортований за y-координатою. Це сортування виконується один раз на початку і займає O(nlogn) часу.

2. Розділення. Множина точок Px ділиться на дві приблизно рівні підмножини PL (ліва) та PR (права) вертикальною лінією L, що проходить через медіанну x-координату.

3. Володарювання. Алгоритм рекурсивно викликається для лівої та правої підмножин. Результатом цих викликів є мінімальні відстані δL (у PL) та δR (у PR).

4. Попереднє комбінування: Знаходимо мінімальну з цих двох відстаней: $\delta = \min(\delta L, \delta R)$. Це поточний найкращий результат.

Найскладніша частина алгоритму − крок комбінування. Можливо, що найближча пара точок складається з однієї точки з PL та однієї з PR (так звана *split pair*). Наївна перевірка всіх таких пар повернула б нас до складності $O(n^2)$.

Однак тут можна застосувати ключову оптимізацію. Якщо існує пара (pL, pR), де $pL \in PL$ та $pR \in PR$, з відстанню меншою за δ, то обидві ці точки повинні міститись у вузькій вертикальній смузі шириною 2δ з центром на розділювальній лінії L. Точки, що містяться поза цією смугою, гарантовано не можуть утворити пару з відстанню меншою за δ.

Отже, на етапі комбінування ми виконуємо такі дії:

1. Створюємо масив strip (смуга), до якого долучаємо всі точки з Py, що потрапляють у смугу шириною 2δ навколо лінії L. Важливо, що точки в цьому масиві вже відсортовані за y-координатою, оскільки ми беремо їх з попередньо відсортованого масиву Py.

2. Проходимо по масиву strip і для кожної точки p перевіряємо відстань до її сусідів у цьому самому масиві.

Виявляється, що для кожної точки p з масиву strip не потрібно перевіряти всіх наступних точок. Достатньо перевірити лише невелику константну кількість сусідів.

Геометричне доведення. Розглянемо будь-яку точку p з масиву strip. Будь-яка інша точка q, що може утворити з p пару з відстанню меншою за δ, повинна міститись у прямокутнику розміром 2δ × δ, розташованому «попереду» точки p (у напрямку зростання y-координати).

Цей прямокутник 2δ × δ складається з двох квадратів δ × δ − один у лівій половині смуги, інший у правій. Згадаймо, що на етапі рекурсивних викликів ми встановили, що мінімальна відстань між будь-якими двома точками в лівій половині (і в правій так само) не менша за δ. Геометрично це означає, що в квадрат δ×δ неможливо «вписати» більше 4 точок так, щоб відстань між будь-якими двома з них була не меншою за δ. Так, у лівій частині нашого прямокутника може бути не більше 4 точок, і в правій − не більше 4. Враховуючи саму точку p, нам потрібно перевірити відстань до щонайбільше 4 + 4 – 1 = 7 наступних точок у масиві strip.

Оскільки внутрішній цикл для кожної точки виконується не більше 7 разів, загальний час обробки смуги стає лінійним, тобто O(n).

Аналіз центральної смуги. Фрагмент коду:

```
graph TD
  subgraph "Площина точок"
    direction LR
    A[ ] -- Вертикальна лінія L --- B[ ]
    subgraph "Ліва підмножина PL"
      P1(p_L1)
      P2(p_L2)
    end
    subgraph "Центральна смуга (ширина 2δ)"
      style A fill:#eee,stroke:#333,stroke-width:2px,stroke-dasharray: 5 5
      style B fill:#eee,stroke:#333,stroke-width:2px,stroke-dasharray: 5 5
      P_strip_L(p)
      subgraph "Область перевірки для p (прямокутник 2δ x δ)"
```

```
            direction TB
            style P_strip_R1 fill:#f9f,stroke:#f0f
            P_strip_R1(q1)
            P_strip_R2(q2)
          end
        end
        subgraph "Права підмножина PR"
          P3(p_R1)
          P4(p_R2)
        end
      end
      note ["Для точки p потрібно перевірити лише точки в затіненій області. Завдяки відстані δ, таких точок не більше 7."]
```

Рекурентне співвідношення для часової складності алгоритму має вигляд:

$$T(n) = 2T(n/2) + O(n),$$

де: $2T(n/2)$ − два рекурсивні виклики для половин множини.
$O(n)$ − вартість кроку комбінування, яка охоплює побудову масиву strip та лінійний прохід по ньому.

Застосовуючи Майстер-теорему ($a = 2$, $b = 2$, $f(n) = O(n)$), ми отримуємо $\log_b a = \log_2 2 = 1$. Оскільки $f(n) = O(n) = \Theta(n^{\log_2 2})$, це Випадок 2 з $k = 0$. Отже, загальна часова складність алгоритму становить:

$$T(n) = \Theta(n\log n).$$

Цей алгоритм є чудовим прикладом того, як інформація, отримана на етапі «володарювання» (значення δ), може бути використана для кардинального скорочення простору пошуку на етапі «комбінування». Це не просто злиття, а «інформоване злиття», де ефективність народжується на перетині алгоритмічної парадигми та геометричних властивостей задачі.

*Псевдокод*

```
FUNCTION closestPair(Px, Py)
    n = length(Px)
    IF n <= 3 THEN
        RETURN bruteForce(Px)
    END IF

    mid = n / 2
    midPoint = Px[mid]

    // Розділення Py на Pyl та Pyr
    Pyl = [p for p in Py if p.x <= midPoint.x]
    Pyr = [p for p in Py if p.x > midPoint.x]

    // Рекурсивні виклики
    delta_L = closestPair(Px[:mid], Pyl)
    delta_R = closestPair(Px[mid:], Pyr)
    delta = min(delta_L, delta_R)

    // Побудова смуги
    strip = [p for p in Py if abs(p.x - midPoint.x) < delta]

    // Аналіз смуги
    min_in_strip = delta
    FOR i FROM 0 TO length(strip) - 1
        FOR j FROM i + 1 TO length(strip) - 1
            IF strip[j].y - strip[i].y >= delta THEN
                BREAK
            END IF
            dist = distance(strip[i], strip[j])
            IF dist < min_in_strip THEN
                min_in_strip = dist
            END IF
        END FOR
    END FOR

    RETURN min(delta, min_in_strip)
END FUNCTION
```

```
Python
import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def dist(p1, p2):
    return math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)

def brute_force(points):
    min_dist = float('inf')
    for i in range(len(points)):
        for j in range(i + 1, len(points)):
            d = dist(points[i], points[j])
            if d < min_dist:
                min_dist = d
    return min_dist

def strip_closest(strip, d):
    min_dist = d
    strip.sort(key=lambda point: point.y)
    for i in range(len(strip)):
        for j in range(i + 1, len(strip)):
            if (strip[j].y - strip[i].y) >= min_dist:
                break
            d = dist(strip[i], strip[j])
            if d < min_dist:
                min_dist = d
    return min_dist

def closest_util(px, py):
    n = len(px)
    if n <= 3:
        return brute_force(px)

    mid = n // 2
    mid_point = px[mid]

    pyl = [p for p in py if p.x <= mid_point.x]
    pyr = [p for p in py if p.x > mid_point.x]
```

```
        dl = closest_util(px[:mid], pyl)
        dr = closest_util(px[mid:], pyr)
        d = min(dl, dr)

        strip = [p for p in py if abs(p.x - mid_point.x) < d]

        return min(d, strip_closest(strip, d))

    def closest_pair(points):
        px = sorted(points, key=lambda point: point.x)
        py = sorted(points, key=lambda point: point.y)
        return closest_util(px, py)

    # Приклад
    points = [Point(2, 3), Point(12, 30), Point(40, 50), Point(5, 1), Point(12, 10),
Point(3, 4)]
    print(f"Найменша відстань: {closest_pair(points)}")
```

```
    C++
    #include <iostream>
    #include <vector>
    #include <cmath>
    #include <algorithm>
    #include <cfloat>

    struct Point {
        double x, y;
    };

    double dist(Point p1, Point p2) {
        return std::sqrt(std::pow(p1.x - p2.x, 2) + std::pow(p1.y - p2.y, 2));
    }

    double bruteForce(const std::vector<Point>& points) {
        double min_dist = DBL_MAX;
        for (size_t i = 0; i < points.size(); ++i) {
            for (size_t j = i + 1; j < points.size(); ++j) {
                min_dist = std::min(min_dist, dist(points[i], points[j]));
            }
        }
        return min_dist;
    }
```

```
        double stripClosest(std::vector<Point>& strip, double d) {
            double min_dist = d;
            std::sort(strip.begin(), strip.end(),(Point a, Point b) {
                return a.y < b.y;
            });

            for (size_t i = 0; i < strip.size(); ++i) {
                for (size_t j = i + 1; j < strip.size() && (strip[j].y - strip[i].y) <
min_dist; ++j) {
                    min_dist = std::min(min_dist, dist(strip[i], strip[j]));
                }
            }
            return min_dist;
        }

        double closestUtil(std::vector<Point>& px, std::vector<Point>& py) {
            size_t n = px.size();
            if (n <= 3) {
                return bruteForce(px);
            }

            size_t mid = n / 2;
            Point midPoint = px[mid];

            std::vector<Point> pyl, pyr;
            for (const auto& p : py) {
                if (p.x <= midPoint.x) {
                    pyl.push_back(p);
                } else {
                    pyr.push_back(p);
                }
            }

            std::vector<Point> pxl_part(px.begin(), px.begin() + mid);
            std::vector<Point> pxr_part(px.begin() + mid, px.end());

            double dl = closestUtil(pxl_part, pyl);
            double dr = closestUtil(pxr_part, pyr);
            double d = std::min(dl, dr);
```

```
        std::vector<Point> strip;
        for (const auto& p : py) {
            if (std::abs(p.x - midPoint.x) < d) {
                strip.push_back(p);
            }
        }

        return std::min(d, stripClosest(strip, d));
    }

    double closestPair(std::vector<Point>& points) {
        std::vector<Point> px = points;
        std::vector<Point> py = points;
        std::sort(px.begin(), px.end(),(Point a, Point b) {
            return a.x < b.x;
        });
        std::sort(py.begin(), py.end(),(Point a, Point b) {
            return a.y < b.y;
        });

        return closestUtil(px, py);
    }

    int main() {
        std::vector<Point> points = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10},
{3, 4}};
        std::cout << "Найменша відстань: " << closestPair(points) <<
std::endl;
        return 0;
    }
```

## 8.5. Розв’язання задач із застосуванням стратегії «розділяй та володарюй»

Ці приклади є «виродженими» випадками стратегії, де на кожному кроці задача зводиться лише до однієї підзадачі ($a = 1$). Вони ідеально ілюструють силу етапу «Розділення» та експоненційного зменшення розміру задачі, що призводить до логарифмічної складності, водночас мінімізуючи складність етапу «Комбінування».

Постановка задачі. Для відсортованого масиву A та цільового значення x знайти індекс i такий, що A[i] = x. Якщо такого елемента немає, повернути відповідний індикатор (наприклад, -1).

Зв’язок з «розділяй та володарюй»:

1. Розділення. Задача пошуку в масиві розміром n зводиться до порівняння цільового значення x з елементом у середині масиву A[mid]. Це порівняння миттєво ділить простір пошуку, відкидаючи половину масиву, де елемента гарантовано немає.

2. Володарювання. Рекурсивно (або ітеративно) застосовуємо пошук до однієї з половин − лівої, якщо x < A[mid], або правої, якщо x > A[mid]. Так, ми переходимо до однієї підзадачі розміром приблизно n/2.

3. Комбінування. Цей етап є тривіальним. Результат, повернутий з рекурсивного виклику (індекс знайденого елемента або індикатор невдачі), є кінцевим результатом для вихідної задачі. Жодного додаткового об’єднання не потрібно.

Аналіз складності. Рекурентне співвідношення для бінарного пошуку має вигляд:

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

Тут T(n/2) − один рекурсивний виклик для половини масиву, а O(1) − час на обчислення індексу середини та порівняння. За Майстер-теоремою ($a = 1$, $b = 2$, $f(n) = O(1) = O(n^0)$), маємо $\log_b a = \log_2 1 = 0$. Оскільки $f(n) = \Theta(n^0)$, це Випадок 2 з $k = 0$. Отже, складність становить $T(n) = \Theta(n^0 \log^{0+1} n) = \Theta(\log n)$.

```
Псевдокод (ітеративний)
FUNCTION binarySearch(arr, target)
    low = 0
    high = length(arr) - 1
    WHILE low <= high
        mid = low + floor((high - low) / 2)
        IF arr[mid] == target THEN
            RETURN mid
        ELSE IF arr[mid] < target THEN
            low = mid + 1
        ELSE
            high = mid - 1
        END IF
    END WHILE
    RETURN -1 // Елемент не знайдено
END FUNCTION
```

```
Python
def binary_search_recursive(arr, low, high, target):
    if high >= low:
        mid = low + (high - low) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] > target:
            return binary_search_recursive(arr, low, mid - 1, target)
        else:
            return binary_search_recursive(arr, mid + 1, high, target)
    else:
        return -1 # Елемент не знайдено

# Приклад
data =
target = 7
result = binary_search_recursive(data, 0, len(data)-1, target)
print(f"Елемент знайдено за індексом: {result}")
```

```
C++
#include <iostream>
#include <vector>

int binarySearchIterative(const std::vector<int>& arr, int target) {
    int low = 0;
    int high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Елемент не знайдено
}

int main() {
    std::vector<int> data = {2, 3, 5, 7, 11, 13, 17};
    int target = 7;
    int result = binarySearchIterative(data, target);
    if (result!= -1) {
        std::cout << "Елемент знайдено за індексом: " << result << std::endl;
    } else {
        std::cout << "Елемент не знайдено." << std::endl;
    }
    return 0;
}
```

Задача 2: швидке піднесення до степеня (бінарне піднесення). Обчислити $x^n$, де x − число, а n − ціле невід'ємне число, за меншу кількість операцій множення, ніж n − 1 (що вимагає наївний ітеративний підхід).

Зв’язок з «розділяй та володарюй»:

Алгоритм базується на властивостях степенів:

− якщо n парне: $x^n = (x^2)^{n/2}$;

− якщо n непарне: $x^n = x \cdot x^{n-1} = x \cdot (x^2)^{(n-1)/2}$.

1. Розділення. Задача обчислення $x^n$ зводиться до задачі обчислення степеня з показником приблизно n/2.

2. Володарювання. Рекурсивно обчислюємо $x^{n/2}$ (або $x^{(n-1)/2}$).

3. Комбінування. Результат підзадачі підноситься до квадрата. Якщо вихідний показник n був непарним, результат додатково множиться на x.

Рекурентне співвідношення ідентичне бінарному пошуку:

$$T(n) = T(n/2) + O(1),$$

де: O(1) − це одна або дві операції множення на етапі комбінування. Розв’язком цього співвідношення є $T(n) = \Theta(\log n)$.

*Псевдокод (рекурсивний)*

```
FUNCTION fast_power(base, power)
   IF power == 0 THEN
      RETURN 1
   END IF

   temp = fast_power(base, floor(power / 2))
   result = temp * temp

   IF power IS ODD THEN
      result = result * base
   END IF

   RETURN result
```

*Python*

```
def fast_power_iterative(base, power):
   result = 1
   while power > 0:
      # Якщо степінь непарний, множимо результат на базу
      if power % 2 == 1:
         result *= base
      # Підносимо базу до квадрата і ділимо степінь на 2
```

```
            base *= base
            power //= 2
        return result

    # Приклад
    print(f"2^10 = {fast_power_iterative(2, 10)}")
```

```
    C++
    #include<iostream>

    long long power(long long base, long long exp) {
        if (exp == 0)
            return 1;
        long long res = power(base, exp / 2);
        res = res * res;
        if (exp % 2 == 1)
            res = res * base;
        return res;
    }

    int main() {
        long long base = 3, exp = 10;
        std::cout << base << "^" << exp << " = " << power(base, exp) <<
std::endl;
        return 0;
    }
```

## Контрольні питання

1. Опишіть три основні етапи парадигми «розділяй та володарюй».

2. Сформулюйте загальне рекурентне співвідношення для алгоритмів цього типу та поясніть значення кожного його компонента.

3. У чому полягає математичний трюк, що лежить в основі алгоритму Карацуби?

4. Чому наївне блокове множення матриць не покращує асимптотичну складність $O(n^3)$?

5. Поясніть, чому в алгоритмі пошуку найближчої пари точок аналізується смуга шириною саме $2\delta$.

6. Доведіть, що для кожної точки в центральній смузі достатньо перевірити константну кількість сусідів.

7. Застосуйте Майстер-теорему для аналізу складності сортування злиттям ($T(n) = 2T(n/2) + O(n)$).

8. Порівняйте часову та просторову складність алгоритму Штрассена та стандартного множення матриць.

9. Що таке «базовий випадок» у рекурсивному алгоритмі і чому він необхідний?

10. Як бінарний пошук ілюструє парадигму «розділяй та володарюй»? Який з трьох етапів у ньому є тривіальним?

11. Поясніть логіку роботи алгоритму швидкого піднесення до степеня для парного та непарного показника степеня.

12. Що таке «точка перетину» (crossover point) і чому вона важлива під час практичної реалізації алгоритмів типу Штрассена?

13. Які обмеження має Майстер-теорема? Наведіть приклад рекурентного співвідношення, яке не можна розв’язати з її допомогою.

14. Які переваги та недоліки має парадигма «розділяй та володарюй» порівняно з динамічним програмуванням?

15. Як обробляються вхідні дані (числа, матриці), розмір яких не є степенем двійки, в алгоритмах Карацуби та Штрассена?

## Задачі для самостійного опрацювання

1. Реалізувати рекурсивну функцію для знаходження максимального елемента в масиві за принципом D&C. ($T(n) = 2T(n/2) + O(1)$).

2. Реалізувати сортування злиттям (Merge Sort).

3. Реалізувати швидке сортування (Quicksort).

4. Написати функцію, що перевіряє, чи є число степенем двійки, використовуючи рекурсію.

5. Модифікувати бінарний пошук для знаходження першого (найлівішого) входження елемента в масив з дублікатами.

6. Задача про більшість (Majority Element). Дано масив розміром n. Знайти елемент, що зустрічається в ньому більше ніж ⌊n/2⌋ разів. Розробити алгоритм зі складністю O(nlogn).

7. Задача про інверсії. Дано масив A. Порахувати кількість пар індексів (i, j) таких, що i < j та A[i] > A[j]. Розробити алгоритм зі складністю O(nlogn) (за допомогою модифікації сортування злиттям).

8. Задача про максимальний підмасив (Maximum Subarray Sum). Дано масив чисел (можуть бути від'ємні). Знайти неперервний підмасив з максимальною сумою елементів. Розробити алгоритм зі складністю O(nlogn).

9. Дано відсортований масив, який циклічно зсунули на невідому кількість позицій (наприклад, ``). Знайти мінімальний елемент у цьому масиві за час O(logn).

10. Реалізувати множення двох поліномів, представлених масивами коефіцієнтів, за допомогою ідеї алгоритму Карацуби.

11. Побудова опуклої оболонки (Convex Hull). Реалізувати алгоритм побудови опуклої оболонки для множини точок на площині за O(nlogn) методом «розділяй та володарюй».

12. Пошук k-тої статистики. Реалізувати алгоритм Quickselect для пошуку k-го за величиною елемента в масиві за середній час O(n).

13. Алгоритм SKYLINE. Дано набір прямокутників, що стоять на одній осі x і мають різну висоту та ширину. Побудувати їх «силует» (skyline), тобто зовнішній контур. Розробити алгоритм на основі D&C.

14. Довести, що будь-який алгоритм сортування, заснований на порівняннях, має в найгіршому випадку часову складність не менше Ω(nlogn).

15. Задача про медіану двох відсортованих масивів. Дано два відсортовані масиви A та B розмірів m та n відповідно. Знайти медіану об'єднаного масиву за час O(log(m + n)).

# ТЕМА 9
# ДИНАМІЧНЕ ПРОГРАМУВАННЯ ТА ЖАДІБНІ АЛГОРИТМИ

## 9.1. Динамічне програмування: принципи та приклади

*Динамічне*[26] *програмування* (ДП) − це потужна алгоритмічна парадигма[27], що застосовується для розв'язання складних оптимізаційних задач через їх розбиття на простіші підзадачі. Ключова ідея полягає в тому, щоб розв'язувати кожну підзадачу лише один раз і зберігати її результат. Це дає змогу уникнути повторних обчислень, коли та сама підзадача зустрічається знову, що є фундаментальною відмінністю від простої рекурсії. Такий підхід дає змогу трансформувати рекурсивні алгоритми з експоненційною часовою складністю в значно ефективніші алгоритми з поліноміальною складністю. Щоб проілюструвати проблему, яку вирішує ДП, розглянемо класичний приклад обчислення чисел Фібоначчі. Послідовність Фібоначчі визначається рекурентним співвідношенням:

F(n) = F(n − 1) + F(n − 2), з базовими випадками F(0) = 0 та F(1) = 1.

Наївна рекурсивна реалізація цієї функції виглядає так:

```
function fib(n)
    if n <= 1
        return n
    return fib(n-1) + fib(n-2)
```

[26] У широкому сенсі, динамічний – це щось, що перебуває в русі, розвитку або постійній зміні. Цей термін походить від грецького слова «dynamikos», що означає «сильний» або «могутній», і часто протиставляється «статичному», яке означає незмінний або нерухомий.

Інформатика та програмування. Тут «динамічний» часто означає щось, що може змінюватися під час виконання програми. Наприклад, динамічне програмування, про яке ми говорили раніше, зберігає та оновлює результати, а динамічний сайт може змінювати свій вміст на основі дій користувача, на відміну від статичного, який завжди виглядає однаково.

[27] Парадигма – це основний, вихідний підхід або модель для розв'язання певних типів задач. У контексті програмування, це фундаментальний стиль або філософія, яка визначає, як створювати та структурувати алгоритми.

Кожна парадигма пропонує свій набір концепцій і абстракцій, що допомагають програмісту мислити про проблему певним чином і організовувати код. Програміст може обирати парадигму залежно від характеру задачі, яку він вирішує.

Хоча цей код є елегантним, він вкрай неефективний. Розглянемо дерево рекурсивних викликів для fib(5).

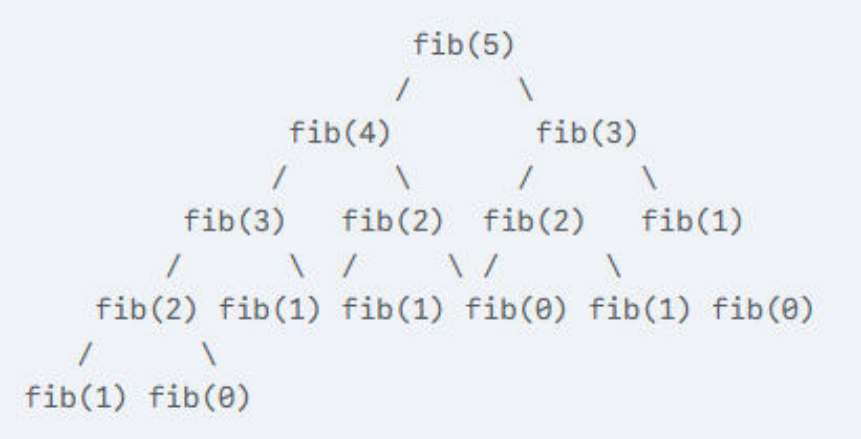



Рисунок 9.1 – Дерево рекурсивних викликів для fib(5)

Зі схеми видно, що fib(3) обчислюється 2 рази, fib(2) − 3 рази, і так далі. Кількість обчислень зростає експоненційно, що призводить до часової складності $O(2^n)$. Динамічне програмування вирішує цю проблему, зберігаючи результат fib(3) після першого обчислення і повторно використовуючи його під час наступних викликів замість того, щоб обчислювати його знову.

Застосування динамічного програмування є доцільним лише тоді, коли задача має дві ключові властивості: оптимальну підструктуру та перетин підзадач. Ці властивості є не просто характеристиками, а необхідними передумовами для ефективного використання ДП. Оптимальна підструктура. Властивість оптимальної підструктури означає, що оптимальний розв’язок задачі може бути сконструйований з оптимальних розв’язків її підзадач. Цей принцип, відомий як «принцип оптимальності», є теоретичною основою ДП. Наприклад, у задачі про найкоротший шлях між двома вершинами в графі, якщо шлях від u до v через проміжну вершину w є найкоротшим, то його частини − шлях від u до w і шлях від w до v − також мають бути найкоротшими. Формальне доведення наявності оптимальної підструктури є важливим кроком під час проєктування ДП-алгоритму. Перетин підзадач. Властивість перетину підзадач означає, що рекурсивний алгоритм розв’язує одні й ті самі підзадачі багато разів. Саме ця властивість робить ДП ефективним. Якщо підзадачі не перетинаються

(як, наприклад, в алгоритмі сортування злиттям, де масив щоразу ділиться на унікальні частини), то зберігання їхніх розв'язків не дає жодних переваг.

У цьому випадку простіший підхід «розділяй і володарюй» є більш доцільним. Дерево рекурсії для fib(5) є яскравою демонстрацією цієї властивості, де підзадачі fib(3) та fib(2) обчислюються багаторазово.

Існують дві основні стратегії реалізації динамічного програмування для зберігання та повторного використання розв'язків підзадач. *Мемоізація* (підхід «зверху-вниз») − це техніка, яка реалізується за допомогою рекурсії та використовує підхід «зверху-вниз» (top-down). Вона є модифікацією стандартної рекурсивної функції. Перед тим, як обчислювати розв'язок підзадачі, алгоритм перевіряє, чи не був він уже обчислений раніше, заглянувши у спеціальну структуру даних для зберігання результатів (наприклад, масив або хеш-таблицю), яку називають «мемо» або «кеш».

Процес виглядає так:

1. Створюється таблиця для зберігання результатів (наприклад, memo), яка ініціалізується спеціальним значенням (наприклад, -1 або NIL), що позначає необчислений стан.

2. Під час виклику функції для розв'язання підзадачі спочатку перевіряється memo.

3. Якщо результат вже є в memo, він негайно повертається.

4. В іншому випадку результат обчислюється рекурсивно, зберігається в memo, а потім повертається.

Перевага цього підходу полягає в тому, що він є більш інтуїтивним під час переходу від чисто рекурсивного визначення задачі. Крім того, мемоізація обчислює розв'язки лише для тих підзадач, які дійсно потрібні для отримання кінцевого результату. *Табуляція (підхід «знизу-вгору»)* − це ітеративна техніка, що використовує підхід «знизу-вгору» (bottom-up). Замість того, щоб починати з основної задачі і рекурсивно спускатися до базових випадків, табуляція починає з розв'язання найменших (базових) підзадач. Результати цих підзадач зберігаються в таблиці (зазвичай масиві), а потім використовуються для ітеративного обчислення розв'язків для все більших підзадач, доки не буде знайдено розв'язок вихідної задачі.

Процес виглядає так:

1. Створюється таблиця dp відповідного розміру.

2. Заповнюються початкові значення таблиці, що відповідають базовим випадкам задачі.

3. У циклах, у визначеному порядку, обчислюються значення для інших елементів таблиці, спираючись на вже обчислені значення.

Перевагою табуляції є відсутність накладних витрат на рекурсивні виклики, що робить її дещо швидшою за мемоізацію (на константний чинник). Також вона унеможливлює ризик переповнення стека викликів, що може статися за великої глибини рекурсії в мемоізованому підході. Вибір між мемоізацією та табуляцією часто залежить від структури простору підзадач: якщо він «щільний» (тобто більшість підзадач потрібно розв'язати), табуляція є ефективнішою; якщо ж він «розріджений» (багато підзадач ніколи не викликаються), мемоізація дає змогу уникнути зайвої роботи.

Таблиця 9.1 – Порівняння мемоізації та табуляції

| Характеристика (Feature) | Мемоізація (Top-Down) | Табуляція (Bottom-Up) |
|---|---|---|
| **Підхід (Approach)** | Зверху-вниз (Top-down), рекурсивний | Знизу-вгору (Bottom-up), ітеративний |
| **Обчислення (Computation)** | Обчислює розв'язки підзадач «на вимогу» | Обчислює розв'язки для всіх підзадач, починаючи з найменших |
| **Переваги (Advantages)** | Інтуїтивно зрозуміліший, якщо задача має рекурсивну природу. Розв'язує лише необхідні підзадачі | Відсутні накладні витрати на рекурсію. Немає ризику переповнення стека. Можлива оптимізація пам'яті |
| **Недоліки (Disadvantages)** | Накладні витрати на рекурсивні виклики. Можливе переповнення стека для великої глибини рекурсії | Може обчислювати розв'язки для підзадач, які не є необхідними для кінцевого результату |
| **Складність коду (Code Complexity)** | Легко модифікувати з чисто рекурсивного розв'язку | Вимагає чіткого порядку обчислення підзадач, що може бути складніше |
| **Складність (Complexity)** | Часова та просторова складність зазвичай однакові з табуляцією | Часова та просторова складність зазвичай однакові з мемоізацією |

## 9.2. Класичні задачі динамічного програмування

Задача обчислення n-го числа Фібоначчі є канонічним прикладом для демонстрації ДП.

Наївний рекурсивний розв’язок, як було показано, має складність $O(2^n)$ через масові повторні обчислення.

Мемоізований розв’язок зберігає обчислені значення в масиві. Часова складність − O(n), просторова − O(n) для масиву memo та стека рекурсії.

```
Python
# Python: Мемоізований розв'язок
def fib_memo(n, memo):
    if n <= 1:
        return n
    if memo[n] is not None:
        return memo[n]
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]

n = 10
memo = [None] * (n + 1)
print(fib_memo(n, memo))
```

Табульований (Bottom-Up) розв’язок, ітеративно заповнює масив dp від 0 до n. Часова складність − O(n), просторова − O(n).

```
C++
// C++: Табульований розв'язок
#include <iostream>
#include <vector>
int fib_tab(int n) {
    if (n <= 1) {
        return n;
    }
    std::vector<int> dp(n + 1);
    dp = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; ++i) {
```

```
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
int main() {
    int n = 10;
    std::cout << fib_tab(n) << std::endl;
    return 0;
}
```

Оптимізований за пам’яттю розв’язок. Для обчислення dp[i] потрібні лише dp[i − 1] та dp[i − 2]. Тому замість цілого масиву можна зберігати лише два попередні значення. Це зменшує просторову складність до O(1), зберігаючи часову складність O(n).

```
Python
# Python: Оптимізований за пам'яттю розв'язок
def fib_optimized(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
print(fib_optimized(10))
```

Формальне визначення**.** Дано набір з n предметів, кожен з яких має вагу wi та вартість vi. Також дано рюкзак з максимальною вантажопідйомністю W. Потрібно вибрати такий набір предметів, щоб сумарна вартість була максимальною, а сумарна вага не перевищувала W. Кожен предмет можна або взяти повністю, або не брати зовсім (звідси назва «0/1»).

Рекурентне співвідношення. Для розв’язання цієї задачі методом ДП визначимо стан dp[i][w] як максимальну вартість, яку можна отримати, розглядаючи перші і предметів за місткості рюкзака w.

Для кожного і-го предмета є два варіанти:

1. Не брати і-й предмет. У цьому випадку максимальна вартість буде такою самою, як і для перших і − 1 предметів з тією самою місткістю w, тобто dp[i −1][w].

2. Взяти i-й предмет (якщо його вага wi ≤ w). У цьому випадку вартість складатиметься з вартості самого предмета vi плюс максимальна вартість, яку можна отримати з перших i − 1 предметів за залишкової місткості w−wi, тобто vi + dp[i − 1][w − wi].

Так, рекурентне співвідношення має вигляд:

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{якщо } w_i > w \\ \max(dp[i-1][w], v_i + dp[i-1][w-w_i]) & \text{якщо } w_i \le w \end{cases}$$

Побудова ДП-таблиці. Створюється таблиця розміром (n + 1) × (W + 1), яка заповнюється ітеративно. Розглянемо приклад: W = 4, val = {1, 2, 3}, weight = {4, 5, 1}. Таблиця dp матиме розмір 4 × 5.

| i\w | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 | 0 |
| **1** (v = 1, w = 4) | 0 | 0 | 0 | 0 | 1 |
| **2** (v = 2, w = 5) | 0 | 0 | 0 | 0 | 1 |
| **3** (v = 3, w = 1) | 0 | 3 | 3 | 3 | **3** |

Рядок 1 (предмет 1): вага 4, вартість 1. Можна покласти лише в рюкзак місткістю 4. dp[1][2] = 1.

Рядок 2 (предмет 2): вага 5, вартість 2. Не можна покласти в рюкзак місткістю до 4. Значення копіюються з попереднього рядка.

Рядок 3 (предмет 3): вага 1, вартість 3.

- dp[3][1] = max(dp[4][1], 3 + dp[4]) = max(0, 3 + 0) = 3.
- dp[3][4] = max(dp[4][4], 3 + dp[4][1]) = max(0, 3 + 0) = 3.
- dp[3][3] = max(dp[4][3], 3 + dp[4][4]) = max(0, 3 + 0) = 3.
- dp[3][2] = max(dp[4][2], 3 + dp[4][3]) = max(1, 3 + 0) = 3.

Кінцевий результат dp[3][2] = 3.

Часова складність алгоритму становить O(nW), а просторова − O(nW). Оскільки складність залежить від значення W, а не лише від кількості вхідних даних n, вона називається псевдополіноміальною.

*Підпослідовність рядка* − це новий рядок, утворений з вихідного через видалення нуля або більше символів без зміни відносного порядку решти символів. Найдовша спільна підпослідовність (LCS) двох рядків − це найдовша підпослідовність, яка є

спільною для обох рядків. Наприклад, для рядків «abcde» та «ace», LCS є «ace» довжиною 3.

Рекурентне співвідношення. Нехай dp[i][j] − це довжина LCS для префіксів першого рядка X довжиною i (X[1..i]) та другого рядка Y довжиною j (Y[1..j]).

Розглядаємо символи X[i] та Y[j]:

1. Якщо X[i]==Y[j]: Символи збігаються. Вони є частиною LCS. Отже, довжина LCS збільшується на 1, і ми шукаємо LCS для решти рядків X[1..i − 1] та Y[1..j − 1].

dp[i][j] = 1 + dp[i − 1][j − 1].

2. Якщо X[i]␣ = Y[j]: Символи не збігаються. У цьому випадку LCS для X[1..i] та Y[1..j] буде найдовшою з двох підпослідовностей: LCS для X[1..i − 1] та Y[1..j] (ігноруємо X[i]) або LCS для X[1..i] та Y[1..j − 1] (ігноруємо Y[j]).

dp[i][j] = max(dp[i − 1][j],dp[i][j − 1]).

Побудова ДП-таблиці та реконструкція LCS. Створюється таблиця розміром (m + 1) × (n + 1), де m і n − довжини рядків. Вона заповнюється згідно з рекурентним співвідношенням. Після заповнення таблиці значення dp[m][n] містить довжину LCS.

Для реконструкції самої підпослідовності виконується зворотний прохід (backtracking) по таблиці, починаючи з dp[m][n]:

− якщо X[i−1]==Y[j−1], додаємо цей символ до LCS і рухаємося по діагоналі до dp[i-1][j-1];

− інакше, рухаємося в напрямку більшого значення: вгору до dp[i-1][j] або вліво до dp[i][j-1].

Часова та просторова складність становлять O(mn).

## 9.3. Жадібні алгоритми: принципи та приклади

*Жадібні алгоритми* − це парадигма, яка розв’язує задачі, роблячи на кожному кроці вибір, що видається найкращим у даний момент (локально оптимальний вибір), з надією, що послідовність таких виборів призведе до глобально оптимального розв’язку. На відміну від динамічного програмування, жадібний алгоритм ніколи не переглядає своїх попередніх рішень.

Для того, щоб жадібний підхід гарантовано знаходив оптимальний розв'язок, задача повинна мати дві властивості.

1. Це ключова властивість, яка стверджує, що глобально оптимальний розв'язок можна отримати, послідовно роблячи локально оптимальні вибори. Іншими словами, вибір, зроблений на поточному кроці, не повинен зашкодити можливості досягти оптимального розв'язку в майбутньому. Доведення цієї властивості для конкретної задачі є центральним елементом доведення коректності жадібного алгоритму. Цей вибір робиться *до* розв'язання підзадач, що відрізняє жадібні алгоритми від ДП, де вибір залежить від розв'язків підзадач.

2. Ця властивість, спільна з динамічним програмуванням, означає, що після здійснення жадібного вибору, задача, що залишилася, є меншим екземпляром вихідної задачі, і її оптимальний розв'язок у поєднанні зі зробленим вибором дає оптимальний розв'язок вихідної задачі. Хоча наявність оптимальної підструктури є необхідною, саме властивість жадібного вибору гарантує, що конкретний жадібний вибір є частиною *деякого* оптимального розв'язку.

Найкращим прикладом для порівняння є задача про рюкзак.

Задача про рюкзак 0/1. Не можна ділити предмети. Локально оптимальний вибір (взяти предмет з найбільшою вартістю або найкращим співвідношенням вартість / вага) може призвести до того, що в рюкзаку не залишиться місця для інших, менш важких, але сумарно більш цінних предметів. Тому ця задача вимагає ДП для перебору всіх можливих комбінацій. Задача про дробовий рюкзак. Предмети можна ділити. Тут жадібний підхід працює ідеально. Ми просто сортуємо предмети за спаданням співвідношення вартість / вага і беремо їх у такому порядку, заповнюючи рюкзак. Якщо черговий предмет не вміщується повністю, ми беремо його частину.

Таблиця 9.2 – Динамічне програмування і Жадібні алгоритми

| Характеристика (Feature) | Динамічне програмування (Dynamic Programming) | Жадібні алгоритми (Greedy Algorithms) |
|---|---|---|
| Гарантія оптимальності (Optimality Guarantee) | Завжди знаходить глобальний оптимум (якщо властивості дотримані) | Не завжди; знаходить оптимум лише для задач з властивістю жадібного вибору |
| Процес прийняття рішень (Decision Process) | Робить вибір на основі розв'язків підзадач (розглядає всі варіанти) | Робить локально оптимальний вибір, не переглядаючи його |
| Основні властивості (Key Properties) | Оптимальна підструктура та перетин підзадач | Властивість жадібного вибору та оптимальна підструктура |
| Підхід (Approach) | Зазвичай знизу-вгору (табуляція) або зверху-вниз з кешуванням (мемоізація) | Зазвичай зверху-вниз, послідовно роблячи вибори |
| Часова складність (Time Complexity) | Зазвичай вища (напр., $O(n^2)$, $O(nW)$) | Зазвичай нижча (напр., O(nlogn) через сортування) |
| Приклад (Example) | Задача про рюкзак 0/1, Редакційна відстань | Задача про дробовий рюкзак, Коди Хаффмана, Алгоритм Дейкстри |

## 9.4. Коди Хаффмана

*Кодування Хаффмана* − це класичний жадібний алгоритм, який використовується для стиснення даних без втрат. Його ідея полягає в тому, щоб призначити символам, які зустрічаються частіше, коротші двійкові коди, а символам, що зустрічаються рідше, − довші. Це створює оптимальний *префіксний код*, де жоден код не є префіксом іншого, що забезпечує однозначне декодування.

Алгоритм:

1. Обчислити частоту входження кожного символу у вхідних даних.

2. Для кожного символу створити листковий вузол дерева, який містить сам символ та його частоту.

3. Побудова дерева. Використовувати мін-пріоритетну чергу (min-heap), куди поміщаються всі листкові вузли. Пріоритет вузла визначається його частотою.

4. Ітеративний процес. Поки в черзі більше одного вузла:

− витягти два вузли з найменшими частотами (найвищим пріоритетом);

− створити новий внутрішній вузол, у якого ці два вузли є лівим та правим нащадками;

− частота нового вузла дорівнює сумі частот його нащадків;

− помістити новий вузол назад у пріоритетну чергу.

5. Завершення. Коли в черзі залишається один вузол, він стає коренем дерева Хаффмана.

6. Генерація кодів. Для отримання двійкового коду кожного символу потрібно пройти шлях від кореня до відповідного листкового вузла. Рух вліво кодується як '0', а вправо − як '1'.

Цей алгоритм є жадібним, оскільки на кожному кроці він об’єднує два найменш імовірні символи, що є локально оптимальним рішенням, яке в кінцевому підсумку призводить до глобально оптимального кодування.

