МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Є.В. Бородавка

О.О. Терентьєв

Т.А. Гончаренко

ДИНАМІЧНЕ ПРОГРАМУВАННЯ

Підручник для студентів технічних закладів вищої освіти, які навчаються за спеціальностями:

121 – Інженерія програмного забезпечення

122 – Комп'ютерні науки

126 – Інформаційні системи та технології

КИЇВ 2025

УДК 004.92

ББК 32.973

Рецензенти: С. Д. Бушуєв, доктор технічних наук, професор, завідувач катедри управління проєктами Київського національного університету будівництва і архітектури
А. О. Олійник, доктор технічних наук, професор, професор катедри програмних засобів Національного університету «Запорізька політехніка»
В. В. Гнатушенко, доктор технічних наук, професор, завідувач катедри інформаційних технологій та комп'ютерної інженерії Національного технічного університету «Дніпровська політехніка»

Рекомендовано Вченою радою Київського національного університету будівництва і архітектури як навчальний підручник для студентів технічних закладів вищої освіти, які навчаються за спеціальностями 121 — Інженерія програмного забезпечення, 122 — Комп'ютерні науки і 126 — Інформаційні системи та технології.

Затверджено на засіданні Вченої ради Київського національного університету будівництва і архітектури, протокол № 1 від 17.01.2025 р.

Бородавка Є.В.

К 80 Динамічне програмування. Підручник / Є.В. Бородавка, О.О. Терентьєв, Т.А. Гончаренко. — К.: Компринт, 2025. — 178 с.:іл.

В підручнику розглянуто деякі підходи до вирішення оптимізаційних задач з теорії графів. Коротко розглянуті числові послідовності та основні типи задач де вони застосовуються. Також приділено увагу геш-таблицям та геш-функціям. Показано як використовувати геш для вирішення задач пошуку в символьних послідовностях. Розглянуті основні способи застосування динамічного програмування для вирішення оптимізаційних задач.

УДК 004.92

ББК 32.973

ЗМІСТ

Вступ

Цей підручник створено спільними зусиллями в.о. декана факультету автоматизації і інформаційних технологій КНУБА Олександра Терентьєва, завідувача катедри інформаційних технологій КНУБА Тетяни Гончаренко і професора катедри інформаційних технологій проєктування та прикладної математики КНУБА Євгенія Бородавки.

Незважаючи на назву, в підручнику розглянуто не лише використання методів динамічного програмування до розв'язання алгоритмічних задач, а й деякі підходи до вирішення задач з теорії графів, пошуку в символьних і числових послідовностях. Підручник призначений для набуття студентами практичних навичок з програмування алгоритмів і складається з десяти розділів. Кожен розділ присвячено окремій групі задач або методу розв'язання алгоритмічних задач. В кінці кожного розділу є перелік вправ для практичного застосування вивченого матеріалу. В переважній більшості розділів, вправи поділені на три категорії: прості завдання, завдання середнього рівня складності та складні завдання. Також кожен розділ закінчується переліком посилань на електронні джерела, що використані під час написання відповідного розділу. В кінці підручника є загальний список джерел, а також алфавітний покажчик для спрощення пошуку різних термінів та визначень.

Головне завдання цього підручника — навчити студентів аналізувати завдання, визначати оптимальний метод їх розв'язання та писати реалізацію будь-якою мовою програмування. Автори вважають, що для досягнення практичних результатів потрібно надавати необхідний мінімум теоретичної інформації і разом з тим давати покроковий розбір типових завдань для кращого розуміння матеріалу студентами. Саме тому кожен розділ містить по декілька прикладів з їх розбором крок за кроком і прикладами реалізації на мові програмування C++.

Для засвоєння вивченого матеріалу, студентам пропонується використовувати онлайн платформи для тренування навичок програмування такі як LeetCode і HackerEarth. Виконання завдань на цих платформах дозволить студентам підготуватися не тільки до виконання кваліфікаційних робіт з окремих освітніх компонент, а й до виконання тестових завдань на співбесідах з прийому на роботу.

Підручник написано українською мовою з використанням оновленого правопису. Англійські терміни подані в офіційній транскрипції.

Якщо у читачів виникнуть зауваження або побажання щодо покращення матеріалу підручника, то надсилайте їх на офіційну електронну адресу професора Бородавки: borodavka.iev@knuba.edu.ua.

1. Жадібний алгоритм

В цьому розділі ми розглянемо що таке «жадібний алгоритм» і чому він власне не є алгоритмом. Також ми дізнаємося які кроки необхідно виконати, щоб застосувати «жадібний алгоритм» та які є обмеження для його застосування.

1.1. Визначення жадібного алгоритму

Жадібний алгоритм — це метод вирішення задач, який завжди обирає оптимальне вирішення базуючись на поточній інформації. І хоча його реалізація приваблює своєю очевидністю, він не завжди оптимальний. Необхідно точно розуміти, коли застосовувати жадібний метод, а коли його варто уникати.

Жадібні алгоритми — це ціле сімейство алгоритмів (інколи ще говорять про жадібний підхід або жадібний метод), тому немає якогось конкретного жадібного алгоритму, який можна закодувати. Проте, всі жадібні алгоритми побудовані згідно одного принципу — обирати оптимальне вирішення на кожному кроці, не зважаючи на кроки, які були зроблені до або будуть зроблені після. Іншими словами, жадібний алгоритм робить локально оптимальний вибір у сподіванні, що він приведе до глобального оптимального розв'язку.

Жадібний підхід в чистому вигляді має час виконання $O(N)$, тобто це означає, що відповідь повинна бути отримана після перебору всіх елементів. Це може бути один перебір для простих задач або декілька для більш складних.

Розглянемо популярний приклад з монетками. Нехай у нас є монети номіналом 1, 2 і 5 копійок і нам необхідно відрахувати 10 копійок таким чином, щоб кількість монет була мінімальною.

За логікою жадібного методу на кожному кроці необхідно обирати монету з найбільшим номіналом і це приведе до мінімальної їх кількості в результаті. Візьмемо 5 копійок, тепер можемо взяти ще одну монету в 5 копійок, не вийшовши за обмеження у 10. В результаті 10 коп. = 5 коп. + 5 коп. Кількість монет — 2 і це, дійсно, мінімально можлива їх кількість за цих умов.

Але якщо в задачу додати монету номіналом 6 копійок, це зробить застосування жадібного підходу не оптимальним. Дійсно, на першому кроці нам необхідно обрати 6, як монету з найбільшим номіналом, але далі ми не можемо обрати ні 6, ні 5, оскільки це перевищить ліміт у 10 копійок. Залишається обрати дві монети по 2 коп. В результаті, ми маємо 10 як суму 6 + 2 + 2. В той час, як оптимальний розв'язок буде все ті ж 2 монети по 5.

В залежності від задачі, яку ми розв'язуємо, жадібний метод може бути оптимальним, а може і не бути. Якщо він дає не оптимальний розв'язок, то досить часто він дозволяє знайти рішення близьке до оптимального. В такому випадку необхідно скористатись іншим підходом, наприклад, повним перебором або динамічним програмуванням. Але, якщо жадібний підхід все ж працює коректно, час виконання алгоритму буде значно меншим за час

виконання повного перебору чи під час використання динамічного програмування.

1.2. Переваги і недоліки жадібного методу

Першою перевагою жадібного методу є те, що він легкий для розуміння і кодування. На кожному кроці алгоритму ми можемо абстрагуватись від попередніх і наступних кроків і думати лише про оптимальний розв'язок на цьому етапі. Жадібний підхід не передбачає скасування вже зробленого вибору (повернення на попередні кроки) і не прогнозує нічого на майбутнє.

Другою перевагою жадібного методу є передбачуваність швидкості виконання програми, тому що складність алгоритму очевидна. Найчастіше, вона лінійна, тобто, час виконання програми лінійно залежить від кількості вхідних даних. З іншими алгоритмічними підходами, такими як, наприклад, «розділяй і володарюй», це не завжди так.

Великим недоліком жадібного методу є те, що у більшості випадків жадібний алгоритм працює некоректно. Необхідно дуже добре розуміти, коли його можна використовувати, а коли — ні. І навіть якщо жадібний алгоритм дає оптимальне вирішення в певних випадках, важко довести, що підхід буде працювати у всіх інших можливих випадках.

1.3. Правило застосування жадібного методу

Існує евристичне правило для розуміння застосовності жадібного методу. Якщо обидві властивості наведені нижче справджуються, жадібний алгоритм може бути застосований до розв'язання задачі.

Принцип жадібного вибору. Послідовність оптимальних виборів на кожному кроці приводить до оптимального рішення в кінці. До оптимізаційної задачі можна застосувати принцип жадібного вибору, якщо послідовність локально оптимальних виборів дає глобальний оптимальний розв'язок. В типовому випадку доведення оптимальності здійснюється за такою схемою: спочатку доводиться, що жадібний вибір на першому етапі не унеможливлює шляху до оптимального розв'язку — для будь-якого розв'язку є інший, узгоджений із жадібним і не гірший від першого. Далі доводиться, що підзадача, яка виникла після жадібного вибору на першому етапі, аналогічна початковій, і міркування закінчується за індукцією. Інакше кажучи, за жадібного алгоритму ніколи не переглядаються попередні вибори для здійснення наступного, на відміну від динамічного програмування.

Оптимальна підструктура. Задача має оптимальну підструктуру, якщо оптимальний розв'язок цілої задачі містить оптимальний розв'язок для будь-якої підзадачі. Інакше кажучи, задача має оптимальну підструктуру, якщо кожен наступний крок веде до оптимального розв'язку. Прикладом «неоптимальної підструктури» може бути ситуація в шахах, коли взяття ферзя (хороший наступний крок) веде до програшу партії в цілому.

1.4. Приклади застосування жадібного методу

Жадібні алгоритми можна охарактеризувати як «короткозорі» і «невідновлювані». Вони ідеальні лише для задач з «оптимальною підструктурою». Попри це, жадібні алгоритми найкраще підходять для простих задач. Для багатьох інших задач жадібні алгоритми зазнають невдачі у продукуванні оптимального розв'язку, і можуть навіть видати найгірший з можливих розв'язків.

Перелік деяких алгоритмів, що використовують жадібний метод та дають оптимальний розв'язок:

- алгоритм Дейкстри для пошуку найкоротшого шляху в графі;
- алгоритм Прима для побудови мінімального кістякового дерева графа;
- алгоритм Крускала для побудови мінімального кістякового дерева графа;
- знаходження послідовності виконання задач;
- кодування Гоффмана — алгоритм, що використовується для стиснення інформації без втрат.

Також існує клас задач, що не можуть бути розв'язані за допомогою жадібного методу, але його використання дає наближений до оптимального результат. Наприклад, до будь-якої NP-повної задачі можна застосувати жадібний метод та отримати наближений результат за значно менший час, ніж отримання оптимального розв'язку.

Розглянемо для прикладу задачу пакування рюкзака. Злодій заліз у квартиру, де знайшов три коштовні речі.

Річ	Вартість	Вага
Золотий ланцюг	60000 грн	100 грам
Мобільний телефон	100000 грн	200 грам
Процесор	120000 грн	300 грам

Але у злодія є рюкзак лише на 500 грам. Яким чином він має вирішити, що взяти, щоб максимізувати свій прибуток?

Не важко побачити, що оптимальним рішенням буде взяти мобільний телефон і процесор, що в сумі дасть 220000 грн. Але, якби злодій застосував жадібний метод, він би почав обирати речі з найбільшою питомою вартістю (відношенням вартості до ваги речі). Найдорожчою річчю є золотий ланцюг, оскільки він коштує 600 грн/гр., тоді як мобільний телефон і процесор коштують 500 і 400 грн/гр. відповідно. Тобто, жадібний злодій обрав би золотий ланцюг і мобільний телефон, як найдорожчі і таким чином зміг би забрати з собою лише 160000 грн, оскільки процесор вже не вліз би у рюкзак.

Повернемося до правила. Вибір першою річчю золотого ланцюга протирічить принципу жадібного відбору, адже не веде до оптимального рішення. Таким

чином, жадібний алгоритм не може бути застосований в загальному випадку до задачі пакування рюкзака.

Жадібний метод — чудовий інтуїтивний і ефективний спосіб розв'язання багатьох популярних задач програмування. Його найбільшим недоліком є необхідність добре розуміти, коли він може бути застосований, а коли необхідно звернутись до іншого підходу. Але навіть в ситуаціях, коли жадібний алгоритм не дає оптимального рішення, його результат може бути близьким до оптимального і він точно буде ефективнішим за метод повного перебору або динамічне програмування.

1.5. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач з використанням жадібного методу. Для таких цілей підійдуть онлайн платформи для вирішення задач. Найпопулярніша така платформа це LeetCode (https://leetcode.com) але існують досить багато аналогічних платформ, які можна легко знайти в інтернеті. Наприклад платформа HackerEarth (https://www.hackerearth.com).

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані за допомогою жадібного методу.

Низький рівень складності

1. Remove Digit from Number to Maximize Result https://leetcode.com/problems/remove-digit-from-number-to-maximize-result.
2. Maximum Sum with Exactly K Elements https://leetcode.com/problems/maximum-sum-with-exactly-k-elements.
3. Distribute Money to Maximum Children https://leetcode.com/problems/distribute-money-to-maximum-children.
4. Lemonade Change https://leetcode.com/problems/lemonade-change.
5. Array Partition https://leetcode.com/problems/array-partition.

Середній рівень складності

1. Find Valid Matrix Given Row and Column Sums https://leetcode.com/problems/find-valid-matrix-given-row-and-column-sums.
2. Max Increase to Keep City Skyline https://leetcode.com/problems/max-increase-to-keep-city-skyline.
3. Divide Array into Arrays with Max Difference https://leetcode.com/problems/divide-array-into-arrays-with-max-difference.
4. Reduce Array Size to the half https://leetcode.com/problems/reduce-array-size-to-the-half.
5. Minimum Processing Time https://leetcode.com/problems/minimum-processing-time.

Високий рівень складності

1. Wildcard matching https://leetcode.com/problems/wildcard-matching.
2. Candy https://leetcode.com/problems/candy.
3. Couples Holding Hands https://leetcode.com/problems/couples-holding-hands.
4. Earliest Possible Day of Full Bloom https://leetcode.com/problems/earliest-possible-day-of-full-bloom.
5. Max Chunks to Make Sorted II https://leetcode.com/problems/max-chunks-to-make-sorted-ii.

1.6. Розбір деяких задач

Розглянемо декілька завдань різної складності та проаналізуємо яким чином застосувати до них жадібний метод. До кожного завдання вказаний його номер на платформі LeetCode.

1.6.1. Найбільший периметр трикутника

Завдання. Заданий масив цілих позитивних чисел, кожне з яких є довжиною потенційного ребра трикутника. Знайти максимальний периметр трикутника, який можна побудувати з цих ребер. Якщо неможливо сформувати трикутник із заданого набору ребер, то видати нуль. (*Низький рівень складності — №976*).

Розв'язання. Очевидно, що найбільшим буде периметр трикутника, який має найбільші можливі ребра. Тобто якщо сконструювати трикутник з найбільших можливих ребер, то його периметр і буде відповіддю. Отже *принцип жадібного вибору* тут полягає у виборі більшого трикутника, ніж той який був сконструйований на попередньому етапі. Якщо ми його дотримуватимемося, то в результаті отримаємо оптимальне рішення. Отже кожен наступний наш крок вестиме до оптимального рішення, тобто задача має *оптимальну підструктуру*.

В цілому ми переконалися, що до цієї задачі застосовний жадібний метод. Але початковий набір даних не впорядкований, тому ми не можемо ефективно обирати ребра з щоразу більшою довжиною. Тому необхідно зробити попередню обробку даних у вигляді сортування масиву ребер. Сортування займає час $O(N*\log N)$ — це і буде складність нашого вирішення.

Тепер, коли масив відсортований, ми можемо тестувати кожну трійку ребер на умову трикутника — сума двох коротших ребер повинна бути більшою за найбільше ребро. Якщо трикутник існує, то рахуємо його периметр і порівнюємо з попереднім порахованим і якщо новий периметр більший, то приймаємо його в якості розв'язку. Коли ми дійдемо до останнього найдовшого ребра, ми отримаємо оптимальний розв'язок.

Для цього алгоритму ми можемо застосувати просту оптимізацію. Оскільки нам відомо, що найкращим розв'язком буде трикутник з найбільшими можливими ребрами, то ми можемо відсортувати масив за спаданням і йти від найдовшого

ребра до найкоротшого. Тоді найперший отриманий трикутник і буде відповіддю. В загальному випадку ця оптимізація ніякого виграшу не дає, оскільки може бути ситуація, коли лише три найменших ребра сформують трикутник і нам все одно доведеться зробити перебір всіх елементів масиву.

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
int largestPerimeter(vector<int>& nums)
{
    sort(nums.begin(), nums.end(), greater<int>());
    for(int i = 0; i < nums.size()-2; i++)
    {
        if(nums[i+2] + nums[i+1] > nums[i])
        {
            return (nums[i] + nums[i+1] + nums[i+2]);
        }
    }
    return 0;
}
```

1.6.2. Контейнер з найбільшою кількістю води

Завдання. Заданий масив цілих позитивних чисел, які вказують висоту сторони контейнера, що розміщена в координаті з індексом числа в масиві. Потрібно знайти такі дві сторони, які утворять контейнер з віссю X, що вміщуватиме найбільшу кількість води. Видайте найбільшу можливу кількість води в контейнері. (*Середній рівень складності — №11*).

Приклад. Вхідний масив `height` = [1, 8, 6, 2, 5, 4, 8, 3, 7] (рис. 1.1).

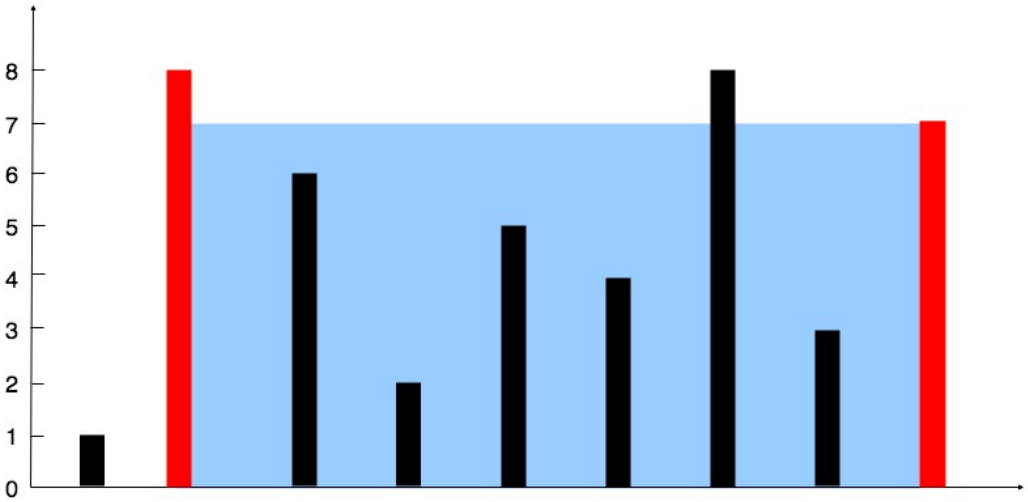


Рис. 1.1. Ілюстрація до завдання

Найбільша кількість води буде якщо ми оберемо в якості сторін контейнера індекс номер 1 з висотою 8 та індекс номер 8 з висотою 7. Тоді загальна кількість буде (8-1)*7=49.

Розв'язання. З опису завдання нам зрозуміло, що необхідно побудувати прямокутник з найбільшою можливою площею за допомогою наявних сторін. Ширина прямокутника це різниця між індексами сторін, а висота — найменша з висот двох сторін. Отже *принцип жадібного вибору* тут полягає у виборі прямокутника з більшою площею, ніж попередній побудований прямокутник. Якщо ми його дотримуватимемося, то в результаті отримаємо оптимальне рішення. Отже кожен наступний наш крок вестиме до оптимального рішення, тобто задача має *оптимальну підструктуру*.

Яким же чином нам обирати наступний прямокутник? Якщо ми спробуємо побудувати всі можливі та вибирати з них, то це буде повний перебір з часом виконання O(N!). Нам же потрібно застосувати жадібний підхід з часом виконання O(N). Зрозуміло, що під час проходження масиву ми будемо рухатися вздовж вісі X. Отже ширина нашого прямокутника завжди буде зменшуватися. Якщо ми будемо рухатися завжди в один бік, то ми не будемо знати коли нам зупинитися. Тому тут потрібно на кожному кроці визначатися, яку зі сторін ми повинні змінити. З двох початкових сторін (лівої та правої) ми повинні обрати ту, яка потенційно приведе нас до більшої площі прямокутника, тобто потрібно змінювати меншу сторону в надії, що наступна буде більшою. Тобто на кожному кроці ми обираємо меншу сторону і робимо рух до центру масиву від неї. Коли лівий та правий індекси зрівняються ми закінчимо роботу і матимемо поточну найбільшу площу прямокутника. За таким алгоритмом ми відвідаємо кожен елемент масиву лише 1 раз, тобто складність алгоритму буде O(N).

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
int maxArea(vector<int>& height)
{
    int s = height.size(), b = 0, e = s - 1, a = 0, h = 0;
    do{
        int l = height[b];
        int r = height[e];
        int w = (e - b);
        if(l < r) { h = l; b++; }
        else      { h = r; e--; }
        int q = h * w;
        a = max(q, a);
    }while(b < e);
    return a;
}
```

Запитання для самоконтролю

1. Що таке «жадібний алгоритм»?
2. Які переваги та недоліки жадібного методу?
3. Які властивості повинна мати задача, щоб до неї можна було застосувати жадібний метод?
4. Назвіть алгоритми, які використовують жадібний метод для знаходження оптимального розв'язку.
5. Для якого класу задач жадібний метод дає наближений до оптимального результат?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Жадібний_алгоритм
2. https://devzone.org.ua/post/zadibni-alhorytmy
3. https://ua5.org/algorithm/1909-zhadibni-algorytmy.html
4. https://drukarnia.com.ua/articles/zhadibni-algoritmi-chastina-1-YZIN3
5. https://drukarnia.com.ua/articles/zhadibni-algoritmi-chastina-2-SWRHw
6. https://www.guru99.com/uk/greedy-algorithm.html
7. https://en.wikipedia.org/wiki/Greedy_algorithm
8. https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial

2. Граф як структура даних

В цьому розділі ми розглянемо граф з точки зору структури даних для зберігання інформації. Визначимо основні терміни, що використовуються в теорії графів, а також розглянемо яким чином подаються графи в комп'ютерних програмах.

2.1. Визначення графа

Граф — це абстрактний тип даних, який призначений для реалізації концепцій неорієнтованого і орієнтованого графів, які походять з математики, а саме, з теорії графів.

2.1.1. Приклад графа

Структура даних графа складається з вузлів або вершин, які містять інформацію та пов'язані з іншими вузлами за допомогою дуг або ребер (рис. 2.1).

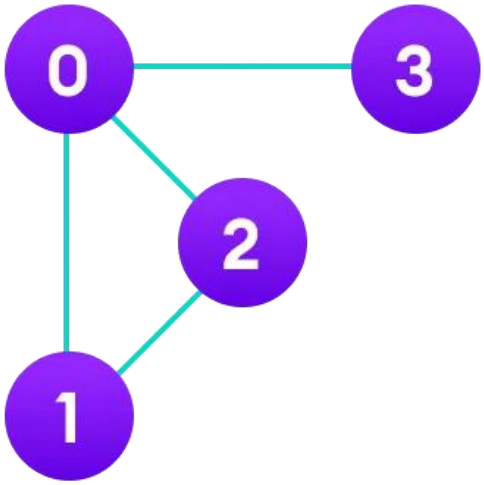


Рис. 2.1. Приклад графа

Граф на рис. 2.1 складається з наступних частин:

- список вершин: V = { 0, 1, 2, 3 };
- список ребер: E = { (0,1), (0,2), (0,3), (1,2) };
- власне граф: G = { V, E }.

2.1.2. Терміни теорії графів

Інцидентність — поняття, що використовується тільки для ребра і вершини: якщо v_1 і v_2 — вершини, а e =(v_1, v_2) — ребро, що їх з'єднує, тоді вершина v_1 і ребро e інцидентні, вершина v_2 і ребро e також інцидентні. Дві вершини (або два ребра) інцидентними бути не можуть. Для позначення найближчих вершин (ребер) використовується поняття суміжності.

Суміжність — поняття, яке використовується по відношенню тільки до двох ребер або двох вершин: Два ребра, інцидентні одній вершині, називаються суміжними; дві вершини, інцидентні одному ребру, також називаються суміжними.

Маршрут або **шлях** в графі — послідовність вершин і ребер v_0, e_1, v_1, e_2, v_2, …, e_k, v_k, що чергуються, в якій будь-які два сусідні елемента інцидентні. Якщо $v_0 = v_k$, то маршрут замкнений, інакше відкритий.

Орієнтований граф (коротко орграф) — (мульти)граф, ребрам якого присвоєно напрямок. Орієнтовані ребра називаються також дугами і просто ребрами.

Направлений граф — орієнтований граф, в якому дві вершини з'єднуються не більше ніж однією дугою.

Зважений граф — граф, кожному ребру якого поставлено у відповідність деяке значення (вага ребра).

Вага ребра — значення, що поставлене у відповідність цьому ребру зваженого графа. Зазвичай вага — дійсне число, в такому випадку його можна інтерпретувати як «довжину» ребра.

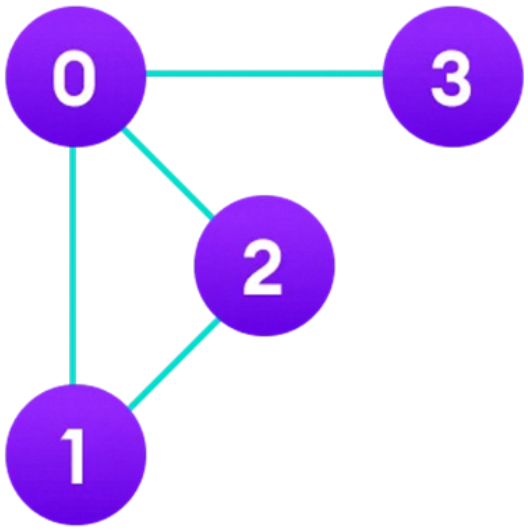
2.2. Подання графів у комп'ютерних програмах

Існує два основних способи подати граф у комп'ютерній програмі: матриці та списки.

2.2.1. Матриця інцидентності

Матриця інцидентності графа — матриця, значення елементів якої характеризуються інцидентністю відповідних вершин графа (по вертикалі) та його ребер (по горизонталі).

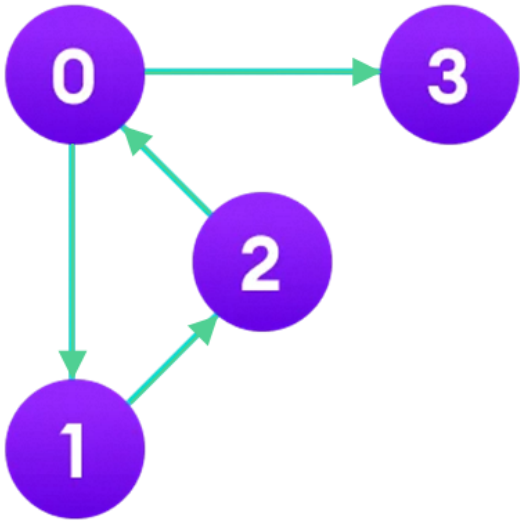
Для *неорієнтованого* графа елемент матриці інцидентності приймає значення 1, якщо вершина і ребро, що відповідають йому, інцидентні; в інших випадках (в тому числі і для петель) значенню елемента присвоюється 0. (рис. 2.2).



M	0-1	0-2	0-3	1-2
0	1	1	1	0
1	1	0	0	1
2	0	1	0	1
3	0	0	1	0

Рис. 2.2. Неорієнтований граф та його матриця інцидентності

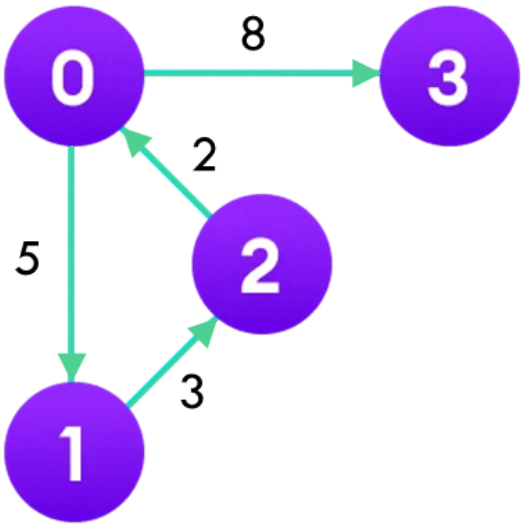
Для *орієнтованого* графа елемент приймає значення 1, якщо інцидентна вершина це початок ребра, значення -1, якщо кінець; в інших випадках (в тому числі і для петель) значенню елемента присвоюється 0 (рис. 2.3).



M	0-1	0-2	0-3	1-2
0	-1	1	-1	0
1	1	0	0	-1
2	0	-1	0	1
3	0	0	1	0

Рис. 2.3. Орієнтований граф та його матриця інцидентності

Для *зваженого* графа елемент матриці приймає значення ваги ребра з відповідним знаком у випадку орієнтованого графа (рис. 2.4).



M	0-1	0-2	0-3	1-2
0	-5	2	-8	0
1	5	0	0	-3
2	0	-2	0	3
3	0	0	8	0

Рис. 2.4. Зважений орієнтований граф та його матриця інцидентності

Серед недоліків матриці інцидентності варто відзначити наступні:

- вимагає багато пам'яті для зберігання даних у випадку щільного графа (граф з великою кількістю ребер);
- складна для реалізації і застосування в комп'ютерних програмах;
- повільна в переборі всіх ребер.

Просторова та часові складності для різних операцій над графом з кількістю вершин V та кількістю ребер E в нотації «O велике» наступні:

- O(V * E) — пам'ять для зберігання графа;
- O(V * E) — час на додавання нової вершини в граф;
- O(V * E) — час на додавання нового ребра в граф;
- O(V * E) — час на видалення вершини з графа;
- O(V * E) — час на видалення ребра з графа;
- O(E) — час на визначення суміжності вершин графа.

Матриця інцидентності добре підходить для застосування у випадку розріджених графів — таких, де кількість ребер невелика. Серед переваг застосування матриці інцидентності також варто виокремити константний час доступу до вершин та ребер.

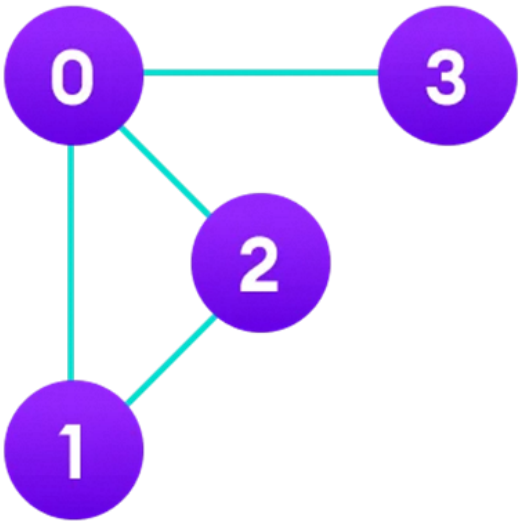
2.2.2. Матриця суміжності

Матриця суміжності — двовимірна матриця, в якій рядки подають початкові вершини, а стовпці — кінцеві вершини.

Матриця суміжності це квадратна матриця розміру V, в якій значення елемента a_{ij} рівне числу ребер з `i`-ї вершини графа в `j`-у вершину.

Іноді, особливо у разі неорієнтованого графа, петля (ребро з `i`-ї вершини в саму себе) вважається за два ребра, тобто значення діагонального елемента a_{ii} в цьому випадку рівне подвоєному числу петель навколо `i`-ї вершини.

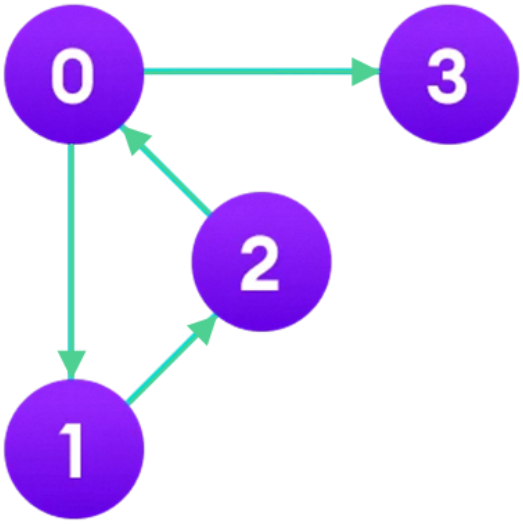
Для *неорієнтованого* графа елемент матриці суміжності приймає значення `1`, якщо вершини, що відповідають йому, поєднані ребром; в інших випадках значенню елемента присвоюється `0`. (рис. 2.5).



M	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Рис. 2.5. Неорієнтований граф та його матриця суміжності

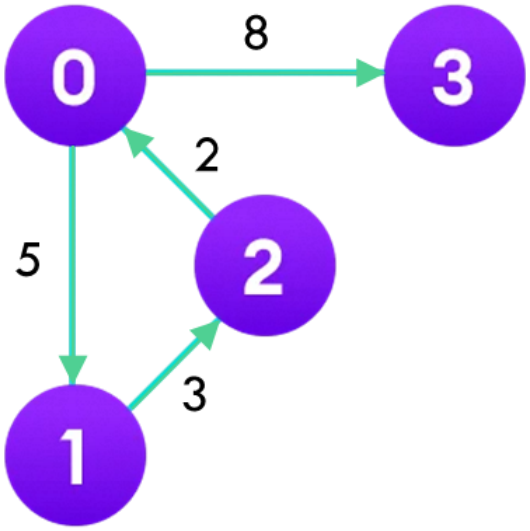
Для *орієнтованого* графа елемент a_{ij} приймає значення 1, якщо ребро виходить з `i`-ї вершини графа в `j`-у вершину, інакше елементу присвоюється `0` (рис. 2.6).



M	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	0	0

Рис. 2.6. Орієнтований граф та його матриця суміжності

Для *зваженого* графа елемент матриці приймає значення ваги (рис. 2.7).



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

Рис. 2.7. Зважений орієнтований граф та його матриця суміжності

Серед недоліків матриці суміжності варто відзначити наступні:

- витрачає багато пам'яті для зберігання нулів у випадку розрідженого графа (граф з малою кількістю ребер);
- повільна в переборі всіх ребер і вершин.

Просторова та часові складності для різних операцій над графом з кількістю вершин V та кількістю ребер E в нотації «O велике» наступні:

- $O(V^2)$ — пам'ять для зберігання графа;
- $O(V^2)$ — час на додавання нової вершини в граф;
- $O(1)$ — час на додавання нового ребра в граф;
- $O(V^2)$ — час на видалення вершини з графа;
- $O(1)$ — час на видалення ребра з графа;
- $O(1)$ — час на визначення суміжності вершин графа.

Матриця суміжності добре підходить для застосування у щільних графах — таких, де кількість ребер близька до квадрату кількості вершин. Серед переваг застосування матриці суміжності також варто виокремити константний час доступу до вершин та ребер, а також додавання та видалення ребер.

2.2.3. Список суміжності

Список суміжності — подає граф у вигляді масиву списків суміжних вершин. Індекс масиву позначає номер вершини, а кожен елемент масиву це зв'язаний список суміжних вершин.

Для *неорієнтованого* графа список суміжності буде містити всі суміжні вершини для кожної вершини (рис. 2.8).

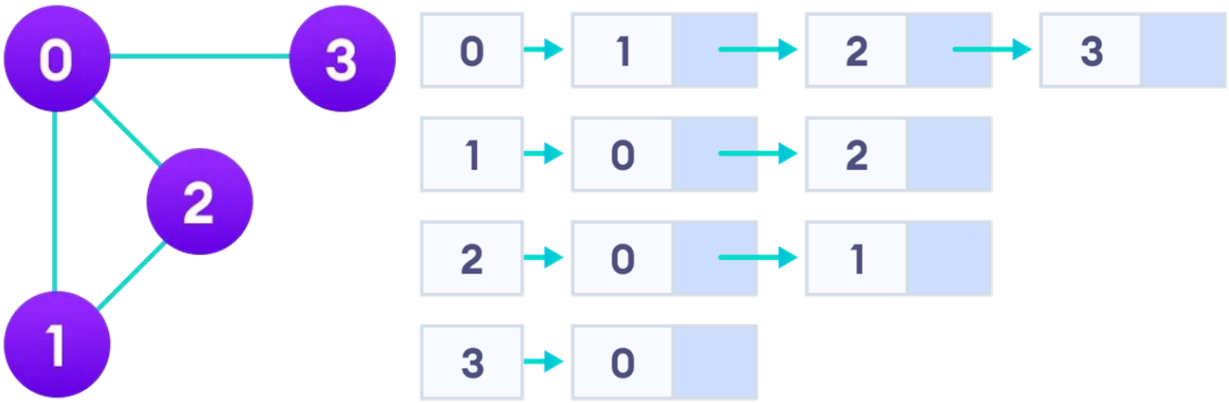


Рис. 2.8. Неорієнтований граф та його список суміжності

Для *орієнтованого* графа використовується лише один з напрямків залежно від поставленого завдання. Це може бути напрямок виходу ребер з вузла (рис. 2.9) або напрямок входу ребер у вузол.

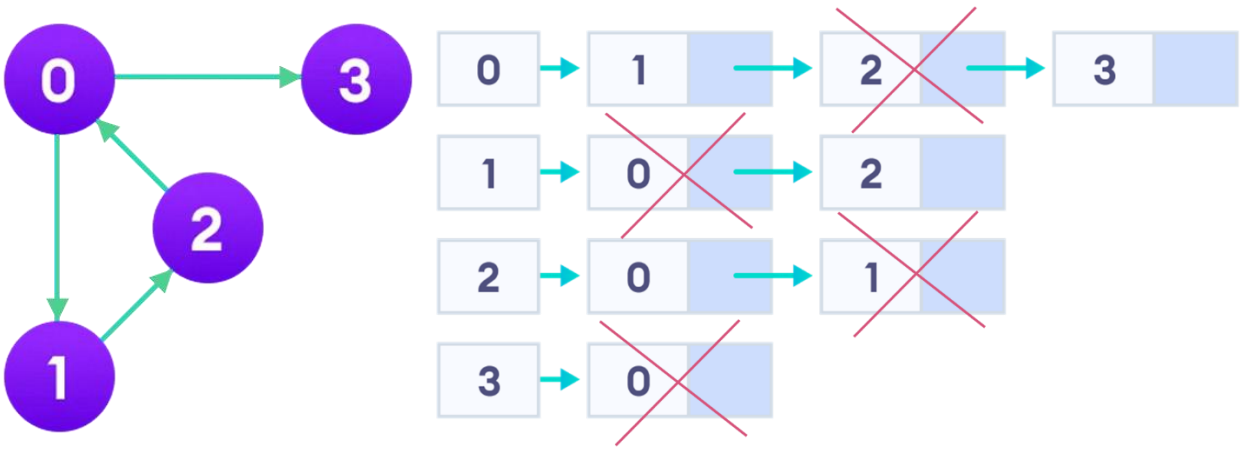


Рис. 2.9. Орієнтований граф та його список суміжності

Для *зваженого* графа елементи зв'язного списку мають додаткові значення — ваги ребер (рис. 2.10).

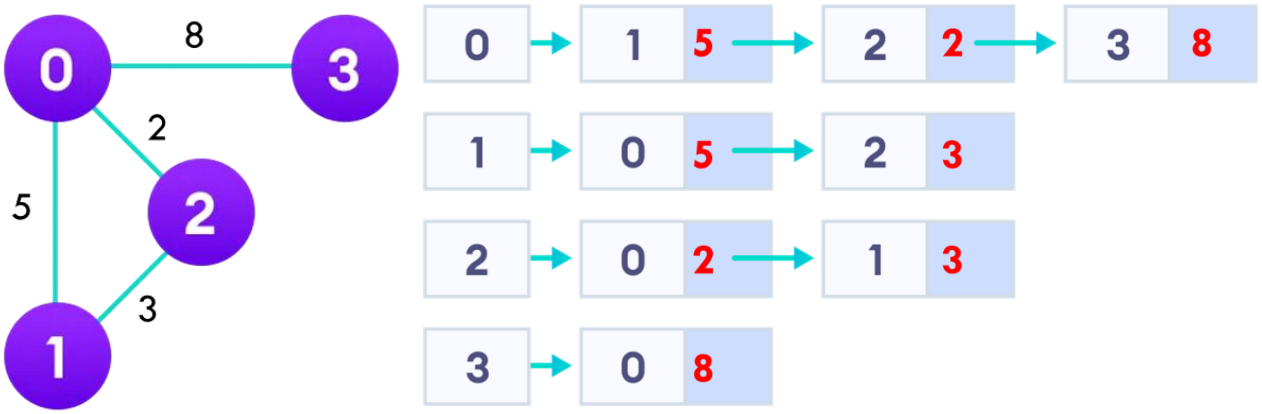


Рис. 2.10. Зважений граф та його список суміжності

Серед недоліків списку суміжності варто відзначити наступні:

- знаходження потрібного ребра значно повільніше, ніж в матриці суміжності, оскільки потрібно просуватися по зв'язному списку;
- визначення суміжності двох вершин повільніше, ніж в матриці суміжності.

Просторова та часові складності для різних операцій над графом з кількістю вершин V та кількістю ребер E в нотації «O велике» наступні:

- O(V + E) — пам'ять для зберігання графа;
- O(1) — час на додавання нової вершини в граф;
- O(1) — час на додавання нового ребра в граф;
- O(E) — час на видалення вершини з графа;
- O(V) — час на видалення ребра з графа;
- O(V) — час на визначення суміжності вершин графа.

Список суміжності зазвичай є кращим, оскільки він ефективніше, з точки зору використання простору, подає розріджені графи. Також використання списків суміжності значно ефективніше для задач з динамічно змінюваними графами — там де потрібно багато додавати і видаляти вершини та ребра.

Найпростіший список зв'язності потребує структуру для опису вузла списку та масив таких вузлів, що подає граф. Будемо розглядати граф у якого вершини пронумеровані числами, а не позначені буквами. Тоді індекс масиву буде відповідати номеру вершини графа. Для незваженого графа структура буде наступною:

```
struct node
{   int vertex;
    struct node* next;
};
```

В цій структурі вузла є номер вершини і покажчик на наступний вузол в списку. Тоді власне граф буде описаний масивом таких вузлів. Якщо максимальна кількість вершин відома наперед, то можна навіть використати статичний масив.

Для зваженого графа структура розшириться додаванням власне ваги ребра, що поєднує поточну вершину з її суміжною вершиною:

```
struct node
{   int vertex;
    int weight;
    struct node* next;
};
```

Такого опису вузла зваженого графа достатньо для вирішення будь-яких задач на графах. Для спрощення використання до структури вузла можна додати булеве поле щоб відмічати чи був цей вузол опрацьований.

2.3. Топологічне сортування вершин графа

Якщо заданий орієнтований граф без циклів, то для нього можна виконати *топологічне сортування*. Під топологічним сортуванням графа розуміють процес лінійного впорядковування його вершин таким чином, що якщо в графі існує ребро (A, B), то, в упорядкованому списку вершин графа, вершина A передує вершині B (рис. 2.11). Якщо в орієнтованому графі є цикли, то упорядкованого таким чином списку для нього не існує.

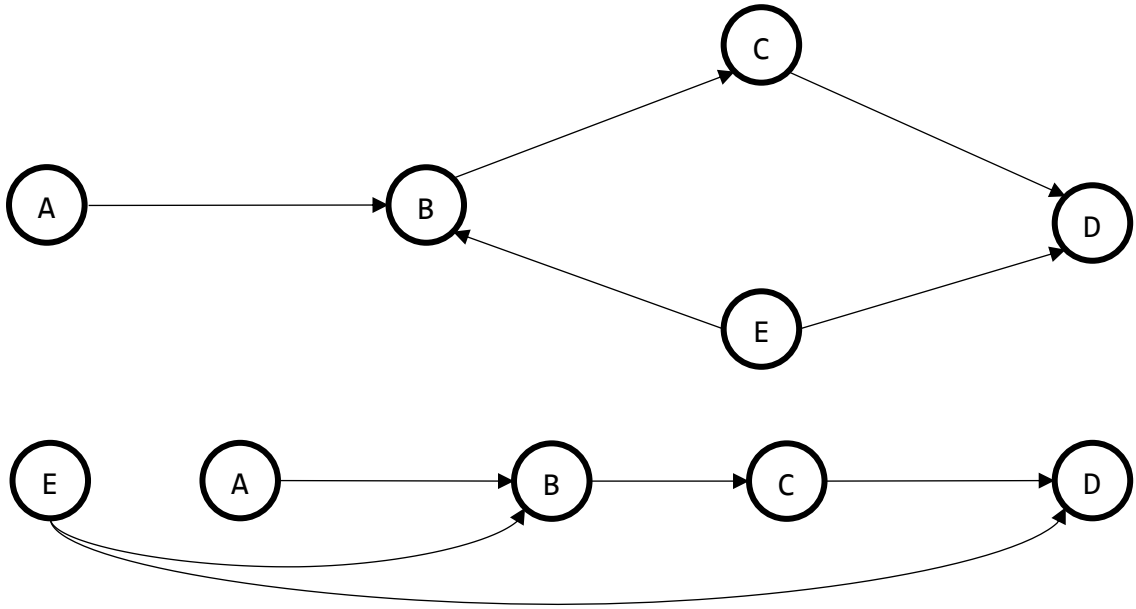


Рис. 2.11. Приклад топологічного сортування

Відмітимо, що задачу про топологічне сортування можна переформулювати наступним чином: розмістити вершини орієнтованого графа на горизонтальній прямій таким чином, щоб всі його ребра йшли зліва направо. У житті це відповідає, наприклад, наступним задачам: в якому порядку слід здійснювати

розв'язок задачі, якщо вона розпадається на підзадачі і виконання деяких підзадач, зазвичай, починається слідом за закінченням інших; в якому порядку слід розташовувати теми в шкільному курсі математики, якщо для кожної теми відомо, знання яких інших тем необхідні для її вивчення; в якому порядку необхідно розподілити опис функцій в програмі, якщо деякі функції можуть містити звернення до інших функцій.

В загальному випадку, топологічне сортування, якщо воно існує, то не обов'язково єдине. Наприклад, на рис. 2.11, показано орієнтований граф і один із варіантів топологічного порядку його вершин: E, A, B, C, D. Вершини A та E можуть розміщуватись по іншому, спочатку A, а потім E. В іншому, порядок топологічного сортування вершин цього графа фіксований.

Одним з алгоритмів розв'язання задачі топологічного сортування є алгоритм Кана (1962). Основна суть цього алгоритму полягає в тому, що на кожній з його ітерацій, здійснюється визначення вершини-джерела орієнтованого графа що залишився (джерелом називається вершина в яку не входить жодне ребро), і, в подальшому, видалення його з усіма, що виходять з нього, ребрами. Якщо таких вершин декілька, довільним чином обирається одна з них. Порядок видалення таким чином вершин, дає рішення задачі про топологічне сортування. Відмітимо, що якщо на деякому кроці виявиться, що для орієнтованого графа, що залишився, вершини-джерела не існує, то задача про топологічне сортування розв'язків не має.

Нижче подано псевдокод алгоритму Кана. Його часова складність O(V + E).

```
L ← Порожній список, що буде містити відсортовані елементи
S ← Набір вершин без ребер, що входять
доки S не порожнє виконати
    видалити вершину n з S
    вставити n в L
    для кожної вершини m з ребром e з n до m виконувати
        видалити ребро e з графа
        якщо m не має більше ребер, що входять тоді
            вставити m в S
якщо граф має ребра тоді
    вивести повідомлення про помилку (граф має принаймні один цикл)
інакше
    вивести повідомлення (пропоноване топологічне сортування: L)
```

Розглянемо приклад покрокового виконання алгоритму Кана. Нехай заданий орієнтовний граф (рис. 2.12).

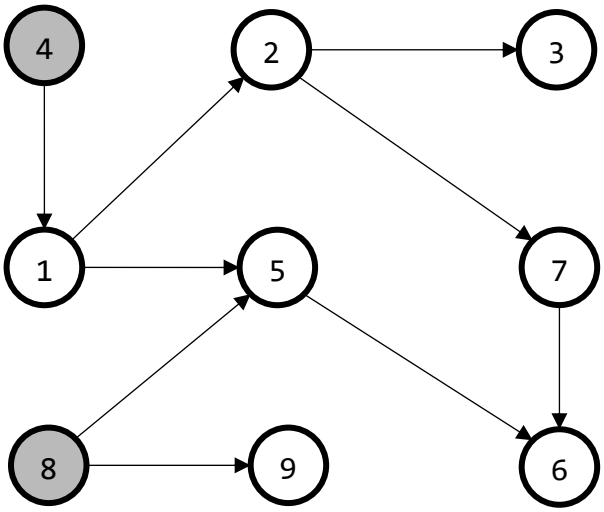


Рис. 2.12. Початковий орієнтований граф

На першому кроці ми знаходимо вершини, які не мають вхідних ребер і додаємо їх у чергу. В нашому випадку таких вершин дві 4 і 8. Далі дістаємо із черги першу вершину і видаляємо з графа всі ребра, що виходять з неї. В нашому випадку ми беремо вершину 4 і видаляємо ребро, що йде до вершини 1 (рис. 2.13). Водночас після кожного видалення ребра ми перевіряємо яка кількість вхідних ребер залишилася для суміжної вершини.

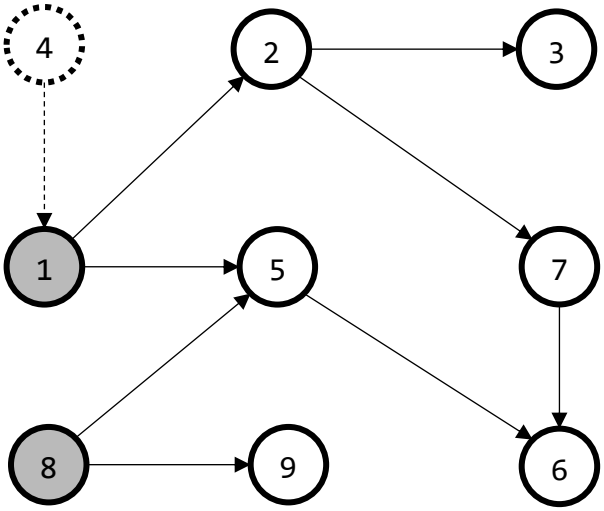


Рис. 2.13. Граф після видалення вершини 4

Оскільки під час видалення ребра з вершини 4 у вершину 1 кількість вхідних ребер для вершини 1 стала рівною 0, то вершина 1 додається у чергу.

Наступної вершиною в черзі є вершина 8. Дістаємо її з черги і видаляємо її вихідні ребра (рис. 2.14).

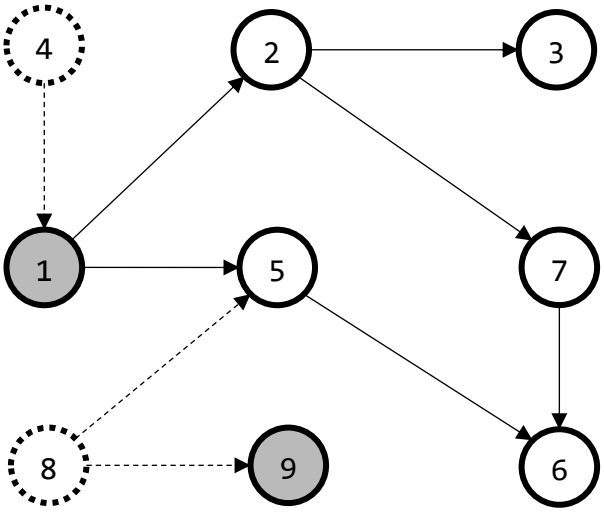


Рис. 2.14. Граф після видалення вершини 8

Після видалення кожного ребра перевіряємо суміжні вершини на кількість вхідних ребер. Після видалення ребра з вершини 8 до вершини 9, остання більше не має вхідних ребер, тому додаємо її в чергу.

Наступної вершиною в черзі є вершина 1. Дістаємо її з черги і видаляємо її вихідні ребра (рис. 2.15).

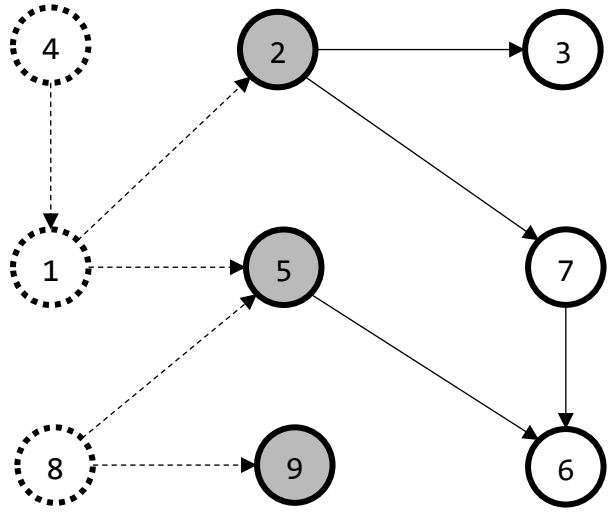


Рис. 2.15. Граф після видалення вершини 1

Після видалення ребер з вершини 1 вершини 2 і 5 більше не мають вхідних ребер, тому додаємо їх в чергу.

Наступної вершиною в черзі є вершина 9. Дістаємо її з черги і оскільки вона не має вихідних ребер, ми переходимо до наступної вершини 2. Видаляємо вихідні ребра з вершини 2 (рис. 2.16).

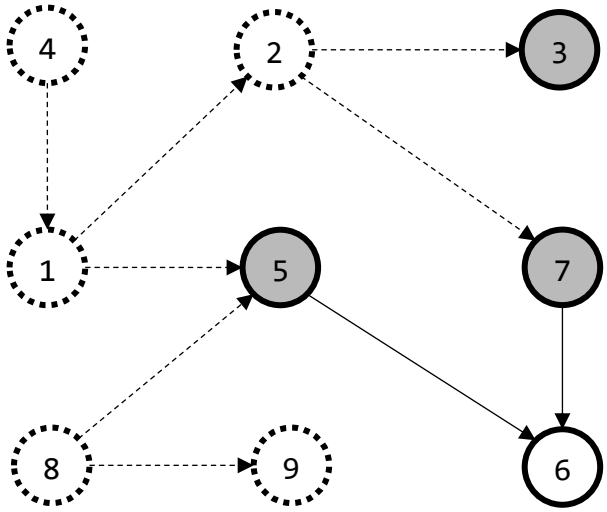


Рис. 2.16. Граф після видалення вершини 2

Після видалення вершини 2 і її вихідних ребер, вершини 3 і 7 залишилися без вхідних ребер, тому додаємо їх в чергу.

Наступною вершиною в черзі є вершина 5. Видаляємо її єдине ребро до вершини 6. Оскільки вершина 6 все ще має вхідне ребро, то ми її не додаємо в чергу.

Наступною з черги беремо вершину 3, яка не має вихідних ребер. Тому наступною беремо вершину 7 і видаляємо її єдине ребро, що сполучає її з вершиною 6 (рис. 2.17).