## 9.5. Реалізація алгоритмів динамічного програмування

Нижче наведено повну реалізацію задачі про рюкзак 0/1 з використанням підходу табуляції (знизу-вгору).

```
Псевдокод (Задача про рюкзак 0/1)
function Knapsack(W, wt, val, n)
   // Створення ДП-таблиці dp[n+1]
   let dp be a new array of size (n + 1) x (W + 1)

   // Заповнення таблиці
   for i from 0 to n do
      for w from 0 to W do
         // Базові випадки: немає предметів або місткість 0
         if i == 0 or w == 0 then
            dp[i][w] = 0
          // Якщо вага і-го предмета менша або дорівнює поточній місткості
         else if wt[i-1] <= w then
            // Вибираємо максимум з двох варіантів:
```

```
            // 1. Не брати i-й предмет
            // 2. Взяти i-й предмет
            dp[i][w] = max(val[i-1] + dp[i-1][w - wt[i-1]], dp[i-1][w])
        // Якщо вага i-го предмета більша за поточну місткість
        else
            // Не можемо взяти i-й предмет
            dp[i][w] = dp[i-1][w]
        end if
    end for
end for

// Результат знаходиться в останній комірці
return dp[n]
```

```
Python
def knapsack_01(W, wt, val):
    """
    Розв'язує задачу про рюкзак 0/1 за допомогою ДП (табуляція).

    :param W: Максимальна місткість рюкзака.
    :param wt: Список ваг предметів.
    :param val: Список вартостей предметів.
    :return: Максимальна вартість, яку можна помістити в рюкзак.
    """
    n = len(val)
    # Створення ДП-таблиці dp[n+1] та ініціалізація нулями.
    # dp[i][w] зберігатиме максимальну вартість для перших i предметів
    # при місткості рюкзака w.
    dp = for _ in range(n + 1)]

    # Побудова таблиці dp знизу-вгору
    for i in range(1, n + 1):
        for w in range(1, W + 1):
            # Вага та вартість поточного (i-1)-го предмета
            # (індексація в масивах з 0)
            current_weight = wt[i - 1]
            current_val = val[i - 1]

            # Якщо вага поточного предмета більша за місткість w,
            # ми не можемо його взяти.
            if current_weight > w:
```

```
                dp[i][w] = dp[i - 1][w]
            else:
                # Розглядаємо два варіанти:
                # 1. Не брати поточний предмет: dp[i - 1][w]
                # 2. Взяти поточний предмет:
                #    його вартість + макс. вартість для решти місця
                include_item = current_val + dp[i - 1][w - current_weight]
                exclude_item = dp[i - 1][w]

                dp[i][w] = max(include_item, exclude_item)

        return dp[n]

    # Приклад використання
    val =
    wt = [5, 6, 7]
    W = 50
    print(f"Максимальна вартість: {knapsack_01(W, wt, val)}")
```

```
    C++
    #include <iostream>
    #include <vector>
    #include <algorithm>

    /**
     * @brief Розв'язує задачу про рюкзак 0/1 за допомогою ДП (табуляція).
     *
     * @param W Максимальна місткість рюкзака.
     * @param wt Вектор ваг предметів.
     * @param val Вектор вартостей предметів.
     * @return int Максимальна вартість, яку можна помістити в рюкзак.
     */
    int knapsack_01(int W, const std::vector<int>& wt, const
std::vector<int>& val) {
        int n = val.size();
        // Створення ДП-таблиці dp[n+1].
        // dp[i][w] зберігатиме максимальну вартість для перших i
предметів
        // при місткості рюкзака w.
        std::vector<std::vector<int>> dp(n + 1, std::vector<int>(W + 1, 0));
```

```
        // Побудова таблиці dp знизу-вгору
        for (int i = 1; i <= n; ++i) {
            for (int w = 1; w <= W; ++w) {
                // Вага та вартість поточного (i-1)-го предмета
                // (індексація в векторах з 0)
                int current_weight = wt[i - 1];
                int current_val = val[i - 1];

                // Якщо вага поточного предмета більша за місткість w,
                // ми не можемо його взяти.
                if (current_weight > w) {
                    dp[i][w] = dp[i - 1][w];
                } else {
                    // Розглядаємо два варіанти:
                    // 1. Не брати поточний предмет: dp[i - 1][w]
                    // 2. Взяти поточний предмет:
                    //    його вартість + макс. вартість для решти місця
                    int include_item = current_val + dp[i - 1][w - current_weight];
                    int exclude_item = dp[i - 1][w];

                    dp[i][w] = std::max(include_item, exclude_item);
                }
            }
        }

        return dp[n];
    }

    int main() {
        std::vector<int> val = {60, 100, 120};
        std::vector<int> wt = {10, 20, 30};
        int W = 50;
        std::cout << "Максимальна вартість: " << knapsack_01(W, wt, val)
<< std::endl;
        return 0;
    }
```

## 9.6. Реалізація жадібних алгоритмів

Нижче наведено повну реалізацію алгоритму кодування Хаффмана з використанням пріоритетної черги.

```
Псевдокод (Коди Хаффмана)
function HuffmanCoding(characters, frequencies)
    // Створення вузла дерева
    class Node:
        char, freq, left, right

    // Створення пріоритетної черги (мін-хіп)
    let pq be a new min-priority-queue

    // Додавання всіх символів як листкових вузлів до черги
    for each character c with frequency f in (characters, frequencies) do
        node = new Node(c, f, null, null)
        pq.insert(node)
    end for

    // Побудова дерева
    while pq.size() > 1 do
        // Витягуємо два вузли з найменшими частотами
        left = pq.extract_min()
        right = pq.extract_min()

        // Створюємо новий внутрішній вузол
        sum_freq = left.freq + right.freq
        parent = new Node('$', sum_freq, left, right)

        // Додаємо новий вузол назад у чергу
        pq.insert(parent)
    end while

    // Корінь дерева Хаффмана
    root = pq.extract_min()

    // Рекурсивна функція для обходу дерева та генерації кодів
    function generateCodes(node, current_code, codes_map)
        if node is null then
            return
        end if
```

```
        // Якщо вузол є листком, зберігаємо його код
        if node.char!= '$' then
            codes_map[node.char] = current_code
        end if

        // Рекурсивний виклик для лівого та правого нащадків
        generateCodes(node.left, current_code + "0", codes_map)
        generateCodes(node.right, current_code + "1", codes_map)
    end function

    // Запуск генерації кодів
    let codes be a new map
    generateCodes(root, "", codes)

    return codes
end function
```

```
Python
import heapq
from collections import defaultdict

class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    # Метод для порівняння вузлів у пріоритетній черзі
    def __lt__(self, other):
        return self.freq < other.freq

def huffman_coding(text):
    """
    Генерує коди Хаффмана для заданого тексту.

    :param text: Вхідний рядок.
    :return: Словник з кодами Хаффмана для кожного символу.
    """
    # 1. Підрахунок частот символів
```

```
    frequency = defaultdict(int)
    for char in text:
        frequency[char] += 1

    # 2. Створення пріоритетної черги з листковими вузлами
    priority_queue = [HuffmanNode(char, freq) for char, freq in frequency.items()]
    heapq.heapify(priority_queue)

    # 3. Побудова дерева Хаффмана
    while len(priority_queue) > 1:
        # Витягуємо два вузли з найменшою частотою
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)

        # Створюємо новий внутрішній вузол
        merged_freq = left.freq + right.freq
        merged_node = HuffmanNode(None, merged_freq)
        merged_node.left = left
        merged_node.right = right

        # Додаємо новий вузол назад у чергу
        heapq.heappush(priority_queue, merged_node)

    # Корінь дерева - єдиний елемент, що залишився в черзі
    root = priority_queue

    # 4. Генерація кодів шляхом обходу дерева
    huffman_codes = {}
    def generate_codes_recursive(node, current_code):
        if node is None:
            return

        # Якщо це листковий вузол, зберігаємо код
        if node.char is not None:
            huffman_codes[node.char] = current_code
            return

        generate_codes_recursive(node.left, current_code + "0")
        generate_codes_recursive(node.right, current_code + "1")
```

```
    generate_codes_recursive(root, "")
    return huffman_codes

# Приклад використання
text = "geeksforgeeks"
codes = huffman_coding(text)
for char, code in sorted(codes.items()):
    print(f"'{char}': {code}")
```

```
C++
#include <iostream>
#include <string>
#include <queue>
#include <vector>
#include <map>

// Вузол дерева Хаффмана
struct HuffmanNode {
    char data;
    unsigned freq;
    HuffmanNode *left, *right;

    HuffmanNode(char data, unsigned freq) : data(data), freq(freq), left(nullptr), right(nullptr) {}
};

// Компаратор для пріоритетної черги (мін-хіп)
struct CompareNodes {
    bool operator()(HuffmanNode* l, HuffmanNode* r) {
        return l->freq > r->freq;
    }
};

// Функція для рекурсивного обходу дерева та збереження кодів
void generateCodes(HuffmanNode* root, std::string code, std::map<char, std::string>&huffmanCodes) {
    if (!root) {
        return;
    }
    // Якщо це листковий вузол
    if (!root->left &&!root->right) {
        huffmanCodes[root->data] = code;
```

```
        }
        generateCodes(root->left, code + "0", huffmanCodes);
        generateCodes(root->right, code + "1", huffmanCodes);
    }

    // Основна функція для побудови дерева та генерації кодів
    std::map<char, std::string> huffmanCoding(const std::string& text) {
        // 1. Підрахунок частот
        std::map<char, unsigned> freqMap;
        for (char c : text) {
            freqMap[c]++;
        }

        // 2. Створення пріоритетної черги
        std::priority_queue<HuffmanNode*,  std::vector<HuffmanNode*>,
CompareNodes> pq;
        for (auto pair : freqMap) {
            pq.push(new HuffmanNode(pair.first, pair.second));
        }

        // 3. Побудова дерева
        while (pq.size() > 1) {
            HuffmanNode* left = pq.top();
            pq.pop();
            HuffmanNode* right = pq.top();
            pq.pop();

            HuffmanNode* parent = new HuffmanNode('$', left->freq + right-
>freq);
            parent->left = left;
            parent->right = right;
            pq.push(parent);
        }

        // 4. Генерація кодів
        HuffmanNode* root = pq.top();
        std::map<char, std::string> huffmanCodes;
        generateCodes(root, "", huffmanCodes);

        // Очищення пам'яті (потрібно реалізувати функцію для видалення
дерева)
        //...
```

```
    return huffmanCodes;
}

int main() {
    std::string text = "geeksforgeeks";
    auto codes = huffmanCoding(text);

    for (auto pair : codes) {
        std::cout << "'" << pair.first << "': " << pair.second << std::endl;
    }

    return 0;
}
```

## 9.7. Редакційна відстань (відстань Левенштейна)

*Редакційна відстань, або відстань Левенштейна*, між двома рядками − це мінімальна кількість односимвольних операцій (вставка, видалення, заміна), необхідних для перетворення одного рядка в інший. Цей показник широко використовується для вимірювання схожості рядків, наприклад, у системах перевірки орфографії.

Рекурентне співвідношення ДП. Нехай dp[i][j] − це редакційна відстань між першими i символами рядка s1 та першими j символами рядка s2.

1. Базові випадки:

− dp[j] = j: перетворення порожнього рядка в префікс s2 довжиною j вимагає j вставок.

− dp[i] = i: перетворення префікса s1 довжиною i в порожній рядок вимагає i видалень.

2. Рекурсивний крок:

− Якщо s1[i − 1] == s2[j − 1]: останні символи збігаються. Операції не потрібні, тому відстань така сама, як і для префіксів без цих символів.

$$dp[i][j] = dp[i - 1][j - 1].$$

− Якщо s1[i − 1]!= s2[j − 1]: останні символи не збігаються. Потрібна одна операція. Ми вибираємо мінімум з трьох можливих операцій:

Вставка. Вставити s2[j − 1] в s1. Відстань буде 1 + dp[i][j − 1].

Видалення. Видалити s1[i − 1] з s1. Відстань буде 1 + dp[i − 1][j].

Заміна. Замінити s1[i − 1] на s2[j − 1]. Відстань буде 1 + dp[i − 1][j − 1].

Отже, dp[i][j] = 1 + min(dp[i − 1][j], dp[i][j − 1],dp[i − 1][j − 1]).

Приклад: «KITTEN» -> «SITTING»

ДП-таблиця розміром 7 × 8:

| | | S | I | T | T | I | N | G |
|---|---|---|---|---|---|---|---|---|
| | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **K** | 1 | **1** | 2 | 3 | 4 | 5 | 6 | 7 |
| **I** | 2 | 2 | **1** | 2 | 3 | 4 | 5 | 6 |
| **T** | 3 | 3 | 2 | **1** | **2** | 3 | 4 | 5 |
| **T** | 4 | 4 | 3 | 2 | **1** | 2 | 3 | 4 |
| **E** | 5 | 5 | 4 | 3 | 2 | **2** | 3 | 4 |
| **N** | 6 | 6 | 5 | 4 | 3 | 3 | **2** | **3** |

Кінцевий результат dp[8][9] = 3.

Часова та просторова складність становлять O(mn), де m і n − довжини рядків.

## 9.8. Оптимальне множення матриць

Дано послідовність матриць $A_1$, $A_2$, …,$A_n$, де матриця $A_i$ має розмірність $p_{i-1} \times p_i$. Задача полягає у знаходженні такого порядку їх перемноження (розстановки дужок), який мінімізує загальну кількість скалярних множень.[46] Множення матриць є асоціативним, тобто (AB)C = A(BC), але кількість операцій може суттєво відрізнятися. Наприклад, для матриць $A_{10\times30}$, $B_{30\times5}$, $C_{5\times60}$, вартість (AB)C становить 4 500 операцій, а A(BC) – 27 000 операцій.

Рекурентне співвідношення ДП. Кількість способів розставити дужки є експоненційною (визначається числами Каталана), тому повний перебір неможливий. Нехай m[i, j] − мінімальна

кількість скалярних множень, необхідна для обчислення добутку Ai…Aj.

Базовий випадок:m[i, i] = 0 (множення однієї матриці не вимагає операцій).

Рекурсивний крок: щоб обчислити m[i, j], ми можемо розбити ланцюжок матриць у точці k ($i \le k < j$), обчисливши $A_i…A_k$ та $A_{k+1}…A_j$ окремо, а потім перемноживши результати. Розмір результуючої матриці для $A_i…A_k$ буде $p_{i-1} \times p_k$, а для $A_{k+1}…A_j$ – $p_k \times p_j$. Вартість їх перемноження складе $p_{i-1} \cdot p_k \cdot p_j$.

Отже, ми повинні перебрати всі можливі точки розбиття k і вибрати найкращу:

$$m[i, j] = \min i \le k < j\{m[i,k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j\}.$$

Алгоритм:

Табульований підхід заповнює ДП-таблицю, розглядаючи ланцюжки матриць зростаючої довжини len від 2 до n. Для кожної довжини len перебираються всі можливі початкові індекси i.

Складність: алгоритм має три вкладені цикли (по len, i та k), тому часова складність становить $O(n^3)$. Просторова складність для зберігання таблиці − $O(n^2)$.

## 9.9. Задача про мінімальне покриття множини

Дано універсальну множину елементів U та колекцію S її підмножин $S_1$, $S_2$…, $S_m$. Покриттям множини називається така підколекція $C \subseteq S$, об’єднання якої дорівнює U. Задача полягає у знаходженні покриття з мінімальною кількістю підмножин.

Ця задача є класичним прикладом NP-складної задачі, що означає, що не існує відомого алгоритму, який би знаходив точний оптимальний розв’язок за поліноміальний час. Тому для її розв’язання часто використовують апроксимаційні алгоритми, які не гарантують оптимального розв’язку, але знаходять «достатньо добрий» розв’язок за прийнятний час.

Жадібний апроксимаційний алгоритм. Найбільш поширений підхід до цієї задачі − жадібний алгоритм, який працює так:

1. Ініціалізувати множину непокритих елементів Uncovered як U.
2. Ініціалізувати порожню множину для результату Cover.
3. Поки Uncovered не порожня:

− жадібний вибір: знайти таку підмножину si∈s, яка містить найбільшу кількість елементів з uncovered;

− додати si до cover;

− видалити всі елементи si з uncovered.

Приклад:U = {1, 2, 3, 4, 5}, S = {{1, 2, 3},{2, 4},{3, 4},{4, 5}}.

1. Uncovered = {1, 2, 3, 4, 5}. Найбільше непокритих елементів містить {1,2,3}. Обираємо її.

Cover = \{\{1, 2, 3\}\}, Uncovered = {4, 5}.

2. Uncovered = {4, 5}. Найбільше непокритих елементів містять {2, 4}, {3, 4}, {4, 5} (по одному). Обираємо {4, 5}.

Cover = \{\{1, 2, 3\}, \{4, 5\}\}, Uncovered = \emptyset.

Алгоритм завершено. Розв’язок: {{1, 2, 3},{4, 5}}.

Коефіцієнт апроксимації. Цей жадібний алгоритм не завжди знаходить оптимальний розв’язок. Однак доведено, що розмір знайденого ним покриття не більше ніж у H(n) ≈ ln(n) разів перевищує розмір оптимального покриття, де n = |U| − кількість елементів в універсальній множині.

H(n) − це n-те гармонійне число. Це означає, що жадібний алгоритм є логарифмічним апроксимаційним алгоритмом.

## 9.10. Приклади розв’язку задач

**Задача 1.** Розмін монет (Coin Change Problem). Дано набір номіналів монет coins та суму amount. Знайти мінімальну кількість монет, потрібну для формування цієї суми. Вважається, що кількість монет кожного номіналу необмежена.

Жадібний підхід (завжди брати монету найбільшого номіналу, що не перевищує залишок суми) працює для стандартних валютних систем (наприклад, 1, 5, 10, 25). Однак для довільного

набору номіналів він може дати неоптимальний результат. Наприклад, для суми 63 та номіналів [1, 10, 5, 11, 12], жадібний підхід дасть 25 + 25 + 10 + 1 + 1 + 1 (6 монет), тоді як оптимальний розв'язок – 21 + 21 + 21 (3 монети).

Розв'язок за допомогою динамічного програмування.

Нехай dp[i] − мінімальна кількість монет для суми i.

Рекурентне співвідношення:

$$dp[i] = \min_{c \in coins, c \le i}\{1 + dp[i - c]\}.$$

Алгоритм (табуляція):

1. Створити масив dp розміром amount + 1.
2. Ініціалізувати dp = 0, а всі інші елементи − нескінченністю (або amount + 1).
3. Ітерувати від i = 1 до amount. Для кожної суми i ітерувати по всіх монетах c.
4. Якщо i >= c, оновити dp[i] = min(dp[i], 1 + dp[i − c]).
5. Результат − dp[amount].

```
Python
def coin_change(coins, amount):
    # dp[i] буде зберігати мінімальну кількість монет для суми i
    dp = [float('inf')] * (amount + 1)
    dp = 0

    for i in range(1, amount + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], 1 + dp[i - coin])

    return dp[amount] if dp[amount] != float('inf') else -1

print(coin_change([1, 4, 10], 11))  # Виведе 3
print(coin_change([1, 10, 5, 11, 12], 63)) # Виведе 3
```

**Задача 2.** Вибір заявок (Activity Selection Problem). Дано n заявок (активностей), кожна з яких має час початку si та час закінчення fi. Потрібно вибрати максимальну кількість взаємно несумісних (неперетинних) заявок, які може виконати одна людина.

Жадібний вибір полягає в тому, щоб на кожному кроці обирати заявку, яка закінчується найраніше. Це залишає максимум часу для виконання таких заявок.

Алгоритм:

1. Відсортувати всі заявки за зростанням їхнього часу закінчення.

2. Вибрати першу заявку з відсортованого списку.

3. Ітерувати по решті заявок. Якщо час початку поточної заявки більший або дорівнює часу закінчення останньої обраної заявки, обрати поточну заявку.

Приклад: start = {1, 3, 0, 5, 8, 5}, finish = {2, 4, 6, 7, 9, 9}.

1. Сортуємо за часом закінчення: (1, 2), (3, 4), (0, 6), (5, 7), (8, 9), (5, 9).

2. Обираємо (1, 2). Остання обрана закінчилась о 2.

3. Наступна несумісна − (3, 4) (бо 3 ≥ 2). Обираємо її. Остання обрана закінчилась о 4.

4. (0, 6) − несумісна.

5. Наступна несумісна − (5, 7) (бо 5 ≥ 4). Обираємо її. Остання обрана закінчилась о 7.

6. Наступна несумісна − (8, 9) (бо 8 ≥ 7). Обираємо її. Остання обрана закінчилась о 9.

7. (5,9) − несумісна.

Результат: 4 заявки {(1, 2), (3, 4), (5, 7), (8, 9)}.

*Python*

```
def activity_selection(start, finish):
    n = len(start)
    activities = sorted(zip(start, finish), key=lambda x: x[1])

    selected =
    # Вибираємо першу активність
    last_finish_time = -1

    for s, f in activities:
        if s >= last_finish_time:
            selected.append((s, f))
            last_finish_time = f
```

```
    return selected

start =
finish = [2, 4, 6, 7, 9, 9]
result = activity_selection(start, finish)
print(f"Максимальна кількість заявок: {len(result)}")
print(f"Обрані заявки: {result}")
```

## Контрольні питання

1. Які дві необхідні властивості повинна мати задача, щоб її можна було розв’язати за допомогою динамічного програмування? Поясніть кожну з них.

2. Порівняйте підходи «зверху-вниз» (мемоізація) та «знизу-вгору» (табуляція). У яких випадках один може бути кращим за інший?

3. Поясніть властивість жадібного вибору. Чим вона відрізняється від процесу прийняття рішень у динамічному програмуванні?

4. Чому задача про рюкзак 0/1 розв’язується за допомогою ДП, а задача про дробовий рюкзак − за допомогою жадібного алгоритму?

5. Виведіть рекурентне співвідношення для задачі про найдовшу спільну підпослідовність (LCS).

6. Поясніть роль мін-пріоритетної черги в алгоритмі кодування Хаффмана.

7. Що означає, що часова складність алгоритму є «псевдополіноміальною»? Використайте задачу про рюкзак 0/1 як приклад.

8. Виведіть рекурентне співвідношення для задачі про редакційну відстань (Левенштейна). Що представляють три рекурсивні терми?

9. Чому повний перебір для задачі оптимального множення матриць є обчислювально неможливим для великої кількості матриць?

10. Що таке апроксимаційний алгоритм і чому він використовується для задачі про мінімальне покриття множини?

11. Що таке коефіцієнт апроксимації жадібного алгоритму для задачі про покриття множини і що він означає?

12. Поясніть, як реконструювати сам рядок LCS після заповнення ДП-таблиці.

13. Чи може задача мати оптимальну підструктуру, але не бути придатною для жадібного алгоритму? Поясніть на прикладі.

14. Що таке «префіксний код» і чому ця властивість є важливою для кодів Хаффмана?

15. Опишіть оптимізацію пам’яті для задачі про числа Фібоначчі, що зменшує просторову складність з O(n) до O(1).

## Задачі для самостійного опрацювання

1. Розмін монет: знайти загальну кількість способів розміняти суму (не мінімальну кількість монет).

2. Найдовша зростаюча підпослідовність.

3. Мінімальна кількість стрибків для досягнення кінця масиву.

4. Максимальна сума підмасиву (алгоритм Кадане).

5. Задача про розрізання стрижня (Rod Cutting Problem).

6. Задача про розбиття слова (Word Break Problem).

7. Задача про суму підмножини (Subset Sum Problem).

8. Найкоротша спільна суперпослідовність (Shortest Common Supersequence).

9. Головоломка з киданням яєць (Egg Dropping Puzzle).

10. Найдовша паліндромна підпослідовність.

11. Задача про зважений вибір заявок (Job Sequencing with Deadlines).

12. Розбиття на паліндроми (Palindrome Partitioning).

13. Задача булевої парентетизації (Boolean Parenthesization Problem).

14. Задача про складання коробок (Box Stacking Problem).

15. Задача комівояжера (Traveling Salesperson Problem) з використанням ДП та бітових масок.

# ТЕМА 10
# ОСНОВИ ТЕОРІЇ ГРАФІВ ТА ОБХОДИ

## 10.1. Вступ до теорії графів: від абстракції до моделювання

Теорія графів є фундаментальним розділом дискретної математики та комп'ютерних наук, що вивчає об'єкти, які називаються графами. Графи є потужним інструментом для візуалізації та аналізу даних і зв'язків між ними. Вони дають змогу моделювати широкий спектр систем і процесів у реальному світі: від соціальних мереж і транспортних систем до комп'ютерних мереж та біологічних процесів.

На інтуїтивному рівні граф − це сукупність точок, які називаються *вершинами*(або вузлами), та ліній, що з'єднують деякі пари цих точок, які називаються *ребрами*(або дугами). Це візуальне представлення допомагає зрозуміти структуру зв'язків, проте важливо пам'ятати, що граф є, насамперед, абстрактним математичним об'єктом, незалежним від його конкретного зображення на площині.

З математичної точки зору, граф G − це впорядкована пара G = (V, E), де: V − скінченна непорожня множина елементів, які називаються *вершинами*(vertices), E − множина пар вершин з V, які називаються *ребрами*(edges).

Більш строге визначення, яке дає змогу враховувати мультиграфи (графи з кратними ребрами), визначає граф як трійку G = (V, E, I), де V та E − це множини вершин та ребер відповідно, а I − це *функція інцидентності*, яка кожному ребру e∈E ставить у відповідність пару вершин {u,v}∈V, які воно з'єднує. Це розширене визначення дозволяє існування двох різних ребер $e_1$ та $e_2$, які з'єднують одну й ту саму пару вершин, що неможливо у простішій моделі G = (V, E), де ребро і є парою вершин. Для більшості завдань програмування, особливо під час роботи з простими графами, використовується спрощене визначення G = (V, E), яке є більш зручним для реалізації.

Для ефективної роботи з графами потріб'но володіти базовою термінологією:

1. *Суміжність (Adjacency).* Дві вершини називаються *суміжними*, якщо вони з'єднані ребром. Два ребра є *суміжними*, якщо вони мають спільну вершину.

2. *Інцидентність (Incidence).* Ребро e = {u, v} є *інцидентним*[28] вершинам u та v. Аналогічно, вершини u та v є інцидентними ребру e.

3. *Степінь вершини (Vertex Degree).* Степенем вершини v, що позначається як d(v) або deg(v), називається кількість ребер, інцидентних їй. Для орієнтованих графів розрізняють напівстепінь виходу (out-degree) та напівстепінь заходу (in-degree).

− *Ізольована вершина* − вершина зі степенем 0.

− *Висяча (кінцева) вершина* − вершина зі степенем 1.

4. *Петля (Loop)* − ребро, що з'єднує вершину саму із собою, тобто ребро виду {v, v}.

Ці поняття є ключовими для аналізу зв'язності та розробки алгоритмів обходу.

1. *Маршрут (Walk/Route)* − скінченна послідовність вершин та ребер, що їх з'єднують, вигляду $v_0$, $e_1$, $v_1$, $e_2$, ..., $e_k$, $v_k$, де кожне ребро $e_i$ з'єднує вершини $v_{i-1}$ та $v_i$. У маршруті вершини та ребра можуть повторюватися.

2. *Шлях (Path)* − це маршрут, у якому всі вершини (а отже, і ребра) є унікальними. Шлях є фундаментальним поняттям для визначення можливості дістатися з однієї вершини до іншої.

3. *Цикл (Cycle)* − замкнений шлях, у якому початкова та кінцева вершини збігаються ($v_0 = v_k$), а всі проміжні вершини є унікальними. Цикли вказують на наявність альтернативних шляхів або надлишкових зв'язків у графі.

[28] У математиці та інформатиці (теорія графів) – «інцидентний» означає «той, що прилягає / пов'язаний». Вершина інцидентна ребру, якщо це ребро до неї підходить. Ребро інцидентне вершинам, які воно з'єднує. Напр.: Ребро (u, v) інцидентне вершинам u та v.

## 10.2. Класифікація та типологія графів

Вибір правильного типу графа є першим і найважливішим кроком у моделюванні реальної задачі. Різні типи графів дають змогу описувати різні за своєю природою зв'язки. Розглянемо основні з них як інструментарій для моделювання.

У неорієнтованому графі ребра не мають напрямку. Ребро {u, v} представляє симетричний, двосторонній зв'язок між вершинами u та v. Порядок вершин у парі не має значення, тобто {u, v} і {v, u} − це одне й те саме ребро.

Приклад. Соціальна мережа типу Facebook, де дружба є взаємною. Якщо А є другом В, то В автоматично є другом А. Інші приклади − карта доріг (якщо можна проїхати з міста А в місто Б, то зазвичай можна і назад), комп'ютерна мережа, де зв'язок між двома машинами є двонаправленим.

В орієнтованому графі ребра, які називаються ***дугами*** (arcs), мають напрямок. Дуга (u, v) представляє асиметричний, односторонній зв'язок від вершини u (початок) до вершини v (кінець). Тут (u, v) = (v, u). Для вершин орграфа вводяться поняття:

1. Напівстепінь виходу (out-degree) − кількість дуг, що виходять з вершини.

2. Напівстепінь заходу (in-degree) − кількість дуг, що входять у вершину.

Приклад. Соціальна мережа Twitter, де можна «слідкувати» за кимось без взаємності. Інші приклади − мережа гіперпосилань між вебсторінками (посилання з А на В не означає посилання з В на А), граф залежностей між задачами у проєкті (задачу А потрібно виконати перед задачею В).

У зваженому графі кожному ребру (або дузі) присвоєно числове значення, яке називається *вагою*(weight). Вага може представляти відстань, вартість, час, пропускну здатність або будь-яку іншу характеристику зв'язку.

Приклад. Карта автомобільних доріг, де вага ребер − це відстань між містами або орієнтовний час у дорозі. Мережа авіаперельотів, де вага − вартість квитка. Комп'ютерна мережа, де вага − затримка передачі сигналу (ping) або пропускна здатність каналу.

Ці типи не є взаємовиключними. Наприклад, граф може бути одночасно орієнтованим і зваженим, як у випадку моделювання мережі вулиць з одностороннім рухом, де ваги ребер представляють час проїзду кожною вулицею.

Для більш точного опису структури графа використовуються додаткові терміни:

1. *Простий граф (Simple Graph)* – це неорієнтований граф, що не містить петель та кратних (паралельних) ребер, тобто будь-яку пару вершин з'єднує не більше одного ребра. Це найпоширеніший тип графів, що вивчається в базових курсах.

2. *Мультиграф (Multigraph)* − це граф, у якому між однією парою вершин може існувати декілька ребер. Приклад: декілька авіарейсів на день між двома містами.

3. *Псевдограф (Pseudograph)* − це мультиграф, який також може містити петлі.

## 10.3. Способи представлення графів у пам'яті комп'ютера

Для того щоб алгоритми могли працювати з графами, їх структуру потрібно представити у пам'яті комп'ютера. Існує два основних способи представлення, кожен з яких має свої переваги та недоліки, що ілюструють класичний компроміс між часом виконання операцій та об'ємом необхідної пам'яті. Вибір оптимального представлення залежить не стільки від самого графа, скільки від операцій, які планується над ним виконувати найчастіше.

Граф з V вершинами (пронумерованими від 0 до V − 1) представляється у вигляді квадратної матриці A розміром V × V.

Для незваженого графа: елемент матриці A[i][j] = 1, якщо існує ребро між вершинами i та j, і A[i][j] = 0 в іншому випадку.

Для зваженого графа: елемент A[i][j] зберігає вагу ребра між i та j. Якщо ребра немає, значення може бути 0, нескінченністю (∞) або іншим спеціальним маркером.

Властивості:

1. Для неорієнтованого графа матриця суміжності завжди є симетричною відносно головної діагоналі (A[i][j] = A[j][i]).

2. Наявність петлі у вершини і позначається ненульовим елементом на головній діагоналі (A[i][i] = 0).

3. Матриця суміжності повного графа містить одиниці скрізь, крім головної діагоналі.

Граф представляється у вигляді масиву (або вектора) з V списків. Для кожної вершини і (від 0 до V − 1) і-й елемент масиву містить список (або іншу динамічну структуру) всіх вершин, суміжних з і.

У C++ зазвичай реалізується як std::vector<std::vector<int>> або std::vector<std::list<int>>.

У Python часто використовується словник, де ключами є вершини, а значеннями − списки їхніх сусідів, наприклад: graph = {'A': 'B':}.

Для зважених графів у списках зберігаються пари (суміжна_вершина, вага_ребра).

Вибір між матрицею та списком суміжності залежить від структури графа (зокрема, його щільності) та шаблону запитів (query pattern) до нього.

Щільність графа. Граф називається розрідженим (sparse), якщо кількість ребер E значно менша за максимально можливу ($E \ll V^2$). Граф називається щільним (dense), якщо E наближається до $V^2$. Більшість реальних графів (соціальні мережі, вебграф) є розрідженими.

Шаблон запитів. Які операції будуть виконуватися найчастіше? Перевірка наявності конкретного ребра? Перебір усіх сусідів вершини?

Матриця суміжності є кращим вибором для щільних графів, тоді як список суміжності − для розріджених. Однак це лише загальна рекомендація. Наприклад, алгоритми обходу (BFS, DFS) інтенсивно використовують операцію «отримати всіх сусідів вершини», яка є значно ефективнішою у списку суміжності (O(degree(v))) порівняно з матрицею (O(V)). Тому для таких завдань список суміжності є кращим вибором майже завжди.

*Додаткове зауваження про апаратне забезпечення.* Матриця суміжності є неперервним блоком пам'яті. Ітерація по рядку матриці має високу ***кеш-локальність,*** що може призвести до значного прискорення на сучасних процесорах. Список суміжності, особливо реалізований через зв'язані списки, може спричиняти промахи кешу через переходи за вказівниками. Для щільних графів цей апаратний ефект може зробити матрицю швидшою на практиці, навіть якщо асимптотична складність операцій однакова. Нижче наведено порівняльну таблицю складності основних операцій для обох представлень.

Таблиця 10.1 – Порівняльна таблиця складності операцій

| Операція | Матриця суміжності | Список суміжності | Примітки |
|---|---|---|---|
| Просторова складність | $O(V^2)$ | O(V + E) | Список ефективніший для розріджених графів ($E \ll V2$) |
| Додавання ребра | O(1) | O(1) | |
| Видалення ребра | O(1) | O(degree(u)) | Потрібно знайти ребро у списку суміжності вершини u |
| Перевірка суміжності (u, v) | O(1) | O(degree(u)) | Ключова перевага матриці |
| Перебір усіх сусідів v | O(V) | O(degree(v)) | Ключова перевага списку для розріджених графів |
| Додавання вершини | $O(V^2)$ | O(1) | Матрицю потрібно повністю перебудовувати |
| Видалення вершини | $O(V^2)$ | O(V + E) | Потрібно видалити всі інцидентні ребра |

## 10.4. Фундаментальні алгоритми обходу: пошук у ширину (BFS) та в глибину (DFS)

Обхід графа − це систематичний спосіб відвідування всіх його вершин та ребер, причому кожна вершина відвідується рівно один раз. Пошук у ширину (BFS) та пошук у глибину (DFS) є двома фундаментальними стратегіями обходу, що лежать в основі багатьох складніших алгоритмів.

Вони являють собою дуальність у підходах до дослідження: BFS є консервативним та обережним, тоді як DFS − агресивним та спекулятивним.

BFS досліджує граф «пошарово». Починаючи зі стартової вершини s, алгоритм спочатку відвідує всіх її безпосередніх сусідів (вершини на відстані 1), потім сусідів цих сусідів (вершини на відстані 2), і так далі. Цю стратегію можна уявити як кола, що розходяться по воді від кинутого каменя, або як «вогонь», що поширюється від однієї вершини до всіх її сусідів одночасно.

Для реалізації цієї пошарової логіки BFS використовує структуру даних *черга (queue),* що працює за принципом FIFO (First-In, First-Out).

*Логіка роботи алгоритму:*

1. Створити чергу та додати до неї стартову вершину s.

2. Створити масив (або множину) visited для відстеження відвіданих вершин і позначити s як відвідану.

3. Поки черга не порожня:

a. Вилучити вершину u з початку черги.

b. Обробити вершину u (наприклад, вивести її на екран).

c. Для кожного сусіда v вершини u:

i. Якщо v ще не була відвідана:

* Позначити v як відвідану.

* Додати v у кінець черги.

У незваженому графі BFS *гарантовано знаходить* найкоротший шлях (за кількістю ребер) від стартової вершини до всіх інших досяжних вершин. Це відбувається тому, що алгоритм досліджує всі шляхи довжиною k перед тим, як перейти до шляхів довжиною k + 1.

На відміну від BFS, DFS досліджує граф, рухаючись якомога глибше вздовж одного шляху, перш ніж повернутися назад (backtracking) і спробувати інший шлях. Цю стратегію можна порівняти з проходженням лабіринту: ви йдете одним коридором доти, доки не дійдете до глухого кута, після чого повертаєтесь до останнього роздоріжжя і пробуєте інший напрямок.

Для реалізації цієї логіки «занурення» DFS використовує структуру даних *стек (stack),* що працює за принципом LIFO

(Last-In, First-Out). Стек може бути явним (ітеративна реалізація) або неявним, реалізованим через системний стек викликів у *рекурсивній* версії алгоритму.

*Логіка роботи алгоритму (рекурсивна версія):*

1. Створити масив (або множину) visited для відстеження відвіданих вершин.

2. Для стартової вершини s викликати функцію DFS(s).

3. Функція DFS(u):

a. Позначити вершину u як відвідану.

b. Обробити вершину u.

c. Для кожного сусіда v вершини u:

i. Якщо v ще не була відвідана, рекурсивно викликати DFS(v).

DFS чудово підходить для задач, пов’язаних зі структурою шляхів, таких як пошук циклів, перевірка зв’язності та топологічне сортування. Алгоритм не гарантує знаходження найкоротшого шляху, але ефективно досліджує всю глибину графа.

## 10.5. Програмна реалізація алгоритмів BFS та DFS

Цей розділ демонструє, як перетворити теоретичні алгоритми на практичний код. Важливо не просто написати код, а й обрати правильні структури даних у конкретній мові програмування для досягнення оптимальної продуктивності.

Псевдокод BFS

```
BFS(graph, start_node):
  queue = new Queue()
  visited = new Set()

  queue.enqueue(start_node)
  visited.add(start_node)

  while queue is not empty:
    current_node = queue.dequeue()
    print(current_node) // Обробка вершини
```

```
    for neighbor in graph.get_neighbors(current_node):
      if neighbor not in visited:
        visited.add(neighbor)
        queue.enqueue(neighbor)

DFS
DFS(graph, start_node):
 visited = new Set()
 DFSHelper(graph, start_node, visited)

DFSHelper(graph, current_node, visited):
 visited.add(current_node)
 print(current_node) // Обробка вершини

 for neighbor in graph.get_neighbors(current_node):
   if neighbor not in visited:
     DFSHelper(graph, neighbor, visited)
```

Реалізація на C++. У C++ для представлення графа зазвичай використовують std::vector<std::vector<int>>. Для BFS ідеально підходить std::queue, а для DFS − рекурсія або std::stack для ітеративної версії.

```
C++
#include <iostream>
#include <vector>
#include <queue>
#include <list> // Можна використовувати замість vector для списків суміжності

// Клас для представлення графа
class Graph {
   int V; // Кількість вершин
   std::vector<std::list<int>> adj; // Список суміжності

public:
   Graph(int V) : V(V), adj(V) {}

   // Додавання ребра в неорієнтований граф
   void addEdge(int u, int v) {
      adj[u].push_back(v);
```

```
        adj[v].push_back(u);
    }

    // Реалізація BFS
    void BFS(int start_node) {
        std::vector<bool> visited(V, false);
        std::queue<int> queue;

        visited[start_node] = true;
        queue.push(start_node);

        std::cout << "BFS traversal starting from vertex " << start_node
<< ": ";

        while (!queue.empty()) {
            int u = queue.front();
            queue.pop();
            std::cout << u << " ";

            // Перебираємо всіх суміжних вершин
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    queue.push(v);
                }
            }
        }
        std::cout << std::endl;
    }

    // Допоміжна рекурсивна функція для DFS
    void DFSHelper(int u, std::vector<bool>& visited) {
        visited[u] = true;
        std::cout << u << " ";

        // Рекурсивно відвідуємо всіх невідіданих сусідів
        for (int v : adj[u]) {
            if (!visited[v]) {
                DFSHelper(v, visited);
            }
        }
    }
```

```
    // Реалізація DFS
    void DFS(int start_node) {
        std::vector<bool> visited(V, false);
        std::cout << "DFS traversal starting from vertex " << start_node
<< ": ";
        DFSHelper(start_node, visited);
        std::cout << std::endl;
    }
};

int main() {
    Graph g(6); // Створюємо граф з 6 вершинами
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);
    g.addEdge(2, 5);
    g.addEdge(3, 4);

    g.BFS(0); // BFS з вершини 0
    g.DFS(0); // DFS з вершини 0

    return 0;
}
```

У Python для списку суміжності зручно використовувати словник зі списками. Для черги в BFS критично важливо використовувати collections.deque, оскільки видалення елемента з початку звичайного списку (list.pop(0)) є операцією складності O(N), що погіршило б загальну складність BFS до O(V·E). deque.popleft() виконується за O(1).

```
Python
from collections import defaultdict, deque

# Клас для представлення графа
class Graph:
    def __init__(self):
        # Використовуємо defaultdict для зручного додавання ребер
        self.adj = defaultdict(list)
```

```
        # Додавання ребра в неорієнтований граф
        def add_edge(self, u, v):
            self.adj[u].append(v)
            self.adj[v].append(u)

        # Реалізація BFS
        def bfs(self, start_node):
            # Використовуємо множину для відвіданих вершин для O(1) перевірки
            visited = {start_node}
            # Використовуємо deque для ефективної черги
            queue = deque([start_node])

            print(f"BFS traversal starting from vertex {start_node}: ", end="")

            while queue:
                u = queue.popleft()
                print(u, end=" ")

                for v in self.adj[u]:
                    if v not in visited:
                        visited.add(v)
                        queue.append(v)
            print()

        # Допоміжна рекурсивна функція для DFS
        def _dfs_helper(self, u, visited):
            visited.add(u)
            print(u, end=" ")

            for v in self.adj[u]:
                if v not in visited:
                    self._dfs_helper(v, visited)

        # Реалізація DFS
        def dfs(self, start_node):
            visited = set()
            print(f"DFS traversal starting from vertex {start_node}: ", end="")
            self._dfs_helper(start_node, visited)
            print()
```

```
# Приклад використання
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(2, 4)
    g.add_edge(2, 5)
    g.add_edge(3, 4)

    g.bfs(0)
    g.dfs(0)
```

Часова складність. Для представлення *списком суміжності* і BFS, і DFS мають часову складність O(V + E). Це пояснюється тим, що кожна вершина додається до черги / стека і вилучається звідти рівно один раз (складова O(V)), і кожне ребро переглядається двічі (по одному разу з кожного кінця) для неорієнтованого графа (складова O(E)).

Для представлення *матрицею суміжності* складність обох алгоритмів стає $O(V^2)$, оскільки для знаходження всіх сусідів однієї вершини потрібно переглянути весь відповідний рядок матриці, що займає O(V) часу. Ця операція виконується для кожної з V вершин.

Просторова складність. Складність за пам’яттю для обох алгоритмів становить O(V). Це охоплює пам’ять для зберігання масиву visited (O(V)) та пам’ять для черги або стека. У найгіршому випадку (наприклад, для графа-зірки в BFS або графа-ланцюжка в DFS) черга / стек може містити до V − 1 вершин.

## 10.6. Застосування обходів: пошук компонент зв'язності

Однією з базових задач аналізу графів є визначення його зв'язності. Чи можна з будь-якої точки мережі дістатися до будь-якої іншої? Якщо ні, то на скільки ізольованих «островів» розпадається мережа? Алгоритми обходу є ідеальним інструментом для відповіді на ці питання.

*Зв'язний граф* (Connected Graph) − це неорієнтований граф, він є зв'язним, якщо для будь-яких двох його вершин u та v існує шлях, що їх з'єднує.

*Компонента зв'язності* (Connected Component) неорієнтованого графа − це максимальний за включенням зв'язний підграф. Іншими словами, це така множина вершин, що будь-які дві вершини з цієї множини є досяжними одна від одної, і жодна вершина з цієї множини не має ребра до вершини за її межами. Довільний неорієнтований граф є об'єднанням однієї або декількох компонент зв'язності.

Алгоритм для знаходження всіх компонент зв'язності є елегантним застосуванням будь-якого з базових обходів (DFS або BFS) і демонструє загальний шаблон для аналізу графів.

Суть цього «мета-алгоритму» або «шаблону виявлення компонент» полягає в такому:

1. Запустити стандартний обхід (наприклад, DFS) з довільної вершини s.
2. Цей обхід відвідає всі вершини, досяжні з s. Ця множина відвіданих вершин і є першою компонентою зв'язності.
3. Після завершення обходу потрібно перевірити, чи залишилися в графі невідвідані вершини.
4. Якщо так, це означає, що граф незв'язний. Потрібно обрати будь-яку невідвідану вершину і запустити з неї новий обхід, який знайде наступну компоненту зв'язності.
5. Цей процес повторюється доти, доки всі вершини графа не будуть відвідані.

Цей підхід, що складається з зовнішнього циклу по всіх вершинах та внутрішнього виклику обходу, є потужним інструментом, який гарантує, що ми запустимо дослідження в кожній ізольованій частині графа.

Розглянемо незв’язний граф з 8 вершинами та наступними ребрами: {0, 1},{1, 2},{2,0},{3, 4},{5, 6},{5, 7}.

*Кроки алгоритму (з використанням DFS):*

1. Створюємо масив visited розміром 8, ініціалізований false.
2. Починаємо цикл від i = 0 до 7.
3. i = 0:visited є false. Запускаємо DFS(0).

− *DFS(0) відвідує 0, потім 1, потім 2. Всі вони позначаються як visited.*

− *Перша компонента: {0, 1, 2}.*

4. i = 1, i = 2:visited та visited вже true, пропускаємо.
5. i = 3:visited є false. Запускаємо DFS(3).

− *DFS(3) відвідує 3, потім 4.*

− *Друга компонента: {3, 4}.*

6. i = 4:visited вже true, пропускаємо.
7. i = 5:visited є false. Запускаємо DFS(5).

− *DFS(5) відвідує 5, потім 6, потім повертається і відвідує 7.*

− *Третя компонента: {5, 6, 7}.*

8. i = 6, i = 7:visited вже true, пропускаємо.

Цикл завершено. Ми знайшли 3 компоненти зв’язності.

Реалізація на Python:

```
Python
from collections import defaultdict

class Graph:
   def __init__(self):
      self.adj = defaultdict(list)
      self.V = 0

   def add_edge(self, u, v):
      self.adj[u].append(v)
      self.adj[v].append(u)
      self.V = max(self.V, u, v) + 1
```

```
    def _dfs_util(self, u, visited, component):
        visited.add(u)
        component.append(u)
        for v in self.adj[u]:
            if v not in visited:
                self._dfs_util(v, visited, component)

    def find_connected_components(self):
        visited = set()
        components =
        for i in range(self.V):
            if i not in visited:
                component =
                self._dfs_util(i, visited, component)
                components.append(component)
        return components

# Приклад використання
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(3, 4)
    g.add_edge(5, 6)
    g.add_edge(5, 7)

    # Додамо ізольовану вершину для повноти
    g.V = 9 # Вручну оновимо кількість вершин
    g.add_edge(8, 8) # Петля для вершини 8

    all_components = g.find_connected_components()
    print(f"Number of connected components: {len(all_components)}")
    for i, comp in enumerate(all_components):
        print(f"Component {i+1}: {sorted(comp)}")
```

## Контрольні питання

1. Дайте формальне визначення графа.
2. Яка різниця між орієнтованим та неорієнтованим графом? Наведіть по одному прикладу для кожного.
3. Що таке зважений граф і що може представляти вага ребра?
4. Поясніть ключові відмінності між матрицею суміжності та списком суміжності. У яких випадках кожен зі способів є кращим?
5. Яка максимальна кількість ребер у простому неорієнтованому графі з V вершинами?
6. Яка структура даних лежить в основі BFS і чому? Яку ключову властивість має цей алгоритм щодо шляхів у незваженому графі?
7. Яка структура даних лежить в основі DFS? Поясніть рекурсивний та ітеративний підходи до його реалізації.
8. Чому часова складність BFS та DFS на графі, представленому списком суміжності, дорівнює $O(V + E)$, а не $O(V \cdot E)$?
9. Опишіть алгоритм знаходження всіх компонент зв'язності в неорієнтованому графі.
10. Чи може простий граф з 15 вершинами мати всі вершини степеня 5? Поясніть, використовуючи лему про рукостискання.

## Задачі для самостійного опрацювання

1. Напишіть клас Graph, який підтримує представлення графа у вигляді списку суміжності. Реалізуйте методи для додавання вершин та ребер.
2. Для заданого графа та стартової вершини виведіть порядок обходу вершин для BFS та DFS.
3. Напишіть функцію, що перевіряє, чи існує шлях між двома заданими вершинами u та v у графі. (Підказка: використайте BFS або DFS).
4. Модифікуйте алгоритм BFS для знаходження та повернення довжини найкоротшого шляху між двома вершинами в незваженому графі.

5. Напишіть програму, яка визначає, чи є заданий неорієнтований граф зв’язним. (Підказка: використайте алгоритм пошуку компонент зв’язності).

6. Напишіть функцію для визначення наявності циклу в неорієнтованому графі. (Підказка: під час DFS, якщо ви зустрічаєте вже відвідану вершину, яка не є вашим безпосереднім предком у дереві обходу, ви знайшли цикл).

7. Уявіть соціальну мережу у вигляді графа. Напишіть програму, яка знаходить усі окремі групи друзів (компоненти зв’язності).