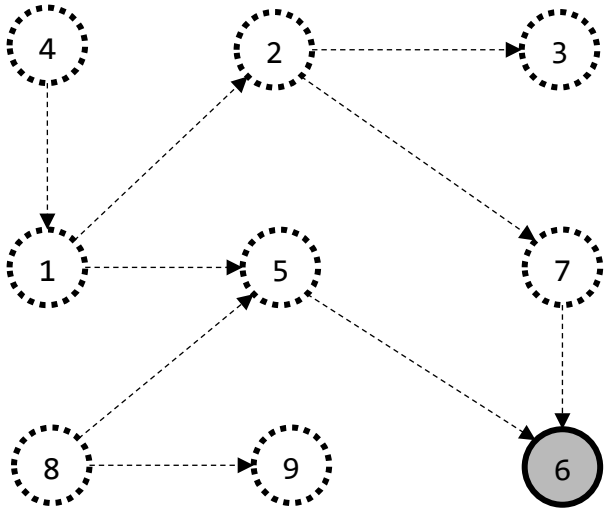


Рис. 2.17. Граф після видалення вершини 7

Вершину 6 додаємо в чергу, оскільки вона більше не має вхідних ребер. Це була остання вершина в графі, отже ми сформували правильний порядок виведення вершин: 4, 8, 1, 9, 2, 5, 3, 7, 6 (рис. 2.18).

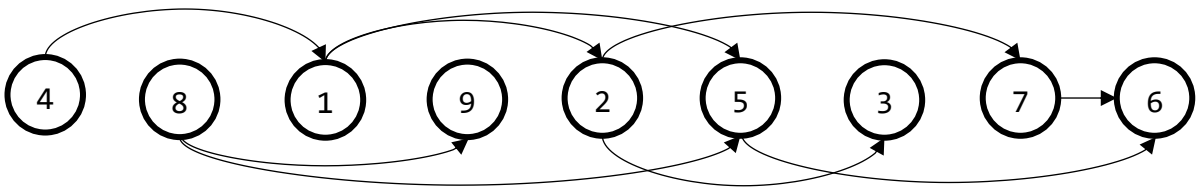


Рис. 2.18. Топологічно відсортований граф

2.4. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач на графах з використанням різних структур даних.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням графів.

Низький рівень складності

1. Find the Town Judge https://leetcode.com/problems/find-the-town-judge.
2. Find Center of Star Graph https://leetcode.com/problems/find-center-of-star-graph.
3. Find if Path Exists in Graph https://leetcode.com/problems/find-if-path-exists-in-graph.

Середній рівень складності

1. Course Schedule https://leetcode.com/problems/course-schedule.
2. Course Schedule II https://leetcode.com/problems/course-schedule-ii.
3. Minimum Height Trees https://leetcode.com/problems/minimum-height-trees.
4. Redundant Connection https://leetcode.com/problems/redundant-connection.
5. Flower Planting With No Adjacent https://leetcode.com/problems/flower-planting-with-no-adjacent.

Високий рівень складності

1. Parallel Courses III https://leetcode.com/problems/parallel-courses-iii.
2. Build a Matrix with Conditions https://leetcode.com/problems/build-a-matrix-with-conditions.

2.5. Розбір деяких задач

Розглянемо декілька завдань середнього рівня складності та проаналізуємо яким чином застосувати до них теорію графів. До кожного завдання вказаний його номер на платформі LeetCode.

2.5.1. Розклад курсів II

Завдання. Задана кількість курсів від `0` до `N-1` і масив з `N` пар курсів, що описують послідовність їх проходження: перший курс повинен проходитися після другого. Повернути масив з номерами курсів у порядку їх проходження згідно заданих послідовностей. Якщо неможливо пройти всі курси, то повернути порожній масив. (*Середній рівень складності — №210*).

Розв'язання. З опису зрозуміло, що нам потрібно побудувати орієнтований граф, де вершинами будуть курси, а дугами — послідовності їх виконання. Тут головне не заплутатися в парах курсів — напрямок дуги від другого до першого в парі.

Після побудови графа, необхідно виконати топологічне сортування вершин за алгоритмом Кана.

Для зберігання орієнтованого графа доцільно обрати список суміжності. Оскільки нас цікавлять лише номери вершин, то його можна побудувати з використанням типу `vector` бібліотеки `std`.

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
vector<int> findOrder(int n, vector<vector<int>>& edges)
{
    queue<int> q;
    vector<int> adj[n], indegree(n,0), ans;
    for(auto& x : edges)
    {
        adj[x[1]].emplace_back(x[0]);
        indegree[x[0]]++;
    }
    for(int i = 0; i < n; i++)
    {
        if(indegree[i] == 0) q.emplace(i);
    }
    while(!q.empty())
    {
        auto front = q.front();
        q.pop();
        ans.emplace_back(front);
        for(auto& x : adj[front])
        {
            indegree[x]--;
            if(indegree[x] == 0) q.emplace(x);
        }
    }
    return ans.size() != n ? vector<int>{} : ans;
}
```

2.5.2. Знаходження безпечних вершин

Завдання. Заданий орієнтований граф з вершинами, що пронумеровані від `0` до `N-1`. Граф поданий у вигляді двовимірного масиву, що фактично є списком суміжності — для кожного `i`-го елементу масиву вказані номери вершин, які є суміжними до цієї вершини.

Вершина називається *термінальною*, якщо у неї немає вихідних ребер. Вершина називається *безпечною*, якщо будь-який маршрут з неї веде до термінальної вершини. Потрібно повернути масив з номерами безпечних вершин в порядку їх зростання. (*Середній рівень складності — №802*).

Приклад. Вхідний граф заданий таким двовимірним масивом, що є списком суміжності: [[1,2], [2,3], [5], [0], [5], [], []] (рис. 2.19).

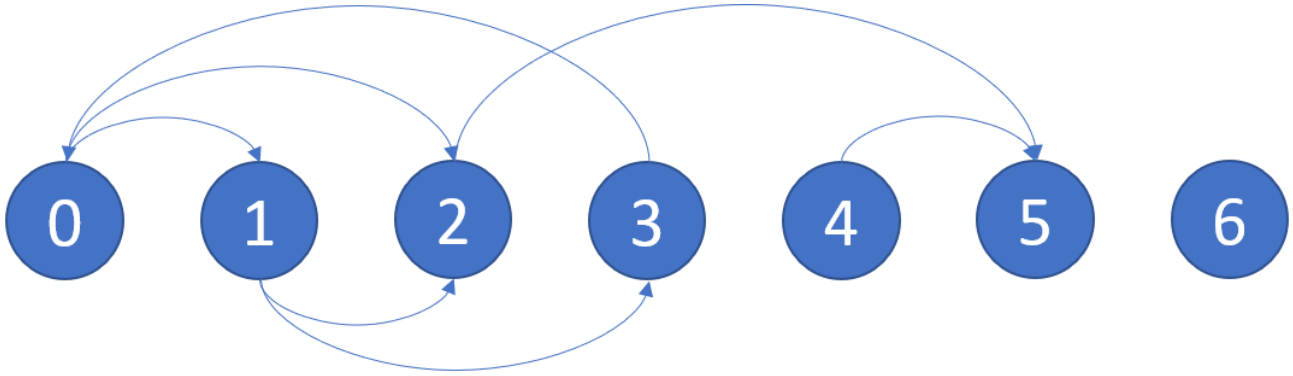


Рис. 2.19. Вхідний граф до завдання

Термінальними вершинами тут є вершини 5 і 6, оскільки в них відсутні вихідні ребра. Всі інші вершини потенційно можуть бути безпечними. Розглянемо кожну з них окремо.

З вершини 0 ми маємо ребра до вершин 1 і 2. З вершини 1 можна потрапити у вершини 2 і 3. З вершини 2 єдине вихідне ребро веде до термінальної вершини 5, а отже вершина 2 точно безпечна. З вершини 3 єдине вихідне ребро веде до вершини 0, яка ще не визначена як безпечна.

Як ми бачимо на графі вершини 0, 1 і 3 утворюють цикл. Це означає, що не всі маршрути з цих вершин ведуть до термінальних вершин, а отже, вони не можуть бути безпечними.

Остання вершина, яку необхідно розглянути, це вершина 4, з якої єдине вихідне ребро веде в термінальну вершину 5, а отже вона є безпечною.

Результат для цього графа має бути таким: [2, 4, 5, 6].

Розв'язання. Необхідно для кожної вершини перевірити чи є вона безпечною чи ні. Якщо вершина має вхідні і вихідні ребра вона може бути безпечною, але також може і не бути, якщо є частиною циклічного шляху. Ми можемо застосувати алгоритм Кана для пошуку топологічного сортування, але якщо в графі є цикл, то ми зупинимося не досягнувши термінальних вершин.

Якщо нам відомо, що термінальні вершини мають лише вхідні дуги, то ми можемо почати рухатися з них і шукати всі вершини з яких можна потрапити в термінальні. Фактично нам потрібно запустити алгоритм Кана в зворотному напрямку. Тобто розвернути напрямок ребер графа і рухатися вже по ним, водночас маркуючи знайдені вершини як безпечні. Якщо в процесі руху ми натрапимо на цикл, то алгоритм зупиниться і всі вершини, що є в циклі не будуть

позначені як безпечні. Після зупинки алгоритму Кана потрібно пройтися по кожній вершині від початкової до останньої і вивести лише ті, які позначені як безпечні.

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
vector<int> eventualSafeNodes(vector<vector<int>>& graph)
{   int n = graph.size(); vector<int> indegree(n);
    vector<vector<int>> adj(n);
    for (int i = 0; i < n; i++)
    {
        for (auto node : graph[i])
        {   adj[node].push_back(i);
            indegree[i]++;
        }
    }
    queue<int> q; vector<bool> safe(n); vector<int> safeNodes;
    for (int i = 0; i < n; i++) if (indegree[i] == 0) q.push(i);
    while (!q.empty())
    {   int node = q.front(); q.pop();
        safe[node] = true;
        for (auto& neighbor : adj[node])
        {   indegree[neighbor]--;
            if (indegree[neighbor] == 0) q.push(neighbor);
        }
    }
    for(int i = 0; i < n; i++) if(safe[i]) safeNodes.push_back(i);
    return safeNodes;
}
```

Запитання для самоконтролю

1. Що таке граф?
2. Які основні терміни теорії графів ви знаєте?
3. Що таке матриця інцидентності?
4. Що таке матриця суміжності?
5. Що таке список суміжності?
6. Які переваги та недоліки матриці суміжності?
7. Які переваги та недоліки списку суміжності?
8. Що таке топологічне сортування графа?

ПОСИЛАННЯ НА ВИКОРИСТАНІ ДЖЕРЕЛА

1. https://uk.wikipedia.org/wiki/Теорія_графів
2. https://uk.wikipedia.org/wiki/Граф_(математика)
3. https://uk.wikipedia.org/wiki/Граф_(абстрактний_тип_даних)
4. https://uk.wikipedia.org/wiki/Матриця_інцидентності
5. https://uk.wikipedia.org/wiki/Матриця_суміжності
6. https://uk.wikipedia.org/wiki/Топологічне_сортування
7. https://algoua.com/algorithms/graphs/topological_sort/
8. https://www.mathros.net.ua/topologichne-sortuvannja-vershyn-orijentovanogo-grafa.html
9. https://en.wikipedia.org/wiki/Topological_sorting
10. https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution
11. https://en.wikipedia.org/wiki/Adjacency_matrix
12. https://www.programiz.com/dsa/graph-adjacency-matrix
13. https://www.programiz.com/dsa/graph-adjacency-list

3. МЕТОДИ ОБХОДУ ГРАФА

Існують два основні методи обходу графів — пошук в ширину та пошук в глибину.

3.1. ПОШУК В ШИРИНУ

Пошук в ширину (Breadth-first search, BFS) — один з основних алгоритмів обходу графа. Ми вже використовували основний його принцип для реалізації алгоритму Кана. Також він використовується для алгоритмів Дейкстри та Прима, які ми будемо розглядати пізніше.

3.1.1. ПРИНЦИП АЛГОРИТМУ BFS

Згідно з цим алгоритмом обхід вершин заданого графа G здійснюється в порядку їх віддаленості від деякої заздалегідь обраної, або зазначеної як початкова, вершини S (називається коренем пошуку в ширину). Інакше кажучи, спочатку відвідується сама вершина S, потім всі вершини, суміжні з S, тобто ті, які знаходяться від неї на відстані в одне ребро, потім вершини, що знаходяться від S на відстані в два ребра, і так далі.

В результаті пошуку в ширину знаходиться шлях найкоротшої довжини у незваженому графі, тобто шлях, що містить найменшу кількість ребер.

На вхід алгоритму подається незважений граф і номер стартової вершини S. Граф може бути як орієнтованим, так і неорієнтованим, для алгоритму це не важливо.

Розглянемо послідовність виконання алгоритму. Спочатку всі вершини позначаються як непройдені. Першою відвідується вершина S, вона стає єдиною пройденою активною вершиною. На наступному кроці, досліджуються ребра, інцидентні цій вершині. У загальному випадку, за такого дослідження, можливі два варіанти подальших дій:

- якщо всі ребра, інцидентні вершині переглянуті, вона перестає бути активною і перетворюється в повністю скановану вершину. В такому випадку вибирається нова активна вершина, і описані дії повторюються;
- якщо існують непереглянуті ребра, інцидентні S, то досліджуємо їх. Якщо таке ребро з'єднує вершину S з новою вершиною, наприклад B, то вершина відвідується і перетворюється у пройдену активну вершину. Якщо ж вершина B, була пройдена раніше, то продовжуємо пошук іншого непройденого ребра, інцидентного до S.

Сам алгоритм можна сприймати як процес «підпалювання» графа: на нульовому кроці підпалюємо тільки вершину S. На кожному наступному кроці вогонь з кожної вершини, що вже горить, поширюється на всіх її сусідів; тобто за одну ітерацію алгоритму відбувається розширення «кільця вогню» в ширину на одиницю (звідси і назва алгоритму).

Для реалізації списку вершин найчастіше використовується черга та принцип `FIFO`. Виконання алгоритму продовжується до досягнення шуканої вершини або

до того часу, коли на певному кроці в список не включається жодна вершина. Другий випадок означає, що всі вершини, доступні з початкової, уже відмічені, як пройдені, а шлях до цільової вершини не знайдений.

3.1.2. Формальний опис та псевдокод

Формальний опис алгоритму обходу в ширину всіх вершин графа наступний.

1. Почати з довільної вершини графа S додавши її в кінець черги та позначивши її як вже відвідану.
2. Дістати вершину з голови черги і розглянути всі суміжні вершини.
3. Якщо суміжна вершина ще не була відвідана, то додати її в кінець черги і позначити як відвідану. Якщо вже була відвідана — пропустити.
4. Повторювати кроки 2 і 3 до тих пір, поки черга не стане порожньою.

В результаті, коли черга стане порожньою, обхід в ширину обійде всі досяжні з S вершини, причому до кожної дійде найкоротшим шляхом. Також можна порахувати довжини найкоротших шляхів (для чого треба створити масив довжин шляхів D), і компактно зберегти інформацію, якої достатньо для відновлення всіх цих найкоротших шляхів (для цього треба створити масив «предків» P, в якому для кожної вершини зберігати номер вершини, з якої ми потрапили в цю вершину).

Часова складність алгоритму BFS O(V + E). Просторова складність O(V). Псевдокод алгоритму BFS для пошуку конкретної вершини графа поданий нижче.

```
procedure BFS(G, root) is
    let Q be a queue
    label root as explored
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as explored then
                label w as explored
                w.parent := v
                Q.enqueue(w)
```

В поданому псевдокоді, граф G заданий списком суміжності, кожним елементом якого є вузол, що описаний у вигляді структури з полями. Які саме поля включати в цю структуру залежить від конкретної поставленої задачі.

3.1.3. Приклад використання BFS

Розглянемо для прикладу граф і виконаємо у ньому пошук в ширину з початкової вершини S (рис. 3.1).

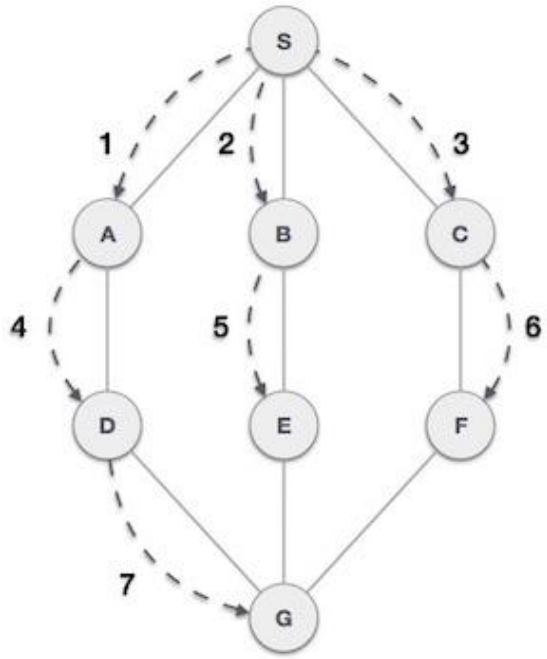


Рис. 3.1. Граф та порядок обходу вершин за BFS

На *першому* кроці ми ініціалізуємо чергу та позначаємо вершину S як таку, що вже відвідана (рис. 3.2).

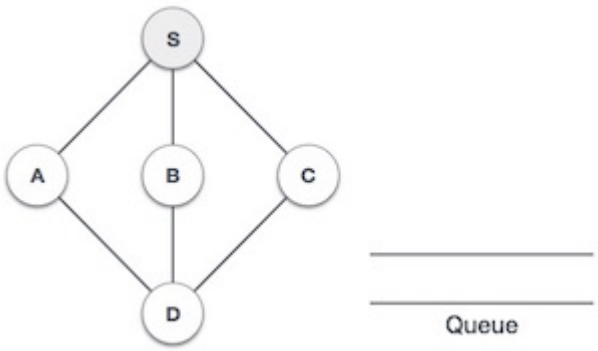


Рис. 3.2. Перший крок алгоритму BFS

На *другому* кроці ми перевіряємо всі суміжні вершини до S, обираємо найпершу невідвідану та додаємо її в чергу (рис. 3.3).

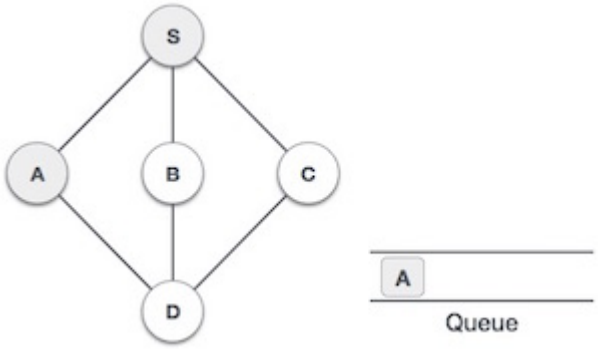


Рис. 3.3. Другий крок алгоритму BFS

На *третьому* кроці ми додаємо наступну невідвідану суміжну вершину до вершини S — це вершина B (рис .3.4).

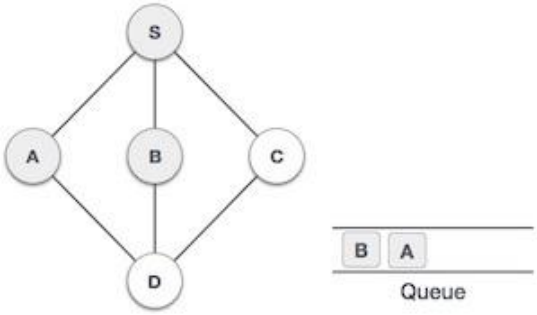


Рис. 3.4. Третій крок алгоритму BFS

На *четвертому* кроці алгоритму ми додаємо останню невідвідану суміжну вершину до S — це вершина C (рис. 3.5).

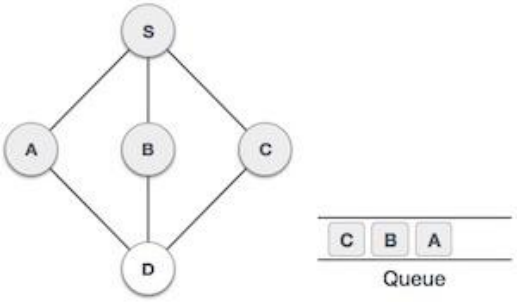


Рис. 3.5. Четвертий крок алгоритму BFS

На *п'ятому* кроці в нас немає невідвіданих суміжних вершин до вершини S. Тому ми дістаємо з черги вершину A (рис. 3.6).

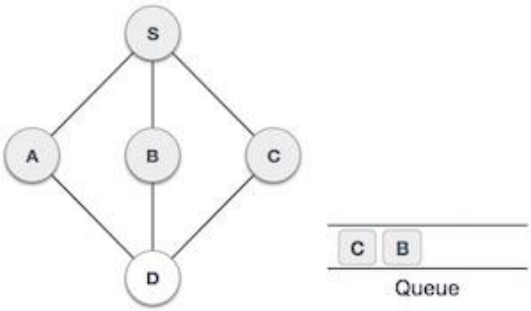


Рис. 3.6. П'ятий крок алгоритму BFS

На *шостому* кроці ми додаємо єдину невідвідану суміжну вершину до вершини A — це вершина D (рис. 3.7).

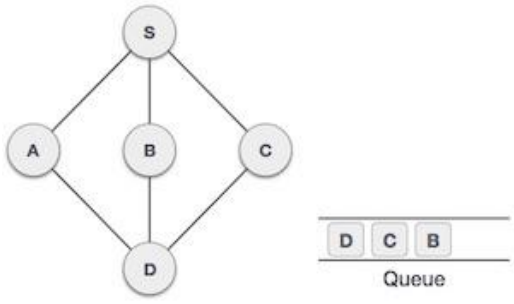


Рис. 3.7. Шостий крок алгоритму BFS

Тепер в нас не залишилося жодної невідвіданої вершини, а тому все, що нам потрібно це діставати вершини з черги поки вона не стане порожньою. На цьому робота алгоритму BFS закінчується.

3.1.4. Застосування BFS

Як вже зазначалося, алгоритм BFS є складовою частиною багатьох інших алгоритмів. В чистому вигляді його можна застосовувати для знаходження найкоротшого маршруту між вершинами в незваженому графі. Нижче подано перелік деяких задач, які розв'язуються із застосуванням алгоритму пошуку в ширину.

1. Пошук компонент зв'язності у графі за O(V + E). Для цього запускаємо обхід в ширину від кожної вершини, за винятком вершин, що залишилися відвіданими після попередніх запусків. Таким чином, ми виконуємо звичайний запуск в ширину від кожної вершини, але не обнуляємо кожний раз масив де зберігаються позначки відвіданих вершин, за рахунок цього ми кожен раз будемо обходити нову компоненту зв'язності, а сумарний час роботи алгоритму буде як і раніше O(V + E).
2. Знаходження розв'язку будь-якої задачі з *найменшим числом ходів*, якщо кожний стан системи можна подати вершиною графа, а переходи з одного стану в інші — ребрами графа. Класичний приклад — гра, де робот рухається по полю, водночас він може переміщати ящики, що знаходяться на цьому ж полі, і потрібно за найменше число ходів пересунути ящики в необхідні позиції. Розв'язується обходом в ширину по графу, де станом (вершиною) є набір координат: координати робота, і координати всіх коробок.
3. Знаходження найкоротшого шляху в *0-1-графі* (зважений граф, але з вагами рівними тільки 0 або 1). Достатньо трохи модифікувати пошук в ширину: якщо поточне ребро нульової ваги, і відбувається покращення відстані до якоїсь вершини, то цю вершину додаємо не в кінець, а в початок черги.
4. Знаходження *найкоротшого циклу* в орієнтованому незваженому графі. Для цього запускаємо пошук в ширину з кожної вершини. Як тільки в процесі обходу ми намагаємося піти з поточної вершини по якомусь ребру до вже відвіданої вершини, то це означає, що ми знайшли найкоротший цикл, і зупиняємо обхід в ширину. Серед всіх таких знайдених циклів (по одному від кожного запуску обходу) вибираємо найкоротший.
5. Знайти всі *ребра*, що лежать на будь-якому найкоротшому шляху між заданою парою вершин (A, B). Для цього треба запустити 2 пошуки в ширину: з вершини A, та з вершини B. Позначимо через D_A — масив найкоротших відстаней, отриманий в результаті першого обходу, а через D_B — в результаті другого обходу. Тепер для будь-якого ребра (U, V) легко перевірити чи лежить воно на будь-якому найкоротшому шляху за формулою $D_A[U] + 1 + D_B[V] = D_A[B]$.

6. Знайти всі *вершини*, що лежать на будь-якому найкоротшому шляху між заданою парою вершин (A, B). Вирішується аналогічно попередньому лише у формулі не додаємо одиницю.

3.2. Пошук в глибину

Пошук в глибину (Depth-first search, DFS) — алгоритм для обходу дерева, структури подібної до дерева, або графа. Робота алгоритму починається з кореня дерева (або іншої обраної вершини в графі) і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

3.2.1. Принцип алгоритму DFS

Пошук в глибину (або обхід в глибину) є одним з основних і найбільш часто вживаних алгоритмів обходу всіх вершин і ребер графа. Згідно з цим алгоритмом обхід здійснюється за наступним правилом: у графі вибираємо довільну вершину, наприклад S, і починаємо з неї пошук. Початкова вершина S (називається коренем пошуку в глибину) після цього вважається пройденою. На наступному кроці, вибираємо ребро (S, B), інцидентне вершині S (орієнтуємо його напрямок з S в B), і з його допомогою, переходимо у вершину B. Відмітимо, що ребро (S, B) після цих дій вважається переглянутим і називається ребром дерева, а вершина S називається батьківською по відношенню до вершини B.

У загальному випадку, коли ми перейшли в будь-яку вершину графа, в нашому випадку це вершина B, виникають два варіанти можливих дій:

- якщо всі ребра, інцидентні B, вже переглянуті, то повертаємося до батьківської вершини для B і продовжуємо пошук з цієї вершини. Вершина B з цього моменту називатиметься повністю сканованою;
- якщо існують непереглянуті ребра, інцидентні B, то вибираємо одне з таких ребер, наприклад, (B, C) і орієнтуємо його з B в C. Ребро (B, C) з цього моменту вважатиметься переглянутим. Водночас, необхідно розглянути два випадки: якщо C раніше не була пройдена, то використовуючи ребро (B, C) переходимо у вершину C і продовжуємо пошук з неї. В цьому випадку ребро (B, C) називається *ребром дерева*, а вершина B — *батьком вершини* C; якщо C раніше була пройдена, то продовжуємо пошук іншого не пройденого ребра, інцидентного B. В цьому випадку ребро (B, C) називається *зворотним ребром*.

Під час пошуку в глибину, коли вершина відвідується в перший раз, їй ставлять у відповідність число, що є порядковим номером проходження вершини. Це число називається *глибиною*.

Обхід графа в глибину завершується, коли ми повертаємося в корінь і всі вершини графа пройдені. Якщо після повернення в корінь залишаються не пройдені вершини, можна вибрати одну з них і повторювати процес поки не пройдемо всі вершини графа.

Для реалізації списку вершин найчастіше використовується стек та принцип LIFO. Також можливо обійтися без прямого використання стеку, а використати рекурсивну функцію для створення стеку виклику функцій.

3.2.2. Формальний опис та псевдокод

Формальний опис алгоритму обходу в ширину всіх вершин графа наступний.

1. Почати з довільної вершини графа S додавши її в стек та позначивши її як вже відвідану.
2. Розглянути вершину з голови стеку. Якщо всі її суміжні вершини вже відвідані, то перейти до кроку 4, інакше — до кроку 3.
3. Знайти першу суміжну вершину, що ще не була відвідана і додати її в голову стеку, позначити як відвідану та перейти до кроку 2.
4. Виключити вершину зі стеку. Якщо стек порожній, то зупинитись, інакше — перейти до кроку 2.

В результаті, коли стек стане порожнім, обхід в глибину обійде всі досяжні з S вершини. Інколи необхідно знати «кольори» вершин:

- 0 — не відвідана вершина;
- 1 — відвідана вершина, але досі у стеці;
- 2 — відвідана і більше немає у стеці.

Часова складність алгоритму DFS O(V + E). Просторова складність O(V). Псевдокод алгоритму DFS для пошуку конкретної вершини графа поданий нижче (версія з рекурсією).

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

Версія без рекурсії, що передбачає використання стеку нижче.

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

В поданих псевдокодах, граф G заданий списком суміжності, кожним елементом якого є вузол, що описаний у вигляді структури з полями. Які саме поля включати в цю структуру залежить від конкретної поставленої задачі.

3.2.3. Приклад використання DFS

Розглянемо для прикладу граф і виконаємо в ньому пошук в глибину з початкової вершини S (рис. 3.8).

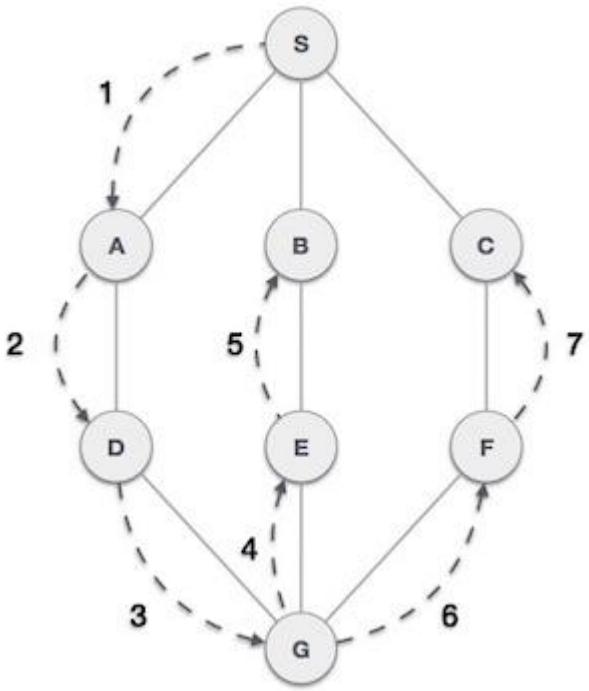


Рис. 3.8. Граф та порядок обходу вершин за DFS

На *першому* кроці ми ініціалізуємо стек, позначаємо вершину S як таку, що вже відвідана та додаємо її в стек (рис. 3.9).

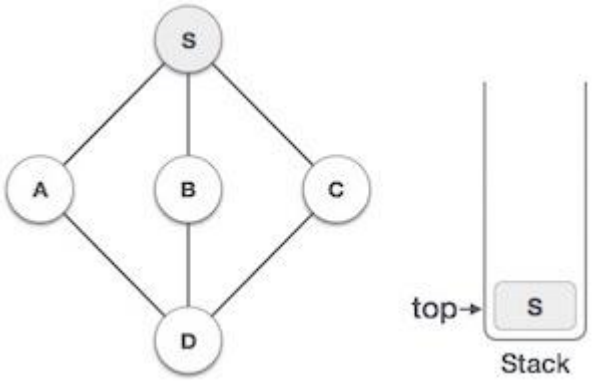


Рис. 3.9. Перший крок алгоритму DFS

На *другому* кроці ми знаходимо будь-яку суміжну вершину до S, що ще не була відвідана, позначаємо її відвіданою та додаємо її в стек (рис. 3.10).

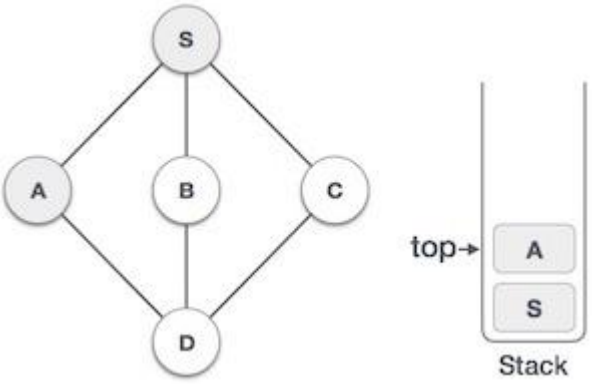


Рис. 3.10. Другий крок алгоритму DFS

На *третьому* кроці знаходимо будь-яку суміжну вершину до A, що ще не була відвідана, позначаємо її відвіданою та додаємо її в стек. Оскільки вершина S вже позначена, як відвідана, тому ми додаємо вершину D (рис .3.11).

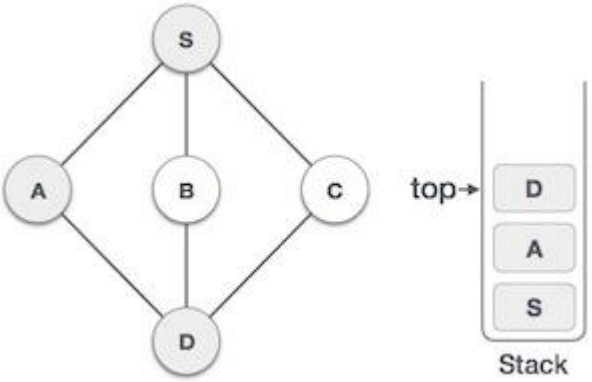


Рис. 3.11. Третій крок алгоритму DFS

На *четвертому* кроці знаходимо будь-яку суміжну вершину до D, що ще не була відвідана, позначаємо її відвіданою та додаємо її в стек. Оскільки вершина A вже позначена, як відвідана, тому ми додаємо вершину B (рис .3.12).

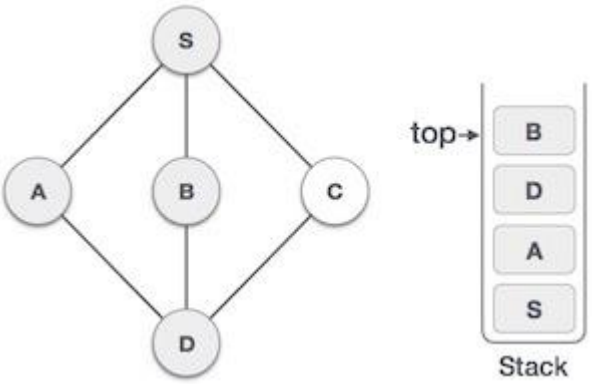


Рис. 3.12. Четвертий крок алгоритму DFS

На *п'ятому* кроці ми перевіряємо вершину B і не знаходимо в неї невідвіданих суміжних вершин, тому видаляємо її зі стеку (рис. 3.13).

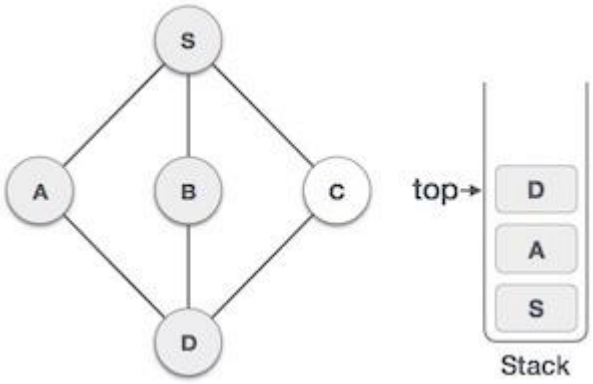


Рис. 3.13. П'ятий крок алгоритму DFS

На *шостому* кроці ми знову перевіряємо вершину D та шукаємо невідвідану суміжну вершину до неї. Єдина така вершина це вершина C, отже ми додаємо її в стек (рис. 3.14).

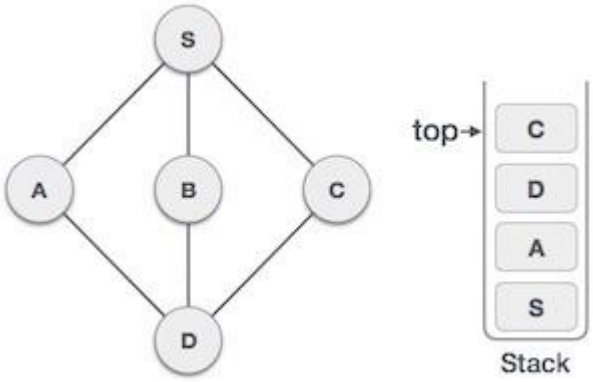


Рис. 3.14. Шостий крок алгоритму DFS

Оскільки вершина C не має невідвіданих суміжних вершин, ми видаляємо її зі стеку. Продовжуємо видаляти зі стеку всі вершини, які не мають невідвіданих суміжних вершин. В нашому випадку це призведе до очищення стеку, а отже, до закінчення алгоритму.

3.2.4. Застосування DFS

Алгоритм DFS передусім застосовується для обходу графа або дерева. Також він є складовою частиною інших алгоритмів, деякі з них розглянуті нижче.

1. Пошук будь-якого шляху у графі.
2. Пошук лексикографічно першого шляху в графі.
3. Перевірка чи одна вершина дерева є предком іншої. На початку і у кінці ітерації пошуку в глибину будемо запам'ятовувати «час» заходу і виходу з кожної вершини. Тепер за O(1) можна знайти відповідь: вершина A є предком вершини B тоді і тільки тоді, коли `time_in`[A] < `time_in`[B] та `time_out`[A] > `time_out`[B].
4. Найменший спільний предок.
5. Топологічне сортування. Ми вже знаємо алгоритм Кана для вирішення цієї задачі, але DFS також дозволяє це зробити. Запускаємо серію пошуків в

глибину, щоб обійти всі вершини графа. Відсортуємо вершини по спаданню часу виходу — це і буде відповіддю.

6. Перевірка графа на ациклічність і знаходження циклу.
7. Пошук компонент сильної зв'язності. Спочатку проводимо топологічне сортування, а потім транспонуємо граф. Після цього знову проводимо серію пошуків у глибину, але в порядку вершин, який було отримано топологічним сортуванням. Кожне дерево пошуку — сильнозв'язана компонента.
8. Пошук мостів. Спочатку перетворюємо граф в орієнтований, роблячи серію пошуків в глибину, і орієнтуючи кожне ребро так, як ми намагалися по ньому пройти. Потім знаходимо сильнозв'язані компоненти. Мостами є ті ребра, кінці яких належать різним сильнозв'язаним компонентам.

3.3. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач на графах з використанням обох варіантів обходу графів.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням або BFS або DFS.

Низький рівень складності

1. Same Tree https://leetcode.com/problems/same-tree.
2. Symmetric Tree https://leetcode.com/problems/symmetric-tree.
3. Maximum Depth of Binary Tree https://leetcode.com/problems/maximum-depth-of-binary-tree.
4. Minimum Depth of Binary Tree https://leetcode.com/problems/minimum-depth-of-binary-tree.
5. Find if Path Exists in Graph https://leetcode.com/problems/find-if-path-exists-in-graph.

Середній рівень складності

1. Maximum Number of Fish in a Grid https://leetcode.com/problems/maximum-number-of-fish-in-a-grid.
2. Check Knight Tour Configuration https://leetcode.com/problems/check-knight-tour-configuration.
3. Minimum Fuel Cost to Report to the Capital https://leetcode.com/problems/minimum-fuel-cost-to-report-to-the-capital.
4. Amount of Time for Binary Tree to Be Infected https://leetcode.com/problems/amount-of-time-for-binary-tree-to-be-infected.
5. Detect Cycles in 2D Grid https://leetcode.com/problems/detect-cycles-in-2d-grid.

Високий рівень складності

1. Last Day Where You Can Still Cross https://leetcode.com/problems/last-day-where-you-can-still-cross.

2. Swim in Rising Water https://leetcode.com/problems/swim-in-rising-water.
3. Contain Virus https://leetcode.com/problems/contain-virus.

3.4. Розбір деяких задач

Розглянемо декілька завдань середнього рівня складності та проаналізуємо яким чином застосувати до них DFS і BFS. До кожного завдання вказаний його номер на платформі LeetCode.

3.4.1. Рівень бінарного дерева з максимальною сумою елементів

Завдання. Задане бінарне дерево, де корінь це рівень 1, наступний рівень 2 і так далі. Потрібно знайти мінімальний рівень дерева, на якому сума елементів максимальна. (*Середній рівень складності — №1161*).

Приклад. Вхідне бінарне дерево задане таким одновимірним масивом, що описує його рівень за рівнем: [1,7,0,7,-8,null,null] (рис. 3.15).

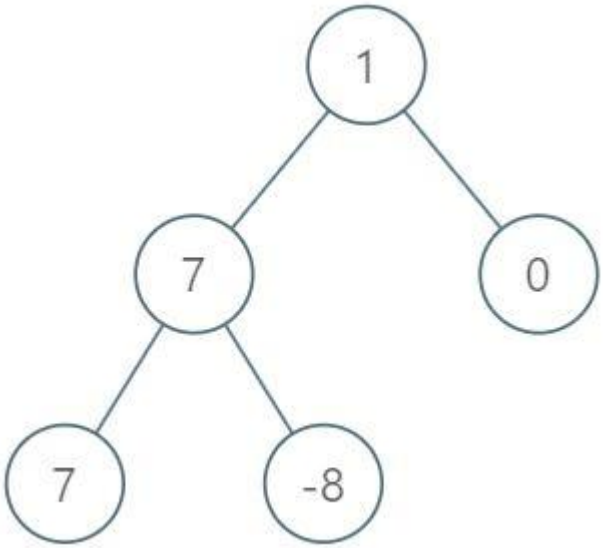


Рис. 3.15. Вхідне бінарне дерево

Сума елементів на першому рівні — 1. Сума на другому рівні — 7 + 0 = 7. Сума на третьому рівні — 7 - 8 = -1. На другому рівні сума найбільша, а тому відповідь 2.

Розв'язання. Оскільки потрібно проходити дерево рівень за рівнем, то тут найкраще підійде метод обходу в ширину — BFS. Єдиний нюанс тут в тому, що потрібно чітко розділяти вузли дерева на кожному рівні. Для цього будемо видаляти з черги вузли не по одному, а групами, що розділені за рівнями.

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
struct TreeNode
{   int val;
    TreeNode *left;
    TreeNode *right;
};
```

```
int maxLevelSum(TreeNode* root)
{   int maxi = INT_MIN;
    queue<TreeNode*> q;
    q.push(root);
    int level = 1, ans = 1;
    while(!q.empty())
    {   int n = q.size(), sum = 0;
        while(n--)
        {   auto temp = q.front();
            q.pop();
            sum += temp->val;
            if(temp->right != NULL) q.push(temp->right);
            if(temp->left  != NULL) q.push(temp->left);
        }
        if(sum > maxi)
        {   maxi=sum;
            ans=level;
        }
        level++;
    }
    return ans;
}
```

3.4.2. Максимальна площа острова

Завдання. Задана карта островів у вигляді двовимірного масиву розмірністю MxN, де значення 0 відповідає воді, а значення 1 — відповідає суші. Островом вважається група з клітинок, що містять одиниці і з'єднані однією з чотирьох сторін (горизонтальними або вертикальними). Потрібно повернути максимальну площу острова або нуль, якщо островів немає. (*Середній рівень складності — №695*).

Приклад. Задана карта островів у вигляді двовимірного масиву: [[0,0,1,0,0,0,0,1,0,0,0,0,0], [0,0,0,0,0,0,0,1,1,1,0,0,0], [0,1,1,0,1,0,0,0,0,0,0,0,0], [0,1,0,0,1,1,0,0,1,0,1,0,0], [0,1,0,0,1,1,0,0,1,1,1,0,0], [0,0,0,0,0,0,0,0,0,0,0,1,0,0], [0,0,0,0,0,0,0,1,1,1,0,0,0], [0,0,0,0,0,0,0,0,1,1,0,0,0,0]] (рис. 3.16).

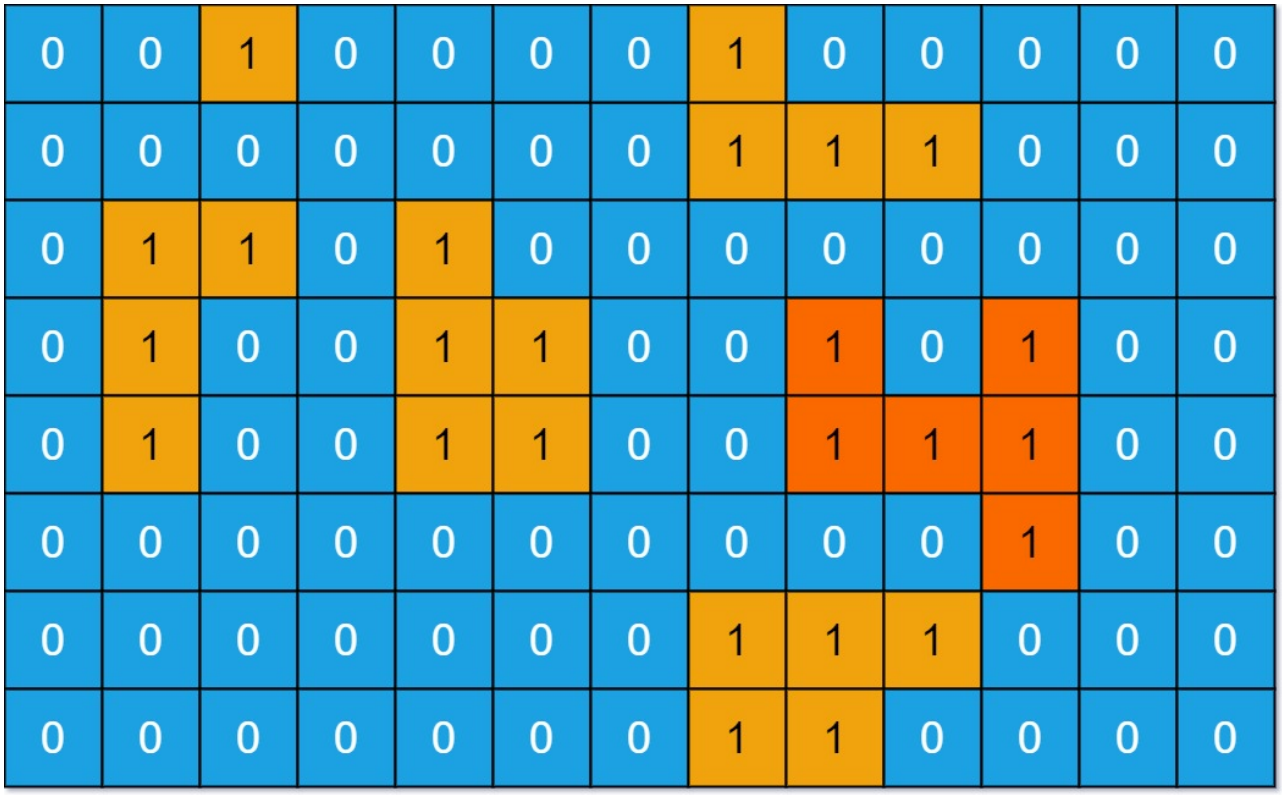


Рис. 3.16. Карта островів

Найбільший за площею острів виділений оранжевим кольором і його площа складає 6. Це і буде відповіддю.

Розв'язання. Всі острови можна зобразити у вигляді графа де вершинами є клітинки, а ребрами будуть зв'язки між ними. Тепер можна запускати обхід в глибину або ширину з кожної клітинки, що позначає сушу і рахувати скільки вершин буде досягнуто. Після відвідування клітинки її можна обнулити, щоб не робити додатковий масив з позначками відвідування.

Код вирішення цієї задачі мовою C++ на платформі LeetCode.

```
int dfs(vector<vector<int>>& grid, int i, int j, int& t)
{   int n = grid.size(), m = grid[0].size();
    if(i >= 0 && i < n && j >= 0 && j < m && grid[i][j] == 1)
    {   grid[i][j]=0;
        t++;
        dfs(grid, i+1, j, t);
        dfs(grid, i-1, j, t);
        dfs(grid, i, j+1, t);
        dfs(grid, i, j-1, t);
    }
    return t;
}
```

```
int maxAreaOfIsland(vector<vector<int>>& grid)
{   int n = grid.size(), m = grid[0].size(), ans = 0;
    for(int i = 0; i < n; i++)
    {   for(int j = 0; j < m; j++)
        {   if(grid[i][j] == 1)
            {   int t = 0;
                ans = max(ans, dfs(grid,i,j,t));
            }
        }
    }
    return ans;
}
```

Запитання для самоконтролю

1. Які методи обходу графів ви знаєте?
2. Яка структура даних використовується для BFS?
3. Яка структура даних використовується для DFS?
4. Який з обходів графа можна реалізувати за допомогою рекурсії?
5. Який з обходів доцільно використовувати для знаходження найкоротшої відстані в незваженому графі?
6. Який з обходів можна використати для топологічного сортування вершин графа?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Пошук_у_ширину
2. https://uk.wikipedia.org/wiki/Пошук_у_глибину
3. https://algoua.com/algorithms/graphs/bfs
4. https://algoua.com/algorithms/graphs/dfs
5. https://www.guru99.com/uk/breadth-first-search-bfs-graph-example.html
6. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph
7. https://www.programiz.com/dsa/graph-bfs
8. https://www.programiz.com/dsa/graph-dfs
9. https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm
10. https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm
11. https://www.simplilearn.com/tutorials/data-structure-tutorial/bfs-algorithm

4. МІНІМАЛЬНЕ КІСТЯКОВЕ ДЕРЕВО ГРАФА

В цьому розділі ми розглянемо поняття кістякового дерева графа та алгоритми знаходження мінімального кістякового дерева.

4.1. ВИЗНАЧЕННЯ КІСТЯКОВОГО ДЕРЕВА

Кістякове дерево (англ. Spanning tree) зв'язаного неорієнтованого графа — ациклічний (тобто, без циклів) зв'язний підграф цього графа, який містить усі його вершини.

Кістякове дерево складається з деякої підмножини ребер графа, таких, що рухаючись цими ребрами можна з будь-якої вершини графа потрапити до будь-якої іншої (рис. 4.1).

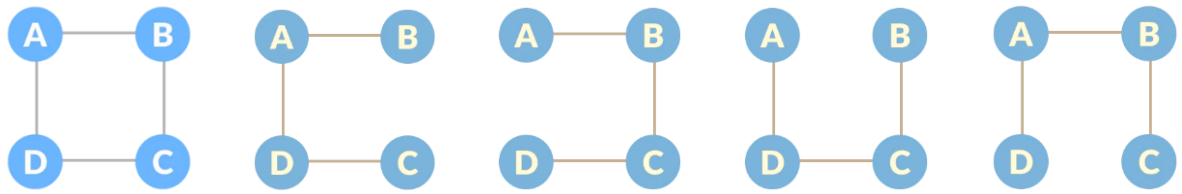


Рис. 4.1. Граф та всі його кістякові дерева

Кістякове дерево також називають *каркасним* деревом, *покриваючим* деревом, *кістяком* або *каркасом* графа.

Будь-яке кістякове дерево у графі з N вершинами містить рівно N-1 ребер. Кількість кістякових дерев у повному графі з N вершинами подається відомою формулою Келі: N^{N-2}.

Кістякове дерево може бути побудовано майже будь-яким алгоритмом обходу графа, наприклад пошуком у глибину або у ширину.

У випадку зваженого графа, різні кістякові дерева будуть мати різну сумарну вагу ребер (рис. 4.2).

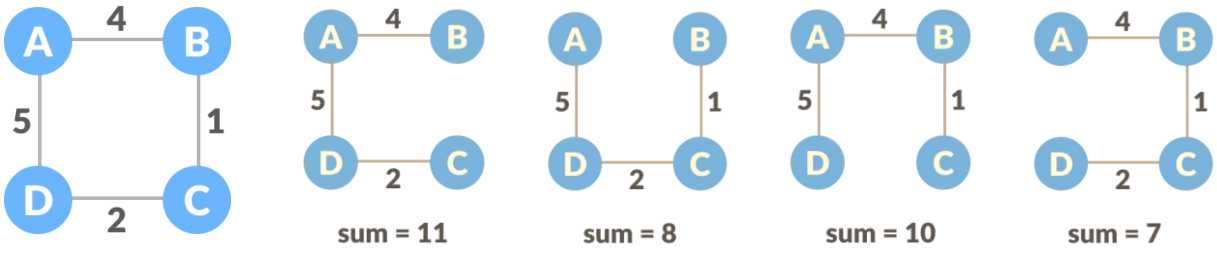


Рис. 4.2. Зважений граф та всі його кістяки

Кістякове дерево з найменшою сумарною вагою ребер називається *мінімальним кістяковим деревом* (англ. Minimum Spanning Tree). Для знаходження мінімального кістякового дерева використовуються наступні алгоритми: Борувки, Прима, Крускала та Чу-Лю (двох китайців). Найчастіше використовуються алгоритми Прима та Крускала, тому розберемо їх детальніше.

4.2. Алгоритм Прима

Алгоритм Прима — жадібний алгоритм побудови мінімального кістякового дерева зваженого зв'язного неорієнтованого графа. Опублікований Робертом Примом у 1957 році.

4.2.1. Формальний опис алгоритму

Нижче подано формальний опис послідовності виконання алгоритму Прима.

1. Ініціалізувати дерево з однією вершиною, довільно вибраною з графа.
2. Збільшити дерево на одне ребро: із ребер, що з'єднують дерево з вершинами, які ще не в дереві, знайти ребро мінімальної ваги та додати його у дерево.
3. Повторювати крок 2, доки всі вершини не будуть у дереві.

4.2.2. Приклад виконання алгоритму

Розглянемо приклад виконання алгоритму на графі (рис. 4.3).

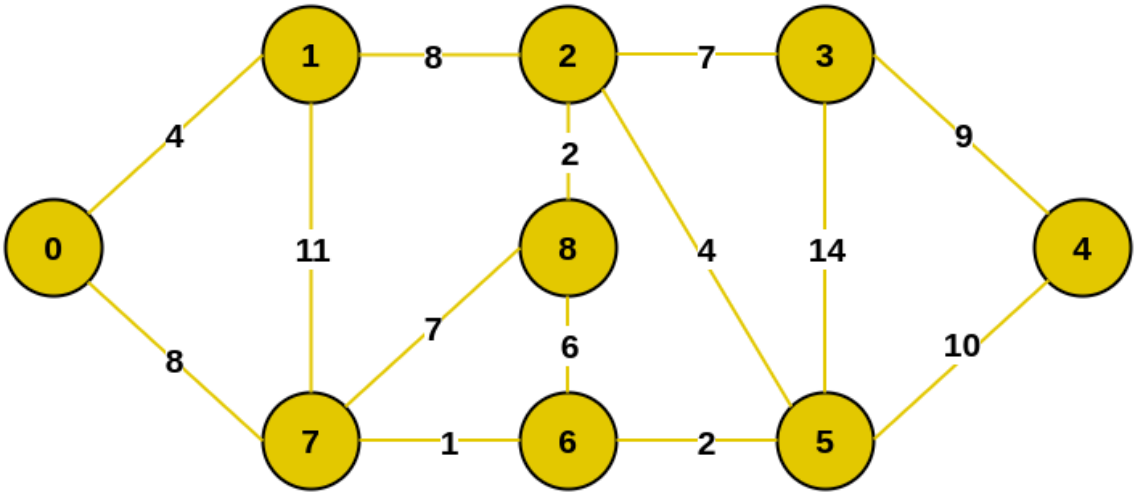


Рис. 4.3. Приклад графа

На *першому* кроці обираємо довільну вершину і додаємо її в кістякове дерево. Нехай це буде вершина з номером `0` (рис. 4.4).

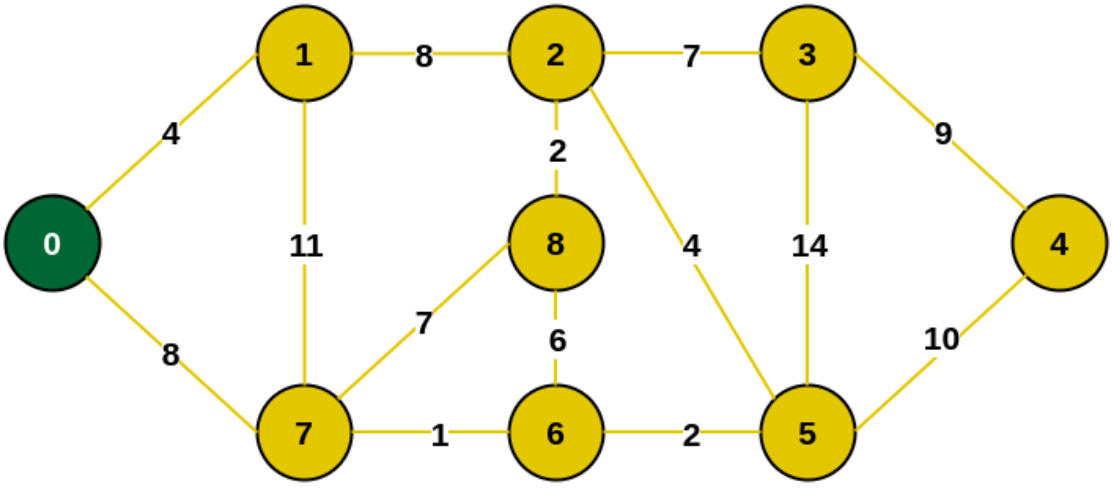


Рис. 4.4. Перша вершина в кістяковому дереві (позначена зеленим)

На *другому* кроці потрібно обрати найменше із ребер, які з'єднують кістякове дерево з іншими вершинами графа, що залишилися. На цьому кроці є два таких ребра: (0,1) з вагою 4 та (0,7) з вагою 8. Оскільки ребро (0,1) коротше, то обираємо його і додаємо вершину з номером 1 до кістякового дерева (рис. 4.5).

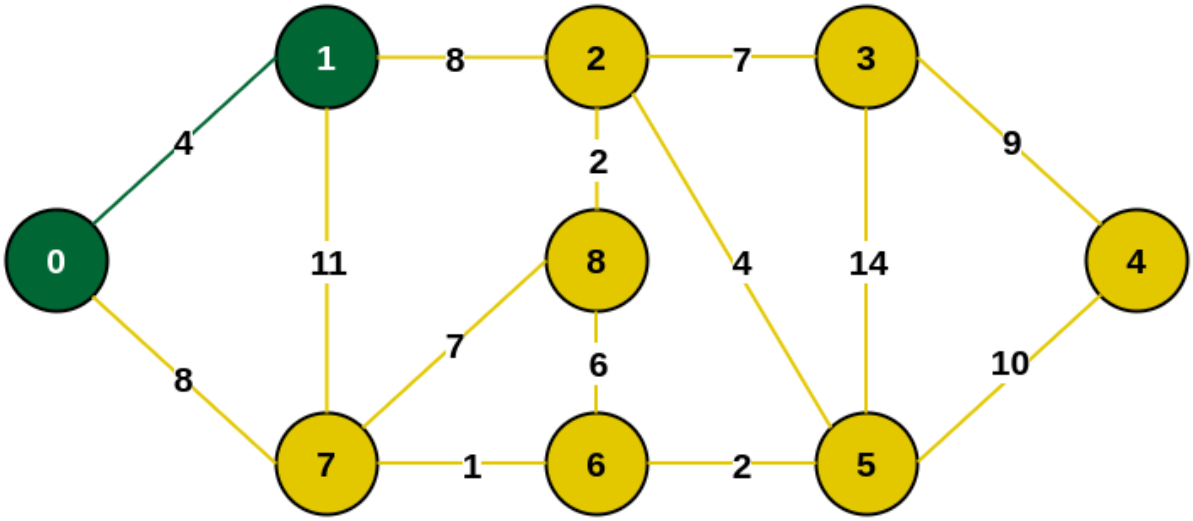


Рис. 4.5. Кістякове дерево графа після додавання другої вершини

На *третьому* кроці ми обираємо найменше ребро серед трьох: (0,7) з вагою 8, (1,2) з вагою 8 та (1,7) з вагою 11. Оскільки два ребра мають однакову найменшу вагу, то обираємо будь-яке з них (рис. 4.6).

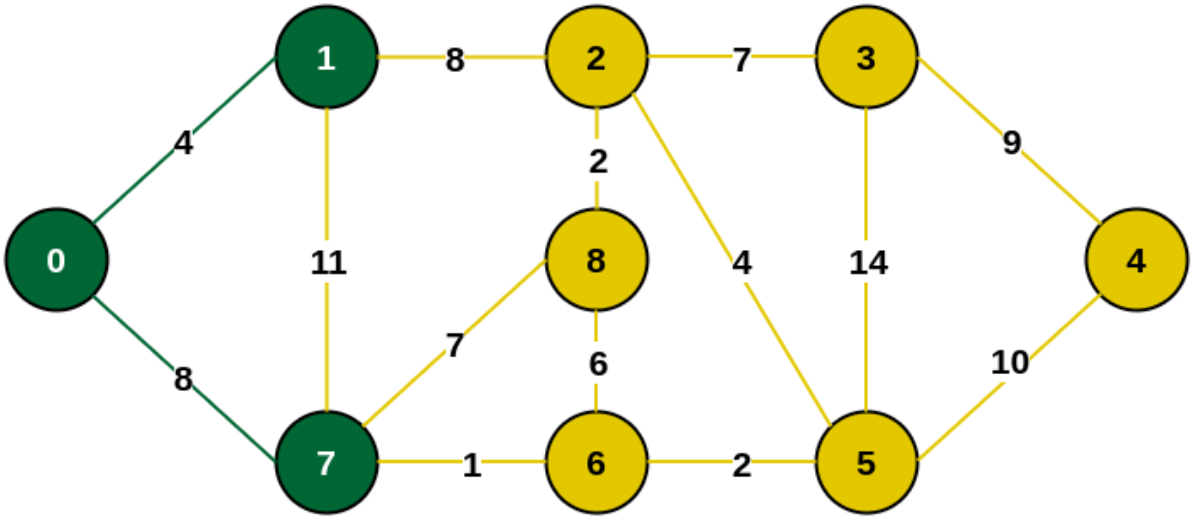


Рис. 4.6. Кістякове дерево графа після додавання третьої вершини

На *четвертому* кроці ми знову маємо три ребра, що з'єднують кістякове дерево з іншими вершинами графа: (1,2) з вагою 8, (7,6) з вагою 1 та (7,8) з вагою 7. Найменше ребро серед них (7,6) з вагою 1 тому додаємо вершину 6 до кістякового дерева (рис. 4.7).

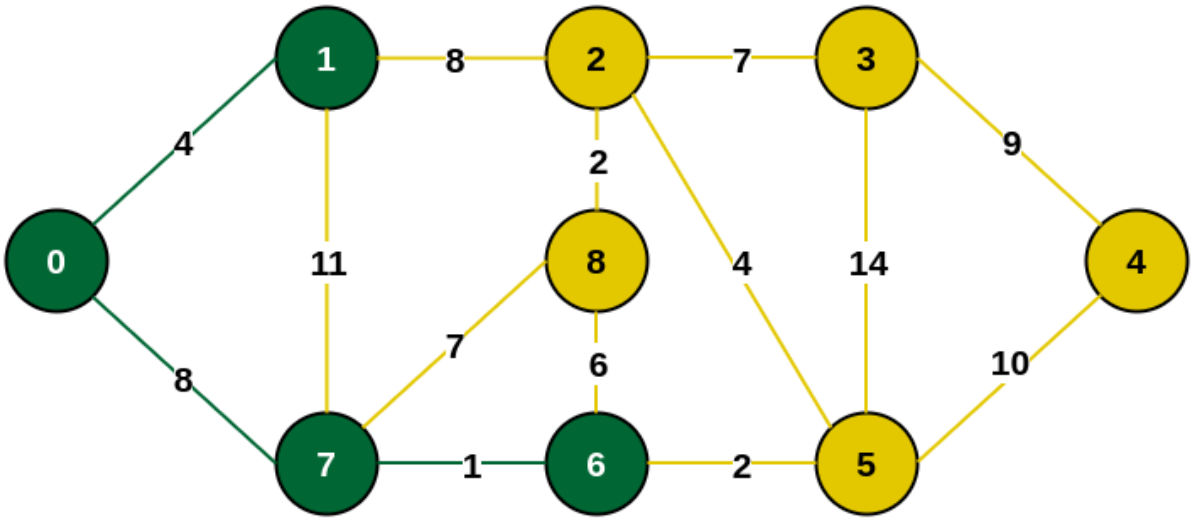


Рис. 4.7. Кістякове дерево графа після додавання четвертої вершини

На *п'ятому* кроці в нас вже є чотири ребра: (1,2) з вагою 8, (7,8) з вагою 7, (6,5) з вагою 2 та (6,8) з вагою 6. Ребро (6,5) має найменшу вагу, а отже додаємо вершину 5 до кістякового дерева (рис. 4.8).

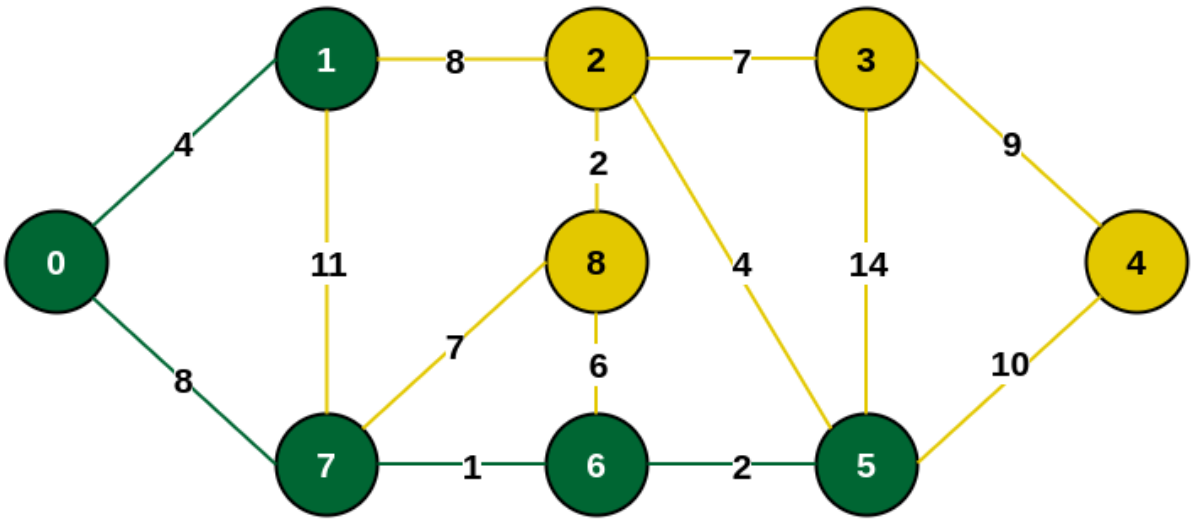


Рис. 4.8. Кістякове дерево графа після додавання п'ятої вершини

На *шостому* кроці в нас вже є шість ребер, які з'єднують кістякове дерево з вершинами графа: (1,2) з вагою 8, (7,8) з вагою 7, (6,8) з вагою 6, (5,2) з вагою 4, (5,3) з вагою 14 та (5,4) з вагою 10. Очевидно, що найменшим є ребро (5,2) з вагою 4, тому додаємо вершину 2 до кістякового дерева (рис. 4.9).

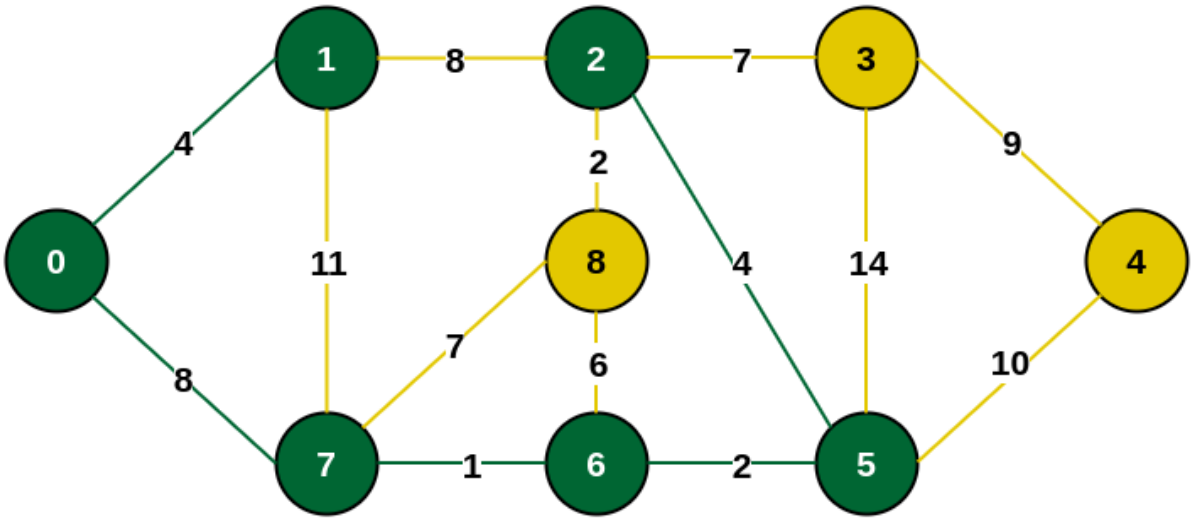


Рис. 4.9. Кістякове дерево графа після додавання шостої вершини

На *сьомому* кроці в нас є шість ребер: (7,8) з вагою 7, (6,8) з вагою 6, (5,3) з вагою 14, (5,4) з вагою 10, (2,3) з вагою 7 та (2,8) з вагою 2. Останнє ребро є найменшим, тому додаємо його до кістяка графа (рис. 4.10).

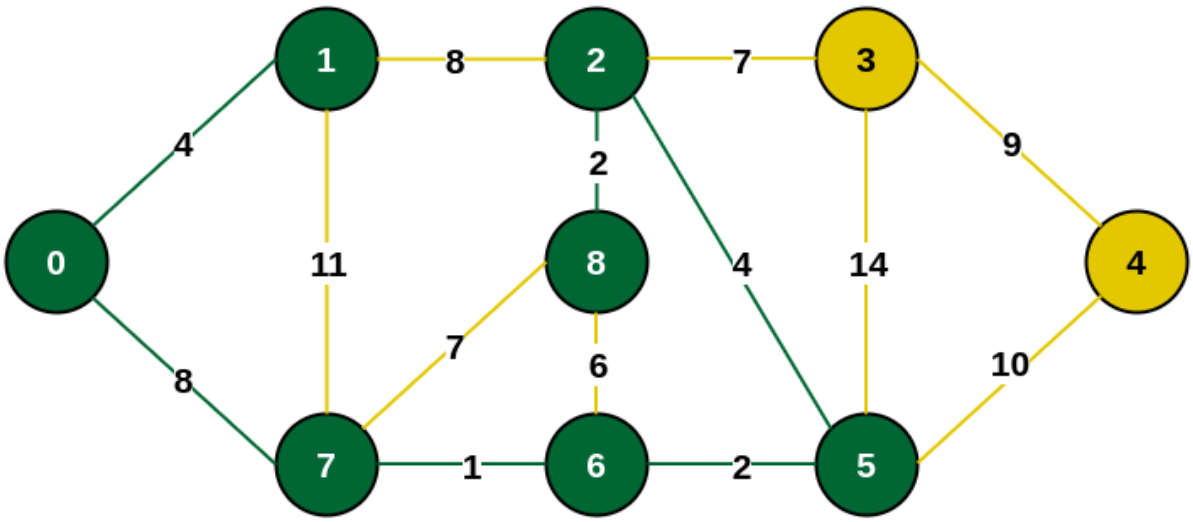


Рис. 4.10. Кістякове дерево графа після додавання сьомої вершини

На *восьмому* кроці алгоритму в нас є лише три ребра, які з'єднують кістякове дерево з іншими вершинами графа: (5,3) з вагою 14, (5,4) з вагою 10 та (2,3) з вагою 7. Останнє з перерахованих ребер є найменшим, а тому ми додаємо його до кістякового дерева графа (рис. 4.11).

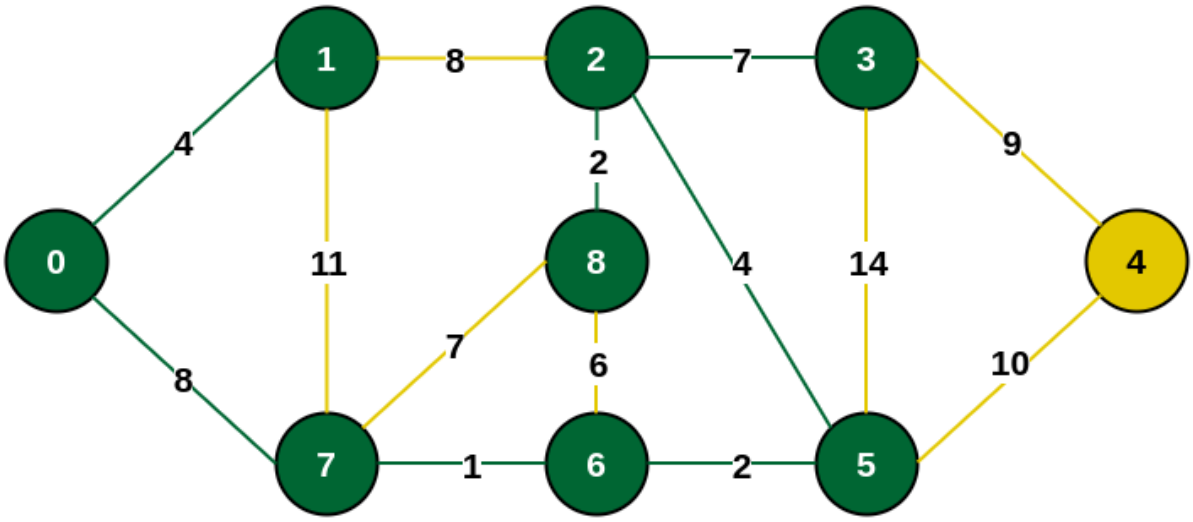


Рис. 4.11. Кістякове дерево графа після додавання восьмої вершини

На *дев'ятому* і останньому кроці алгоритму ми розглядаємо лише два ребра, що залишилися: (3,4) з вагою 9 та (5,4) з вагою 10. Обираємо ребро (3,4) з вагою 9 та додаємо його до кістякового дерева графа (рис. 4.12).

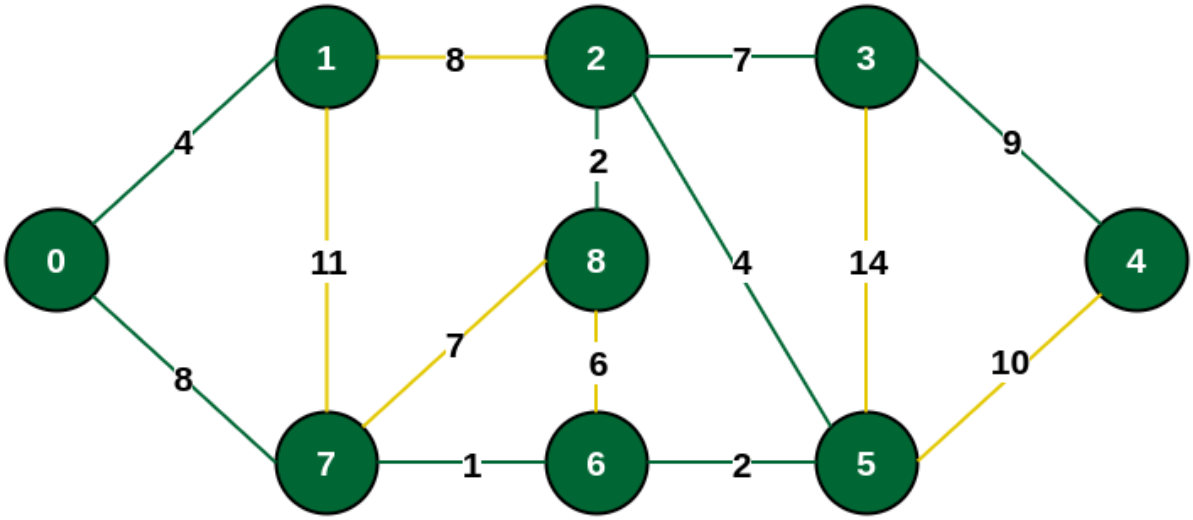


Рис. 4.12. Кістякове дерево графа після додавання дев'ятої вершини

За результатом роботи алгоритму Прима ми отримали кістякове дерево графа, яке є мінімальним. Сума ваг його ребер: (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.

Часова складність алгоритму Прима залежить від структури даних, що використовується в реалізації. Якщо використовувати матрицю суміжності, то час виконання буде $O(V^2)$. З використанням двійковою купи (binary heap) час виконання буде $O(E \log V)$. Використання купи Фібоначчі (Fibonacci heap) час виконання буде $O(E + V \log V)$.

4.2.3. Код реалізації алгоритму

Реалізація алгоритму Прима з використанням матриці суміжності (V — кількість вершин графа, G[V][V] — матриця суміжності). Результат не зберігається в окремому дереві, а відразу виводиться в процесі роботи.

```
int selected[V];
int no_edge = 0;
selected[0] = true;
while (no_edge < V - 1)
{
    int x = 0, y = 0, min = INT_MAX;
    for (int i = 0; i < V; i++)
    {
        if (selected[i])
        {
            for(int j = 0; j < V; j++)
            {
                if (!selected[j] && G[i][j])
                {
                    if (min > G[i][j])
                    {
                        min = G[i][j];
                        x = i;
                        y = j;
                    }
                }
            }
        }
    }
    cout << x << " - " << y << " :  " << G[x][y] << endl;
    selected[y] = true;
    no_edge++;
}
```

Реалізація алгоритму Прима з використанням списку суміжності та черги з пріоритетом. В цій реалізації використовується структура для опису ребра, яка містить індекс вершини до якої приходить ребро та вагу цього ребра.

```
struct Edge
{
    int to;
    int weight;
};
```

Також використовуються додаткові вектори для організації алгоритму.

```
void primMST(vector<vector<Edge>>& graph, int V)
{
    vector<int> parent(V, -1);
    vector<int> key(V, INT_MAX);
    vector<bool> inMST(V, false);
    priority_queue<pair<int, int>> pq;
    key[0] = 0;
    pq.push({0, 0});
    while (!pq.empty())
    {
        int u = pq.top().second;
        pq.pop();
        inMST[u] = true;
        for (Edge& edge : graph[u])
        {
            int v = edge.to;
            int weight = edge.weight;
            if (!inMST[v] && weight < key[v])
            {
                key[v] = weight;
                parent[v] = u;
                pq.push({key[v], v});
            }
        }
    }
}
```

4.3. Алгоритм Крускала

Алгоритм Крускала — алгоритм побудови мінімального кістякового дерева зваженого неорієнтовного графу. Вперше описаний Джозефом Крускалом у 1956 році.

4.3.1. Формальний опис алгоритму

Нижче подано формальний опис послідовності виконання алгоритму Крускала.

1. Створити ліс дерев, кожне з яких спочатку складається з окремої вершини графа.
2. Відсортувати ребра графа за збільшенням ваги.

3. Взяти ребро зі списку і перевірити чи не створює воно цикл в будь-якому дереві лісу. Якщо створює цикл — відкинути, якщо не створює — додати в ліс та об'єднати два дерева, яким воно належить.
4. Повторювати крок 3, доки всі вершини не будуть в одному дереві.

4.3.2. Приклад виконання алгоритму

Розглянемо приклад виконання алгоритму на графі (рис. 4.13).

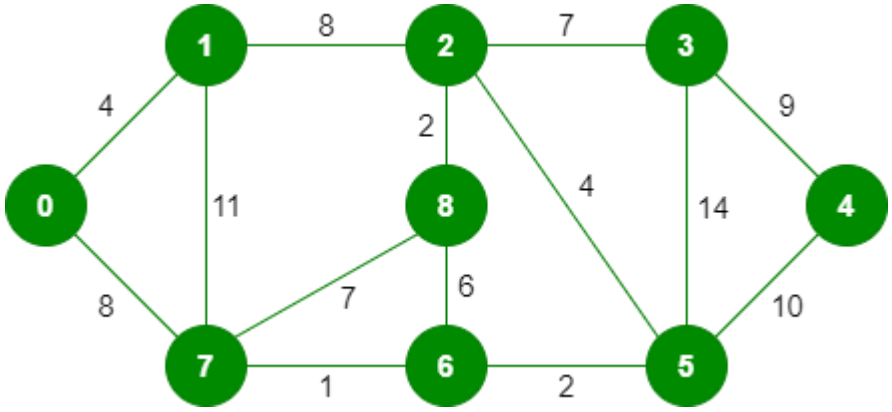


Рис. 4.13. Приклад графа

Спочатку сортуємо всі ребра графа в порядку зростання ваги (рис. 4.14).

Вага	Початок	Кінець
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Рис. 4.14. Таблиця з відсортованими ребрами графа

На *першому* кроці обираємо найменше ребро 7 - 6. Воно не створює циклів, тому додаємо його в кістякове дерево (рис. 4.15).

Рис. 4.15. Кістякове дерево графа після додавання першого ребра

На *другому* кроці беремо наступне найменше ребро 8-2. Воно не створює циклів, тому додаємо його в кістякове дерево (рис. 4.16).

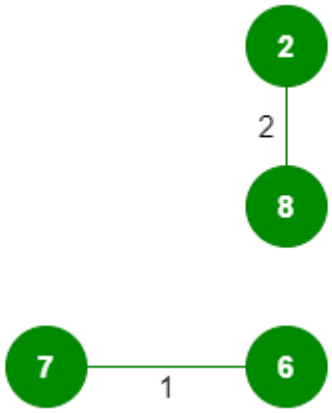


Рис. 4.16. Кістякове дерево графа після додавання другого ребра

На *третьому* кроці ми обираємо наступне найменше ребро 6-5. Воно теж не створює циклів, тому додаємо його в кістякове дерево (рис. 4.17).

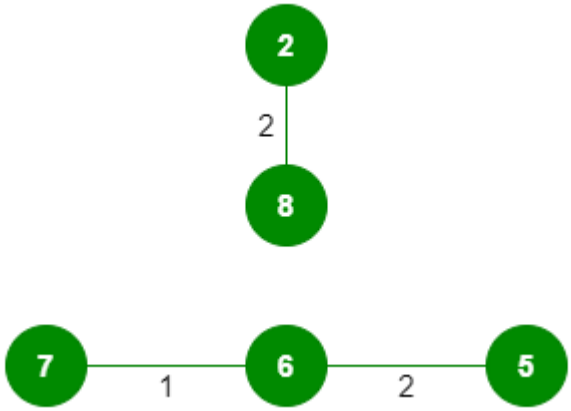


Рис. 4.17. Кістякове дерево графа після додавання третього ребра

На *четвертому* кроці ми знову обираємо найкоротше ребро 0-1 та додаємо його в кістякове дерево (рис. 4.18).

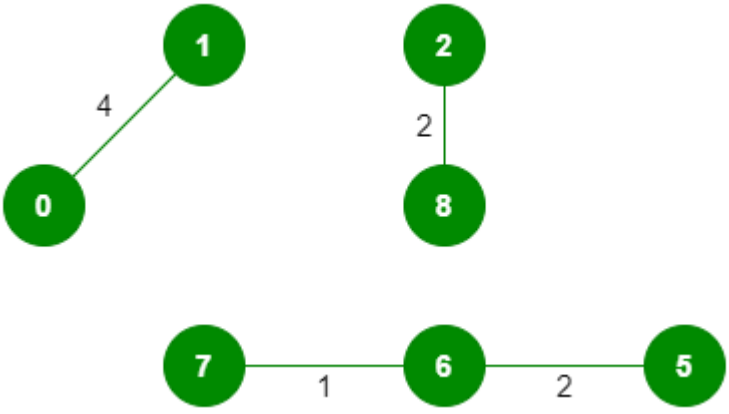


Рис. 4.18. Кістякове дерево графа після додавання четвертого ребра

На *п'ятому* кроці ми обираємо ребро 2-5 та додаємо його в кістякове дерево оскільки воно не створює цикл (рис. 4.19).

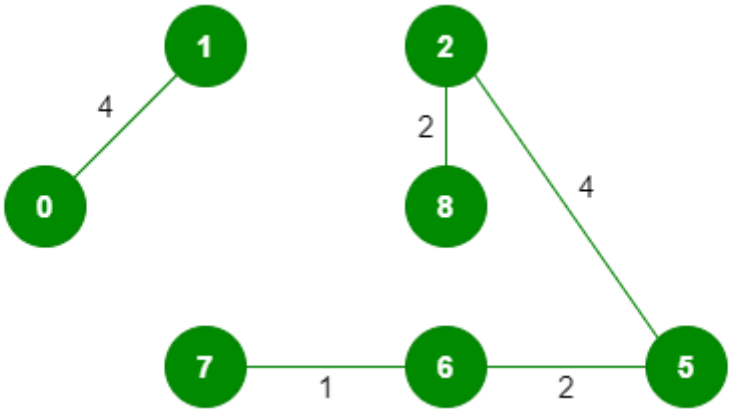


Рис. 4.19. Кістякове дерево графа після додавання п'ятого ребра

На *шостому* кроці ми перевіряємо ребро 6-8, але оскільки воно створює цикл, то його не додаємо в кістяк. Обираємо наступне ребро 2-3 і додаємо його в кістякове дерево (рис. 4.20).

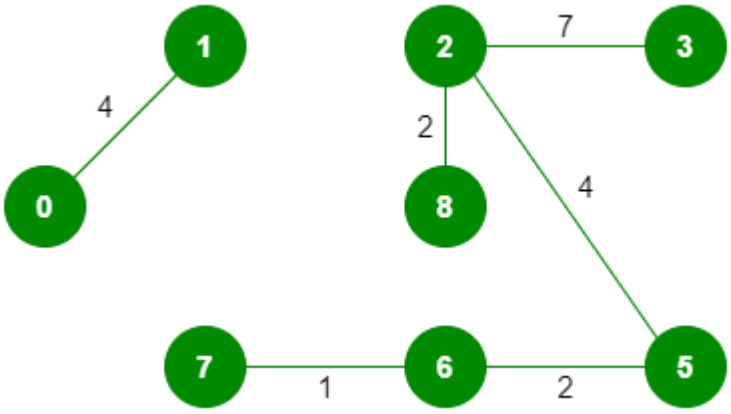


Рис. 4.20. Кістякове дерево графа після додавання шостого ребра

На *сьомому* кроці ми перевіряємо ребро 7-8, але воно створює цикл, а тому пропускаємо його. Наступним ребром розглядаємо ребро 0-7 і додаємо його в кістякове дерево (рис. 4.21).

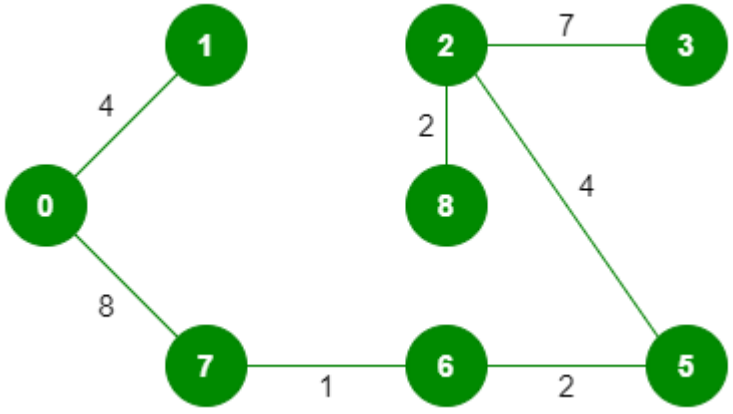


Рис. 4.21. Кістякове дерево графа після додавання сьомого ребра

На *восьмому* кроці ми перевіряємо ребро 1-2, але воно створює цикл, а тому пропускаємо його. Наступне ребро в списку 3-4, воно не створює цикл, а тому додаємо його в кістякове дерево (рис. 4.22).

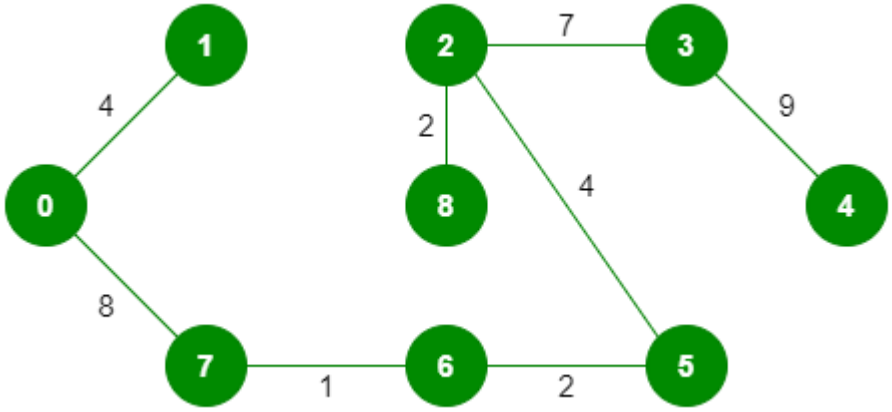


Рис. 4.22. Кістякове дерево графа після додавання восьмого ребра

Оскільки кількість ребер дорівнює кількості вершин мінус один, то кістякове дерево сформоване. За результатом роботи алгоритму Крускала отримане кістякове дерево є мінімальним і відповідає дереву, що отримане алгоритмом Прима.

Обчислювальна складність алгоритму Крускала залежить від методу реалізації. Якщо використовувати просту реалізацію з матрицею суміжності, то час виконання буде $O(V^2)$. З використанням системи неперетинних множин (disjoint-set-union, DSU) час виконання буде $O(E \log E)$.

В реальності, найчастіше використовується алгоритм Крускала з використанням системи неперетинних множин. Власне використання DSU буде розглянуте в наступному розділі.

4.3.3. Код реалізації алгоритму

Тут наведена проста реалізація з використанням матриці суміжності (V — кількість вершин графа, E — кількість ребер).

```
vector<pair<int, pair<int, int>>> g(E);
int cost = 0;
vector<pair<int, int>> res;
sort(g.begin(), g.end());
vector<int> tree_id(V);
for (int i = 0; i < V; i++) tree_id[i] = i;
for (int i = 0; i < E; i++)
{
    int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
    if (tree_id[a] != tree_id[b])
    {
```

```
        cost += l;
        res.push_back(make_pair(a, b));
        int old_id = tree_id[b], new_id = tree_id[a];
        for (int j = 0; j < V; j++)
        {
            if (tree_id[j] == old_id) tree_id[j] = new_id;
        }
    }
}
```

4.4. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач на графах з використанням обох алгоритмів побудови мінімального кістякового дерева.

Ось перелік задач на платформі HackerEarth середнього рівня складності, які можуть бути розв'язані з використанням або алгоритму Прима або алгоритму Крускала.

1. I'll buy you anything
   https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/practice-problems/algorithm/ill-buy-you-anything-58bd4544.
2. Reduce the Array
   https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/practice-problems/algorithm/reduce-the-array-2-2a1e3e02.
3. Hacker Land
   https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/practice-problems/algorithm/hacker-land-a4c9de07.
4. Mr. President
   https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/practice-problems/algorithm/mr-president.
5. 3 types
   https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/practice-problems/algorithm/3-types.

Запитання для самоконтролю

1. Що таке кістякове дерево графа?
2. Які ще терміни використовуються для позначення кістякового дерева?
3. Скільки кістякових дерев можна побудувати на графі з N вершинами?
4. Що таке мінімальне кістякове дерево графа?
5. Які алгоритми побудови мінімального кістякового дерева ви знаєте?

6. Яка найкраща часова складність для алгоритмів побудови мінімального кістякового дерева?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Кістякове_дерево
2. https://uk.wikipedia.org/wiki/Мінімальне_кістякове_дерево
3. https://uk.wikipedia.org/wiki/Алгоритм_Прима
4. https://uk.wikipedia.org/wiki/Алгоритм_Крускала
5. https://uk.wikipedia.org/wiki/Алгоритм_двох_китайців
6. https://en.wikipedia.org/wiki/Minimum_spanning_tree
7. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5
8. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2
9. https://www.simplilearn.com/tutorials/data-structure-tutorial/prims-algorithm
10. https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm
11. https://www.programiz.com/dsa/prim-algorithm
12. https://www.programiz.com/dsa/kruskal-algorithm
13. https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial

5. СИСТЕМА НЕПЕРЕТИННИХ МНОЖИН

Система неперетинних множин (англ. disjoint-set-union або DSU, також використовують назви англ. union–find data structure, англ. merge–find set) — структура даних, яка дозволяє відстежувати множину елементів, розбиту на неперетинні підмножини. В кожній підмножині призначається її представник — елемент цієї підмножини.

5.1. ОСНОВНІ ХАРАКТЕРИСТИКИ DSU

Структура даних надає такі можливості. Спочатку є кілька елементів, кожен з яких знаходиться в окремій (своїй власній) множині. За одну операцію можна об'єднати дві будь-які множини, а також можна запитати, в якій множині зараз знаходиться зазначений елемент. У класичному варіанті вводиться ще одна операція — створення нового елемента, який поміщається в окрему множину — синґлетон.

Таким чином, базовий інтерфейс цієї структури даних складається всього з трьох операцій:

- MakeSet — додання нового елемента; розміщення його в нову множину, що складається з одного нього;
- Union — об'єднання двох зазначених множин;
- Find — повернення значення, в якій множині знаходиться зазначений елемент. Насправді повертається один з елементів множини званий представником (англ. representative) або лідером (leader). Цей представник вибирається в кожній множині самою структурою даних (і може змінюватися з плином часу). Наприклад, якщо виклик для якихось двох елементів повернув одне і те ж значення, то це означає, що ці елементи знаходяться в одній і тій же множині, а в іншому випадку — в різних множинах.

Ця структура даних дозволяє робити кожну з цих операцій майже за O(1) в середньому.

5.2. РЕАЛІЗАЦІЯ З КВАДРАТИЧНИМ ЧАСОМ ВИКОНАННЯ

Нехай у нас є множина з 10 елементів S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} (рис. 5.1).

Рис. 5.1. Множина з 10 елементів

Ми можемо використати масив Arr для керування зв'язністю елементів. Розмір цього масиву дорівнює кількості елементів у множині (рис. 5.2).

Рис. 5.2. Масив для збереження зв'язності елементів

Кожен індекс масиву відповідає порядковому номеру елемента множини. Значення масиву містить номер множини, до якої належить елемент. Наприклад, якщо об'єкти A і B зв'язані, тоді Arr[A] = Arr[B].

Ініціалізація масиву Arr.

```
void Init(int Arr[ ], int N)
{
    for(int i = 0; i < N; i++)
    {
        Arr[i] = i;
    }
}
```

Виконаємо операцію об'єднання двох підмножин Union(2, 1) (рис. 5.3).

Рис. 5.3. Множина після об'єднання двох підмножин 1 і 2

Після виконання операції об'єднання двох підмножин 1 і 2 масив буде мати наступний вигляд (рис. 5.4).

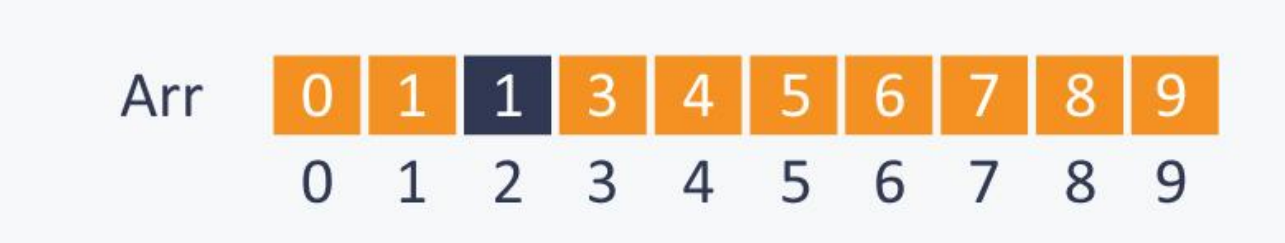


Рис. 5.4. Масив після об'єднання двох підмножин 1 і 2

Тепер виконаємо цілу серію об'єднань: Union(4, 3), Union(8, 4) та Union(9, 3) (рис. 5.5).

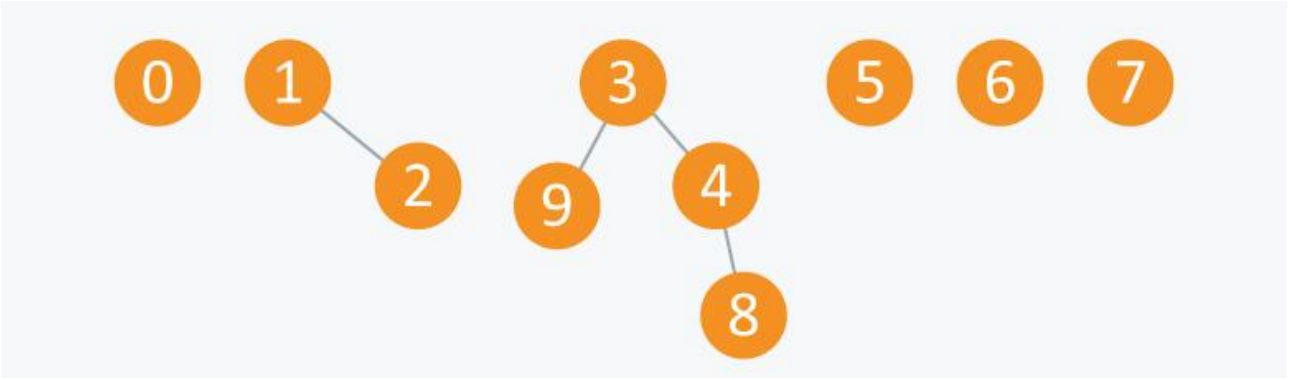


Рис. 5.5. Множина після серії об'єднань підмножин

Після такої серії об'єднань, масив буде мати наступний вигляд (рис. 5.6).

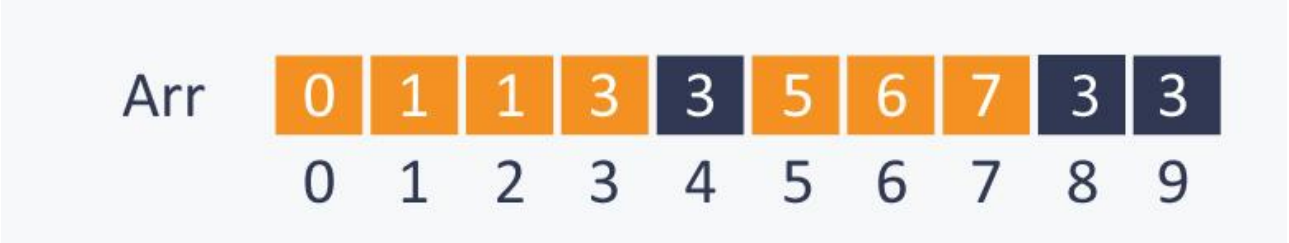


Рис. 5.6. Масив після серії об'єднань підмножин

Тепер виконаємо ще одне об'єднання: `Union`(5, 6) (рис. 5.7).

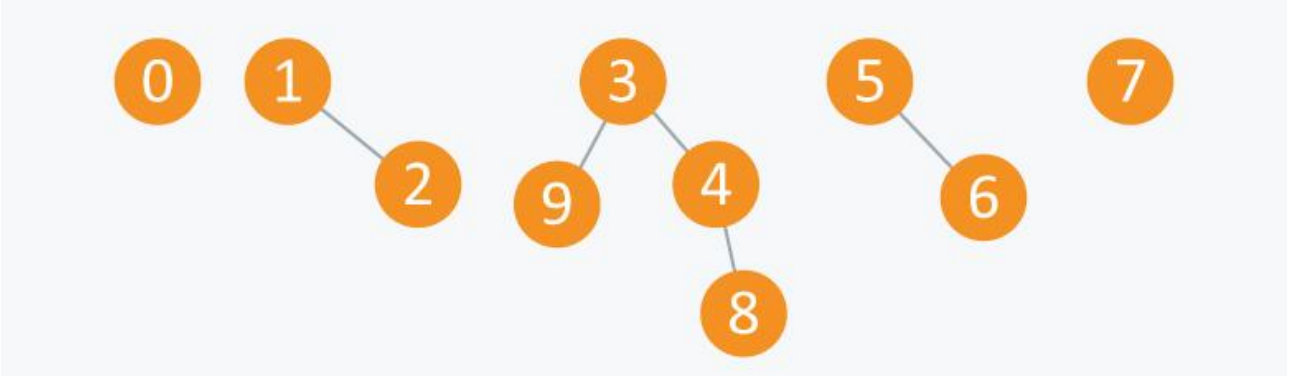


Рис. 5.7. Множина після об'єднання двох підмножин 5 і 6

Після виконання операції об'єднання двох підмножин 5 і 6 масив буде мати наступний вигляд (рис. 5.8).

Рис. 5.8. Масив після об'єднання двох підмножин 5 і 6

Після виконання всіх об'єднань в нас є множина, що складається з п'яти підмножин: перша — {3, 4, 8, 9}; друга — {1, 2}; третя — {5, 6}; четверта — {0} і п'ята — {7}. Всі ці підмножини можна розглядати як частини одного графа. Такі частини графа, в яких вузли зв'язані між собою називаються компонентами

зв'язності графа. Таким чином використання системи неперетинних множин важливі для задач з графами, таких як знаходження компонент зв'язності, об'єднання вузлів та інших.

Проста реалізація функції Union.

```
void Union(int Arr[ ], int N, int A, int B)
{
    int tmp = Arr[A];
    for(int i = 0; i < N; i++)
    {
        if(Arr[i] == tmp) Arr[i] = Arr[B];
    }
}
```

Цикл в цій функції проходить по всіх N елементах множини для об'єднання двох елементів. Тобто виконання цієї функції для N елементів займе $O(N^2)$ часу, що зовсім не ефективно.

В простій реалізації функція Find просто перевіряє чи два елементи знаходяться в одній множині. Виконаємо перевірку двох елементів на предмет знаходження їх в одній множині: Find(0, 7). Оскільки ці два елементи знаходяться в різних підмножинах, то результат буде false.

Тепер перевіримо наступні два елементи: Find(8, 9). Ці два елементи не з'єднані безпосередньо, але вони знаходяться в одній підмножині, а тому існує маршрут з 8 до 9. Отже результатом перевірки буде значення true.

Як ми бачили вище, операція Union(A, B) заміняє значення в масиві за індексами A і B на значення підмножини, яке відповідає індексу поточного представника множини. Операція Find(A, B) перевіряє чи значення за індексами A і B однакові чи ні. Якщо однакові, то обидва елементи належать одній множині, якщо різні — то різним.

Проста реалізація функції Find.

```
bool Find(int Arr[ ], int A, int B)
{
    if(Arr[A] == Arr[B]) return true;
    else                 return false;
}
```

Ця функція працює константний час O(1). Проте, зважаючи на час виконання функції Union загальний час роботи буде $O(N^2)$.

5.3. Реалізація з лінійним часом виконання

Головна ідея покращення це заміна масиву `Arr` з індексами підмножин на масив батьківських елементів (`parent`). Тобто за кожним індексом в масиві знаходиться не індекс підмножини, а індекс батьківського елемента для поточного елемента.

Ми можемо розглядати кореневий (`root`) елемент в кожній підмножині, який єдиний має в якості батьківського самого себе `Arr[R] = R`. Цей елемент є представником підмножини.

Розглянемо більш короткий приклад для демонстрації принципу лінійної реалізації. Нехай у нас є множина з 6 елементів S = {0, 1, 2, 3, 4, 5} якій відповідає початковий масив батьківських елементів (рис. 5.9).

Рис. 5.9. Масив для збереження батьків елементів

Виконаємо операцію об'єднання двох підмножин `Union(1, 0)`. Ця операція зробить елемент `0` батьківським для елемента `1`, а тому запис в масиві зміниться (рис. 5.10).

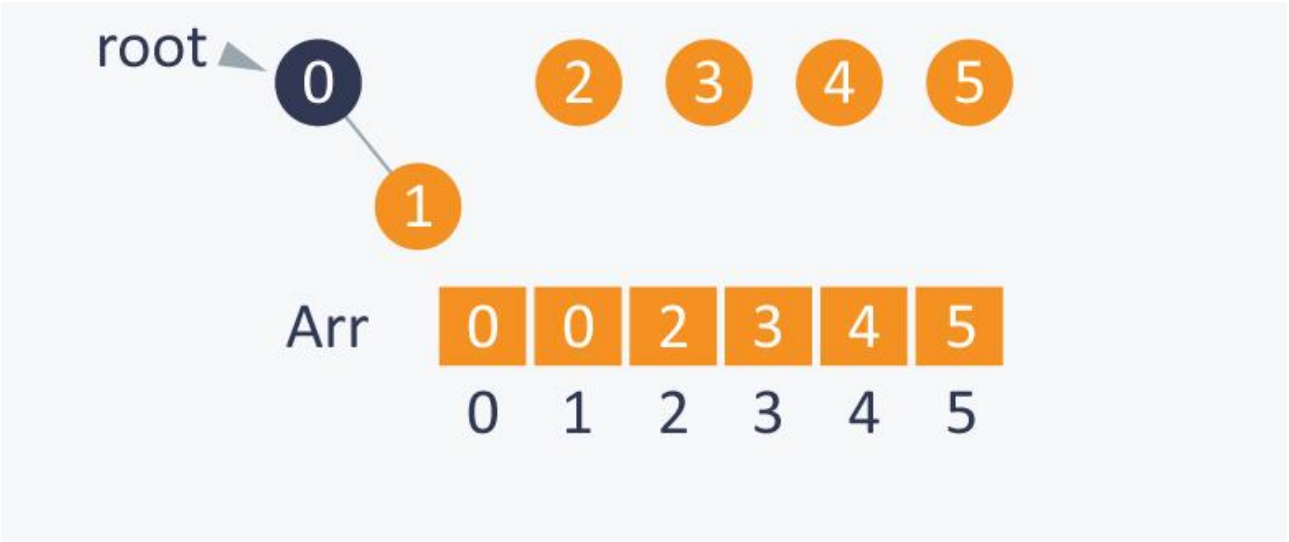


Рис. 5.10. Множина і масив батьків після об'єднання елементів `0` та `1`

Тепер виконаємо операцію `Union(0, 2)`, яка з'єднає елементи `0` і `2`. Значення в масиві батьків за індексом `0` зміниться на двійку (рис. 5.11). Ця операція зробить двійку кореневим елементом множини елементів {2, 0, 1}.

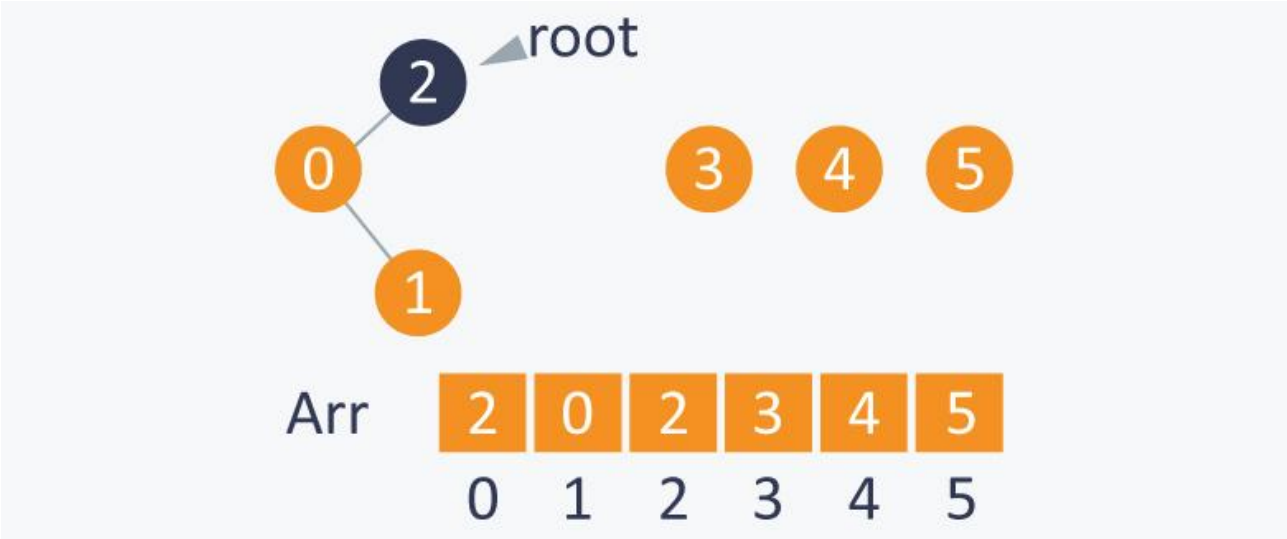


Рис. 5.11. Множина і масив батьків після об'єднання елементів 0 та 2

Таким саме чином операція Union(3, 4) об'єднає елементи 3 і 4 шляхом призначення елемента 4 батьком для елемента 3 (рис. 5.12).

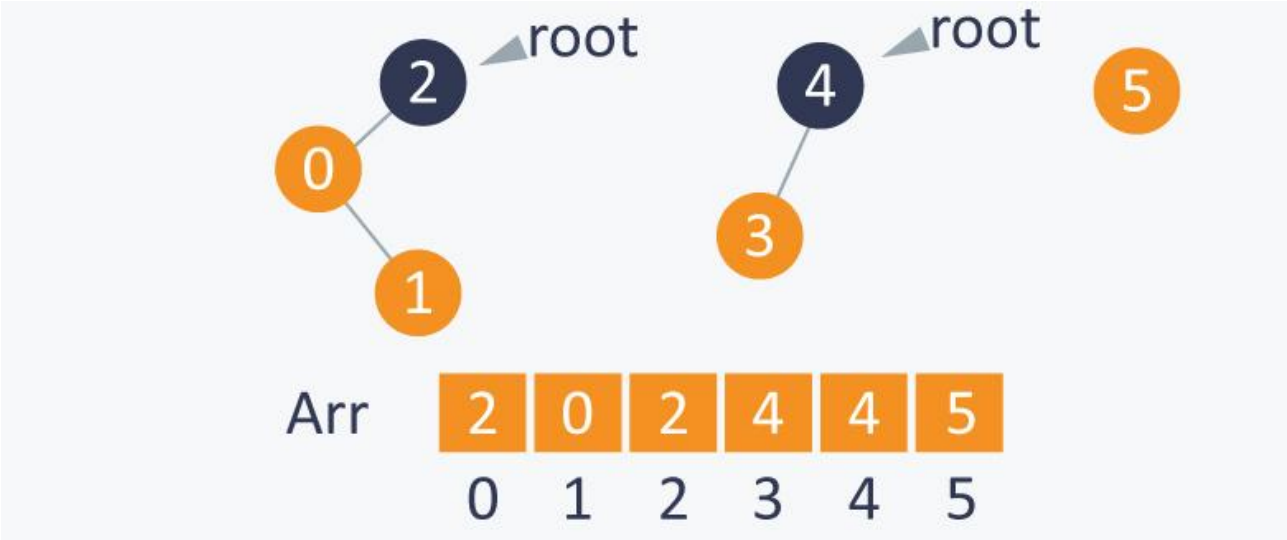


Рис. 5.12. Множина і масив батьків після об'єднання елементів 3 та 4

Тепер, якщо ми зробимо операцію Union(1, 4), то отримаємо об'єднання двох підмножин, що містять елементи 1 і 4 відповідно. Оскільки кореневий елемент підмножини, де знаходиться елемент 1, є елемент 2, то така операція замінить значення кореневого елемента для двійки на четвірку (рис. 5.13).

Тепер елемент 4 є кореневим елементом для підмножини, що складається з елементів {0, 1, 2, 3, 4}.

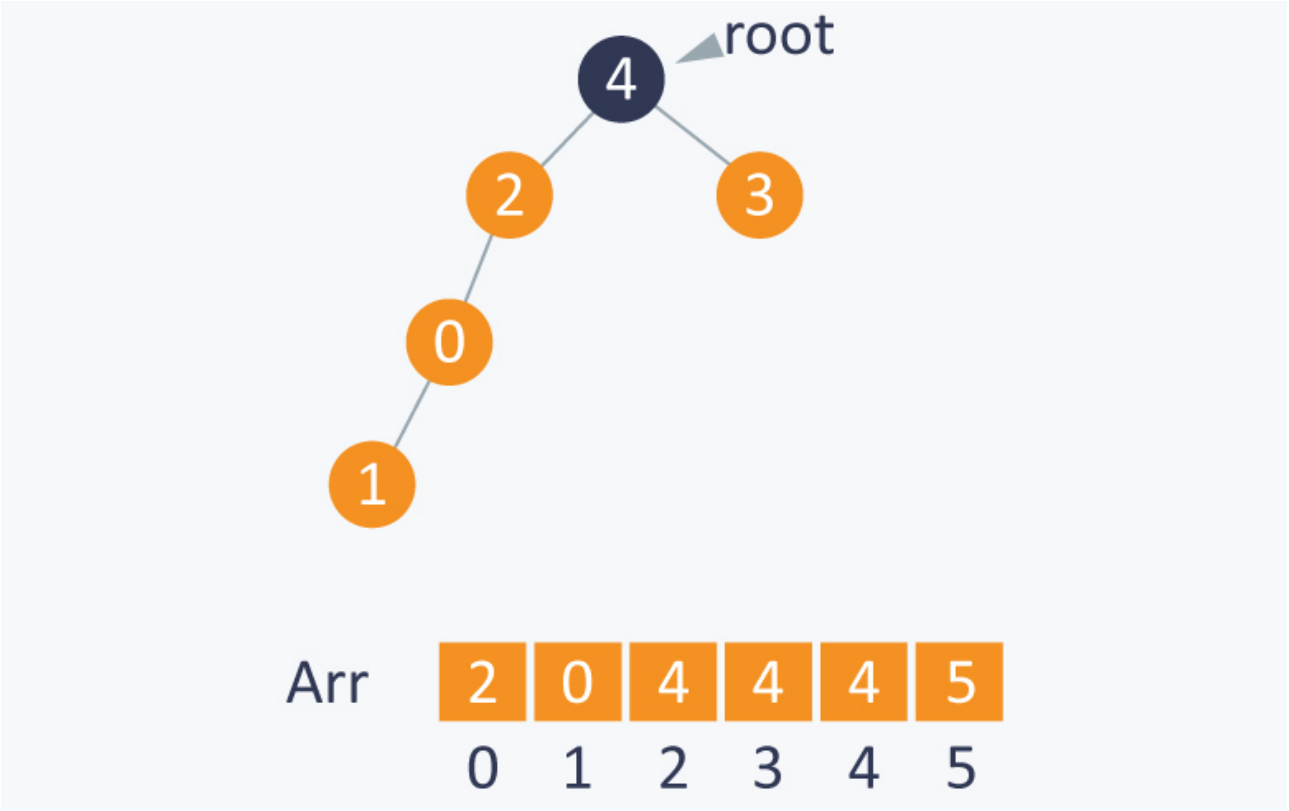


Рис. 5.13. Множина і масив батьків після об'єднання елементів 1 та 4

Яким же чином нам знайти кореневий елемент множини? Цілком очевидно, що необхідно переходити за батьківськими елементами доти, доки не знайдено елемент який є сам собі батьківським. В нашому прикладі це елемент 4. Пошук можна здійснювати в циклі або рекурсивно. В такій реалізації нам необхідно, щоб функція пошуку `Find` повертала індекс кореневого елемента підмножини (лідера).

Підсумовуючи розглянуте, ми можемо сказати наступне:

- щоб створити новий елемент (операція `make_set(v)`), ми просто створюємо дерево з коренем у вершині, зазначаючи, що її предок — це вона сама;

```
void make_set (int v) { parent[v] = v; }
```

- щоб об'єднати дві множини (операція `union_set(a, b)`), ми спочатку знайдемо лідерів першої і другої множини. Якщо лідери збіглися, то нічого не робимо — це означає, що множини і так вже були об'єднані. В іншому випадку можна просто вказати, що предок першої вершини дорівнює другій (або навпаки) — тим самим приєднавши одне дерево до іншого;

```
void union_sets (int a, int b)
{
    a = find_set (a);
    b = find_set (b);
    if (a != b) parent[b] = a;
}
```

- реалізація операції пошуку лідера (find_set(v)) проста: ми піднімаємося по предках від вершини, поки не дійдемо до кореня. Цю операцію зручніше реалізувати рекурсивно (особливо це буде зручно пізніше, у зв'язку з подальшими оптимізаціями).