# ТЕМА 11
# НАЙКОРОТШІ ШЛЯХИ У ГРАФАХ

## 11.1. Вступ до задачі пошуку найкоротшого шляху

У теорії графів однією з найбільш фундаментальних та широко застосовуваних є задача пошуку найкоротшого шляху. Вона виникає в різноманітних сферах, від повсякденної GPS-навігації до складної маршрутизації мережевих пакетів в інтернеті та аналізу біологічних мереж.

Формально, ми працюємо зі зваженим графом G = (V, E, w), де V − це множина вершин (вузлів), E − множина ребер (з'єднань), а w − функція ваги, що кожному ребру e∈E ставить у відповідність деяке числове значення w(e), яке можна інтерпретувати як «вартість», «відстань» або «час» проходження цього ребра. Шлях у графі від вершини s до вершини t − це послідовність вершин ($v_0$, $v_1$, …, $v_k$), де $v_0$ = s, $v_k$ = t, і для кожного i від 0 до k − 1 існує ребро ($v_i$, $v_{i+1}$). Вагою шляху є сума ваг усіх ребер, що його складають: $\sum_{i=0}^{k-1} w(v_i,\ v_{i+1})$.

Задача пошуку найкоротшого шляху полягає у знаходженні такого шляху між заданою парою вершин, вага якого є мінімальною серед усіх можливих шляхів між цими вершинами.

Залежно від постановки, задачі пошуку найкоротшого шляху можна класифікувати на два основні типи:

1. Single-Source Shortest Path (SSSP) − задача пошуку найкоротших шляхів від однієї, наперед заданої, початкової вершини s до всіх інших вершин графу v∈V. Це найпоширеніший варіант задачі, який розв'язують, зокрема, алгоритми Дейкстри та Белл-мана-Форда.

2. All-Pairs Shortest Path (APSP) − задача пошуку найкоротших шляхів між кожною можливою парою вершин (u, v) у графі. Цю задачу можна розв'язати, запустивши SSSP-алгоритм |V| разів (по одному для кожної вершини), але існують і більш спеціалізовані методи, як-от алгоритм Флойда-Воршелла, що може бути ефективнішим для певних типів графів.

В основі більшості алгоритмів пошуку найкоротшого шляху лежить операція, що називається *релаксацією* (або послабленням) ребра. Ідея полягає в ітеративному покращенні оцінки відстані до певної вершини. На початку роботи алгоритмів ми знаємо лише, що відстань до початкової вершини s дорівнює 0, а до всіх інших − нескінченність (тобто ми ще не знайшли жодного шляху до них).

Нехай dist[v] − це поточна відома найменша вага шляху від s до вершини v. Релаксація ребра (u, v) з вагою w(u, v) полягає в перевірці, чи можемо ми покращити шлях до вершини v, пройшовши через вершину u. Це перевіряється умовою:

$$\text{If} dist[u] + w(u, v) < dist[v].$$

Якщо ця умова істинна, це означає, що ми знайшли коротший шлях до v, і ми оновлюємо його вагу:

$$dist[v] := dist[u] + w(u, v).$$

Ця проста операція є будівельним блоком для всіх алгоритмів, що розглядаються далі. Ключова відмінність між алгоритмами полягає не в самій операції, а в стратегії та порядку, в якому вони виконують релаксацію ребер графу.

## 11.2. Алгоритм Дейкстри

Алгоритм Дейкстри, запропонований Едсгером Дейкстрою 1956 року, є одним із найвідоміших алгоритмів для розв’язання задачі SSSP. Він знаходить найкоротші шляхи від однієї вершини до всіх інших у зваженому орієнтованому графі за умови, що ваги всіх ребер є невід’ємними.

Алгоритм Дейкстри є класичним прикладом *жадібного алгоритму*. Його стратегія полягає в тому, щоб на кожному кроці робити локально оптимальний вибір, а саме − обирати для обробки ще не відвідану вершину, до якої на даний момент відомий найкоротший шлях від початкової вершини s. Алгоритм підтримує множину відвіданих вершин, для яких найкоротший шлях вже остаточно знайдений. На кожній ітерації він «розширює» цю множину, додаючи до неї найближчу «прикордонну» вершину.

Критична умова невід'ємності ваг ребер є запорукою коректності цього жадібного підходу. Якби в графі були ребра з від'ємною вагою, вибір найближчої вершини не гарантував би, що пізніше не знайдеться обхідний шлях через інші вершини, який завдяки від'ємному ребру виявиться коротшим.

Для своєї роботи алгоритм використовує такі структури даних:

1. Масив dist, де dist[v] зберігає поточну довжину найкоротшого шляху від s до v.

2. Множину відвіданих вершин, щоб не обробляти одну й ту саму вершину двічі.

3. Структуру даних для ефективного вибору невідвіданої вершини з мінімальним значенням dist.

Розглянемо роботу алгоритму на прикладі графа, зображеного нижче. Нехай початковою вершиною є A.

*Граф:*

- Вершини: {A, B, C, D, E}
- Ребра: (A, B, 4), (A, C, 2), (B, C, 5), (B, D, 10), (C, D, 3), (C, E, 8), (D, E, 4)

*Крок 0: Ініціалізація*

- dist = {A: 0, B: ∞, C: ∞, D: ∞, E: ∞}
- Множина невідвіданих вершин: {A, B, C, D, E}
- Відвідані вершини: {}

*Крок 1: Обробка вершини A*

1. Обираємо вершину з найменшою відстанню серед невідвіданих. Це A (dist[A] = 0).

2. Переміщуємо A до множини відвіданих.

3. Релаксуємо ребра, що виходять з A:

− Ребро (A, B): dist[A] + 4 = 4 < dist = \infty. Оновлюємо dist = 4.

− Ребро (A, C): dist[A] + 2 = 2 < dist[C] = \infty. Оновлюємо dist[C] = 2.

4. Стан dist: {A: 0, B: 4, C: 2, D: ∞, E: ∞}.

*Крок 2: Обробка вершини C*

1. Серед невідвіданих {B, C, D, E} обираємо вершину з найменшою відстанню. Це C (dist[C] = 2).

2. Переміщуємо C до множини відвіданих.

3. Релаксуємо ребра, що виходять з C:

− Ребро (C, D): dist[C] + 3 = 2 + 3 = 5 < dist = \infty. Оновлюємо dist = 5.

− Ребро (C, E): dist[C] + 8 = 2 + 8 = 10 < dist[E] = \infty. Оновлюємо dist[E] = 10.

− Ребро (C, B): dist[C] + 5 = 2 + 5 = 7 \nless dist = 4. dist не змінюється.

4. Стан dist: {A: 0, B: 4, C: 2, D: 5, E: 10}.

*Крок 3: Обробка вершини B*

1. Серед невідвіданих {B, D, E} обираємо вершину B (dist = 4).

2. Переміщуємо B до множини відвіданих.

3. Релаксуємо ребра, що виходять з B: ребро (B, D): dist + 10 = 4 + 10 = 14 \nless dist = 5. dist не змінюється.

4. Стан dist: {A: 0, B: 4, C: 2, D: 5, E: 10}.

*Крок 4: Обробка вершини D*

1. Серед невідвіданих {D, E} обираємо вершину D (dist = 5).

2. Переміщуємо D до множини відвіданих.

3. Релаксуємо ребра, що виходять з D: ребро (D, E): dist + 4 = 5 + 4 = 9 < dist[E] = 10. Оновлюємо dist[E] = 9.

4. Стан dist: {A: 0, B: 4, C: 2, D: 5, E: 9}.

*Крок 5: Обробка вершини E*

1. Серед невідвіданих залишається лише E (dist[E] = 9). Обираємо її.

2. З E немає вихідних ребер до невідвіданих вершин.

Завершення

Множина невідвіданих вершин порожня. Алгоритм завершено. Остаточні найкоротші відстані від A: {A: 0, B: 4, C: 2, D: 5, E: 9}.7

Нижче наведено псевдокод алгоритму Дейкстри.

```
function Dijkstra(Graph, source):
  // Ініціалізація
  for each vertex v in Graph.Vertices:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
  dist[source] ← 0
```

```
Q ← a priority queue of all vertices in Graph.Vertices
// Головний цикл
while Q is not empty:
  u ← vertex in Q with minimum dist[u]
  remove u from Q

  for each neighbor v of u:
    alt ← dist[u] + weight(u, v)
    if alt < dist[v]:
      dist[v] ← alt
      prev[v] ← u
      // Оновити пріоритет v в Q (decrease-key)
return dist, prev
```

Доведення коректності алгоритму зазвичай виконується методом від супротивного та спирається на інваріант циклу. Інваріант полягає в тому, що для кожної вершини v, яка вже була вилучена з пріоритетної черги (тобто відвідана), значення dist[v] є остаточною довжиною найкоротшого шляху від source до v.

Припустимо, що це не так. Нехай u − перша вершина, для якої алгоритм установив неправильну (завищену) відстань dist[u]. Це означає, що існує якийсь інший, коротший шлях до u. Розглянемо такий найкоротший шлях P від source до u. Цей шлях має починатися у множині вже відвіданих вершин і в якийсь момент «вийти» з неї, перейшовши до невідвіданої вершини. Нехай (x, y) − перше ребро на шляху P, де x − відвідана вершина, а y – невідвідана.

Оскільки x була оброблена раніше за u, і алгоритм обрав u як вершину з мінімальною відстанню, то dist[u] <= dist[x]. За інваріантом, dist[x] − це правильна найкоротша відстань до x. Шлях до y через x має довжину dist[x] + w(x, y). Оскільки ваги невід’ємні, w(x, y) >= 0, отже, довжина шляху до y не менша за dist[x]. Коли алгоритм обробляв вершину x, він релаксував ребро (x, y), тому dist[y] було встановлено в значення, не більше ніж dist[x] + w(x, y).

Оскільки y лежить на найкоротшому шляху до u і передує u (або є самою u), то найкоротший шлях до y не довший за найкоротший шлях до u. Звідси випливає, що dist[y] <= dist(source, u) < dist[u].

Але це суперечить нашому вибору u як невідвіданої вершини з мінімальною відстанню. Отже, наше припущення було хибним, і dist[u] є коректним.

Ефективність алгоритму Дейкстри сильно залежить від способу реалізації пріоритетної черги Q. Найпростіший спосіб − зберігати всі невідвідані вершини у звичайному масиві або списку. На кожному кроці головного циклу для знаходження вершини u з мінімальною відстанню потрібно буде переглянути весь масив. Ця операція займає час O(V). Оскільки головний цикл виконується V разів (по одному для кожної вершини), загальна часова складність такої реалізації становить $O(V^2)$. Цей підхід є виправданим для щільних графів, де кількість ребер E наближається до $V^2$. Для розріджених графів, де $E \ll V^2$, значно ефективнішою є реалізація з використанням мінімальної пріоритетної черги (min-heap). Ініціалізація займає O(V). Основний цикл виконується V разів. На кожній ітерації виконується операція вилучення мінімуму (extract-min), яка для бінарної купи коштує O(logV). Крім того, відбувається релаксація всіх вихідних ребер. Усього за час роботи алгоритму кожне ребро буде релаксовано один раз. Після успішної релаксації ребра (u, v) потрібно оновити пріоритет вершини v в черзі (операція decrease-key), що також коштує O(logV). Загальна складність становить O(VlogV + ElogV). Для зв’язних графів ($E \geq V - 1$) це спрощується до O(ElogV).

Варто зазначити, що стандартні бібліотеки C++ (std::priority_queue) та Python (heapq) не надають прямої операції decrease-key. Цю проблему зазвичай вирішують за допомогою «лінивого оновлення»: замість зміни пріоритету існуючого елемента, в чергу просто додається новий елемент (нова_відстань, вершина). Під час вилучення елемента з черги перевіряється, чи не є він «застарілим» (тобто чи не знайдено вже коротший шлях до цієї вершини). Хоча це може збільшити розмір черги до O(E), асимптотична складність залишається O(ElogE), що для графів без паралельних ребер еквівалентно O(ElogV).

Приклади коду

*Python*

```
import heapq

def dijkstra_python(graph, start_vertex):
    """
    Реалізація алгоритму Дейкстри з використанням пріоритетної черги.

    :param graph: Граф, представлений у вигляді словника списків суміжності.
                  Формат: {'A': {'B': 4, 'C': 2},...}
    :param start_vertex: Початкова вершина.
    :return: Словник з найкоротшими відстанями.
    """
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start_vertex] = 0

    # Пріоритетна черга зберігає кортежі (відстань, вершина)
    pq = [(0, start_vertex)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)

        # "Ліниве оновлення": ігноруємо застарілі записи в черзі
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

# Приклад використання
example_graph = {
    'A': {'B': 4, 'C': 2},
```

```
    'B': {'C': 5, 'D': 10},
    'C': {'D': 3, 'E': 8},
    'D': {'E': 4},
    'E': {}
}
print(dijkstra_python(example_graph, 'A'))
# Вивід: {'A': 0, 'B': 4, 'C': 2, 'D': 5, 'E': 9}
```

```
C++
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <limits>

using namespace std;

const int INF = numeric_limits<int>::max();

// Для зручності використовуємо псевдонім для пари (відстань, вершина)
using iPair = pair<int, int>;

void dijkstra_cpp(const map<int, vector<pair<int, int>>>& adj, int start_vertex, int num_vertices) {
    // Пріоритетна черга для зберігання вершин (min-heap)
    // greater<iPair> робить чергу мінімальною за першим елементом пари
    priority_queue<iPair, vector<iPair>, greater<iPair>> pq;

    // Вектор для зберігання відстаней
    vector<int> dist(num_vertices, INF);

    // Ініціалізація
    pq.push(make_pair(0, start_vertex));
    dist[start_vertex] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
```

```
            // Ітерація по всіх суміжних вершинах
            if (adj.count(u)) {
               for (auto const& [v, weight] : adj.at(u)) {
                  // Релаксація ребра (u, v)
                  if (dist[u]!= INF && dist[u] + weight < dist[v]) {
                     dist[v] = dist[u] + weight;
                     pq.push(make_pair(dist[v], v));
                  }
               }
            }
         }

         // Вивід результатів
         cout << "Вершина \t Відстань від початкової" << endl;
         for (int i = 0; i < num_vertices; ++i) {
            cout << i << "\t\t" << (dist[i] == INF? "INF" : to_string(dist[i]))
<< endl;
         }
      }

      int main() {
         int V = 5; // Кількість вершин
         map<int, vector<pair<int, int>>> adj;

         // Додавання ребер: adj[u].push_back({v, weight})
         adj.push_back({1, 4});
         adj.push_back({2, 2});
         adj.[1]push_back({2, 5});
         adj.[1]push_back({3, 10});
         adj.[2]push_back({3, 3});
         adj.[2]push_back({4, 8});
         adj.[3]push_back({4, 4});

         dijkstra_cpp(adj, 0, V);

         return 0;
      }
```

## 11.3. Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда є більш універсальним методом розв'язання задачі SSSP, ніж алгоритм Дейкстри. Його ключова перевага полягає в здатності коректно працювати з графами, що містять ребра з *від'ємною вагою*. Крім того, він дає змогу виявляти наявність у графі циклів з від'ємною сумарною вагою, що є важливою задачею саме по собі.

Жадібна стратегія Дейкстри зазнає невдачі, коли в графі є від'ємні ваги. Розглянемо простий приклад: граф з вершинами S, A, B та ребрами (S, A, 5), (S, B, 2), (A, B, -4).

1. Дейкстра починає з S. Відстані: dist[A] = 5, dist = 2.

2. Наступною обирається вершина B, оскільки dist = 2 є мінімальною. Алгоритм фіксує цю відстань як остаточну.

3. Потім обирається A. Релаксується ребро (A, B): dist[A] + (−4) = 5 − 4 = 1. Нова відстань до B (1) менша за стару (2). Але алгоритм Дейкстри вже «завершив» обробку B і не повернеться до неї. У результаті буде знайдено неправильний шлях до B.

Алгоритм Беллмана-Форда уникає цієї проблеми, оскільки він не робить жадібних припущень. Замість цього він систематично переглядає всі ребра графа, що гарантує знаходження правильного шляху, якщо він існує.

В основі алгоритму Беллмана-Форда лежить принцип динамічного програмування. Він ґрунтується на тому факті, що найкоротший простий шлях (без циклів) у графі з |V| вершинами не може містити більше ніж |V| − 1 ребер.

Алгоритм працює ітеративно. Після i-ї ітерації він гарантовано знаходить правильні найкоротші шляхи до всіх вершин, які можна досягти з початкової вершини, використовуючи не більше ніж i ребер. Відповідно, після |V| − 1 ітерацій будуть знайдені всі найкоротші прості шляхи.

Процес виглядає так:

1. Ініціалізація. Відстань до початкової вершини dist[s] встановлюється в 0, а до всіх інших − у нескінченність.

2. Ітерації релаксації. Алгоритм виконує |V| − 1 разів такий крок: для кожного ребра (u, v) в графі виконується його релаксація.

Тобто перевіряється умова if dist[u] + w(u, v) < dist[v] і, за потреби, оновлюється dist[v].

Цей систематичний підхід гарантує, що інформація про короткі шляхи поступово «поширюється» по графу від початкової вершини.

Покроковий розбір алгоритму на прикладі.

Розглянемо граф з вершинами {S, A, B, C, D} і ребрами (S, A, 6), (S, B, 7), (A, C, 5), (A, D, -4), (B, C, 8), (B, D, 9), (C, A, -2), (D, S, 2), (D, C, 7). Початкова вершина − S. Кількість вершин |V| = 5, отже, потрібно виконати 4 ітерації.

*Ініціалізація:*dist = {S: 0, A: ∞, B: ∞, C: ∞, D: ∞}

*Ітерація 1:*

Релаксуємо всі ребра

(S, A, 6): dist[A] = 6

(S, B, 7): dist = 7

... (інші ребра не релаксуються, бо dist їх початків − ∞)

*Результат 1*:dist = {S: 0, A: 6, B: 7, C: ∞, D: ∞}

*Ітерація 2:*

(A, C, 5): dist[C] = min(\infty, 6 + 5) = 11

(A, D, -4): dist = min(\infty, 6 - 4) = 2

(B, C, 8): dist[C] = min(11, 7 + 8) = 11

(B, D, 9): dist = min(2, 7 + 9) = 2

(C, A, -2): dist[A] = min(6, 11 - 2) = 6 (якщо dist[C] з попереднього кроку)

...

*Результат 2:*dist = {S: 0, A: 6, B: 7, C: 11, D: 2}

*Ітерація 3:*

(C, A, -2): dist[A] = min(6, 11 - 2) = 6

(D, C, 7): dist[C] = min(11, 2 + 7) = 9

(D, S, 2): dist не змінюється

...

*Результат 3:*dist = {S: 0, A: 6, B: 7, C: 9, D: 2}

*Ітерація 4:*

(C, A, -2): dist[A] = min(6, 9 - 2) = 6

... (більше оновлень не відбувається)

*Результат 4:*dist = {S: 0, A: 6, B: 7, C: 9, D: 2}

Після 4 ітерацій масив відстаней стабілізувався. Це остаточні значення найкоротших шляхів.

*Виявлення циклів від'ємної ваги: логіка V-ї ітерації*

Якщо в графі існує цикл, сума ваг ребер якого від'ємна, і цей цикл досяжний з початкової вершини, то поняття «найкоротшого шляху» стає невизначеним. Проходячи по такому циклу знову і знову, ми можемо нескінченно зменшувати вагу шляху.

Алгоритм Беллмана-Форда дає змогу виявити таку ситуацію. Після завершення |V| − 1 основних ітерацій, які гарантовано знаходять найкоротші *прості* шляхи, виконується ще одна, *|V|-та ітерація*релаксації всіх ребер.

Якщо на цій додатковій ітерації вдається релаксувати хоча б одне ребро (тобто dist[u] + w(u, v) < dist[v] для якогось ребра (u, v)), це є беззаперечним свідченням наявності циклу від'ємної ваги. Логіка тут така: якщо після |V| − 1 ітерацій шлях все ще можна скоротити, це означає, що найкоротший шлях до якоїсь вершини містить щонайменше |V| ребер. У графі з |V| вершинами такий шлях обов'язково містить цикл. А оскільки шлях став коротшим, цей цикл повинен мати від'ємну сумарну вагу.

Псевдокод

```
function BellmanFord(Graph, source):
  // Крок 1: Ініціалізація
  for each vertex v in Graph.Vertices:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
  dist[source] ← 0

  // Крок 2: Релаксація ребер |V|-1 разів
  for i from 1 to |V|-1:
    for each edge (u, v) with weight w in Graph.Edges:
      if dist[u] + w < dist[v]:
        dist[v] ← dist[u] + w
        prev[v] ← u

  // Крок 3: Перевірка на цикли від'ємної ваги
  for each edge (u, v) with weight w in Graph.Edges:
    if dist[u] + w < dist[v]:
      error "Граф містить цикл від'ємної ваги"

  return dist, prev
```

Алгоритм складається з двох основних частин. Перша частина − це головний цикл, що виконується |V| − 1 разів. Усередині цього циклу відбувається перебір усіх |E| ребер. Отже, складність цієї частини − O(V · E). Друга частина − перевірка на цикли, яка також вимагає одного проходу по всіх |E| ребрах, що має складність O(E). Так, загальна часова складність алгоритму Беллмана-Форда становить O(V · E).

Приклади коду

*Python*

```
def bellman_ford_python(edges, num_vertices, start_vertex):
    """
    Реалізація алгоритму Беллмана-Форда.

    :param edges: Список ребер графу. Формат: [(u, v, weight),...]
    :param num_vertices: Кількість вершин у графі.
    :param start_vertex: Початкова вершина.
    :return: Список відстаней або повідомлення про цикл.
    """
    dist = [float('inf')] * num_vertices
    dist[start_vertex] = 0

    # |V|-1 ітерацій релаксації
    for _ in range(num_vertices - 1):
        for u, v, weight in edges:
            if dist[u] != float('inf') and dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight

    # Перевірка на цикли від'ємної ваги
    for u, v, weight in edges:
        if dist[u] != float('inf') and dist[u] + weight < dist[v]:
            return "Граф містить цикл від'ємної ваги"

    return dist

# Приклад використання
V = 5
edges_list = [(0, 1, -1), (0, 2, 4), (1, 2, 3), (1, 3, 2), (1, 4, 2), (3, 2, 5), (3, 1, 1), (4, 3, -3)]
print(bellman_ford_python(edges_list, V, 0))
# Вивід: [0, -1, 2, -2, 1]
```

```
C++
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

const int INF = numeric_limits<int>::max();

struct Edge {
   int source, destination, weight;
};

void bellman_ford_cpp(const vector<Edge>& edges, int num_vertices,
int start_vertex) {
   vector<int> dist(num_vertices, INF);
   dist[start_vertex] = 0;

   // |V|-1 ітерацій
   for (int i = 0; i < num_vertices - 1; ++i) {
      for (const auto& edge : edges) {
         if (dist[edge.source]!= INF && dist[edge.source] + edge.weight
< dist[edge.destination]) {
            dist[edge.destination] = dist[edge.source] + edge.weight;
         }
      }
   }

   // Перевірка на цикли
   for (const auto& edge : edges) {
      if (dist[edge.source]!= INF && dist[edge.source] + edge.weight <
dist[edge.destination]) {
         cout << "Граф містить цикл від'ємної ваги" << endl;
         return;
      }
   }

   // Вивід результатів
   cout << "Вершина \t Відстань від початкової" << endl;
   for (int i = 0; i < num_vertices; ++i) {
      cout << i << "\t\t" << (dist[i] == INF? "INF" : to_string(dist[i]))
<< endl;
   }
}
```

```
int main() {
    int V = 5;
    vector<Edge> edges = {
        {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},
        {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}
    };

    bellman_ford_cpp(edges, V, 0);

    return 0;
}
```

## 11.4. Алгоритм Флойда-Воршелла

На відміну від алгоритмів Дейкстри та Беллмана-Форда, які розв'язують задачу SSSP, алгоритм Флойда-Воршелла призначений для розв'язання задачі All-Pairs Shortest Path (APSP) − знаходження найкоротших шляхів між усіма парами вершин у графі. Він також може працювати з ребрами від'ємної ваги, але, як і Беллман-Форд, не може обробляти графи з циклами від'ємної ваги.

Як зазначалося раніше, найпростіший спосіб розв'язати задачу APSP − це запустити SSSP-алгоритм для кожної вершини графа. Якщо ваги ребер невід'ємні, можна |V| разів запустити алгоритм Дейкстри, що дасть загальну складність O(V · ElogV). Якщо є від'ємні ваги, можна використати алгоритм Беллмана-Форда, що дасть складність $O(V^2 \cdot E)$.

Алгоритм Флойда-Воршелла пропонує інший підхід, заснований на динамічному програмуванні, зі стабільною часовою складністю $O(V^3)$. Це робить його особливо привабливим для *щільних графів*, де E наближається до $V^2$. У такому випадку $O(V^3)$ є кращим або порівнянним з O(V · ElogV) або $O(V^2 \cdot E)$.

Основна ідея алгоритму полягає в тому, щоб ітеративно будувати розв'язок. Він розглядає всі вершини по черзі як можливі проміжні вершини на шляху між будь-якою парою вершин (i, j).

Нехай dist[i][j] − довжина найкоротшого шляху від i до j. Алгоритм ітерує по всіх вершинах k від 0 до |V| − 1. На k-тій ітерації він перевіряє для кожної пари (i, j), чи є шлях від i до j через проміжну вершину k коротшим, ніж поточний відомий шлях. Це виражається таким рекурентним співвідношенням:

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j]).$$

Після того, як алгоритм перебере всі можливі вершини k як проміжні, матриця dist буде містити довжини найкоротших шляхів між усіма парами вершин, оскільки були розглянуті всі можливі шляхи.

Розглянемо невеликий граф з 4 вершинами {0, 1, 2, 3}. Початкова матриця відстаней D(0) (матриця суміжності, де відсутність ребра позначається як ∞, а відстань від вершини до себе − 0):

$$D^{(0)} = \begin{pmatrix} 0 & 5 & \infty & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

k = 0 (проміжна вершина 0):

Матриця не змінюється, оскільки 0 є початковою вершиною для першого рядка і не є проміжною для інших шляхів.

k = 1 (проміжна вершина 1):

Перевіряємо всі шляхи, що проходять через 1. Наприклад, шлях з 0 в 2. dist = ∞. dist[1] + dist[1]2 = 5 + 3 = 8$. Отже, dist2` стає 8.

$$D^{(1)} = \begin{pmatrix} 0 & 5 & \mathbf{8} & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

k = 2 (проміжна вершина 2):

Шлях (0, 3): dist[3] = 10. dist + dist[3] = 8 + 1 = 9. Оновлюємо dist[3] до 9.

Шлях (1, 3): dist[1][3] = ∞. dist[1]2 + dist2[3] = 3 + 1 = 4. Оновлюємо dist[1][3] до 4.

$$D^{(2)} = \begin{pmatrix} 0 & 5 & 8 & \mathbf{9} \\ \infty & 0 & 3 & \mathbf{4} \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

k = 3 (проміжна вершина 3):

Матриця не змінюється, оскільки 3 є кінцевою вершиною для всіх оновлених шляхів.

Остаточна матриця D(3) містить найкоротші відстані між усіма парами вершин.

Алгоритм має дуже просту структуру з трьома вкладеними циклами.

Псевдокод

```
function FloydWarshall(graph):
  let dist be a |V| × |V| matrix of distances
  // Ініціалізація
  for i from 1 to |V|:
    for j from 1 to |V|:
      if edge (i, j) exists:
        dist[i][j] ← weight(i, j)
      else:
        dist[i][j] ← INFINITY
  for i from 1 to |V|:
    dist[i][i] ← 0

  // Головні цикли
  for k from 1 to |V|:
    for i from 1 to |V|:
      for j from 1 to |V|:
        if dist[i][k] + dist[k][j] < dist[i][j]:
          dist[i][j] ← dist[i][k] + dist[k][j]

  return dist
```

Структура з трьох вкладених циклів, кожен з яких виконується |V| разів, безпосередньо призводить до часової складності $O(V^3)$. Для зберігання матриці відстаней потрібна пам’ять об’ємом $O(V^2)$.

```
Python
def floyd_warshall_python(graph):
    """
    Реалізація алгоритму Флойда-Воршелла.

    :param graph: Матриця суміжності графа.
    :return: Матриця найкоротших відстаней.
    """
    num_vertices = len(graph)
    dist = [list(row) for row in graph] # Копіюємо матрицю

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                # Перевіряємо, що шляхи через k існують
                if dist[i][k] != float('inf') and dist[k][j] != float('inf'):
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

# Приклад використання
INF = float('inf')
graph_matrix = [0, 5, INF, 10],
    [INF, 0, 3, INF],
    [INF, INF, 0, 1],
    [INF, INF, INF, 0]
shortest_paths = floyd_warshall_python(graph_matrix)
for row in shortest_paths:
    print(row)
# Вивід:
#
# [inf, 0, 3, 4]
# [inf, inf, 0, 1]
# [inf, inf, inf, 0]
```

```
C++
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

const int INF = 1e9; // Використовуємо велике число як нескінченність

void floyd_warshall_cpp(vector<vector<int>>& dist) {
    int V = dist.size();

    for (int k = 0; k < V; ++k) {
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                if (dist[i][k]!= INF && dist[k][j]!= INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}

int main() {
    vector<vector<int>> graph = {
        {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };

    floyd_warshall_cpp(graph);

    for (const auto& row : graph) {
        for (int val : row) {
            if (val == INF) cout << "INF ";
            else cout << val << "   ";
        }
        cout << endl;
    }

    return 0;
}
```

Вибір конкретного алгоритму для пошуку найкоротшого шляху залежить від характеристик задачі: типу графа, наявності ребер з від'ємною вагою та вимог до результату (SSSP чи APSP). Наведена нижче таблиця узагальнює ключові відмінності між розглянутими алгоритмами, що допоможе зробити обґрунтований вибір.

Таблиця 11.1 – Порівняльна таблиця алгоритмів

| Характеристика | Алгоритм Дейкстри | Алгоритм Беллмана-Форда | Алгоритм Флойда-Воршелла |
|---|---|---|---|
| Тип задачі | SSSP (від однієї вершини) | SSSP (від однієї вершини) | APSP (між усіма парами) |
| Обмеження на ваги | Невід'ємні ($w \geq 0$) | Довільні, без циклів від'ємної ваги | Довільні, без циклів від'ємної ваги |
| Часова складність | $O(ElogV)$ з пріоритетною чергою | $O(V \cdot E)$ | $O(V^3)$ |
| Парадигма | Жадібний алгоритм | Динамічне програмування | Динамічне програмування |

## 11.5. Кейс-стаді: оптимізація маршрутів в Uber та Google Maps

Теоретичні алгоритми, розглянуті вище, є фундаментом, на якому побудовані сучасні навігаційні системи, такі як Google Maps та Uber. Однак у реальному світі ці системи стикаються з проблемами величезного масштабу та динамічності, що вимагає застосування більш складних, багаторівневих підходів. Розв'язання задачі маршрутизації в таких системах − це не просто застосування одного алгоритму, а побудова цілої системи алгоритмів, де кожен шар вирішує свою специфічну проблему.

В основі будь-якої навігаційної системи лежить представлення дорожньої мережі у вигляді графа. Перехрестя та розв'язки моделюються як вершини, а ділянки доріг між ними − як ребра.

Вагою ребра зазвичай є очікуваний час у дорозі, який може бути функцією від відстані, дозволеної швидкості та поточних дорожніх умов.

Пряме застосування алгоритму Дейкстри до графів континентального масштабу (з десятками мільйонів вершин) є непрактичним через його «неінформованість» − він досліджує шляхи у всіх напрямках від початкової точки. Для вирішення задачі «точка-точка» значно ефективнішим є *алгоритм A** (A-star). A* є розширенням алгоритму Дейкстри, яке використовує *евристичну*[29] *функцію* h(n) для оцінки відстані від поточної вершини n до цілі. Пріоритет вершини в черзі визначається функцією f(n) = g(n) + h(n), де g(n) − це вже відома вартість шляху від старту до n (як у Дейкстри), а h(n) − евристична оцінка. Якщо евристика є «допустимою» (тобто ніколи не переоцінює реальну вартість), A* гарантовано знаходить оптимальний шлях, але робить це значно швидше, оскільки цілеспрямовано рухається в напрямку цілі.

Навіть A* може бути занадто повільним для миттєвих відповідей у вебсервісах. Тому для статичних мереж (де ваги ребер не змінюються) застосовують потужні техніки попередньої обробки. Однією з найефективніших є *ієрархії скорочень* (Contraction Hierarchies, CH).

Ідея CH полягає в тому, щоб на етапі попередньої обробки (який може тривати хвилини або години) «спростити» граф. Алгоритм ітеративно «скорочує» (видаляє) менш важливі вершини (наприклад, вузли на житлових вулицях), додаючи «шорткати» (shortcut edges) − нові ребра, що зберігають інформацію про найкоротші шляхи через видалені вершини. У результаті створюється ієрархічна структура, де «високорівневі» вершини представляють магістралі та важливі транспортні вузли.

---

[29] Евристика (від грец. *heurisko* – «віднаходжу, відкриваю») – це метод або прийом мислення, який допомагає знаходити рішення, але не гарантує його оптимальності чи правильності. Це своєрідна «корисна підказка», що дає змогу скоротити пошук або спростити задачу. В інформатиці та штучному інтелекті – метод пошуку рішень, який не гарантує точного результату, але істотно зменшує обчислювальні витрати. Використовується в алгоритмах пошуку, оптимізації, планування. Евристична функція в алгоритмі A* оцінює «наскільки близько» поточний стан до мети. Приклад: у шахових програмах – правила на кшталт «король у центрі на початку гри – погано», що допомагають швидше відкидати невигідні ходи.

Під час запиту (який має бути миттєвим) виконується дуже швидкий двонаправлений пошук Дейкстри на цьому ієрархічному графі. Пошук від початкової точки рухається тільки «вгору» по ієрархії, а від кінцевої − «вниз». Ці два пошуки зустрічаються на найвищій точці ієрархії на найкоротшому шляху, досліджуючи при цьому лише крихітну частину вершин графа (сотні замість мільйонів).

Найбільший виклик для навігаційних систем − це динамічність реального світу. Час у дорозі постійно змінюється через трафік, аварії, погодні умови. Попередньо обчислена ієрархія CH, що базується на статичних вагах (наприклад, відстані), тут стає менш ефективною.

Саме на цьому етапі в гру вступають штучний інтелект (ШІ) та машинне навчання (ML). Системи, як Google Maps та Uber, використовують складні ML-моделі для прогнозування ваг ребер (часу в дорозі) у реальному часі. Ці моделі аналізують величезні обсяги даних:

1. Історичні дані − патерни трафіку в певний день тижня та час доби.

2. Дані в реальному часі − анонімні дані з GPS-сигналів смартфонів користувачів, що дають змогу бачити поточну швидкість руху на ділянках доріг.

На основі цих прогнозів ваги ребер у графі динамічно оновлюються. Маршрут, оптимальний хвилину тому, може стати неоптимальним через раптовий затор. Системи постійно перераховують маршрути, іноді навіть під час поїздки, щоб адаптуватися до нових умов. Uber також використовує моделі глибокого навчання та навчання з підкріпленням для оптимізації не тільки часу, але й інших параметрів, таких як вартість поїздки для водія та пасажира, безпека маршруту та загальний досвід користувача.

Отже, сучасна маршрутизація − це гібридна система, що поєднує класичні алгоритми (A*), потужні техніки попередньої обробки (CH) для масштабування та передові моделі ШІ для адаптації до динамічного середовища.

## Контрольні питання

1. Поясніть, чому алгоритм Дейкстри не працює з ребрами від'ємної ваги. Наведіть контрприклад.
2. У чому полягає «жадібний» вибір в алгоритмі Дейкстри?
3. Скільки ітерацій релаксації виконує алгоритм Беллмана-Форда і чому саме стільки?
4. Як алгоритм Беллмана-Форда виявляє наявність циклу від'ємної ваги?
5. Для яких типів графів (щільних чи розріджених) алгоритм Флойда-Воршелла є більш ефективним, ніж |V|-кратний запуск алгоритму Дейкстри? Поясніть чому.
6. Опишіть основну ідею, що лежить в основі техніки Contraction Hierarchies.

## Задачі для самостійного опрацювання

1. Дано граф міст України та вартість авіаперельотів між ними. Напишіть програму, яка знаходить найдешевший маршрут з Києва до всіх обласних центрів.
2. Уявіть собі ринок обміну валют, де ребра графа представляють обмінні курси між валютами. Деякі операції можуть долучати комісію, що можна змоделювати як шлях з вагою, що відповідає логарифму курсу. Напишіть програму, яка знаходить найвигідніший ланцюжок обмінів з однієї валюти в іншу або виявляє можливість «арбітражу» (цикл, що дозволяє отримати прибуток, еквівалентний циклу від'ємної ваги).
3. Дано матрицю відстаней між усіма складами великої логістичної компанії. Потрібно знайти такий склад для розміщення центрального розподільчого центру, щоб сума відстаней від нього до всіх інших складів була мінімальною.
4. Реалізуйте алгоритм Дейкстри для двох різних представлень графа: у вигляді матриці суміжності та у вигляді списків суміжності. Порівняйте продуктивність обох реалізацій на розріджених та щільних графах.

5. Модифікуйте алгоритм Дейкстри для знаходження не найкоротшого, а «найнадійнішого» шляху в мережі. Надійність ребра − це ймовірність його успішного функціонування (число від 0 до 1). Надійність шляху − це добуток надійностей усіх ребер, що його складають. Потрібно знайти шлях з максимальною надійністю. (Підказка: розгляньте перетворення ваг за допомогою логарифмування, щоб звести задачу до пошуку найкоротшого шляху).

# ТЕМА 12
# МІНІМАЛЬНІ КІСТЯКОВІ ДЕРЕВА

## 12.1. Проблема проєктування ефективних мереж

Уявіть, що телекомунікаційна компанія отримала завдання з'єднати новою кабельною мережею декілька міст. Інженери мають карту місцевості, де позначені міста та можливі шляхи прокладання кабелю між ними. Для кожної ділянки між двома містами відома вартість робіт, яка залежить від відстані, рельєфу та інших чинників. Мета компанії − розробити таку мережу, щоб з будь-якого міста можна було дістатися до будь-якого іншого (напряму або через транзитні міста), а загальна вартість прокладання всіх кабелів була мінімальною.

Ця практична задача є класичним прикладом проблеми, яку в теорії графів називають пошуком *мінімального кістякового дерева* (Minimum Spanning Tree, MST). Для її формалізації представимо міста як вершини (vertices) графа, а потенційні кабельні з'єднання − як ребра (edges). Кожному ребру присвоїмо вагу (weight), що дорівнює вартості його прокладання. У такому випадку задача полягає в тому, щоб знайти такий набір ребер, який би з'єднував усі вершини графа, не утворював циклів і мав би найменшу можливу суму ваг.

*Кістяковим (або покривним) деревом* зв'язного, неорієнтованого графа $G = (V, E)$ називається підграф $T = (V, E')$, де $E' \subseteq E$, який є деревом і містить усі вершини з множини $V$.

Ключовими властивостями будь-якого кістякового дерева для графа з $|V|$ вершинами є:

1. Воно містить усі $|V|$ вершин.
2. Воно є зв'язним та ациклічним.
3. Воно містить рівно $|V| - 1$ ребро.

*Мінімальним кістяковим деревом* зв'язного, неорієнтованого зваженого графа $G = (V, E, w)$ називається кістякове дерево $T$, сума ваг ребер якого $w(T) = \sum_{e \in E(T)} w(e)$ є мінімально можливою серед усіх кістякових дерев графа $G$.

Важливо розуміти, що поняття «вага ребра» є абстрактним. Це не обов'язково грошова вартість чи фізична відстань. Вага може представляти час передачі даних у комп'ютерній мережі, рівень затримки (latency), споживання енергії або навіть міру «відмінності» між об'єктами в задачах кластерного аналізу. Ця універсальність робить алгоритми пошуку MST потужним інструментом у найрізноманітніших галузях: від проєктування транспортних мереж та електромереж до біоінформатики (побудова філогенетичних дерев) та машинного навчання.

Для розв'язання цієї задачі існують ефективні алгоритми, які належать до класу жадібних алгоритмів (greedy algorithms). Вони будують розв'язок крок за кроком, на кожному етапі роблячи вибір, який здається найкращим у поточний момент (наприклад, обираючи ребро з найменшою вагою).Примітно, що для багатьох оптимізаційних задач такий підхід не гарантує знаходження глобального оптимуму. Проте, як ми побачимо далі, для задачі пошуку MST жадібна стратегія працює бездоганно, і це є прямим наслідком фундаментальних математичних властивостей цієї проблеми.

## 12.2. Фундаментальні властивості мінімальних кістякових дерев

Перш ніж переходити до алгоритмів, потрібно зрозуміти теоретичні засади, які гарантують їхню коректність. Ці властивості пояснюють, чому локально оптимальні «жадібні» рішення приводять до глобально оптимального результату.

Будь-яке кістякове дерево, включно з мінімальним, має ряд спільних характеристик, що випливають з визначення дерева.

1. *Кількість ребер.* Для графа з V вершинами будь-яке кістякове дерево завжди містить рівно $V - 1$ ребро.

2. *Ациклічність.* У кістяковому дереві немає циклів. Додавання до нього будь-якого ребра з початкового графа, яке не входило до дерева, неминуче створює унікальний цикл.

3. *Зв'язність.* Кістякове дерево є *мінімально зв'язним*. Це означає, що видалення хоча б одного ребра з дерева призведе до втрати зв'язності, і граф розпадеться на два окремі компоненти.

Властивість розрізу є наріжним каменем теорії MST і лежить в основі як алгоритму Пріма, так і алгоритму Крускала.

*Розрізом* графа G = (V, E) називається поділ множини вершин V на дві непусті, неперетинні підмножини S та V−S. Ребро (u, v)∈E*еретинає розріз,* якщо один його кінець належить S, а інший − V−S (тобто, u∈S та v∈V−S).

Теорема 1 (Властивість розрізу). Для будь-якого розрізу (S, V−S) графа G, якщо ребро e має вагу, строго меншу за вагу будь-якого іншого ребра, що перетинає цей розріз, то ребро e належить усім мінімальним кістяковим деревам графа G. Таке ребро називається *легким ребром* (light edge) для даного розрізу.

Ця властивість є логічним двійником властивості розрізу і дає критерій для вилучення ребер з MST.

Теорема 2 (Властивість циклу). Для будь-якого циклу C у графі G, ребро e з цього циклу, вага якого строго більша за вагу будь-якого іншого ребра в C, не може належати жодному мінімальному кістяковому дереву.

Властивості розрізу та циклу є двома сторонами однієї медалі. Властивість розрізу дає критерій для включення ребра в MST («це ребро безпечно додати») і лежить в основі *конструктивних* алгоритмів, таких як алгоритми Пріма та Крускала. Властивість циклу, навпаки, дає критерій для видалення ребра («це ребро небезпечно додавати») і може бути використана в *деструктивних* алгоритмах, які починають з повного графа й послідовно видаляють найважчі ребра, що не порушують зв’язності.

Теорема 3 (Про унікальність MST). Якщо ваги всіх ребер у зв’язному графі є попарно різними, то існує рівно одне, унікальне мінімальне кістякове дерево.

Якщо ж у графі є ребра з однаковою вагою, то для деякого розрізу може існувати декілька легких ребер. У такому випадку вибір будь-якого з них буде коректним, що може призвести до існування кількох різних мінімальних кістякових дерев, які, однак, матимуть однакову загальну вагу.

## 12.3. Алгоритм Крускала

Алгоритм Крускала, запропонований Джозефом Крускалом 1956 року, є одним із найвідоміших жадібних алгоритмів для знаходження MST. Його стратегія полягає в тому, щоб розглядати ребра графа в порядку зростання їхньої ваги і додавати до результуючого набору ті, що не створюють циклів.

Алгоритм Крускала розглядає граф як сукупність окремих компонент зв'язності, які на початковому етапі складаються з окремих вершин. На кожному кроці він обирає найлегше з доступних ребер і використовує його для об'єднання двох різних компонент в одну. Цей процес продовжується доти, доки всі вершини не будуть об'єднані в єдиний компонент, який і стане мінімальним кістяковим деревом.

Так, алгоритм будує MST, не «вирощуючи» його з однієї точки, а скоріше збираючи його з окремих фрагментів («лісу» дерев), поступово з'єднуючи їх найдешевшими «мостами».

Розглянемо роботу алгоритму на прикладі графа, зображеного на рисунку нижче.