```
int find_set (int v)
{
    if (v == parent[v]) return v;
    return find_set (parent[v]);
}
```

Утім, така реалізація системи неперетинних множин дуже неефективна. Легко побудувати приклад, коли після кількох об'єднань множин вийде ситуація, що множина — це дерево, звиродніле в довгий ланцюжок. У результаті кожен виклик буде працювати на такому тесті за час порядку глибини дерева, тобто за лінійний час відносно кількості елементів — O(N).

5.4. Реалізація з «майже константним часом» виконання

Щоб прискорити реалізацію з лінійним часом роботи необхідно застосувати деякі покращення алгоритму. Розглянемо їх один за одним.

5.4.1. Стиснення шляху

Цей метод призначений для прискорення роботи операції пошуку лідера find_set(v). Він полягає в тому, що коли після виклику ми знайдемо шуканого лідера множини, то запам'ятаємо, що у вершині v і всіх пройдених по шляху вершин — саме цей лідер. Найпростіше це зробити, перенаправивши їх parent[] на цю вершину.

Таким чином, у масиву предків parent сенс дещо змінюється: тепер це стислий масив предків, тобто для кожної вершини там може зберігатися не безпосередній предок, а предок предка, предок предка предка, і т. д.

З іншого боку, зрозуміло, що не можна зробити, щоб ці покажчики parent завжди вказували на лідера: інакше під час виконання операції довелося б оновлювати лідерів у елементів.

Таким чином, до масиву слід підходити саме як до масиву предків, можливо, частково стиснутого.

Нова реалізація операції має такий вигляд:

```
int find_set (int v)
{
    if (v == parent[v]) return v;
    return parent[v] = find_set (parent[v]);
}
```

Така проста реалізація робить все, що задумувалося: спочатку шляхом рекурсивних викликів знаходиться лідер множини, а потім, в процесі розкрутки стека, цей лідер присвоюється `parent` посиланнями для всіх пройдених елементів.

Реалізувати цю операцію можна і не рекурсивно, але тоді доведеться здійснювати два проходи по дереву: перший знайде шуканого лідера, другий — проставить його всім вершинам шляху. Втім, на практиці нерекурсивна реалізація не дає істотного виграшу.

5.4.2. Об'єднання за рангом

Розглянемо інший підхід, який сам по собі здатен прискорити час роботи алгоритму, а в поєднанні з методом стиснення шляхів і зовсім здатен досягти практично константного часу роботи на один запит в середньому.

Цей метод полягає в невеликій зміні роботи функції `union_set`: якщо в наївній реалізації те, яке дерево буде приєднано до якого, визначається випадково, то тепер ми будемо це робити на основі рангів.

Є два варіанти рангового підходу: в одному варіанті рангом дерева називається кількість вершин в ньому, в іншому — глибина дерева (точніше, верхня межа на глибину дерева, оскільки за одночасного застосування методу стиснення шляхів реальна глибина дерева може зменшуватися).

В обох варіантах суть підходу одна й та ж: під час виконання `union_set` будемо приєднувати дерево з меншим рангом до дерева з більшим рангом (рис. 5.14).

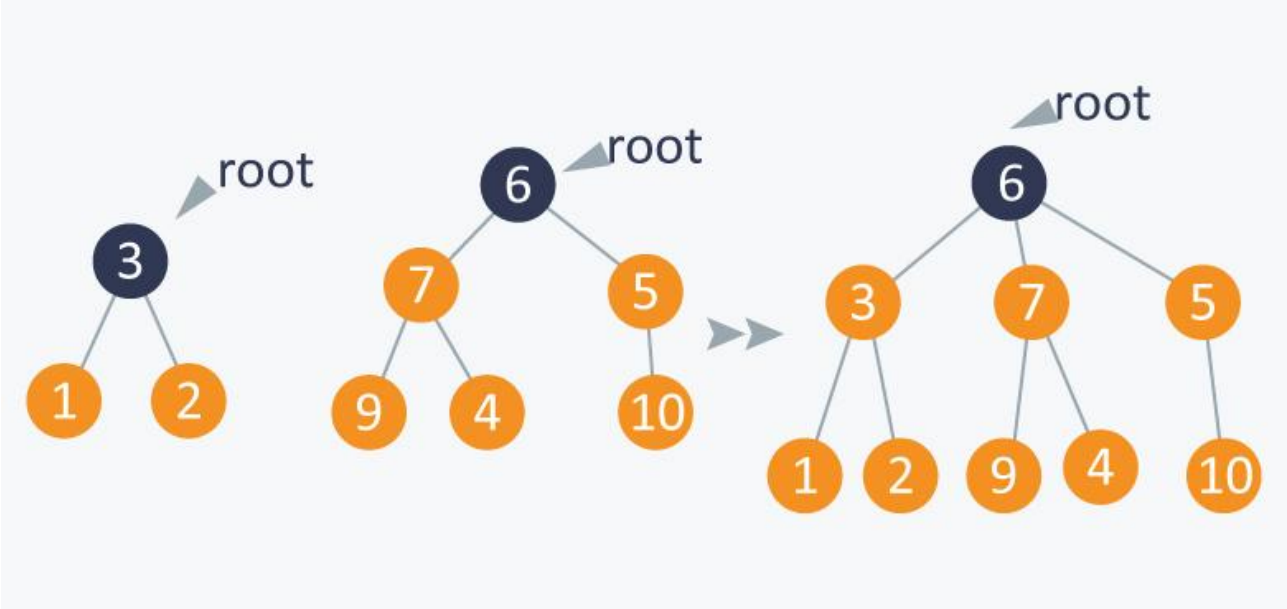


Рис. 5.14. Приклад об'єднання дерев за рангом

Реалізація рангової системи на основі розміру дерев подана нижче.

```
void make_set (int v)
{   parent[v] = v;
    size[v] = 1;}
```

```
void union_sets (int a, int b)
{
    a = find_set (a);
    b = find_set (b);
    if (a != b)
    {
        if (size[a] < size[b]) swap (a, b);
        parent[b] = a;
        size[a] += size[b];
}
```

Реалізація на основі глибини дерев дещо відрізняється, але в цілому схожа.

```
void make_set (int v)
{
    parent[v] = v;
    rank[v] = 0;
}
void union_sets (int a, int b)
{
    a = find_set (a);
    b = find_set (b);
    if (a != b)
    {
        if (rank[a] < rank[b]) swap (a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) ++rank[a];
    }
}
```

Обидва варіанти рангового підходу є еквівалентними з точки зору асимптотики, тому на практиці можна застосовувати будь-який з них.

5.4.3. Приклад виконання

Розглянемо приклад множини до якої застосуємо стиснення шляху та об'єднання за рангом. Візьмемо в якості початкового прикладу множину, масив для якої був зображений на рис. 5.9.

Виконаємо операцію Union(0, 1). Оскільки обидві підмножини 0 і 1 містять всього по одному елементу, то ми можемо просто приєднати їх один до одної в будь-якому порядку. Ми приєднаємо елемент 1 до елемента 0 і збільшимо розмір підмножини з лідером 0 (рис. 5.15).

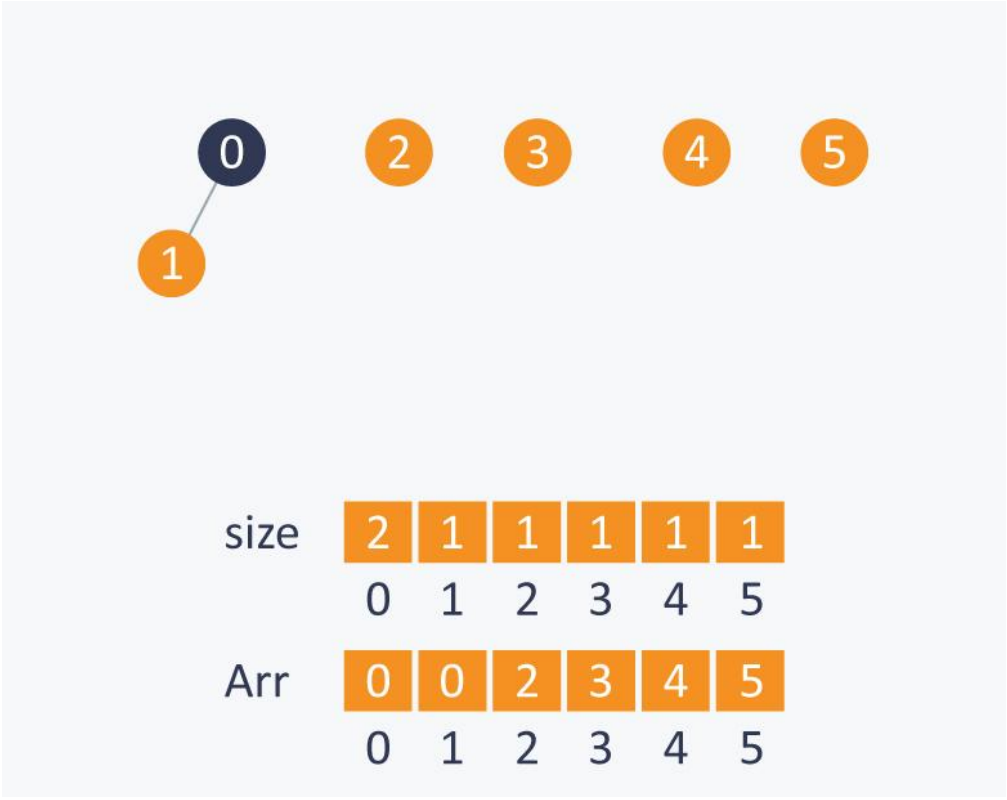


Рис. 5.15. Множина і масиви розмірів та батьків після об'єднання `0` і `1`

Тепер виконаємо операцію `Union(1, 2)`. Нам потрібно об'єднати кореневий вузол підмножини в яку входить елемент `1` з кореневим вузлом підмножини в яку входить елемент `2`. Оскільки підмножина з елементом `2` менша (містить лише `1` елемент), то ми повинні приєднати цю підмножину до підмножини з лідером `0` (рис. 5.16).

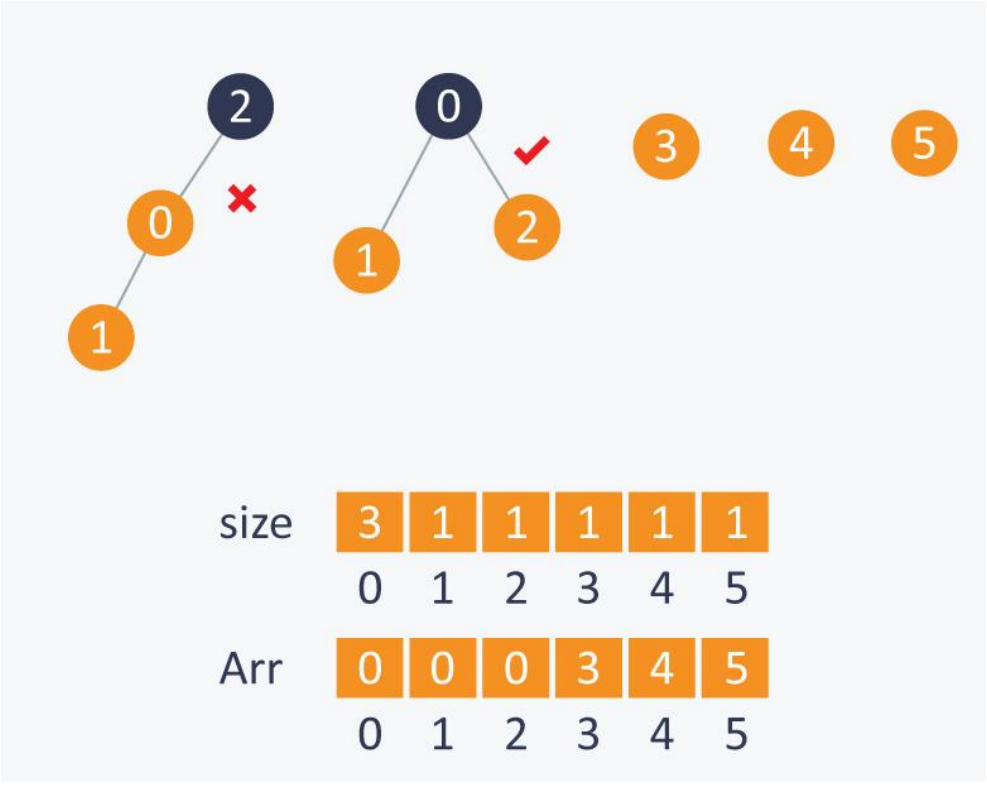


Рис. 5.16. Множина і масиви розмірів та батьків після об'єднання `1` і `2`

Аналогічним чином операція `Union(3, 2)` приєднає елемент 3 до підмножини з лідером `0`, оскільки вона більша за розміром (рис. 5.17).

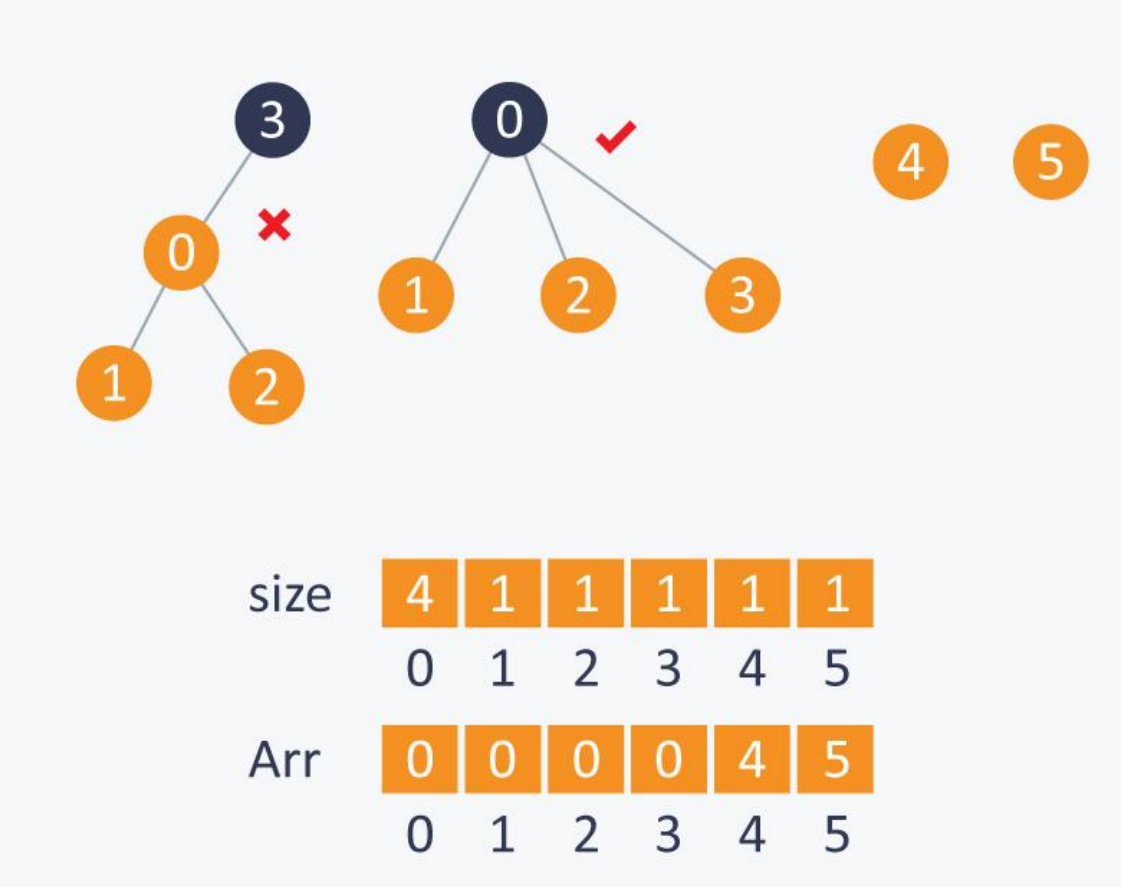


Рис. 5.17. Множина і масиви розмірів та батьків після об'єднання 3 і 2

Балансування дерева дозволить зменшити обчислювальну складність з лінійної до логарифмічної. А додавання стиснення шляху дає результат O(A(n)), де A(n) — обернена функція Акермана, яка зростає дуже повільно, настільки повільно, що для всіх розумних обмежень вона не перевершує 4 (для n <= 10^{600}). Саме тому про асимптотику роботи системи неперетинних множин доречно говорити «майже константний час роботи».

5.5. Застосування DSU

Системи неперетинних множин застосовуються для розв'язання задач на графах або таких, що можуть бути зведені до графового подання. Найбільш розповсюджені задачі, що розв'язуються за допомогою DSU наступні.

1. Визначення зв'язних вузлів у графі або дереві.
2. Пошук компонент зв'язності.
3. Визначення наявності циклу в графі після додавання нового ребра.
4. Для знаходження мінімального кістякового дерева за алгоритмом Крускала.

5.6. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач на графах з використанням системи неперетинних множин.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням системи неперетинних множин.

Середній рівень складності

1. Number of Provinces https://leetcode.com/problems/number-of-provinces.
2. Number of Islands https://leetcode.com/problems/number-of-islands.
3. Number of Enclaves https://leetcode.com/problems/number-of-enclaves.
4. Number of Closed Islands https://leetcode.com/problems/number-of-closed-islands.
5. Surrounded Regions https://leetcode.com/problems/surrounded-regions.

Високий рівень складності

1. Redundant Connection https://leetcode.com/problems/redundant-connection-ii.
2. Making A Large Island https://leetcode.com/problems/making-a-large-island.
3. Minimize Malware Spread https://leetcode.com/problems/minimize-malware-spread.
4. Minimize Malware Spread II https://leetcode.com/problems/minimize-malware-spread-ii.
5. Find All People With Secret https://leetcode.com/problems/find-all-people-with-secret.

Запитання для самоконтролю

1. Що таке система неперетинних множин?
2. Як називається новостворена множина з одного елемента?
3. Які основні операції застосовуються до DSU?
4. Які методи стиснення використовуються для DSU?
5. В яких задачах застосовують DSU?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Система_неперетинних_множин
2. https://en.wikipedia.org/wiki/Disjoint-set_data_structure
3. https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm
4. https://www.geeksforgeeks.org/disjoint-set-data-structures
5. https://www.hackerearth.com/practice/notes/disjoint-set-union-union-find
6. https://medium.com/@rishu__2701/getting-started-with-disjoint-set-data-structure-0a971a68f731

6. МАКСИМАЛЬНИЙ ПОТІК

В теорії оптимізації та теорії графів, задача про максимальний потік (maximum flow) полягає у знаходженні такого потоку за транспортною мережею, щоб сума потоків з джерела, або, що означає те ж саме, сума потоків до стоку була максимальна.

Задача про максимальний потік є окремим випадком більш складних задач, таких, як, наприклад, задача про циркуляцію.

Вперше задача про максимальний потік була розв'язана в 1948 році, а сформульована в загальному вигляді Джорджем Данцигом в 1951 році.

Задачу про максимальний потік можна розв'язати за допомогою лінійного програмування, але час розв'язання буде експоненціальний, якщо застосовувати симплекс-метод. Ми розглянемо спеціалізовані алгоритми для вирішення цієї задачі на графах.

Формальний опис задачі. Задана мережа G з джерелом (s) та стоком (t). Знайти максимальний потік від джерела до стоку. Мережа це граф G=(V, E) де кожне ребро це напрямлений потік з позитивною вагою (ємністю).

6.1. АЛГОРИТМ ФОРДА-ФАЛКЕРСОНА

Алгоритм був опублікований американськими математиками Фордом та Фалкерсоном в 1956 році та названий на їх честь. Основою алгоритму Форда-Фалкерсона є концепція залишкової мережі (residual network) та доповнювальний шлях (augmenting path) .

Алгоритм працює ітеративно: спочатку максимальному потоку присвоюється значення 0, яке збільшується на кожній ітерації за допомогою пошуку шляху збільшення. Цей процес повторюється до тих пір, поки вже неможливо буде відшукати шлях збільшення.

6.1.1. ЗАЛИШКОВА МЕРЕЖА

Залишкова мережа — граф, який має вершини такі ж самі, як і початковий граф та одне або два ребра замість кожного ребра в початковому графі. Вона показує залишкові потенційні потоки в мережі. Для кожного ребра розраховується залишковий потік таким чином:

залишковий потік = ємність ребра – потік через ребро

Тобто, для створення залишкової мережі потрібно взяти початковий граф і оновити ваги ребер з врахуванням потоків через них. Також ми додаємо зворотні ребра щоб позначити вагу потоку, що циркулює в початковому графі.

Для кращого розуміння потоку розглянемо приклад (рис. 6.1).

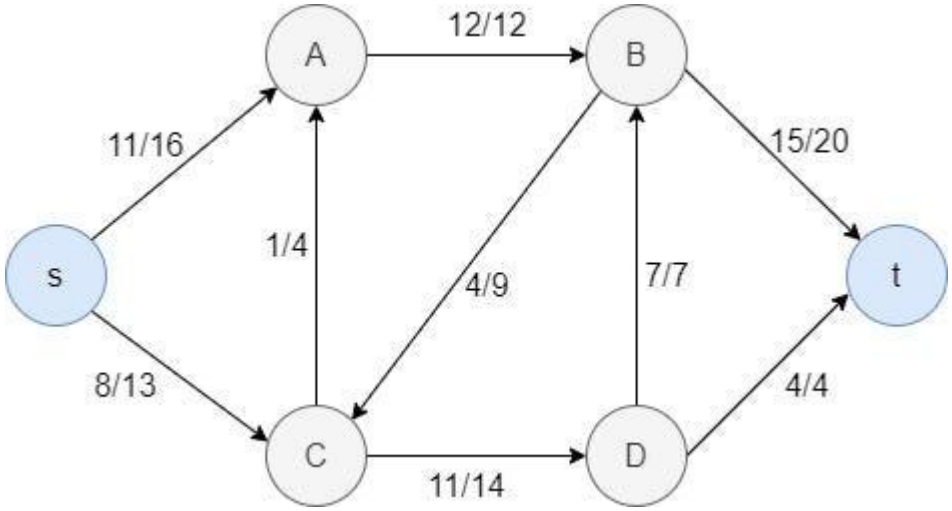


Рис. 6.1. Оригінальна мережа

Для кожного ребра позначені два числа: перше число це потік, а друге — ємність ребра. Наприклад, розглянемо ребро C->D. Тут 11 це потік, а 14 — ємність ребра (максимально можливий потік через це ребро). Тобто залишковий потік для цього ребра буде 14-11=3. Таким чином, в залишковій мережі це ребро буде мати вагу 3, а зворотне ребро — 11.

Якщо повторити ці дії для кожного ребра, то ми отримаємо таку залишкову мережу (рис. 6.2).

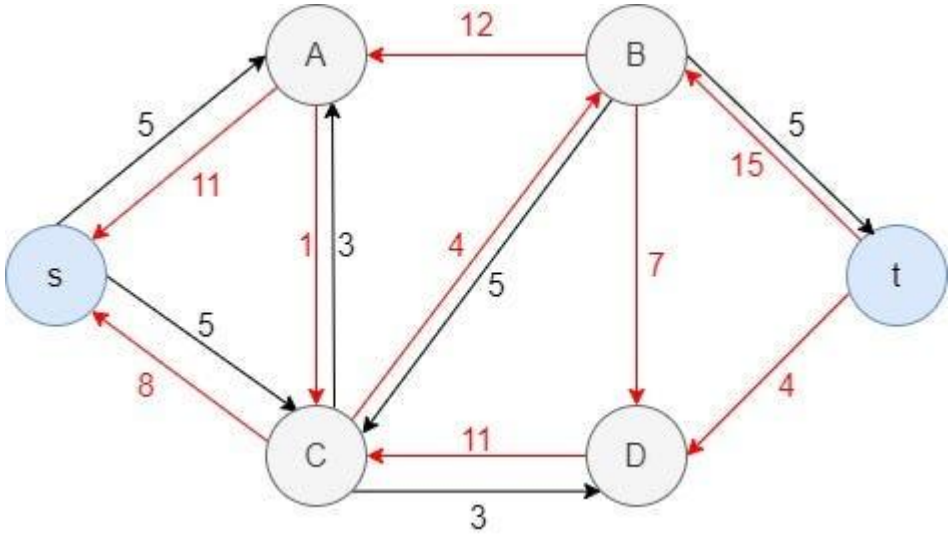


Рис. 6.2. Залишкова мережа (зворотні ребра червоні)

6.1.2. Доповнювальний шлях

Для заданої мережі G=(V, E), доповнювальний шлях це простий маршрут від джерела до стоку у відповідній залишковій мережі (рис. 6.3.).

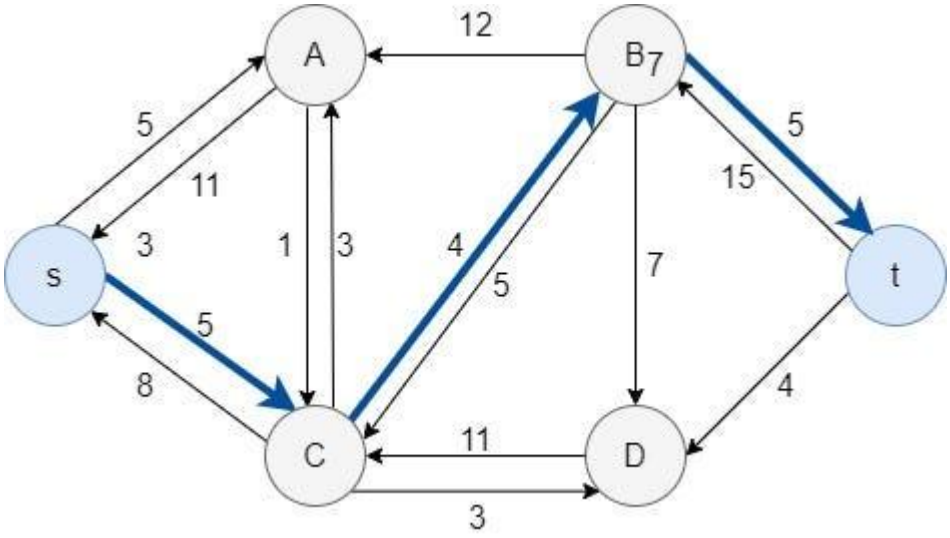


Рис. 6.3. Доповнювальний шлях

6.1.3. Приклад виконання алгоритму

Розглянемо покроковий приклад виконання алгоритму Форда-Фалкерсона. На кожному кроці будемо будувати оригінальну мережу та залишкову мережу.

Крок 1. Потоки на всіх ребрах виставляються нульовими (рис. 6.4).

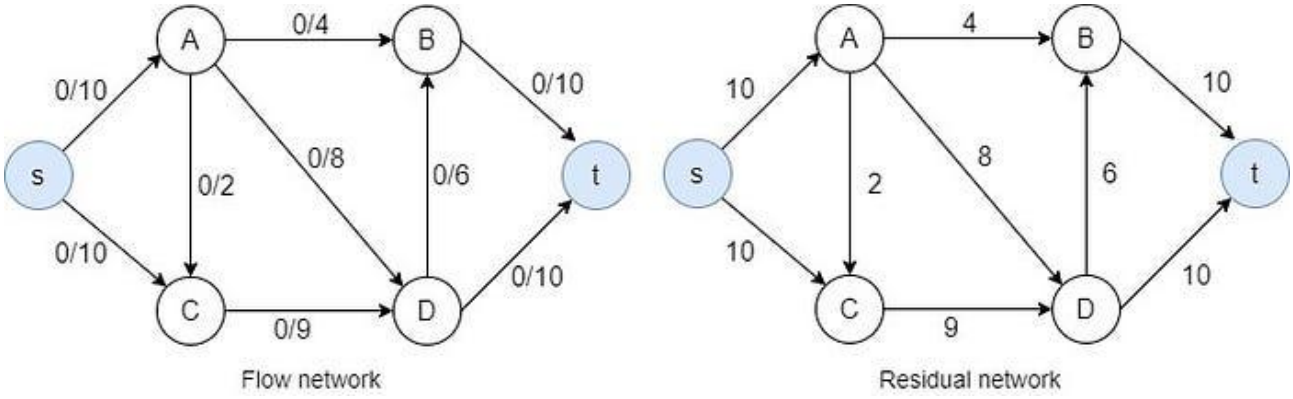


Рис. 6.4. Оригінальна та залишкова мережі після кроку 1

Крок 2. Знаходимо доповнювальний шлях в залишковій мережі. Ми можемо взяти будь-який доступний шлях, наприклад оберемо такий: `s->A->D->t`. Тепер потрібно визначити максимальний потік на цьому шляху. Для цього потрібно знайти ребро з мінімальною вагою — це і буде максимально можливий потік цього шляху. Тепер необхідно оновити значення ваг ребер в доповнювальному шляху. Після цього ми отримаємо наступні мережі (рис. 6.5).

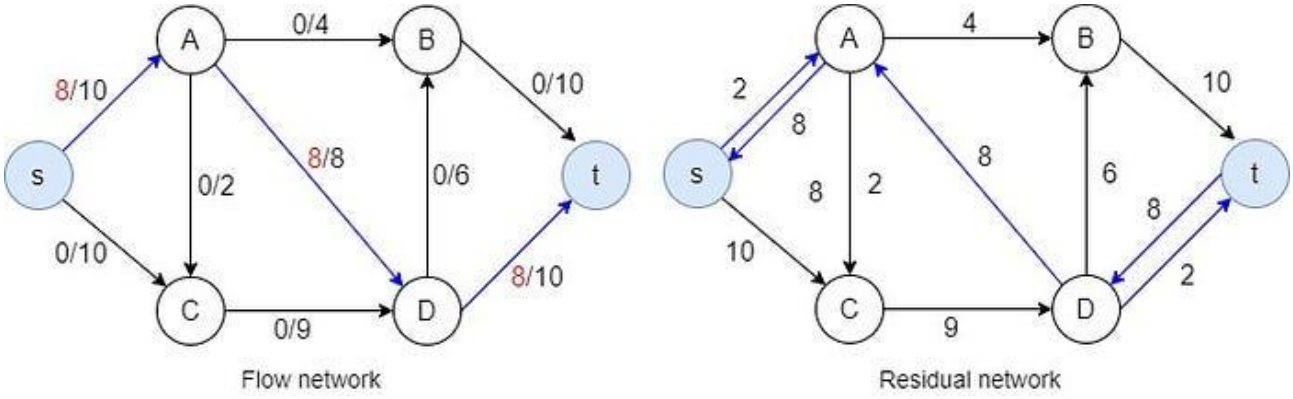


Рис. 6.5. Оригінальна та залишкова мережі після кроку 2

Крок 3. Знову знаходимо доповнювальний шлях в залишковій мережі. На цей раз оберемо шлях `s->C->D->t`. Найменше ребро на цьому маршруті має вагу 2. Отже оновлюємо всі ребра цього шляху з врахуванням цього значення (рис. 6.6).

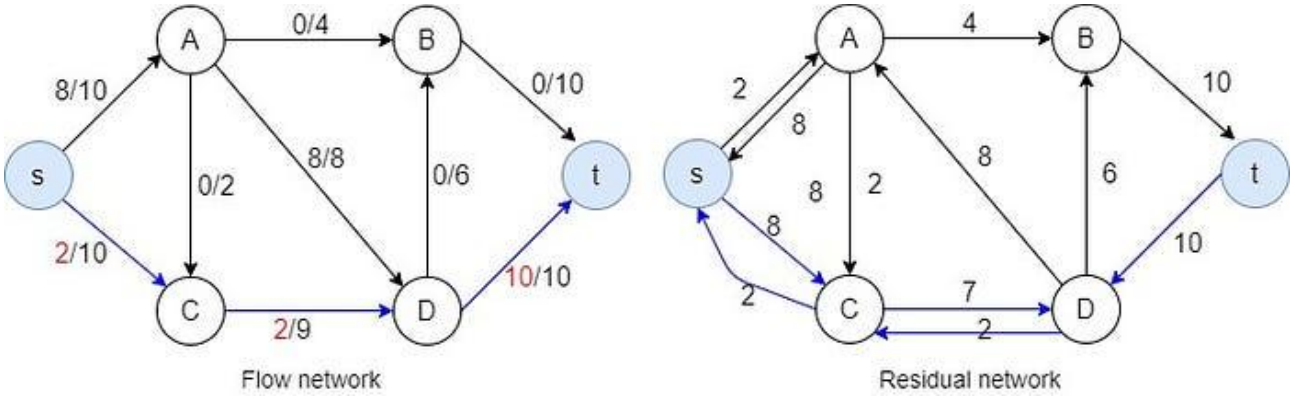


Рис. 6.6. Оригінальна та залишкова мережі після кроку 3

Крок 4. Знову будуємо доповнювальний шлях залишковій мережі. Цього разу це буде шлях `s->A->B->t`. Цей маршрут також має найменше ребро з вагою 2, як і попередній (рис. 6.7).

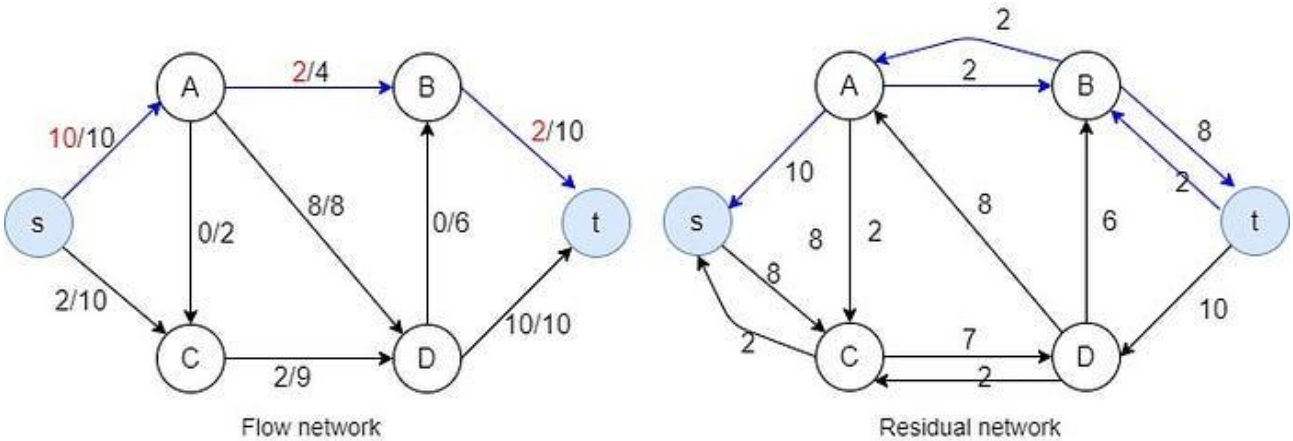


Рис. 6.7. Оригінальна та залишкова мережі після кроку 4

Крок 5. Доповнювальний шлях обираємо `s->C->D->B->t`. Найменше ребро на цьому маршруті має вагу 6, тому оновлюємо всі ребра шляху з врахуванням цього числа (рис. 6.8).

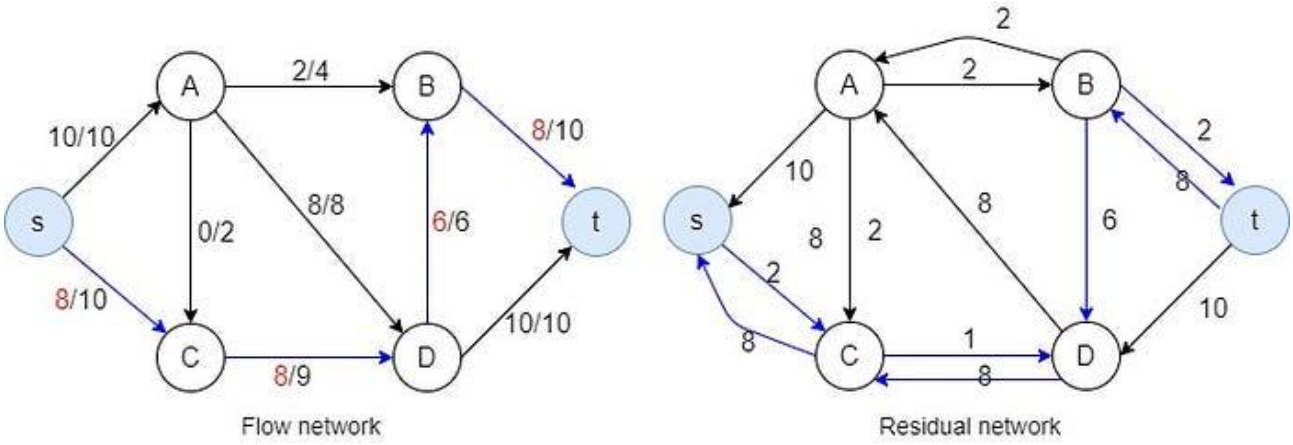


Рис. 6.8. Оригінальна та залишкова мережі після кроку 5

Крок 6. Наступний доповнювальний шлях це `s->C->D->A->B->t` з найменшим ребром вагою 1. Після оновлення ребер шляху ми отримаємо наступні мережі (рис. 6.9).

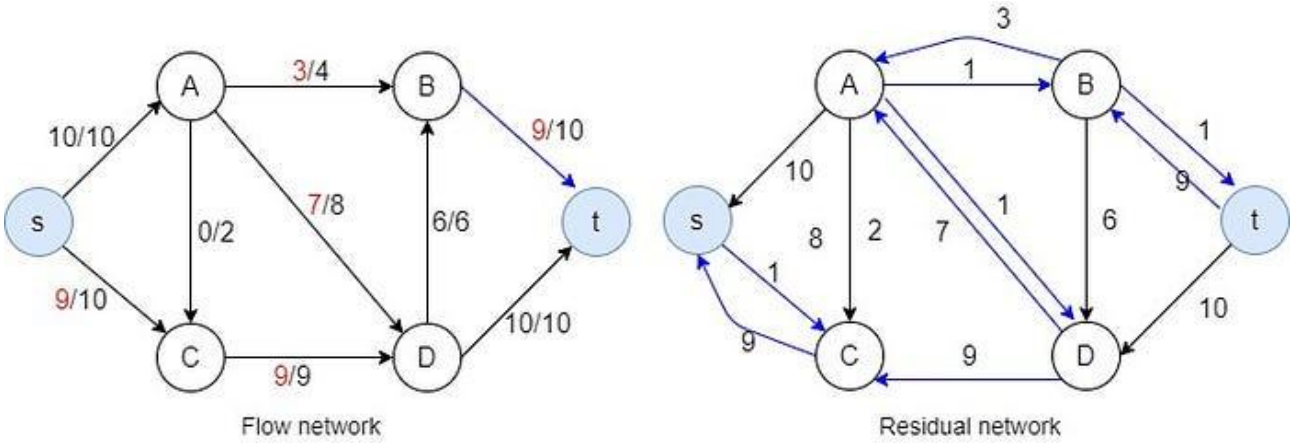


Рис. 6.9. Оригінальна та залишкова мережі після кроку 6

Тепер ми не можемо побудувати жодного маршруту на залишковій мережі, а отже більше немає можливості збільшити потік від джерела до стоку. Це означає, що алгоритм Форда-Фалкерсона закінчив свою роботу і ми отримали максимальний потік.

Значення максимального потоку дорівнює сумі потоків з джерела. В нашому прикладі це `10+9=19`.

6.1.4. Часова складність

Алгоритм закінчує роботу, коли вже неможливо побудувати жодного шляху збільшення. Але можлива ситуація, коли алгоритм працюватиме вічно і не зможе закінчитися та знайти навіть приблизно максимальний потік. Така ситуація можлива у випадку використання не цілих чисел у якості значень потоку на ребрах. В разі використання цілих чисел, часова складність алгоритму буде $O(Ef)$, де E — кількість ребер, а f — максимальний потік. Це тому, що кожен шлях збільшення може бути знайдений за час $O(E)$ та збільшить потік на ціле число в межах від 1 до f.

6.2. Алгоритм Едмондса-Карпа

Алгоритм розв'язує задачу знаходження максимального потоку в транспортній мережі. Алгоритм був незалежно відкритий американськими вченими Едмондсом та Карпом в 1972 році та названий на їх честь. Насправді алгоритм це окремий випадок методу Форда-Фалкерсона і працює за час $O(VE^2)$.

Відмінність від алгоритму Форда-Фалкерсона полягає в тому, що на кожному кроці вибирають найкоротший доповнювальний шлях від джерела до стоку в залишковій мережі. Для пошуку найкоротшого шляху використовують алгоритм BFS і вважають, що кожне ребро має одиничну вагу.

В процесі роботи алгоритм знаходить наступний доповнювальний шлях за час O(E). На кожному кроці одне з E ребер стає ребром з найменшою вагою, а також довжина шляху на кожному кроці буде не більше, ніж V. Звідси ми отримуємо загальну часову складність алгоритму $O(VE^2)$.

6.2.1. Формальний опис алгоритму

1. Обнуляємо всі потоки. Залишкова мережа спочатку збігається з початковою мережею.
2. У залишковій мережі знаходимо найкоротший шлях з джерела в стік. Якщо такого шляху немає, зупиняємося.
3. Пускаємо через знайдений шлях (він називається збільшувальним шляхом або збільшувальним ланцюгом) максимально можливий потік:
   - на знайденому шляху в залишковій мережі шукаємо ребро з мінімальною пропускною здатністю c_{min};
   - для кожного ребра на знайденому шляху збільшуємо потік на c_{min}, а в протилежному йому — зменшуємо на c_{min};
   - модифікуємо залишкову мережу. Для всіх ребер на знайденому шляху, а також для протилежних їм ребер, обчислюємо нову пропускну здатність. Якщо вона стала ненульовою, додаємо ребро до залишкової мережі, а якщо обнулилась, стираємо його.
4. Повертаємося на крок 2.

Алгоритм пошуку в ширину був описаний в підрозділі 3.1 цього підручника.

6.2.2. Реалізація алгоритму

В цьому параграфі поданий один з варіантів реалізації алгоритму Едмондса-Карпа мовою C++.

Матриця `capacity` зберігає ваги ребер. Матриця `adj` це список суміжності, який описує мережу у вигляді неорієнтованого графа, оскільки нам потрібно мати зворотні ребра для пошуку доповнювального шляху.

Функція `maxflow` повертає значення максимального потоку. Протягом виконання алгоритму, матриця `capacity` зберігає залишкові ваги ребер.

Значення потоку для кожного ребра не зберігається, але за потреби це можна зробити додавши окрему матрицю.

```
int n;
vector<vector<int>> capacity, adj;
int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1); parent[s] = -2;
    queue<pair<int, int>> q; q.push({s, INF});
    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();
        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({next, new_flow});
            }
        }
    }
    return 0;
}
int maxflow(int s, int t) {
    int flow = 0, new_flow = 0;
    vector<int> parent(n);
    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }
    return flow;
}
```

6.3. Алгоритм Дініца

Вперше покращення алгоритму Форда-Фалкерсона було описане радянським (в теперішній час ізраїльським) вченим Юхимом Дініцом у 1970 році. Алгоритм отримав назву на честь свого винахідника . Обчислювальна складність алгоритму становить $O(V^2E)$. Отримати таку оцінку дозволяє введення понять допоміжної мережі та блокувального (псевдомаксимального) потоку.

6.3.1. Основні визначення

В доповнення до поняття залишкова мережа, алгоритм Дініца вводить поняття **допоміжної мережі**. В цій мережі використовуються вершини з початкової мережі але окрім мітки, кожна вершина також має рівень віддаленості від джерела. Віддаленість вимірюється кількістю вершин між джерелом і поточною вершиною. Рівні вершин в цьому графі отримуються шляхом застосування BFS до залишкової мережі. Ребра в допоміжній мережі відповідають ребрам в залишковій мережі. Але не всі ребра із залишкової мережі присутні в допоміжній. В допоміжну мережу додаються лише ті ребра, які йдуть від вершин з нижчим рівнем до вершин з вищим рівнем. Це дозволяє виключити з розгляду ребра, що йдуть в зворотному напрямку — від стоку до джерела. Ще однією умовою додавання ребра у допоміжну мережу є його залишковий потік — він повинен бути більшим за 0.

Блокувальний потік — це такий потік, після якого неможливо побудувати маршрут від джерела до стоку в допоміжній мережі. Блокувальний потік може складатися з кількох шляхів від джерела до стоку і кожен з них знаходиться алгоритмом DFS.

6.3.2. Формальний опис алгоритму

1. Створити допоміжну мережу у вигляді графу використавши алгоритм BFS для залишкової мережі.
2. Якщо стік недосяжний під час побудови допоміжної мережі, то зупинитися і повернути максимальний потік.
3. Використавши алгоритм DFS на допоміжній мережі, побудувати всі можливі шляхи від джерела до стоку. Додати всі найменші ребра на кожному шляху для отримання максимального блокувального потоку.
4. Перейти до першого кроку.

6.3.3. Приклад виконання алгоритму

Розглянемо покроковий приклад виконання алгоритму Дініца. На кожному кроці будемо оновлювати залишкову та допоміжну мережі. Початкова залишкова мережа подана у вигляді графа (рис. 6.10).

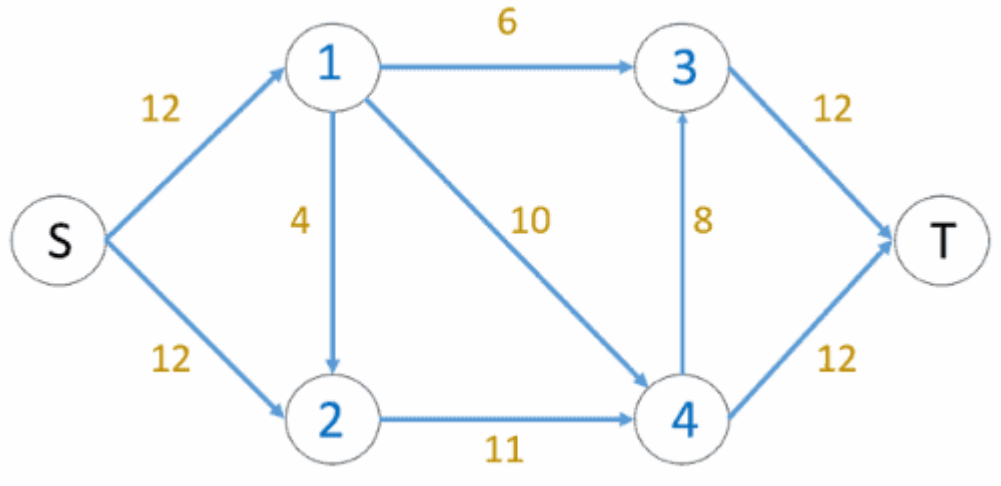


Рис. 6.10. Залишкова мережа

Крок 1. Використовуємо алгоритм BFS для визначення рівнів вершин допоміжної мережі. Видаляємо ребра, що не з'єднують нижчі рівні з вищими і таким чином отримуємо граф допоміжної мережі (рис. 6.11).

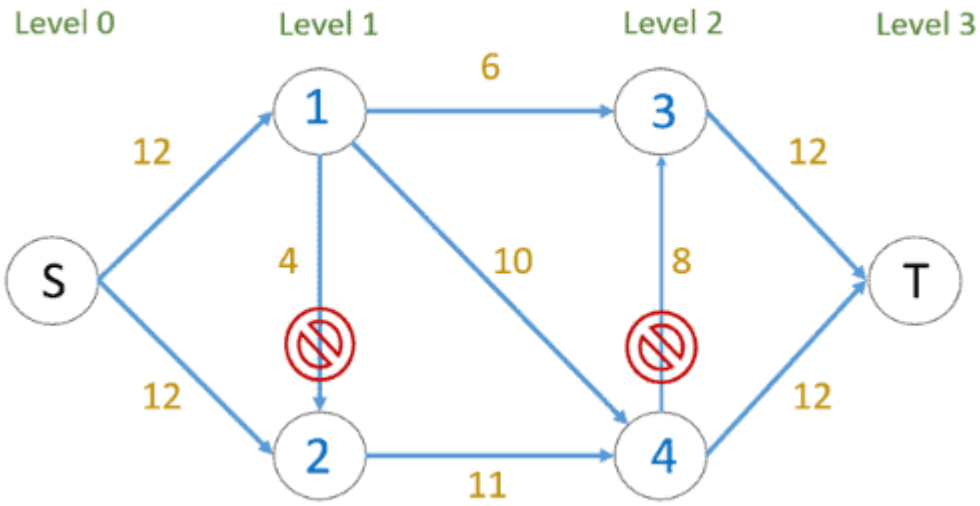


Рис. 6.11. Допоміжна мережа на першому кроці

Знаходимо блокувальний потік використовуючи рівні на графі. Кожен маршрут в блокувальному потоці повинен містити вершини з рівнями 0, 1, 2, 3 саме в такій послідовності. На цьому кроці ми маємо три таких потоки:

- S->1->3->T з максимальним потоком 6;
- S->1->4->T з максимальним потоком 6;
- S->2->4->T з максимальним потоком 6.

Загальний блокувальний потік розраховується як сума всіх потоків: 6+6+6=18.

Оновлюємо залишкову мережу з врахуванням знайдених потоків (так само як і в алгоритмі Форда-Фалкерсона) і отримуємо наступний граф (рис. 6.12).

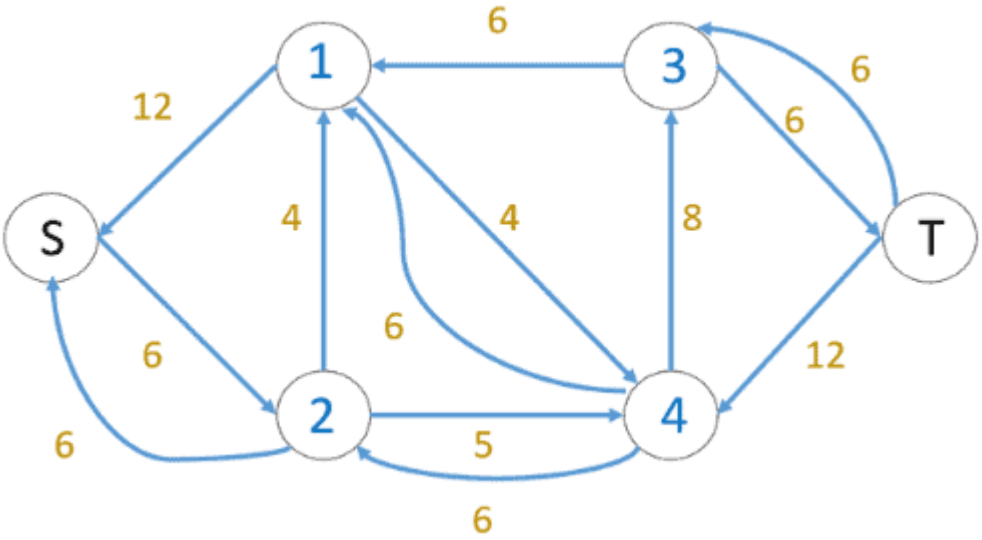


Рис. 6.12. Залишкова мережа після першого кроку

Крок 2. Знову використовуємо алгоритм BFS на оновленій залишковій мережі та будуємо допоміжну мережу (рис. 6.13).

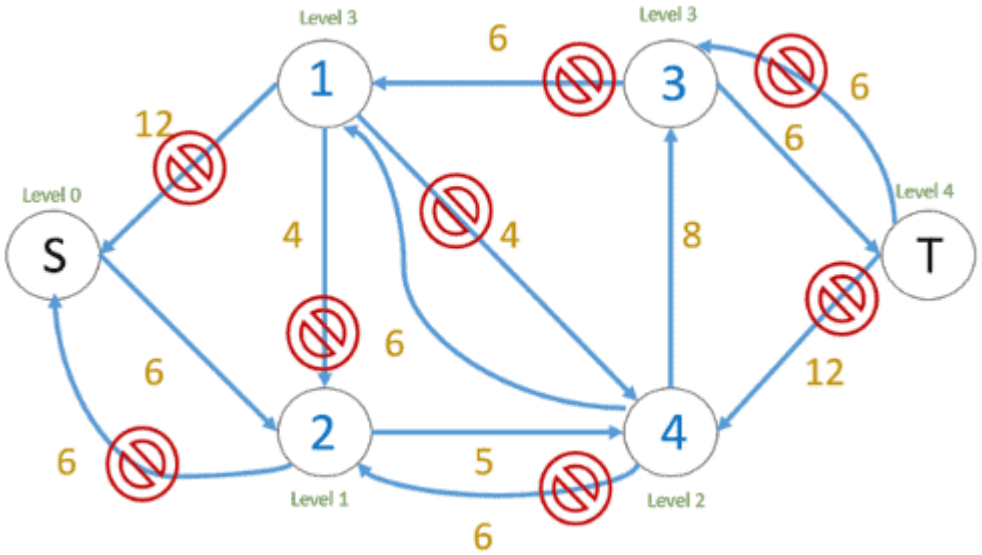


Рис. 6.13. Допоміжна мережа на другому кроці

Ребра, що не відповідають умовам додавання в допоміжну мережу, позначені знаком блокування. На цьому кроці в нас є вже п'ять рівнів у допоміжній мережі, а отже кожен шлях в блокувальному потоці повинен містити вершини з рівнями 0, 1, 2, 3, 4 саме в такій послідовності. В цьому випадку в нас є лише один такий шлях: S->2->4->3->T з максимальним потоком 5. Загальний блокувальний потік розраховується як сума всіх потоків додана до попереднього максимального потоку: 5+18=23.

Оновлюємо залишкову мережу з врахуванням знайденого потоку і отримуємо наступний граф (рис. 6.14).

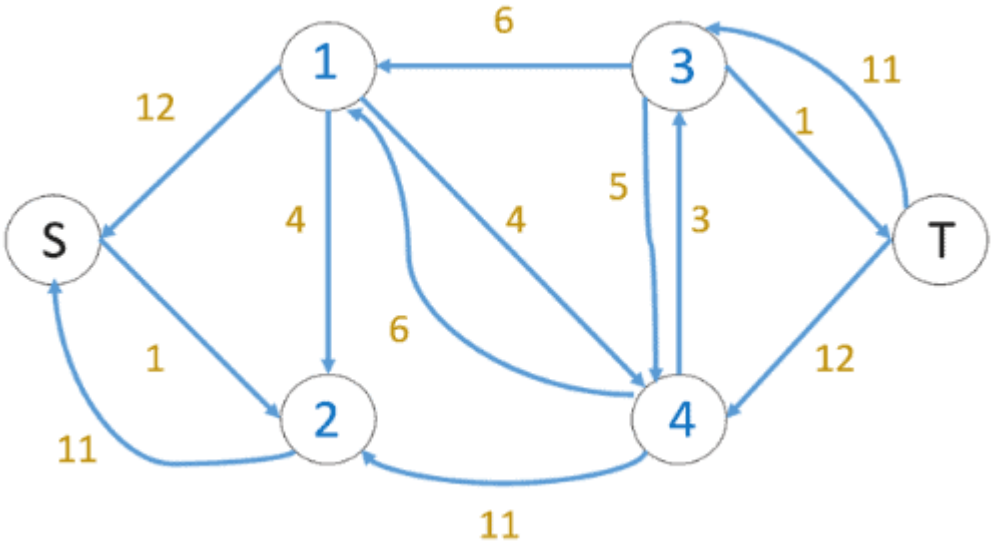


Рис. 6.14. Залишкова мережа після другого кроку

Крок 3. Знову запускаємо алгоритм BFS на залишковій мережі. Цього разу ми не можемо досягнути стоку, а отже максимальний потік не може бути розширений і алгоритм закінчує свою роботу. Поточний максимальний потік є дійсно максимальним.

6.4. Алгоритм просування передпотоку

Алгоритм просування передпотоку (Preflow Push or Push Relabel) розроблений американськими вченими Ґолдбергом і Тарджаном у 1986 році. Використовується для визначення максимального потоку в мережі.

Ідея, що лежить в основі цього алгоритму, в тому, що ребра графа розглядаються як труби, а вершини графа — як з'єднання. Тобто граф описує водопровідну мережу, де відбуваються перетоки рідини від вершини з найбільшою висотою до вершини з найменшою висотою. Для цього кожній вершині присвоюється мітка — її висота. Відповідно джерело має найбільшу початкову висоту, що дорівнює кількості вершин графа (V), а стік має нульову висоту. Всі висоти подаються цілими числами.

6.4.1. Основні визначення

Окрім поняття **висота**, алгоритм також оперує поняттям **надмірний потік** у вершині. Надмірний потік розраховується як різниця між всіма вхідними потоками у вершину та всіма вихідними потоками із вершини (рис. 6.15).

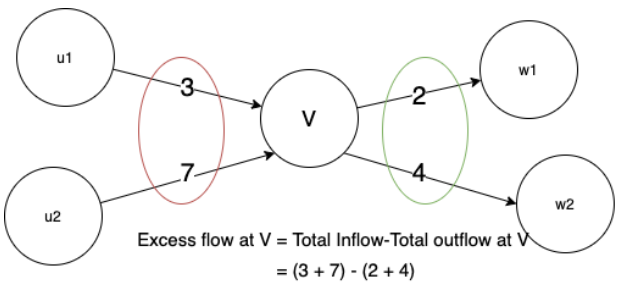


Рис. 6.15. Ілюстрація надмірного потоку

Також алгоритм виконує дві операції, що визначені в одній з його англійських назв.

Просування (Push). Якщо вершина має надмірний потік, то він просувається до сусідньої вершини з меншою висотою.

Підйом (Relabel). Якщо вершина має надмірний потік і немає сусідньої вершини з меншою висотою щоб зробити просування, тоді відбувається підйом цієї вершини — збільшення її висоти на одиницю.

6.4.2. Формальний опис та псевдокод

Кожна вершина мережі в алгоритмі просування передпотоку має три значення: номер, висоту і значення надмірного потоку. Надмірний потік не використовується для джерела і стоку, оскільки він не має сенсу для них.

На початку алгоритму джерело ініціалізується максимальною висотою, що дорівнює кількості вершин графа. Всі інші вершини отримують нульове значення висоти та надмірного потоку. Операції просування та підйому виконуються до тих пір, поки надмірний потік в кожній вершині (окрім джерела і стоку) не стане нульовим.

Нижче подано узагальнений псевдокод алгоритму (G — мережа, c — пропускна здатність ребер (ємність), s — джерело, t — стік, ℓ — висота, v — вершина).

```
generic-push-relabel(G, c, s, t):
    create a pre-flow f that saturates all out-arcs of s
    let ℓ[s] = |V|
    let ℓ[v] = 0 for all v ∈ V \ {s}
    while there is an applicable push or relabel operation do
        execute the operation
```

Операція просування відбувається якщо надмірний потік (x_f) більший за 0 і висота поточної вершини (u) більша на 1 за висоту сусідньої вершини (v). Тоді розраховується величина потоку, який буде просунуто, як мінімум між надмірним потоком поточної вершини і надмірним потоком сусідньої — в яку буде здійснюватися просування. Далі відбувається коригування потоків на прямому і зворотному ребрах між поточною і сусідньою вершинами, а також надмірних потоків цих вершин. Псевдокод для операції просування наступний:

```
push(u, v):
    assert xf[u] > 0 and ℓ[u] == ℓ[v] + 1
    Δ = min(xf[u], c[u][v] - f[u][v])
    f[u][v] += Δ
    f[v][u] -= Δ
    xf[u] -= Δ
    xf[v] += Δ
```

Операція підйому поточної вершини (u) відбувається якщо її надмірний потік більший за 0 та її висота не більша за висоту будь-якої із сусідніх вершин до яких є залишковий потік. В такому випадку її висота стає на одиницю більшою за мінімальну висоту серед всіх її сусідів. Псевдокод для операції підйому наступний:

```
relabel(u):
    assert x_f[u] > 0 and ℓ[u] <= ℓ[v] for all v such that c[u][v] > 0
    ℓ[u] = 1 + min(ℓ[v] for all v such that c[u][v] > 0)
```

Обчислювальна складність алгоритму становить $O(V^2E)$.

6.4.3. Приклад виконання алгоритму

Розглянемо покрокове виконання алгоритму на прикладі мережі (рис. 6.16).

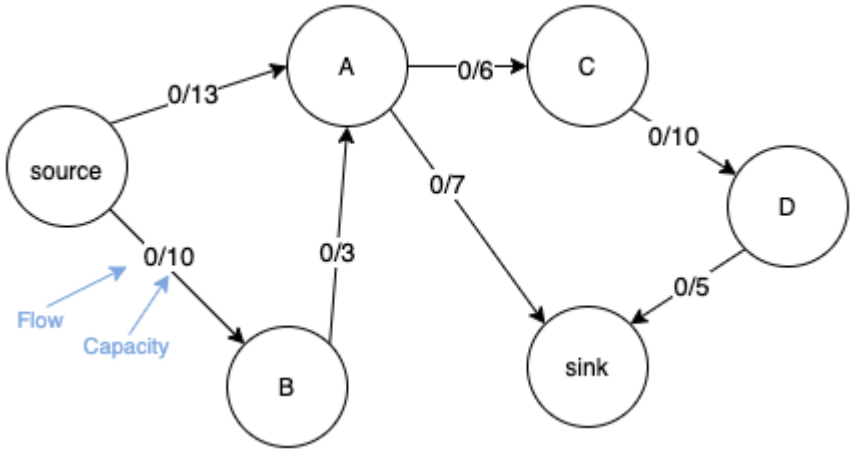
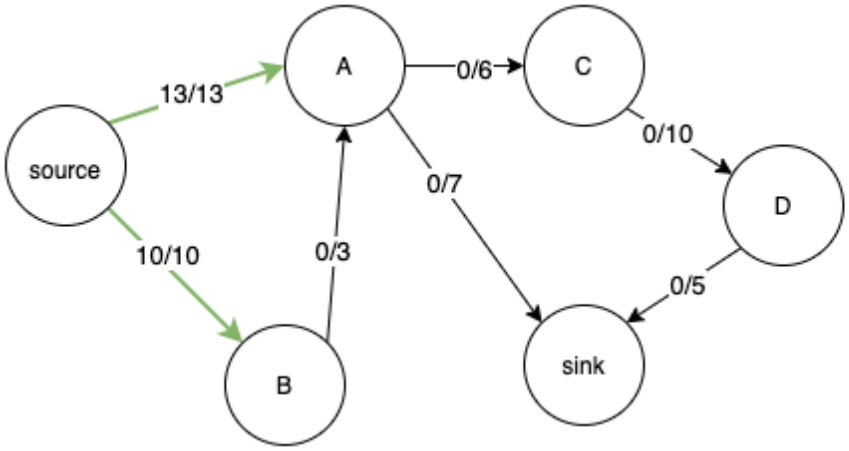


Рис. 6.16. Граф мережі

Крок 1. Ініціалізуємо граф — задаємо початкову висоту вершин і виконуємо операцію просування для джерела (рис. 6.17).

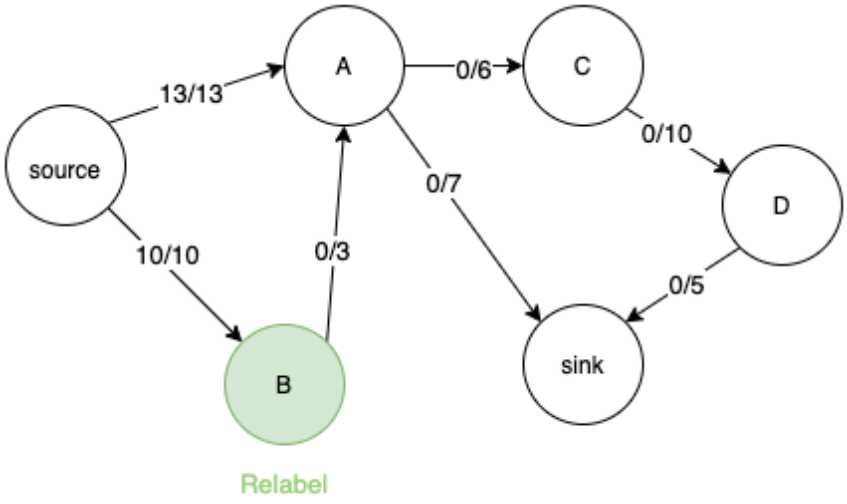


Node	Height	Excess Flow
source	6	
A	0	13
B	0	10
C	0	
D	0	
sink	0	

Рис. 6.17. Мережа після ініціалізації та початкового просування джерела

Крок 2. Вершина B, має найменший надмірний потік, тому ми розглядаємо її наступною. Ми не можемо здійснити просування надмірного потоку з цієї

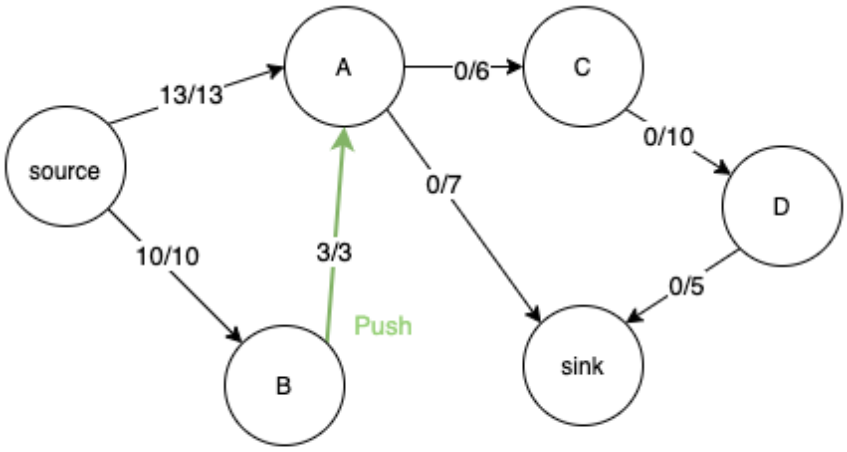
вершини, оскільки немає сусідньої вершини з меншою висотою. Тому нам необхідно здійснити підйом цієї вершини (рис. 6.18).



Node	Height	Excess Flow
source	6	-
A	0	13
B	1	10
C	0	
D	0	
sink	0	-

Рис. 6.18. Підйом вершини B

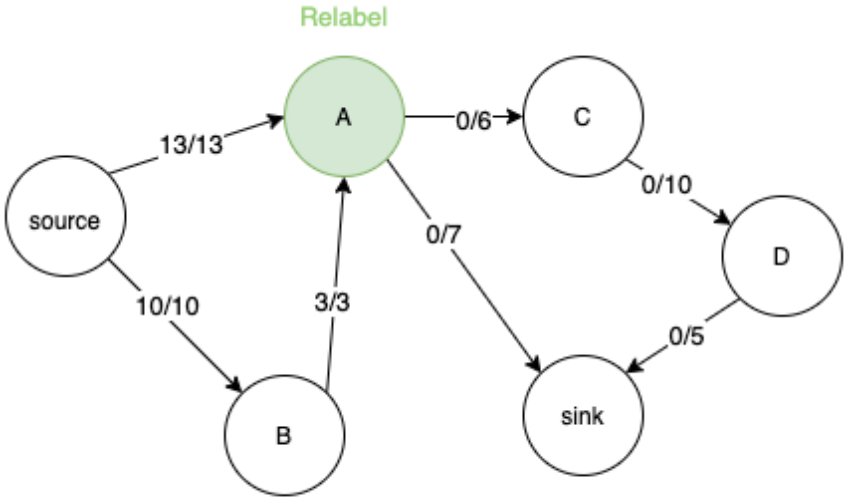
Крок 3. Після підйому вершини B, вона стала вищою за сусідню вершину A, а тому ми можемо здійснити просування від B до A (рис. 6.19).



Node	Height	Excess Flow
source	6	-
A	0	16
B	1	7
C	0	
D	0	
sink	0	-

Рис. 6.19. Просування потоку від B до A

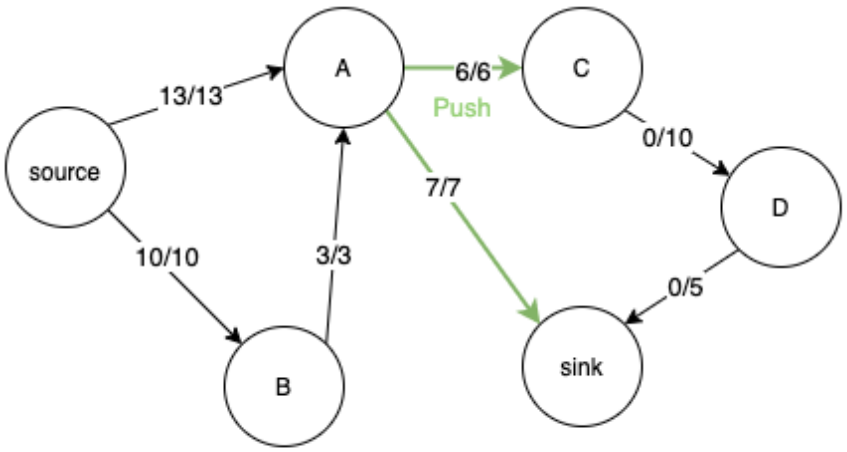
Крок 4. Тепер надмірний потік серед всіх вершин із залишковим потоком має лише вершина A. Ми не можемо здійснити просування з цієї вершини, оскільки немає сусідньої вершини з меншою висотою, а тому робимо підйом A (рис. 6.20).



Node	Height	Excess Flow
source	6	-
A	1	16
B	1	7
C	0	
D	0	
sink	0	-

Рис. 6.20. Підйом вершини A

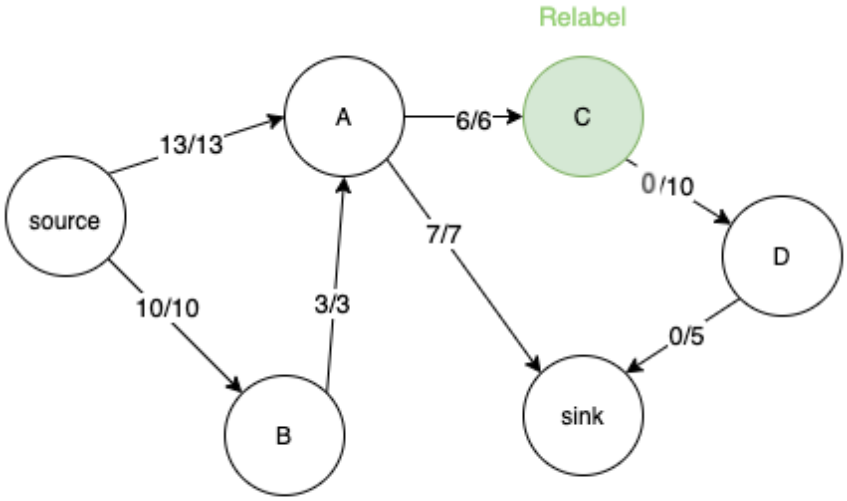
Крок 5. Після підйому вершини A, вона може здійснити просування потоку до своїх сусідніх вершин C і `sink` (стік) (рис. 6.21).



Node	Height	Excess Flow
source	6	-
A	1	3
B	1	7
C	0	6
D	0	
sink	0	-

Рис. 6.21. Просування потоку від A до C і `sink`

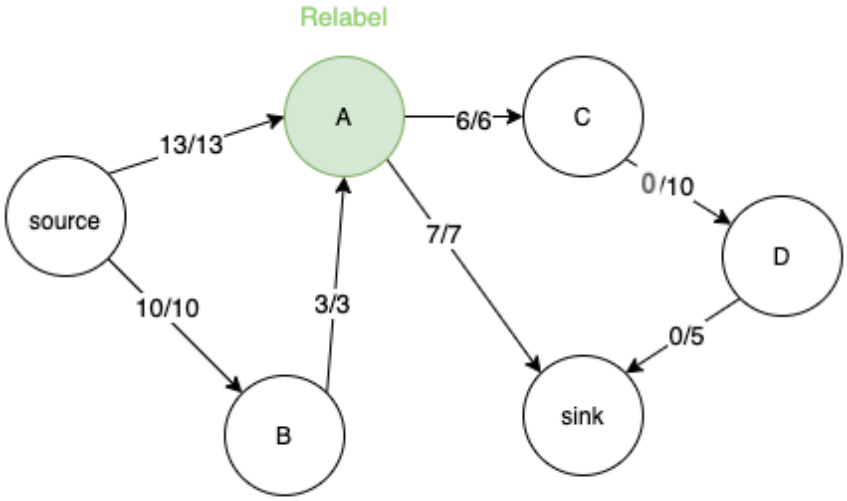
Крок 6. Наступна вершина з надмірним потоком і доступним залишковим потоком це вершина C. Ми не можемо зробити операцію просування для цієї вершини, оскільки немає сусідньої вершини з меншою висотою, а тому ми здійснюємо підйом вершини C (рис. 6.22).



Node	Height	Excess Flow
source	6	-
A	1	3
B	1	7
C	1	6
D	0	0
sink	0	-

Рис. 6.22. Підйом вершини C

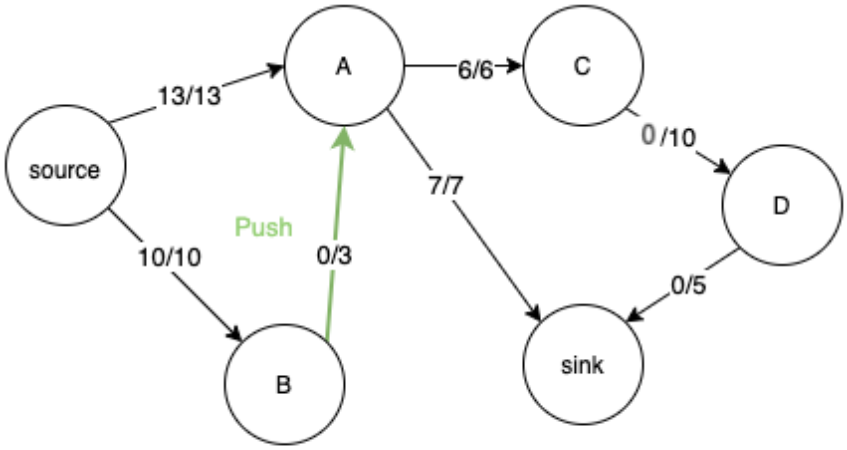
Крок 7. Після підйому вершини C, потік не може рухатися від A до C, оскільки обидві вершини мають однакову висоту. Тому необхідно знову здійснити підйом вершини A (рис. 6.23).



Node	Height	Excess Flow
source	6	-
A	2	3
B	1	7
C	1	6
D	0	0
sink	0	-

Рис. 6.23. Підйом вершини A

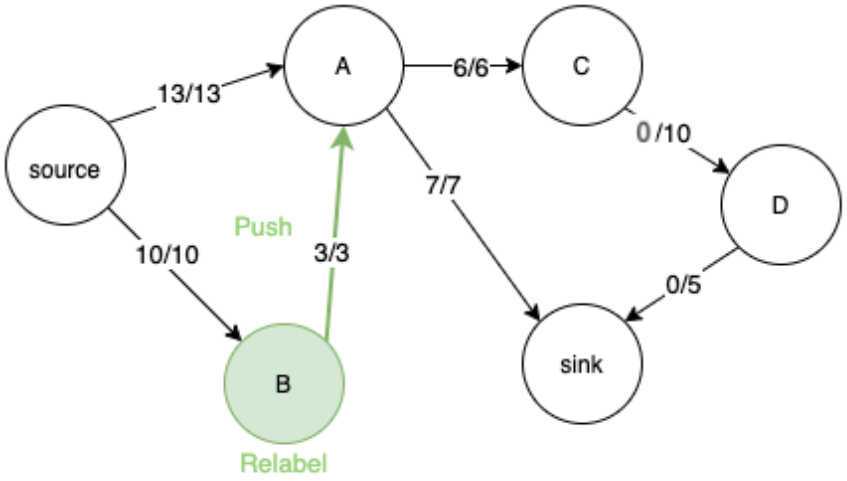
Крок 8. Тепер, після підйому вершини A, ми можемо здійснити просування потоку назад до вершини B, оскільки її висота менша, за висоту A (рис. 6.24).



Node	Height	Excess Flow
source	6	-
A	2	0
B	1	10
C	1	6
D	0	0
sink	0	-

Рис. 6.24. Просування потоку від A до B

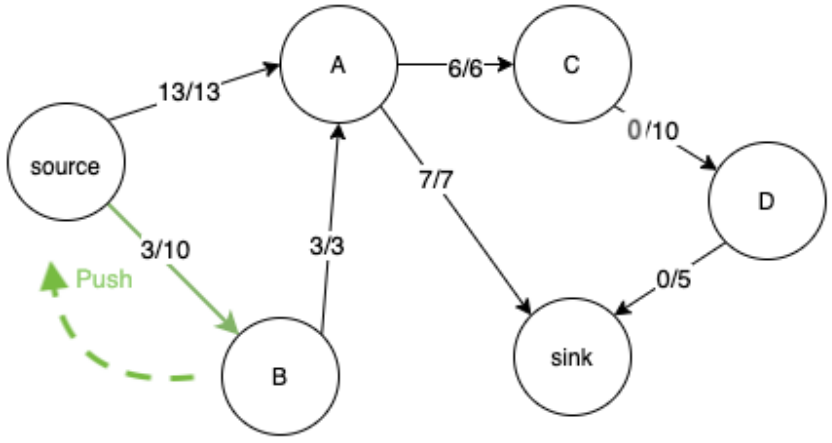
Крок 9. Оскільки вершина B тепер має надмірний потік нам потрібно здійснити її підйом, а потім просування потоку назад до вершини A (рис. 6.25).



Node	Height	Excess Flow
source	6	-
A	2	3
B	3	7
C	1	6
D	0	0
sink	0	-

Рис. 6.25. Просування потоку від B до A

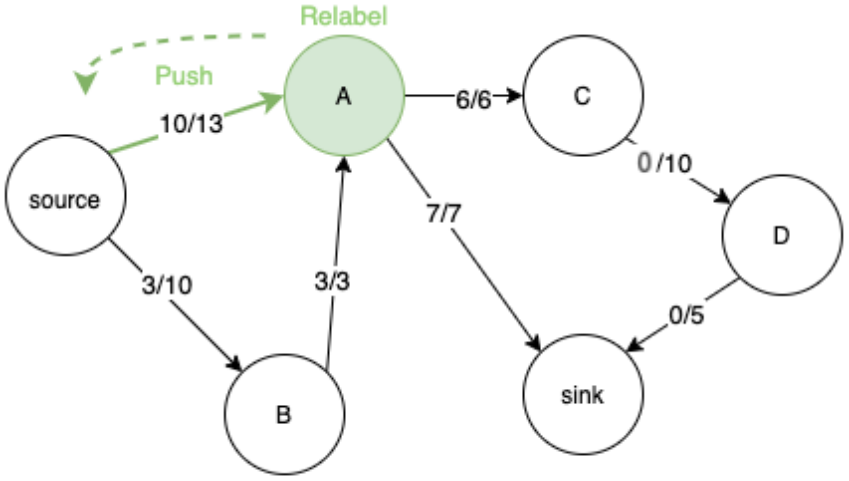
Крок 10. Просування між вершинами A і B відбуватимуться доти, доки вершина B не стане вищою за `source`. Тоді надмірний потік просунеться назад (рис .6.26).



Node	Height	Excess Flow
source	6	-
A	6	3
B	7	0
C	1	6
D	0	0
sink	0	-

Рис. 6.26. Просування потоку від B до `source`

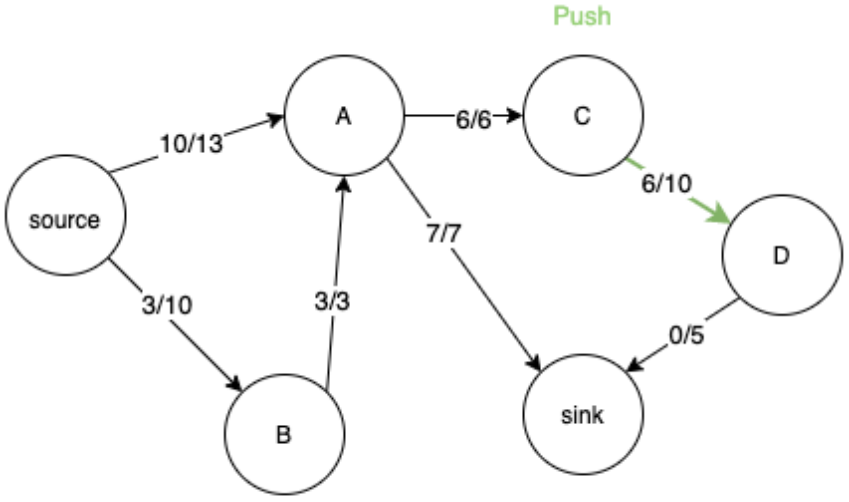
Крок 11. Подібним чином після підйому вершини A, вона стане вищою за джерело і зможе просунути надмірний потік назад (рис. 6.27).



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	1	6
D	0	0
sink	0	-

Рис. 6.27. Просування потоку від A до `source`

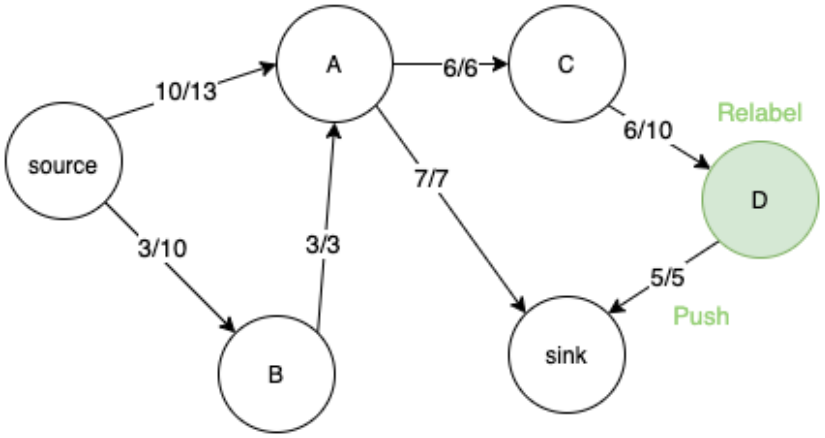
Крок 12. Тепер єдина вершина з надмірним потоком це вершина C. Вона вища за сусідню вершину D, а тому ми робимо просування потоку до неї (рис. 6.28).



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	1	0
D	0	6
sink	0	-

Рис. 6.28. Просування потоку від C до D

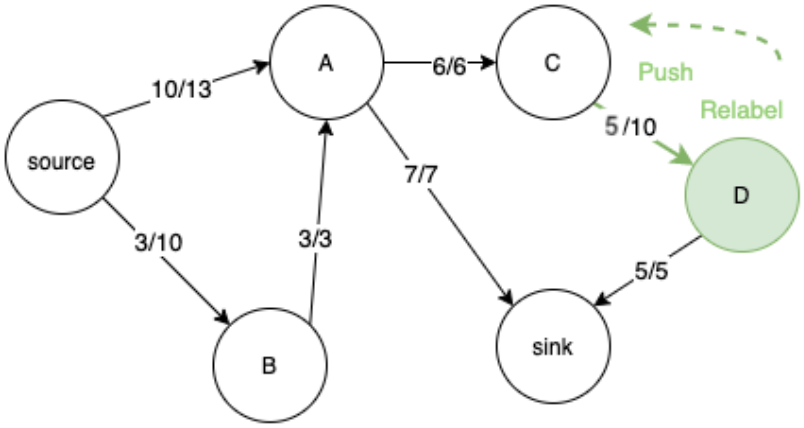
Крок 13. Тепер здійснюємо підйом вершини D та просування надмірного потоку до `sink` (стік) (рис. 6.29). Проте, після просування потоку в напрямку стоку, в нас ще залишається одна одиниця надмірного потоку у вершині D, а тому алгоритм не закінчено і він продовжує працювати далі.



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	1	0
D	1	1
sink	0	-

Рис. 6.29. Просування потоку від D до `sink`

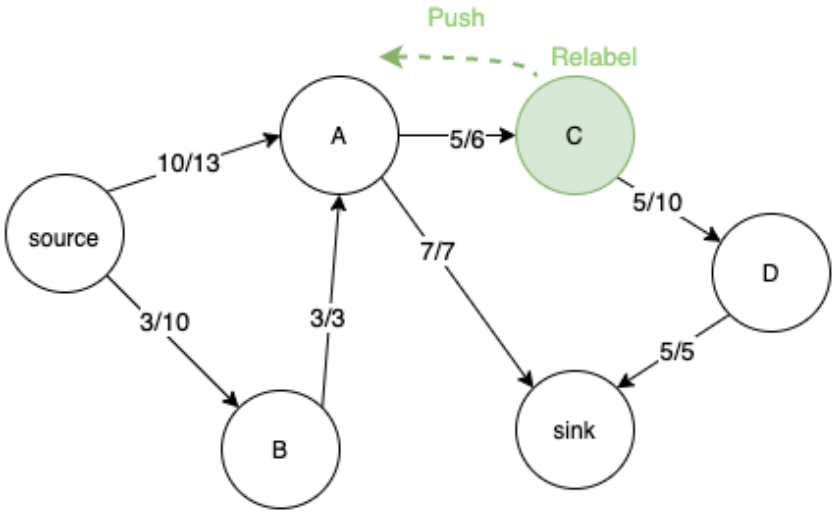
Крок 14. Тепер ми здійснюємо підйом вершини D та просування потоку назад до вершини C (рис. 6.30).



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	1	1
D	2	0
sink	0	-

Рис. 6.30. Просування потоку від D до C

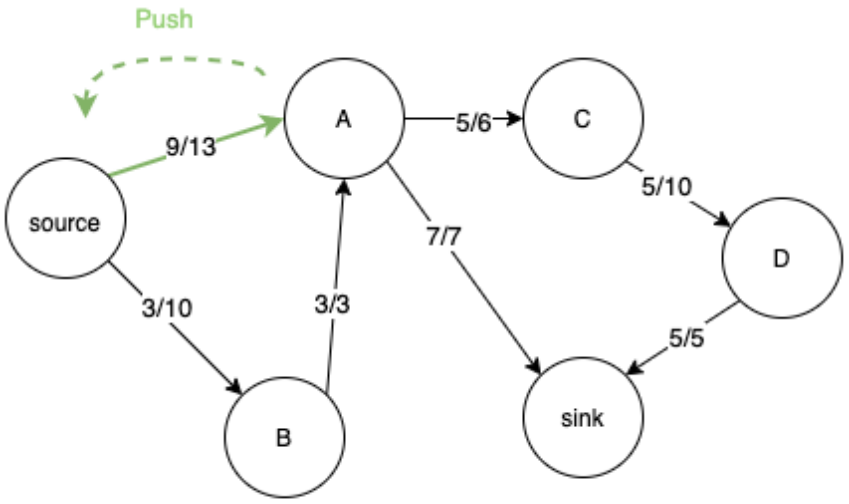
Крок 15. Здійснюємо підйом C та просування потоку назад до A (рис. 6.31).



Node	Height	Excess Flow
source	6	-
A	8	1
B	7	0
C	9	0
D	8	0
sink	0	-

Рис. 6.31. Просування потоку від C до A

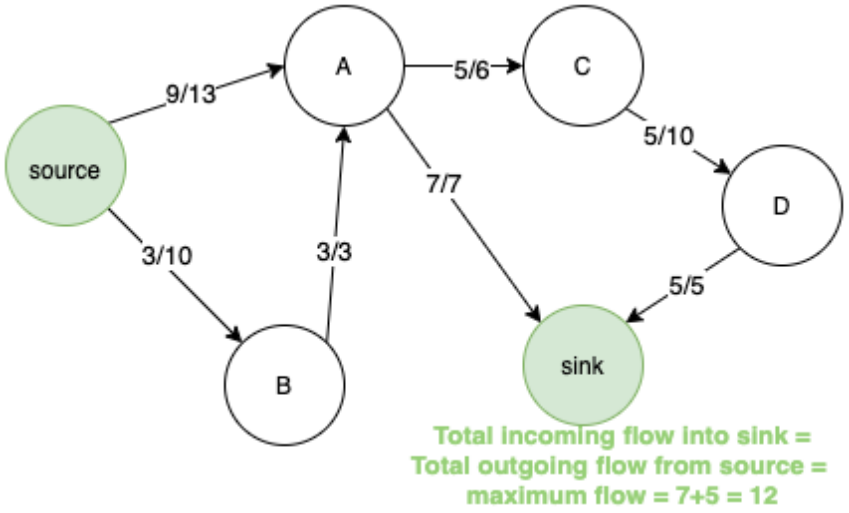
Крок 16. Оскільки висота вершини A більша за висоту джерела, то ми здійснюємо просування потоку від вершини A назад до джерела (рис. 6.32).



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	9	0
D	8	0
sink	0	-

Рис. 6.32. Просування потоку від A до `source`

Крок 17. Тепер надмірні потоки у всіх вершинах між джерелом і стоком дорівнюють нулю, а отже алгоритм закінчено. Щоб розрахувати максимальний потік, необхідно додати всі вихідні потоки з джерела (9+3=12) або всі вхідні потоки в стік (7+5=12) (рис. 6.33).



Node	Height	Excess Flow
source	6	-
A	8	0
B	7	0
C	9	0
D	8	0
sink	0	-

Рис. 6.33. Максимальний потік мережі

6.5. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач на максимальний потік з використанням різних алгоритмів, що були розглянуті в цьому розділі.

Ось перелік задач про максимальний потік на платформі HackerEarth середнього та максимального рівня складності.

Середній рівень складності

1. Shil and Lab Assignment
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/shil-and-lab-assignment-14.
2. Find the Flow
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/find-the-flow.
3. Scheduling War
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/scheduling-war.

Високий рівень складності

1. Decaying Roads
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/decaying-roadsnov-easy-8e930584.
2. Telecom Towers
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/telecom-towers-06c98fbd.
3. Shubham and Grid
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/shubham-and-grid-806c2c66.
4. Replace
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/replace.
5. Beautiful Badges
   https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/practice-problems/algorithm/beautiful-badges.

Запитання для самоконтролю

1. Сформулюйте за дачу про максимальний потік.
2. Хто і коли вперше сформулював задачу про максимальний потік в загальному вигляді?
3. Які алгоритми для розв'язку задачі про максимальний потік ви знаєте?
4. Що таке залишкова мережа?
5. Що таке доповнювальний шлях?
6. Які додаткові поняття ввів Дініц для розв'язку задачі про максимальний потік?
7. Які методи обходу графа використовує алгоритм Дініца?
8. Які додаткові властивості вершин графа використовуються в алгоритмі просування передпотоку?
9. Які основні операції використовуються в алгоритмі просування передпотоку?

10. Яка обчислювальна складність розглянутих алгоритмів?

ПОСИЛАННЯ НА ВИКОРИСТАНІ ДЖЕРЕЛА

1. https://uk.wikipedia.org/wiki/Задача_про_максимальний_потік
2. https://uk.wikipedia.org/wiki/Алгоритм_Форда_—_Фалкерсона
3. https://uk.wikipedia.org/wiki/Алгоритм_Едмондса_—_Карпа
4. https://en.wikipedia.org/wiki/Maximum_flow_problem
5. https://en.wikipedia.org/wiki/Ford—Fulkerson_algorithm
6. https://en.wikipedia.org/wiki/Edmonds—Karp_algorithm
7. https://www.programiz.com/dsa/ford-fulkerson-algorithm
8. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem
9. https://medium.com/@jithmisha/solving-the-maximum-flow-problem-with-ford-fulkerson-method-3fccc2883dc7
10. https://cp-algorithms.com/graph/edmonds_karp.html
11. https://www.w3schools.com/dsa/dsa_algo_graphs_edmondskarp.php
12. https://www.baeldung.com/cs/dinics
13. https://medium.com/smucs/understanding-dinics-algorithm-ebf892e66227
14. https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow
15. https://cp-algorithms.com/graph/dinic.html
16. https://iq.opengenus.org/push-relabel-algorithm

7. НАЙКОРОТШИЙ ШЛЯХ В ГРАФІ

В теорії графів, задача про найкоротший шлях полягає в знаходженні такого шляху між двома вершинами (або вузлами) графа, що сума ваг ребер, з яких він складається, мінімальна. Прикладом може бути знаходження найкоротшого шляху між двома пунктами на дорожній мапі; в цьому випадку, вершинами є пункти, а ребрами — відтинки дороги із вагами, рівними часу, необхідному для подолання цього відрізку.

Існує декілька варіацій постановки задачі для знаходження найкоротшого шляху:

- між двома вершинами графа (найбільш розповсюджений);
- між однією стартовою вершиною і всіма іншими вершинами;
- між всіма вершинами і однією кінцевою вершиною;
- між всіма парами вершин в графі.

Також для вирішення задачі пошуку найкоротшого шляху в графі необхідно знати, який саме тип графа використовується. Для різних типів графів та різних обмежень на їхні ребра використовуються різні алгоритми.

Щоб знайти найкоротший шлях в незваженому графі цілком достатньо використати алгоритм BFS . Якщо граф зважений, тоді потрібно застосувати один зі спеціалізованих алгоритмів, які будуть розглянуті в цьому розділі.

7.1. Алгоритм Дейкстри

Алгоритм Дейкстри винайдений нідерландським вченим Едсгером Дейкстрою в 1956 році для знаходження найкоротшого шляху від однієї вершини до всіх інших вершин графа. За твердженням самого Дейкстри, він винайшов алгоритм за 20 хвилин сидячи на терасі кафе в Амстердамі без використання паперу і олівця.

Класичний алгоритм Дейкстри працює тільки для графів без ребер від'ємної довжини, що робить його використання обмеженим. Хоча в більшості реальних задач застосовується саме він.

7.1.1. Формальний опис та псевдокод

Розглянемо порядок знаходження найкоротших шляхів від стартової вершини до всіх інших вершин графа за алгоритмом Дейкстри.

1. Позначити всі вершини як невідвідані алгоритмом і додати їх у список.
2. Визначити стартову вершину шляху і присвоїти відстань до неї 0. Для всіх інших вершин встановити відстань, що дорівнює нескінченності.
3. Вибрати зі списку невідвіданих вершин таку, що має найкоротшу відстань від стартової вершини.
4. Для поточної обраної вершини розглянути всі невідвідані суміжні вершини та оновити відстані до них, якщо потрібно.

5. Позначити поточну вершину як відвідану та вилучити її зі списку невідвіданих.
6. Повторювати кроки 3 – 5 до тих пір, поки список невідвіданих не стане порожнім або в ньому залишаться лише вершини з нескінченними відстанями (це означає, що вони недосяжні зі стартової вершини).

Нижче подано узагальнений псевдокод алгоритму.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with minimum dist[u]
        remove u from Q
    for each neighbor v of u still in Q:
        alt ← dist[u] + Graph.Edges(u, v)
        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
    return dist[], prev[]
```

Поданий вище псевдокод використовує структуру графа, що містить список вершин та матрицю ваг ребер. Час роботи такої реалізації буде $O(V^2)$.

Для покращення часу роботи алгоритму Дейкстри в його реалізації використовують чергу з пріоритетами. Ця структура даних фактично виконує сортування елементів під час додавання або зміни пріоритетів (в залежності від реалізації). Якщо розглядати класичну чергу з пріоритетами без функції зміни пріоритетів, то тоді її необхідний розмір буде $O(E)$, а час роботи такої реалізації буде $O(E \log V)$.

Псевдокод реалізації на черзі з пріоритетами поданий нижче.

```
function Dijkstra(Graph, source):
    create vertex priority queue Q
    dist[source] ← 0    // Ініціалізація
    Q.push(source, 0)   // Додаємо стартову вершину з відстанню 0
    for each vertex v in Graph.Vertices:
        if v ≠ source
            prev[v] ← UNDEFINED // Попередня вершина до вершини v
            dist[v] ← INFINITY  // Невідома відстань до вершини v
```

```
    while Q is not empty:          // Головний цикл
        u ← Q.pop()                // Витягуємо вершину з черги
        for each neighbor v of u: // Перевіряємо всі v суміжні до u
            alt ← dist[u] + Graph.Edges(u, v)
                if alt < dist[v]:
                    prev[v] ← u
                    dist[v] ← alt
                    Q.push(v, alt) // Додаємо вершину в чергу
    return dist, prev
```

В чергу з пріоритетами ми додаємо вершини і відстані до них від стартової вершини. Оскільки відстань до однієї і тієї ж вершини може бути отримана через різні ребра, то максимальна кількість таких вершин в черзі може бути рівній кількості ребер. Це також означає, що ми можемо мати одночасно декілька однакових вершин в черзі, але з різними відстанями до них.

7.1.2. Приклад виконання алгоритму

Розглянемо покроковий приклад виконання алгоритму Дейкстри на прикладі простого неорієнтованого зваженого графа (рис. 7.1).

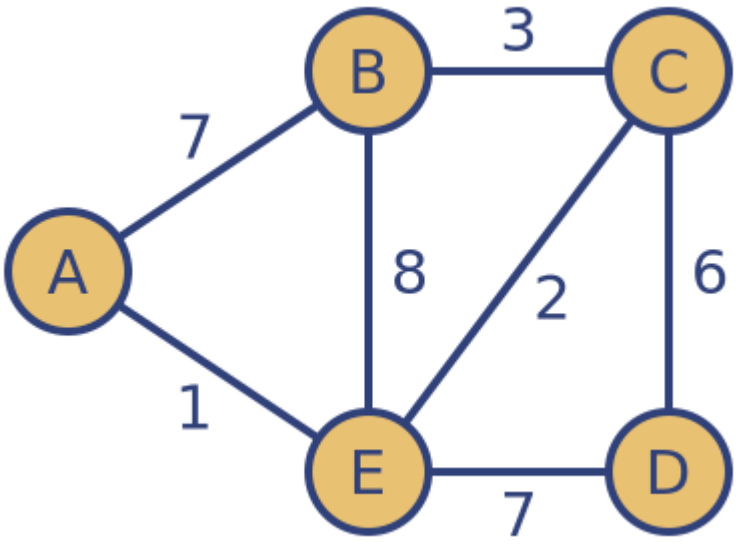
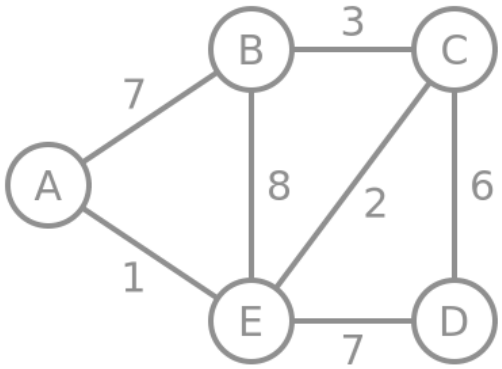


Рис. 7.1. Неорієнтований зважений граф

Алгоритм працює крок за кроком і знаходить найкоротші шляхи до всіх вершин графа з початкової вершини A. На початку позначимо всі вершини і ребра, які ще не опрацьовані, сірим кольором та складемо таблицю станів вершин (рис. 7.2).



	Vis	Dist	Prev
A	0	∞	-
B	0	∞	-
C	0	∞	-
D	0	∞	-
E	0	∞	-

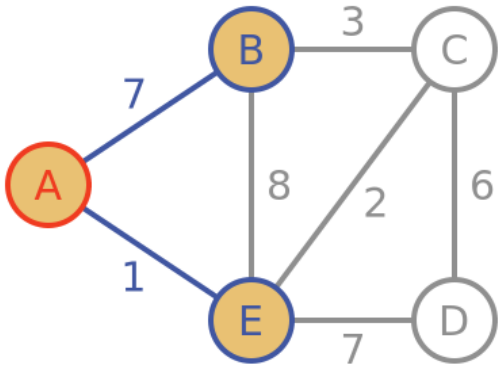
Рис. 7.2. Початковий стан алгоритму

Таблиця містить наступні значення:

- `Vis` — відвідана вершина. Вершина стає відвіданою, коли вже відома найкоротша відстань до неї від стартової вершини;
- `Dist` — найкоротша відстань від стартової вершини до поточної;
- `Prev` — попередня вершина в найкоротшому маршруті. Це дозволить відновити найкоротший шлях коли алгоритм закінчить свою роботу.

Крок 1. Алгоритм починає роботу зі стартової вершини A, а тому ми знаємо, що відстань від A до A нуль. Тобто ми можемо відмітити, що вершина A є відвіданою.

Далі необхідно перевірити всі суміжні вершини до вершини A. Такими є вершини B та E. Поточний найкоротший шлях з A до B стає 7, а з A до E — 1. Для обох вузлів попередньою вершиною стає A. Оскільки ми ще не дослідили всі можливі шляхи в суміжні вершини, то вони ще не стають відвіданими. На цьому етапі вони стають досяжними, а тому ми їх позначимо синім кольором. Відвіданою залишається лише вершина A, яку ми позначимо червоним кольором (рис. 7.3).



	Vis	Dist	Prev
A	1	0	-
B	0	7	A
C	0	∞	-
D	0	∞	-
E	0	1	A

Рис. 7.3. Стан алгоритму після першого кроку

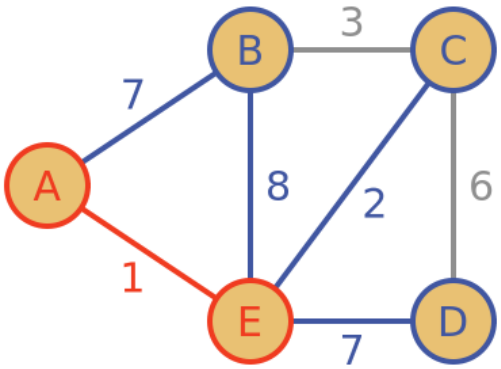
Крок 2. На цьому кроці ми перевіряємо всі невідвідані вершини та обираємо серед них вершину з найменшою поточною відстанню. В нашому випадку це буде вершина E з поточною відстанню 1. Тепер ми точно знаємо, що найкоротший шлях з вершини A до вершини E це 1. Як ми це дізналися, адже до вершини E ведуть ще три шляхи? Оскільки ми обрали вершину з найкоротшим

поточним шляхом, це означає, що шляхи які будуть проходити через всі інші сусідні до A вершини, будуть більшими, а отже, цей шлях найкоротший.

Оскільки ми знайшли найкоротшу відстань до вершини E, то позначаємо її як відвідану і розглядаємо її суміжні невідвідані вершини: B, C і D. Для кожної суміжної вершини ми розраховуємо відстань до неї від вершини A через вершину E (рис. 7.4).

Для вершини B відстань AEB буде 1+8=9. Ця відстань більша, ніж поточна відстань 7, а тому залишаємо її незмінною.

Поточна відстань до вершини C нескінченність, а отже ми замінюємо її на довжину шляху AEC, яка є 1+2=3. Аналогічним чином замінюємо відстань до вершини D на довжину шляху AED, яка є 1+7=8.



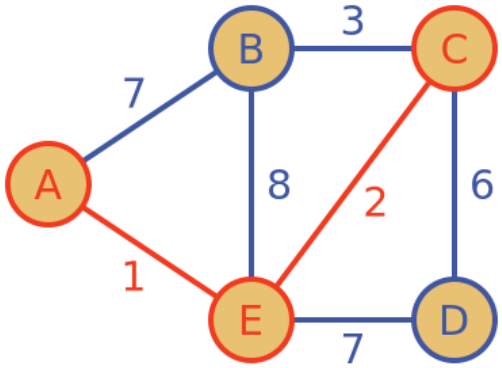
	Vis	Dist	Prev
A	1	0	-
B	0	7	A
C	0	3	E
D	0	8	E
E	1	1	A

Рис. 7.4. Стан алгоритму після другого кроку

Крок 3. На цьому кроці ми знову обираємо вершину з найкоротшим шляхом від початкової вершини серед тих вершин, які ще не відвідані. В нас три невідвідані вершини: B, C і D. Найкоротшим є шлях до вершини C, а отже ми позначаємо цю вершину як відвідану і розглядаємо шляхи через неї до її суміжних невідваних вершин, якими є B і D.

Поточний шлях з вершини A до вершини B через вершину C буде 3+3=6. Ця відстань менша, ніж прямий шлях до вершини A до вершини B, який зараз становить 7. Отже ми оновлюємо шлях до вершини B та ставимо попередником вершину C.

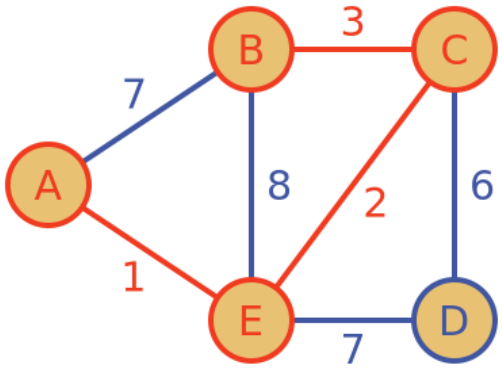
Тепер розглядаємо шлях до вершини D через вершину C. Цей шлях буде 3+6=9 і він довший, ніж попередній шлях через вершину E, який був 8. Таким чином шлях до вершини D залишається незмінним (рис. 7.5).



	Vis	Dist	Prev
A	1	0	-
B	0	6	C
C	1	3	E
D	0	8	E
E	1	1	A

Рис. 7.5. Стан алгоритму після третього кроку

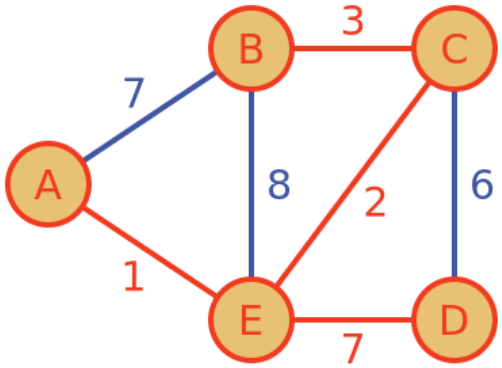
Крок 4. На четвертому кроці в нас залишилися дві невідвідані вершини: B і D. Поточний найкоротший шлях до вершини B, а тому позначаємо її відвіданою та розглядаємо її суміжні вершини. Вершина B не має суміжних невідвіданих вершин, а тому ми завершуємо цей крок (рис. 7.6).



	Vis	Dist	Prev
A	1	0	-
B	1	6	C
C	1	3	E
D	0	8	E
E	1	1	A

Рис. 7.6. Стан алгоритму після четвертого кроку

Крок 5. В нас залишилася єдина невідвідана вершина D. Тому ми позначаємо її відвіданою з найменшою відстанню 8 та завершуємо алгоритм (рис. 7.7).



	Vis	Dist	Prev
A	1	0	-
B	1	6	C
C	1	3	E
D	1	8	E
E	1	1	A

Рис. 7.7. Стан алгоритму після завершення

В результаті роботи алгоритму Дейкстри, ми отримали граф мінімальних шляхів від початкової вершини A до будь-якої іншої вершини в графі (рис. 7.8).