Крок 1: Сортування ребер

Спочатку створимо список усіх ребер графа та відсортуємо його за зростанням ваги.

| Ребро | Вага |
|---|---|
| (C, E) | 2 |
| (D, F) | 3 |
| (A, B) | 4 |
| (E, F) | 4 |
| (B, C) | 5 |
| (F, G) | 5 |
| (A, D) | 6 |
| (C, D) | 7 |
| (B, E) | 8 |

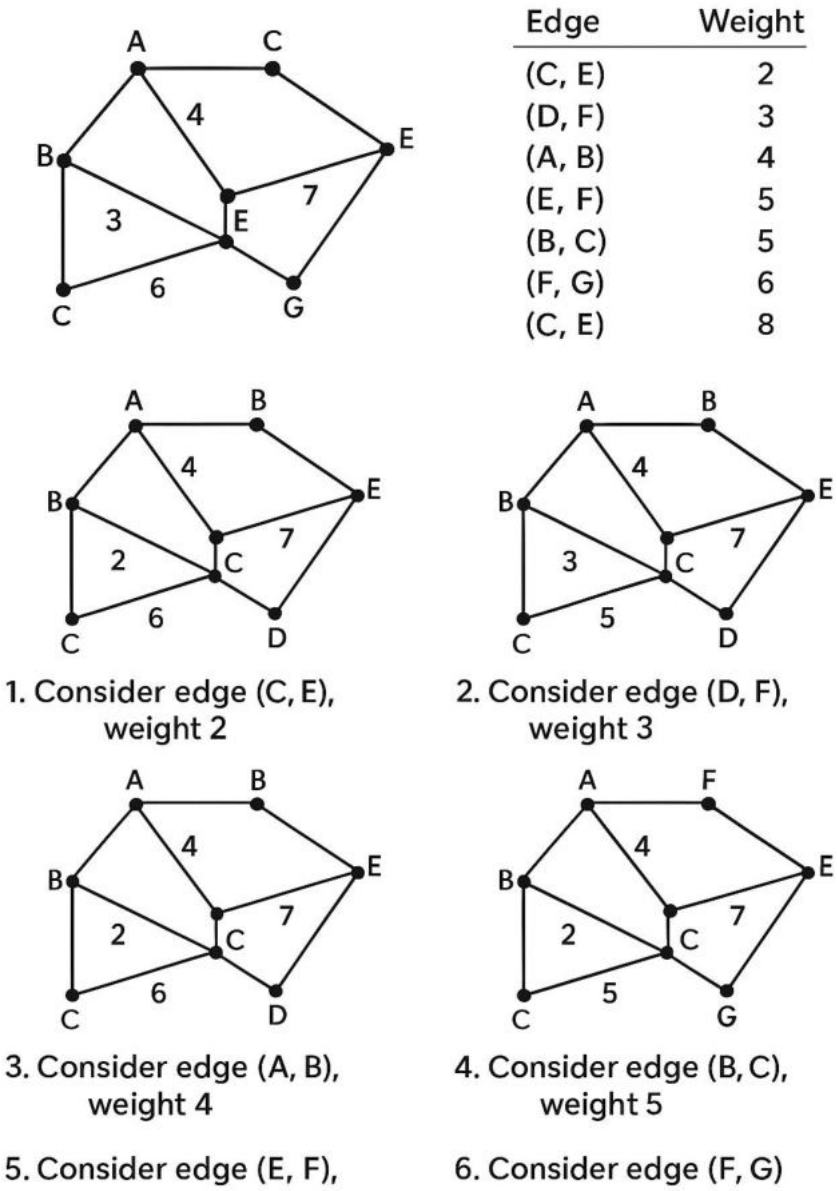



Рисунок 12.1 – Алгоритм Крускала

Крок 2: Побудова MST

Ініціалізуємо MST як порожню множину. Починаємо ітерацію по відсортованому списку ребер. На початку кожна вершина {A, B, C, D, E, F, G} є окремим компонентом.

1. Розглядаємо (C, E), вага 2: Вершини C і E в різних компонентах. Додаємо ребро.

− MST: {(C, E)}

− Компоненти: {A}, {B}, {C, E}, {D}, {F}, {G}

2. Розглядаємо (D, F), вага 3: Вершини D і F в різних компонентах. Додаємо ребро.

− MST: {(C, E), (D, F)}

− Компоненти: {A}, {B}, {C, E}, {D, F}, {G}

3. Розглядаємо (A, B), вага 4: Вершини A і B в різних компонентах. Додаємо ребро.

− MST: {(C, E), (D, F), (A, B)}

− Компоненти: {A, B}, {C, E}, {D, F}, {G}

4. Розглядаємо (E, F), вага 4: Вершини E і F у різних компонентах ({C, E} та {D, F}). Додаємо ребро.

− MST: {(C, E), (D, F), (A, B), (E, F)}

− Компоненти: {A, B}, {C, E, D, F}, {G}

5. Розглядаємо (B, C), вага 5: Вершини B і C в різних компонентах ({A, B} та {C, E, D, F}). Додаємо ребро.

− MST: {(C, E), (D, F), (A, B), (E, F), (B, C)}

− Компоненти: {A, B, C, D, E, F}, {G}

6. Розглядаємо (F, G), вага 5: Вершини F і G в різних компонентах ({A, B, C, D, E, F} та {G}). Додаємо ребро.

− MST: {(C, E), (D, F), (A, B), (E, F), (B, C), (F, G)}

− Компоненти: {A, B, C, D, E, F, G}

На цьому етапі ми додали V – 1 = 7 – 1 = 6 ребер. Усі вершини об’єднані в один компонент. Алгоритм завершує роботу.

7. Розглядаємо (A, D), вага 6: Вершини A і D вже в одному компоненті. Додавання цього ребра утворить цикл A-B-C-E-F-D-A. Відкидаємо ребро.

8. Розглядаємо (C, D), вага 7: Вершини C і D вже в одному компоненті. Додавання утворить цикл C-E-F-D-C. Відкидаємо ребро.

9. Розглядаємо (B, E), вага 8: Вершини B і E вже в одному компоненті. Додавання утворить цикл B-C-E-B. Відкидаємо ребро.

*Результат.* Мінімальне кістякове дерево складається з ребер {(C, E), (D, F), (A, B), (E, F), (B, C), (F, G)}. Загальна вага: 2 + 3 + 4 + 4 + 5 + 5 = 23.

Ключовим питанням під час реалізації алгоритму є ефективна перевірка того, чи утворює нове ребро цикл. Додавання ребра (u, v) створює цикл тоді і тільки тоді, коли вершини u та v вже містяться в одній компоненті зв’язності. Наївна перевірка цього за допомогою DFS або BFS на кожному кроці була б надто повільною (складність O(V + E) для кожного ребра). Тому для ефективного відстеження компонент зв’язності використовується спеціалізована структура даних.

Псевдокод алгоритму Крускала:

```
function Kruskal(Graph G):
  MST = Ø
  // Ініціалізувати V множин, кожна з яких містить одну вершину
  for each vertex v in G.V:
    MakeSet(v)

  // Відсортувати всі ребра за зростанням ваги
  Sort G.E into non-decreasing order by weight w

  // Ітерувати по відсортованих ребрах
  for each edge (u, v) in the sorted list:
    // Якщо u та v знаходяться в різних компонентах, додавання ребра не створить цикл
    if Find(u)!= Find(v):
      Add edge (u, v) to MST
      Union(u, v) // Об'єднати компоненти

  return MST
```

Коректність алгоритму Крускала безпосередньо випливає з властивості розрізу. Розглянемо крок, на якому алгоритм розглядає ребро e = (u, v) і приймає рішення додати його до MST. Це відбувається, коли Find(u)!= Find(v), що означає, що вершини u та v належать до різних компонент зв’язності.

Нехай S − це множина вершин у компоненті, до якої належить u. Тоді v належить до множини V−S. Так, ребро e перетинає розріз (S, V−S). Оскільки алгоритм розглядає ребра в порядку зростання їхньої ваги, а жодне з попередніх ребер не з’єднало компоненти S та V−S, це означає, що e є ребром з найменшою вагою, що перетинає цей розріз. Тобто e є легким ребром для розрізу (S, V−S).

Згідно з властивістю розрізу, таке ребро є «безпечним» для додавання, оскільки воно гарантовано буде частиною деякого MST. Оскільки алгоритм на кожному кроці робить такий безпечний вибір, у кінцевому підсумку він побудує мінімальне кістякове дерево.

## 12.4. Структура даних «Система неперетинних множин» (Union-Find)

Ефективність алгоритму Крускала критично залежить від швидкості виконання операцій перевірки належності до компоненти (Find) та об'єднання компонент (Union). Для вирішення цієї задачі була розроблена спеціалізована структура даних, відома як *система неперетинних множин* (Disjoint Set Union, DSU), або Union-Find.

DSU − це структура даних, яка керує колекцією множин, що не перетинаються (кожен елемент належить рівно одній множині). У контексті алгоритму Крускала кожна така множина представляє одну компоненту зв'язності графа. Структура даних підтримує три основні операції.

1. MakeSet(x) − створює нову множину, що містить єдиний елемент x. В алгоритмі Крускала ця операція виконується на початку для кожної вершини.

2. Find(x) − повертає унікального «представника» (або ідентифікатор) множини, до якої належить елемент x. Ця операція дає змогу перевірити, чи належать два елементи x та y до однієї множини, просто порівнявши результати Find(x) та Find(y).

3. Union(x, y) − об'єднує дві множини, що містять елементи x та y, в одну.

Найефективнішою реалізацією DSU є представлення кожної множини у вигляді дерева, де кожен вузол зберігає посилання лише на свого «батька». Корінь дерева є представником множини і вказує сам на себе.

Наївна реалізація.

Find(x) − починаючи з вузла x, рухається вгору по батьківських посиланнях, доки не досягне кореня.

Union(x, y) − знаходить корені для x та y (нехай це root_x та root_y) і робить один корінь дочірнім елементом іншого (наприклад, parent[root_x] = root_y).

Проблема наївної реалізації полягає в тому, що послідовні операції Union можуть створювати дуже високі, «вироджені» дерева, що нагадують зв'язний список. У такому випадку операція

Find може займати час, пропорційний кількості елементів у множині, тобто O(N) у найгіршому випадку. Для усунення цього недоліку застосовують дві потужні оптимізації.

Оптимізація 1. Об'єднання за рангом / розміром (Union by Rank/Size). Ідея полягає в тому, щоб під час об'єднання двох дерев завжди приєднувати менше дерево до більшого. Це запобігає надмірному зростанню висоти дерев.

*Об'єднання за розміром* (Union by Size). Для кожного кореня зберігається кількість вузлів у його піддереві. Під час об'єднання дерево з меншою кількістю вузлів приєднується до кореня дерева з більшою кількістю вузлів.

*Об'єднання за рангом* (Union by Rank). Для кожного кореня зберігається «ранг» − верхня оцінка висоти його піддерева. Під час об'єднання дерево з меншим рангом приєднується до кореня дерева з більшим рангом. Якщо ранги однакові, одне з дерев приєднується до іншого, а ранг нового кореня збільшується на одиницю.

Ця оптимізація гарантує, що висота будь-якого дерева не перевищуватиме O(logN).

Оптимізація 2. Стиснення шляху (Path Compression). Ця евристика значно прискорює майбутні операції Find. Коли виконується Find(x), ми проходимо шлях від вузла x до його кореня r. Після того, як корінь знайдено, ми можемо пройти цей шлях ще раз і зробити всі вузли на ньому прямими нащадками кореня r. Це «сплющує» дерево, роблячи наступні виклики Find для цих вузлів та їхніх нащадків значно швидшими.

Поєднання обох оптимізацій − об'єднання за рангом / розміром та стиснення шляху − дає надзвичайно ефективну структуру даних. Амортизований час виконання послідовності з m операцій Union або Find на n елементах становить $O(m \cdot \alpha(n))$, де $\alpha(n)$ − *обернена функція Аккермана.*

Функція Аккермана зростає неймовірно швидко, а її обернена функція, $\alpha(n)$, зростає надзвичайно повільно. Для всіх практично мислимих значень n (наприклад, n менше за кількість атомів у видимому Всесвіті) значення $\alpha(n)$ не перевищує 5. Тому на практиці амортизований час виконання кожної операції Union або Find вважається майже константним.

Важливо розуміти, що це амортизована складність. Окремий виклик Find все ще може зайняти O(logN) часу, якщо йому доведеться пройти довгий шлях до кореня. Однак ця «повільна» операція одночасно виконує корисну роботу зі стиснення шляху, що робить багато наступних операцій дуже швидкими (практично O(1)). Висока вартість рідкісних повільних операцій «розподіляється» на велику кількість швидких, що і забезпечує чудову середню продуктивність у довгостроковій перспективі.

## 12.5. Реалізація алгоритму Крускала з використанням Union-Find

Озброївшись ефективною структурою даних DSU, ми можемо реалізувати алгоритм Крускала, який буде швидким на практиці.

Нижче наведено приклади реалізації алгоритму Крускала мовами C++ та Python, що охоплюють повну реалізацію класу DSU з оптимізаціями.

```
C++
#include <iostream>
#include <vector>
#include <algorithm>

// Структура для представлення ребра графа
struct Edge {
    int u, v, weight;
};

// Структура для порівняння ребер за вагою
bool compareEdges(const Edge& a, const Edge& b) {
    return a.weight < b.weight;
}

// Клас для системи неперетинних множин (DSU)
class DSU {
private:
    std::vector<int> parent;
    std::vector<int> rank;
```

```
public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i; // Кожна вершина - корінь власного дерева
        }
    }

    // Знайти представника множини для елемента i (зі стисненням шляху)
    int find(int i) {
        if (parent[i] == i) {
            return i;
        }
        return parent[i] = find(parent[i]); // Рекурсивний виклик зі стисненням
    }

    // Об'єднати множини, що містять i та j (з об'єднанням за рангом)
    void unite(int i, int j) {
        int root_i = find(i);
        int root_j = find(j);

        if (root_i!= root_j) {
            if (rank[root_i] < rank[root_j]) {
                parent[root_i] = root_j;
            } else if (rank[root_i] > rank[root_j]) {
                parent[root_j] = root_i;
            } else {
                parent[root_j] = root_i;
                rank[root_i]++;
            }
        }
    }
};

// Функція для знаходження MST за алгоритмом Крускала
std::vector<Edge> kruskalMST(int V, std::vector<Edge>& edges) {
    // 1. Сортуємо ребра за зростанням ваги
    std::sort(edges.begin(), edges.end(), compareEdges);
```

```
    DSU dsu(V);
    std::vector<Edge> mst;
    int edge_count = 0;

    // 2. Ітеруємо по відсортованих ребрах
    for (const auto& edge : edges) {
        if (dsu.find(edge.u)!= dsu.find(edge.v)) {
            mst.push_back(edge);
            dsu.unite(edge.u, edge.v);
            edge_count++;
            if (edge_count == V - 1) {
                break;
            }
        }
    }
    return mst;
}

int main() {
    int V = 7;
    std::vector<Edge> edges = {
        {0, 1, 4}, {0, 3, 6}, {1, 2, 5}, {1, 4, 8},
        {2, 3, 7}, {2, 4, 2}, {3, 5, 3}, {4, 5, 4},
        {4, 6, 5}, {5, 6, 5}
    };

    std::vector<Edge> result = kruskalMST(V, edges);
    int total_weight = 0;
    std::cout << "Ребра в MST:" << std::endl;
    for (const auto& edge : result) {
        std::cout << edge.u << " - " << edge.v << " (вага " << edge.weight
<< ")" << std::endl;
        total_weight += edge.weight;
    }
    std::cout << "Загальна вага MST: " << total_weight << std::endl;

    return 0;
}
```

```
Python
# Клас для системи неперетинних множин (DSU)
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank =  * n

    def find(self, i):
        if self.parent[i] == i:
            return i
        self.parent[i] = self.find(self.parent[i])  # Стиснення шляху
        return self.parent[i]

    def union(self, i, j):
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i!= root_j:
            if self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            elif self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            else:
                self.parent[root_j] = root_i
                self.rank[root_i] += 1

# Функція для знаходження MST за алгоритмом Крускала
def kruskal_mst(V, edges):
    # edges - це список кортежів (вага, u, v)
    # 1. Сортуємо ребра за зростанням ваги
    edges.sort()

    dsu = DSU(V)
    mst =
    edge_count = 0

    # 2. Ітеруємо по відсортованих ребрах
    for weight, u, v in edges:
        if dsu.find(u)!= dsu.find(v):
            mst.append((u, v, weight))
            dsu.union(u, v)
            edge_count += 1
            if edge_count == V - 1:
                break
```

```
    return mst

if __name__ == '__main__':
    V = 7
    # Список ребер: (вага, u, v)
    edges = [
        (4, 0, 1), (6, 0, 3), (5, 1, 2), (8, 1, 4),
        (7, 2, 3), (2, 2, 4), (3, 3, 5), (4, 4, 5),
        (5, 4, 6), (5, 5, 6)
    ]

    result = kruskal_mst(V, edges)
    total_weight = 0
    print("Ребра в MST:")
    for u, v, weight in result:
        print(f"{u} - {v} (вага {weight})")
        total_weight += weight
    print(f"Загальна вага MST: {total_weight}")
```

Часова складність.

1. Сортування E ребер займає O(ElogE) часу.

2. Ініціалізація структури DSU для V вершин займає O(V) часу.

3. Цикл по ребрах виконується до E разів. Усередині циклу виконуються дві операції Find і, можливо, одна операція Union. Завдяки оптимізаціям, сукупний час для цих операцій становить $O(E \cdot \alpha(V))$.

Загальна часова складність алгоритму: $O(ElogE + V + E \cdot \alpha(V))$. Оскільки функція $\alpha(V)$ зростає дуже повільно, а в зв'язному графі $E \geq V-1$, домінуючим членом є сортування ребер. Так, фінальна складність алгоритму Крускала становить O(ElogE). Оскільки logE асимптотично еквівалентне O(logV) для більшості типів графів, складність часто записують як O(ElogV).

Просторова складність. Для зберігання списку ребер потрібно O(E) пам'яті. Для структури DSU (масиви parent та rank) потрібно O(V) пам'яті. Отже, загальна просторова складність становить O(V + E).

## 12.6. Алгоритм Пріма

Алгоритм Пріма, розроблений 1957 року Робертом Прімом (і незалежно від нього Войтехом Ярніком 1930 року), є ще одним класичним жадібним алгоритмом для знаходження MST. На відміну від Крускала, який працює з ребрами та лісом дерев, алгоритм Пріма «вирощує» єдине дерево, починаючи з довільної вершини.

Стратегія алгоритму Пріма полягає в такому:

1. Почати з довільної вершини, яка утворює початкове дерево.
2. На кожному кроці знаходити найлегше ребро, яке з'єднує вершину, що вже міститься в дереві, з вершиною, якої в дереві ще немає.
3. Додавати це ребро та нову вершину до дерева.
4. Повторювати крок 2 і 3 доти, доки всі вершини графа не будуть додані до дерева.

Цей процес безпосередньо реалізує властивість розрізу. На кожному кроці множина вершин, які вже додані до дерева, утворює одну частину розрізу (S), а всі інші вершини − іншу (V−S). Алгоритм Пріма знаходить і додає саме те легке ребро, що перетинає цей розріз, гарантуючи у такий спосіб коректність кожного свого кроку.

Використаємо той самий граф, що й для алгоритму Крускала, щоб порівняти підходи.

Крок 1. Вибір стартової вершини. Оберемо довільну вершину, наприклад, A.

− MST-вершини: {A}

− Ребра MST: {}

Крок 2. Розглянемо всі ребра, інцидентні вершинам з MST (поки що тільки A): (A, B) з вагою 4 та (A, D) з вагою 6. Обираємо найлегше − (A, B).

− MST-вершини: {A, B}

− Ребра MST: {(A, B)}

Крок 3. Тепер розглядаємо ребра, що виходять з {A, B} до вершин, яких ще немає в MST: (A, D) вага 6, (B, C) вага 5, (B, E) вага 8. Найлегше з них − (B, C).

− MST-вершини: {A, B, C}

− Ребра MST: {(A, B), (B, C)}

Крок 4. Кандидати: (A, D) вага 6, (B, E) вага 8, (C, E) вага 2, (C, D) вага 7. Найлегше − (C, E).

− MST-вершини: {A, B, C, E}

− Ребра MST: {(A, B), (B, C), (C, E)}

Крок 5. Кандидати: (A, D) вага 6, (B, E) − вершина E вже в MST, ігноруємо, (C, D) вага 7, (E, F) вага 4. Найлегше − (E, F).

− MST-вершини: {A, B, C, E, F}

− Ребра MST: {(A, B), (B, C), (C, E), (E, F)}

Крок 6. Кандидати: (A, D) вага 6, (C, D) вага 7, (F, D) вага 3, (F, G) вага 5. Найлегше − (F, D).

− MST-вершини: {A, B, C, E, F, D}

− Ребра MST: {(A, B), (B, C), (C, E), (E, F), (F, D)}

Крок 7. Кандидати: (A, D) − D вже в MST, (C, D) − D вже в MST, (F, G) вага 5. Єдиний кандидат − (F, G).

− MST-вершини: {A, B, C, E, F, D, G}

− Ребра MST: {(A, B), (B, C), (C, E), (E, F), (F, D), (F, G)}

Усі 7 вершин додано до дерева. Алгоритм завершено.

Результат. Загальна вага: 4 + 5 + 2 + 4 + 3 + 5 = 23. Зверніть увагу, що набір ребер відрізняється від того, що ми отримали в алгоритмі Крускала, але загальна вага є такою самою. Це ілюструє можливість існування кількох різних MST, якщо в графі є ребра з однаковою вагою.

Існує дві основні стратегії реалізації алгоритму Пріма, вибір між якими залежить від щільності графа.

Реалізація з використанням масиву (для матриці суміжності).

На кожному з V − 1 кроків нам потрібно знайти найлегше ребро, що з’єднує вже побудоване дерево з рештою графа. Для цього можна підтримувати масив key[v], що зберігає вагу найлегшого ребра від вершини v до дерева. На кожному кроці ми лінійно скануємо всі V вершин, щоб знайти ту, що не в дереві і має мінімальне значення key. Потім оновлюємо значення key для її сусідів.

Часова складність.V ітерацій, кожна з яких вимагає O(V) для пошуку мінімуму та оновлення. Загальна складність − $O(V^2)$.

Застосування. Цей підхід є ефективним для щільних графів, де кількість ребер E наближається до $V^2$.

Реалізація з пріоритетною чергою (min-heap) (для списків суміжності).

Для ефективнішого пошуку найлегшого ребра можна використати пріоритетну чергу (зазвичай реалізовану як бінарна купа), яка зберігатиме вершини, що ще не в дереві. Пріоритетом кожної вершини буде вага найлегшого ребра, що з'єднує її з деревом.

1. На кожному кроці ми витягуємо вершину з мінімальним пріоритетом з черги (операція ExtractMin, складність O(logV)).

2. Для всіх її сусідів, якщо ми знаходимо дешевший спосіб дістатися до них, ми оновлюємо їхній пріоритет у черзі (операція DecreaseKey, складність O(logV)).

Часова складність. Всього виконується V операцій ExtractMin та до E операцій DecreaseKey. Загальна складність − O(VlogV + ElogV), що для зв'язного графа ($E \geq V - 1$) спрощується до O(ElogV).

Застосування. Цей підхід є значно ефективнішим для розріджених графів, де E набагато менше за $V^2$.

Структура алгоритму Пріма з пріоритетною чергою дуже нагадує алгоритм Дейкстри для пошуку найкоротших шляхів. Обидва алгоритми поступово «захоплюють» вершини, використовуючи пріоритетну чергу для вибору наступної. Однак між ними є ключова відмінність у критерії вибору:

1. Дейкстра. Пріоритет вершини − це сумарна вага шляху від початкової вершини (d[v] = d[u] + w(u,v)).

2. Прім. Пріоритет вершини − це вага лише одного ребра, що з'єднує її з уже побудованим деревом (key[v] = w(u,v)).

Ця, на перший погляд, незначна різниця в «жадібному» критерії призводить до розв'язання абсолютно різних задач: пошуку дерева найкоротших шляхів та мінімального кістякового дерева відповідно.

```
C++
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

// Пара для пріоритетної черги: {вага, вершина}
using iPair = pair<int, int>;

void primMST(int V, const vector<vector<iPair>>& adj) {
    // Пріоритетна черга для зберігання вершин, які є кандидатами на
додавання
    // greater<iPair> робить її min-heap
    priority_queue<iPair, vector<iPair>, greater<iPair>> pq;

    int src = 0; // Починаємо з вершини 0

    // Вектори для зберігання ключів, батьків та відвіданих вершин
    vector<int> key(V, numeric_limits<int>::max());
    vector<int> parent(V, -1);
    vector<bool> inMST(V, false);

    // Починаємо з початкової вершини
    pq.push({0, src});
    key[src] = 0;

    int total_weight = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        // Якщо вершина вже в MST, пропускаємо її
        if (inMST[u]) {
            continue;
        }

        inMST[u] = true;
        total_weight += key[u];
```

```
            // Ітеруємо по всіх сусідах u
            for (auto const& [v, weight] : adj[u]) {
                // Якщо v ще не в MST і вага ребра (u, v) менша за поточний
ключ v
                if (!inMST[v] && key[v] > weight) {
                    key[v] = weight;
                    pq.push({key[v], v});
                    parent[v] = u;
                }
            }
        }

        cout << "Ребра в MST (алгоритм Пріма):" << endl;
        for (int i = 1; i < V; ++i) {
            cout << parent[i] << " - " << i << endl;
        }
        cout << "Загальна вага MST: " << total_weight << endl;
    }

    int main() {
        int V = 7;
        vector<vector<iPair>> adj(V);

        auto addEdge = [&](int u, int v, int w) {
            adj[u].push_back({v, w});
            adj[v].push_back({u, w});
        };

        addEdge(0, 1, 4); addEdge(0, 3, 6); addEdge(1, 2, 5);
        addEdge(1, 4, 8); addEdge(2, 3, 7); addEdge(2, 4, 2);
        addEdge(3, 5, 3); addEdge(4, 5, 4); addEdge(4, 6, 5);
        addEdge(5, 6, 5);

        primMST(V, adj);

        return 0;
    }
```

```
Python
import heapq

def prim_mst(V, adj):
    # Пріоритетна черга (min-heap)
    pq = [(0, 0)]  # (вага, вершина)

    # Масиви для ключів, батьків та відвіданих вершин
    key = [float('inf')] * V
    parent = [-1] * V
    in_mst = [False] * V

    key = 0
    total_weight = 0

    while pq:
        weight, u = heapq.heappop(pq)

        if in_mst[u]:
            continue

        in_mst[u] = True
        total_weight += weight

        for v, w in adj[u]:
            if not in_mst[v] and key[v] > w:
                key[v] = w
                heapq.heappush(pq, (key[v], v))
                parent[v] = u

    print("Ребра в MST (алгоритм Пріма):")
    for i in range(1, V):
        print(f"{parent[i]} - {i}")
    print(f"Загальна вага MST: {total_weight}")

if __name__ == '__main__':
    V = 7
    adj = [ for _ in range(V)]

    def add_edge(u, v, w):
        adj[u].append((v, w))
        adj[v].append((u, w))
```

```
add_edge(0, 1, 4); add_edge(0, 3, 6); add_edge(1, 2, 5)
add_edge(1, 4, 8); add_edge(2, 3, 7); add_edge(2, 4, 2)
add_edge(3, 5, 3); add_edge(4, 5, 4); add_edge(4, 6, 5)
add_edge(5, 6, 5)

prim_mst(V, adj)
```

## 12.7. Підсумки та порівняльний аналіз

Алгоритми Крускала та Пріма є двома фундаментальними жадібними підходами до розв'язання задачі пошуку мінімального кістякового дерева. Обидва гарантують знаходження оптимального розв'язку, спираючись на властивість розрізу, але роблять це різними шляхами.

1. *Алгоритм Крускала* має «глобальний» погляд: він сортує всі ребра графа й послідовно обирає найдешевші з них, які не створюють циклів, поступово об'єднуючи розрізнені компоненти в єдине дерево.

2. *Алгоритм Пріма* має «локальний» погляд: він починає з однієї вершини і на кожному кроці розширює єдине дерево, що будується, додаючи найдешевше ребро, яке веде до ще не відвіданої вершини.

Вибір між ними часто залежить від характеристик вхідного графа, зокрема від його щільності.

Рекомендації щодо вибору. Алгоритм Крускала часто є простішим в реалізації і є чудовим вибором для розріджених графів, де вартість сортування ребер є прийнятною.

Алгоритм Пріма з пріоритетною чергою також добре працює на розріджених графах. Його реалізація на основі матриці суміжності є асимптотично швидшою за алгоритм Крускала на дуже щільних графах.

Таблиця 12.1 – Порівняльна таблиця алгоритмів

| Характеристика | Алгоритм Крускала | Алгоритм Пріма |
|---|---|---|
| Основна ідея | Обирає ребра з найменшою вагою, що не утворюють циклу. Працює з лісом дерев | Вирощує одне дерево, додаючи найближчу вершину |
| Ключова структура даних | Система неперетинних множин (Union-Find) для детекції циклів | Пріоритетна черга (min-heap) для швидкого пошуку найлегшого ребра до нової вершини |
| Часова складність | O(ElogE) або O(ElogV) | O(ElogV) з пріоритетною чергою; $O(V^2)$ з матрицею суміжності |
| Ефективність (розріджені графи, $E \approx V$) | Дуже ефективний | Дуже ефективний (з пріоритетною чергою) |
| Ефективність (щільні графи, $E \approx V^2$) | Менш ефективний ($O(V^2 \log V)$) | Дуже ефективний ($O(V^2)$ з матрицею суміжності, що є асимптотично кращим) |
| Робота з незв’язними графами | Так, знаходить мінімальний кістяковий ліс (MST для кожного компонента зв’язності) | Ні, працює тільки для зв’язного графа. Потребує запуску для кожного компонента окремо |

## Контрольні питання

1. Що таке кістякове дерево? Скільки ребер воно має для графу з V вершинами?

2. Сформулюйте властивість розрізу. Поясніть, чому вона є ключовою для доведення коректності жадібних MST алгоритмів.

3. Сформулюйте властивість циклу. Як вона пов’язана з властивістю розрізу?

4. Чи завжди мінімальне кістякове дерево є унікальним? Поясніть, за яких умов існує єдине MST, а за яких − декілька.

5. Опишіть основну ідею алгоритму Крускала. Яку ключову підзадачу вирішує структура даних Union-Find у цьому алгоритмі?

6. Опишіть основну ідею алгоритму Пріма. Як цей алгоритм на кожному кроці неявно використовує властивість розрізу?

7. Порівняйте часову складність алгоритмів Крускала та Пріма (в обох реалізаціях) для розріджених та щільних графів.

8. Чи будуть коректно працювати алгоритми Пріма та Крускала, якщо в графі є ребра з від'ємною вагою? Обґрунтуйте свою відповідь.

9. Як можна модифікувати один з розглянутих алгоритмів для пошуку максимальногокістякового дерева (тобто кістякового дерева з максимальною сумою ваг ребер)?

10. У чому полягає принципова різниця між мінімальним кістяковим деревом та деревом найкоротших шляхів, яке будує алгоритм Дейкстри?

## Задачі для самостійного опрацювання

1. Трасування алгоритмів. Для графа, наведеного нижче, виконайте покроково:

a) Алгоритм Крускала. Запишіть послідовність ребер, що розглядаються, та позначте, які з них додаються до MST.

b) Алгоритм Пріма, починаючи з вершини A. Запишіть послідовність доданих ребер.

c) Обчисліть фінальну вагу MST.

2. Теоретична задача:

Доведіть, що якщо до ваги кожного ребра в графі додати одну й ту саму додатну константу C, то множина ребер, що утворюють мінімальне кістякове дерево, не зміниться. Чи зміниться вона, якщо помножити вагу кожного ребра на $C > 0$?

3. Алгоритмічна варіація. Вам дано зв'язний граф G та його MST T. До графа G додають нове ребро (u, v) з вагою w. Опишіть алгоритм, який знаходить MST для оновленого графа за час O(V), не запускаючи алгоритми Пріма або Крускала з нуля. (Підказка: додавання нового ребра до T створює цикл. Використайте властивість циклу.)

4. Програмування (Базовий рівень). Реалізуйте алгоритм Крускала. На вхід програма отримує кількість вершин V, кількість ребер E, а потім E рядків, кожен з яких описує ребро у форматі u v w

(дві вершини та вага). Програма повинна вивести єдине число − сумарну вагу мінімального кістякового дерева.

5. Програмування (Середній рівень). Проєктування мережі Інтернет-провайдера. Вам дана карта з N будинками та можливими місцями прокладання оптоволоконного кабелю між ними. Деякі з'єднання є стандартними (з певною вартістю), а деякі – «преміальними» (з вищою вартістю, але більшою надійністю). Вам потрібно з'єднати всі будинки в єдину мережу з мінімальною загальною вартістю, але з умовою, що в мережі має бути рівно K преміальних з'єднань. Якщо це неможливо, виведіть -1.

6. Програмування (Просунутий рівень). Задача про кластеризацію. Вам дано N точок на площині та ціле число K. Відстань між будь-якими двома точками є евклідовою. Побудуйте повний граф на цих точках. Ваша задача − розділити точки на K кластерів (неперетинних підмножин) так, щоб мінімальна відстань між точками з різних кластерів була максимальною. Виведіть цю максимальну мінімальну відстань. (Підказка: ця задача тісно пов'язана з алгоритмом Крускала. Подумайте, які ребра в MST з'єднують найбільш віддалені групи точок).

# ТЕМА 13
# МЕРЕЖЕВІ ПОТОКИ ТА ТОПОЛОГІЧНЕ СОРТУВАННЯ

## 13.1. Вступ до орієнтованих графів та лінійне впорядкування

Топологічне сортування є одним із ключових алгоритмів для роботи з орієнтованими ациклічними графами (DAG), що знаходить широке застосування в плануванні завдань, аналізі залежностей та багатьох інших галузях комп'ютерних наук. Воно являє собою лінійне впорядкування вершин $V = \{v_1, v_2, ..., v_{|V|}\}$ графа у такий спосіб, що для кожного орієнтованого ребра *(u, v) in E* вершина *u* завжди розміщена раніше за вершину *v* в цьому порядку. Концептуально, це еквівалентно розташуванню всіх вершин на одній прямій так, що всі ребра в графі спрямовані виключно зліва направо.

*Топологічне сортування орієнтованого графа* − це лінійне впорядкування його вершин у такий спосіб, що для кожного орієнтованого ребра з вершини *u* у вершину *v* вершина *u* йде раніше за вершину *v* в цьому впорядкуванні. Іншими словами, це таке розташування вершин на прямій, за якого всі ребра графа спрямовані зліва направо.

Ключовою вимогою для існування топологічного сортування є ациклічність графа. Якщо граф містить хоча б один цикл (наприклад, A→B→A), то створити таке впорядкування неможливо. Наявність циклу призводить до логічного протиріччя: вершина A повинна передувати вершині B, а вершина B − вершині A, що є неможливим. Так, топологічне сортування існує тоді й лише тоді, коли граф є орієнтованим ациклічним графом (DAG).

Для одного й того самого графа може існувати декілька коректних топологічних сортувань. Це трапляється тоді, коли на певному етапі алгоритму існує кілька вершин, які можуть бути обрані наступними (наприклад, кілька вершин без вхідних ребер).

Важливо зазначити, що для більшості DAGів, які не є простими ланцюгами, існує декілька коректних топологічних сортувань. Ця неоднозначність виникає, коли на певному етапі алгоритму доступні для вибору кілька вершин, які не мають вхідних ребер (або їхні передумови вже виконані). Цей факт свідчить про гнучкість системи залежностей, де незалежні завдання можуть бути виконані в будь-якій послідовності, що дає простір для оптимізації за вторинними критеріями, наприклад, для максимізації паралельної обробки.

Топологічне сортування має широке застосування в комп'ютерних науках та інженерії для моделювання й вирішення проблем залежностей.

Найбільш поширене застосування − це планування завдань. У системах збирання проєктів (наприклад, Make або Maven) або в операційних системах управління пакетами сортування визначає коректну послідовність компіляції або встановлення залежних модулів. Якщо завдання X залежить від завдання Y, то Y має бути виконано раніше за X.

Інші приклади містять розклад навчальних курсів, де потрібно визначити послідовність вивчення, враховуючи передумови (prerequisites), або розв'язання залежностей під час завантаження бібліотек чи модулів у програмних системах. Оскільки DAG є фундаментальною структурою для моделювання причинно-наслідкових зв'язків, його сортування є алгоритмічною основою для будь-якої системи, що керує послідовністю виконання.

## 13.2. Алгоритми топологічного сортування: DFS та Кана

Для знаходження топологічного сортування розроблено два основні алгоритмічні підходи: один базується на DFS, а інший − на BFS (алгоритм Кана). Обидва методи досягають оптимальної лінійної часової складності O(|V| + |E|).

DFS-підхід використовує властивості часу завершення обходу вершин. Основна ідея полягає в тому, що топологічне сортування є зворотним до порядку завершення обходу (post-order traversal).

Під час виконання DFS вершина вважається «завершеною» лише тоді, коли рекурсивний виклик для неї та всіх її нащадків закінчився. Вершина, яка завершується останньою, є тією, що не має залежностей від інших невідвіданих вершин, і, отже, вона повинна бути першою в топологічному порядку. Цей підхід будує сортування «від кінця до початку».

У реалізації DFS-алгоритм неявно використовує стек рекурсії. Вершина додається до списку сортування (на початок або на стек) лише після того, як були оброблені всі її залежності. Виявлення циклів у DFS відбувається, якщо алгоритм знаходить зворотне ребро до вершини, яка вже міститься в поточному стеку рекурсії (тобто вона перебуває у стані обробки).

Псевдокод:

```
function TopologicalSort_DFS(graph):
  L = new empty List // Список для відсортованих вершин
  visited = new Set  // Множина відвіданих вершин

  for each vertex u in graph:
    if u is not in visited:
      DFS_Visit(u, visited, L)

  return L

function DFS_Visit(u, visited, L):
  add u to visited

  for each neighbor v of u:
    if v is not in visited:
      DFS_Visit(v, visited, L)

  prepend u to L // Додати u на початок списку
```

Приклад реалізації на Python:

```
Python
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def _topological_sort_util(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self._topological_sort_util(i, visited, stack)
        stack.insert(0, v)

    def topological_sort_dfs(self):
        visited = [False] * self.V
        stack =

        for i in range(self.V):
            if not visited[i]:
                self._topological_sort_util(i, visited, stack)

        return stack

# Приклад використання:
g = Graph(6)
g.add_edge(5, 2)
g.add_edge(5, 0)
g.add_edge(4, 0)
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)

sorted_order = g.topological_sort_dfs()
print("Топологічне сортування (DFS):", sorted_order)
# Можливий результат: 10
```

Алгоритм Кана використовує ітеративний підхід, заснований на підрахунку вхідних ступенів (*in_degree*) вершин. Його основна ідея − імітувати процес вирішення залежностей, фокусуючись на питанні: «що можна зробити зараз?».

На початку обчислюються вхідні ступені для всіх вершин. Вершини з *in_degree* = 0 додаються до черги Q як початкові завдання. На кожному кроці алгоритм витягує вершину *u* з черги, додає її до сортування та концептуально «видаляє» всі вихідні з неї ребра, зменшуючи *in_degree* для всіх її сусідів *v*. Якщо вхідний ступінь сусіда *v* стає 0, це означає, що всі його передумови виконані, і *v* додається до черги. Цей метод будує сортування «від початку до кінця».

Перевагою алгоритму Кана є явне виявлення циклу: якщо кількість відсортованих вершин наприкінці менша за загальну кількість |V|, то решта вершин утворюють цикл або залежать від нього, і сортування неможливе.

Обидва алгоритми, DFS-базований та Кан-алгоритм, мають лінійну часову складність O(|V| + |E|), що є оптимальним для будь-якого алгоритму, який повинен опрацювати всі вершини та ребра.

Незважаючи на ідентичну асимптотичну складність, їхня архітектура має важливі відмінності. Кан-алгоритм, завдяки фокусу на вершинах з in_degree = 0, краще підходить для паралельних обчислень, оскільки кілька незалежних вершин можуть бути оброблені одночасно. Крім того, алгоритм Кана може бути адаптований для знаходження лексикографічно найменшого сортування, якщо замінити звичайну чергу на чергу з пріоритетом (min-heap), що дає можливість оптимізувати сортування за додатковими критеріями.

Псевдокод:

```
function TopologicalSort_Kahn(graph):
  L = new empty List // Список для відсортованих вершин
  in_degree = array of size |V|, initialized to 0

  // Обчислити вхідні ступені для всіх вершин
  for each vertex u in graph:
    for each neighbor v of u:
      in_degree[v] = in_degree[v] + 1

  Q = new Queue
  // Додати всі вершини з вхідним ступенем 0 до черги
  for each vertex i from 0 to |V|-1:
    if in_degree[i] == 0:
      Q.enqueue(i)

  count = 0 // Лічильник відсортованих вершин

  while Q is not empty:
    u = Q.dequeue()
    add u to L
    count = count + 1

    // Зменшити вхідний ступінь для всіх сусідів
    for each neighbor v of u:
      in_degree[v] = in_degree[v] - 1
      if in_degree[v] == 0:
        Q.enqueue(v)

  if count!= |V|:
    // Граф містить цикл
    return error "Graph has a cycle"
  else:
    return L
```

Приклад реалізації на Python:

```
Python
from collections import defaultdict, deque

class GraphKahn:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def topological_sort_kahn(self):
        in_degree =  * self.V
        for i in self.graph:
            for j in self.graph[i]:
                in_degree[j] += 1

        queue = deque([i for i in range(self.V) if in_degree[i] == 0])

        sorted_order =

        while queue:
            u = queue.popleft()
            sorted_order.append(u)

            for i in self.graph[u]:
                in_degree[i] -= 1
                if in_degree[i] == 0:
                    queue.append(i)

        if len(sorted_order) == self.V:
            return sorted_order
        else:
            return "Граф містить цикл"
# Приклад використання:
g_kahn = GraphKahn(6)
g_kahn.add_edge(5, 2)
g_kahn.add_edge(5, 0)
g_kahn.add_edge(4, 0)
```

```
g_kahn.add_edge(4, 1)
g_kahn.add_edge(2, 3)
g_kahn.add_edge(3, 1)

sorted_order_kahn = g_kahn.topological_sort_kahn()
print("Топологічне сортування (Kahn):", sorted_order_kahn)
# Можливий результат:
```

Таблиця 13.1 – Порівняння алгоритмів топологічного сортування

| Характеристика | Алгоритм на основі DFS | Алгоритм Кана |
|---|---|---|
| Основний принцип | Обернений порядок завершення вершин (post-order) | Ітеративна обробка вершин із вхідним ступенем 0 |
| Метод обходу | Пошук у глибину (DFS) | Пошук у ширину (BFS) |
| Основна структура даних | Стек рекурсії (неявний) | Черга (явна) |
| Виявлення циклу | Знаходження зворотного ребра до вершини в поточному стеку рекурсії | Кінцева кількість відсортованих вершин менша за загальну кількість вершин |
| Порядок виводу | Один з багатьох можливих топологічних порядків | Може бути адаптований для знаходження лексикографічно найменшого порядку |

## 13.3. Структурна декомпозиція: сильно зв’язані компоненти (SCC)

Аналіз сильно зв’язаних компонент (SCC) є необхідною технікою для розуміння внутрішньої структури довільних орієнтованих графів, що містять цикли, даючи можливість розкласти їх на простіші, ациклічні підструктури.

Сильно зв’язаний граф − це орієнтований граф, у якому для будь-якої пари вершин (*u, v*) існує шлях від *u* до *v* і шлях від *v* до *u*.

Сильно зв’язана компонента орієнтованого графа G − це максимально можливий підграф, який сам по собі є сильно зв’язним. SCC є, по суті, наборами вершин, які утворюють замкнуті цикли і можуть бути досягнуті одна з одної.

Якщо кожна SCC графа G «стискається» в одну супер-вершину, а ребра проводяться між цими супер-вершинами, якщо в початковому графі існували ребра між відповідними компонентами, отриманий граф називається графом конденсації. Фундаментальною властивістю графа конденсації є те, що він завжди є орієнтованим ациклічним графом (DAG). Ця властивість дає змогу звести аналіз складного циклічного графа до аналізу простої структури DAG, що лежить в основі багатьох подальших алгоритмів.

Алгоритм Косараджу є елегантним і надійним методом знаходження всіх SCC з лінійною часовою складністю O(|V| + |E|). Він базується на використанні двох послідовних проходів пошуку в глибину (DFS) та концепції транспонованого графа $G^T$, де всі ребра G обернені.

1. Крок 1. Перший DFS на G. Алгоритм виконує DFS на G, щоб отримати порядок завершення обходу вершин. Вершини додаються до стека S у порядку завершення. Цей порядок є зворотним до топологічного сортування графа конденсації.
2. Крок 2. Транспонування. Граф G транспонується в $G^T$.
3. Крок 3. Другий DFS на $G^T$. DFS запускається на $G^T$ у порядку, визначеному стеком S (від вершини, завершеної останньою в G). Кожен новий запуск DFS ідентифікує та збирає рівно одну SCC.

Ключ до коректності полягає в такому: перший прохід DFS на G гарантує, що вершина, з якої ми починаємо другий DFS (вершина на вершині стека), належить до «джерельної» SCC у G. У транспонованому графі $G^T$ ця компонента стає «стічною». Транспоновані ребра гарантують, що пошук, запущений у $G^T$, буде «замкненим» у межах цієї компоненти. Це дає змогу алгоритму ідентифікувати одну компоненту, концептуально «видалити» її і повторити процес для решти графа.

Псевдокод:

```
function Kosaraju_SCC(graph G):
  // Крок 1: Перший DFS на G для отримання порядку завершення
  stack S = new empty Stack
  visited = new Set
  for each vertex u in G:
    if u is not in visited:
      DFS_Fill_Order(G, u, visited, S)

  // Крок 2: Транспонування графа
  GT = transpose(G)

  // Крок 3: Другий DFS на GT для знаходження компонент
  visited.clear()
  SCC_List = new empty List
  while S is not empty:
    u = S.pop()
    if u is not in visited:
      current_SCC = new List
      DFS_Collect_SCC(GT, u, visited, current_SCC)
      add current_SCC to SCC_List

  return SCC_List

function DFS_Fill_Order(G, u, visited, S):
  add u to visited
  for each neighbor v of u in G:
    if v is not in visited:
      DFS_Fill_Order(G, v, visited, S)
  S.push(u)

function DFS_Collect_SCC(GT, u, visited, current_SCC):
  add u to visited
  add u to current_SCC
  for each neighbor v of u in GT:
    if v is not in visited:
      DFS_Collect_SCC(GT, v, visited, current_SCC)
```

Приклад реалізації на Python:

```
Python
from collections import defaultdict

class GraphSCC:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def _dfs1(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if not visited[i]:
                self._dfs1(i, visited, stack)
        stack.append(v)

    def _get_transpose(self):
        gt = GraphSCC(self.V)
        for i in self.graph:
            for j in self.graph[i]:
                gt.add_edge(j, i)
        return gt

    def _dfs2(self, v, visited, component):
        visited[v] = True
        component.append(v)
        for i in self.graph[v]:
            if not visited[i]:
                self._dfs2(i, visited, component)

    def find_sccs(self):
        stack =
        visited = [False] * self.V
        for i in range(self.V):
            if not visited[i]:
                self._dfs1(i, visited, stack)

        gt = self._get_transpose()
```

```
        visited = [False] * self.V
        sccs =
        while stack:
            i = stack.pop()
            if not visited[i]:
                component =
                gt._dfs2(i, visited, component)
                sccs.append(component)
        return sccs

# Приклад використання:
g_scc = GraphSCC(5)
g_scc.add_edge(1, 0)
g_scc.add_edge(0, 2)
g_scc.add_edge(2, 1)
g_scc.add_edge(0, 3)
g_scc.add_edge(3, 4)

scc_list = g_scc.find_sccs()
print("Сильно зв'язані компоненти:", scc_list)
# Результат: [, , ]
```

## 13.4. Фундаментальні поняття мережевих потоків

Теорія мережевих потоків є інструментом дискретної оптимізації, що використовується для моделювання задач, пов’язаних з обмеженою пропускною здатністю.

*Потокова мережа* − це орієнтований граф G = (V, E), що містить джерело s та стік t. Кожне ребро *(u, v) in E* має невід’ємну пропускну здатність(capacity) c(u, v) >= 0.

Потік f − це функція, що присвоює кожному ребру числовий потік, який повинен відповідати трьом умовам:

1. Обмеження пропускної здатності**:** f(u, v) <= c(u, v).
2. Кососиметричність: f(u, v) = − f(v, u) (потік у зворотному напрямку є протилежним).
3. Збереження потоку: чистий потік у будь-яку внутрішню вершину v≠s, t дорівнює нулю ($\sum_{u \in V} f(u, v) = 0$).

Величина потоку $|f|$ − це сумарний потік, що виходить із джерела s: |f| = ($\sum_{u \in V} f(s, v)$. Головна задача − знайти потік із максимально можливою величиною.

Для ітеративного збільшення потоку використовується поняття залишкової мережі $G_f$. Цей граф показує, скільки додаткового потоку $c_f(u, v)$ ще можна пропустити по кожному шляху.

Залишкова пропускна здатність для прямого ребра $(u, v)$ визначається як $c_f(u, v) = c(u, v) - f(u, v)$.

Крім того, $G_f$ містить зворотні ребра $(v, u)$ з пропускною здатністю $c_f(v, u) = f(u, v)$. Ці зворотні ребра є математичним інструментом, що моделює можливість «скасування» існуючого потоку на ребрі $(u, v)$ для його подальшого перенаправлення іншим шляхом, що є критично важливим для виправлення неефективних жадібних виборів, зроблених на ранніх етапах.

*Доповнювальний шлях* − це шлях від s до t у залишковій мережі $G_f$. Його існування свідчить про те, що поточний потік не є максимальним, і його можна збільшити на величину, обмежену мінімальною залишковою пропускною здатністю ребер на цьому шляху (вузьке місце).

## 13.5. Теорема про максимальний потік та мінімальний розріз

Ця теорема, також відома як теорема Форда-Фалкерсона, встановлює глибоку двоїстість між оптимізаційними задачами Max Flow та Min Cut.

Розріз та його пропускна здатність. *s-t розріз* − це поділ множини вершин V на дві підмножини, що не перетинаються, S і T, такий, що s in S і t in T.

*Пропускна здатність розрізу* c(S, T) − це сума пропускних здатностей усіх ребер, що ведуть виключно з S до T: c(S, T) = $\sum_{u \in S, v \in T} c(u, v)$. Величина будь-якого потоку $|f|$ завжди менша або дорівнює пропускній здатності будь-якого розрізу c(S, T).

Теорема. У будь-якій потоковій мережі величина максимального потоку $|f_{max}|$ від джерела s до стоку t дорівнює пропускній здатності мінімального s-t розрізу $c_{min}(S, T)$.

Ця теорема забезпечує потужний критерій оптимальності: потік f є максимальним тоді й лише тоді, коли в залишковій мережі G_f не існує жодного доповнювального шляху від s до t. Двоїстість Max Flow / Min Cut перетворює задачу максимізації пропускної здатності на задачу мінімізації вузького місця, що має значні практичні наслідки. Щойно максимальний потік знайдено, мінімальний розріз може бути ідентифікований як множина S, яка міститьє усі вершини, доступні з s у G_f.

## 13.6. Алгоритм Едмондса-Карпа для знаходження максимального потоку

Алгоритм Едмондса-Карпа є поліноміальною реалізацією загального методу Форда-Фалкерсона, який вирішує проблему повільної збіжності, що виникає за довільного вибору шляхів.

Тоді як загальний метод Форда-Фалкерсона дає змогу збільшувати потік по *будь-якому* доповнювальному шляху, що може призвести до експоненціального часу роботи, Алгоритм Едмондса-Карпа вимагає, щоб на кожному кроці обирався найкоротший доповнювальний шлях за кількістю ребер.

Для знаходження найкоротшого шляху використовується Пошук у ширину (BFS).[1] Ця стратегія забезпечує, що загальна кількість ітерацій (збільшень потоку) є обмеженою, гарантуючи поліноміальний час роботи, на відміну від потенційно експоненціального часу загального методу Форда-Фалкерсона.

Алгоритм працює ітеративно:

1. Ініціалізація: Потік f = 0.[6]
2. Цикл: Поки BFS знаходить шлях P від s до t у залишковій мережі G_f.

3. Знахідка шляху: BFS ідентифікує найкоротший шлях і визначає його вузьке місце m (мінімальна залишкова пропускна здатність на шляху).

4. Збільшення: Потік f збільшується на m. Залишкові ємності оновлюються: c_f(u, v) зменшується, а c_f(v, u) (для зворотного ребра) збільшується на m.[1]

5. Завершення: Алгоритм зупиняється, коли BFS більше не може знайти шляху від s до t.

Псевдокод:

```
function EdmondsKarp(graph G, source s, sink t):
    flow = 0
    while true:
        // Знайти найкоротший шлях в залишковій мережі за допомогою BFS
        m, P = BreadthFirstSearch(G, s, t) // BFS повертає вузьке місце m та батьківський масив P

        if m == 0:
            break // Немає більше доповнювальних шляхів

        flow = flow + m

        // Оновити потік та залишкову мережу вздовж шляху P
        v = t
        while v!= s:
            u = P[v]
            // Оновлення залишкової ємності прямого та зворотного ребра
            residual_capacity(u, v) = residual_capacity(u, v) - m
            residual_capacity(v, u) = residual_capacity(v, u) + m
            v = u

    return flow
```

## 13.7. Аналіз часової складності алгоритму Едмондса-Карпа

Часова складність алгоритму Едмондса-Карпа становить $O(|V||E|^2)$. Це обчислюється на основі двох ключових чинників.

1. Складність однієї ітерації. Пошук найкоротшого доповнювального шляху за допомогою BFS займає час $O(|E|)$, оскільки BFS переглядає всі ребра в залишковій мережі $G_f$, кількість яких не перевищує $2|E|$.

2. Обмеження на кількість ітерацій. Доведено, що загальна кількість збільшень потоку, виконаних цим алгоритмом, не перевищує $O(|V||E|)$.

Загальна часова складність є добутком цих двох величин: $O(N_{aug} \times O(|E|)) = O(|V||E| \times |E|) = O(|V||E|^2)$.

Поліноміальність забезпечується тим, що вибір найкоротшого шляху гарантує: довжина найкоротшого шляху $d(s, t)$ у $G_f$ між збільшеннями потоку є неспадною функцією.

Доведення використовує поняття *критичного ребра* − ребра на шляху, яке насичується і його залишкова ємність стає нульовою. Щоб це ребро *(u, v)* могло знову використовуватися (у зворотному напрямку як *(v, u)*), має бути обраний доповнювальний шлях, який скасовує частину потоку, а це можливо лише тоді, коли відстань від s до його кінцевих вершин у $G_f$ зросла. Оскільки максимальна відстань обмежена $|V|$, кожне з $|E|$ ребер може стати критичним не більше ніж $|V|/2$ разів. Цей факт обмежує загальну кількість ітерацій до $O(|V||E|)$.

Потреба у використанні $O(|V||E|^2)$ алгоритму для Max Flow, на відміну від майже лінійного часу $O(|E| \log |V|)$ для MST, ілюструє, що задача пошуку максимальної пропускної здатності є фундаментально складнішою, ніж задача мінімальної вартості зв’язності.

## 13.8. Алгоритм пошуку A*: оптимальний пошук шляху

Алгоритм A* (A-star) є евристичним алгоритмом пошуку найкоротшого шляху, що є розширенням алгоритму Дейкстри.

A* оцінює кожен шлях, використовуючи оцінну функцію *f(n),* яка визначається як:

*f(n) = g(n) + h(n).*

1. g(n) (точна вартість) − вартість, уже витрачена на шлях від старту до поточної вершини n. Це компонент, успадкований від Дейкстри, що гарантує, що вже пройдений шлях є найкоротшим.

2. h(n) (евристична оцінка) − оцінка вартості шляху від n до цільової вершини (goal). Цей компонент, евристика, є ключовим для спрямування пошуку.

Алгоритм використовує пріоритетну чергу, впорядковану за мінімальним значенням *f(n)*, що дає змогу йому віддавати пріоритет вершинам, які, згідно з евристикою, перебувають на найкоротшому очікуваному шляху до цілі.

Для того, щоб A* гарантував знаходження оптимального шляху, евристична функція *h(n)* повинна відповідати умовам:

1. Допустимість (Admissibility) − обов'язкова умова для гарантії оптимальності. Евристика ніколи не повинна переоцінювати реальну вартість шляху до цілі: h(n) <= cost(n, goal). Якщо евристика є нижньою оцінкою, алгоритм не пропустить оптимальний шлях.

2. Монотонність (Consistency) − сильніша умова. Вона вимагає, щоб для будь-якого ребра *(u, v)* виконувалась нерівність трикутника: *h(u) <= cost(u, v) + h(v)*. Монотонна евристика забезпечує, що *f(n)* не зменшується під час розширення шляху, що дає змогу уникнути повторного розкриття вершин та спрощує обчислення.

У найгіршому випадку (коли евристика неінформативна, h(n) = 0), A* має ту саму складність, що й Дейкстра: O(|E|log |V|) (за умови використання пріоритетної черги). У загальному випадку, часова складність є експоненціальною відносно глибини рішення: $O(b^d)$, де b − коефіцієнт розгалуження, а d − глибина.

Проте, на практиці, завдяки ефективній евристиці, A* значно скорочує простір пошуку порівняно з Дейкстрою, яка завжди обчислює найкоротші шляхи до всіх вершин. A* знаходить найкоротший шлях до конкретної цілі, балансуючи між швидкістю, яку надає *h(n),* і гарантованою коректністю, яку забезпечує *g(n)* та допустимість евристики.

## 13.9. Поглиблений аналіз складності: обернена функція Акермана (α(n))

Аналіз складності деяких високоефективних алгоритмів, таких як алгоритм Крускала для MST, вимагає знання про обернену функцію Акермана α(n), яка є показником продуктивності структури даних «Система неперетинних множин» (DSU).

*Функція Акермана* A(m, n) − це рекурсивна функція, відома своїм надзвичайно швидким, майже вибуховим зростанням. Вона є обчислюваною, але не належить до класу примітивно-рекурсивних функцій. Це означає, що її швидкість росту перевищує швидкість росту будь-якої функції, яку можна визначити за допомогою лише базових операцій рекурсії (наприклад, A(4, 2) вже є астрономічно великим числом).

*Обернена функція Акермана*α(n) визначається як найменше k, для якого A(k, k) >= n.

На відміну від A(m, n), α(n) зростає надзвичайно повільно. Для будь-яких практично мислимих значень n (охоплюючи будь-яку кількість елементів, що можуть бути оброблені сучасними комп'ютерами) значення α(n) не перевищує 5.

Структура даних DSU, яка використовується для ефективної детекції циклів у алгоритмі Крускала, досягає своєї високої продуктивності завдяки двом оптимізаціям: об'єднання за рангом / розміром та стиснення шляху.

Поєднання цих оптимізацій забезпечує, що амортизований час виконання m операцій Find та Union на n елементах становить O(m × α(n)). Це є теоретично найнижчою можливою амортизованою складністю.

Оскільки α(n) на практиці є константою (не більше ніж 5), час виконання операцій DSU є майже константним. У контексті алгоритму Крускала, загальна складність якого O(|E| log |E| + |E| α(|V|)), сортування ребер O(|E| log |E|) залишається домінуючим чинником, що підтверджує успіх оптимізації структури DSU.

## 13.10. Практичне застосування мережевих потоків та графів

Знаходження максимального паросполучення у двочастковому графі G = (L ∪ R, E) можна звести до задачі про максимальний потік.

Мережа будується через додавання джерела s і стоку t. Ребра додаються від s до всіх вершин L з пропускною здатністю 1, від усіх вершин R до t з пропускною здатністю 1, а також між вершинами L і R (від L до R) з пропускною здатністю 1.

Еквівалентність. Величина максимального потоку, знайденого в цій мережі, чисельно дорівнює розміру максимального паросполучення. Завдяки цілочисельності пропускних здатностей, потік по ребрах (L, R) буде 0 або 1, що прямо відображає долучення ребра в паросполучення.

Теорема Max-Flow Min-Cut дає змогу вирішувати проблеми сегментації зображень, де потрібно розділити пікселі на передній план (foreground) та фон (background).

Зображення моделюється як граф, де пікселі − це вершини. Джерело s символізує передній план, а стік t − фон.

1. Термінальні ребра (t-links): З’єднання пікселів з s та t. Їхня пропускна здатність відображає «вартість» або штраф за неправильну класифікацію пікселя на основі його властивостей (наприклад, кольору).

2. Сусідні ребра (n-links): З’єднання сусідніх пікселів. Їхня пропускна здатність моделює «штраф» за розміщення двох схожих сусідніх пікселів у різні сегменти. Висока ємність заохочує гладкі межі сегментації.

Еквівалентність. Мінімальний s-t розріз у цьому графі розділяє пікселі на дві множини (S і T), мінімізуючи сумарний штраф за неправильну класифікацію та негладкість межі. Отже, мінімальний розріз ідентифікує оптимальну сегментацію.

## 13.11. Порівняльний аналіз

Структурні алгоритми (Топологічне сортування, SCC) та оптимізаційні алгоритми (MST, Max Flow) вирішують різні класи проблем, але всі є невід'ємною частиною теорії графів. Тоді як структурні алгоритми демонструють лінійну часову складність O(|V| + |E|), оптимізаційні задачі, які вимагають зваженого вибору (Max Flow) або пошуку глобального оптимуму (MST, A*), мають вищу складність, хоча MST завдяки DSU є надзвичайно ефективним.

Таблиця 13.2 – Зведена таблиця алгоритмів графової теорії

| Алгоритм | Задача | Граф |
|---|---|---|
| Крускала | Мінімальне кістякове дерево (MST) | Неорієнтований, зважений |
| Пріма | Мінімальне кістякове дерево (MST) | Неорієнтований, зважений |
| Топологічне сортування | Лінійне впорядкування | Орієнтований, ациклічний (DAG) |
| Косараджу | Сильно зв'язані компоненти (SCC) | Орієнтований |
| Едмондса-Карпа | Максимальний потік | Орієнтований, зважений (ємність) |
| A* | Оптимальний пошук шляху | Зважений (додатні ваги) |

## Контрольні питання

1. Чому топологічне сортування неможливе для графа, що містить цикл? Поясніть на прикладі.

2. Порівняйте механізми виявлення циклів в алгоритмі топологічного сортування на основі DFS та в алгоритмі Кана.

3. Яку роль відіграє транспонований граф в алгоритмі Косараджу для знаходження сильно зв’язаних компонент? Поясніть, чому другий DFS гарантовано не виходить за межі однієї SCC.

4. Дайте визначення залишкової мережі та поясніть значення зворотних ребер у ній.

5. Сформулюйте теорему про максимальний потік та мінімальний розріз. Поясніть її інтуїтивний зміст, використовуючи поняття вузького місця.

6. У чому полягає ключова відмінність між методом Форда-Фалкерсона та алгоритмом Едмондса-Карпа, і чому ця відмінність є критичною для часової складності?

7. Опишіть процес зведення задачі про максимальне паросполучення у двочастковому графі до задачі про максимальний потік.

8. Як визначаються пропускні здатності термінальних (t-links) та сусідніх (n-links) ребер під час побудови графа для сегментації зображень? Який штраф вони моделюють?

9. Поясніть різницю між функціями g(n) та h(n) в алгоритмі A*. Що означає, що евристична функція є «допустимою»?

10. Які дві оптимізації дають змогу досягти майже константної амортизованої складності O(\alpha(n)) у структурі даних DSU?

11. Поясніть, чому функція Акермана не є примітивно-рекурсивною, та наведіть приклад її застосування в аналізі складності.

12. Чи може граф мати єдине унікальне топологічне сортування? Якщо так, наведіть приклад такого графа.

## Задачі для самостійного опрацювання

1. Програмування (Топологічне сортування): Реалізуйте алгоритм Кана так, щоб він завжди знаходив лексикографічно найменше топологічне сортування, використовуючи чергу з пріоритетом.

2. Теоретична задача (SCC): Доведіть, що граф конденсації будь-якого орієнтованого графа є ациклічним.

3. Зведення (Max Flow): Спроєктуйте потокову мережу для розв’язання задачі про максимальне призначення робітників на завдання (кожне завдання та кожен робітник використовуються не більше одного разу).

4. Трасування (Max Flow): Для заданої потокової мережі вручну прослідкуйте виконання алгоритму Едмондса-Карпа, показуючи залишкову мережу після кожної ітерації.

5. Алгоритмічна варіація (A*): Реалізуйте алгоритм A* для знаходження найкоротшого шляху на сітці, де вартість переходу залежить від типу місцевості, використовуючи Манхеттенську відстань як допустиму евристику.

6. Теоретична задача (A*): Доведіть, що якщо евристика h(n) в алгоритмі A* є монотонною, то послідовність значень f(n) для вершин, що витягуються з черги з пріоритетом, є неспадною.

7. Програмування (DSU/Акерман): Реалізуйте структуру даних DSU з оптимізаціями об’єднання за рангом та стиснення шляху.

# ТЕМА 14
# ПРОСТОРОВІ ТА РЯДКОВІ СТРУКТУРИ ДАНИХ

У сучасному світі обчислювальної техніки ефективна обробка великих масивів даних є фундаментальною задачею. Часто ці дані мають певну структуру − або просторову, де важливе взаємне розташування елементів, або послідовну, як у випадку текстових рядків.

## 14.1. Просторові структури даних для запитів на відрізках

Часто в програмуванні виникає задача обробки запитів на відрізках (або інтервалах) масиву. Класична постановка задачі виглядає так: дано масив A розміром N та послідовність запитів двох типів.

1. *Запит на відрізку (range query)* − обчислити деяку функцію (суму, мінімум, максимум, НСД) для елементів масиву на відрізку від індексу L до R включно.

2. *Точкове оновлення (point update)* − змінити значення елемента масиву за індексом i на нове значення x.

Наївний підхід до розв’язання цієї задачі полягає в простому ітеруванні по елементах від L до R для кожного запиту, що дає часову складність O(N) на один запит. Для оновлення достатньо O(1) часу. Якщо ж використовувати допоміжну структуру, таку як масив префіксних сум, для прискорення запитів на суму до O(1), то кожне оновлення елемента масиву вимагатиме перерахунку всіх наступних префіксних сум, що займе O(N) часу в найгіршому випадку. Обидва підходи є неефективними, якщо кількість запитів та оновлень є великою. Для ефективного розв’язання цієї проблеми були розроблені спеціалізовані структури даних, такі як дерево відрізків та дерево Фенвіка, які дають змогу виконувати обидві операції за логарифмічний час.

Дерево відрізків − це структура даних, що базується на принципі «розділяй та володарюй» (divide and conquer). Основна ідея полягає в тому, щоб заздалегідь обчислити та зберегти значення агрегатної функції для різних відрізків масиву. Кожен вузол у дереві відрізків відповідає за певний інтервал вихідного масиву. Кореневий вузол представляє весь масив, наприклад A[0…N − 1]. Кожен внутрішній вузол має двох нащадків, які представляють ліву та праву половини відрізка батьківського вузла. Цей процес поділу триває рекурсивно, доки відрізки не стануть одиничної довжини. Такі вузли, що відповідають за один елемент, є листками дерева.

Сила дерева відрізків полягає в попередніх обчисленнях. Структура жертвує пам'яттю об'ємом O(N) та часом побудови O(N) для того, щоб зменшити час виконання запитів на відрізку з O(N) до O(logN). Цей компроміс є сутністю багатьох просунутих структур даних. Оновлення значення елемента A[i] впливає лише на ті вузли, які лежать на шляху від листка, що відповідає індексу i, до кореня. Оскільки дерево є збалансованим, довжина цього шляху становить O(logN). Аналогічно, будь-який запит на відрізку можна декомпозувати на O(logN) заздалегідь обчислених відрізків (вузлів) у дереві. Саме ця декомпозиція забезпечує логарифмічну продуктивність для обох операцій.

Дерево відрізків є повним бінарним деревом, що дає змогу зручно реалізувати його за допомогою масиву (неявне представлення дерева). За 0-індексації масиву для вузла з індексом i його лівий нащадок міститься за індексом $2 \cdot i + 1$, а правий − за індексом $2 \cdot i + 2$. Батьківський вузол для вузла i міститься за індексом $\lfloor (i - 1)/2 \rfloor$.

Просторова складність. Висота дерева відрізків для масиву розміром N дорівнює $\lceil \log_2 N \rceil$. Кількість вузлів у повному бінарному дереві висотою h може сягати $2^{h+1} - 1$. Якщо N є степенем двійки, $N = 2^k$, то кількість вузлів дорівнює $2 \cdot N - 1$. В іншому випадку кількість вузлів буде меншою за $2 \cdot (2N) - 1 = 4N - 1$. Тому для зберігання дерева відрізків зазвичай виділяють масив розміром 4N, що є безпечною верхньою межею, яка гарантує відсутність виходу за межі масиву.

Побудова дерева виконується рекурсивною функцією build(node, start, end), де node − індекс поточного вузла в масиві дерева, а [start, end] − відрізок вихідного масиву, за який цей вузол відповідає.

Базовий випадок рекурсії. Якщо start == end, це означає, що ми досягли листка дерева. Вузол tree[node] зберігає значення A[start].

Рекурсивний крок. Якщо start < end, ми знаходимо середину відрізка mid = start + (end − start) / 2. Далі рекурсивно викликаємо build для лівого нащадка (індекс 2*node + 1, відрізок [start, mid]) та правого нащадка (індекс 2*node + 2, відрізок [mid+1, end]). Після повернення з рекурсивних викликів значення поточного вузла обчислюється комбінуванням значень його нащадків. Для задачі пошуку суми це буде tree[node] = tree[2*node + 1] + tree[2*node + 2].

Приклад побудови.

Розглянемо побудову дерева відрізків для масиву A = [1, 3,− 2, 8,− 7] для задачі знаходження суми на відрізку.

1. build(0, 0, 4): mid = 2. Викликаємо build(1, 0, 2) та build(2, 3, 4).
2. build(1, 0, 2): mid = 1. Викликаємо build(3, 0, 1) та build(4, 2, 2).
3. build(3, 0, 1): mid = 0. Викликаємо build(7, 0, 0) та build(8, 1, 1).
   − build(7, 0, 0): Листок. tree[1] = A = 1.
   − build(8, 1, 1): Листок. tree[2] = A[3] = 3.
   − Повертаємось до build(3, 0, 1). tree[4] = tree[1] + tree[2] = 1 + 3 = 4.
4. build(4, 2, 2): Листок. tree[5] = A[6] = −2.
5. Повертаємось до build(1, 0, 2). tree[3] = tree[4] + tree[5] = 4 + (−2) = 2.
6. Аналогічно для правої гілки:

- build(2, 3, 4): mid = 3. Викликаємо build(5, 3, 3) та build(6, 4, 4).
- build(5, 3, 3): Листок. tree[7] = A[4] = 8.
- build(6, 4, 4): Листок. tree[8] = A[5] = − 7.
- Повертаємось до build(2, 3, 4). tree[6] = tree[7] + tree[8] = 8 + (−7) = 1.

7. Повертаємось до кореня build(0, 0, 4). tree = tree[3] + tree[6] = 2 + 1 = 3.

У результаті отримуємо дерево, де кожен вузол зберігає суму відповідного відрізка.

Часова складність побудови становить O(N), оскільки кожен з приблизно 2N вузлів дерева відвідується рівно один раз.

Запит на відрізку (Query). Запит на відрізку [l, r] виконується рекурсивною функцією query(node, start, end, l, r). Під час виконання запиту розглядаються три можливі випадки.

1. Повне непотрапляння. Відрізок поточного вузла [start, end] повністю міститься поза межами запиту [l, r]. У цьому випадку вузол не робить внеску в результат, і функція повертає нейтральний елемент для операції (0 для суми, +∞ для мінімуму).

2. Повне потрапляння. Відрізок поточного вузла [start, end] повністю міститься всередині запиту [l, r]. У цьому випадку ми можемо використати заздалегідь обчислене значення і повернути tree[node].

3. Частковий перетин. Відрізок поточного вузла частково перетинається із запитом. У цьому випадку потрібно рекурсивно звернутися до лівого та правого нащадків і скомбінувати їхні результати.

Приклад запиту. Виконаємо запит query(0, 0, 4, 2, 4) (сума від A[6] до A[5]) на побудованому дереві.

1. query(0, 0, 4, 2, 4): Частковий перетин. mid = 2. Рекурсивно викликаємо query(1, 0, 2, 2, 4) та query(2, 3, 4, 2, 4).

2. query(1, 0, 2, 2, 4): Частковий перетин. mid = 1. Рекурсивно викликаємо query(3, 0, 1, 2, 4) та query(4, 2, 2, 2, 4).

− query(3, 0, 1, 2, 4): Повне непотрапляння. Повертаємо 0.

− query(4, 2, 2, 2, 4): Повне потрапляння. Повертаємо tree[5] = −2.

− Результат для query(1,...): 0 + (−2) = −2.

3. query(2, 3, 4, 2, 4): Повне потрапляння. Повертаємо tree[6] = 1.

4. Результат для кореня query(0,...): − 2 + 1= −1.

Правильна відповідь: A[6] + A[4] + A[5] = −2 + 8 − 7 = −1.

Часова складність запиту становитьO(logN). На кожному рівні дерева ми відвідуємо не більше двох вузлів, а висота дерева − O(logN).

Точкове оновлення (Update). Оновлення значення елемента A[idx] на val виконується рекурсивною функцією update(node, start, end, idx, val). Алгоритм спускається по дереву до листка, що відповідає індексу idx, змінює його значення, а потім, на зворотному шляху рекурсії, переобчислює значення всіх батьківських вузлів на шляху до кореня.

Приклад оновлення. Виконаємо update(0, 0, 4, 2, 10) (змінимо A[6] з −2 на 10).

1. update(0, 0, 4, 2, 10): mid = 2. Оскільки 2 ≤ 2, йдемо в ліве піддерево: update(1, 0, 2, 2, 10).

2. update(1, 0, 2, 2, 10): mid = 1. Оскільки 2 > 1, йдемо в праве піддерево: update(4, 2, 2, 2, 10).

3. update(4, 2, 2, 2, 10): Листок. tree[5] = 10.

4. Повертаємось до update(1, 0, 2,...): tree[3] = tree[4] + tree[5] = 4 + 10 = 14.

5. Повертаємось до update(0, 0, 4,...): tree = tree[3] + tree[6] = 14 + 1 = 15.

Часова складність оновлення становить O(logN), оскільки оновлюється лише один шлях у дереві від листка до кореня.

Задача «Range Minimum Query» (RMQ). Дерево відрізків легко адаптується для пошуку мінімуму (або максимуму) на відрізку. Для цього потрібно внести дві зміни.

1. Під час побудови та оновлення значення у внутрішньому вузлі обчислюється як мінімум значень його нащадків: tree[node] = min(tree[2*node + 1], tree[2*node + 2]).

2. Нейтральним елементом для операції min є додатна нескінченність (+∞). Функція query має повертати це значення у випадку повного непотрапляння відрізка вузла в запит.

*Задача з відкладеними операціями (Lazy Propagation).* Якщо потрібно оновити не один елемент, а цілий відрізок (наприклад, додати число X до всіх елементів від L до R), виконання R – L + 1 точкових оновлень призведе до загальної складності O(NlogN) на один запит, що є неефективним.

Техніка «відкладених операцій» дає змогу виконувати оновлення на відрізку за O(logN). Суть полягає в тому, щоб не оновлювати всі елементи відрізка одразу. Замість цього, якщо відрізок вузла повністю покривається запитом на оновлення, ми оновлюємо значення тільки в цьому вузлі і робимо для нього позначку («лінивий тег») про те, що всі його нащадки потребують оновлення. Це відкладене оновлення буде «проштовхнуте» на один рівень вниз до нащадків тільки тоді, коли ми звернемося до них під час наступних операцій (запиту або оновлення).

Ця оптимізація амортизує вартість оновлень на відрізку. Вона втілює принцип «виконуй роботу лише за потреби». Ключовим є те, що один «лінивий» тег на вузлі високого рівня представляє оновлення для великої кількості листкових вузлів, ефективно стискаючи інформацію. Наприклад, оновлення на відрізку $$ може охоплювати багато вузлів. Оновлення кожного листка окремо є повільним. Однак деякі вузли в дереві відрізків матимуть свої відрізки [start, end], що повністю потрапляють усередину. Для такого вузла ми знаємо, що *всі* його нащадки мають бути оновлені. Замість подальшої рекурсії ми можемо просто оновити значення цього вузла і прикріпити до нього «ліниву» позначку (наприклад, lazy[node] += val). Ця позначка означає, що всі його нащадки згодом потребуватимуть цього оновлення. Коли ми пізніше проходимо через цей вузол для іншого запиту, ми спочатку застосовуємо «ліниве» оновлення до його безпосередніх нащадків (і передаємо «ліниву» позначку вниз), перш ніж продовжити. Це амортизує вартість оновлення до O(logN).

Дерева відрізків є надзвичайно гнучкою структурою. Їх можна адаптувати для розв’язання широкого кола задач, таких як пошук найбільшого спільного дільника (НСД) на відрізку, підрахунок кількості нулів, пошук k-ї одиниці, задачі з обчислювальної геометрії та обробки зображень.

Дерево Фенвіка (або бінарне індексоване дерево, BIT) − це структура даних, оптимізована для ефективного обчислення префіксних сум та виконання точкових оновлень. На відміну від дерева відрізків, яке є явним бінарним деревом, дерево Фенвіка є

неявною структурою, представленою одним масивом того самого розміру, що й вихідний (плюс один елемент для зручності 1-індексації).

Кожен елемент BIT[i] зберігає суму елементів вихідного масиву в певному діапазоні [g(i) + 1, i], де функція g(i) визначається через двійкове представлення індексу i.

В основі дерева Фенвіка лежить той факт, що будь-яке додатне ціле число можна особливим чином представити як суму степенів двійки. Це дає змогу розбити будь-який префікс [1, k] на O(logk) блоків, суми яких зберігаються в дереві.

Для навігації по цих блоках використовується бітова операція i & − i (побітове «І» числа з його двійковим доповненням). Ця операція дає змогу ізолювати молодший встановлений (одиничний) біт числа i. Наприклад, для i = 12($1100_2$), − i у двійковому доповненні буде $0100_2$. Тоді $1100_2$&$0100_2$ = $0100_2$, що дорівнює 4. Це значення відповідає довжині відрізка, за який відповідає BIT[9].

Навігація по структурі:

1. *Для обчислення суми*i −= i & − i. Ця операція видаляє молодший встановлений біт, що відповідає переходу до наступного меншого блоку, який входить до складу префікса.

2. *Для оновлення*i += i & − i. Ця операція переходить до наступного більшого блоку, який містить поточний елемент, і який також потрібно оновити.

Операції (з 1-індексацією)

Обчислення суми префіксу get_sum(k). Щоб знайти суму елементів від 1 до k, потрібно додати значення BIT[k], потім перейти до індексу k' = k − (k & − k) і додати BIT[k'], і так далі, доки індекс не стане нулем.

Приклад. Нехай вихідний масив A = [2, 1, 1, 3, 2, 3, 4, 5]. Обчислимо get_sum(7).

1. sum = 0. Починаємо з i = 7 (0111_2).
2. sum += BIT[1]. i -= i & -i ->7 − 1 = 6. Новий i = 6 (0110_2).
3. sum += BIT[8]. i -= i & -i ->6 − 2 = 4. Новий i = 4 (0100_2).
4. sum += BIT[5]. i -= i & -i ->4 − 4 = 0. Новий i = 0.
5. Цикл завершується. Результат: BIT[1] + BIT[8] + BIT[5].

Оновлення елемента update(idx, delta). Щоб додати delta до елемента A[idx], потрібно додати delta до BIT[idx], потім перейти до індексу idx' = idx + (idx & -idx) і додати delta до BIT[idx'], і так далі, доки індекс не вийде за межі масиву.

Приклад. Оновимо A[4], додавши 2 (idx = 3, delta = 2).

1. Починаємо з i = 3 (0011_2).
2. BIT[4] += 2. i += i & -i ->3 + 1 = 4. Новий i = 4 (0100_2).
3. BIT[5] += 2. i += i & -i ->4 + 4 = 8. Новий i = 8 (1000_2).
4. BIT[2] += 2. i += i & -i ->8 + 8 = 16. Новий i = 16.
5. Цикл завершується (припускаючи, що розмір масиву менший за 16). Оновлено BIT[4], BIT[5], BIT[2].

Часова складність. Обидві операції, get_sum та update, виконуються за час O(logN), оскільки кількість ітерацій у циклі не перевищує кількість бітів в індексі.

*Порівняльний аналіз*

Вибір між деревом відрізків та деревом Фенвіка є класичним компромісом між універсальністю, просторовою ефективністю та складністю реалізації. Дерево Фенвіка можна розглядати як спеціалізовану оптимізацію для підмножини задач, які може розв’язувати дерево відрізків.

Дерево відрізків може працювати з будь-якою асоціативною операцією (сума, мінімум, максимум, НСД). Дерево Фенвіка, навпаки, фундаментально покладається на обчислення префіксних сум і знаходження суми на відрізку за формулою sum(R) - sum(L-1). Це вимагає, щоб операція мала обернену (віднімання для додавання, ділення для множення). Такі операції, як min або max, не мають простої оберненої, тому дерева Фенвіка не можуть безпосередньо розв’язувати задачі типу RMQ. Однак для задач, які вони *можуть* розв’язувати, дерева Фенвіка є кращим вибором: вони використовують O(N) пам’яті  проти O(4N) у дерева відрізків, а їхня реалізація значно простіша та коротша, що робить їх ідеальними для змагального програмування.

Таблиця 14.1 – Порівняння дерева відрізків та дерева Фенвіка

| Критерій | Дерево відрізків | Дерево Фенвіка |
|---|---|---|
| Основне призначення | Запити на довільних відрізках | Запити на префіксах |
| Підтримувані операції | Будь-яка асоціативна (сума, min/max, НСД) | Лише оборотні (сума, добуток, XOR) |
| Час побудови | O(N) | O(NlogN) |
| Час запиту | O(logN) | O(logN) |
| Час точкового оновлення | O(logN) | O(logN) |
| Час оновлення на відрізку | O(logN) (з Lazy Propagation) | Неефективно (або через складніші техніки) |
| Просторова складність | O(4N) | O(N) |
| Складність реалізації | Висока | Низька |

## 14.2. Багатовимірні просторові структури: KD-дерева

Коли виникає потреба організувати дані не в одновимірному масиві, а в багатовимірному просторі (наприклад, географічні координати на площині або характеристики об’єктів у просторі ознак), прості структури, як-от дерево відрізків, стають непридатними. KD-дерево (k-dimensional tree) є узагальненням ідеї бінарного дерева пошуку на багатовимірний простір. Воно належить до класу дерев двійкового поділу простору (Binary Space Partitioning tree), які ієрархічно ділять простір на менші регіони.

Побудова KD-дерева відбувається рекурсивно. На кожному кроці простір ділиться на дві частини за допомогою гіперплощини, паралельної одній з координатних осей.

1. *Вибір осі для поділу.* Вісь обирається циклічно залежно від глибини вузла в дереві. Для кореня (глибина 0) обирається перша вісь (наприклад, X), для його нащадків (глибина 1) − друга вісь (Y), для їхніх нащадків (глибина 2) − знову перша, і так далі. Формально, axis = depth % k, де k − розмірність простору.

2. *Пошук медіани.* Серед усіх точок у поточному підпросторі міститься медіанна точка за обраною віссю. Ця точка стає вузлом дерева.

3. *Поділ точок.* Решта точок ділиться на два підмасиви. Ті, у яких координата за обраною віссю менша за медіанну, потрапляють у лівий підмасив, а решта − у правий.

4. *Рекурсивна побудова.* Процес повторюється рекурсивно для лівого та правого підмасивів, створюючи ліве та праве піддерева.

Візуальний приклад (2D). Нехай є набір точок: {(2, 3), (5, 4), (9 ,6), (4, 7), (8, 1), (7, 2)}.

1. Рівень 0 (вісь X). Сортуємо за X: {(2, 3), (4, 7), (5, 4), (7, 2), (8, 1), (9, 6)}. Медіана − (7,2). Вона стає коренем. Простір ділиться вертикальною лінією x = 7.

- Ліворуч: {(2, 3), (5, 4), (4, 7)}.
- Праворуч: {(9,6), (8,1)}.

2. Рівень 1 (вісь Y). Ліве піддерево: Сортуємо {(2, 3), (5, 4), (4, 7)} за Y: {(2,3), (5,4), (4,7)}. Медіана − (5,4). Вона стає лівим нащадком кореня. Простір ділиться горизонтальною лінією y = 4.

− Ліворуч (нижче): {(2, 3)}.

- Праворуч (вище): {(4, 7)}.

Праве піддерево: Сортуємо {(9, 6), (8, 1)} за Y: {(8, 1), (9, 6)}. Медіана − (9, 6). Вона стає правим нащадком кореня. Простір ділиться горизонтальною лінією y = 6.

- Ліворуч (нижче): {(8, 1)}.
- Праворуч (вище): немає точок.

Цей процес продовжується доти, доки в кожному вузлі не залишиться по одній точці.

Складність побудови. Якщо на кожному кроці використовувати ефективний алгоритм пошуку медіани за O(N), загальна складність побудови дерева буде O(NlogN). Якщо використовувати сортування, складність буде O(kNlogN).

KD-дерева особливо ефективні для задачі пошуку найближчої до заданої точки Q точки з набору.

Алгоритм складається з двох фаз: спуск до листка та зворотний хід з можливим відсіканням гілок.

1. Спуск до листка. Починаючи з кореня, рухаємося вниз по дереву. На кожному вузлі порівнюємо координату точки запиту Q з координатою точки вузла за поточною віссю поділу й обираємо ліве або праве піддерево. Процес триває, доки не буде досягнуто листка. Цей листок стає поточним найкращим кандидатом на роль найближчого сусіда, а відстань до нього − поточним найкращим радіусом пошуку.

2. Зворотний хід (Backtracking). Повертаємося вгору по шляху спуску. На кожному вузлі, через який проходимо:

a) Оновлення кандидата. Перевіряємо, чи є точка в поточному вузлі ближчою за поточного найкращого кандидата. Якщо так, оновлюємо кандидата і радіус пошуку.

b) Ключовий крок (pruning − відсікання). Перевіряємо, чи перетинає сфера (коло в 2D) з центром у точці запиту Q і радіусом, що дорівнює поточній найкращій відстані, гіперплощину поділу поточного вузла.

c) Перевірка іншого піддерева. Якщо перетин є, це означає, що в іншій (ще не відвіданій) половині простору можуть існувати точки, ближчі за поточного кандидата. Тому потрібно рекурсивно запустити пошук для іншого піддерева. Якщо ж перетину немає, то інше піддерево можна безпечно проігнорувати, оскільки всі точки в ньому гарантовано будуть далі, ніж поточний найкращий кандидат. Це і є крок відсікання, який забезпечує ефективність алгоритму.

Після повернення до кореня процес завершується, і знайдений кандидат є найближчим сусідом.

*«Прокляття розмірності» (Curse of Dimensionality).* Геометрична інтуїція, що лежить в основі відсікання гілок у KD-дереві, руйнується в просторах високої розмірності. Об’єм гіперсфери стає незначним порівняно з об’ємом її обмежувального гіперкуба, що призводить до того, що сфера пошуку перетинає майже кожну гіперплощину поділу.

Ефективність пошуку найближчого сусіда залежить від кроку відсікання. Ми відсікаємо піддерево, якщо сфера пошуку не

перетинає відповідну йому область простору. У просторах високої розмірності ($k > 20$), простір стає величезним і розрідженим. Для випадкової точки відстані до її найближчого та найвіддаленішого сусідів стають майже однаковими. Внаслідок цього радіус сфери пошуку стає великим, і вона з високою ймовірністю перетинає більшість гіперплощин поділу, що зустрічаються під час зворотного ходу. Це змушує алгоритм досліджувати майже все дерево, знижуючи його продуктивність до $O(N)$, що не краще за повний перебір. Це пояснює, чому KD-дерева є ефективними переважно для просторів малої розмірності.

## 14.3. Алгоритми пошуку в рядках

Пошук підрядка (шаблону) в тексті є однією з найбільш фундаментальних задач в інформатиці. Розглянемо кілька класичних алгоритмів для її розв'язання.

Це найпростіший підхід. Вікно розміром з шаблон P (довжина M) пересувається по тексту T (довжина N) з кроком в один символ. На кожній позиції виконується посимвольне порівняння вмісту вікна з шаблоном.

Часова складність. У найгіршому випадку, наприклад, під час пошуку шаблону «AAAAAB» у тексті «AAAAAAAAAAAAAAAAB», на кожній з $N - M + 1$ позицій буде виконано M порівнянь. Так, загальна складність становить $O((N - M + 1) \cdot M)$, або просто $O(N \cdot M)$.

Замість повільного посимвольного порівняння рядків, алгоритм Рабіна-Карпа порівнює їхні хеш-значення. Хеш-функція перетворює рядок у число.

1. Обчислюється хеш шаблону P.
2. Обчислюється хеш для першого підрядка тексту довжиною M.
3. Якщо хеші збігаються, виконується посимвольна перевірка, щоб упевнитися, що це не колізія (два різні рядки мають однаковий хеш).
4. Якщо хеші не збігаються, вікно зсувається на один символ, і хеш для нового підрядка тексту ефективно переобчислюється.

*Поліноміальний ковзний хеш (Polynomial Rolling Hash).* Для ефективного переобчислення хешу використовується поліноміальний ковзний хеш. Рядок s = s0s1…sm − 1 розглядається як число в системі числення з основою p, де p − просте число, що перевищує розмір алфавіту. Хеш обчислюється за формулою:

$$\text{hash}(s) = (s_0 \cdot p^{m-1} + s_1 \cdot p^{m-2} + \cdots + s_{m-1} \cdot p^0) (\text{mod} q),$$

де q − велике просте число для уникнення переповнення та зменшення кількості колізій.

«Прокручування» хешу за O(1). Центральною інновацією алгоритму Рабіна-Карпа є ковзний хеш. Він перетворює операцію перехешування кожного вікна, що займає O(M), в операцію за O(1) за рахунок повторного використання попередніх обчислень. Це є формою динамічного програмування. Якщо відомий хеш $h_{old}$ для підрядка T[i…i + M − 1], то хеш $h_{new}$ для наступного підрядка T[i + 1…i + M] можна обчислити за O(1), не перебираючи всі символи заново. Для цього потрібно «відняти» внесок першого символа T[i] і «додати» внесок нового символа T[i + M]:

$$h_{new} = ((h_{old} - T[i] \cdot p^{M-1}) \cdot p + T[i + M]) (\text{mod} q).$$

Це дає змогу обчислити хеші для всіх підрядків тексту за час O(N) після початкового обчислення за O(M).