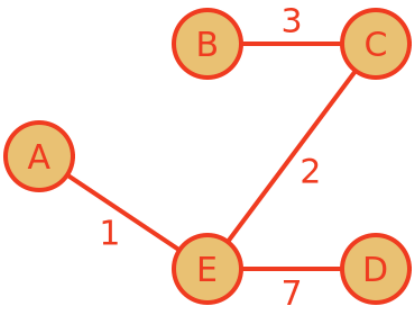


Рис. 7.8. Граф найкоротших шляхів від вершини A

Зауважте, що граф маршрутів правильний лише для стартової вершини A. Для будь-якої іншої стартової вершини, цей граф може бути неправильним. Наприклад, якщо взяти за стартову вершину C, то найкоротший шлях до вершини D, буде CED з відстанню 9, що насправді є неправильним, адже прямий шлях CD в початковому графі має вагу 6.

7.1.3. Приклад використання черги з пріоритетами

Розглянемо покрокове виконання алгоритму Дейкстри з використанням черги з пріоритетами. Черга з пріоритетами автоматично сортує елементи, що додаються в чергу, таким чином, щоб найменший завжди був на початку черги, а найбільший — в кінці.

В якості прикладу розглянемо орієнтований граф. На покрокових рисунках використовуються наступні позначення:

- червоні вершини і ребра — ті, що оброблюються зараз;
- червоні числа — позначають оновлення найкоротшої відстані;
- червоні кортежі — позначають нові пари (відстань, вершина) додані до черги;
- сині кортежі — позначають ті пари, що видаляються з черги.

Крок 1. На першому кроці виставляємо шлях до початкової вершини A в нуль. Відстані до всіх інших вершин виставляються в нескінченність. Додаємо кортеж (0, A) в чергу (рис. 7.9).

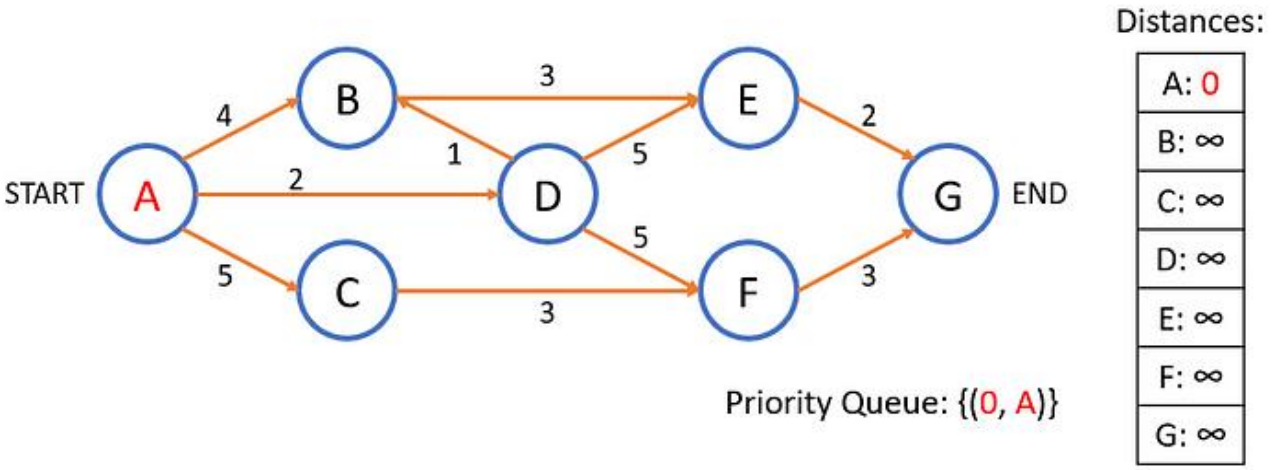


Рис. 7.9. Граф і черга після першого кроку

Крок 2. На другому кроці видаляємо з черги кортеж (A, 0), який знаходиться на початку. Розглядаємо суміжні до A вершини та оновлюємо відстані до них якщо потрібно. Додаємо в чергу кортежі: (4, B), (2, D) та (5, C).

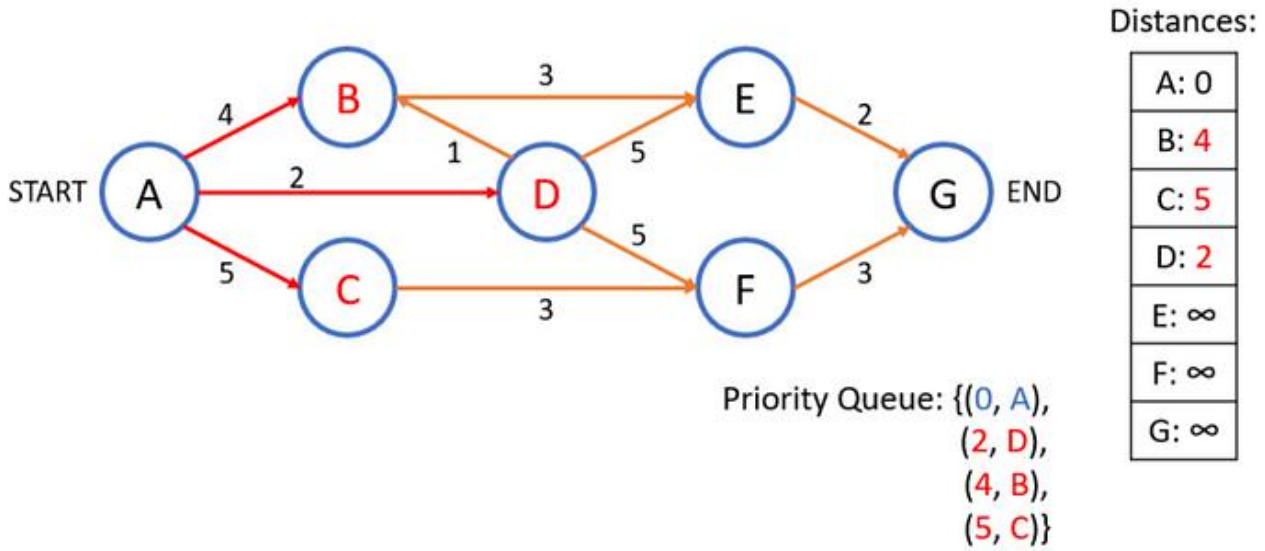


Рис. 7.10. Граф і черга після другого кроку

Крок 3. На третьому кроці розглядаємо вершину D. Видаляємо кортеж (2, D) з черги і оновлюємо відстані до суміжних вершин, якщо це необхідно. Додаємо кортежі (3, B), (7, E) та (7, F) до черги (рис. 7.11).

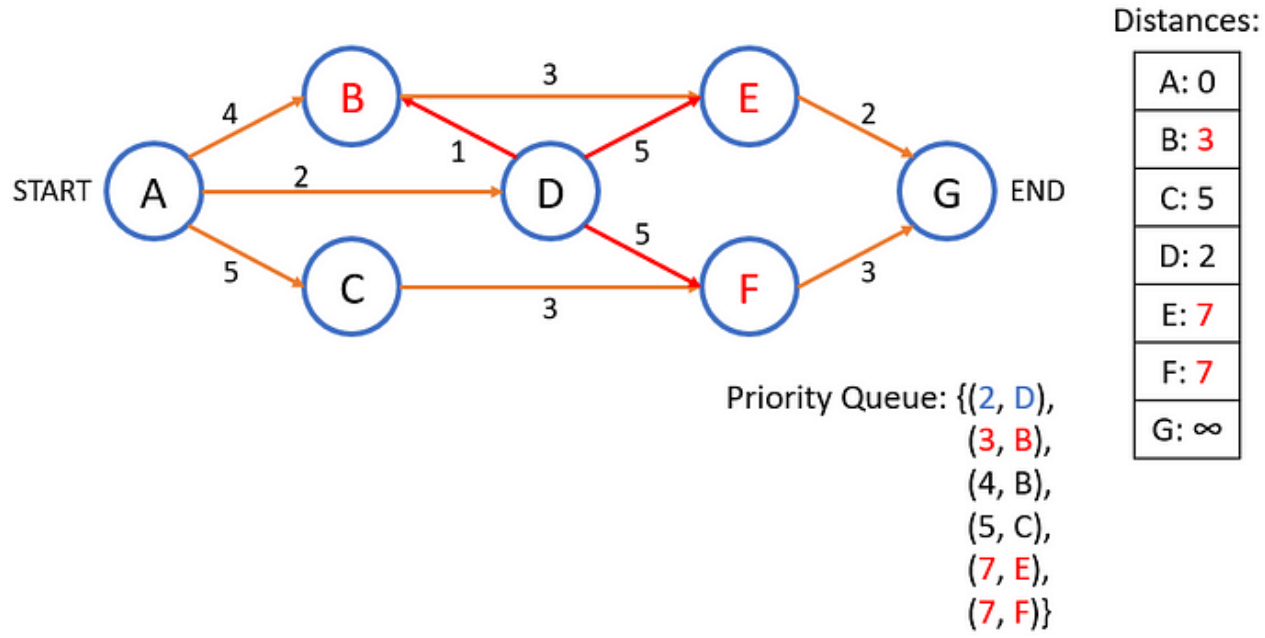


Рис. 7.11. Граф і черга після третього кроку

Крок 4. На четвертому кроці розглядаємо вершину B. Видаляємо кортеж (3, B) з черги та додаємо кортеж (6, E) до неї. Оновлюємо поточну відстань до вершини E (рис. 7.12).

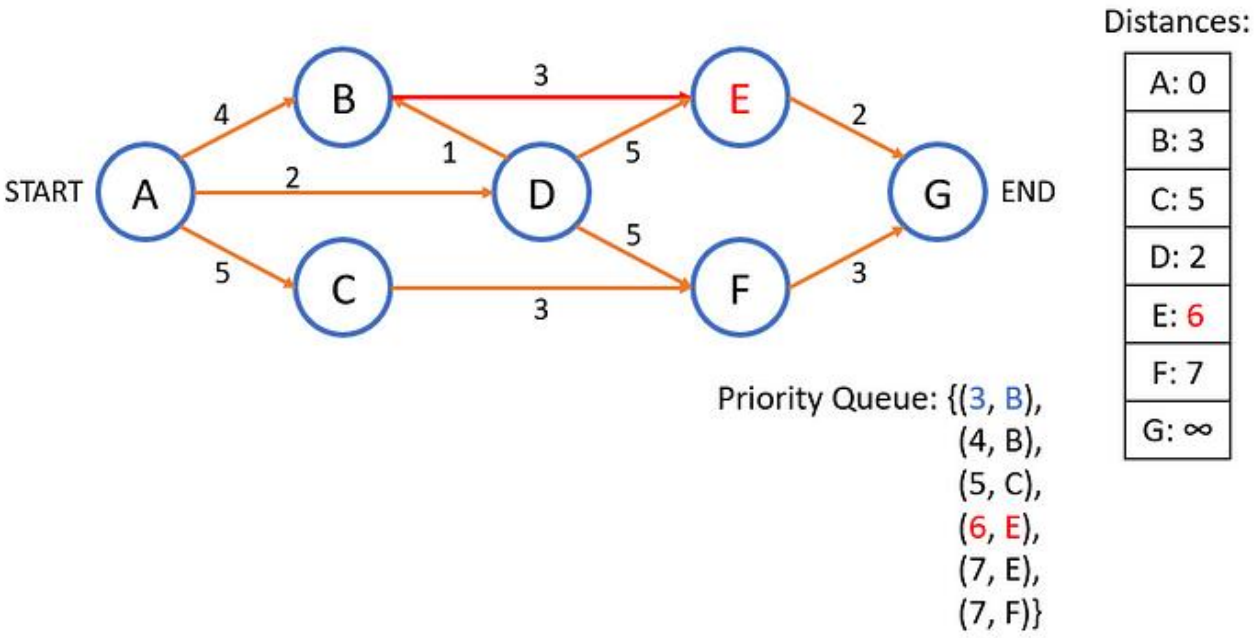


Рис. 7.12. Граф і черга після четвертого кроку

Крок 5. Цей крок трохи відрізняється від попередніх. Розглядаємо кортеж (4, B), який знаходиться на початку черги. Поточна відстань до вершини B — 3, що є меншою, ніж в кортежі, а отже нам не потрібно розглядати цей кортеж. Тому ми просто його видаляємо і переходимо до наступного кортежу (5, C). З вершини C є лише один шлях до вершини F, а тому ми додаємо його до черги. Оскільки нова відстань до вершини F більша, за поточну, то вона не оновлюється (рис. 7.13).

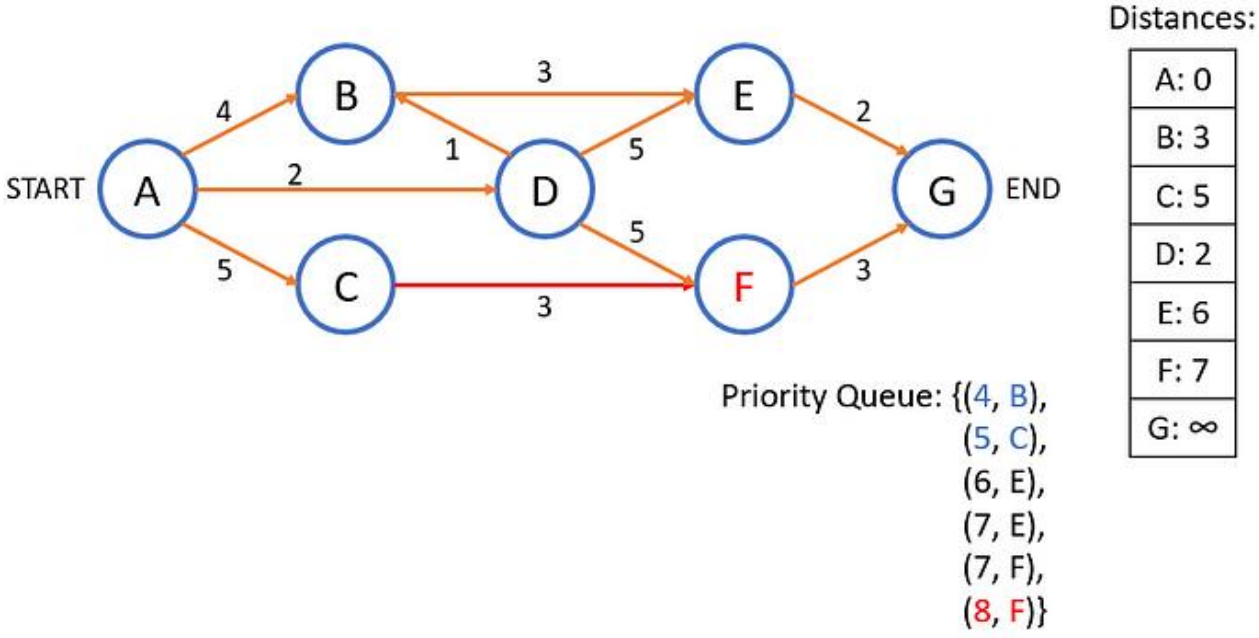


Рис. 7.13. Граф і черга після п'ятого кроку

Крок 6. На цьому кроці розглядається кортеж (6, E). Додаємо новий кортеж (8, G) до черги (рис. 7.14).

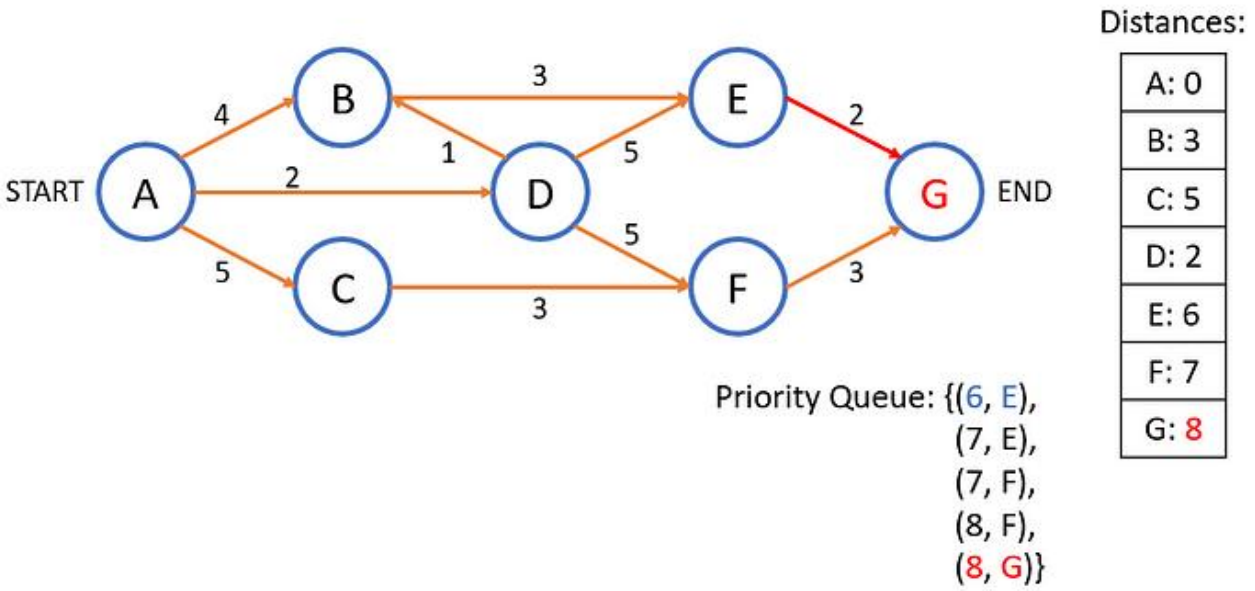


Рис. 7.14. Граф і черга після шостого кроку

Крок 7. На цьому кроці повторюється ситуація, яка була на кроці 5, тільки тепер з вершиною E. Пропускаємо кортеж (7, E) і переходимо до кортежу (7, F). В результаті додаємо кортеж (10, G) до черги (рис. 7.15).

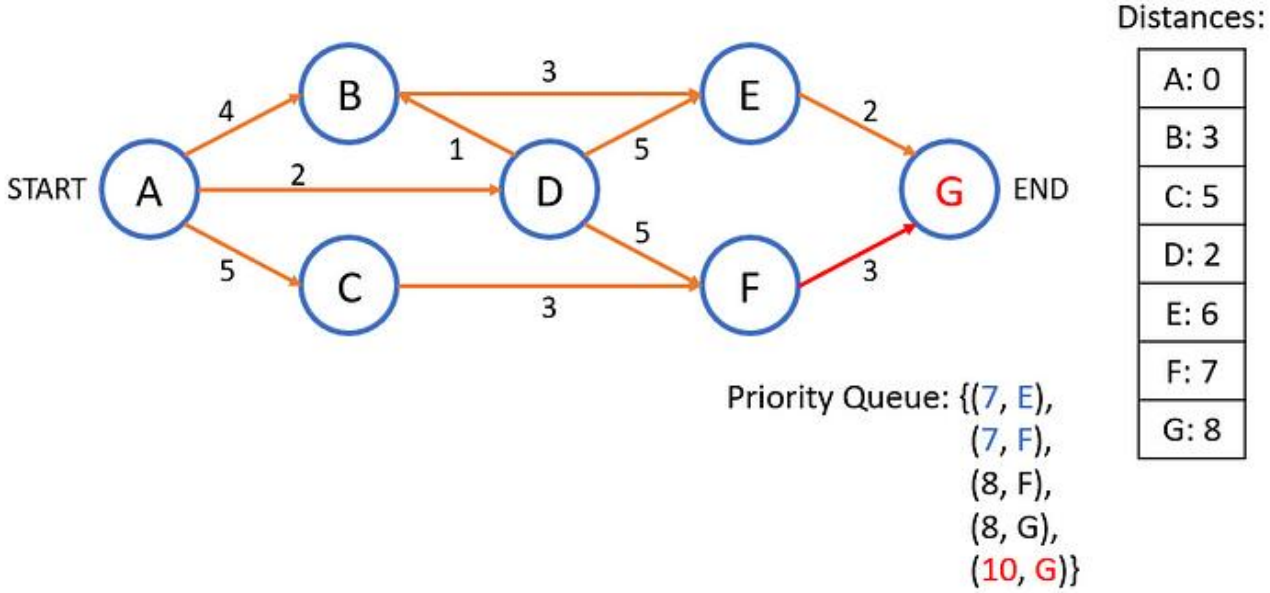


Рис. 7.15. Граф і черга після сьомого кроку

Крок 8. Жоден з кортежів не містить меншої відстані за поточні в таблиці відстаней, а тому алгоритм завершує свою роботу (рис. 7.16).

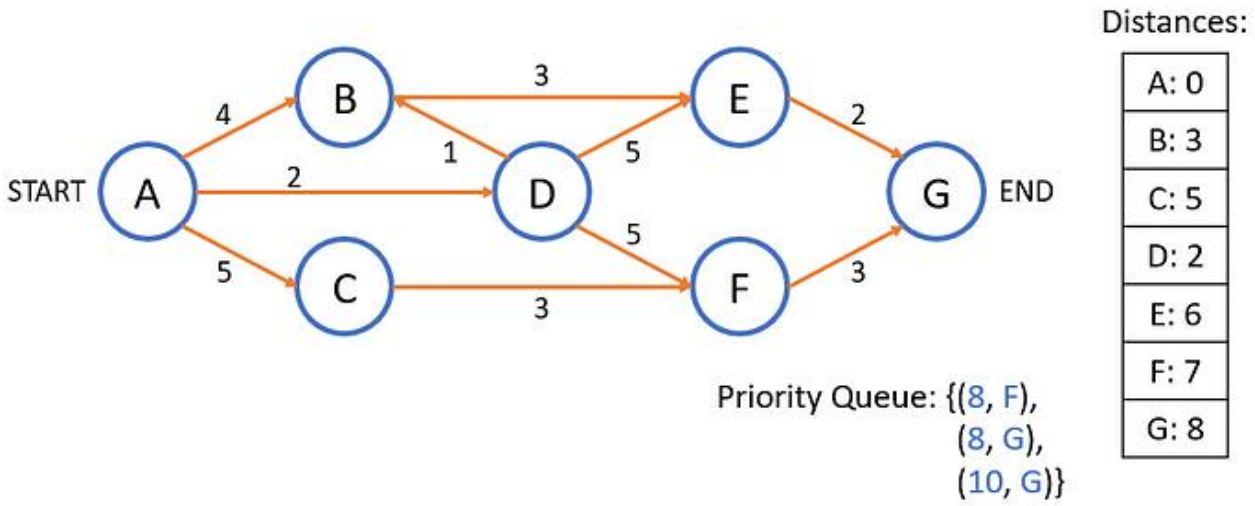


Рис. 7.16. Граф і черга після восьмого кроку

Використання черги з пріоритетами значно спрощує та пришвидшує реалізацію алгоритму Дейкстри. Головне пам'ятати, що розмір черги повинен дорівнювати кількості ребер графа, а не кількості вершин.

7.2. Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда був незалежно винайдений двома американськими вченими: Фордом у 1956 році та опублікований Беллманом у 1958 році. Алгоритм так само, як і алгоритм Дейкстри, розв'язує задачу пошуку найкоротшого шляху у графі але допускає наявність від'ємних ребер.

Алгоритм не зможе знайти найкоротший шлях в графі, якщо в ньому існує негативний цикл. Саме тому важливою рисою цього алгоритму є можливість знаходити негативні цикли.

7.2.1. Формальний опис та псевдокод

Алгоритм Беллмана-Форда працює ітеративно і знаходить найкоротший шлях за $V-1$ ітерацій. На останній додатковій ітерації перевіряється наявність негативних циклів у графі. Отже загалом кількість ітерацій дорівнює кількості вершин графа.

На кожній ітерації алгоритм намагається покращити відстань від стартової вершини до всіх інших вершин графа шляхом релаксації кожного ребра. Власне релаксація це спроба поліпшити поточну відстань від початкової вершини до поточної шляхом її заміни на відстань до попередньої вершини плюс вага ребра.

Нижче подано узагальнений псевдокод алгоритму.

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
    distance[S] <- 0
    for each vertex V in G
```

```
        for each edge(U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    for each edge(U,V) in G
        if distance[U] + edge_weight(U, V) < distance[V]
            Error: Negative Cycle Exists
    return distance[], previous[]
```

Як випливає з опису та псевдокоду, обчислювальна складність алгоритму Беллмана-Форма складає O(VE). Це значно гірше, ніж в алгоритму Дейкстри, проте, алгоритм Беллмана-Форда є більш універсальним, оскільки дозволяє наявність негативних ребер.

7.2.2. Приклад виконання алгоритму

Розглянемо покроковий приклад виконання алгоритму Беллмана-Форда на прикладі орієнтованого зваженого графа, що має негативні ребра (рис. 7.17).

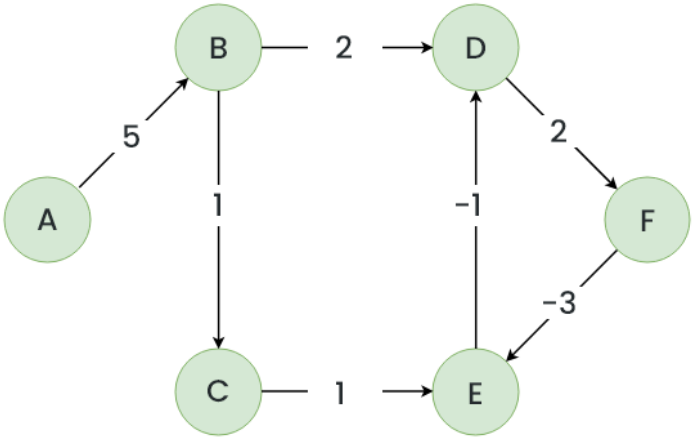


Рис. 7.17. Граф з негативними ребрами

Початок. Ініціалізуємо масив `Dist[]` для збереження відстаней від початкової до кожної вершини графа. Відстань до стартової вершини нуль, а до всіх інших — нескінченність (рис. 7.18).

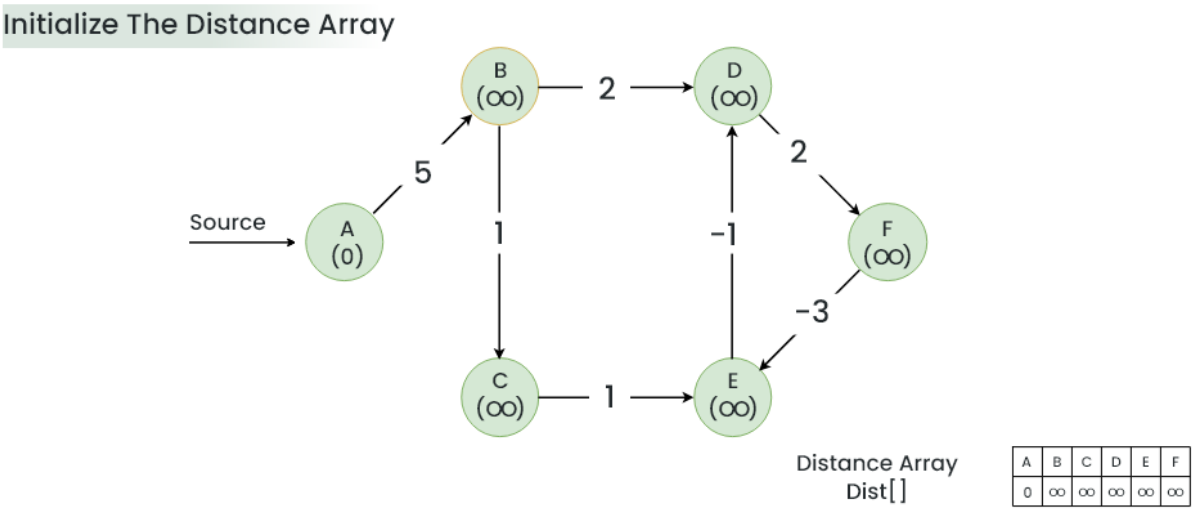


Рис. 7.18. Граф і таблиця відстаней після ініціалізації

Крок 1. Виконуємо першу релаксацію ребер. Під час цієї релаксації можемо задіяти лише одне ребро. Поточна відстань до вершини B більша, ніж відстань до вершини A плюс вага ребра AB (∞>0+5), а тому ми оновлюємо відстань до вершини B: `Dist[B]=5` (рис. 7.19).

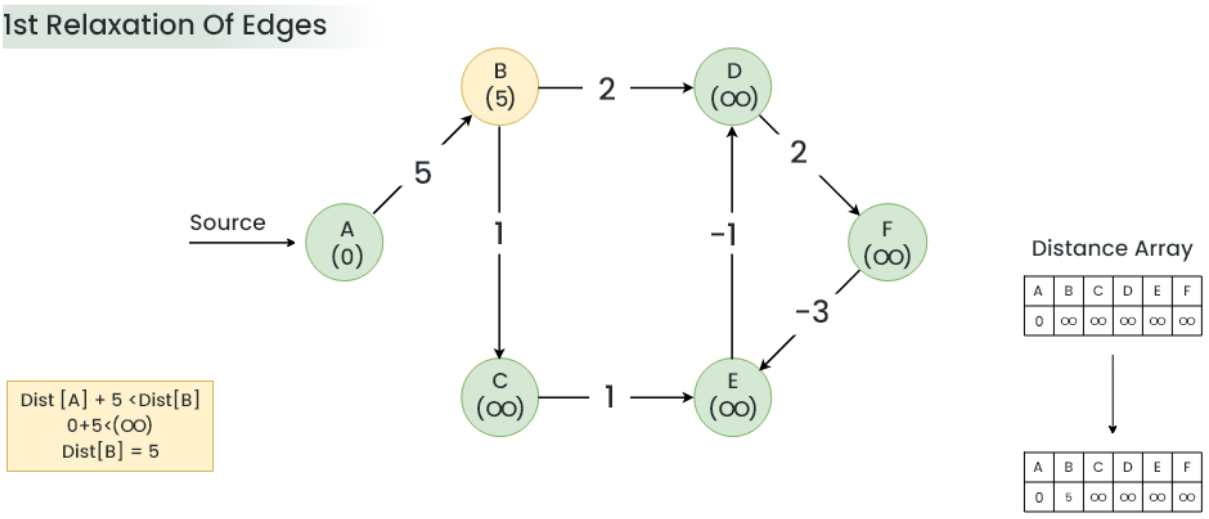


Рис. 7.19. Граф і таблиця відстаней після першої релаксації

Крок 2. Виконуємо другу релаксацію ребер. Під час цієї релаксації можемо задіяти вже два ребра (рис. 7.20):

- поточна відстань до вершини D більша за шлях через вершину B (∞>5+2), а тому оновлюємо відстань до неї — `Dist[D]=7`;
- поточна відстань до вершини C більша за шлях через вершину B (∞>5+1), а тому оновлюємо відстань до неї — `Dist[C]=6`.

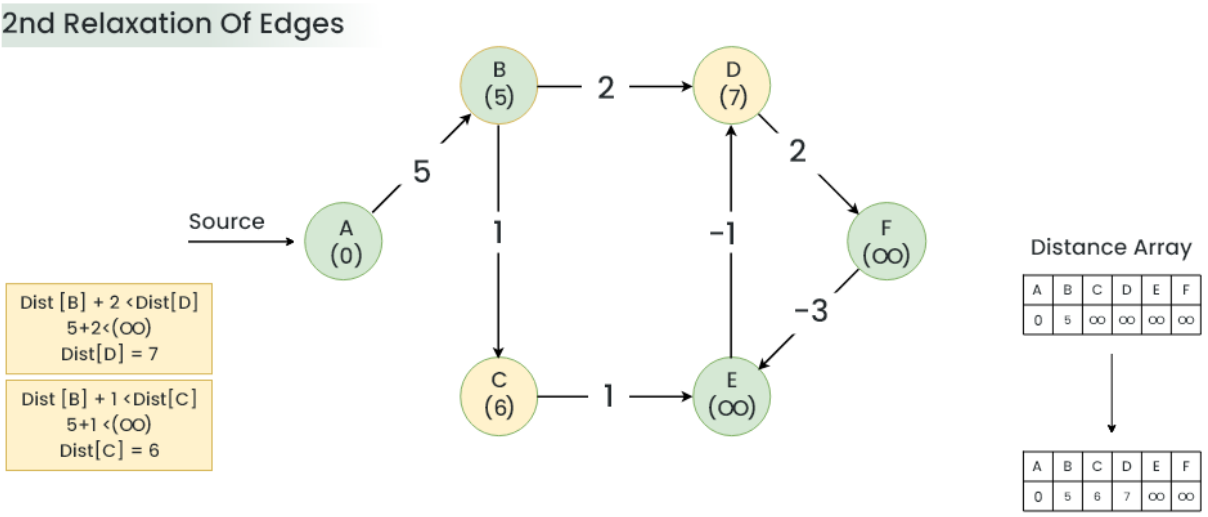


Рис. 7.20. Граф і таблиця відстаней після другої релаксації

Крок 3. Виконуємо третю релаксацію ребер. Під час цієї релаксації також можемо задіяти два ребра (рис. 7.21):

- поточна відстань до вершини F більша за шлях через вершину D (∞>7+2), а тому оновлюємо відстань до неї — `Dist[F]=9`;
- поточна відстань до вершини E більша за шлях через вершину C (∞>6+1), а тому оновлюємо відстань до неї — `Dist[E]=7`.

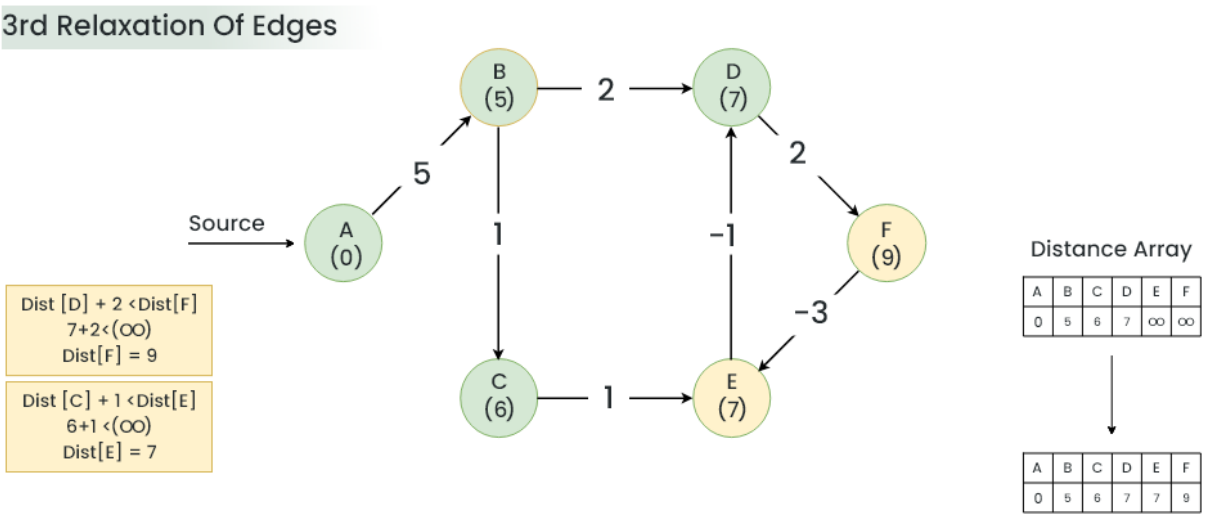


Рис. 7.21. Граф і таблиця відстаней після третьої релаксації

Крок 4. Виконуємо четверту релаксацію ребер. Під час цієї релаксації також можемо задіяти два ребра (рис. 7.22):

- поточна відстань до вершини D більша за шлях через вершину E (7>7-1), а тому оновлюємо відстань до неї — `Dist[D]=6`;
- поточна відстань до вершини E більша за шлях через вершину F (9>9-3), а тому оновлюємо відстань до неї — `Dist[E]=6`.

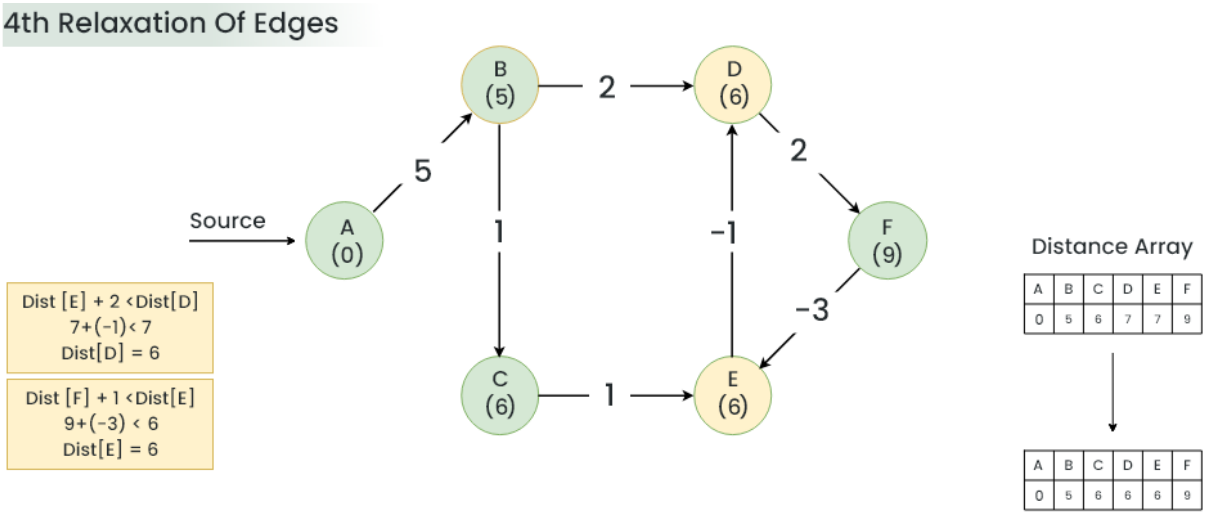


Рис. 7.22. Граф і таблиця відстаней після четвертої релаксації

Крок 5. Виконуємо п'яту релаксацію ребер. Під час цієї релаксації також можемо задіяти два ребра (рис. 7.23):

- поточна відстань до вершини F більша за шлях через вершину D (9>6+2), а тому оновлюємо відстань до неї — `Dist[F]=8`;
- поточна відстань до вершини D більша за шлях через вершину E (6>6-1), а тому оновлюємо відстань до неї — `Dist[D]=5`.

Оскільки в графі шість вершин, то після п'ятої релаксації ми отримали найкоротші відстані від початкової вершини A до всіх інших вершин графа.

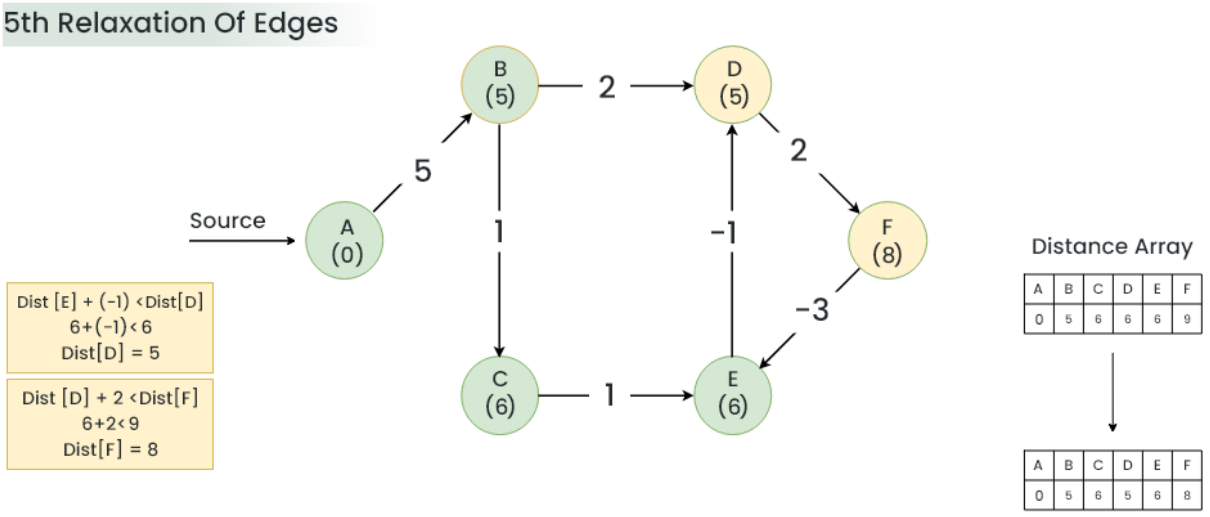


Рис. 7.23. Граф і таблиця відстаней після п'ятої релаксації

Крок 6. Виконуємо додаткову релаксацію, щоб перевірити наявність негативних циклів у графі. Під час цієї релаксації ми можемо задіяти два ребра (рис. 7.24):

- поточна відстань до вершини E більша за шлях через вершину F (6>8-3), а тому оновлюємо відстань до неї — `Dist[E]=5`;
- поточна відстань до вершини F більша за шлях через вершину D (8>5+2), а тому оновлюємо відстань до неї — `Dist[F]=7`.

Оскільки протягом додаткової релаксації ми зафіксували зміни в масиві відстаней, це означає присутність негативного циклу в графі.

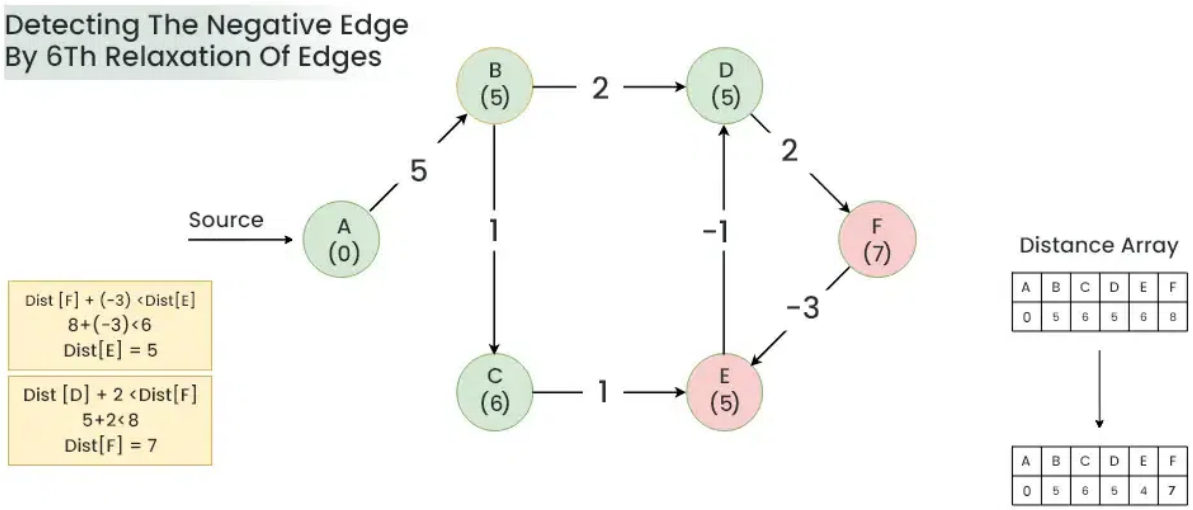


Рис. 7.24. Граф і таблиця відстаней після додаткової релаксації

Як і для алгоритму Дейкстри, найкоротші маршрути правильні лише для початкової вершини A. Для знаходження найкоротших шляхів для іншої початкової вершини алгоритм потрібно запускати заново з іншою ініціалізацією початкових відстаней.

7.3. Алгоритм Флойда-Воршелла

Алгоритм Флойда-Воршелла призначений для пошуку найкоротших шляхів між всіма вершинами графа. Був незалежно відкритий американськими вченими Робертом Флойдом та Стівеном Воршелом у 1962 році. Проте, це практично той самий алгоритм, що був опублікований французьким вченим Бернаром Руа в 1959 році. Саме тому існують різні варіації назви цього алгоритму.

Як і алгоритм Беллмана-Форда, алгоритм Флойда-Воршелла працює на орієнтованих і неорієнтованих зважених графах, що можуть мати від'ємні ваги ребер. Алгоритм також має обмеження на наявність в графі негативних циклів — їх бути не повинно, інакше вирішення знайдене не буде.

7.3.1. Формальний опис та псевдокод

Нехай в нас є граф G з кількістю вершин V, що пронумеровані від 1 до N. Необхідно розрахувати матрицю найкоротших шляхів SPM (Shortest Path Matrix) розмірністю VxV, де кожен елемент матриці SPM[i][j] це найкоротший шлях між вершинами `i` та `j`.

Очевидно, що найкоротший шлях між `i` та `j` буде містити деяку кількість k проміжних вузлів. Основна ідея алгоритму в тому, щоб почергово перевірити всі вершини як проміжні вузли на шляху між початковою та кінцевою вершинами і таким чином знайти найкоротший шлях.

Алгоритм використовує принцип динамічного програмування, який ми будемо розглядати в десятому розділі. Його графічна інтерпретація подана на рис. 7.25.

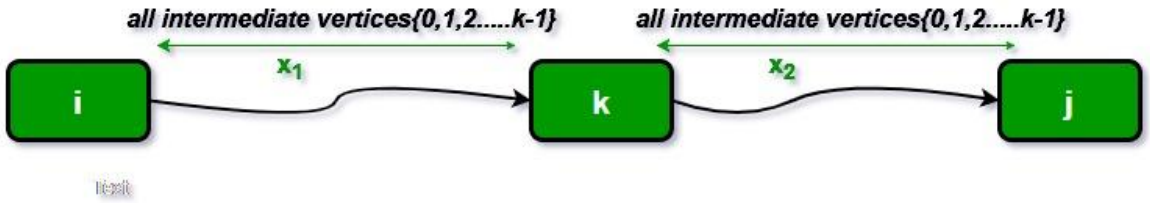


Рис. 7.25. Принцип оптимальної підструктури алгоритму

Формальний опис алгоритму наступний.

1. Створити матрицю суміжності A^0 з вагами всіх ребер, що є в графі. Якщо між двома вершинами немає ребра, то позначити шлях між ними як нескінченність.
2. Створити копію початкової матриці A^1 і зафіксувати перший рядок та перший стовпчик з оригінальної матриці. Для всіх елементів матриці $A^1[i][j]$, якщо $A^0[i][j] > A^0[i][k] + A^0[k][j]$, то замінити значення $A^1[i][j]$ на значення $A^0[i][k] + A^0[k][j]$, в інакшому випадку значення залишається незмінним. Тут k=1, тобто перша вершина розглядається як проміжний вузол.
3. Повторити крок 2 для всіх вершин графа, змінюючи значення k почергово для кожної вершини: {1,2,3…N}.
4. Коли k=N, це означає, що отримана фінальна матриця, що містить найкоротші шляхи між всіма парами вершин в графі.

Нижче подано узагальнений псевдокод алгоритму.