Оскільки хеш-значення обчислюються за модулем q, можливі ситуації, коли два різні рядки мають однаковий хеш (колізії). Тому, коли хеші шаблону та підрядка тексту збігаються, потрібно виконати повне посимвольне порівняння для підтвердження збігу.

Аналіз складності.

Середній час.O(N + M). Час O(M) йде на обчислення хешу шаблону, а O(N) − на ітерацію по тексту з переобчисленням хешу за O(1).

Найгірший час.O(N · M). Такий випадок може виникнути в разі невдалого вибору параметрів p і q, що призводить до великої кількості колізій, і на кожному кроці доводиться виконувати посимвольне порівняння.

Наївний алгоритм під час розбіжності символів просто зсуває шаблон на одну позицію і починає порівняння заново, «забуваючи» інформацію про те, яка частина тексту вже збіглася з префіксом шаблону. Алгоритм КМП використовує знання про внутрішню структуру самого шаблону, щоб під час розбіжності зробити максимально можливий «розумний» зсув, не пропускаючи потенційних входжень.

*Префікс-функція (масив LPS − Longest Proper Prefix which is also Suffix)*

Центральним елементом алгоритму КМП є префікс-функція, що зазвичай реалізується у вигляді LPS-масиву.

Визначення. Для кожного індексу і шаблону, lps[i] зберігає довжину найдовшого власного префікса підрядка pattern[0...i], який одночасно є його суфіксом. Власний префікс − це будь-який префікс, що не збігається з усім рядком.

Призначення. LPS-масив попередньо обчислює «внутрішні симетрії» або «перекриття» в самому шаблоні. Це дає змогу на етапі пошуку ніколи не переміщати вказівник тексту назад, досягаючи лінійної часової складності. Коли виникає розбіжність між text[i] та pattern[j], ми знаємо, що text[i − j...i − 1] збігся з pattern[0...j − 1]. Замість того, щоб зсунути шаблон на один символ і повторно порівнювати з text[i – j + 1], ми можемо запитати: «Чи існує коротший префікс pattern, який є суфіксом щойно зіставленої частини (pattern[0...j − 1])?». Довжина цього префікса − це саме те, що нам говорить lps[j − 1]. Ми можемо зсунути шаблон уперед так, щоб цей префікс вирівнявся з кінцем зіставленого сегмента тексту, і відновити порівняння з pattern[lps[j − 1]], не переміщаючи вказівник тексту і. Це дає змогу уникнути зайвої роботи.

*Алгоритм обчислення LPS-масиву.* LPS-масив можна ефективно обчислити за час O(M) за допомогою динамічного програмування з двома вказівниками (len та i).len відстежує довжину поточного найдовшого префікса-суфікса, а і ітерує по шаблону.

Таблиця 14.2 – Покрокове обчислення LPS-масиву для шаблону «AABAACAABAA»

| i | pat[i] | len | pat[i] == pat[len]? | Дія | lps[i] |
|---|---|---|---|---|---|
| 0 | A | – | – | lps = 0 | 0 |
| 1 | A | 0 | A == pat (A) | len++, lps[3] = len | 1 |
| 2 | B | 1 | B!= pat[3] (A) | len = lps[len-1] = lps = 0 | 0 |
| 3 | A | 0 | A == pat (A) | len++, lps[4] = len | 1 |
| 4 | A | 1 | A == pat[3] (A) | len++, lps[5] = len | 2 |
| 5 | C | 2 | C!= pat[6] (B) | len = lps[len-1] = lps[3] = 1. Спробувати з len=1 | |
| | | 1 | C!= pat[3] (A) | len = lps[len-1] = lps = 0. Спробувати з len = 0 | 0 |
| 6 | A | 0 | A == pat (A) | len++, lps[8] = len | 1 |
| 7 | A | 1 | A == pat[3] (A) | len++, lps[1] = len | 2 |
| 8 | B | 2 | B == pat[6] (B) | len++, lps[2] = len | 3 |
| 9 | A | 3 | A == pat[4] (A) | len++, lps[10] = len | 4 |
| 10 | A | 4 | A == pat[5] (A) | len++, lps[11] = len | 5 |

Пошук виконується за допомогою двох вказівників: i для тексту та j для шаблону.

Якщо text[i] == pattern[j], обидва вказівники збільшуються.

Якщо text[i]!= pattern[j] (розбіжність):

− Якщо j!= 0, вказівник i залишається на місці, а вказівник j «відкочується» на позицію lps[j − 1]. Це відповідає зсуву шаблону вправо.

− Якщо j == 0, це означає, що навіть перший символ шаблону не збігся, тому просто переходимо до наступного символу тексту, збільшуючи i.

Аналіз складності:попередня обробка (побудова LPS-масиву) займає O(M). Фаза пошуку займає O(N), оскільки вказівник i ніколи не рухається назад. Загальна складність алгоритму КМП − O(N + M).

Таблиця 14.3 – Порівняльна характеристика алгоритмів пошуку рядків

| Алгоритм | Час попередньої обробки | Час пошуку (найгірший) | Час пошуку (середній) | Просторова складність |
|---|---|---|---|---|
| Наївний | O(1) | O(N · M) | O(N · M) | O(1) |
| Рабіна-Карпа | O(M) | O(N· M) | O(N + M) | O(1) |
| Кнута-Морріса-Пратта | O(M) | O(N) | O(N) | O(M) |

## 14.4. Реалізація алгоритму КМП

Ця функція обчислює масив найдовших власних префіксів, що є також суфіксами (LPS), для заданого шаблону.

```
C++
#include<vector>
#include<string>

// Функція для обчислення LPS-масиву для шаблону `pat`
void computeLPSArray(const std::string& pat, std::vector<int>& lps) {
   int M = pat.length();
   lps.assign(M, 0); // Ініціалізуємо масив нулями

   int length = 0; // Довжина попереднього найдовшого префікса-суфікса
   lps = 0; // lps завжди дорівнює 0

   int i = 1;
   while (i < M) {
      if (pat[i] == pat[length]) {
         // Якщо символи збігаються, збільшуємо довжину поточного
         // префікса-суфікса і записуємо її в lps[i]
         length++;
         lps[i] = length;
         i++;
      } else {
         // Якщо символи не збігаються
         if (length!= 0) {
            // "Відкочуємося" до довжини префікса-суфікса для попереднього символу.
            // Це дозволяє нам спробувати знайти коротший префікс,
            // який може збігтися.
            // Важливо: ми не збільшуємо `i` на цьому кроці.
            length = lps[length - 1];
         } else {
            // Якщо довжина вже 0, то збігу немає.
            // Записуємо 0 і переходимо до наступного символу.
            lps[i] = 0;
            i++;
         }
      }
   }
```

Ця функція використовує попередньо обчислений LPS-масив для ефективного пошуку шаблону в тексті.

*C++*

```
#include<iostream>

// Основна функція, що реалізує алгоритм КМП
void KMPSearch(const std::string& txt, const std::string& pat) {
   int N = txt.length();
   int M = pat.length();

   // Створюємо та обчислюємо LPS-масив для шаблону
   std::vector<int> lps(M);
   computeLPSArray(pat, lps);

   int i = 0; // Вказівник для тексту
   int j = 0; // Вказівник для шаблону

   while (i < N) {
      if (pat[j] == txt[i]) {
         // Символи збігаються, рухаємо обидва вказівники
         i++;
         j++;
      }

      if (j == M) {
         // Знайдено повний збіг шаблону
         std::cout << "Знайдено входження шаблону з індексу " << i - j
<< std::endl;
         // Зсуваємо шаблон, використовуючи lps, для пошуку наступних
входжень
         j = lps[j - 1];
      } else if (i < N && pat[j]!= txt[i]) {
         // Розбіжність після j збігів
         if (j!= 0) {
            // Не втрачаємо прогрес. Зсуваємо вказівник шаблону
            // на позицію, що відповідає довжині префікса-суфікса
            // для вже зіставленої частини.
            j = lps[j - 1];
         } else {
```

```
            // Якщо розбіжність на першому символі шаблону,
            // просто рухаємо вказівник тексту.
            i++;
        }
    }
}
}
```

## Контрольні питання

1. Поясніть принцип «розділяй та володарюй» на прикладі Дерева відрізків.

2. Чому масив для зберігання Дерева відрізків зазвичай має розмір 4N?

3. У яких випадках виникає потреба у «відкладених операціях» (lazy propagation)?

4. Поясніть, як бітова операція i & − i використовується для навігації в Дереві Фенвіка.

5. У чому принципова різниця між Деревом відрізків та Деревом Фенвіка? Коли варто віддати перевагу кожному з них?

6. Що таке «прокляття розмірності» і як воно впливає на ефективність KD-дерев?

7. Поясніть концепцію «ковзного хешу» в алгоритмі Рабіна-Карпа. Чому він дає змогу досягти складності O(1) для перерахунку хешу?

8. Що таке колізія в алгоритмі Рабіна-Карпа і як з нею боротися?

9. Яке призначення префікс-функції (LPS-масиву) в алгоритмі КМП?

10. Поясніть, як КМП уникає повторних порівнянь символів тексту, які вже були зіставлені.

## Задачі для самостійного опрацювання

1. Реалізувати Дерево відрізків для пошуку найбільшого спільного дільника (НСД) на відрізку.

2. Використовуючи Дерево Фенвіка, написати програму для підрахунку кількості інверсій у масиві.

3. Реалізувати Дерево відрізків з відкладеними операціями для задачі «додати число X до всіх елементів на відрізку $$» та «знайти суму на відрізку $$».

4. Дано масив. Знайти для кожного елемента кількість елементів праворуч від нього, які є меншими за нього. Розв’язати за допомогою Дерева Фенвіка або Дерева відрізків.

5. Реалізувати побудову 2D-дерева та функцію пошуку всіх точок у заданому прямокутнику.

6. Реалізувати алгоритм Рабіна-Карпа для пошуку всіх входжень шаблону в текст. Використати подвійне хешування для зменшення ймовірності колізій.

7. Дано рядок. Знайти довжину його найкоротшого періоду (найменший рядок t, повторенням якого можна утворити вихідний рядок). Розв’язати за допомогою префікс-функції.

8. Реалізувати пошук k-го найближчого сусіда в KD-дереві.

9. 2D Дерево Фенвіка. Реалізувати структуру для відповіді на запити про суму в прямокутнику $(x_1, y_1) - (x_2, y_2)$ у 2D-матриці з точковими оновленнями.

10. Персистентне дерево відрізків. Реалізувати структуру, яка дає змогу робити запити на будь-якій попередній версії масиву.

# ТЕМА 15
# СУЧАСНІ АЛГОРИТМИ

Класична теорія алгоритмів, розроблена в епоху, коли обчислювальні ресурси були обмеженими, а набори даних − відносно невеликими, заклала фундаментальні основи інформатики. Такі алгоритми, як сортування злиттям, пошук Дейкстри та обхід бінарних дерев, залишаються наріжними каменями дисципліни. Однак сучасний цифровий світ характеризується безпрецедентним вибухом даних, відомим як Big Data[30]. Цей феномен, що описується трьома «V» − величезним об'ємом (Volume), високою швидкістю надходження (Velocity) та надзвичайним різноманіттям (Variety), − виявив обмеження класичних підходів. Алгоритми, розраховані на роботу з даними, що повністю вміщуються в оперативну пам'ять, стають неефективними, коли стикаються з петабайтами інформації, що генерується щосекунди.

## 15.1. Алгоритми для машинного навчання та Big Data

З появою машинного навчання[31] (ML) та аналізу великих даних виникла потреба в алгоритмах, здатних ефективно обробляти багатовимірні та високопов'язані дані. Класичні структури даних часто виявляються недостатньо гнучкими або продуктивними для таких завдань.

В основі сучасних фреймворків машинного навчання, таких як TensorFlow та PyTorch, лежить математичний об'єкт під назвою тензор. У контексті ML тензор − це узагальнення векторів та

[30] Big Data (Великі дані) – це надзвичайно великі, складні та швидко зростаючі обсяги даних, які не можуть бути ефективно оброблені традиційними методами та інструментами управління базами даних. Цей термін відноситься як до самих даних, так і до технологій, що використовуються для їх збору, зберігання, аналізу та візуалізації.

[31] Машинне навчання (Machine Learning, ML) – це розділ штучного інтелекту, що дає комп'ютерам здатність навчатися на даних без явного програмування. Замість того, щоб програмувати кожен крок для розв'язання задачі, ви створюєте алгоритм, який сам знаходить закономірності в даних і використовує їх для прийняття рішень або прогнозування.

матриць на довільну кількість вимірів. Простіше кажучи, це багатовимірний масив чисел, який слугує основною структурою даних для всіх вхідних даних, ваг, зсувів та вихідних значень у нейронних мережах.

Ключовою характеристикою тензора є його ранг (або кількість вимірів). Ранг визначає, скільки індексів потрібно для доступу до окремого елемента тензора.

Тензор рангу 0 (скаляр) − одне число. Не має вимірів. Приклад: 7. Тензор рангу 1 (вектор) − одновимірний масив (список) чисел. Тензор рангу 2 (матриця) − двовимірний масив (таблиця) чисел. Тензор рангу 3 − тривимірний масив (куб) чисел. Тензори вищих рангів − узагальнення на 4, 5 і більше вимірів.

Форма (shape) тензора − це кортеж цілих чисел, що описує розмір кожного виміру. Наприклад, матриця з 2 рядків та 3 стовпців має форму (2, 3).

Тензори є надзвичайно гнучкими для представлення різноманітних типів даних, що використовуються в ML.

1. Зображення.

1.1. Чорно-біле зображення розміром 28 x 28 пікселів можна представити як 2D тензор (матрицю) форми (28, 28), де кожен елемент − це яскравість пікселя.

1.2. Кольорове (RGB) зображення того самого розміру представляється як 3D тензор форми (28, 28, 3), де 3 − це кількість колірних каналів (червоний, зелений, синій).

2. Пакетна обробка (Batch Processing). Для підвищення ефективності навчання нейронні мережі часто обробляють дані не по одному зразку, а пакетами (batches). Це додає ще один вимір на початку форми тензора. Наприклад, пакет зі 100 кольорових зображень розміром 28 x 28 буде представлений 4D тензором форми (100, 28, 28, 3), де 100 − розмір пакета.

3. Текстові дані. Текст також перетворюється на тензори. Один із поширених підходів − векторні представлення слів (word embeddings). Кожне слово представляється щільним вектором фіксованої довжини (наприклад, 300). Тоді речення з 10 слів стає 2D тензором (матрицею) форми (10, 300), а пакет з 50 таких речень − 3D тензором форми (50, 10, 300).

Ефективність тензорів у ML зумовлена не лише гнучкістю представлення даних, а й можливістю виконувати над ними оптимізовані математичні операції. Ця оптимізація досягається завдяки паралелізації обчислень на графічних (GPU) та тензорних (TPU) процесорах, для яких структура тензора є ідеальною.

1. Поелементні операції (Element-wise Operations). Операція застосовується до кожного елемента тензора незалежно. Приклади: додавання, віднімання, множення тензорів однакової форми, застосування функцій активації (наприклад, ReLU).

2. Множення матриць (Matrix Multiplication). Ключова операція в повнозв'язних шарах нейронних мереж. Для множення двох матриць A (форми (m, k)) та B (форми (k, n)) кількість стовпців першої матриці має дорівнювати кількості рядків другої. Результатом буде матриця форми (m, n).

3. Трансляція (Broadcasting). Це потужний механізм, що дає змогу виконувати операції над тензорами різних, але сумісних форм, без явного створення копій даних. Правила трансляції:

1. Якщо ранги тензорів різні, до тензора з меншим рангом додаються виміри розміром 1 на початку, доки ранги не зрівняються.

2. Два тензори є сумісними, якщо для кожного виміру їх розміри або однакові, або один з них дорівнює 1.

3. Під час операції виміри розміром 1 «розтягуються» (копіюються) до розміру відповідного виміру іншого тензора.

Приклад трансляції. Додавання вектора зсуву (bias) до матриці активацій.

Нехай є матриця активацій A форми (3, 4) і вектор зсуву b форми (4,).

4. Ранг b (1) менший за ранг A (2). Форма b розширюється до (1, 4).

5. Порівнюємо форми: A(3, 4) та b(1, 4).

Останній вимір: 4 і 4 (однакові).

Перший вимір: 3 і 1 (сумісні).

2.1. Під час додавання вектор b «розтягується» вздовж першого виміру, щоб відповідати формі A. Фактично, він додається до кожного рядка матриці A.

```
Псевдокод: Поелементне додавання та множення матриць
function element_wise_add(tensorA, tensorB):
  // Припускаємо, що tensorA та tensorB мають однакову форму (rows, cols)
  result = create_tensor(rows, cols)
  for i from 0 to rows-1:
    for j from 0 to cols-1:
      result[i][j] = tensorA[i][j] + tensorB[i][j]
  return result

function matrix_multiply(tensorA, tensorB):
  // Припускаємо форму A: (m, k), B: (k, n)
  m = rows(tensorA), k = cols(tensorA), n = cols(tensorB)
  result = create_tensor(m, n)
  for i from 0 to m-1:
    for j from 0 to n-1:
      sum = 0
      for p from 0 to k-1:
        sum = sum + tensorA[i][p] * tensorB[p][j]
      result[i][j] = sum
  return result
```

```
Python
import numpy as np
import torch

# Створення тензорів
# NumPy
a_np = np.array([, ])
b_np = np.array([, ])
bias_np = np.array()

# PyTorch
a_pt = torch.tensor([, ])
b_pt = torch.tensor([, ])
bias_pt = torch.tensor()

# Поелементне додавання
add_np = a_np + b_np
add_pt = a_pt + b_pt
# Результат: [, ]
```

```
# Множення матриць
matmul_np = a_np @ b_np
matmul_pt = torch.matmul(a_pt, b_pt)
# Результат: [, ]

# Трансляція (Broadcasting)
broadcast_np = a_np + bias_np
broadcast_pt = a_pt + bias_pt
# Результат: [, ]
```

На C++ реалізація тензорних операцій без спеціалізованих бібліотек є значно складнішою і демонструє, яку роботу виконують фреймворки. Тензор зазвичай представляється як одновимірний масив (вектор) даних разом з метаданими про його форму та кроки (strides).

```
C++
#include<iostream>
#include<vector>
#include<numeric>

// Проста структура для представлення 2D тензора
struct Tensor2D {
   std::vector<float> data;
   int rows;
   int cols;

   Tensor2D(int r, int c) : rows(r), cols(c), data(r * c, 0.0f) {}

   float& at(int r, int c) {
      return data[r * cols + c];
   }
};

void print_tensor(const Tensor2D& t) {
   for (int i = 0; i < t.rows; ++i) {
      for (int j = 0; j < t.cols; ++j) {
         std::cout << t.data[i * t.cols + j] << " ";
      }
      std::cout << std::endl;
   }
}
```

```
Tensor2D matrix_multiply(const Tensor2D& a, const Tensor2D& b) {
    if (a.cols!= b.rows) {
        throw std::runtime_error("Matrix dimensions are not compatible for
multiplication");
    }
    Tensor2D result(a.rows, b.cols);
    for (int i = 0; i < a.rows; ++i) {
        for (int j = 0; j < b.cols; ++j) {
            for (int k = 0; k < a.cols; ++k) {
                result.at(i, j) += a.data[i * a.cols + k] * b.data[k * b.cols + j];
            }
        }
    }
    return result;
}

int main() {
    Tensor2D a(2, 2);
    a.at(0, 0) = 1; a.at(0, 1) = 2;
    a.at(1, 0) = 3; a.at(1, 1) = 4;

    Tensor2D b(2, 2);
    b.at(0, 0) = 5; b.at(0, 1) = 6;
    b.at(1, 0) = 7; b.at(1, 1) = 8;

    Tensor2D result = matrix_multiply(a, b);
    std::cout << "Matrix Multiplication Result:" << std::endl;
    print_tensor(result); // Очікуваний результат: [, ]

    return 0;
}
```

У світі Big Data значна частина інформації має структуру мережі: соціальні мережі, інтернет, біологічні мережі, логістичні ланцюги. Графи є природною моделлю для представлення таких пов’язаних даних, де об’єкти − це вершини (nodes), а зв’язки між ними − ребра (edges). Графові алгоритми дають змогу аналізувати ці структури, виявляти приховані закономірності, важливі вузли та спільноти.

Графові алгоритми можна класифікувати за їхнім призначенням:

1. Пошук шляхів (Pathfinding) − визначення найкоротших або найоптимальніших маршрутів між вершинами (наприклад, алгоритм Дейкстри).

2. Аналіз центральності (Centrality Analysis) − ідентифікація найважливіших або найвпливовіших вершин у мережі.

3. Виявлення спільнот (Community Detection) − групування вершин у щільно пов'язані кластери.

Розглянемо два ключові алгоритми, що широко застосовуються для аналізу великих графів: PageRank та Louvain Modularity.

PageRank − це алгоритм аналізу посилань, розроблений засновниками Google для ранжування вебсторінок. Його основна ідея полягає в тому, що важливість сторінки визначається не лише кількістю посилань на неї, але й важливістю сторінок, що на неї посилаються. Кожне посилання розглядається як «голос» за важливість.

Алгоритм моделює поведінку «випадкового серфера», який переміщується по вебу, з однаковою ймовірністю натискаючи на будь-яке з посилань на поточній сторінці. Ранг сторінки (її PageRank) відповідає ймовірності того, що серфер опиниться на цій сторінці після великої кількості переходів.

PageRank обчислюється ітераційно. Ранг сторінки u (PR(u)) залежить від рангів усіх сторінок v, що входять до множини $B_u$ (сторінки, які посилаються на u). Кожна сторінка v передає свою «важливість» порівну між усіма своїми вихідними посиланнями, кількість яких позначається як L(v).

Проста формула має вигляд:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Однак ця формула має проблеми з «тупиковими» сторінками (які не мають вихідних посилань) та циклами. Для їх вирішення вводиться коефіцієнт затухання (damping factor) d, зазвичай рівний 0.85. Він моделює ймовірність того, що серфер продовжить переходити за посиланнями. З імовірністю (1 − d) серфер

«нудьгує» і переходить на випадкову сторінку з усієї мережі. Це гарантує, що процес не зациклиться і кожна сторінка отримає певний ранг.

Повна формула PageRank:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

де N − загальна кількість сторінок у мережі.

Процес обчислення:

1. Ініціалізація − усім N сторінкам присвоюється початковий ранг 1/N.

2. Ітерація − для кожної сторінки обчислюється новий ранг за наведеною вище формулою, використовуючи ранги з попередньої ітерації.

3. Повторення − крок 2 повторюється доти, доки ранги не стабілізуються (зміни між ітераціями не стануть меншими за заданий поріг) або не буде досягнуто максимальної кількості ітерацій.

Приклад обчислення PageRank. Розглянемо простий граф з 4 вузлів. Нехай d = 0.85. Початковий ранг кожного вузла $PR_0 = ¼ = 0.25$.

PR(A) = PR(C)

PR(B) = PR(A)/2 + PR(D)

PR(C) = PR(A)/2

PR(D) = PR(B)

Після кількох ітерацій ранги збігаються до стабільних значень, які відображають відносну важливість вузлів.

Алгоритм Louvain − це ієрархічний кластерний алгоритм, призначений для виявлення спільнот у великих мережах. Спільнота − це група вершин, які мають щільніші зв’язки всередині групи, ніж із вершинами поза нею.

В основі алгоритму лежить оптимізація метрики під назвою модулярність. Модулярність (Q) вимірює якість розбиття графа на спільноти. Вона порівнює частку ребер, що містяться всередині спільнот, з очікуваною часткою таких ребер у випадковому графі з такою самою послідовністю степенів вершин. Високе значення модулярності вказує на сильний поділ на спільноти.

Двофазний ітераційний процес – алгоритм, який працює ітераційно, повторюючи два кроки, доки модулярність не перестане зростати.

1. Фаза 1: оптимізація модулярності.

1.1. Спочатку кожна вершина поміщається у власну спільноту.

1.2. Потім для кожної вершини i розглядається можливість її переміщення до спільноти кожного з її сусідів j.

1.3. Обчислюється зміна модулярності (ΔQ) для кожного такого переміщення.

1.4. Вершина i переміщується до тієї спільноти, яка дає найбільший приріст модулярності (якщо він додатний).

1.5. Цей процес повторюється для всіх вершин, доки жодне переміщення не зможе покращити загальну модулярність.

2. Фаза 2: агрегація спільнот.

2.1. Спільноти, знайдені на першому етапі, «стискаються» в нові супер-вершини.

2.2. Будується новий, менший граф, де вершини − це спільноти з попереднього кроку.

2.3. Вага ребра між двома новими супер-вершинами дорівнює сумі ваг ребер між вершинами відповідних вихідних спільнот. Ребра всередині спільноти стають петлями на нових вершинах.

2.4. Після цього алгоритм повертається до Фази 1 на цьому новому агрегованому графі.

Цей процес створює ієрархію спільнот, де спільноти, знайдені на пізніших ітераціях, складаються зі спільнот, знайдених на ранніх.

*Псевдокод: ітерація PageRank*

```
function PageRank_Iteration(graph, ranks, d):
  N = number_of_nodes(graph)
  new_ranks = create_map_with_zeros()
  for each node u in graph:
    rank_sum = 0
    for each node v that links to u:
      rank_sum += ranks[v] / out_degree(v)
    new_ranks[u] = (1 - d) / N + d * rank_sum
```

```
  return new_ranks
Псевдокод: Фаза 1 алгоритму Louvain
function Modularity_Optimization(graph, communities):
  improved = true
  while improved:
    improved = false
    for each node u in graph:
      best_community = current_community(u)
      max_delta_Q = 0
      for each neighbor v of u:
        neighbor_community = current_community(v)
        delta_Q = calculate_modularity_gain(u, neighbor_community)
        if delta_Q > max_delta_Q:
          max_delta_Q = delta_Q
          best_community = neighbor_community
      if best_community!= current_community(u):
        move_node(u, to=best_community)
        improved = true
```

```
Python
import networkx as nx
import community as community_louvain # pip install python-louvain

# Створення графа
G = nx.karate_club_graph()

# Обчислення PageRank
pagerank_scores = nx.pagerank(G, alpha=0.85)
print("PageRank Scores:", pagerank_scores)

# Виявлення спільнот за допомогою алгоритму Louvain
partition = community_louvain.best_partition(G)
print("Louvain Communities:", partition)
```

```
C++
#include<iostream>
#include<vector>
#include<map>
#include<iomanip>
```

```
    void run_pagerank(const std::map<int, std::vector<int>>& adj, int iterations) {
        int n = adj.size();
        std::map<int, double> ranks;
        for (const auto& pair : adj) {
            ranks[pair.first] = 1.0 / n;
        }

        double d = 0.85;

        for (int iter = 0; iter < iterations; ++iter) {
            std::map<int, double> new_ranks;
            for (const auto& pair : adj) {
                int u = pair.first;
                double rank_sum = 0.0;
                // Потрібно побудувати зворотний граф для знаходження вхідних посилань
                // Для простоти, припустимо, ми його маємо
                // for (int v : incoming_links_to_u) {
                //     rank_sum += ranks[v] / adj.at(v).size();
                // }
                // Ця частина коду є концептуальною і потребує реалізації зворотних посилань
            }
            // ranks = new_ranks; // Оновлення рангів
        }

        // Вивід результатів
        for (const auto& pair : ranks) {
            std::cout << "Node " << pair.first << ": " << std::fixed << std::setprecision(4) << pair.second << std::endl;
        }
    }
```

Під час роботи з величезними наборами даних часто виникає задача перевірки, чи належить певний елемент до множини. Використання хеш-таблиць або збалансованих дерев є ефективним, але вимагає зберігання самих елементів, що може бути надто витратним для пам’яті. Фільтр Блума − це компактна імовірнісна

структура даних, яка дає змогу вирішити цю задачу з мінімальними витратами пам’яті, жертвуючи водночас детермінованістю відповіді.

Фільтр Блума складається з двох компонентів.

1. Бітовий масив (bit array) розміром m, ініціалізований нулями.

2. k незалежних хеш-функцій, кожна з яких відображає елемент у діапазон індексів [0, m − 1].

Операції над фільтром виконуються так.

Додавання елемента (add):

1. Елемент хешується k разів за допомогою k різних хеш-функцій.

2. Кожна хеш-функція генерує індекс у бітовому масиві.

3. Біти за всіма k індексами встановлюються в 1.

Перевірка належності (check):

4. Елемент хешується тими самими k хеш-функціями.

5. Перевіряються біти за всіма k отриманими індексами.

6. Якщо хоча б один з цих бітів дорівнює 0, елемент гарантовано відсутній у множині.

7. Якщо всі біти дорівнюють 1, елемент, ймовірно, присутній у множині.

На схемі показано додавання елементів «x, "y", "z" з k = 3 хеш-функціями. Під час перевірки елемента "w" виникає хибнопозитивне спрацювання, оскільки всі три його біти були встановлені попередніми елементами.

Аналіз помилок. Ключова особливість фільтра Блума − асиметричність помилок

1. Хибнопозитивні спрацювання (False Positives). Фільтр може помилково повідомити, що елемент присутній, хоча насправді його не додавали. Це відбувається, коли всі біти, що відповідають елементу, були встановлені в 1 унаслідок додавання інших елементів.

2. Відсутність хибнонегативних спрацювань (No False Negatives). Фільтр ніколи не повідомить, що елемент відсутній, якщо він насправді був доданий. Це пов’язано з тим, що операція додавання лише встановлює біти в 1 і ніколи їх не скидає.

Імовірність хибнопозитивного спрацювання (p) залежить від розміру масиву (m), кількості доданих елементів (n) та кількості хеш-функцій (k). Приблизна формула:

$$p \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

Для заданих m та n існує оптимальна кількість хеш-функцій k, яка мінімізує p:

$$k = \frac{m}{n} \ln(2)$$

За оптимального k фільтр з 1 % ймовірністю помилки вимагає лише близько 9.6 біт на елемент, незалежно від розміру самих елементів.

Фільтр Блума є класичним прикладом архітектурного патерну, де швидка, але імовірнісна перевірка в пам'яті використовується для уникнення повільних та дорогих операцій, таких як дискове I/O або мережеві запити. Він діє як «воротар».

1. Кешування в базах даних. Перед виконанням дорогого запиту до диска система може перевірити наявність ключа у фільтрі Блума. Якщо відповідь «гарантовано ні», запит до диска не виконується.

2. Веббраузери. Google Chrome використовував фільтри Блума для швидкої перевірки URL на приналежність до списку шкідливих сайтів. Негативна відповідь означає, що сайт безпечний, а позитивна − вимагає повної перевірки.

3. Розподілені системи. Для уникнення дублікатів записів у великих розподілених системах.

*Псевдокод*

```
class BloomFilter:
  bit_array m
  hash_functions k

  function add(item):
    for i from 1 to k:
      index = hash_i(item) % m.size
      m[index] = 1
```

```
 function check(item):
   for i from 1 to k:
     index = hash_i(item) % m.size
     if m[index] == 0:
       return "Definitely Not Present"
   return "Possibly Present"
```

```
Python
# pip install mmh3 bitarray
import mmh3
from bitarray import bitarray

class BloomFilter:
    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = bitarray(size)
        self.bit_array.setall(0)

    def add(self, item):
        for i in range(self.hash_count):
            digest = mmh3.hash(item, i) % self.size
            self.bit_array[digest] = True

    def check(self, item):
        for i in range(self.hash_count):
            digest = mmh3.hash(item, i) % self.size
            if not self.bit_array[digest]:
                return False
        return True

# Приклад використання
bf = BloomFilter(100, 3)
bf.add("apple")
bf.add("banana")

print(bf.check("apple"))    # True
print(bf.check("cherry"))   # False (швидше за все)
```

```
C++
#include<iostream>
#include<vector>
#include<string>

// Прості хеш-функції для демонстрації
size_t h1(const std::string& s, size_t arrSize) {
    size_t hash = 0;
    for (char c : s) {
        hash = (hash + c) % arrSize;
    }
    return hash;
}

size_t h2(const std::string& s, size_t arrSize) {
    size_t hash = 7;
    for (char c : s) {
        hash = (hash * 31 + c) % arrSize;
    }
    return hash;
}

class BloomFilter {
private:
    std::vector<bool> bit_array;
    size_t size;

public:
    BloomFilter(size_t s) : size(s), bit_array(s, false) {}

    void add(const std::string& item) {
        bit_array[h1(item, size)] = true;
        bit_array[h2(item, size)] = true;
        std::cout << item << " inserted" << std::endl;
    }

    bool check(const std::string& item) {
        return bit_array[h1(item, size)] && bit_array[h2(item, size)];
    }
};
```

```
int main() {
    BloomFilter bf(100);
    bf.add("apple");
    bf.add("banana");

    std::cout << "Check 'apple': " << (bf.check("apple")? "Possibly
Present" : "Not Present") << std::endl;
    std::cout << "Check 'cherry': " << (bf.check("cherry")? "Possibly
Present" : "Not Present") << std::endl;

    return 0;
}
```

## 15.2. Потокові алгоритми (streaming algorithms) та Count-Min Sketch

Класичні алгоритми зазвичай припускають, що всі вхідні дані доступні в пам’яті для багаторазового доступу. Однак у багатьох сучасних застосунках дані надходять у вигляді безперервного, потенційно нескінченного потоку (stream). Прикладами є мережевий трафік, дані з сенсорів, лог-файли серверів, потік транзакцій або стрічка новин у соціальних мережах. Зберігати такі обсяги даних повністю неможливо або недоцільно. Це призвело до розробки потокових алгоритмів.

*Потокові алгоритми* − це алгоритми, призначені для оброблення даних, що надходять послідовно, за один або невелику кількість проходів, використовуючи водночас сублінійний обсяг пам’яті, тобто значно менший, ніж обсяг самого потоку даних.

Основні обмеження та цілі потокових алгоритмів:

1. Один прохід. Дані обробляються в тому порядку, в якому вони надходять, і після обробки елемент потоку зазвичай відкидається. Повернення до попередніх елементів неможливе.

2. Обмежена пам’ять. Обсяг пам’яті (S(n)) має бути значно меншим за розмір потоку (n), ідеально − полілогарифмічним (O(polylog n)).

3. Швидка обробка. Час обробки кожного елемента потоку має бути мінімальним.

Через ці жорсткі обмеження потокові алгоритми часто є імовірнісними та апроксимаційними. Вони не дають точної відповіді, а лише її оцінку з певними гарантіями точності та ймовірності.

Однією з фундаментальних задач в аналізі потоків є підрахунок частот елементів. Наприклад, визначення найпопулярніших товарів на сайті електронної комерції в реальному часі. Зберігати лічильник для кожного унікального елемента неможливо, якщо їхня кількість дуже велика.

*Count-Min Sketch (CM Sketch)* − це імовірнісна структура даних, яка дає змогу ефективно оцінювати частоти елементів у потоці, використовуючи сублінійний простір.

CM Sketch є двовимірною матрицею (таблицею) лічильників розміром d x w. З нею пов'язано d незалежних хеш-функцій $h_1, h_2, \ldots, h_d$. Кожна хеш-функція $h_j$ відображає елемент потоку в індекс стовпця від 0 до w − 1 для j-го рядка. Параметр d (глибина) – кількість рядків, що відповідає кількості хеш-функцій, w (ширина) − кількість стовпців (лічильників) у кожному рядку.

Операції.

1. Оновлення (update(item, count)) − коли в потоці з'являється елемент item з частотою count (зазвичай count = 1).

1.1. Для кожного рядка j від 1 до d обчислюється індекс стовпця: $col_j = h_j(item)(mod w)$.

1.2. Значення лічильника за цим індексом збільшується на count: matrix[j][col_j] += count.

Час оновлення становить O(d).

2. Оцінка (estimate(item)). Щоб отримати приблизну частоту елемента item:

2.1. Для кожного рядка j від 1 до d обчислюється індекс стовпця: $col_j = h_j(item)(mod w)$.

2.2. Зчитуються значення лічильників matrix[j][col_j] для всіх j.

2.3. Оцінкою частоти є мінімальне з цих d значень.[33]

Час оцінки становить O(d).

Через колізії хеш-функцій один і той самий лічильник може інкрементуватися для різних елементів. Це призводить до того, що оцінка частоти завжди є завищеною (або точною), але ніколи

не є заниженою. Вибір мінімального значення з усіх рядків дає змогу зменшити вплив колізій, оскільки ймовірність того, що один і той самий елемент зіткнеться з «шумними» елементами в усіх d рядках, є низькою.

CM Sketch надає такі гарантії: нехай $a_i$ − справжня частота елемента i, а $a^i$ − його оцінка.

1. $a^i \geq a_i$ (оцінка ніколи не менша за правду).
2. З імовірністю щонайменше 1 − δ, похибка не перевищує $\varepsilon \cdot \|a\|_1$:

$$a^i \leq a_i + \varepsilon \cdot \|a\|_1,$$

де $\|a\|_1$ − сума всіх частот у потоці (загальна кількість елементів), а ε та δ − параметри точності та ймовірності, які задаються користувачем.

Розміри скетчу w та d визначаються через ці параметри:

- $w = \lceil e/\varepsilon \rceil$ (де $e \approx 2.718$);
- $d = \lceil \ln(1/\delta) \rceil$.

Так, простір, необхідний для скетчу, становить $O((1/\varepsilon)\log(1/\delta))$, що є сублінійним і не залежить від кількості унікальних елементів у потоці.

Таблиця 15.1 – Фільтр Блума та Count-Min Sketch

| Характеристика | Фільтр Блума | Count-Min Sketch |
|---|---|---|
| Основне призначення | Перевірка належності до множини | Оцінка частоти елементів |
| Структура даних | Одновимірний бітовий масив | Двовимірна матриця цілочисельних лічильників |
| Результат запиту | «Точно ні» / «Ймовірно так» (булеве значення) | Приблизна кількість (ціле число) |
| Тип помилки | Хибнопозитивні спрацювання (false positives) | Завищена оцінка (overestimation) |
| Неможлива помилка | Хибнонегативні спрацювання (false negatives) | Занижена оцінка (underestimation) |
| Основна операція | check(item) | estimate(item) |
| Типовий випадок використання | Фільтрація запитів до повільного сховища, перевірка унікальності | Пошук «важких гравців» (heavy hitters), аналіз трендів у потоці |

*Псевдокод*

```
class CountMinSketch:
  matrix[d][w]
  hash_functions[d]

  function initialize(d, w):
    matrix = new int[d][w] initialized to 0
    hash_functions = generate_d_hash_functions()

  function update(item, count=1):
    for j from 0 to d-1:
      col = hash_functions[j](item) % w
      matrix[j][col] += count

  function estimate(item):
    min_count = infinity
    for j from 0 to d-1:
      col = hash_functions[j](item) % w
      min_count = min(min_count, matrix[j][col])
    return min_count
```

*Python*

```
import hashlib
import math

class CountMinSketch:
    def __init__(self, epsilon=0.01, delta=0.01):
        self.width = math.ceil(math.e / epsilon)
        self.depth = math.ceil(math.log(1 / delta))
        self.matrix = [ * self.width for _ in range(self.depth)]

    def _hash(self, item, seed):
        # Використовуємо один хеш з різними seed для симуляції кількох
хеш-функцій
        hasher = hashlib.sha256()
        hasher.update(str(seed).encode() + str(item).encode())
        return int(hasher.hexdigest(), 16)

    def update(self, item, count=1):
        for i in range(self.depth):
            col = self._hash(item, i) % self.width
            self.matrix[i][col] += count
```

```
    def estimate(self, item):
        min_count = float('inf')
        for i in range(self.depth):
            col = self._hash(item, i) % self.width
            min_count = min(min_count, self.matrix[i][col])
        return min_count

# Приклад використання
cms = CountMinSketch(epsilon=0.001, delta=0.01)
stream = ["apple", "banana", "apple", "orange", "apple", "banana"]
for item in stream:
    cms.update(item)

print(f"Estimated frequency of 'apple': {cms.estimate('apple')}") # Очікується ~3
print(f"Estimated frequency of 'banana': {cms.estimate('banana')}") # Очікується ~2
print(f"Estimated frequency of 'grape': {cms.estimate('grape')}") # Очікується ~0
```

```
C++
#include<iostream>
#include<vector>
#include<string>
#include<functional>
#include<algorithm>
#include<cmath>

class CountMinSketch {
private:
    int depth;
    int width;
    std::vector<std::vector<int>> matrix;
    std::vector<std::function<size_t(const std::string&)>> hash_functions;

public:
    CountMinSketch(double epsilon, double delta) {
        width = static_cast<int>(ceil(exp(1) / epsilon));
        depth = static_cast<int>(ceil(log(1 / delta)));
        matrix.resize(depth, std::vector<int>(width, 0));
```

```
        // Прості хеш-функції для демонстрації (з різними початковими значеннями)
        for (int i = 0; i < depth; ++i) {
            hash_functions.push_back([this, seed = i * 1000 + 7](const std::string& item) {
                size_t hash = seed;
                for (char c : item) {
                    hash = (hash * 31 + c) % this->width;
                }
                return hash;
            });
        }
    }

    void update(const std::string& item, int count = 1) {
        for (int i = 0; i < depth; ++i) {
            size_t col = hash_functions[i](item);
            matrix[i][col] += count;
        }
    }

    int estimate(const std::string& item) {
        int min_count = std::numeric_limits<int>::max();
        for (int i = 0; i < depth; ++i) {
            size_t col = hash_functions[i](item);
            min_count = std::min(min_count, matrix[i][col]);
        }
        return min_count;
    }
};

int main() {
    CountMinSketch cms(0.001, 0.01);
    std::vector<std::string> stream = {"apple", "banana", "apple", "orange", "apple", "banana"};
    for (const auto& item : stream) {
        cms.update(item);
    }
```

```
        std::cout << "Estimated frequency of 'apple': " <<
cms.estimate("apple") << std::endl;
        std::cout << "Estimated frequency of 'banana': " <<
cms.estimate("banana") << std::endl;
        std::cout << "Estimated frequency of 'grape': " <<
cms.estimate("grape") << std::endl;

        return 0;
    }
```

## 15.3. Суфіксні масиви та дерева

Обробка рядків є центральною задачею в багатьох галузях, від біоінформатики (аналіз ДНК) до повнотекстового пошуку. Для ефективного вирішення таких задач, як пошук підрядка, знаходження найдовшого спільного підрядка або виявлення повторів, були розроблені спеціалізовані структури даних, що індексують усі суфікси заданого рядка. Найвідомішими з них є суфіксні дерева та суфіксні масиви.

*Суфіксне дерево* (Suffix Tree) для рядка S довжиною n − це стиснене префіксне дерево (trie), що містить усі n суфіксів рядка S. Щоб гарантувати, що жоден суфікс не є префіксом іншого, до кінця рядка зазвичай додають унікальний символ-термінатор.

Ключові властивості суфіксного дерева:

1. Має рівно n листків, пронумерованих від 0 до n − 1 (відповідно до початкових індексів суфіксів).
2. Кожен внутрішній вузол (крім кореня) має щонайменше двох нащадків.
3. Ребра позначаються непустими підрядками з S.
4. Жодні два ребра, що виходять з одного вузла, не починаються з однакового символу.

Основна перевага суфіксного дерева полягає в тому, що багато складних рядкових задач можна вирішити за лінійний час після його побудови. Наприклад, пошук підрядка довжиною m у тексті довжиною n виконується за час O(m) незалежно від довжини тексту.

Наївна реалізація, де на кожному ребрі зберігається підрядок, призводить до квадратичної складності за пам'яттю ($O(n^2)$). Ефективна реалізація використовує трюк: замість зберігання самих підрядків, на ребрах зберігаються лише пари індексів (start, end), що вказують на відповідний підрядок у вихідному тексті S. Це дає змогу зменшити складність за пам'яттю до O(n).

На ребрах вказані підрядки, а в листках − початкові індекси відповідних суфіксів.

Алгоритм побудови Укконена. Побудова суфіксного дерева за наївним алгоритмом (додавання кожного суфікса по черзі) займає $O(n^2)$ часу. Алгоритм Укконена − це складний, але ефективний онлайн-алгоритм, що дає змогу побудувати суфіксне дерево за лінійний час O(n) для алфавіту постійного розміру.

Алгоритм працює інкрементально, обробляючи рядок по одному символу і розширюючи «імпліцитне» суфіксне дерево на кожному кроці. Ключовими концепціями, що забезпечують лінійну складність, є:

1. Імпліцитні суфіксні дерева − проміжні дерева, в яких не всі суфікси обов'язково закінчуються в листках.

2. Активна точка (Active Point) − трійка (node, edge, length), що відстежує місце в дереві, де має відбутися наступна вставка.

3. Суфіксні посилання (Suffix Links) − посилання з внутрішнього вузла, що представляє рядок *xa*, до вузла, що представляє рядок *a*. Вони дають змогу швидко переходити між вузлами під час вставки.

4. Три правила розширення − визначають, як додати новий символ до дерева: просто розширити ребро листка, розщепити існуюче ребро або нічого не робити, якщо шлях уже існує.

*Суфіксний масив (Suffix Array)* − це простіша та більш компактна за пам'яттю альтернатива суфіксному дереву. Він являє собою масив цілих чисел, що зберігає початкові позиції всіх суфіксів рядка S у лексикографічному (відсортованому) порядку.

Приклад: для рядка S = "banana$"

Суфікси:

0: banana$

1: anana$

2: nana$
3: ana$
4: na$
5: a$
6: $

Відсортовані суфікси та відповідний суфіксний масив SA:

| **Індекс в SA** | **SA[i]** | **Суфікс** |
|---|---|---|
| 0 | 6 | $ |
| 1 | 5 | a$ |
| 2 | 3 | ana$ |
| 3 | 1 | anana$ |
| 4 | 0 | banana$ |
| 5 | 4 | na$ |
| 6 | 2 | nana$ |

Побудова суфіксного масиву за $O(n\log^2 n)$

Цей алгоритм базується на ідеї «подвоєння префіксів». Сортування відбувається ітеративно.

1. Крок 0 (довжина 1). Суфікси сортуються за першим символом. Кожному суфіксу присвоюється «клас еквівалентності» (ранг). Суфікси з однаковим першим символом отримують однаковий ранг.

2. Крок k (довжина $2^k$). Суфікси сортуються за першими $2^k$ символами. Для цього кожен такий префікс розглядається як пара префіксів довжиною $2^{k-1}$. Оскільки ранги для префіксів довжиною $2^{k-1}$вже обчислені на попередньому кроці, порівняння двох префіксів довжиною $2^k$ зводиться до порівняння двох пар рангів, що виконується за O(1). Сортування n таких пар займає O(nlogn).

3. Процес повторюється logn разів, що дає загальну складність $O(n\log^2 n)$. Якщо на кожному кроці використовувати сортування підрахунком або радіксне сортування, складність можна зменшити до O(nlogn).

Сам по собі суфіксний масив менш функціональний, ніж суфіксне дерево. Його можливості значно розширюються за допомогою масиву LCP (Longest Common Prefix). LCP[i] − це масив,

що зберігає довжину найдовшого спільного префікса між суфіксами, розташованими на позиціях i − 1 та i у відсортованому суфіксному масиві (SA[i − 1] та SA[i]).

Для прикладу S = "banana$":

SA =

LCP = [?, 0, 1, 3, 0, 0, 2]

(LCP не визначений)

- lcp($, a$) = 0
- lcp(a$, ana$) = 1
- lcp(ana$, anana$) = 3
- ... і так далі.

Масив LCP дає змогу ефективно симулювати навігацію по суфіксному дереву і вирішувати багато задач, наприклад, знаходити кількість унікальних підрядків.

*Алгоритм Касаї* дає змогу побудувати масив LCP за лінійний час O(n) за наявності суфіксного масиву та його оберненого масиву (де invSA[i] − позиція суфікса, що починається з індексу i, у SA). Алгоритм базується на ключовому спостереженні: якщо LCP для суфікса S[i..] та його попередника в SA дорівнює h, то LCP для суфікса S[i + 1..] та його попередника в SA буде не менше ніж h − 1. Це дає змогу обчислювати LCP, обходячи суфікси в порядку їх появи в тексті, а не в лексикографічному порядку, що усуває необхідність у повторних порівняннях символів.

Таблиця 15.2 – Суфіксне дерево та суфіксний масив

| Характеристика | Суфіксне дерево | Суфіксний масив (+ LCP) |
|---|---|---|
| Складність за пам’яттю | O(n) слів (більша константа, ~ 10−20 байт / символ) | O(n) цілих чисел (менша константа, ~ 5−9 байт / символ) |
| Час побудови | O(n) (алгоритм Укконена) | O(nlogn) або O(n) (складніші алгоритми) |
| Пошук підрядка (довжина m) | O(m) | O(mlogn) (бінарний пошук) або O(m + logn) (з LCP) |
| Найдовший спільний підрядок | O(n + m) | O(n + m) |
| Складність реалізації | Висока | Середня |
| Локальність кешу | Погана (довільні переходи за пам’яттю) | Відмінна (послідовний доступ до масивів) |

Побудова суфіксного масиву за O(nlog2n)

```
C++
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>

struct Suffix {
   int index;
   int rank;
};

int cmp(struct Suffix a, struct Suffix b) {
   return (a.rank == b.rank)? (a.rank < b.rank? 1 : 0) :
         (a.rank < b.rank? 1 : 0);
}

std::vector<int> buildSuffixArray(const std::string& txt) {
   int n = txt.length();
   struct Suffix suffixes[n];

   for (int i = 0; i < n; i++) {
      suffixes[i].index = i;
      suffixes[i].rank = txt[i];
      suffixes[i].rank = ((i + 1) < n)? (txt[i + 1]) : -1;
   }

   std::sort(suffixes, suffixes + n, cmp);

   std::vector<int> ind(n);
   for (int k = 4; k < 2 * n; k = k * 2) {
      int rank = 0;
      int prev_rank = suffixes.rank;
      suffixes.rank = rank;
      ind[suffixes.index] = 0;

      for (int i = 1; i < n; i++) {
         if (suffixes[i].rank == prev_rank &&
            suffixes[i].rank == suffixes[i - 1].rank) {
            prev_rank = suffixes[i].rank;
```

```
            suffixes[i].rank = rank;
        } else {
            prev_rank = suffixes[i].rank;
            suffixes[i].rank = ++rank;
        }
        ind[suffixes[i].index] = i;
    }

    for (int i = 0; i < n; i++) {
        int nextindex = suffixes[i].index + k / 2;
        suffixes[i].rank = (nextindex < n)?
                        suffixes[ind[nextindex]].rank : -1;
    }
    std::sort(suffixes, suffixes + n, cmp);
  }

  std::vector<int> suffixArr(n);
  for (int i = 0; i < n; i++)
    suffixArr[i] = suffixes[i].index;

  return suffixArr;
}
```

```
Python
def kasai(text, suffix_arr):
    n = len(text)
    lcp =  * n
    inv_suff =  * n

    for i in range(n):
        inv_suff[suffix_arr[i]] = i

    k = 0
    for i in range(n):
        if inv_suff[i] == n - 1:
            k = 0
            continue

        j = suffix_arr[inv_suff[i] + 1]

        while i + k < n and j + k < n and text[i + k] == text[j + k]:
            k += 1
```

```
        lcp[inv_suff[i] + 1] = k

        if k > 0:
            k -= 1

    return lcp
```

## 15.4. Алгоритм Ахо-Корасіка

Задача пошуку одного рядка-зразка в тексті ефективно вирішується за допомогою алгоритмів Кнута-Морріса-Пратта або Боєра-Мура. Однак, коли потрібно одночасно знайти в тексті всі входження з великого набору зразків (словника), наївний підхід − послідовний пошук кожного зразка − стає неефективним. Його складність становить O(k · n + m), де k − кількість зразків, n – довжина тексту, m − сумарна довжина всіх зразків.

*Алгоритм Ахо-Корасіка* вирішує цю задачу за один прохід по тексту з лінійною складністю O(n + m + z), де z − загальна кількість знайдених входжень. Цей алгоритм є основою для багатьох практичних застосувань, від команди fgrep в Unix до систем виявлення вторгнень та антивірусного програмного забезпечення.

Алгоритм використовує скінченний автомат, побудований на основі префіксного дерева (trie) та доповнений спеціальними посиланнями невдач (failure links).

Побудова автомата складається з двох основних етапів.

Етап 1. Побудова префіксного дерева (функція переходів goto).

Спочатку всі зразки зі словника додаються у префіксне дерево. Кожен вузол дерева відповідає певному префіксу одного або кількох зразків. Вузли, що відповідають кінцям зразків, позначаються як «термінальні» або «вихідні», і в них зберігається інформація про те, які саме зразки тут закінчуються.

Етап 2. Додавання посилань невдач (функція failure).

Це ключовий крок, який перетворює префіксне дерево на ефективний автомат. Посилання невдачі для вузла u, що відповідає

рядку w, вказує на вузол v, що відповідає найдовшому власному суфіксу рядка w, який також є префіксом одного зі зразків у словнику.

Призначення посилань невдач: якщо під час руху по тексту з поточного стану автомата немає переходу по наступному символу, замість того, щоб починати пошук з початку, автомат переходить за посиланням невдачі. Це дає змогу миттєво «перескочити» до іншого потенційного збігу, не втрачаючи інформації про вже оброблений суфікс тексту.

Посилання невдач обчислюються за допомогою обходу в ширину (BFS), починаючи з кореня. Для кореня посилання невдачі вказує на самого себе. Для будь-якого іншого вузла u (до якого веде ребро з символом c від батька p) його посилання невдачі обчислюється так: ми йдемо за посиланням невдачі його батька p і звідти намагаємося зробити перехід по символу c. Цей процес повторюється, доки не знайдеться перехід або не буде досягнуто кореня.

Після побудови автомата пошук у тексті T виконується так:

1. Починаємо з кореневого стану автомата (стан 0) і першого символу тексту.

2. Для кожного символу тексту c з поточного стану s:

− намагаємося здійснити перехід за символом c;

− якщо прямий перехід існує, переходимо в новий стан;

− якщо прямого переходу немає, йдемо за посиланням невдачі з поточного стану s і повторюємо спробу переходу за символом c з нового стану. Цей процес триває, доки не буде знайдено перехід або не буде досягнуто кореня.

3. Після кожного переходу в новий стан s′ перевіряємо, чи є він термінальним. Якщо так, ми знайшли збіг.

4. Крім того, потрібно перевірити наявність збігів, слідуючи ланцюжку посилань невдач (або спеціальних «вихідних посилань») з поточного стану s′, оскільки коротші зразки можуть бути суфіксами довшого зразка (наприклад, знаходження "he" за збігу "she").

```
Псевдокод: побудова посилань невдач (BFS)
function build_failure_links(trie):
  queue = new Queue()
  for each child c of root:
    queue.enqueue(c)
    c.failure_link = root

  while queue is not empty:
    current_node = queue.dequeue()
    for each character ch and child v of current_node:
      queue.enqueue(v)
      failure_node = current_node.failure_link
      while failure_node has no transition for ch and failure_node is not root:
        failure_node = failure_node.failure_link

      if failure_node has transition for ch:
        v.failure_link = failure_node.go(ch)
      else:
        v.failure_link = root

      // Об'єднання вихідних даних
      v.output.add(v.failure_link.output)
```

```
Python
from collections import deque

def aho_corasick_automaton(patterns):
    trie = [{'children': {}, 'output':, 'fail': 0}]

    # 1. Побудова префіксного дерева
    for i, pattern in enumerate(patterns):
        node = 0
        for char in pattern:
            if char not in trie[node]['children']:
                trie[node]['children'][char] = len(trie)
                trie.append({'children': {}, 'output':, 'fail': 0})
            node = trie[node]['children'][char]
        trie[node]['output'].append(i)
```

```
    # 2. Побудова посилань невдач (BFS)
    queue = deque()
    for char, next_node in trie['children'].items():
        queue.append(next_node)

    while queue:
        current_node_idx = queue.popleft()
      for char, next_node_idx in trie[current_node_idx]['children'].items():
            queue.append(next_node_idx)
            fail_node_idx = trie[current_node_idx]['fail']
           while char not in trie[fail_node_idx]['children'] and fail_node_idx!= 0:
                fail_node_idx = trie[fail_node_idx]['fail']

            if char in trie[fail_node_idx]['children']:
              trie[next_node_idx]['fail'] = trie[fail_node_idx]['children'][char]

            # Додаємо вихідні дані з посилання невдачі
            fail_output = trie[trie[next_node_idx]['fail']]['output']
            trie[next_node_idx]['output'].extend(fail_output)

    return trie

def search_patterns(text, patterns, trie):
    node = 0
    results =
    for i, char in enumerate(text):
        while char not in trie[node]['children'] and node!= 0:
            node = trie[node]['fail']
        if char in trie[node]['children']:
            node = trie[node]['children'][char]

        if trie[node]['output']:
            for pattern_idx in trie[node]['output']:
                pattern = patterns[pattern_idx]
                results.append((i - len(pattern) + 1, pattern))
    return results

# Приклад
patterns = ["he", "she", "his", "hers"]
text = "ahishers"
trie = aho_corasick_automaton(patterns)
matches = search_patterns(text, patterns, trie)
print(matches) # [(1, 'his'), (3, 'she'), (4, 'he'), (4, 'hers')]
```

```
C++
#include<iostream>
#include<vector>
#include<string>
#include<queue>
#include<map>

const int ALPHABET_SIZE = 26;

struct Node {
    int next;
    int parent;
    char parent_char;
    int link;
    std::vector<int> output_indices;
    int go;

    Node(int p = -1, char ch = '$') : parent(p), parent_char(ch), link(-1) {
        std::fill(next, next + ALPHABET_SIZE, -1);
        std::fill(go, go + ALPHABET_SIZE, -1);
    }
};

std::vector<Node> trie(1);

void add_string(const std::string& s, int pattern_idx) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back(v, ch);
        }
        v = trie[v].next[c];
    }
    trie[v].output_indices.push_back(pattern_idx);
}

int go(int v, char ch);
```

```
int get_link(int v) {
   if (trie[v].link == -1) {
      if (v == 0 |

| trie[v].parent == 0)
         trie[v].link = 0;
      else
         trie[v].link = go(get_link(trie[v].parent), trie[v].parent_char);
   }
   return trie[v].link;
}

int go(int v, char ch) {
   int c = ch - 'a';
   if (trie[v].go[c] == -1) {
      if (trie[v].next[c]!= -1)
         trie[v].go[c] = trie[v].next[c];
      else
         trie[v].go[c] = (v == 0)? 0 : go(get_link(v), ch);
   }
   return trie[v].go[c];
}
```

## 15.5. Зовнішні алгоритми та системи Big Data (MapReduce)

Коли обсяг даних перевищує розмір оперативної пам'яті (RAM), продуктивність системи починає визначатися не швидкістю процесора, а швидкістю доступу до повільних пристроїв зберігання, таких як жорсткі диски (HDD) або твердотільні накопичувачі (SSD). Алгоритми, розроблені для таких умов, називаються *зовнішніми*або *out-of-corealгоритмами*.

Модель зовнішньої пам'яті (I/O Model). Для аналізу зовнішніх алгоритмів використовується модель, яка враховує ієрархію пам'яті. Вона складається з швидкої внутрішньої пам'яті розміром M та повільної, але необмеженої зовнішньої пам'яті. Дані між ними передаються блоками розміром B. Основною метрикою

ефективності є не кількість процесорних операцій, а кількість операцій вводу-виводу (I/O), тобто кількість прочитаних або записаних блоків.

Ефективні зовнішні алгоритми мінімізують кількість I/O, максимально використовуючи дані, що вже містяться в блоці, який завантажено в пам'ять. Класичним прикладом є зовнішнє сортування, яке використовує модифіковане багатопрохідне сортування злиттям для обробки файлів, що не вміщуються в RAM. Його складність I/O становить $O((N/B)\log_{M/B}(N/B))$, що є асимптотично оптимальним.

Для оброблення даних у масштабі Big Data, які не тільки не вміщуються в пам'ять, але й розподілені по сотнях або тисячах машин, була розроблена програмна модель *MapReduce*. Вперше запропонована Google, вона стала основою для екосистеми Apache Hadoop[32].

MapReduce абстрагує від розробника складнощі розподілених обчислень, такі як паралелізація, розподіл даних, балансування навантаження та відмовостійкість. Замість цього розробник має реалізувати лише дві функції: map та reduce.

Основний принцип: «Переміщуй обчислення до даних, а не дані до обчислень». Дані зберігаються в розподіленій файловій системі (наприклад, HDFS), і MapReduce-завдання виконуються на тих самих вузлах, де зберігаються фрагменти даних, що мінімізує мережевий трафік.

Процес виконання MapReduce-завдання складається з трьох основних етапів.

1. Фаза Map:

1.1. Вхідні дані розбиваються на незалежні частини (splits).

1.2. Для кожної частини запускається окремий процес-мапер (mapper).

1.3. Мапер зчитує вхідні дані у вигляді пар <ключ, значення> (наприклад, <номер рядка, текст рядка>) і застосовує до них користувацьку функцію map.

[32] Apache Hadoop – це відкритий (open-source) фреймворк, призначений для зберігання та розподіленої обробки дуже великих обсягів даних на кластерах звичайних комп'ютерів. Він дає змогу обробляти Big Data паралельно, використовуючи тисячі машин, що робить його надзвичайно ефективним для масштабних завдань.

1.4. Функція map генерує нуль або більше проміжних пар <ключ, значення>.

2. Фаза Shuffle and Sort:

2.1. Це «серце» фреймворку, що виконується автоматично між фазами Map та Reduce.

2.2. Shuffle (перетасовування): проміжні пари, згенеровані всіма маперами, збираються і передаються на вузли, де будуть виконуватися редьюсери. Фреймворк гарантує, що всі пари з однаковим ключем потраплять до одного й того самого редьюсера.

2.3. Sort (сортування): на вузлі редьюсера отримані пари сортуються за ключем. Це дає змогу згрупувати всі значення, що належать до одного ключа, разом.

3. Фаза Reduce:

3.1. Для кожного унікального ключа з відсортованих проміжних даних запускається процес-редьюсер (reducer).

3.2. Редьюсер отримує на вхід ключ та ітератор по всіх значеннях, що відповідають цьому ключу (<ключ, список значень>).

3.3. Він застосовує користувацьку функцію reduce для агрегації цих значень (наприклад, підсумовування, знаходження середнього) і генерує кінцевий результат у вигляді нуль або більше пар <ключ, значення>.

Канонічний приклад − Word Count

Задача: підрахувати кількість входжень кожного слова у великому текстовому корпусі.

1. Вхід: текстові файли.

2. Фаза Map: мапер отримує рядки тексту. Для кожного слова в рядку він генерує пару <слово, 1>.

2.1. Вхід: (1, "Hello World Bye World")

2.2. Вихід: ("Hello", 1), ("World", 1), ("Bye", 1), ("World", 1)

3. Фаза Shuffle and Sort: фреймворк збирає виходи всіх маперів і групує їх за ключем (словом).

3.1. Вхід для редьюсерів:

("Bye", )

("Hello", )

("World", )

4. Фаза Reduce: редьюсер для кожного слова підсумовує одиниці в списку значень.

4.1. Вхід: ("Hello", )

4.2. Вихід: ("Hello", 2)

5. Кінцевий результат: файл з парами <слово, загальна_кількість>.

```
Псевдокод для Word Count
function map(document_id, document_text):
  for each word in document_text:
    emit(word, 1)

function reduce(word, counts):
  sum = 0
  for each count in counts:
    sum += count
  emit(word, sum)
```

Hadoop Streaming дає змогу використовувати будь-які виконувані файли як мапери та редьюсери, які читають з stdin і пишуть у stdout.

```
mapper.py
Python
#!/usr/bin/env python
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        # Виводимо пару <слово, 1>, розділену табуляцією
        print(f'{word}\t1')

reducer.py
Python
#!/usr/bin/env python
import sys

current_word = None
current_count = 0
word = None
```

```
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)

    try:
        count = int(count)
    except ValueError:
        continue

    # Hadoop сортує вихід маперів, тому ми отримуємо
    # всі значення для одного ключа послідовно
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print(f'{current_word}\t{current_count}')
        current_count = count
        current_word = word

# Виводимо останнє слово
if current_word == word:
    print(f'{current_word}\t{current_count}')
```

Ці скрипти можна запустити на кластері Hadoop, вказавши їх як мапер та редьюсер.

## 15.6. Приклади розв'язку задач

**Задача 1.** Пошук генетичних маркерів за допомогою алгоритму Ахо-Корасіка. Дано довгу послідовність ДНК (наприклад, T = "GATTACAGATTACA") та набір коротких генетичних маркерів (наприклад, D = {"GATTA", "TACA", "ACA"}). Знайти всі входження цих маркерів у послідовність T.

*Розв'язок*

1. Побудова автомата.

1.1. Префіксне дерево (Trie): Будуємо префіксне дерево для словника D. Вузли, що відповідають кінцям маркерів, позначаються як вихідні.

1.2. Посилання невдач: Обчислюємо посилання невдач для кожного вузла за допомогою BFS.

1.2.1. link(G) -> root

1.2.2. link(GA) -> A

1.2.3. link(GAT) -> T

1.2.4. link(T) -> root

1.2.5. link(TA) -> A

1.2.6. ...тощо.

1.3. Вихідні посилання: для вузла TACA (суфікс ACA) встановлюємо вихідне посилання на вузол ACA.

2. Пошук у тексті T = "GATTACAGATTACA".

2.1. G: Перехід у стан G.

2.2. A: Перехід у стан GA.

2.3. T: Перехід у стан GAT.

2.4. T: Перехід у стан GATT.

2.5. A: Перехід у стан GATTA. Збіг: "GATTA" на позиції 0.

2.6. C: З GATTA немає переходу по C. Йдемо за посиланням невдач link(GATTA) -> A. З A є перехід по C у стан AC. З AC йдемо за link(AC) -> C. У C є вихідне посилання на ACA. Збіг: "ACA" на позиції 4.

2.7. A: З C немає переходу по A. link(C) -> root. З root переходимо в A.

2.8. G: З A немає переходу. link(A) -> root. З root переходимо в G.

2.9. ... і так далі, продовжуючи обробку до кінця рядка.

**Задача 2.** Підрахунок кількості унікальних підрядків за допомогою суфіксного та LCP масивів. Дано рядок S = "banana". Знайти кількість його унікальних підрядків.

*Розв'язок*

Ключова ідея: кожен підрядок є префіксом деякого суфікса. Загальна кількість усіх підрядків (з повтореннями) дорівнює сумі довжин усіх суфіксів. Кількість унікальних підрядків можна знайти, якщо від цієї суми відняти кількість повторюваних префіксів. Масив LCP якраз і зберігає довжини спільних префіксів між сусідніми суфіксами в відсортованому порядку.

1. Побудова суфіксного масиву для S$ = "banana$":

SA =

Суфікси в відсортованому порядку: ($, a$, ana$, anana$, banana$, na$, nana$)

2. Побудова LCP масиву:

LCP = (LCP ігнорується)

3. Обчислення кількості унікальних підрядків:

3.1. Загальна кількість підрядків (префіксів усіх суфіксів):

Довжина banana$ = 6. Довжина anana$ = 5.... Довжина $ = 0.

Сума = 6 + 5 + 4 + 3 + 2 + 1 = 21.

3.2. Сума значень LCP масиву: 1 + 3 + 0 + 0 + 2 = 6.

3.3. Кількість унікальних підрядків = (Загальна кількість підрядків) − (Сума LCP)

21 − 6 = 15.

Унікальні підрядки: a, b, n, an, ba, na, ana, ban, nan, anan, bana, nana, anana, banan, banana.

**Задача 3.** Проєктування MapReduce для розрахунку середньої температури. Є великий набір логів погоди, кожен запис має формат (timestamp, city, temperature). Розробити MapReduce-завдання для обчислення середньої температури для кожного міста.

*Розв'язок*

1. Функція map.

1.1. Вхід:<ключ_рядка, рядок_тексту>, наприклад <101, "2023-10-27T12:00:00, Kyiv,15">.

1.2. Логіка: мапер парсить рядок, витягує назву міста та температуру.

1.3. Вихід: генерує проміжну пару <місто, температура>, наприклад <"Kyiv", 15>.

2. Функція reduce:

2.1. Вхід:<місто, список_температур>, наприклад <"Kyiv", [15, 16, 14, 15.5,...]>.

2.2. Логіка: редьюсер обчислює суму та кількість температур у списку, а потім знаходить середнє значення.

2.3. Вихід: генерує кінцеву пару <місто, середня_температура>, наприклад <"Kyiv", 15.125>.

Цей підхід дає змогу паралельно обробляти мільярди записів логів, розподілених по кластеру, й ефективно агрегувати результати.

## Контрольні питання

1. Який ранг має тензор, що представляє пакет з 50 чорно-білих зображень розміром 128 x 128?
2. Поясніть призначення коефіцієнта затухання в алгоритмі PageRank.
3. Чому фільтр Блума може дати хибнопозитивне спрацювання, але ніколи − хибнонегативне?
4. Як алгоритм Louvain визначає, коли потрібно зупинити ітераційний процес?
5. Який фундаментальний компроміс лежить в основі потокових алгоритмів?
6. Чому для отримання кінцевої оцінки в Count-Min Sketch використовується функція min?
7. Порівняйте складність за пам’яттю суфіксного дерева та суфіксного масиву для рядка довжиною n.
8. Яку проблему вирішує масив LCP під час використання разом із суфіксним масивом?
9. У чому полягає основна перевага алгоритму Укконена для побудови суфіксних дерев?
10. Що таке «посилання невдачі» в алгоритмі Ахо-Корасіка та яке його призначення?
11. Яка основна роль фази "Shuffle and Sort" у моделі MapReduce?
12. Поясніть принцип «переміщення обчислень до даних» у контексті Hadoop та HDFS.
13. У чому полягає різниця між фільтром Блума та Count-Min Sketch з точки зору їхньої основної функції?
14. Як працює механізм трансляції (broadcasting) тензорів? Наведіть простий приклад.
15. Що таке «імпліцитне суфіксне дерево» і як воно використовується в алгоритмі Укконена?

## Задачі для самостійного опрацювання

1. Реалізуйте функцію, що виконує множення двох 2D тензорів (матриць), не використовуючи функції бібліотек машинного навчання.

2. Для невеликого орієнтованого графа вручну обчисліть PageRank кожного вузла для трьох ітерацій з коефіцієнтом затухання 0.85.

3. Спроєктуйте фільтр Блума для 1 мільйона елементів з бажаною ймовірністю хибнопозитивного спрацювання 1 %. Обчисліть необхідний розмір бітового масиву (m) та оптимальну кількість хеш-функцій (k).

4. Ви відстежуєте потік IP-адрес. Розробіть систему з використанням Count-Min Sketch для виявлення IP-адрес, які з'являлися більше ніж 1000 разів.

5. Напишіть програму на Python для побудови суфіксного масиву та LCP масиву для заданого рядка, використовуючи алгоритм складності $O(n\log^2 n)$.

6. Використовуючи суфіксний та LCP масиви з попередньої задачі, напишіть функцію для знаходження найдовшого підрядка, що повторюється, у вихідному рядку.

7. Побудуйте автомат Ахо-Корасіка (префіксне дерево з посиланнями невдач) для словника {"cat", "bat", "at", "car"}.

8. Прослідкуйте процес пошуку тексту "thecatinthebat" за допомогою автомата з попередньої задачі.

9. Напишіть псевдокод для MapReduce-завдання, яке знаходить максимальне значення у дуже великому наборі чисел.

10. Модифікуйте приклад Word Count для MapReduce так, щоб він був нечутливим до регістру символів.

11. Поясніть, як би ви використали узагальнене суфіксне дерево для знаходження найдовшого спільного підрядка між двома рядками, S1 та S2.

12. Система повинна перевіряти, чи бачив користувач певну статтю. Кількість статей дуже велика. Що буде доцільніше використати: фільтр Блума чи хеш-таблицю? Обґрунтуйте відповідь з точки зору використання пам'яті.

13. Реалізуйте першу фазу алгоритму Louvain (оптимізацію модулярності) для заданого зваженого графа.

14. Маючи потік пар (user_id, item_id), що представляють покупки, як можна використати Count-Min Sketch для знаходження топ-10 найбільш купованих товарів? Які обмеження має такий підхід?

15. Напишіть клас на C++ для тензора, що підтримує поелементне додавання з іншим тензором такої самої форми. Дані тензора повинні зберігатися в «пласкому» std::vector.

# ТЕМА 16
# ТЕОРІЯ СКЛАДНОСТІ

У попередніх розділах ми зосереджувалися на аналізі ефективності конкретних алгоритмів, використовуючи такі інструменти, як О-нотація, для оцінки їхньої часової та просторової складності. Ми навчилися розрізняти алгоритми, що виконуються за час O(nlogn) та $O(n^2)$, і розуміти практичні наслідки цього розрізнення. Однак теорія алгоритмів виходить за межі аналізу окремих рішень. Вона ставить більш фундаментальні питання про саму природу обчислювальних проблем.

Теорія складності класифікує проблеми на основі ресурсів, потрібних для їх розв'язання, і має відповісти на два ключові питання:

1. Обчислюваність. Які проблеми взагалі можливо розв'язати за допомогою алгоритму? Чи існують коректно поставлені задачі, для яких в основному не існує алгоритмічного розв'язку, незалежно від того, наскільки потужним є наш комп'ютер?

2. Складність. Якщо проблема є розв'язною, то наскільки важко знайти її розв'язок? Чи можемо ми класифікувати проблеми на «легкі» (ті, що розв'язуються ефективно) та «важкі» (ті, для яких відомі лише неефективні, перебірні методи)?

## 16.1. Межі обчислюваності. Розв'язні, напівроз в'язні та нерозв'язні проблеми

Перш ніж аналізувати, наскільки *швидко* можна розв'язати проблему, потрібно встановити, чи можна її розв'язати *взагалі*. Теорія обчислюваності формалізує це питання за допомогою математичної моделі обчислень − машини Тюрінга. У контексті цієї моделі всі обчислювальні проблеми (які зазвичай формулюються як задачі розпізнавання мов) можна розділити на три фундаментальні категорії.

Проблема називається *розв'язною* (або *рекурсивною*), якщо існує алгоритм (машина Тюрінга), який для будь-яких вхідних даних гарантовано зупиняється за скінченний час і дає правильну відповідь: «так» або «ні».

Це клас проблем, для яких ми можемо написати програму, що завжди завершує свою роботу з коректним результатом. Більшість проблем, з якими ви стикаєтесь у повсякденному програмуванні, належать до цього класу.

Приклади розв'язних проблем.

Сортування масиву. Для будь-якого масиву чисел існує безліч алгоритмів (наприклад, сортування злиттям), які гарантовано завершаться і повернуть відсортований масив.

Перевірка на простоту. Для будь-якого цілого числа $n$ існує алгоритм, який визначить, чи є $n$ простим.

Пошук найкоротшого шляху в графі. Алгоритм Дейкстри гарантовано знаходить найкоротший шлях у графі з невід'ємними вагами ребер за скінченний час.

Перевірка належності рядка мові, що генерується контекстно-вільною граматикою (ACCEPTCFG). Існують алгоритми (наприклад, алгоритм Кока-Янгера-Касамі), які можуть визначити, чи належить такий рядок мові, описаній цією граматикою.

Проблема називається *напіврозв'язною* (або *рекурсивно перелічуваною*), якщо існує алгоритм, який для будь-яких вхідних даних, що є «так»-випадком проблеми, зупиняється і дає відповідь «так». Однак для «ні»-випадків алгоритм може або зупинитися і дати відповідь «ні», або працювати нескінченно (зациклитися).

Ключова відмінність полягає в асиметрії: ми можемо бути впевнені в позитивній відповіді, але ніколи не можемо бути остаточно впевнені в негативній, оскільки програма може просто ще не завершила свою роботу.

Приклад напіврозв'язної проблеми − проблема зупинки (Halting Problem). Як ми детально розглянемо в наступному підрозділі, ця проблема є канонічним прикладом. Ми можемо написати симулятор, який запустить таку програму на цих вхідних даних. Якщо програма зупиниться, наш симулятор також зупиниться і дасть відповідь «так». Але якщо програма зациклюється, наш симулятор також буде працювати нескінченно.

Проблема називається *нерозв'язною*, якщо для неї не існує алгоритму, який би гарантовано зупинявся для всіх можливих вхідних даних і давав правильну відповідь.

Це проблеми, які лежать за межами можливостей алгоритмічного розв'язання. Важливо розуміти, що це не означає, що окремі випадки таких проблем не можна розв'язати. Це означає, що не існує *єдиного універсального* алгоритму, який би працював для всіх можливих входів.

Приклади нерозв'язних проблем:

Проблема зупинки є нерозв'язною, але напіврозв'язною.

Проблема еквівалентності машин Тюрінга: визначити, чи дві машини Тюрінга приймають одну й ту саму мову. Ця проблема навіть не є напіврозв'язною.

Виявлення «мертвого коду» − не існує загального алгоритму, який міг би для будь-якої програми визначити, які її частини коду ніколи не будуть виконані.

Наведена нижче таблиця підсумовує ключові відмінності між цими класами.

Таблиця 16.1 – Порівняльна характеристика класів розв'язності

| Властивість | Розв'язні (Decidable) | Напіврозв'язні (Semi-Decidable) | Нерозв'язні (Undecidable) |
|---|---|---|---|
| Поведінка на «так»-вході | Зупиняється і приймає | Зупиняється і приймає | Може зупинитись, може зациклитись |
| Поведінка на «ні»-вході | Зупиняється і відхиляє | Може зупинитись і відхилити, або зациклитись | Може зупинитись, може зациклитись |
| Можливість підтвердити «так» | Так, за скінченний час | Так, за скінченний час | Не завжди |
| Можливість підтвердити «ні» | Так, за скінченний час | Ні, не гарантовано | Не завжди |
| Формальна назва мови | Рекурсивна (Recursive) | Рекурсивно перелічувана (Recursively Enumerable) | Не рекурсивно перелічувана |

Існує важливий теоретичний зв'язок між цими класами: проблема є розв'язною тоді і тільки тоді, коли і сама проблема, і її доповнення (де відповіді «так» і «ні» міняються місцями) є напівроз'язними. Якщо ми маємо алгоритм, що перелічує всі «так»-випадки, і інший алгоритм, що перелічує всі «ні»-випадки, ми можемо запустити їх паралельно. Для будь-яких вхідних даних один з них гарантовано зупиниться, що дає нам повний розв'язок.

## 16.2. Проблема зупинки

Проблема зупинки є однією з найвідоміших і фундаментальних проблем у теорії обчислюваності. Вона була сформульована і доведена як нерозв'язна Аланом Тюрінгом 1936 року, і її результат має глибокі наслідки для всієї галузі комп'ютерних наук.

Формальне визначення проблеми. Дано опис довільного алгоритму (програми) P та його вхідні дані I. Потрібно визначити, чи завершить програма P свою роботу на вхідних даних I за скінченний час (зупиниться), чи буде працювати нескінченно (зациклиться).

На перший погляд, задача може здатися вирішуваною. Чому б просто не запустити програму P на даних I і не почекати? Якщо вона зупиниться, ми отримаємо відповідь «так». Проблема виникає, якщо програма не зупиняється. Ми ніколи не можемо бути впевнені, чи вона дійсно зациклилася, чи просто виконує дуже довгі обчислення і зупиниться в майбутньому. Саме тому проблема зупинки є напівроз'язною: ми можемо підтвердити «так», але не можемо гарантовано підтвердити «ні».

Доведення нерозв'язності від супротивного. Доведення нерозв'язності проблеми зупинки є класичним прикладом використання методу доведення від супротивного, який також відомий як діагоналізація.

1. Припущення. Припустимо, що проблема зупинки є розв'язною. Це означає, що існує універсальний алгоритм (функція), назвемо його halts(program, input), який приймає опис програми

та її вхідні дані і завжди повертає true, якщо program зупиняється на input, і false в іншому випадку. Ця функція сама по собі завжди повинна зупинятися.

2. Побудова парадоксальної програми. Використовуючи нашу гіпотетичну функцію halts, ми можемо сконструювати нову, парадоксальну програму, назвемо її paradox. Ця програма приймає на вхід опис іншої програми p і робить таке:

1) Вона викликає halts(p, p), тобто перевіряє, чи зупиниться програма p, якщо подати їй на вхід її власний опис.

2) Якщо halts(p, p) повертає true (тобто p зупиняється на власному описі), то програма paradox навмисно входить у нескінченний цикл.

3) Якщо halts(p, p) повертає false (тобто p зациклюється на власному описі), то програма paradox негайно зупиняється.

Іншими словами, paradox робить прямо протилежне тому, що робить її вхідна програма зі своїм власним кодом.

3. Суперечність. Тепер постає ключове питання: що станеться, якщо ми запустимо програму paradox, подавши їй на вхід її власний опис? Тобто що станеться під час виклику paradox(paradox)?

*Випадок 1.* Припустимо, paradox(paradox) зупиняється. Якщо вона зупиняється, це означає, що наш аналізатор halts(paradox, paradox) повинен повернути true.Але, згідно з визначенням програми paradox, якщо halts повертає true, то paradox повинна увійти в нескінченний цикл. Це суперечить нашому припущенню, що paradox(paradox) зупиняється.

*Випадок 2.* Припустимо, paradox(paradox) зациклюється. Якщо вона зациклюється, це означає, що наш аналізатор halts(paradox, paradox) повинен повернути false. Але, згідно з визначенням програми paradox, якщо halts повертає false, то paradox повинна негайно зупинитися. Це суперечить нашому припущенню, що paradox(paradox) зациклюється.

В обох можливих випадках ми приходимо до логічної суперечності. Це означає, що наше початкове припущення про існування універсальної функції halts є хибним. Отже, проблема зупинки є алгоритмічно нерозв’язною.

Приклад. Парадокс програми-аналізатора. Хоча ми не можемо реалізувати функцію halts, ми можемо написати код, що ілюструє логіку парадоксальної функції.

```
Псевдокод
function paradox(program_description p):
  if halts(p, p) == true:
    loop forever
  else:
    return "halted"
```

```
Python
# Гіпотетична функція, існування якої ми спростовуємо.
# В реальному коді її реалізувати неможливо.
def halts(program_string, input_string):
  #... магічна реалізація...
  pass

def paradox(program_string):
  """
  Ця функція ілюструє логічний парадокс.
  Вона аналізує, чи зупиниться програма, подана як рядок,
  на власному коді, і робить протилежне.
  """
  if halts(program_string, program_string):
    while True:
      pass  # нескінченний цикл
  else:
    return "halted"

# Суперечність виникає під час спроби аналізувати саму себе:
# paradox_code_as_string = "..." # рядок, що містить код функції paradox
```

```
C++
#include <iostream>

// Гіпотетична функція, існування якої ми спростовуємо.
// В реальному коді її реалізувати неможливо.
bool halts(const std::string& program_string, const std::string& input_string) {
```

```
    //... магічна реалізація...
    return false;
}

void paradox(const std::string& program_string) {
    /*
    Ця функція ілюструє логічний парадокс.
    Вона аналізує, чи зупиниться програма, подана як рядок,
    на власному коді, і робить протилежне.
    */
    if (halts(program_string, program_string)) {
        while (true) {
            // нескінченний цикл
        }
    } else {
        // зупинка
        return;
    }
}

int main() {
    // Суперечність виникає при спробі аналізувати саму себе:
    // std::string paradox_code_as_string = "..."; // рядок, що містить код функції paradox
    // paradox(paradox_code_as_string);
    return 0;
}
```

Наслідки проблеми зупинки. Нерозв'язність проблеми зупинки – це не просто теоретична абстракція. Вона встановлює фундаментальні обмеження на те, що можуть робити комп'ютери. Наприклад, неможливо створити:

1. Ідеальний компілятор, який би міг виявляти всі нескінченні цикли в будь-якій програмі під час компіляції.

2. Універсальний верифікатор програм, який би міг автоматично довести, що будь-яка програма є коректною для всіх можливих вхідних даних.

3. Загальний аналізатор вірусів, який би міг гарантовано визначити, чи є довільний фрагмент коду шкідливим, не запускаючи його.

Проблема зупинки є «еталоном» нерозв'язності. Багато інших проблем доводяться як нерозв'язні через *зведення* до неї. Тобто доводиться, що якби ми могли розв'язати якусь нову проблему X, ми б також могли розв'язати проблему зупинки. Оскільки останнє неможливо, то і проблема X є нерозв'язною.

## 16.3. Класи часової складності: P та NP

Після того, як ми встановили, що проблема є розв'язною, наступне логічне питання − наскільки ефективно її можна розв'язати. Теорія складності класифікує розв'язні проблеми на основі ресурсів (найчастіше − часу), потрібних для їх розв'язання на детермінованій машині Тюрінга, що є моделлю звичайного комп'ютера. Межею між «ефективними» та «неефективними» алгоритмами умовно вважається межа між поліноміальним та експоненціальним часом виконання.

Клас P (Polynomial Time). Клас P − це множина всіх задач розпізнавання, які можна розв'язати за поліноміальний час на детермінованій машині Тюрінга.

Алгоритм працює за поліноміальний час, якщо його часова складність обмежена зверху поліномом від розміру вхідних даних n, тобто $T(n) = O(n^k)$ для деякої константи k. Проблеми з класу P вважаються трактованими або ефективно розв'язними. Хоча алгоритм зі складністю $O(n^{100})$ формально є поліноміальним, на практиці він може бути непридатним. Однак більшість важливих поліноміальних алгоритмів мають значно менший степінь ($k \leq 4$).

Приклади задач з класу P:

1. Сортування. Алгоритм сортування злиттям має складність O(nlogn), що є поліноміальною.

2. Пошук у масиві. Лінійний пошук (O(n)) та бінарний пошук (O(logn)) є поліноміальними.

3. Знаходження найбільшого спільного дільника. Алгоритм Евкліда працює за поліноміальний час.

4. Перевірка зв'язності графа. Пошук у ширину або в глибину розв'язує цю задачу за час O(V + E), де V − кількість вершин, E − кількість ребер.

Клас NP (Nondeterministic Polynomial Time). Клас NP є одним з найважливіших і найцікавіших у теорії складності. Його назва «Недетермінований поліноміальний час» часто призводить до плутанини. Важливо розуміти, що NP не означає «не поліноміальний». Існує два еквівалентних означення цього класу.

Означення 1 (Через верифікацію). Клас NP − це множина задач розпізнавання, для яких, якщо відповідь «так», існує «сертифікат» (або «свідок»), за допомогою якого можна перевірити правильність цієї відповіді за поліноміальний час на детермінованій машині.

Іншими словами, хоча знайти розв'язок може бути важко, перевірити готовий розв'язок − легко.

Приклад (Задача про гамільтонів цикл):

1. Задача: Чи існує в цьому графі G цикл, що проходить через кожну вершину рівно один раз?

2. Знаходження розв'язку: невідомо ефективного алгоритму. Для великих графів повний перебір усіх можливих шляхів є обчислювально неможливим.

3. Перевірка розв'язку: якщо хтось запропонує вам послідовність вершин, яка, на його думку, є гамільтоновим циклом (це і є «сертифікат»), ви можете дуже швидко (за поліноміальний час) перевірити, чи це правда: 1) чи всі вершини графа присутні в послідовності рівно один раз, і 2) чи існує ребро між кожною парою сусідніх вершин у послідовності.

Оскільки перевірка є легкою, задача про гамільтонів цикл належить до класу NP.

Означення 2 (Через недетермінізм). Клас NP − це множина задач розпізнавання, які можна розв'язати за поліноміальний час на недетермінованій машині Тюрінга.

Недетермінована машина Тюрінга − це теоретична модель, яка на кожному кроці може «вгадати» правильний вибір з кількох можливих і розгалужити обчислення. Якщо хоча б одна гілка обчислень приводить до відповіді «так» за поліноміальний час, задача вважається розв'язаною. Послідовність «правильних вгадувань» і є тим самим сертифікатом з першого означення.

Проблема рівності класів P та NP. З означень випливає, що будь-яка задача з класу P також належить до класу NP. Якщо ми можемо розв'язати задачу за поліноміальний час, то ми точно можемо і перевірити її розв'язок (просто розв'язавши її знову і порівнявши результати). Так, $P \subseteq NP$.

Основне відкрите питання теорії складності, за розв'язання якого Інститут Клея пропонує премію в мільйон доларів, звучить так: Чи дорівнює P класу NP? (P = NP?).

Якщо P = NP, це означатиме, що будь-яка задача, розв'язок якої можна швидко перевірити, також можна і швидко знайти. Це мало б революційні наслідки для науки, економіки, криптографії та багатьох інших галузей. Багато складних оптимізаційних задач (логістика, розробка ліків, фінансове моделювання) стали б ефективно розв'язними.

Якщо P = NP (що вважається більш імовірним серед більшості вчених), це означатиме, що існують задачі, які є принципово «важчими» для розв'язання, ніж для перевірки. Це підтверджує наш інтуїтивний досвід, що творчий процес знаходження розв'язку (наприклад, доведення теореми) є складнішим, ніж перевірка готового доведення.

Наведена нижче таблиця підсумовує порівняння цих двох класів.

Таблиця 16.2 – Порівняння класів P та NP

| Характеристика | Клас P | Клас NP |
|---|---|---|
| Повна назва | Поліноміальний час (Polynomial time) | Недетермінований поліноміальний час (Nondeterministic Polynomial time) |
| Ключова ідея | Задачу можна швидко розв'язати | Запропонований розв'язок можна швидко перевірити |
| Модель обчислень | Детермінована машина Тюрінга | Недетермінована машина Тюрінга (або детермінований «верифікатор») |
| Приклад задачі | Сортування масиву, Пошук найкоротшого шляху | Цикл Гамільтона, Задача комівояжера (версія для прийняття рішень) |
| Складність | Вважаються «ефективно розв'язними» | Можливо, «важкі» для розв'язання, але «легкі» для перевірки |

## 16.4. NP-повнота та NP-важкість

Всередині класу NP існує підклас задач, які вважаються «найважчими» в NP. Для формального визначення цих задач нам потрібне поняття поліноміального зведення.