```
n = Кількість вершин графа
A = Матриця розмірністю n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i, j] = min (A^(k-1)[i, j], A^(k-1)[i, k] + A^(k-1)[k, j])
return A
```

Обчислювальна складність такого алгоритму буде $O(V^3)$, оскільки ми маємо потрійний вкладений цикл по всім вершинам графа.

7.3.2. Приклад виконання алгоритму

Розглянемо покрокове виконання алгоритму Флойда-Воршелла на прикладі орієнтованого зваженого графа (рис. 7.26).

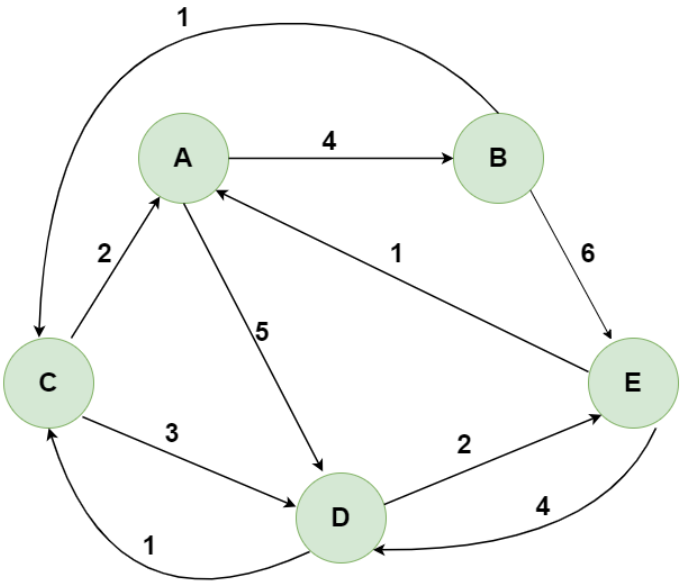


Рис. 7.26. Орієнтований зважений граф

Крок 1. Ініціалізуємо матрицю відстаней `Dist` вагами ребер графа. Якщо не існує ребра між вершинами, то ставимо нескінченну відстань (рис. 7.27).

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Рис. 7.27. Матриця відстаней після ініціалізації

Крок 2. Розглядаємо вершину A, як проміжну вершину шляху. Перераховуємо відстані між всіма парами вершин за формулою: `Dist[i][j]=min(Dist[i][j], Dist[i][A]+Dist[A][j])` (рис. 7.28).

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

→

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Рис. 7.28. Матриця відстаней після розгляду вершини A

Крок 3. Розглядаємо вершину B, як проміжну вершину шляху. Перераховуємо відстані між всіма парами вершин за формулою: `Dist[i][j]=min(Dist[i][j], Dist[i][B]+Dist[B][j])` (рис. 7.29).

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

Рис. 7.29. Матриця відстаней після розгляду вершини B

Крок 4. Розглядаємо вершину C, як проміжну вершину шляху. Перераховуємо відстані між всіма парами вершин за формулою: `Dist[i][j]=min(Dist[i][j], Dist[i][C]+Dist[C][j])` (рис. 7.30).

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

Рис. 7.30. Матриця відстаней після розгляду вершини C

Крок 5. Розглядаємо вершину D, як проміжну вершину шляху. Перераховуємо відстані між всіма парами вершин за формулою: `Dist[i][j]=min(Dist[i][j], Dist[i][D]+Dist[D][j])` (рис. 7.31).

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Рис. 7.31. Матриця відстаней після розгляду вершини D

Крок 6. Розглядаємо вершину E, як проміжну вершину шляху. Перераховуємо відстані між всіма парами вершин за формулою: `Dist[i][j]=min(Dist[i][j], Dist[i][E]+Dist[E][j])` (рис. 7.32).

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Рис. 7.32. Матриця відстаней після розгляду вершини E

Оскільки всі вершини були розглянуті як проміжні вузли, то фінальна матриця містить найкоротші шляхи між кожною парою вузлів графа.

7.4. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач для знаходження найкоротших шляхів у графі.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням одного з розглянутих алгоритмів.

Середній рівень складності

1. Find the City with the Smallest Number of Neighbors at a Threshold Distance https://leetcode.com/problems/find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance.
2. Path with Maximum Probability https://leetcode.com/problems/path-with-maximum-probability.
3. Number of Restricted Paths From First to Last Node https://leetcode.com/problems/number-of-restricted-paths-from-first-to-last-node.

Високий рівень складності

1. Minimum Time to Visit a Cell in a Grid https://leetcode.com/problems/minimum-time-to-visit-a-cell-in-a-grid.
2. Modify Graph Edge Weights https://leetcode.com/problems/modify-graph-edge-weights.
3. Find Edges in Shortest Paths https://leetcode.com/problems/find-edges-in-shortest-paths.

Запитання для самоконтролю

1. Які алгоритми для пошуку найкоротшого шляху в графі ви знаєте?
2. Які обмеження має алгоритм Дейкстри?
3. Яка обчислювальна складність алгоритму Дейкстри?
4. Який алгоритм обходу графа лежить в основі алгоритму Дейкстри?
5. Яку перевагу має алгоритм Беллмана-Форда над алгоритмом Дейкстри?
6. Які обмеження має алгоритм Беллмана-Форда?
7. Яка обчислювальна складність алгоритму Беллмана-Форда?
8. Яка основна відмінність алгоритму Флойда-Воршелла від інших алгоритмів пошуку найкоротшого шляху в графі?
9. Яка обчислювальна складність алгоритму Флойда-Воршелла?
10. Який з розглянутих алгоритмів дозволяє виявити негативні цикли в графі?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Задача_про_найкоротший_шлях
2. https://uk.wikipedia.org/wiki/Алгоритм_Дейкстри
3. https://en.wikipedia.org/wiki/Dijkstra's_algorithm
4. https://graphicmaths.com/computer-science/graph-theory/dijkstras-algorithm
5. https://medium.com/@alejandro.itoaramendia/a-guide-to-dijkstras-algorithm-all-you-need-99635dcd6d94
6. https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm
7. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction
8. https://uk.wikipedia.org/wiki/Алгоритм_Беллмана_—_Форда
9. https://www.mathros.net.ua/rozvjazok-zadachi-pro-najkorotshyj-shljah-vykorystovujuchy-algorytm-bellmana-forda.html
10. https://en.wikipedia.org/wiki/Bellman–Ford_algorithm
11. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23
12. https://www.programiz.com/dsa/bellman-ford-algorithm
13. https://www.baeldung.com/cs/bellman-ford
14. https://uk.wikipedia.org/wiki/Алгоритм_Флойда_—_Воршелла
15. https://en.wikipedia.org/wiki/Floyd–Warshall_algorithm
16. https://www.programiz.com/dsa/floyd-warshall-algorithm
17. https://www.tutorialspoint.com/data_structures_algorithms/floyd_warshall_algorithm.htm
18. https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16

8. Числові послідовності та перестановки

В цьому розділі ми згадаємо основні числові послідовності та способи їх подання в графічному вигляді. Також буде розглянута задача генерації перестановок та способи її вирішення.

8.1. Визначення послідовності

Послідовність — функція визначена на множині натуральних чисел яка набуває значення на об'єктах довільної природи. Короткий запис послідовності має такий вигляд: $\{x_n\}$. Елементи x_1, x_2, … x_n називаються членами послідовності. Можна розглядати послідовність як впорядковану (занумеровану натуральними числами) множину її членів.

В залежності від типу елементів, послідовності поділяють на числові та функціональні. Ми детально будемо розглядати числові послідовності, оскільки задачі з перестановками використовують саме їх.

Послідовність може визначатись на скінченній підмножині натуральних чисел, тоді вона називається скінченною. Кількість членів послідовності називають довжиною послідовності. Скінченна послідовність на відміну від нескінченної має скінченну довжину. Також для скінченних послідовностей використовується інше позначення: $\{x_i\}^n$. У цьому випадку `i` — лічильник, а `n` — кількість елементів.

8.1.1. Числова послідовність

Числова́ послідовність — послідовність дійсних чисел, тобто відображення, яке кожному натуральному числу n ставить у відповідність дійсне число x_n.

У загальному випадку члени послідовності, як правило, позначають малими буквами з індексами внизу. Кожний індекс вказує порядковий номер члена послідовності.

Щоб задати послідовність, потрібно вказати спосіб, за допомогою якого можна знайти будь-який його член.

1. Послідовність можна задати описом знаходження її членів.
2. Скінченну послідовність можна задати переліком її членів.
3. Послідовність можна задати таблицею, у якій навпроти кожного члена послідовності вказують його порядковий номер.
4. Послідовність можна задати формулою, за якою можна знайти будь-який член послідовності, знаючи його номер.
5. Спочатку вказати перший або кілька перших членів послідовності, а потім — умову, за якою можна визначити будь-який член послідовності за попереднім. Такий спосіб задання послідовності називають рекурентним.

8.1.2. Типи послідовностей

В математиці існує декілька спеціальних послідовностей, що мають певні властивості. Розглянемо деякі з них.

Арифметична прогресія — це послідовність дійсних чисел, кожен член якої, починаючи з другого, утворюється додаванням до попереднього члена одного й того ж числа (рис. 8.1).

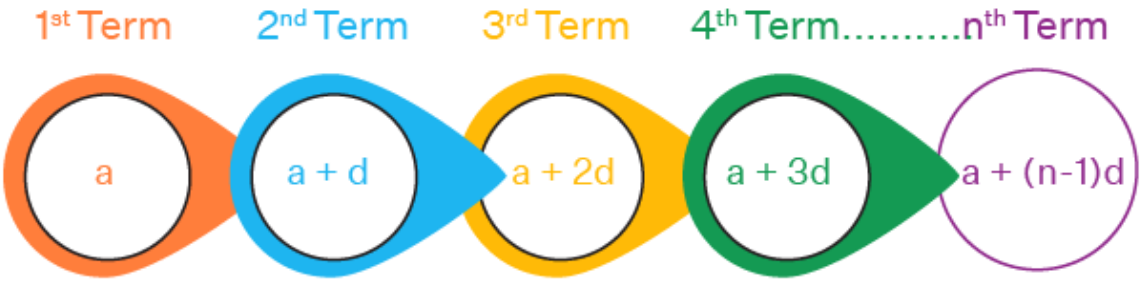


Рис. 8.1. Арифметична прогресія

Загальний вид арифметичної прогресії: a_1, a_1+d, a_1+2d, …, $a_1+(n-1)d$, … де a_1 — перший член прогресії, $d=a_{n+1}-a_n$ — різниця арифметичної прогресії.

Геометрична прогресія — послідовність чисел, перший член якої не дорівнює нулю, а відношення будь-якого елемента послідовності до попереднього є сталим числом, що називається знаменником прогресії та позначається символом r (рис. 8.2).

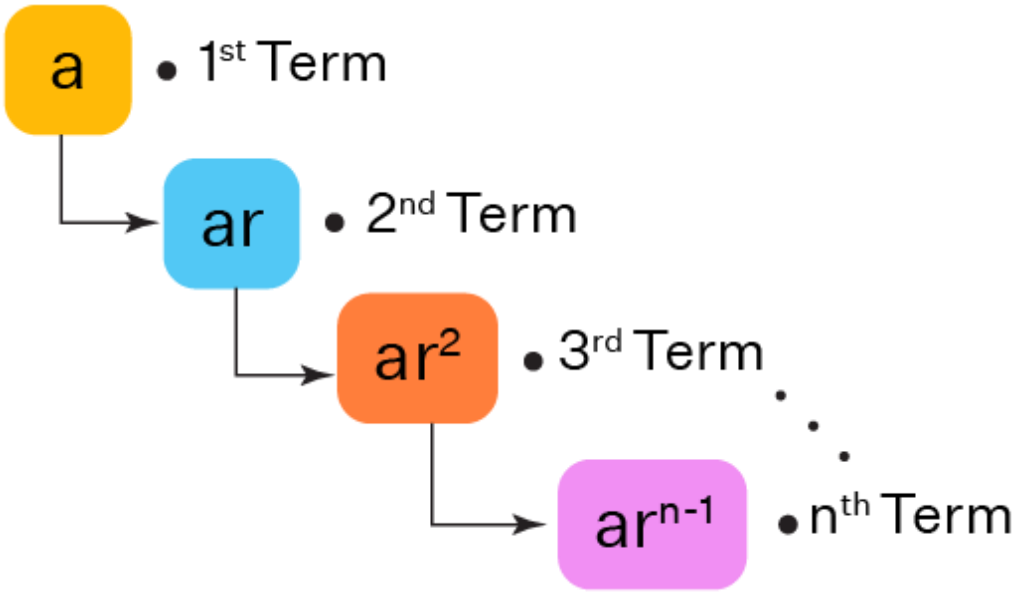


Рис. 8.2. Геометрична прогресія

Якщо знаменник прогресії дорівнює 1 (одиниці), то прогресія вважається стаціонарною. Знаменник геометричної прогресії не може дорівнювати нулю. Якщо модуль знаменника прогресії більше одиниці — прогресія зростаюча, якщо він менше одиниці — прогресія спадна. У випадку коли знаменник прогресії менше нуля — прогресія знакозмінна.

Фігурні числа — це числа, які можна подати у вигляді регулярних дискретних геометричних об'єктів (наприклад, множин кругів чи куль), які щільно виповнюють правильні геометричні фігури. Наприклад, трикутне число — це кількість кругів однакового діаметру з яких можна скласти правильний трикутник (рис. 8.3).

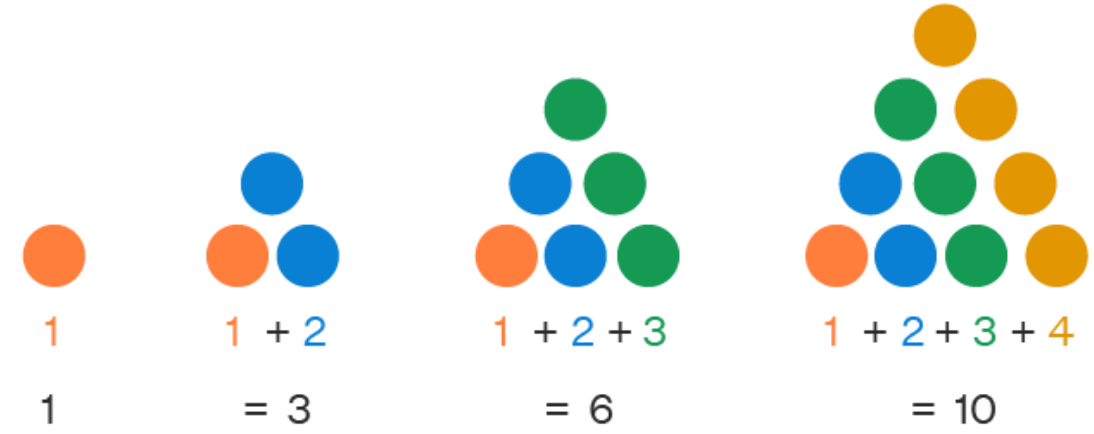


Рис. 8.3. Послідовність трикутних чисел

Аналогічно визначають квадратні (рис. 8.4), п'ятикутні та інші числа.

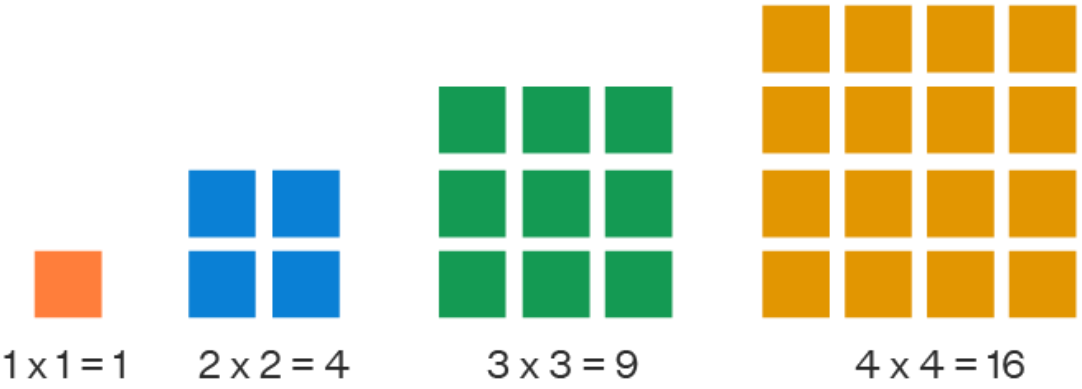


Рис. 8.4. Послідовність квадратних чисел

Назва конкретного виду фігурних чисел відображає назву відповідної геометричної фігури. Вважається, що від цих чисел пішов вираз «піднести число до квадрата чи куба» (рис. 8.5).

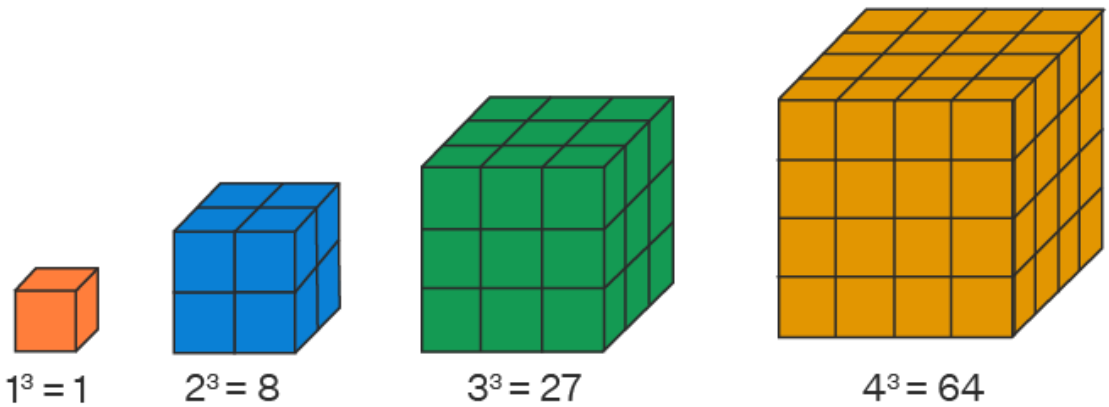


Рис. 8.5. Послідовність кубічних чисел

Послідовність Фібона́ччі, числа Фібона́ччі — у математиці числова послідовність F_n, задана рекурентним співвідношенням другого порядку: $F_{n+2}=F_n+F_{n+1}$. Простіше кажучи, перші два члени послідовності — одиниці, а кожний наступний — сума значень двох попередніх чисел. Іноді послідовність починають з нуля, тоді перші два члени це 0 і 1.

Ця послідовність відкрита італійським математиком Леонардо Пізанським на прізвисько Фібоначчі у XIII столітті під час розв'язання задачі про кількість пар кроликів на фермі (рис. 8.6).

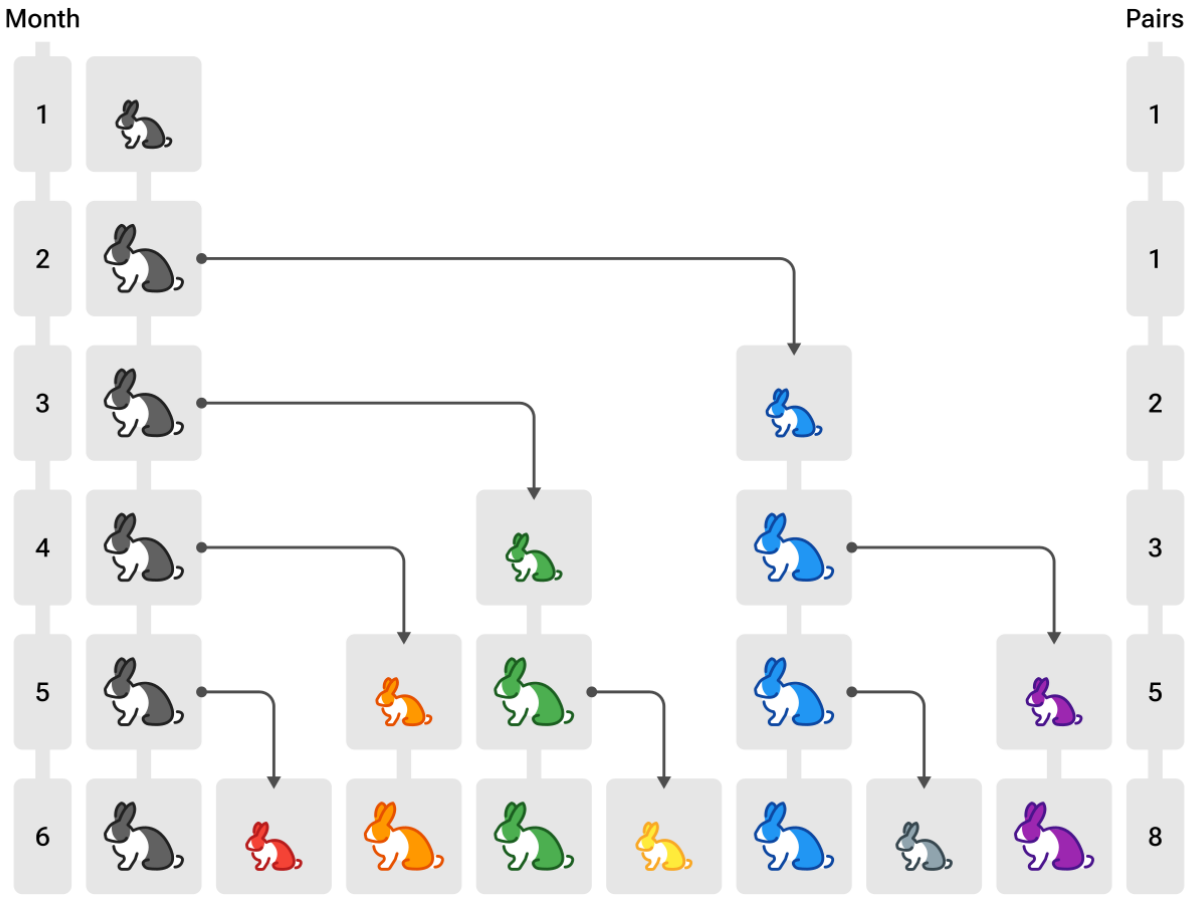


Рис. 8.6. Кількість пар кроликів відповідає послідовності Фібоначчі

Послідовність Фібоначчі можна записати у вигляді матричного рівняння:

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix}.$$

Якщо через A позначити матрицю $\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$, то отримаємо наступне:

$$\begin{pmatrix} F_{2n} \\ F_{2n+1} \end{pmatrix} = A^n \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Тобто для знаходження членів послідовності Фібоначчі з порядковими номерами 2n та 2n+1 необхідно піднести матрицю A до степені n.

8.2. Перестановки

Перестановкою скінченної множини X називається впорядкований набір без повторів із її елементів. Усього існує n! різних перестановок, де n — кількість елементів множини.

Задачі з генерацією перестановок зустрічаються в алгоритмах виявлення помилок і корекції. Нижче розглянемо основні алгоритми генерації перестановок.

8.2.1. Алгоритм генерації перестановок в лексикографічному порядку

Перестановки в лексикографічному порядку генеруються таким чином, що кожна наступна містить зростаючу послідовність елементів, що відмінна від попередньої. Наприклад для трьох перших натуральних чисел це будуть наступні перестановки: {1, 2, 3}; {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}.

Наївний алгоритм для створення такої послідовності полягає в генерації всіх можливих перестановок з подальшим їх сортуванням.

Значно ефективнішим буде алгоритм, що відразу генерує лексикографічно наступну перестановку. Формальний опис цього алгоритму поданий нижче.

1. Розглянемо масив цілих чисел A. Проглядаючи елементи з кінця масиву, знаходимо найбільший індекс `i`, що задовольняє умові `A[i]<A[i+1]`. Якщо такого немає, то завершуємо роботу.
2. Переглядаємо елементи з індексами, що більші за `i` та знаходимо максимальний індекс елемента `j`, що задовольняє умові `A[i]<A[j]`.
3. Міняємо місцями елементи `A[i]` та `A[j]`.
4. Інвертуємо порядок елементів починаючи з індексу `i+1` і до кінця масиву.

Описаний алгоритм генерує лексикографічно наступну перестановку за час `O(n)`. Відповідно, щоб згенерувати всі перестановки, потрібно `O(n*n!)` часу.

Розглянемо покроковий приклад алгоритму генерації лексикографічно наступної перестановки.

Крок 1. Шукаємо найбільший індекс `i`, такий що задовольняє умові `A[i]<A[i+1]` (рис. 8.7).

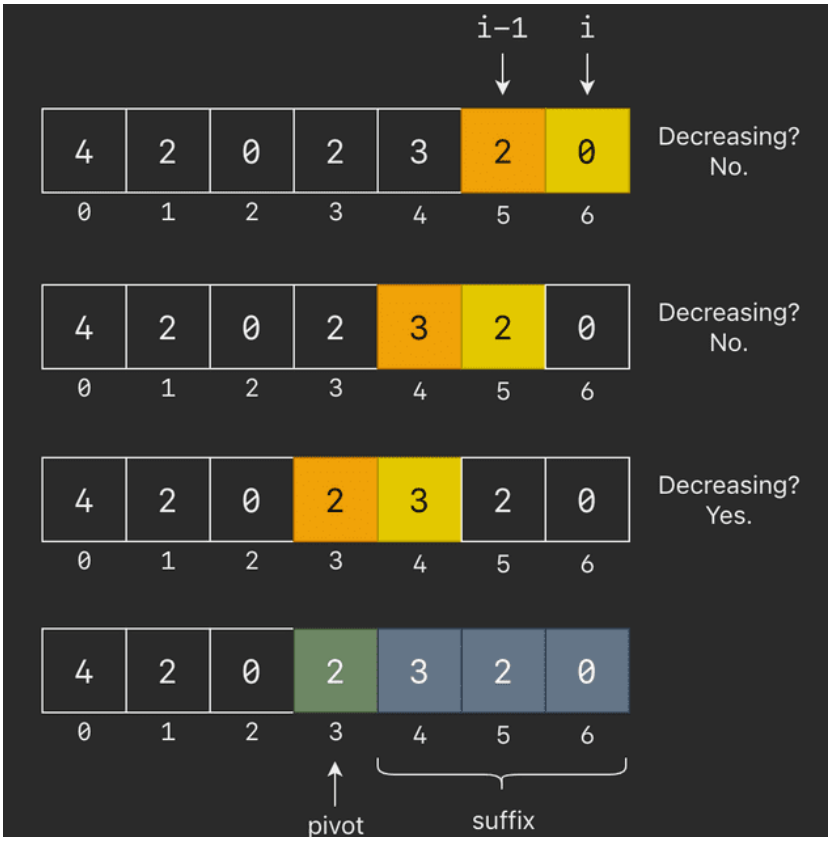


Рис. 8.7. Пошук початкового індексу `i`

Крок 2. Серед елементів, що мають індекси більші за `i`, шукаємо елемент з максимальним індексом `j` такий, що `A[i]<A[j]` (рис. 8.8).

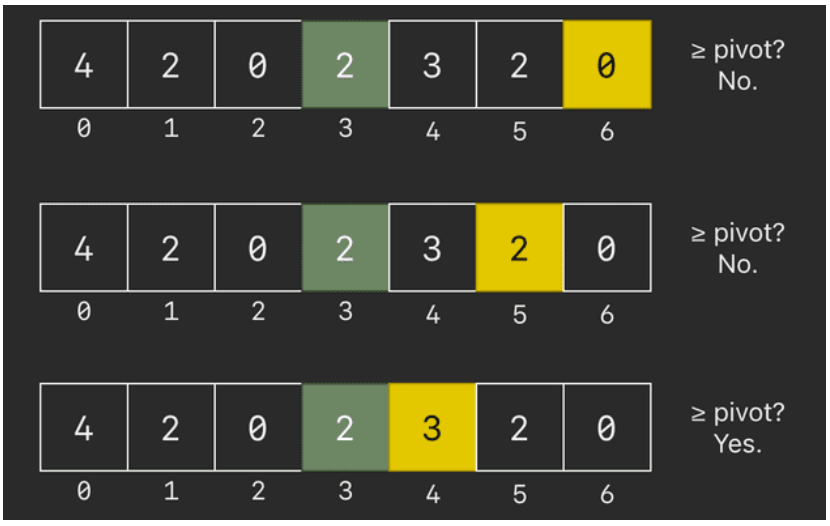


Рис. 8.8. Пошук наступного індексу `j`

Крок 3. Міняємо елементи з індексами `i` та `j` місцями (рис. 8.9).

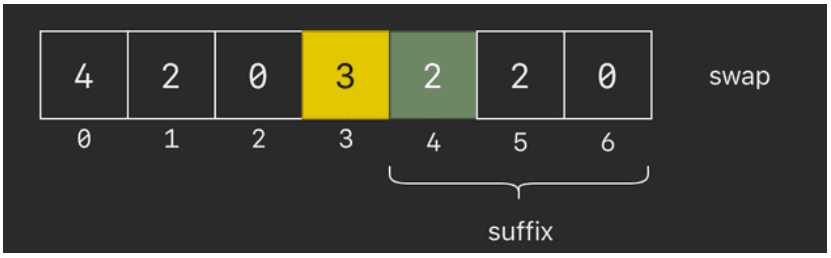


Рис. 8.9. Обмін місцями елементів `A[i]` та `A[j]`

Крок 4. Інвертуємо елементи суфікса масиву (рис. 8.10).

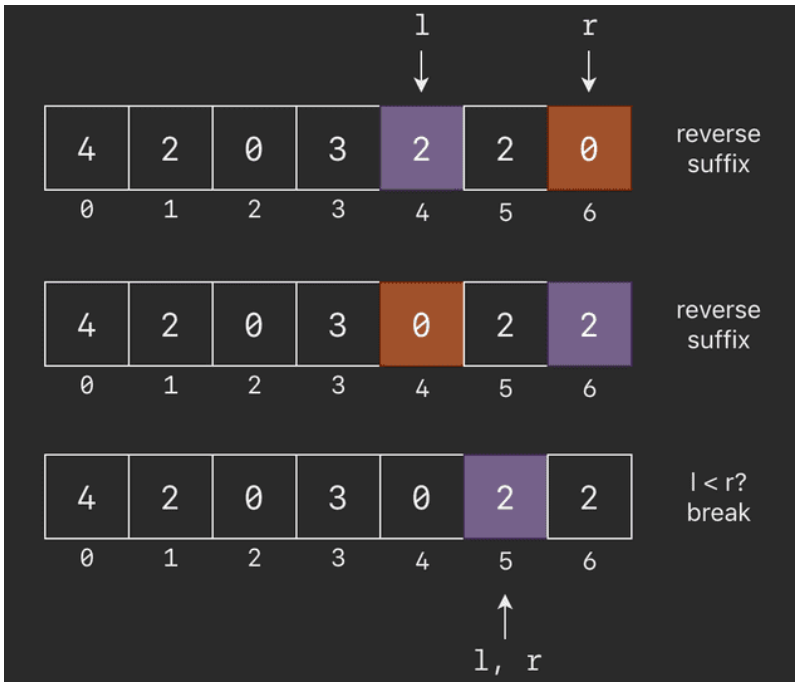


Рис. 8.10. Інверсія елементів суфікса масиву

Таким чином ми отримали наступну лексикографічну перестановку. Якщо початкова множина буде відсортована, то ми отримаємо впорядкований набір перестановок після застосування цього алгоритму n! разів.

8.2.2. Алгоритм Джонсона–Троттера

Алгоритм Джонсона-Троттера (також зустрічається назва Стейнгауса-Джонсона-Троттера) — генерує перестановки, що відрізняються між собою лише однією заміною елементів. Це дозволяє досягти константного середнього часу на генерацію кожної перестановки. Алгоритм був незалежно описаний американським вченим Селмером Джонсоном та канадсько-американським вченим Гейлом Троттером на початку 1960 років. Пізніше виявилося, що цей алгоритм був описаний в праці польського вченого Гюґо Стейнгауса, що вийшла друком в 1958 році.

Алгоритм має рекурсивну структуру, хоча його ефективна реалізація використовує не рекурсію, а ітеративний підхід. Суть алгоритму в наступному. Для генерації перестановок з n елементів потрібно взяти всі можливі перестановки з (n-1) елементів і додати до кожної з них n-ний елемент в усі можливі позиції. В якості ілюстрації розглянемо послідовність натуральних чисел.

Перестановка з одного елемента буде {**1**}.

Для отримання перестановок з двох елементів необхідно додати наступний елемент в обидві можливі позиції попередньої перестановки: {1, **2**}; {**2**, 1}.

Для отримання перестановок з трьох елементів додаємо трійку спочатку в першу перестановку в спадному порядку, а потім в другу перестановку в зростаючому порядку: {1, 2, **3**}; {1, **3**, 2}; {**3**, 1, 2}; {**3**, 2, 1}; {2, **3**, 1}; {2, 1, **3**}.

Зміна порядку додавання необхідна, щоб виконувалася умова про різницю всього в одну заміну елементів між сусідніми перестановками.

Оригінальні реалізації, що були запропоновані Джонсоном і Троттером виконували одну перестановку за час `O(n)`. Проте, ізраїльський вчений Шимон Евен запропонував покращену реалізацію алгоритму, що виконував одну перестановку за час `O(i)`, де `i` — індекс елемента, що рухається. Хоча реальний час генерації кожної перестановки різний, середній час генерації однієї перестановки є константним.

Евен запропонував для кожного елемента зберігати не тільки його позицію, але також напрямок руху: позитивний (рух праворуч), негативний (рух ліворуч) та нульовий. На початку напрямок першого елемента нульовий, а всіх інших — негативний:

```
1 -2 -3
```

На кожному кроці алгоритм шукає найбільший елемент з ненульовим напрямком і рухає його у відповідному напрямку:

```
1 -3 -2
```

Коли поточний елемент досягає першої чи останньої позиції в перестановці або наступний елемент в напрямку руху є більшим за поточний, то напрям руху поточного елемента стає нульовим:

```
3 1 -2
```

Після кожного кроку, всі елементи що є більшими за поточний і мають нульовий напрямок змінюють свій напрямок руху в напрямку поточного елемента. Тобто всі більші елементи за поточний, які мають нульовий напрямок і знаходяться ліворуч від поточного, змінюють свій напрямок на позитивний. Аналогічно елементи, що більші за поточний з нульовими напрямками і знаходяться праворуч від нього, змінюють свій напрямок на негативний. В нашому прикладі, після руху двійки, трійка змінює свій напрямок з нульового на додатній:

```
+3 2 1
```

Наступні два кроки це рух трійки праворуч:

```
2 +3 1
2 1 3
```

Оскільки всі елементи мають нульовий напрямок руху, то алгоритм закінчено. Ми отримали шість перестановок, кожна наступна відрізняється від попередньої лише однією зміною елементів.

8.2.3. Алгоритм Гіпа

Алгоритм Гіпа — генерує перестановки для множини з N елементів. Запропонований Гіпом в 1963 році. Алгоритм генерує наступну перестановку шляхом зміни місцями двох елементів з попередньої перестановки, тоді як інші N-2 елементи залишаються незмінними. На відміну від алгоритму Джонсона-Троттера має значно простішу і компактнішу рекурсивну реалізацію, а тому визнаний найбільш придатним для комп'ютерної імплементації.

Гіп знайшов систематичний метод, що дозволяє обрати на кожному кроці пару елементів для зміни їх місцями, щоб утворити всі можливі перестановки.

Алгоритм Гіпа використовує підхід «поділяй і володарюй» на кожному кроці, для поточних K елементів. На початку алгоритму K=N, а потім зменшується з кожною рекурсивною ітерацією. Кожен крок алгоритму генерує K! перестановок для одних і тих же N-K елементів. Це відбувається завдяки рекурсивного виклику коли K-тий елемент не змінився, а потім K-1 разів коли K-тий елемент міняється місцями з кожним з K-1 елементів.

Формальний опис алгоритму

1. Ізолювати останній елемент і згенерувати перестановки для перших K-1 елементів.
2. Взяти наступний унікальний елемент і поміняти його місцями з останнім елементом.
3. Повторювати кроки 1 і 2 доти, доки всі елементи не стануть останніми по одному разу.

Як бачимо, основою алгоритму є вибір наступного унікального елемента. Яким чином здійснюється цей вибір, розглянемо на прикладі генерації перестановок для множини {1, 2, 3, 4}. Дерево перестановок такої множини для K=3 і зафіксованим останнім елементом 4 подане на рис. 8.11.

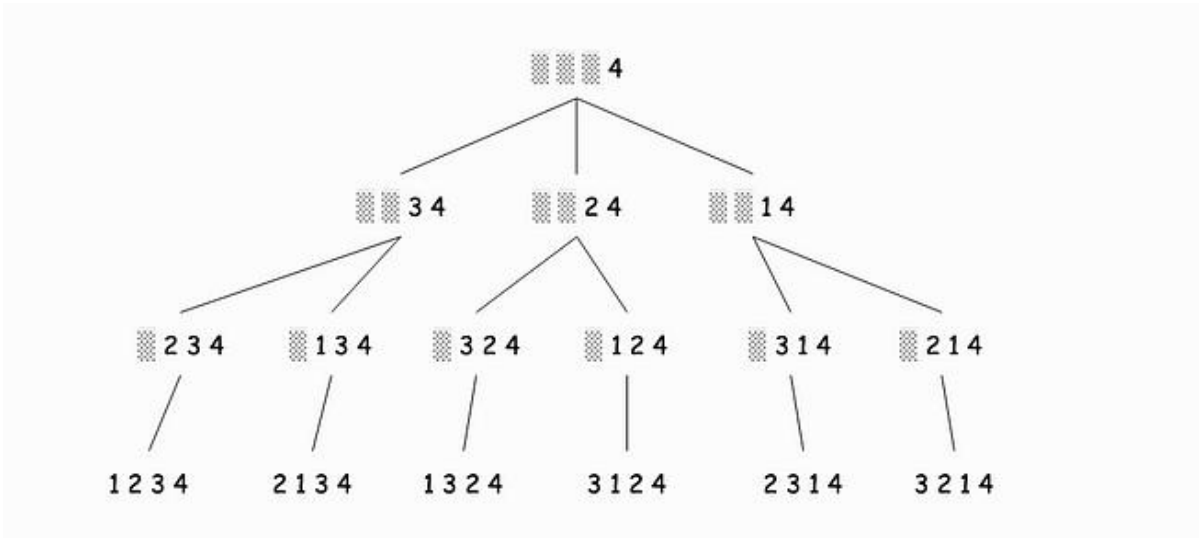


Рис. 8.11. Дерево перестановок множини {1, 2, 3, 4}

Аналогічні три дерева можна побудувати для інших фіксованих останніх елементів.

Зобразимо необхідні параметри для виклику функції генерації перестановки в кожному вузлу дерева (рис . 8.12).

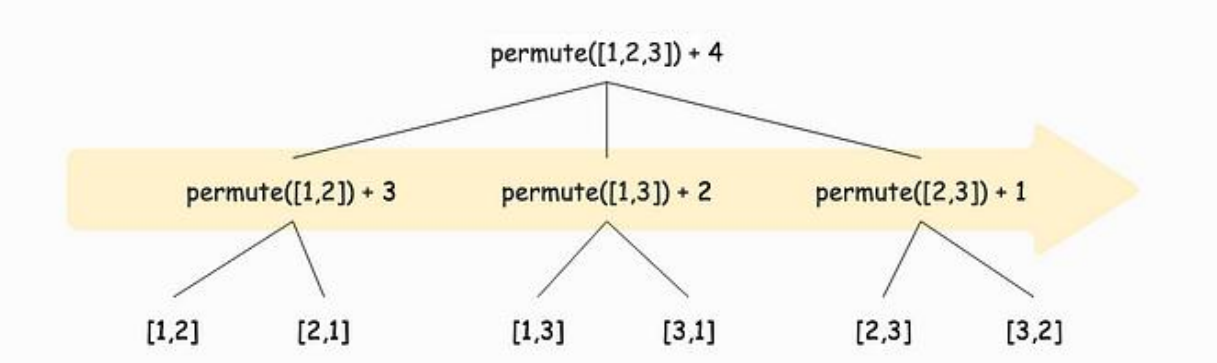


Рис. 8.12. Параметри для функції генерації перестановки

З прикладу вище, ми бачимо, що алгоритм Гіпа обирає наступний унікальний елемент в порядку спадання. Тобто, для кожної наступної ітерації ми обираємо елементи 3 -> 2 -> 1.

Перестановка для двох елементів генерується наступним чином (рис. 8.13).

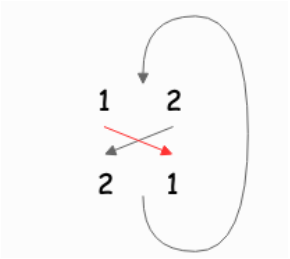


Рис. 8.13. Генерація перестановки для двох елементів

Перестановка для трьох елементів генерується наступним чином (рис. 8.14).

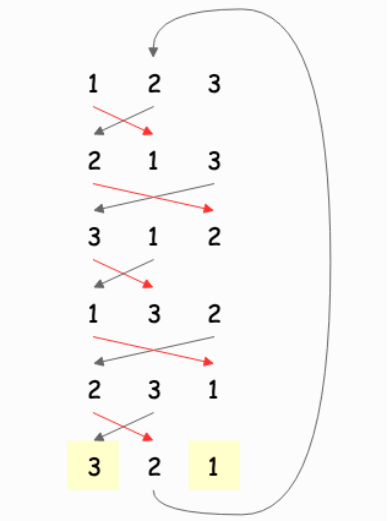


Рис. 8.14. Генерація перестановки для трьох елементів

Як видно на рис. 8.14 після всіх ітерацій з елементами, перший і останній елементи помінялися місцями.

Тепер розглянемо послідовність перестановок для множини з чотирьох елементів (рис. 8.15).

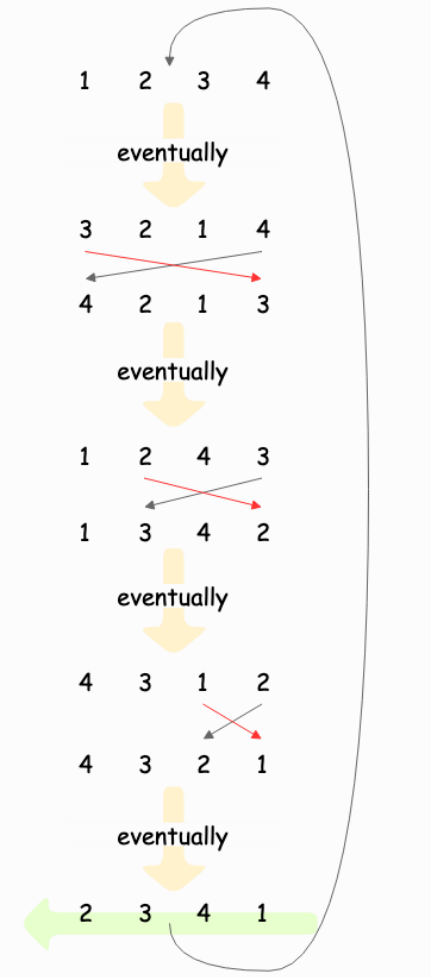


Рис. 8.15. Генерація перестановки для чотирьох елементів

Як бачимо з рис. 8.15 в кінці першої ітерації для перестановки чотирьох елементів ми отримали циклічний зсув елементів ліворуч.

Які висновки тепер ми можемо зробити? Як сформулювати правило вибору наступного унікального елемента? З розглянутих прикладів ми побачили, що для трьох елементів завжди змінюється перший елемент на останній, а для чотирьох елементів — кожен елемент по черзі змінюється з останнім. Для двох елементів ситуацію можна трактувати так само, як і для трьох або так само як і для чотирьох. Тоді яке ж загальне правило для вибору наступного унікального елемента для зміни місцями з останнім?

Правило заміни елементів є основою алгоритму Гіпа. Метод, який використовується для формування пари елементів для зміни місцями, заснований на парності кількості елементів множини. Якщо поточна кількість елементів множини K парна, то виконується почергова заміна останнього елемента з кожним елементом множини. Якщо ж кількість елементів множини K непарна, тоді останній елемент завжди змінюється місцями з першим.

Алгоритм Гіпа працює для будь-якої кількості елементів. Його псевдокод поданий нижче.

```
procedure generate(k : integer, A : array of any):
    if k = 1 then
        output(A)
    else
        // Генеруємо початкову перестановку для незмінного k-го елемента
        generate(k - 1, A)
        // Генеруємо перестановки для k-го елемента,
        // що замінюється з кожним k-1 елементом
        for i := 0; i < k-1; i += 1 do
            // Вибір наступного елемента для зміни залежить від парності
            if k is even then // парна кількість елементів
                swap(A[i], A[k-1])
            else // непарна кількість елементів
                swap(A[0], A[k-1])
            end if
            generate(k - 1, A)
        end for
    end if
```

Поданий псевдокод є досить компактним і зручним для використання.

8.2.4. Інші алгоритми перестановок

Окрім трьох розглянутих алгоритмів генерації перестановок існують також інші алгоритми. Кожен алгоритм робить мінімальну кількість змін в попередній згенерованій перестановці, а тому мають однаковий час виконання O(n*n!). Відмінність між цими алгоритмами полягає в порядку генерації перестановок (рис. 8.16). Якщо цей порядок не важливий, то обирають алгоритм Гіпа.

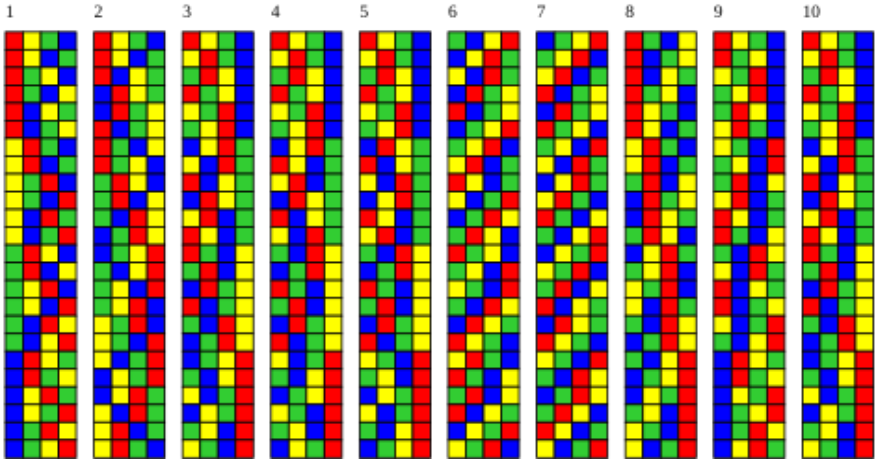


Рис. 8.16. Порядок генерації перестановок для 10 різних алгоритмів

На рис. 8.16 числами від 1 до 10 позначені такі алгоритми.

1. Алгоритм генерації перестановок в лексикографічному порядку.
2. Алгоритм Джонсона-Троттера.
3. Алгоритм Гіпа.
4. Алгоритм зіркової транспозиції Еріха.
5. Алгоритм інвертації префіксу Зака.
6. Алгоритм Савада-Вільямса.
7. Алгоритм Корбетта.
8. Однопрохідний порядок.
9. Однопрохідний код Грея.
10. Алгоритм генерації вкладених перестановок.

8.3. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач, що потребують генерації перестановок або рекурсивної генерації послідовностей.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням одного з розглянутих алгоритмів.

Низький рівень складності

1. Power of Two https://leetcode.com/problems/power-of-two.
2. Power of Three https://leetcode.com/problems/power-of-three.
3. Power of Four https://leetcode.com/problems/power-of-four.
4. Fibonacci Number https://leetcode.com/problems/fibonacci-number.

Середній рівень складності

1. Count Sorted Vowel Strings https://leetcode.com/problems/count-sorted-vowel-strings.
2. Pow(x, n) https://leetcode.com/problems/powx-n.
3. K-th Symbol in Grammar https://leetcode.com/problems/k-th-symbol-in-grammar.
4. Generate Binary Strings without Adjacent Zeros https://leetcode.com/problems/generate-binary-strings-without-adjacent-zeros.

Високий рівень складності

1. Permutation Sequence https://leetcode.com/problems/permutation-sequence.
2. Median of Two Sorted Arrays https://leetcode.com/problems/median-of-two-sorted-arrays.
3. Basic Calculator https://leetcode.com/problems/basic-calculator.

Запитання для самоконтролю

1. Що таке послідовність?
2. Які числові послідовності ви знаєте?
3. Які є способи задання послідовностей?
4. Що таке перестановка?
5. Які алгоритми генерації перестановок ви знаєте?
6. Що таке лексикографічний порядок генерації перестановок?
7. Яка відмінність між алгоритмами Джонсона-Троттера і Гіпа?
8. Які ще алгоритми генерації перестановок ви знаєте?

Посилання на використані джерела

1. https://uk.wikipedia.org/wiki/Послідовність
2. https://uk.wikipedia.org/wiki/Арифметична_прогресія
3. https://uk.wikipedia.org/wiki/Геометрична_прогресія
4. https://uk.wikipedia.org/wiki/Фігурні_числа
5. https://uk.wikipedia.org/wiki/Послідовність_Фібоначчі
6. https://en.wikipedia.org/wiki/Sequence
7. https://www.cuemath.com/algebra/sequences
8. https://www.smoon.me/next-permutation/
9. https://en.wikipedia.org/wiki/Steinhaus-Johnson-Trotter_algorithm
10. https://en.wikipedia.org/wiki/Heap's_algorithm
11. https://medium.com/sodalabs/heaps-algorithm-fun-observation-4986a188a80

9. Гешування

Гешування відноситься до процесу генерації виходів фіксованого розміру із входів змінного розміру. Це робиться за допомогою математичних формул, відомих як геш-функції (реалізовані як алгоритми гешування). Використовується для організації швидкого доступу до даних, швидкого пошуку в текстах, криптографії та інших галузях.

9.1. Визначення

Гешування (хешування, англ. hashing) — перетворення вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини. Такі перетворення також називаються геш-функціями, або функціями згортання, а їхні результати називають гешем, геш-кодом, геш-сумою, або дайджестом повідомлення (англ. message digest).

Гешування застосовується для побудови асоціативних масивів, пошуку дублікатів в серіях наборів даних, побудови унікальних ідентифікаторів для наборів даних, контрольного підсумовування з метою виявлення випадкових або навмисних помилок під час зберігання або передачі, для зберігання паролів в системах захисту (у цьому випадку доступ до області пам'яті, де знаходяться паролі, не дозволяє відновити сам пароль), під час створення електронного підпису (на практиці часто підписується не саме повідомлення, а його геш-образ).

Найбільш вживаний спосіб використання гешування це створення геш-таблиць (рис. 9.1). Геш-таблиця містить пари «ключ-значення», які доступні через індекс.

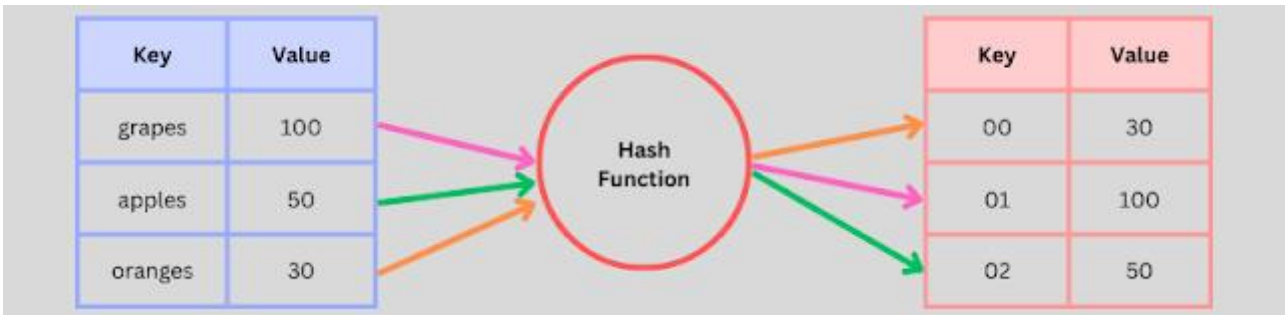


Рис. 9.1. Приклад геш-таблиці

Геш-функція забезпечує перетворення ключа на індекс таблиці. Ключем може бути будь-яка інформація, як числова так і символьна.

У загальному випадку однозначної відповідності між вихідними даними і геш-кодом немає в силу того, що кількість значень геш-функцій менша, ніж число варіантів значень вхідного масиву. Існує безліч масивів з різним вмістом, що дають однакові геш-коди — так звані колізії. Імовірність виникнення колізій відіграє важливу роль в оцінці якості геш-функцій.

Розроблено багато алгоритмів гешування з різними властивостями (розрядність, обчислювальна складність, крипостійкість тощо). Вибір тієї чи іншої геш-функції

визначається специфікою розв'язуваної задачі. Найпростішими прикладами геш-функцій можуть служити контрольна сума або CRC (Cyclic Redundancy Check) .

Розглянемо приклад створення геш-таблиці для множини {«ab», «cd», «efg»}. Мета створення геш-таблиці — мати константний час доступу до елементів множини під час пошуку або оновлення інформації. Нехай рядки з множини одночасно виступають і ключами і значеннями, які нам необхідні для роботи. Тепер нам необхідно визначити, яким чином нам зберегти ці рядки в масиві, щоб мати унікальний індекс для кожного з них. Ми знаємо, що для цієї мети нам необхідно визначити геш-функцію, яка перетворить ключі у вигляді рядків на унікальні індекси. Визначимо просту функцію, яка рахуватиме геш на основі суми символів рядка. Для цього використаємо порядкові номери букв в абетці в якості їх числових еквівалентів: a=1, b=2, c=3 і т.д. Тоді числові відповідники для кожного рядка будуть такі:

- «ab»=1+2=3
- «cd»=3+4=7
- «efg»=5+6+7=18.

Наприклад ми хочемо зберегти рядки в таблицю, що має розмір 7. Використаємо просту геш-функцію, що використовує залишок від ділення числового подання ключа на розмір таблиці: `key mod Table size`. Таким чином ми отримаємо такі індекси геш-таблиці для зберігання рядків (рис. 9.2):

- «ab» зберігається в `3 mod 7 = 3`
- «cd» зберігається в `7 mod 7 = 0`
- «efg» зберігається в `18 mod 7 = 4`.

0	1	2	3	4	5	6
cd			ab	egf		

Рис. 9.2. Геш-таблиця для {«ab», «cd», «efg»}

Розглянутий приклад є одним із найпростіших застосувань гешування. Оскільки вибір гарної геш-функції є основою ефективного гешування, ми детальніше розглянемо типи геш-функцій.

9.2. Геш-функція

Геш-функція (хеш-функція) — функція, що перетворює вхідні дані будь-якого (як правило великого) розміру в дані фіксованого розміру. Ключові властивості геш-функції наступні:

- **детермінованість** — результат роботи функції повинен бути завжди однаковим для одних і тих же вхідних даних;
- **фіксований вихідний розмір** — розмір вихідних даних функції повинен бути однаковим незалежно від розміру вхідних даних;

- **ефективність** — функція повинна швидко обробляти вхідні дані;
- **рівномірність** — функція повинна генерувати результати з нормальним розподілом у вихідній множині, щоб уникати кластеризації;
- **стійкість до знаходження першовзору** — неможливість обчислити вхідні значення функції на основі її вихідних значень;
- **стійкість до колізій** — низька ймовірність знайти два різних входи для яких буде згенеровано один і той же результат;
- **ефект лавини** — незначна зміна у вхідних даних призводить до значної зміни результату.

Існує досить багато різних типів геш-функцій. Кожен з них відрізняється від іншого математичним підходом до формування геша. Розглянемо деякі з них більш детально.

Метод на основі ділення використовує ділення ключа (k) на просте число (m) з подальшим використанням залишку від ділення в якості геша: $h(k)=k \bmod m$.

Переваги: легкий для реалізації.

Недоліки: генерує геші з поганим розподілом, якщо неправильно обрати m.

Метод на основі множення використовує константу ($0 < A < 1$) для множення на ключ. Для отримання геша, дробова частина результату множиться на константу (m) і потім береться ціла частина: $h(k)=[m(kA \bmod 1)]$.

Переваги: менш чутливий до значення m.

Недоліки: більш складний в реалізації, ніж метод на основі ділення.

Метод середини квадрату використовує піднесення ключа до другої степені з подальшим вибором середніх цифр в якості гешу.

Переваги: генерує геші з гарним розподілом.

Недоліки: вимагає більше обчислювальних зусиль.

Метод складання ділить ключ на однакові частини, сумує їх, а потім бере модуль по m від суми.

Переваги: простий та легкий в реалізації.

Недоліки: залежить від вибору схеми розбиття ключа на частини.

Криптографічний метод використовується для створення зашифрованих геш-функцій, що стійкі до знаходження першого і другого першовзорів, а також стійкі до колізій. Прикладами таких геш-функцій є MD5, SHA-1, SHA-256.

Переваги: висока безпека.

Недоліки: складність в обчисленні.

Універсальне гешування це сімейство геш-функцій, що мінімізують шанси на колізію та використовують наступну формулу: $h(k)=((a*k+b) \bmod p) \bmod m$.

В цій формулі a та b випадково обрані константи, p — просте число більше за m, k — ключ.

Переваги: зменшує імовірність колізії.

Недоліки: вимагає більше розрахунків та пам'яті.

Досконале гешування має на меті створити геш-функцію, що гарантовано немає колізій для визначеного статичного набору ключів. Існує два види досконалого гешування: мінімальне та не мінімальне. Перший вид продукує індекси в межах кількісті ключів. Другий — продукує індекси в межах, що перевищує кількість ключів.

Переваги: відсутність колізій.

Недоліки: складність створення.

Звичайно, що це далеко не всі типи геш-функцій, а лише деякі з них.

9.3. Колізії та їх вирішення

Як вже зазначалося, колізії в гешах виникають коли для двох різних ключів генеруються однакові індекси масиву. Ймовірність виникнення колізії в геші залежить від геш-функції, яка використовується, а також від розподілу значень геша.

В процесі гешування для великого значення ключів генерується менша кількість індексів. Саме тому й виникає ймовірність отримання двох однакових індексів. У випадку, якщо згенерований індекс вже присутній в геш-таблиці, необхідно застосувати технологію вирішення колізій (рис. 9.3).

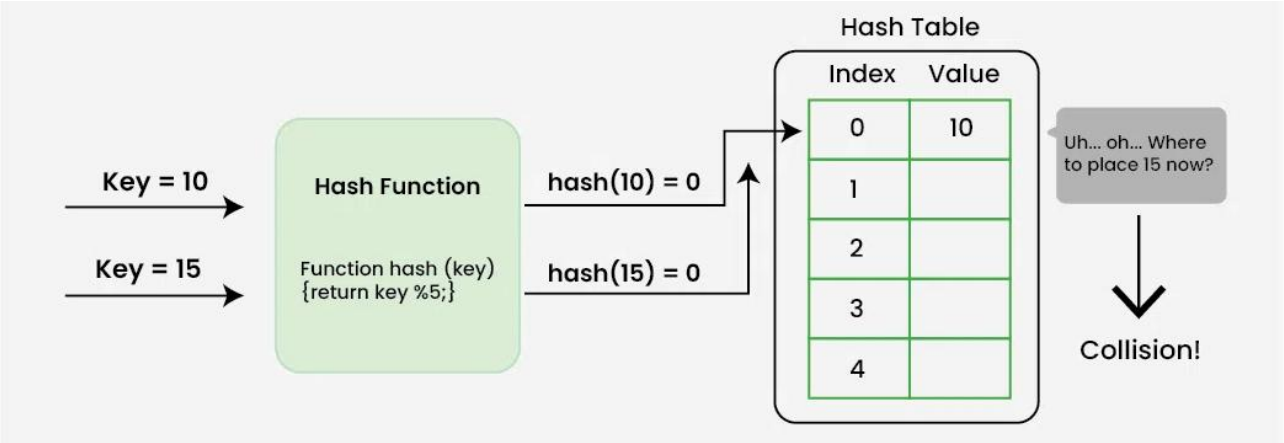


Рис. 9.3. Колізія гешування

Для вирішення колізій використовують дві основні техніки: метод ланцюгів та метод відкритого адресування. Другий метод в свою чергу ділиться на три: лінійне пробування, квадратичне пробування та подвійне гешування (рис. 9.4).

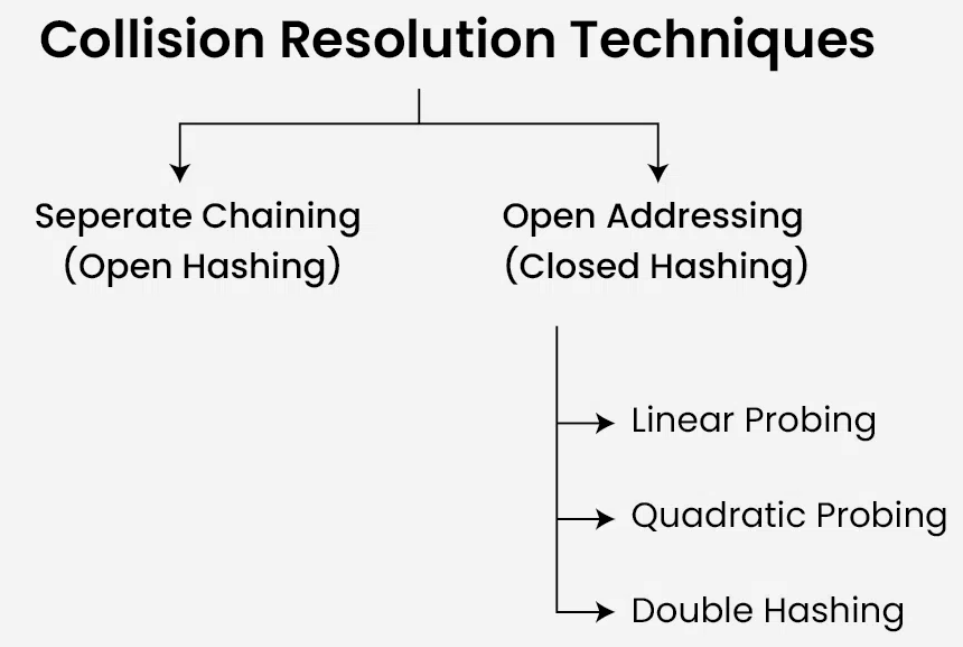


Рис. 9.4. Техніки вирішення колізій

Розглянемо кожен метод більш детально.

9.3.1. Метод ланцюгів

Ідея цього методу полягає в створенні зв'язного списку для ключів, що отримали однакові індекси. Посилання на початок цього списку записується в геш-таблицю. Метод ланцюгів простий, але вимагає додаткової пам'яті за межами геш-таблиці.

Розглянемо приклад застосування методу ланцюгів. Нехай в нас є проста геш-функція: `hash = key%5` та елементи, які необхідно помістити в геш-таблицю: 12, 22, 15, 25, 37.

Крок 1. Створюємо порожню геш-таблицю зі значеннями геш індексів від 0 до 4 (рис. 9.5).

Рис. 9.5. Створення порожньої геш-таблиці

Крок 2. Гешуємо перший ключ 12 та отримуємо індекс 2 для нього (рис. 9.6).

Рис. 9.6. Гешування першого ключа

Крок 3. Гешуємо другий ключ 22 і отримуємо індекс 2 для нього. Але індекс 2 вже зайнятий ключем 12, а тому створюємо зв'язний список і додаємо його в індекс 2 (рис. 9.7).

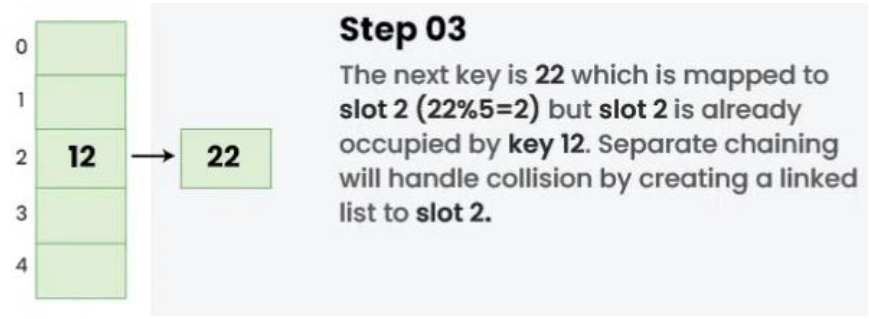


Рис. 9.7. Гешування другого ключа і створення зв'язного списку

Крок 4. Гешуємо третій ключ 15 і отримуємо індекс 0 для нього (рис. 9.8).

Рис. 9.8. Гешування третього ключа

Крок 5. Гешуємо четвертий ключ 25 і отримуємо індекс 0 для нього. Оскільки індекс 0 вже зайнятий, то знову створюємо зв'язний список і додаємо його до індексу 0 (рис. 9.9).

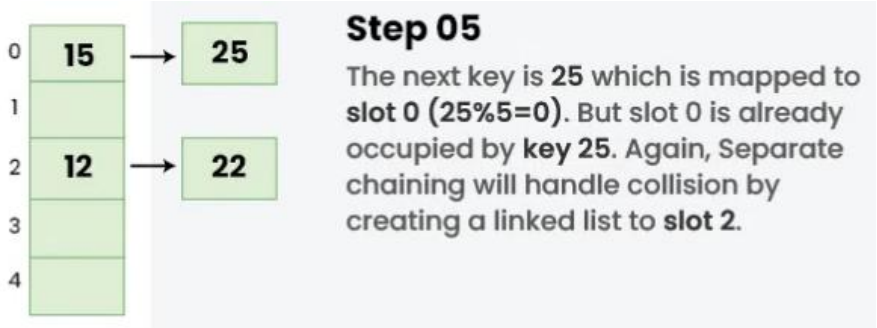


Рис. 9.9. Гешування четвертого ключа і створення нового зв'язного списку

Останній ключ 37 також буде доданий до списку, що міститься в індексі 2. Таким чином ми отримали геш-таблицю на 5 індексів, але лише 2 з них зайняті. До того ж нам знадобилося два додаткових списки на 2 і 3 елементи, щоб зберегти всі ключі. Проблема колізії вирішена, але за рахунок додаткової пам'яті та збільшення часу доступу до елементів, що зберігаються в геш-таблиці.

9.3.2. Метод відкритої адресації

Метод відкритої адресації забезпечує зберігання всіх елементів в геш-таблиці без створення додаткових структур даних. Кожна комірка таблиці містить елемент або порожнє значення. Під час пошуку індексу в таблиці перевіряються комірки на предмет їх доступності для запису нового елемента. В залежності від методу пошуку вільної комірки розрізняють три підходи.

Лінійне пробування

Під час лінійного пробування здійснюється послідовна перевірка всіх індексів геш-таблиці починаючи з оригінального індексу, в якому виникла колізія. Спочатку перевіряються індекси в сторону збільшення, а якщо досягнуто кінця геш-таблиці, то продовжується пошук з нульового індексу. Як тільки буде знайдено порожню комірку, туди буде записано значення.

Розглянемо приклад формування геш-таблиці з використарнням лінійного пробування. Нехай в нас є проста геш-функція: `hash = key%5` та елементи, які необхідно помістити в геш-таблицю: 50, 70, 76, 85, 93.

Крок 1. Створюємо порожню таблицю на 5 елементів.

Крок 2. Гешуємо перший ключ 50 та отримуємо індекс 0 для нього (рис. 9.10).

Рис. 9.10. Гешування першого ключа

Крок 3. Гешуємо другий ключ 70 та отримуємо індекс 0 для нього. Оскільки нульовий індекс вже зайнятий, то перевіряємо наступний індекс 1. Цей індекс вільний, а отже записуємо елемент 70 туди (рис. 9.11).

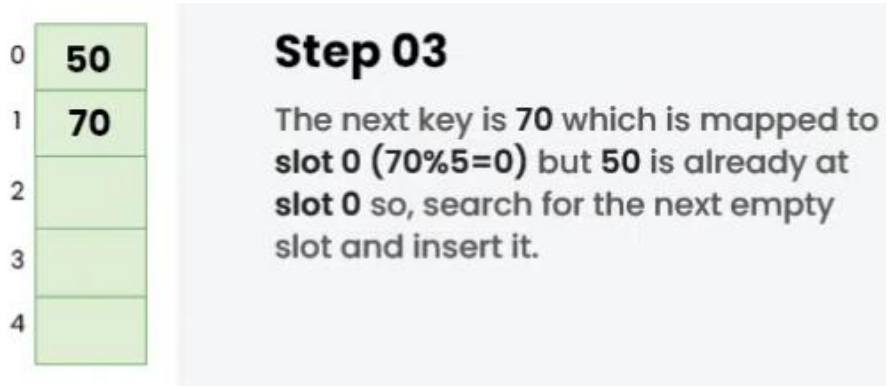


Рис. 9.11. Гешування другого ключа

Крок 4. Гешуємо третій ключ 76 та отримуємо індекс 1 для нього. Оскільки перший індекс вже зайнятий, то перевіряємо наступний індекс 2. Цей індекс вільний, а отже записуємо елемент 76 туди (рис. 9.12).

Рис. 9.12. Гешування третього ключа

Крок 5. Гешуємо четвертий ключ 85 та отримуємо індекс 0 для нього. Оскільки нульовий індекс вже зайнятий, то перевіряємо наступний індекс 1. Цей індекс також зайнятий, тому переходимо до наступного, який вільний, і записуємо елемент 85 туди (рис. 9.13).

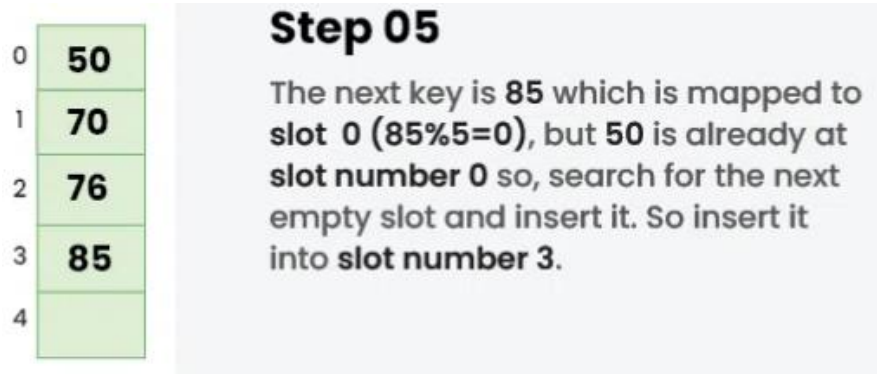


Рис. 9.13. Гешування четвертого ключа

Крок 6. Гешуємо останній ключ 93 та отримуємо індекс 3 для нього. Оскільки третій індекс вже зайнятий, то перевіряємо наступний індекс 4. Цей індекс вільний, а тому записуємо елемент 93 туди (рис. 9.14).

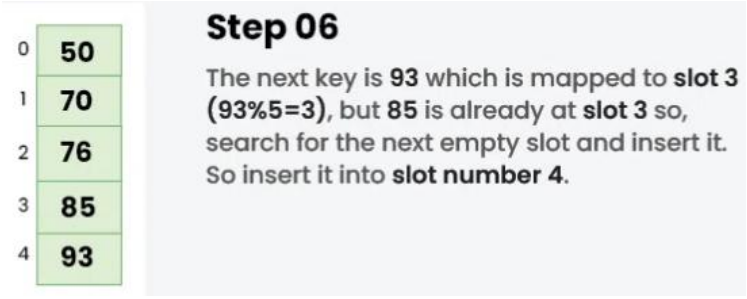


Рис. 9.14. Гешування останнього ключа

Використавши лінійне пробування ми заповнили всі комірки геш-таблиці за рахунок збільшення часу пошуку елементів. Наприклад для елемента 85 необхідно переглянути 4 комірки замість однієї.

Квадратичне пробування

На відміну від лінійного пробування, квадратичне перевіряє не кожен наступний індекс, а квадрат кожного наступного індексу.

Розглянемо приклад формування геш-таблиці з використанням квадратичного пробування. Нехай в нас є проста геш-функція: `hash = key%7` та елементи, які необхідно помістити в геш-таблицю: 22, 30, 50.

Крок 1. Створюємо порожню таблицю на 7 елементів.

Крок 2. Гешуємо перший ключ 22 та отримуємо індекс 1 для нього (рис. 9.15).

Рис. 9.15. Гешування першого ключа

Крок 3. Гешуємо другий ключ 30 та отримуємо індекс 2 для нього. Оскільки цей індекс вільний, то записуємо елемент 30 туди (рис. 9.16).

Рис. 9.16. Гешування другого ключа

Крок 4. Гешуємо третій ключ 50 та отримуємо індекс 1 для нього. Оскільки перший індекс вже зайнятий, то перевіряємо індекс $1+1^2=2$. Цей індекс теж зайнятий, а тому перевіряємо наступний індекс $1+2^2=5$. Цей індекс вільний, а отже записуємо елемент 50 туди (рис. 9.17).

Рис. 9.17. Гешування останнього ключа

Квадратичне пробування забезпечує вирішення колізій в геш-таблиці шляхом збільшення кількості перевірок для пошуку необхідного елемента.

Подвійне гешування

Як зрозуміло з назви, цей метод відкритої адресації використовує дві геш-функції: перша використовує ключ, як вхідний параметр і генерує індекс геш-таблиці, а друга — використовує результат першої для генерації нового індексу. Комбінація цих геш-функцій описується формулою:

$$h(k, i) = (h1(k) + i * h2(k)) \% n,$$

де i — натуральне число, що позначає номер колізії;

k — ключ який гешується;

n — розмір геш-таблиці.

Розглянемо приклад застосування подвійного гешування. Задані такі ключі для геш-таблиці: 27, 43, 692, 72. Розмір геш-таблиці 7. Перша геш-функція: h1(k)=k%7. Друга геш-функція: h2(k)=1+(k%5).

Крок 1. Створюємо порожню таблицю на 7 елементів.

Крок 2. Гешуємо перший ключ 27 та отримуємо індекс 6 для нього. Оскільки цей індекс не зайнятий, то додаємо в нього елемент 27 (рис. 9.18).

Рис. 9.18. Гешування першого ключа

Крок 3. Гешуємо другий ключ 43 та отримуємо індекс 1 для нього. Оскільки цей індекс вільний, то записуємо елемент 43 туди (рис. 9.19).

Рис. 9.19. Гешування другого ключа

Крок 4. Гешуємо третій ключ 692 та отримуємо індекс 6 для нього. Оскільки цей індекс вже зайнятий, то застосовуємо подвійне гешування і отримуємо індекс 2. Цей індекс вільний, отже записуємо елемент 692 туди (рис. 9.20).

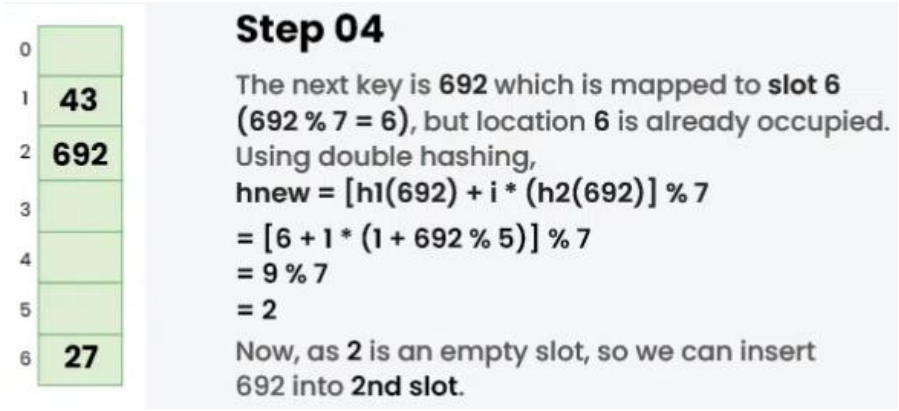


Рис. 9.20. Гешування третього ключа

Крок 5. Гешуємо останній ключ 72 та отримуємо індекс 2 для нього. Оскільки цей індекс вже зайнятий, то застосовуємо подвійне гешування і отримуємо індекс 5. Цей індекс вільний, отже записуємо елемент 72 туди (рис. 9.21).

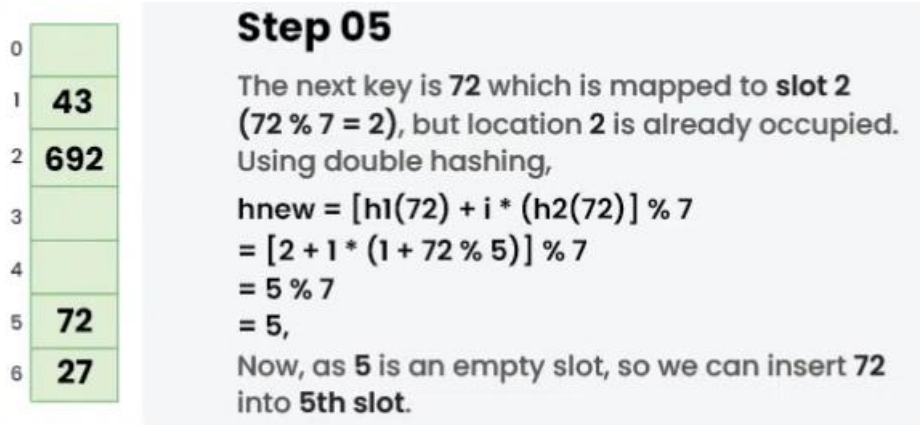


Рис. 9.21. Гешування останнього ключа

Якщо після подвійного гешування ми отримуємо індекс, що вже містить елемент, то це означає, що наша геш-таблиця має високий фактор завантаження (load factor) — відношення кількості ключів до розміру геш-таблиці. У випадку

значення фактору завантаження `0.75` і вище, геш-таблиця потребує повторного гешування (rehashing) зі збільшенням свого розміру вдвічі.

9.4. Гешування рядків

Алгоритми гешування допомагають у вирішенні багатьох задач. Серед них задача порівняння двох рядків. Якщо використовувати метод «грубої сили» (brute force) і порівнювати рядки символ за символом, то такий підхід буде мати часову складність O(`min`(n_1,n_2)), де n_1 і n_2 — розміри першого та другого рядка відповідно. Для швидкого порівняння рядків нам необхідно застосувати гешуваня. Тоді порівняння двох рядків зведеться до порівняння їх гешів, що матиме часову складність O(1).

9.4.1. Поліноміальний ковзний геш

Щоб перетворити рядок на число, необхідно застосувати до нього геш-функцію. На початку цього розділу ми вже використовували числове подання символів для додавання їх у геш-таблицю. Але для вирішення задачі порівняння двох рядків використання геш-таблиці не потрібне. Необхідно застосувати геш-функцію не до окремо взятого символу рядка, а до всього рядка. Ця функція повинна генерувати однакові геші для однакових рядків. Тобто, якщо рядки `s` і `t` однакові, то їхні геші теж мають бути однакові: h(`s`) = h(`t`).

Варто зазначити, що зворотне твердження не є істинним. Тобто якщо два геші однакові h(`s`) = h(`t`), то це не означає, що початкові рядки теж є однаковими. Все через вже розглянуті колізії в гешах. Наприклад, для рядків довжиною не більше 15 маленьких латинських літер, кількість комбінацій буде більшою за 2^{64}. Використання чисел, що перевищують 64 біти не доцільна, оскільки це призводить до часової складності O(n), що аналогічно застосуванню методу «грубої сили». Тому в реальності ми оперуємо числами, розрядність яких не перевищує 64 біти, а отже наявність колізій неминуча.

Найкращим варіантом для нас буде перетворити будь-який рядок в ціле число у фіксованому проміжку [`0`, `m`), а потім порівняти два таких числа. Звичайно ми хочемо, щоб умова h(`s`) != h(`t`) була дуже ймовірною у випадку `s != t`.

Найчастіше в задачах з рядками використовується поліноміальний ковзний геш (polynomial rolling hash). Для рядка `s` довжини n він розраховується за наступною формулою:

$$hash(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + \cdots + s[n-1] \cdot p^{n-1} \; mod \; m$$

де p та `m` деякі додатні цілі числа.

Має сенс обрати p простим числом, що приблизно дорівнює кількості літер абетки, що використовується в задачі. Наприклад, якщо використовуються лише маленькі латинські літери, яких є 26 штук, то доцільно обрати p=31. Якщо використовуються і великі і маленькі літери, тоді їх загальна кількість стає 52, і p=53 найкращий вибір для такої задачі.

Щодо числа m, то очевидно, що це має бути велике число, адже ймовірність колізії дорівнює приблизно 1/m. Може здатися, що вибір $m=2^{64}$ є найкращим варіантом, але це не так. Існує метод генерації рядків, який породжує багато колізій для такого значення числа m, а тому на практиці не рекомендується обирати $m=2^{64}$. Гарним вибором для числа m буде велике просте число, наприклад $m=10^9+9$. Це велике число, але воно дозволяє виконувати множення в межах 64 бітного цілого.

Нижче подано приклад розрахунку геша для рядка s, який містить лише маленькі латинські букви. Кожна буква з рядка конвертується в число, що відповідає її порядковому номеру в абетці: a -> 1, b -> 2, …, z -> 26. Використовувати значення 0 в якості початкового недоцільно, оскільки тоді геші таких рядків як: a, aa, aaa, … всі будуть пораховані як 0.