Кажуть, що задача A *поліноміально зводиться* до задачі B (позначається як $A \le pB$), якщо існує алгоритм, що працює за поліноміальний час і перетворює будь-який екземпляр x задачі A на екземпляр y задачі B так, що відповідь на x є «так» тоді і тільки тоді, коли відповідь на y є «так».

Зведення − це формальний спосіб сказати, що «задача B є щонайменше такою ж складною, як задача A». Якщо ми маємо ефективний алгоритм для розв'язання B, то ми можемо ефективно розв'язати й A: спочатку звести A до B (за поліноміальний час), а потім розв'язати отриманий екземпляр B (також за поліноміальний час).

Задача H називається *NP-важкою* (NP-hard), якщо будь-яка задача L з класу NP поліноміально зводиться до H ($L \le pH$ для всіх $L \in NP$).

NP-важкі задачі є «найскладнішими» в тому сенсі, що вони щонайменше такі ж складні, як і будь-яка задача з NP. Важливо, що NP-важка задача необов'язково повинна належати до класу NP. Наприклад, проблема зупинки є NP-важкою, але вона не в NP, оскільки є нерозв'язною.

Задача C називається *NP-повною* (NP-complete, NPC), якщо вона задовольняє дві умови 17:

1. $C \in NP$ (розв'язок можна швидко перевірити).
2. C є NP-важкою (будь-яка задача з NP зводиться до неї).

NP-повні задачі − це найважчі задачі всередині класу NP. Вони мають велике практичне та теоретичне значення, оскільки є «універсальними» для всього класу NP. Якщо вдасться знайти поліноміальний алгоритм хоча б для однієї NP-повної задачі, то, завдяки властивості зведення, всі задачі з класу NP можна буде розв'язати за поліноміальний час, що означатиме P = NP.

Як довести, що задача є NP-важкою, якщо для цього потрібно показати зведення від *усіх* задач з NP? Цю проблему розв'язала теорема, доведена незалежно Стівеном Куком та Леонідом Левіним у 1970-х роках.

Теорема Кука-Левіна. Задача здійсненності булевих формул (SAT) є NP-повною.

Ця теорема стала наріжним каменем теорії NP-повноти. Вона надала першу «точку опори»: тепер, щоб довести NP-повноту нової задачі Z, не потрібно зводити до неї всі задачі з NP. Достатньо показати, що $Z \in NP$ і звести до неї *будь-яку* вже відому NP-повну задачу (наприклад, SAT). Завдяки транзитивності зведення ($A \le pB$ і $B \le pC \Longrightarrow A \le pC$), цього буде достатньо, щоб довести NP-важкість Z.

Приклади відомих NP-повних задач:

1. Задача здійсненності (SAT) та її варіант 3-SAT.
2. Задача про гамільтонів цикл.
3. Задача комівояжера (у версії для прийняття рішень).
4. Задача про вершинне покриття (Vertex Cover).
5. Задача про кліку (Clique).
6. Задача про розфарбовування графа (Graph Coloring).

Знання того, що задача є NP-повною, є важливим практичним результатом. Воно сигналізує розробнику, що пошук ефективного точного алгоритму, ймовірно, є марною тратою часу, і варто зосередитися на альтернативних підходах, таких як евристики, апроксимаційні алгоритми або розв'язання задачі для окремих випадків.

## 16.5. Методи доведення NP-повноти

Доведення того, що нова задача є NP-повною, є стандартною процедурою в теорії алгоритмів, яка дає змогу класифікувати її обчислювальну складність. Цей процес майже завжди складається з двох кроків, що спираються на вже відомі NP-повні задачі.

Стандартна процедура доведення NP-повноти задачі Z:

1. Довести, що $Z \in NP$. Для цього потрібно показати, що для будь-якого «так»-випадку задачі Z існує сертифікат, який можна перевірити на коректність за поліноміальний час. Зазвичай цей крок є відносно простим.

2. Довести, що Z є NP-важкою. Для цього потрібно вибрати відому NP-повну задачу $Z'$ і побудувати поліноміальне зведення $Z' \leq pZ$. Це означає створення алгоритму, який за поліноміальний час перетворює будь-який екземпляр задачі $Z'$ на екземпляр задачі Z, зберігаючи при цьому відповідь («так» / «ні»).

Цей другий крок є найскладнішою та найкреативнішою частиною доведення.

Детальний приклад доведення: зведення 3-SAT до задачі про вершинне покриття (Vertex Cover). Продемонструємо цю процедуру на класичному прикладі, довівши, що задача про вершинне покриття є NP-повною.

Означення задачі 3-SAT:

1. Вхід: булева формула $\phi$ у кон'юнктивній нормальній формі, де кожен диз'юнкт (клауза) містить рівно три літерали. Літерал − це змінна або її заперечення (наприклад, xi або ¬xi).

2. Питання: чи існує такий набір значень true / false для змінних, за якого формула $\phi$ стає істинною?

Означення задачі про вершинне покриття (Vertex Cover, VC):

1. Вхід: неорієнтований граф G = (V, E) та ціле число k.

2. Питання: чи існує в графі G підмножина вершин $V' \subseteq V$ розміром $|V'| \leq k$ така, що кожне ребро з E має хоча б один кінець у $V'$?

Доведення NP-повноти Vertex Cover:

Крок 1: Довести, що $VC \in NP$.

Це просто. Сертифікатом для «так»-випадку є сама множина вершин $V'$. Щоб перевірити її, потрібно виконати дві дії:

1. Перевірити, чи $|V'| \leq k$. Це займає час $O(|V'|)$.

2. Для кожного ребра $(u,v) \in E$ перевірити, чи $u \in V'$ або $v \in V'$. Це займає час $O(|E|)$.

Обидві перевірки виконуються за поліноміальний час, отже, $VC \in NP$.

Крок 2: Довести, що VC є NP-важкою за рахунок зведення $3 - SAT \leq pVC$.

Ми повинні побудувати алгоритм, який перетворює будь-яку формулу 3-SAT $\phi$ на пару (G,k) для задачі VC.

Побудова зведення: нехай формула ϕ має l змінних (x1,...,xl) та m клауз (C1,...,Cm). Ми будуємо граф G так:

1. «Гаджети» для змінних (Variable Gadgets): для кожної змінної xi створюємо дві вершини, з'єднані ребром: одну для літерала xi та іншу для ¬xi. Назвемо ці вершини vi, true та vi, false. Ребро між ними гарантує, що будь-яке вершинне покриття повинно буде в долучити хоча б одну з цих двох вершин, що відповідає присвоєнню змінній значення true або false.

2. «Гаджети» для клауз (Clause Gadgets): для кожної клаузи Cj = (a∨b∨c), де a, b, c − літерали, створюємо три нові вершини cj1, cj2, cj3. З'єднуємо їх між собою, утворюючи трикутник. Це гарантує, що для покриття трьох внутрішніх ребер цього трикутника будь-яке вершинне покриття повинно буде взяти щонайменше дві з цих трьох вершин.

3. З'єднання гаджетів: з'єднуємо гаджети клауз з гаджетами змінних. Для кожної клаузи Cj = (a∨b∨c) з'єднуємо:

1) Вершину cj1 з вершиною в гаджеті змінних, що відповідає літералу a.

2) Вершину cj2 з вершиною, що відповідає літералу b.

3) Вершину cj3 з вершиною, що відповідає літералу c.

4. Визначення k:встановлюємо цільовий розмір покриття $k = l + 2m$.

Ця конструкція виконується за поліноміальний час, оскільки кількість вершин у G дорівнює 2l + 3m, а кількість ребер також є поліномом від l та m.

Доведення коректності зведення. Тепер ми повинні показати, що формула ϕ є здійсненною тоді і тільки тоді, коли граф G має вершинне покриття розміром k.

(⇒) Якщо ϕ здійсненна, то існує VC розміром k.

Припустимо, існує набір значень, що задовольняє ϕ. Побудуємо покриття V′:

1. Для кожної змінної xi: якщо xi є true, додаємо вершину vi, true до V′. Якщо false − додаємо vi, false. Це додає рівно l вершин до V′ і покриває всі ребра всередині гаджетів змінних.

2. Розглянемо будь-яку клаузу Cj = (a∨b∨c). Оскільки формула здійсненна, хоча б один з її літералів є істинним. Нехай це

буде літерал a. Тоді відповідна вершина для a вже є в нашому покритті V′. Це означає, що ребро, яке з'єднує цю вершину з гаджетом клаузи (з вершиною cj1), вже покрите. Щоб покрити два ребра, що залишилися всередині трикутника ( (c_j2, c_j3) та (c_j1, c_j2) або (c_j1, c_j3) ), нам достатньо додати до V′ дві вершини, що не з'єднані з істинним літералом, наприклад, cj2 та cj3.

3. Ми робимо це для кожної з m клауз, додаючи по 2 вершини з кожного гаджета клаузи.

Загальний розмір покриття буде l (від змінних) + 2m (від клауз) = k. Отже, ми побудували покриття потрібного розміру.

(⇐) Якщо існує VC розміром k, то ϕ здійсненна.

Припустимо, існує покриття V′ розміром k = l + 2m.

1. Щоб покрити l ребер у гаджетах змінних, V′ повинна містити щонайменше l вершин − по одній з кожної пари (vi, true, vi, false).

2. Щоб покрити 3m ребер всередині m трикутників у гаджетах клауз, V′ повинна містити щонайменше 2m вершин.

3. Оскільки загальний розмір покриття рівно l + 2m, то V′ повинна містити *рівно* одну вершину з кожного гаджета змінних і *рівно* дві вершини з кожного гаджета клауз.

4. Побудуємо набір значень для змінних: якщо vi, true∈V′, то xi = true. Якщо vi, false∈V′, то xi = false. Оскільки в V′ є рівно одна вершина з кожної пари, це присвоєння є несуперечливим.

5. Розглянемо будь-який гаджет клаузи Cj. У ньому V′ містить дві вершини. Нехай це будуть cj1 та cj2. Третя вершина, cj3, не входить до V′. Це означає, що три ребра, інцидентні cj3 (два внутрішні та одне зовнішнє), повинні бути покриті її сусідами. Зокрема, ребро, що з'єднує cj3 з відповідним літералом (назвемо його c) у гаджеті змінних, повинно бути покрите. Оскільки cj3∈/V′, то вершина для літерала c *повинна* бути у V′. Згідно з нашим правилом, це означає, що літерал c є істинним, і, отже, клауза Cj є істинною.

6. Оскільки це правильно для кожної клаузи, вся формула ϕ є здійсненною.

Ми показали, що ϕ є здійсненною ⇔ G має VC розміром k. Так, ми звели 3-SAT до VC, довівши, що VC є NP-важкою. Оскільки VC також в NP, задача про вершинне покриття є NP-повною.

Приклад. Побудова графа для формули 3-SAT
Розглянемо формулу ϕ = (x1 ∨ ¬ x2 ∨ ¬x3) ∧ (¬x1 ∨ x2 ∨ x4).
Тут l = 4 змінні, m = 2 клаузи. K = l + 2m = 4 + 2∗2 = 8.

```
Псевдокод для побудови графа
function build_vc_graph(formula):
  G = new Graph()
  l = number_of_variables(formula)
  m = number_of_clauses(formula)

  // Створення гаджетів змінних
  for i from 1 to l:
    add_vertex(G, "x" + i)
    add_vertex(G, "¬x" + i)
    add_edge(G, "x" + i, "¬x" + i)

  // Створення гаджетів клауз та з'єднань
  for j from 1 to m:
    clause = formula.clauses[j]
    add_vertex(G, "c" + j + "1")
    add_vertex(G, "c" + j + "2")
    add_vertex(G, "c" + j + "3")
    add_edge(G, "c" + j + "1", "c" + j + "2")
    add_edge(G, "c" + j + "2", "c" + j + "3")
    add_edge(G, "c" + j + "3", "c" + j + "1")

    connect("c" + j + "1", clause.literal1, G)
    connect("c" + j + "2", clause.literal2, G)
    connect("c" + j + "3", clause.literal3, G)

  k = l + 2*m
  return (G, k)
```

```
Python
def build_vc_instance_from_3sat(formula, num_vars):
    """
    Перетворює екземпляр 3-SAT на екземпляр Vertex Cover.
    formula: список клауз, де кожна клауза - це список з 3 літералів.
             Позитивне число і означає x_i, негативне -і означає ¬x_i.
    num_vars: кількість змінних.
    """
    graph = {}
```

```
    def add_edge(u, v):
        if u not in graph: graph[u] =
        if v not in graph: graph[v] =
        graph[u].append(v)
        graph[v].append(u)

    # Гаджети змінних
    for i in range(1, num_vars + 1):
        add_edge(f"x{i}", f"~x{i}")

    # Гаджети клауз
    for j, clause in enumerate(formula):
        c_nodes = [f"c{j}_1", f"c{j}_2", f"c{j}_3"]
        add_edge(c_nodes, c_nodes)
        add_edge(c_nodes, c_nodes)
        add_edge(c_nodes, c_nodes)

        for i, literal in enumerate(clause):
            var_node = f"x{abs(literal)}" if literal > 0 else f"~x{abs(literal)}"
            add_edge(c_nodes[i], var_node)

    k = num_vars + 2 * len(formula)
    return graph, k

# Приклад: (x1 ∨ ¬x2 ∨ ¬x3) ∧ (¬x1 ∨ x2 ∨ x4)
formula_example = [[1, -2, -3], [-1, 2, 4]]
num_vars_example = 4
g, k = build_vc_instance_from_3sat(formula_example, num_vars_example)
print(f"Target k: {k}")
# print(g) # Виведе список суміжності графа
```

```
C++
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <cmath>

using namespace std;
```

```
        // Представлення графа у вигляді списку суміжності
        using Graph = map<string, vector<string>>;

        void add_edge(Graph& g, const string& u, const string& v) {
            g[u].push_back(v);
            g[v].push_back(u);
        }

        pair<Graph, int> build_vc_instance_from_3sat(const
vector<vector<int>>& formula, int num_vars) {
            Graph g;

            // Гаджети змінних
            for (int i = 1; i <= num_vars; ++i) {
                add_edge(g, "x" + to_string(i), "~x" + to_string(i));
            }

            // Гаджети клауз
            for (int j = 0; j < formula.size(); ++j) {
                string c1 = "c" + to_string(j) + "_1";
                string c2 = "c" + to_string(j) + "_2";
                string c3 = "c" + to_string(j) + "_3";
                add_edge(g, c1, c2);
                add_edge(g, c2, c3);
                add_edge(g, c3, c1);

                const auto& clause = formula[j];
                string l1_node = (clause > 0)? "x" + to_string(abs(clause)) : "~x" +
to_string(abs(clause));
                string l2_node = (clause > 0)? "x" + to_string(abs(clause)) : "~x" +
to_string(abs(clause));
                string l3_node = (clause > 0)? "x" + to_string(abs(clause)) : "~x" +
to_string(abs(clause));

                add_edge(g, c1, l1_node);
                add_edge(g, c2, l2_node);
                add_edge(g, c3, l3_node);
            }

            int k = num_vars + 2 * formula.size();
            return {g, k};
        }
```

```
int main() {
    // Приклад: (x1 ∨ ¬x2 ∨ ¬x3) ∧ (¬x1 ∨ x2 ∨ x4)
    vector<vector<int>> formula = {{1, -2, -3}, {-1, 2, 4}};
    int num_vars = 4;

    auto [graph, k] = build_vc_instance_from_3sat(formula, num_vars);

    cout << "Target k: " << k << endl;
    cout << "Graph adjacency list:" << endl;
    for (const auto& pair : graph) {
        cout << pair.first << ": ";
        for (const auto& neighbor : pair.second) {
            cout << neighbor << " ";
        }
        cout << endl;
    }

    return 0;
}
```

## 16.6. Робота з NP-важкими задачами. Апроксимаційні алгоритми

Виявлення того, що задача є NP-повною або NP-важкою, є важливим, але часто невтішним результатом. Це означає, що, якщо P = NP, не існує алгоритму, здатного знайти точний оптимальний розв’язок для всіх випадків задачі за поліноміальний час. Проте в реальному світі такі задачі (наприклад, оптимізація логістичних маршрутів, планування ресурсів, розробка мікросхем) виникають постійно і вимагають розв’язання.

Коли точний розв’язок є недосяжним, ми можемо вдатися до *апроксимаційних алгоритмів.* Це алгоритми, які працюють за поліноміальний час і знаходять необов’язково оптимальний, але гарантовано «достатньо добрий» розв’язок.

Означення (Коефіцієнт апроксимації). Для задачі мінімізації алгоритм має *коефіцієнт апроксимації* $\alpha \geq 1$, якщо для будь-якого екземпляра задачі він знаходить розв'язок вартістю C, такий що $C \leq \alpha \cdot Copt$, де Copt − вартість оптимального розв'язку. Такий алгоритм називають α-апроксимаційним.

Приклад 1. 2-апроксимаційний алгоритм для задачі про вершинне покриття.

Задача про вершинне покриття є NP-повною, але для неї існує дуже простий і елегантний 2-апроксимаційний алгоритм.

Ідея алгоритму: алгоритм будує вершинне покриття, ітеративно вибираючи довільне ребро з графа, додаючи обидві його кінцеві вершини до покриття, і видаляючи з графа всі ребра, інцидентні цим двом вершинам. Процес триває, доки в графі не залишиться ребер.

```
Псевдокод
function Approx_Vertex_Cover(Graph G):
  C = {} // Порожня множина для вершинного покриття
  E' = G.edges // Копія множини ребер графа

  while E' is not empty:
    pick an arbitrary edge (u, v) from E'
    add u to C
    add v to C
    remove from E' every edge incident on either u or v

  return C
```

Доведення коефіцієнта апроксимації 2:

1. Коректність. Побудована множина C є вершинним покриттям. Жодне ребро не може залишитися непокритим, оскільки якби таке ребро існувало, воно мало б бути вибране на якомусь кроці алгоритму, і його кінці були б додані до C.

2. Коефіцієнт. Нехай M − це множина ребер, які алгоритм вибирав на кожному кроці. Оскільки жодні два ребра в M не мають спільних вершин (бо після вибору ребра (u, v) всі інші ребра, інцидентні u або v, видаляються), M є *максимальним паруванням* (maximal matching). Розмір знайденого покриття $|C| = 2 \cdot |M|$.

3. З іншого боку, будь-яке вершинне покриття (охоплюючи оптимальне, Copt) повинно покривати всі ребра, зокрема й ребра з M. Оскільки ребра в M не мають спільних вершин, для покриття |M| ребер потрібно щонайменше |M| вершин. Отже, |Copt| ≥ |M|.

4. Поєднуючи ці два факти, отримуємо: |C| = 2·|M| ≤ 2·|Copt|. Так, розмір знайденого покриття не більше ніж удвічі перевищує розмір оптимального.

Приклад. Покрокове виконання на графі

Розглянемо граф з вершинами {1, 2, 3, 4, 5} і ребрами {(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)}.

1. Ініціалізація: C = ∅, E′ = {(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)}.

2. Крок 1: Вибираємо ребро (1,2). Додаємо 1 і 2 до C. C = {1,2}. Видаляємо ребра (1, 2), (1, 3), (2, 3), (2, 4). E′ = {(3, 5), (4, 5)}.

3. Крок 2: Вибираємо ребро (3, 5). Додаємо 3 і 5 до C. C = {1, 2, 3, 5}. Видаляємо ребра (3, 5), (4, 5 ). E′ = ∅.

4. Результат: Алгоритм завершується. Знайдене вершинне покриття C = {1, 2, 3, 5}, розмір 4. (Оптимальне покриття для цього графа − {2, 3, 4} або {1, 2, 5}, розмір 3. 4 ≤ 2·3, коефіцієнт дотримано).

*Python*

```
def approx_vertex_cover(graph):
    """
    2-апроксимаційний алгоритм для вершинного покриття.
    graph: словник, що представляє список суміжності.
    """
    cover = set()
    edges = set()

    # Створюємо множину унікальних ребер
    for u in graph:
        for v in graph[u]:
            if (u, v) not in edges and (v, u) not in edges:
                edges.add((u, v))

    while edges:
        u, v = edges.pop()
        cover.add(u)
        cover.add(v)
```

```
            # Видаляємо всі ребра, інцидентні u або v
            edges_to_remove =
            for edge in edges:
                if u in edge or v in edge:
                    edges_to_remove.append(edge)

            for edge in edges_to_remove:
                edges.remove(edge)

        return list(cover)

    # Приклад графа
    graph_example = {
        1: ,
        2: ,
        3: ,
        4: ,
        5:
    }
    print(f"Vertex Cover: {approx_vertex_cover(graph_example)}")
```

```
    C++
    #include <iostream>
    #include <vector>
    #include <set>
    #include <algorithm>

    using namespace std;

    // Граф представлений списком ребер
    vector<int> approx_vertex_cover(int num_vertices, vector<pair<int,
int>>& edges) {
        set<int> cover;
        vector<bool> edge_removed(edges.size(), false);
        int remaining_edges = edges.size();

        while (remaining_edges > 0) {
            // Знаходимо перше невидалене ребро
            int edge_idx = -1;
            for (int i = 0; i < edges.size(); ++i) {
                if (!edge_removed[i]) {
                    edge_idx = i;
```

```
            break;
        }
    }

    if (edge_idx == -1) break;

    int u = edges[edge_idx].first;
    int v = edges[edge_idx].second;

    cover.insert(u);
    cover.insert(v);

    // Видаляємо всі ребра, інцидентні u або v
    for (int i = 0; i < edges.size(); ++i) {
        if (!edge_removed[i] && (edges[i].first == u |

| edges[i].first == v |
|
                                edges[i].second == u |

| edges[i].second == v)) {
            edge_removed[i] = true;
            remaining_edges--;
        }
    }
  }

  return vector<int>(cover.begin(), cover.end());
}

int main() {
  int num_vertices = 5;
  vector<pair<int, int>> edges = {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}};

  vector<int> vc = approx_vertex_cover(num_vertices, edges);

  cout << "Vertex Cover: ";
  for (int vertex : vc) {
    cout << vertex << " ";
  }
  cout << endl;

  return 0;
}
```

Приклад 2. Алгоритм Крістофідеса для задачі комівояжера.

Задача комівояжера (Traveling Salesman Problem, TSP) − це одна з найвідоміших NP-важких задач оптимізації. Для її метричного варіанту (де відстані задовольняють нерівність трикутника: $d(A, C) \leq d(A, B) + d(B, C)$) існує елегантний алгоритм Крістофідеса, який дає 1.5-апроксимацію.

Кроки алгоритму Крістофідеса:

1. Побудувати мінімальне кістякове дерево (MST). Знайти MST для графа міст. Вага MST, w(T), є нижньою оцінкою для довжини оптимального туру, w(Copt), оскільки будь-який тур без одного ребра є кістяковим деревом. Отже, $w(T) \leq w(Copt)$.

2. Знайти вершини непарного степеня. У MST знайти всі вершини O, які мають непарний степінь. Згідно з лемою про рукостискання, кількість таких вершин завжди парна.

3. Знайти ідеальне парування мінімальної ваги. На підграфі, що складається тільки з вершин з O, знайти ідеальне парування M з мінімальною сумарною вагою ребер. Доведено, що $w(M) \leq 0.5 \cdot w(Copt)$.

4. Об’єднати ребра. Створити мультиграф H, об’єднавши ребра з MST та парування M. У цьому новому графі всі вершини матимуть парний степінь.

5. Знайти Ейлерів цикл. Оскільки всі вершини в H мають парний степінь, у ньому існує Ейлерів цикл (цикл, що проходить по кожному ребру рівно один раз). Вага цього циклу дорівнює $w(T) + w(M) \leq 1.5 \cdot w(Copt)$.

6. Перетворити на Гамільтонів цикл. Перетворити Ейлерів цикл на Гамільтонів (тур комівояжера), пропускаючи повторно відвідані вершини («зрізаючи кути»). Завдяки нерівності трикутника цей крок не збільшує загальну довжину шляху.

*Псевдокод*

```
function Christofides_TSP(Graph G):
  // Крок 1: Побудувати MST
  T = MinimumSpanningTree(G)

  // Крок 2: Знайти вершини непарного степеня
  O = {v in T.vertices where degree(v) is odd}
```

```
// Крок 3: Знайти ідеальне парування мінімальної ваги
Subgraph_O = G.induced_subgraph(O)
M = MinimumWeightPerfectMatching(Subgraph_O)

// Крок 4: Об'єднати ребра
Multigraph H = T.edges U M.edges

// Крок 5: Знайти Ейлерів цикл
EulerianCircuit E = find_eulerian_circuit(H)

// Крок 6: Перетворити на Гамільтонів цикл
HamiltonianCircuit Tour = shortcut(E)

return Tour
```

Python (з використанням бібліотеки networkx)

```
Python
import networkx as nx
from itertools import combinations

def christofides_tsp(graph):
    """
    1.5-апроксимаційний алгоритм Крістофідеса для метричного TSP.
    graph: об'єкт графа NetworkX, повний, з вагами ребер 'weight'.
    """
    # Крок 1: Побудувати MST
    mst = nx.minimum_spanning_tree(graph, weight='weight')

    # Крок 2: Знайти вершини непарного степеня
    odd_degree_nodes = [v for v, d in mst.degree() if d % 2 == 1]

    # Крок 3: Знайти ідеальне парування мінімальної ваги
    odd_subgraph = graph.subgraph(odd_degree_nodes)
    # NetworkX min_weight_matching працює для неповних графів,
    # але для нашої задачі підграф має бути повним.
    matching = nx.min_weight_matching(odd_subgraph, weight='weight')

    # Крок 4: Об'єднати ребра
    multigraph = nx.MultiGraph(mst)
    multigraph.add_edges_from(matching)
```

```
        # Крок 5: Знайти Ейлерів цикл
        eulerian_circuit_edges = list(nx.eulerian_circuit(multigraph))

        # Крок 6: Перетворити на Гамільтонів цикл
        tour =
        visited = set()
        start_node = eulerian_circuit_edges
        tour.append(start_node)
        visited.add(start_node)

        for u, v in eulerian_circuit_edges:
            if v not in visited:
                tour.append(v)
                visited.add(v)

        tour.append(start_node) # Повернутися до початкової точки
        return tour

    # Приклад (потрібно створити повний граф з вагами)
    G = nx.complete_graph(5)
    # Додамо ваги, що задовольняють нерівність трикутника (наприклад, з 2D точок)
    pos = {0: (0,0), 1: (1,5), 2: (3,1), 3: (6,4), 4: (8,0)}
    for u, v in G.edges():
        dist = ((pos[u] - pos[v])**2 + (pos[u] - pos[v])**2)**0.5
        G.edges[u,v]['weight'] = dist

    tour = christofides_tsp(G)
    print(f"Approximated TSP Tour: {tour}")
```

Реалізація алгоритму Крістофідеса на C++ з нуля є значно складнішою, оскільки вимагає реалізації алгоритмів для MST (наприклад, Крускала або Пріма), пошуку ідеального парування мінімальної ваги (наприклад, алгоритм Едмондса, що є нетривіальним) та пошуку Ейлерового циклу. На практиці для таких задач використовують спеціалізовані бібліотеки (наприклад, LEMON Graph Library або Boost.Graph). Нижче наведено лише структуру, що ілюструє логіку.

```
C++
#include <iostream>
#include <vector>
#include <numeric>
//... потрібні будуть складніші структури даних та алгоритми

struct Edge { int u, v; double weight; };

// --- Потрібні реалізації для наступних функцій ---
vector<Edge> kruskal_mst(int num_vertices, const vector<Edge>& all_edges);
vector<Edge> min_weight_perfect_matching(const vector<int>& odd_nodes, const vector<Edge>& all_edges);
vector<int> find_eulerian_tour(int start_node, const vector<Edge>& multigraph_edges);
vector<int> shortcut_tour(const vector<int>& eulerian_tour);
// ------------------------------------------------

vector<int> christofides_tsp(int num_vertices, const vector<Edge>& all_edges) {
    // Крок 1: MST
    vector<Edge> mst = kruskal_mst(num_vertices, all_edges);

    // Крок 2: Вершини непарного степеня
    map<int, int> degrees;
    for (const auto& edge : mst) {
        degrees[edge.u]++;
        degrees[edge.v]++;
    }
    vector<int> odd_degree_nodes;
    for (const auto& pair : degrees) {
        if (pair.second % 2!= 0) {
            odd_degree_nodes.push_back(pair.first);
        }
    }

    // Крок 3: Парування
    vector<Edge> matching =
min_weight_perfect_matching(odd_degree_nodes, all_edges);
```

```
    // Крок 4: Об'єднання
    vector<Edge> multigraph_edges = mst;
    multigraph_edges.insert(multigraph_edges.end(), matching.begin(),
matching.end());

    // Крок 5: Ейлерів цикл
    vector<int> eulerian_tour = find_eulerian_tour(0, multigraph_edges);

    // Крок 6: "Зрізання кутів"
    vector<int> final_tour = shortcut_tour(eulerian_tour);

    return final_tour;
}

int main() {
    cout << "C++ implementation of Christofides algorithm requires
specialized graph libraries." << endl;
    //... код для створення графа та виклику функції
    return 0;
}
```

## 16.7. Приклади розв’язку задач

Цей розділ містить приклади розв’язання типових задач, а також питання та завдання для самостійної роботи, що допоможуть закріпити матеріал, викладений у цьому параграфі.

**Задача 1.** Доведення NP-повноти задачі про незалежну множину

Незалежна множина у графі G = (V, E) − це підмножина вершин $V' \subseteq V$ така, що жодні дві вершини у $V'$ не з’єднані ребром. Задача розпізнавання IS: для даного графа G та цілого числа k, чи існує в G незалежна множина розміром щонайменше k?

Довести: задача IS є NP-повною.

*Розв’язок*

Використаємо стандартну двокрокову процедуру.

1. Довести, що $IS \in NP$.

Сертифікатом є множина вершин $V'$. Щоб перевірити її, потрібно:

a) Перевірити, чи $|V'| \geq k$. Це займає час $O(|V'|)$.

b) Перевірити, чи для кожної пари вершин $\{u, v\} \subseteq V'$ ребро $(u,v)$ не належить $E$. Це можна зробити за час $O(|V'|2 \cdot$ час перевірки ребра), що є поліноміальним.

Отже, $IS \in NP$.

2. Довести, що IS є NP-важкою.

Зведемо до неї відому NP-повну задачу про вершинне покриття ($VC \leq pIS$). Для цього нам потрібно встановити зв'язок між цими двома задачами.

Ключове спостереження: Множина вершин $V'$ є вершинним покриттям графа $G = (V, E)$ тоді і тільки тоді, коли її доповнення $V \setminus V'$ є незалежною множиною.

Доведення спостереження:

($\Rightarrow$) Нехай $V'$ − вершинне покриття. Припустимо, що $V \setminus V'$ не є незалежною множиною. Тоді існують дві вершини $u,v \in V \setminus V'$, з'єднані ребром. Але якщо u і v не належать $V'$, то ребро $(u,v)$ не покрите множиною $V'$, що суперечить тому, що $V'$ є вершинним покриттям.

($\Leftarrow$) Нехай $V \setminus V'$ − незалежна множина. Розглянемо довільне ребро $(u,v) \in E$. Оскільки $V \setminus V'$ є незалежною множиною, неможливо, щоб і u, і v одночасно належали $V \setminus V'$. Отже, принаймні одна з них (або u, або v) повинна належати $V'$. Це означає, що $V'$ покриває ребро (u, v). Оскільки це вірно для будь-якого ребра, $V'$ є вершинним покриттям.

Побудова зведення. Нехай нам дано екземпляр задачі VC: граф $G = (V, E)$ та ціле число kVC. Хочемо знайти відповідь на питання: «Чи існує в G вершинне покриття розміром щонайбільше kVC?».

Перетворюємо цей екземпляр на екземпляр задачі IS:

- Граф залишається тим самим: $G' = G$.
- Нове цільове число kIS встановлюємо як $kIS = |V| - kVC$.

Це перетворення очевидно виконується за поліноміальний час.

Доведення коректності зведення:

Граф G має вершинне покриття V′ розміром $|V'| \leq kVC \Leftrightarrow$ (за нашим спостереженням) існує незалежна множина $V \setminus V'$ розміром $|V \setminus V'| = |V| - |V'| \geq |V| - kVC = kIS$.

Так, відповідь на екземпляр VC є «так» тоді і тільки тоді, коли відповідь на наш сконструйований екземпляр IS є «так». Ми успішно звели $VC \leq pIS$.

Оскільки $IS \in NP$ і IS є NP-важкою, задача про незалежну множину є NP-повною.

**Задача 2.** Виконання апроксимаційного алгоритму для TSP

Дано 5 міст з матрицею відстаней (метричний випадок). Застосувати алгоритм Крістофідеса для знаходження апроксимованого туру, починаючи з міста A.

| | **A** | **B** | **C** | **D** | **E** |
|---|---|---|---|---|---|
| **A** | 0 | 3 | 4 | 2 | 7 |
| **B** | 3 | 0 | 4 | 5 | 6 |
| **C** | 4 | 4 | 0 | 5 | 8 |
| **D** | 2 | 5 | 5 | 0 | 6 |
| **E** | 7 | 6 | 8 | 6 | 0 |

*Розв'язок*

1. Крок 1: Побудувати MST.

Використовуючи алгоритм Пріма або Крускала, знаходимо MST. Ребра MST: (A, D) вага 2, (A, B) вага 3, (A, C) вага 4, (B, E) вага 6. (Зауваження: ребро (B, C) з вагою 4 створює цикл з (A, B) та (A, C), тому не додається).

Ребра MST: {(A, D), (A, B), (A, C), (B, E)}.

Загальна вага MST: 2 + 3 + 4 + 6 = 15.

2. Крок 2: Знайти вершини непарного степеня в MST.

degree(A) = 3 (непарний)

degree(B) = 2 (парний)

degree(C) = 1 (непарний)

degree(D) = 1 (непарний)

degree(E) = 1 (непарний)

Множина вершин непарного степеня O = {A, C, D, E}.

3. Крок 3: Знайти ідеальне парування мінімальної ваги на O.

Розглядаємо підграф з вершинами {A, C, D, E} і всіма ребрами між ними з початкового графа. Можливі ідеальні парування:

{(A, C), (D, E)}: вага 4 + 6 = 10.

{(A, D), (C, E)}: вага 2 + 8 = 10.

{(A, E), (C, D)}: вага 7 + 5 = 12.

Мінімальна вага − 10. Виберемо парування M = {(A, C), (D, E)}.

4. Крок 4: Об’єднати ребра MST та M.

Мультиграф H має ребра: {(A, D), (A, B), (A, C), (B, E)} (з MST) та {(A, C), (D,E)} (з M).

Ребро (A, C) тепер присутнє двічі.

5. Крок 5: Знайти Ейлерів цикл.

Почнемо з A. Можливий цикл: A → B → E → D → A → C → A.

Послідовність відвіданих вершин: A-B-E-D-A-C-A.

6. Крок 6: Перетворити на Гамільтонів цикл.

Проходимо по Ейлеровому циклу, пропускаючи вершини, які вже відвідали.

A (додаємо)

B (додаємо)

E (додаємо)

D (додаємо)

A (вже є, пропускаємо)

C (додаємо)

A (повертаємось в початок)

Отриманий тур: A -> B -> E -> D -> C -> A.

7. Результат:

Апроксимований тур: A-B-E-D-C-A.

Його довжина: d(A, B) + d(B, E) + d(E, D) + d(D, C) + d(C, A) = 3 + 6 + 6 + 5 + 4 = 24.

(Для порівняння, оптимальний тур тут A-D-B-C-E-A з довжиною 25, або A-B-C-D-E-A з довжиною 26. У такому випадку апроксимація виявилась кращою за очевидні тури, що іноді трапляється).

## Контрольні питання

1. У чому полягає ключова відмінність між розв'язною та напівроз в'язною проблемою? Наведіть приклад для кожної.

2. Сформулюйте проблему зупинки. Чому її не можна розв'язати, просто симулюючи виконання програми?

3. Поясніть логіку доведення нерозв'язності проблеми зупинки від супротивного. У чому полягає парадокс?

4. Що таке клас складності P? Чому задачі з цього класу вважаються «ефективно розв'язними»?

5. Дайте два еквівалентних означення класу складності NP. Поясніть на прикладі задачі про гамільтонів цикл, чому вона належить до NP.

6. Що таке проблема P = NP? Які практичні наслідки мав би доказ того, що P = NP?

7. Що таке поліноміальне зведення $A \leq pB$? Що воно говорить про відносну складність задач A та B?

8. Які дві умови повинна задовольняти задача, щоб бути NP-повною?

9. Чому теорема Кука-Левіна є фундаментальною для теорії NP-повноти?

10. У чому різниця між NP-важкою та NP-повною задачею? Чи може задача бути NP-важкою, але не належати до NP? Наведіть приклад.

11. Опишіть загальну стратегію доведення NP-повноти нової задачі.

12. Що таке апроксимаційний алгоритм і коефіцієнт апроксимації? Коли доцільно використовувати такі алгоритми?

13. Поясніть, чому простий жадібний алгоритм для задачі про вершинне покриття має коефіцієнт апроксимації 2.

14. Яка основна вимога до вхідних даних для алгоритму Крістофідеса і чому вона важлива на останньому кроці алгоритму?

15. Опишіть роль мінімального кістякового дерева та ідеального парування в алгоритмі Крістофідеса.

## Задачі для самостійного опрацювання

1. Класифікуйте наведені проблеми як розв’язні, напівроз-в’язні або нерозв’язні:

a) Для даного цілого числа N визначити, чи існують цілі числа x, y, z такі, що x3 + y3 = z3 + N.

b) Для даної контекстно-вільної граматики визначити, чи генерує вона порожній рядок.

c) Для даної програми на Python визначити, чи виведе вона колись рядок "Hello, World!".

2. Чи належать наведені задачі до класу P? Обґрунтуйте.

a) Знаходження медіани в масиві з N чисел.

b) Перевірка, чи є два графи ізоморфними.

c) Множення двох матриць розміром N × N.

3. Чи належать наведені задачі до класу NP? Обґрунтуйте.

a) Задача про розбиття множини: для даної множини цілих чисел, чи можна її розбити на дві підмножини з однаковою сумою?

b) Задача про 2-розфарбовування: чи можна розфарбувати вершини графа у два кольори так, щоб жодні дві суміжні вершини не мали однакового кольору?

c) Задача про найдовший простий шлях у графі: для даного графа G та числа k, чи існує в G простий шлях довжиною щонайменше k?

4. Доведіть, що якщо P = NP, то задача факторизації цілих чисел (розкладання числа на прості множники) може бути розв’язана за поліноміальний час.

5. Задача про кліку (Clique) полягає у визначенні, чи містить граф G повний підграф (кліку) розміром k. Доведіть, що задача Clique є NP-повною, звівши до неї задачу про незалежну множину. (Підказка: використайте доповнення графа).

6. Розгляньте задачу про покриття множини (Set Cover): дано універсум U та колекцію підмножин S = {S1, S2,..., Sm}, чи можна вибрати k підмножин з S, об’єднання яких дорівнює U? Доведіть, що Set Cover є NP-повною, звівши до неї задачу про вершинне покриття.

7. Напишіть програму (Python або C++), яка реалізує 2-апроксимаційний алгоритм для вершинного покриття. Протестуйте її на кількох графах.

8. Для графа з задачі 2 у розділі «Приклади розв'язку задач» виконайте 2-апроксимаційний алгоритм для вершинного покриття, починаючи з ребра (2, 4). Порівняйте результат з оптимальним.

9. Алгоритм сортування має часову складність O(nlogn) і сортує 10,000 елементів за 5 мс. Скільки часу, орієнтовно, займе сортування 10,000,000 елементів?

10. Доведіть, що якщо проблема L1 поліноміально зводиться до L2, а $L2 \in P$, то $L1 \in P$.

11. Доведіть, що якщо L є NP-повною задачею, і $L \in P$, то $P = NP$.

12. Задача 3-SAT є NP-повною. Розгляньте задачу 2-SAT, де кожна клауза містить рівно два літерали. Покажіть, що 2-SAT належить до класу P. (Підказка: побудуйте граф імплікацій).

13. Розгляньте задачу про розфарбовування графа. Доведіть, що задача 3-розфарбовування (використання 3 кольорів) є NP-повною.

14. Напишіть програму, яка ілюструє зведення 3-SAT до вершинного покриття: на вхід вона приймає булеву формулу, а на вихід видає граф та число k.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Безущак О., Ганюшкін О. Математична логіка : навч. посіб. К. : ВПЦ «Київський університет». 2023. 143 с.
2. Бородкіна І. Л., Бородкін Г. О. Теорія алгоритмів : навчальний посібник. Київ : Центр учбової літератури, 2023. 184 с.
3. Зубенко В. В., Шкільняк С. С. Основи математичної логіка : навчальний посібник. К. : НУБіП України, 2020. 102 с.
4. Ільман В. М., Іванов О. П., Панік Л. О. Алгоритми, дані і структури / Дніпропет. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. Дніпро, 2019. 134 с.
5. Караванова Т. П. Теорія алгоритмів. Частина 2 : Обчислювальні алгоритми : навч. посіб. Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2022. 288 с.
6. Назарук М. В., Сінчук А. М. Математична логіка та теорія алгоритмів : навчально-методичний посібник. Рівне : РДГУ, 2020. 118 с.
7. Троцько В. В. Теорія алгоритмів : навчально-методичний посібник. Київ : Університет економіки та права «КРОК», 2023. 123 с.
8. Шкільняк С. С. Математична логіка. Приклади й задачі : навч. посіб. К. : ВПЦ «Київський університет», 2022. 304 с.
9. Capelli, F., & Strozecki, Y. (2021). From Amortized to Worst Case Delay in Enumeration Algorithms. arXiv. URL : https://arxiv.org/abs/2108.10208
10. Code2flow – interactive code to flowchart converter. URL : https://code2flow.com/ (date of access: 24.09.2025).
11. Conte, A., Grossi, R., Kobayashi, Y. et al. Enumerating Graphlets with Amortized Time Complexity Independent of Graph Size. *Algorithmica* 87, 1247–1273 (2025). URL : https://doi.org/10.1007/s00453-025-01312-0
12. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press. URL : https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/

13. Gatiaher, Survey of data structures for large-scale information retrieval / Gatiaher. 2021. URL : https://gatiaher.github.io/projects/survey-of-data-structures-for-large-scale-information-retrieval/?utm_source=chatgpt.com

14. GDB online Debugger | Compiler – Code, Compile, Run, Debug online C, C++. *GDB online Debugger*. URL : https://www.onlinegdb.com/ (date of access: 24.09.2025).

15. Han J., et al. A survey of geometric graph neural networks: data structures, models and applications. *Frontiers of Computer Science.* 2025. Vol. 19.

16. Jansson, J. (2023). Editorial: Surveys in Algorithm Analysis and Complexity Theory (Special Issue). *Algorithms, 16* (4), 188. URL : https://doi.org/10.3390/a16040188

17. Mtaho A. Difficulties in learning the Data Structures course. *The Journal of Information and Application.* 2024. URL : https://journals.iaa.ac.tz/index.php/tji/article/view/136?utm_source=chatgpt.com

18. Roughgarden, T. (2020). Algorithms illuminated: Part 3: Greedy algorithms and dynamic programming. Soundlikeyourself Publishing, LLC.

19. Roughgarden, T. (2022). Algorithms illuminated: Part 4: Algorithms for NP-hard problems. Soundlikeyourself Publishing, LLC.

20. Skiena, S. S. (2020). The algorithm design manual (3rd ed.). Springer.

21. Turing machine visualization. URL : https://turingmachine.io/ (date of access: 24.09.2025).

22. VP Online – Online Drawing Tool. *Visual Paradigm – Online Productivity Suite*. URL : https://online.visual-paradigm.com/ (date of access: 24.09.2025).

23. Wengrow, J. (2020). A common-sense guide to data structures and algorithms: Level up your core programming skills (2nd ed.). The Pragmatic Programlers, LLC.

*Навчальне видання*

Лаговський Володимир Вікторович,
Омельчук Антон Анатолійович,
Скасків Лілія Василівна

# ТЕОРІЯ АЛГОРИТМІВ ТА СТРУКТУРА ДАНИХ

*Навчальний посібник*

| | |
|---|---|
| Відповідальний за випуск | *Н. Б. Добрянська* |
| Редактор | *М. П. Клименко* |
| Форматування та комп’ютерна верстка | *Д. П. Завальніцька* |

Здано до друку 30.12.2025. Формат 29,7 × 41,99
Папір офсетний № 1. Гарнітура «Times New Roman»
Ум. друк. арк. 13.6
Наклад 300 прим. Замовлення № 1172

Підготовлено до друку науково-організаційним відділом
Державного податкового університету
08205, вул. Університетська, 31, м. Ірпінь, Київська область,
Україна

*Свідоцтво про внесення суб’єкта видавничої справи
до державного реєстру видавців, виготовлювачів і
розповсюджувачів видавничої продукції
Серія ДК № 7669 від 20.09.2022*