```
long long compute_hash(string const& s)
{   const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s)
    {   hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Тепер розберемо чому в назві геша присутнє слово *ковзний*. В попередньому прикладі ми шукали геш всього рядка. У випадку, коли нам потрібен геш якоїсь частини рядка, ми можемо порахувати його використавши початковий та кінцевий індекс символів:

$$hash(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \ mod\ m$$

Помноживши обидві сторони рівняння на p^i, отримаємо наступне:

$$hash(s[i \dots j]) \cdot p^i = \sum_{k=i}^{j} s[k] \cdot p^k \ mod\ m$$

тобто

$$hash(s[i \dots j]) \cdot p^i = hash(s[0 \dots j]) - hash(s[0 \dots i-1])\ mod\ m.$$

Знаючи геш кожного префіксу, ми можемо вирахувати геш підрядка використавши цю формулу. Тобто ковзаючи шаблоном підрядка заданої довжини по всьому рядку від початкового символу до останнього ми можемо

розрахувати геш шляхом відкидання старого символу і додаванням нового символу. Саме тому цей геш і називають ковзним.

Щоб отримати геш підрядка з попередньої формули нам необхідно поділити обидві частини на p^i, що є складною операцією. Тому на практиці рахують не точний геш, а геш помножений на деяку степінь числа p.

9.4.2. Алгоритм Рабіна-Карпа

Алгоритм Рабіна-Карпа це алгоритм пошуку рядків у тексті. Створений американцем Річардом Карпом та ізраїльтянином Майклом Рабіном в 1987 році. Алгоритм знаходить рядок довжиною s в тексті довжиною t за час O(s+t).

Ключовим аспектом цього алгоритму є швидке обчислення ковзного гешу. Тому геш який застосовують в цьому алгоритмі дещо відрізняється від розглянутого нами в попередньому параграфі. Зазвичай використовують схему відбитків пальців Рабіна або аналогічні за ефективністю методи. Але для простоти розрахунків ми використаємо вже відому нам формулу ковзного гешу з p=31 та $m=10^9+9$.

Для розрахунку гешів усіх префіксів, що починаються з індексу i, використаємо наступну форму:

$$hash(s[i]) = \big(hash(s[i-1]) + s[i] \cdot p^i\big) \bmod m$$

Для розрахунку геша підрядка довжиною k, що починається з індексу i, будемо використовувати таку формулу:

$$hash(s[i+k]) = (hash(s[i+k]) + m - hash(s[i])) \bmod m$$

Формальний опис алгоритму

1. Обрати базове число p та модуль m.
2. Ініціалізувати значення геша тексту нулем.
3. Розрахувати значення гешу для рядка s.
4. Рухатися вздовж тексту від початку до кінця і розраховувати геші поточного підрядка довжиною s.
5. Порівняти значення гешів поточного підрядка і шуканого рядка. Якщо вони співпадають, то це потенційно знайдений підрядок. Тепер його необхідно порівняти з шуканим рядком символ за символом, щоб уникнути можливого фальшивого спрацювання через колізію.

Псевдокод алгоритму

```
function RabinKarp(string s[1..n], string pattern[1..m])
    hpattern := hash(pattern[1..m]);
    for i from 1 to n-m+1
        hs := hash(s[i..i+m-1])
        if hs = hpattern
```

```
            if s[i..i+m-1] = pattern[1..m]
                return i
    return not found
```

Приклад виконання алгоритму

В якості прикладу розглянемо текст з десяти латинських літер «ABABCABDAB» в якому будемо шукати рядок з трьох літер «ABD». Виконаємо покрокова алгоритм Рабіна-Карпа. В якості кодів літер будемо використовувати їх позицію в абетці.

На першому кроці оберемо базове число `p=31` та модуль `m=10`. Далі ініціалізуємо геш тексту нулем `hash(t[0])=0` та розраховуємо геш рядка «ABD»:

$$hash(s) = 1 + 2 \cdot 31 + 4 \cdot 31^2 \; mod \; 10 = 7$$

Тепер запускаємо цикл ковзного гешу вздовж тексту і рахуємо геші кожного підрядка тексту довжиною 3. Але перед цим розрахуємо геші всіх префіксів тексту для ефективної роботи алгоритму.

$$hash(t[1]) = (0 + 1 \cdot 31^0) \; mod \; 10 = 1$$
$$hash(t[2]) = (1 + 2 \cdot 31^1) \; mod \; 10 = 3$$
$$hash(t[3]) = (3 + 1 \cdot 31^2) \; mod \; 10 = 4$$
$$hash(t[4]) = (4 + 2 \cdot 31^3) \; mod \; 10 = 6$$
$$hash(t[5]) = (6 + 3 \cdot 31^4) \; mod \; 10 = 9$$
$$hash(t[6]) = (9 + 1 \cdot 31^5) \; mod \; 10 = 0$$
$$hash(t[7]) = (0 + 2 \cdot 31^6) \; mod \; 10 = 2$$
$$hash(t[8]) = (2 + 4 \cdot 31^7) \; mod \; 10 = 6$$
$$hash(t[9]) = (6 + 1 \cdot 31^8) \; mod \; 10 = 7$$
$$hash(t[10]) = (7 + 1 \cdot 31^9) \; mod \; 10 = 8$$

Ітерація 1. Рахуємо геш підрядка «ABA» (рис. 9.22).

$$hash(t[3]) = (4 + 10 - 0) \; mod \; 10 = 4$$

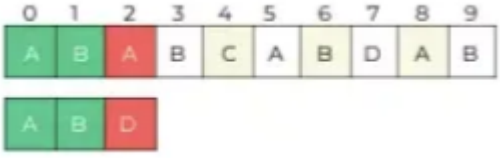


Рис. 9.22. Перша ітерація алгоритму

Геші рядка та підрядка не співпадають `hash(s) != hash(t[3])` отже шукаємо далі.

Ітерація 2. Рахуємо геш підрядка «BAB» (рис. 9.23).

$$hash(t[4]) = (6 + 10 - 1) \; mod \; 10 = 5$$

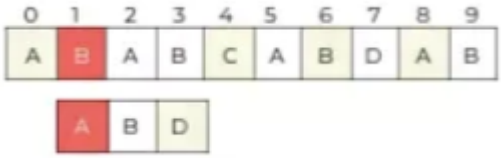


Рис. 9.23. Друга ітерація алгоритму

Геші рядка та підрядка не співпадають hash(s) != hash(t[4]) отже шукаємо далі.

Ітерація 3. Рахуємо геш підрядка «ABC» (рис. 9.24).

$$hash(t[5]) = (9 + 10 - 3)\ mod\ 10 = 6$$

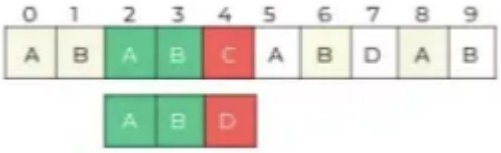


Рис. 9.24. Третя ітерація алгоритму

Геші рядка та підрядка не співпадають hash(s) != hash(t[5]) отже шукаємо далі.

Ітерація 4. Рахуємо геш підрядка «BCA» (рис. 9.25).

$$hash(t[6]) = (0 + 10 - 4)\ mod\ 10 = 6$$

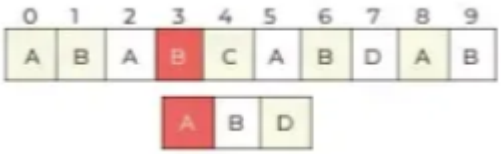


Рис. 9.25. Четверта ітерація алгоритму

Геші рядка та підрядка не співпадають hash(s) != hash(t[6]) отже шукаємо далі.

Ітерація 5. Рахуємо геш підрядка «CAB» (рис. 9.26).

$$hash(t[7]) = (2 + 10 - 6)\ mod\ 10 = 6$$

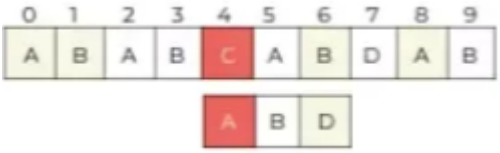


Рис. 9.26. П'ята ітерація алгоритму

Геші рядка та підрядка не співпадають hash(s) != hash(t[7]) отже шукаємо далі.

Ітерація 6. Рахуємо геш підрядка «ABD» (рис. 9.27).

$$hash(t[8]) = (6 + 10 - 9)\ mod\ 10 = 7$$

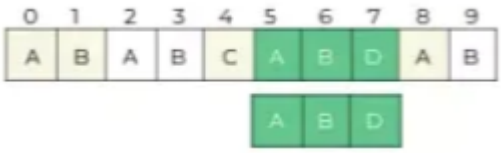


Рис. 9.27. Шоста ітерація алгоритму

Геші рядка та підрядка співпадають hash(s) = hash(t[8]) отже ми знайшли потенційного кандидата на співпадіння. Порівнюємо рядок та підрядок посимвольно і отримуємо повне співпадіння. Таким чином алгоритм закінчено.

9.5. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач, що використовують гешування.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням гешів.

Низький рівень складності

1. Design HashSet https://leetcode.com/problems/design-hashset.
2. Design HashMap https://leetcode.com/problems/design-hashmap.
3. Two Sum https://leetcode.com/problems/two-sum.
4. Roman to Integer https://leetcode.com/problems/roman-to-integer.
5. Word Pattern https://leetcode.com/problems/word-pattern.

Середній рівень складності

1. Integer to Roman https://leetcode.com/problems/integer-to-roman.
2. Valid Sudoku https://leetcode.com/problems/valid-sudoku.
3. Set Matrix Zeroes https://leetcode.com/problems/set-matrix-zeroes.
4. Bulls and Cows https://leetcode.com/problems/bulls-and-cows.
5. Repeated DNA Sequences https://leetcode.com/problems/repeated-dna-sequences.

Високий рівень складності

1. Shortest Palindrome Integer to Raman https://leetcode.com/problems/integer-to-roman.
2. Longest Happy Prefix https://leetcode.com/problems/longest-happy-prefix.
3. Distinct Echo Substrings https://leetcode.com/problems/distinct-echo-substrings.
4. Find Substring With Given Hash Value https://leetcode.com/problems/find-substring-with-given-hash-value.
5. Longest Duplicate Substring https://leetcode.com/problems/longest-duplicate-substring.

Запитання для самоконтролю

1. Що таке гешування?
2. Які типи геш-функцій ви знаєте?
3. Що таке геш-таблиця?
4. Що таке колізії в геш-таблицях?
5. Які є способи усунення колізій?
6. Як обчислюється геш рядка?
7. Що таке поліноміальний ковзний геш?
8. Для чого призначений алгоритм Рабіна-Карпа?

Посилання на використані джерела

1. https://en.wikipedia.org/wiki/Hash_table.
2. https://uk.wikipedia.org/wiki/Хеш-функція.
3. https://en.wikipedia.org/wiki/Hash_function.
4. https://www.techtarget.com/searchdatamanagement/definition/hashing.
5. https://www.geeksforgeeks.org/introduction-to-hashing-2.
6. https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions.
7. https://cp-algorithms.com/string/string-hashing.html
8. https://en.wikipedia.org/wiki/Rabin-Karp_algorithm
9. https://cp-algorithms.com/string/rabin-karp.html
10. https://medium.com/@Roshan-jha/understanding-rabin-karp-algorithm-for-string-matching-e968dbe296b2
11. https://www.programiz.com/dsa/rabin-karp-algorithm

10. Динамічне програмування

Термін «динамічне програмування» вперше введений в обіг Річардом Беллманом у 1940-х роках. Слово «динамічне» було обране Беллманом, тому що звучало більш переконливо і краще підходило для передачі того факту, що проблема оптимального управління, яку він розв'язував цим методом, має аспект залежності від часу. Слово «програмування» в цьому словосполученні в дійсності до «традиційного» програмування (написання тексту програм) майже ніякого відношення не має. Це використання таке саме як і в словосполученнях лінійне програмування та математичне програмування, які фактично є синонімами для математичної оптимізації.

10.1. Визначення

Динамічне програмування є водночас і методом математичної оптимізації і методом комп'ютерного програмування. В обох контекстах воно використовує підхід спрощення пошуку розв'язку складної задачі, розбиттям її на простіші підзадачі, часто методом рекурсії. Хоча деякі задачі не можуть бути розв'язані таким чином, рішення, які охоплюють кілька точок у часі дійсно часто розбиваються рекурсивно на підзадачі. Беллман називав це принципом оптимальності. Подібно до цього, в комп'ютерних науках про проблему, яка може бути розбита на підзадачі рекурсивно, говорять що вона має *оптимальну підструктуру*.

Простими словами, динамічне програмування — це метод розв'язання задач, що ґрунтується на розбитті складної задачі на безліч дрібніших.

Важливо враховувати такі моменти:

- для ефективного застосування цього методу необхідно запам’ятовувати рішення підзадач;
- підзадачі мають загальну структуру, що дає змогу використовувати однорідний спосіб їх розв'язання, замість того, щоб кожну вирішувати окремо різними алгоритмами.

За допомогою динамічного програмування ефективно розв'язують задачі з оптимізації, наприклад, якщо потрібно знайти найбільше або найменше значення функції. Динамічне програмування також активно застосовується в задачах планування, де потрібно визначити оптимальну послідовність дій.

Динамічне програмування застосовується для задач які мають оптимальну підструктуру та задачі, що перекриваються.

Оптимальна підструктура вже згадувалася в першому розділі коли ми розглядали жадібний підхід.

Задачі, що перекриваються. Тут ми можемо зіткнутися з ситуацією, коли різні підзадачі мають спільні частини. У такому разі кажуть, що в нас є підзадачі, які перекриваються. Щоб не розв'язувати одну й ту саму задачу декілька разів, ми

зберігаємо результати підзадач у пам'яті та знову використовуємо їх під час розв'язання більших задач. Це допомагає значно прискорити процес розв'язання.

Наприклад послідовність Фібоначчі, яку ми розглядали у восьмому розділі, має наступний граф підзадач (рис. 10.1).

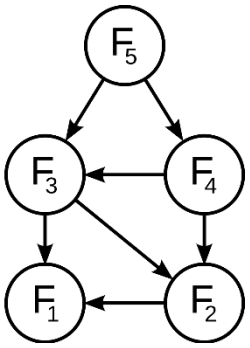


Рис. 10.1. Граф підзадач для чисел Фібоначчі

Оскільки підзадачі для знаходження чисел Фібоначчі утворюють граф, а не дерево, це означає, що вони перекриваються.

Застосування рекурсії в чистому вигляді для задач, що перекриваються призводить до повторного розв'язання одних і тих же підзадач. Щоб уникнути цього, динамічний підхід використовує запам'ятовування вже вирішених підзадач для уникнення їх повторного розв'язання в рекурсії. Такий підхід «згори до низу» в динамічному програмуванні називається «запам'ятовування» (memoization) . Інший підхід «знизу до гори» використовує ітераційний процес і в динамічному програмуванні називається «табуляцією» (tabulation) .

10.1.1. Підхід «згори до низу» (Memoization)

В підході «згори до низу», що також називається *memoization*, розв'язання починається з фінальної задачі і рекурсивно розбивається на менші підзадачі. Для уникнення повторних розрахунків, результати вже розв'язаних підзадач зберігаються в масиві або іншій структурі даних в залежності від конкретної задачі. Такий підхід має переваги в задачах де кількість підзадач велика і вони часто повторюються.

Розглянемо для прикладу задачу знаходження факторіала числа. Реалізуємо функцію, що розраховує факторіал у вигляді рекурсії.

```
int factorial(unsigned int n)
{
    if (n == 0) return 1;
    return n * factorial(n – 1);
}
```

Тепер знайдемо факторіали чисел 2, 3, 9 і 5 використавши написану функцію. В такому випадку наша програма буде складатися з послідовних викликів функції факторіала.

```
factorial(2)
factorial(3)
factorial(9)
factorial(5)
```

Дерево рекурсивних викликів для такої програми буде наступним (рис. 10.2).

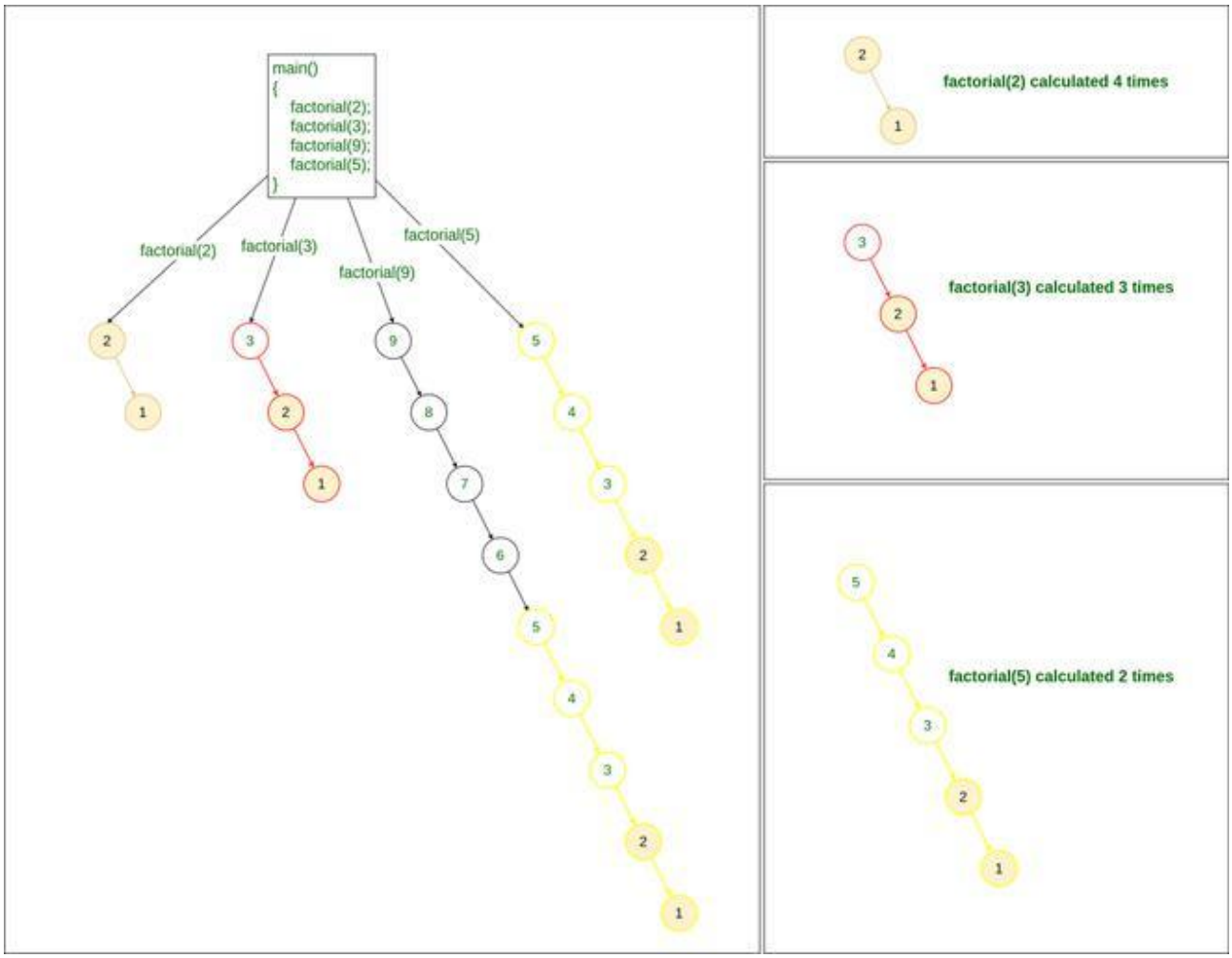


Рис. 10.2. Рекурсивні виклики функції факторіала

Зі схеми вище, стає зрозуміло, що для знаходження факторіалів K чисел, часові витрати будуть наступними:

- O(N) щоб знайти факторіал числа N;
- O(K) для виклику функції `factorial()` K разів.

Тобто в загальному випадку обчислювальна складність буде O(K*N). Але ж ми бачимо на схемі, що деякі підзадачі вирішуються по кілька разів, і, якби їхні результати були збережені, то повторне їх вирішення було б не потрібне, а час на отримання розв'язку був би константним O(1). Саме таким чином і працює «запам'ятовування».

Якщо ми позначимо за N максимальне число для якого нам потрібно буде знайти факторіал, то код функції можна переписати наступним чином.

```
int fact[N+1] = {0};
int factorial(unsigned int n)
{
    if (fact[n] != 0) return fact[n];
    if (n == 0) fact[n] = 1;
    else fact[n] = n * factorial(n – 1);
    return fact[n];
}
```

Така реалізація не буде розв'язувати підзадачі, якщо вони вже мають розв'язок. Застосувавши динамічне програмування ми зменшили обчислювальну складність з O(K*N) до O(N).

10.1.2. Підхід «знизу до гори» (Tabulation)

В підході «знизу до гори», що також відомий як *tabulation*, розв'язання починаються з найдрібнішої підзадачі й продовжується для задач вищих рівнів аж до фінальної задачі. Результати розв'язків підзадач на всіх рівнях зберігаються в таблиці — звідси власне і назва підходу. Такий підхід має переваги в задачах де кількість підзадач невелика і оптимальний розв'язок можна отримати послідовним розв'язанням менших підзадач.

Повернемося до нашого прикладу з розрахунком факторіала числа. У випадку, якщо нам будуть потрібні всі факторіали чисел від 0 до N, ми можемо застосувати табуляцію. В такому випадку функція не буде рекурсивною, а міститиме ітеративний процес, що реалізується циклом.

```
int fact[N+1];
int factorial(void)
{
    fact[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        fact[i] = fact[i-1] * i;
    }
}
```

Таким чином ми витратили час O(N) на розрахунок всіх факторіалів. Для отримання розв'язку для будь-якого числа, що менше або дорівнює N необхідно витратити константний час. Тобто загальна обчислювальна складність буде аналогічною до методу «запам'ятовування».

10.1.3. Порівняння Memoization і Tabulation

Ми вже з'ясували, що обчислювальна складність обох підходів є однаковою. Проте необхідна кількість пам'яті для збереження результатів розв'язання підзадач може бути різною. Підсумкове порівняння цих двох підходів подане в таблиці нижче.

	Tabulation	Memoization
Стани	Перехід між станами важко зрозуміти	Перехід між станами легко зрозуміти
Кодування	Кодування стає складним коли виникає багато умов в задачі	Кодування легке й менш складне
Швидкість	Швидкий, оскільки ми отримаємо прямий доступ до попередніх станів у таблиці	Повільний через можливу велику кількість рекурсивних викликів для попередніх станів
Вирішення підзадач	Якщо всі підзадачі повинні бути вирішені хоча б один раз, то підхід «знизу до гори» зазвичай переважає підхід «згори до низу» на константний час	Якщо не всі підзадачі повинні бути вирішені, тоді підхід «згори до низу» має перевагу, оскільки вирішуються лише ті підзадачі, які дійсно необхідні
Записи в таблиці	Починаються з першого елементу і заповнюються один за одним	Не всі елементи пам'яті заповнюються, а лише ті, що необхідні
Підхід	Ітеративний	Рекурсивний

Як бачимо кожен з підходів має свої переваги й недоліки. Отже який саме підхід обрати залежить від конкретної задачі, яку потрібно вирішити.

Далі ми розглянемо деякі класичні задачі, що розв'язуються методом динамічного програмування.

10.2. Послідовність Фібоначчі

Ми вже розглядали послідовність Фібоначчі в параграфі 8.1.2. Це послідовність, де кожен наступний елемент є сумою двох попередніх. Тепер розглянемо, яким чином можна порахувати число Фібоначчі із заданим індексом. Спочатку розглянемо простий рекурсивний підхід.

```
int fib(int n)
{
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Дерево викликів функції для четвертого елементу послідовності буде мати наступний вигляд (рис. 10.3).

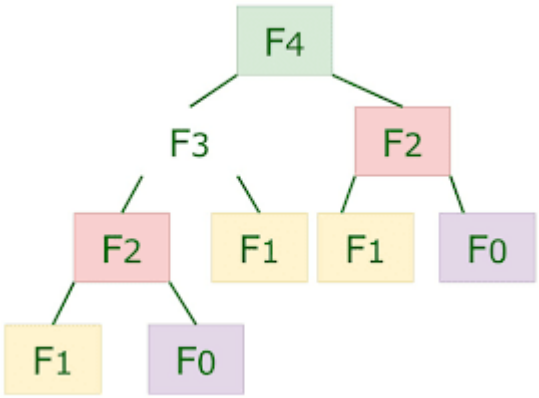


Рис. 10.3. Дерево викликів функції розрахунку чисел Фібоначчі

З рис. 10.3 нам стає зрозуміло, що деякі підзадачі розраховуються більше одного разу. Їх кількість буде зростати зі збільшенням індексу числа Фібоначчі. Такий рекурсивний підхід має обчислювальну складність $O(2^N)$.

Перепишемо рекурсію із застосуванням підходу «згори до низу». Для цього буде потрібен масив розмірністю N+1, де N — індекс числа Фібоначчі. Спочатку ініціалізуємо масив значеннями -1. Це дозволить нам легко перевірити чи було вже розраховане число Фібоначчі з поточним індексом.

```
int f[N+1] = {-1};
int fib(int n)
{   if (n <= 1) return n;
    if (f[n] != -1) return f[n];
    f[n] = fib(n - 1) + fib(n - 2);
    return f[n];
}
```

Рекурсія із «запам'ятовуванням» має обчислювальну складність O(N) та вимагає O(N) пам'яті.

Тепер детальніше поглянемо на задачу і визначимо чи можемо ми застосувати зворотній підхід «знизу до гори». Оскільки ми знаємо значення двох перших елементів послідовності Фібоначчі, то ми можемо послідовно порахувати всі наступні значення. За такого підходу рекурсія замінюється на цикл.

```
int f[N+1];
int fib(void)
{   f[0] = 0; f[1] = 0;
    for (int i = 2; i <= N; i++) f[i] = f[i - 1] + f[i - 2];
    return f[N];
}
```

Підхід «знизу до гори» має обчислювальну складність таку ж, як і підхід «згори до низу». Проте для табуляції є можливість оптимізувати пам'ять. Якщо нам потрібно порахувати число Фібоначчі з конкретним індексом, а всі попередні значення не потрібні, то ми можемо не використовувати масив для зберігання проміжних обчислень. Для розрахунку кожного наступного числа Фібоначчі нам не обхідно пам'ятати лише два попередніх числа. Таким чином ми можемо змінити код реалізації на наступний.

```
int fib(int n)
{
    int prevPrev = 0, prev = 1, curr = 1;
    for (int i = 2; i <= n; i++)
    {
        curr = prev + prevPrev;
        prevPrev = prev;
        prev = curr;
    }
    return curr;
}
```

Остання реалізація виконується за час O(N) та вимагає O(1) пам'яті.

Знаходження чисел Фібоначчі одна з класичних задач, що вирішується методом динамічного програмування. Звичайно для знаходження чисел Фібоначчі з великими індексами краще використати метод множення матриць, але в загальному випадку динамічне програмування доволі ефективний метод.

10.3. Найдовша спільна підпослідовність

Пошук найдовшої спільної підпослідовності (Longest Common Subsequence, LCS) — це завдання пошуку послідовності, яка є підпослідовністю кількох послідовностей (зазвичай — двох). Часто завдання визначається як пошук всіх найбільших спільних підпослідовностей. Ця задача відрізняється від пошуку найдовшого спільного підрядка: на відміну від підрядків, підпослідовності не повинні займати суміжні позиції в оригінальних послідовностях. Це класична задача інформатики, яка має застосування, зокрема, в задачі порівняння текстових файлів, а також у біоінформатиці.

Підпослідовність можна отримати з деякої послідовності, якщо видалити з неї деяку множину елементів (можливо, порожню). Наприклад, BCDB є підпослідовністю послідовності ABCDBAB. Також вона буде підпослідовністю послідовності XBXCDXBX. Послідовність Z є спільною підпослідовність послідовностей X і Y, якщо Z є підпослідовністю як X, так і Y. Потрібно для двох послідовностей X і Y знайти спільну підпослідовність найбільшої довжини. Зауважимо, що таких підпослідовностей може бути кілька.

Якщо вирішувати цю задачу методом повного перебору, то часова складність алгоритму буде $O(2^{\min(n,m)})$, де n та m — розміри першого та другого рядків.

Ця задача має *оптимальну підструктуру* — щоб визначити найдовшу спільну підпослідовність для рядків з довжинами n та m спочатку необхідно вирішити задачу для рядків з довжинами (n-1) та (m-1). Також тут присутні *задачі, що перекриваються*, а отже ми можемо скористатися методом динамічного програмування.

Для вирішення цієї задачі краще застосовувати підхід «знизу до гори». Ми будемо послідовно вирішувати задачу для все більших довжин рядків. Почнемо з нульових довжин, а потім будемо збільшувати довжину другого рядку на 1 з кожним кроком і вирішувати задачу для поточної комбінації довжин. Коли другий рядок досягне максимальної довжини, ми збільшимо на 1 перший рядок, а другий почнемо з нульової довжини. Алгоритм вирішення цієї задачі наступний.

1. Створити матрицю A розмірністю (n+1)*(m+1), де n та m — розміри першого та другого рядків відповідно. Перший рядок та перший стовпчик заповнити нулями.
2. Заповнюємо таблицю рядок за рядком за наступною логікою:
   - A[i][j]=A[i-1][j-1]+1 — якщо i-й символ першого рядка дорівнює j-му символу другого рядка;
   - A[i][j]=max(A[i-1][j],A[i][j-1]) — якщо i-й символ першого рядка не дорівнює j-му символу другого рядка.
3. Повторюємо крок 2 поки не досягнемо кінця таблиці. Значення в останній комірці таблиці буде довжиною найдовшої спільної підпослідовності.

Розглянемо цей метод на прикладі. Нехай у нас є дві послідовності символів — X та Y (рис. 10.4).

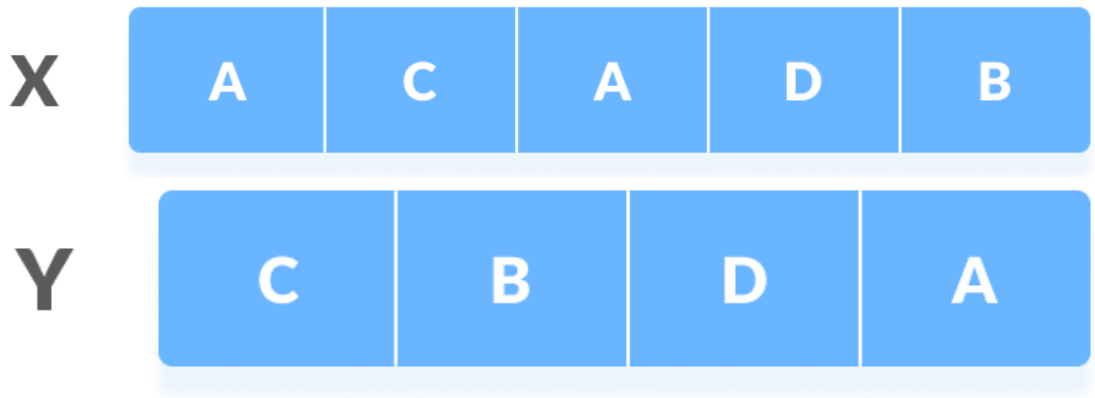


Рис. 10.4. Дві послідовності символів

Крок 1. Створюємо матрицю 6x7 та заповнюємо перший рядок і перший стовпчик нулями (рис. 10.5).

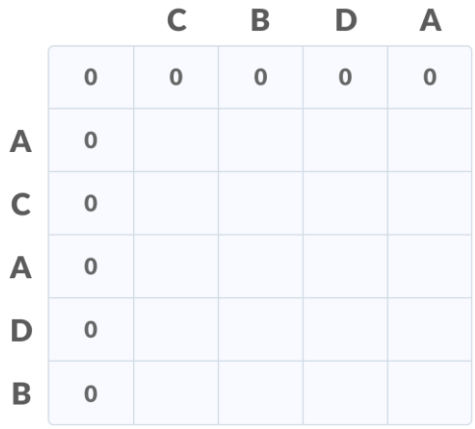


Рис. 10.5. Початкова матриця

Крок 2. Заповнюємо матрицю рядок за рядком. В результаті заповнення першого рядка отримуємо наступну матрицю (рис. 10.6).

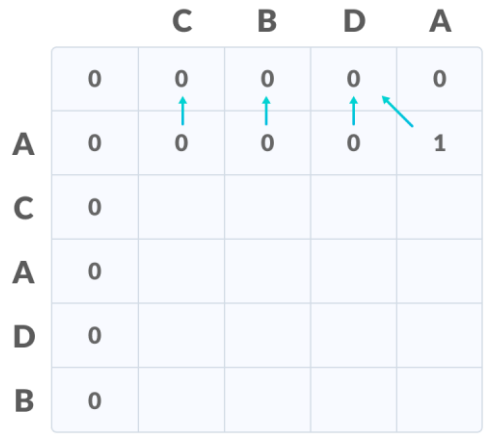


Рис. 10.6. Матриця після заповнення першого рядка

Крок N. Продовжуємо заповнювати матрицю до кінця й отримуємо наступний результат (рис. 10.7).

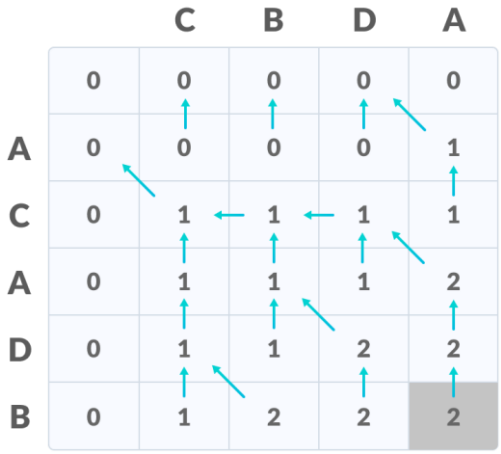


Рис. 10.7. Матриця після закінчення алгоритму

Число в останній клітинці матриці є довжиною найбільшої спільної підпослідовності. Щоб знайти власне цю послідовність, необхідно почати з останньої клітинки і рухатися за стрілками. Таким чином ми отримаємо шлях на якому потрібно виділити комірки з діагональними стрілками, в яких і містяться символи найбільшої спільної підпослідовності (рис. 10.8).

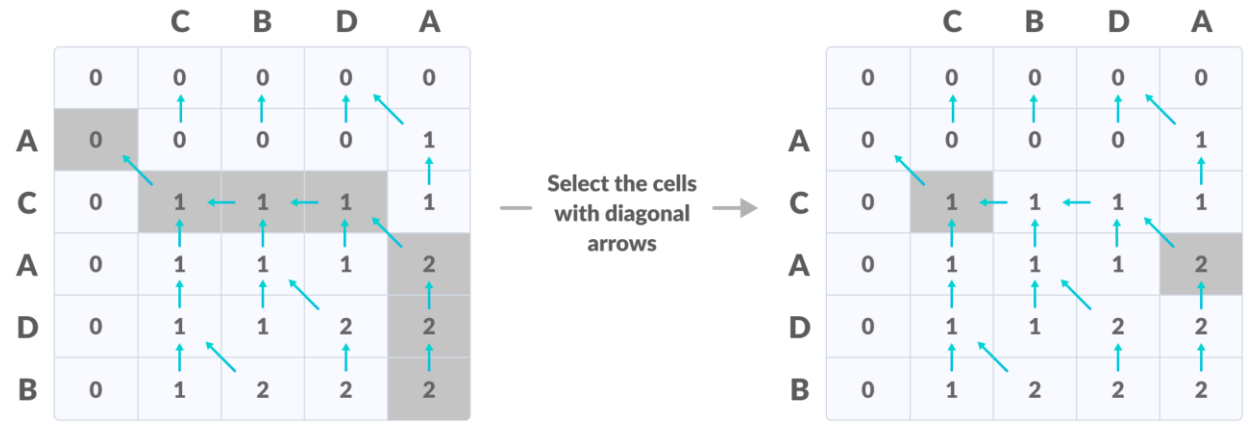


Рис. 10.8. Шлях алгоритму та символи, що складають LSC

Варто зазначити, що в поданому прикладі існує декілька найбільших спільних підпослідовностей: CA, CD і CB. У всіх відповідних клітинках з числом 2 є діагональні стрілки, а отже маючі фінальну матрицю ми можемо знайти всі найбільші підпослідовності шляхом її аналізу.

Часова і просторова складності розглянутого методу динамічного програмування O(n*m). Якщо часову складність покращити вже не можливо, то просторову складність можна оптимізувати, якщо задача стоїть знайти лише довжину LSC. Для обрахунку кожного наступного рядка матриці, нам необхідно мати лише попередній рядок, а тому можна зменшити розмір матриці до 2xn. Але і це ще не всі можливі оптимізації. Насправді з попереднього рядку нам потрібне лише одне число, що є діагональним до поточного. Це число можна зберігати в тимчасовій змінній та оновлювати під час просування алгоритму, а для зберігання значень поточного рядка використовувати лише вектор довжиною n.

10.4. Задача пакування рюкзака

Задача пакування рюкзака (англ. Knapsack problem) — задача комбінаторної оптимізації: для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною. Задача бере свою назву від доволі відомої ситуації, коли потрібно наповнити рюкзак, що здатен витримати деяку максимальну масу, предметами, кожен з яких має вартість (або корисність) та масу. Необхідно обрати об'єкти в такий спосіб, аби максимізувати сумарну вартість (або користь), але не перевищити максимально припустиму

масу. Задачу пакування рюкзака використовують для моделювання різних проблем та у великій кількості промислових задач.

Існує декілька різновидів задачі про рюкзак в залежності від встановлених обмежень. Ми розглянемо класичну задачу, що називається `0-1` задача пакування рюкзака. В цьому різновиді задачі, для кожного предмета може бути лише два варіанти пакування: `1` — предмет запаковано в рюкзак, `0` — якщо не запаковано. Також є обмеження, що кожен предмет подано в одному екземплярі, він має визначену вагу `W` (weight) та визначену цінність `V` (value).

В першому розділі ми вже розглядали приклад цієї задачі та її вирішення жадібним алгоритмом. Тоді ми побачили, що жадібний метод дає приблизний результат, який не завжди є оптимальним.

10.4.1. Метод повного перебору

Спочатку розглянемо як розв'язати цю задачу методом повного перебору. Нехай ми маємо N=3 предмети з такими цінностями v={1, 7, 11} та вагами w={1, 2, 3}. Максимальна ємність рюкзака W=5.

Для розв'язання цієї задачі, нам необхідно знайти всі підмножини предметів, сумарна вага яких буде не більше п'яти. Потім серед всіх цих підмножин необхідно обрати ту, цінність якої буде найбільшою.

Для кожного з предмета буде лише два можливих варіанти: доданий в оптимальну підмножину або не доданий. Це буде нашою *оптимальною підструктурою*.

Алгоритм розв'язку задачі полягає у виборі максимального значення з описаних вище варіантів.

Варіант 1 (`i`-й предмет доданий в оптимальну підмножину): цінність `i`-го предмету v_i плюс максимальне значення отримане від `N-1` предмету, що залишилися з урахуванням залишкової ємності рюкзака ($W-w_i$).

Варіант 2 (`i`-й предмет не доданий в оптимальну підмножину): максимальне значення отримане від `N-1` предмету, що залишилися та ємності рюкзака `W`.

Якщо вага `i`-го предмету w_i більша за максимальну доступну ємність рюкзака `W`, то для такого предмету можливий лише варіант 2.

Дерево викликів рекурсії для такого алгоритму подане на рис. 10.9.

Часова складність такого алгоритму буде складати $O(2^N)$, а просторова — O(N).

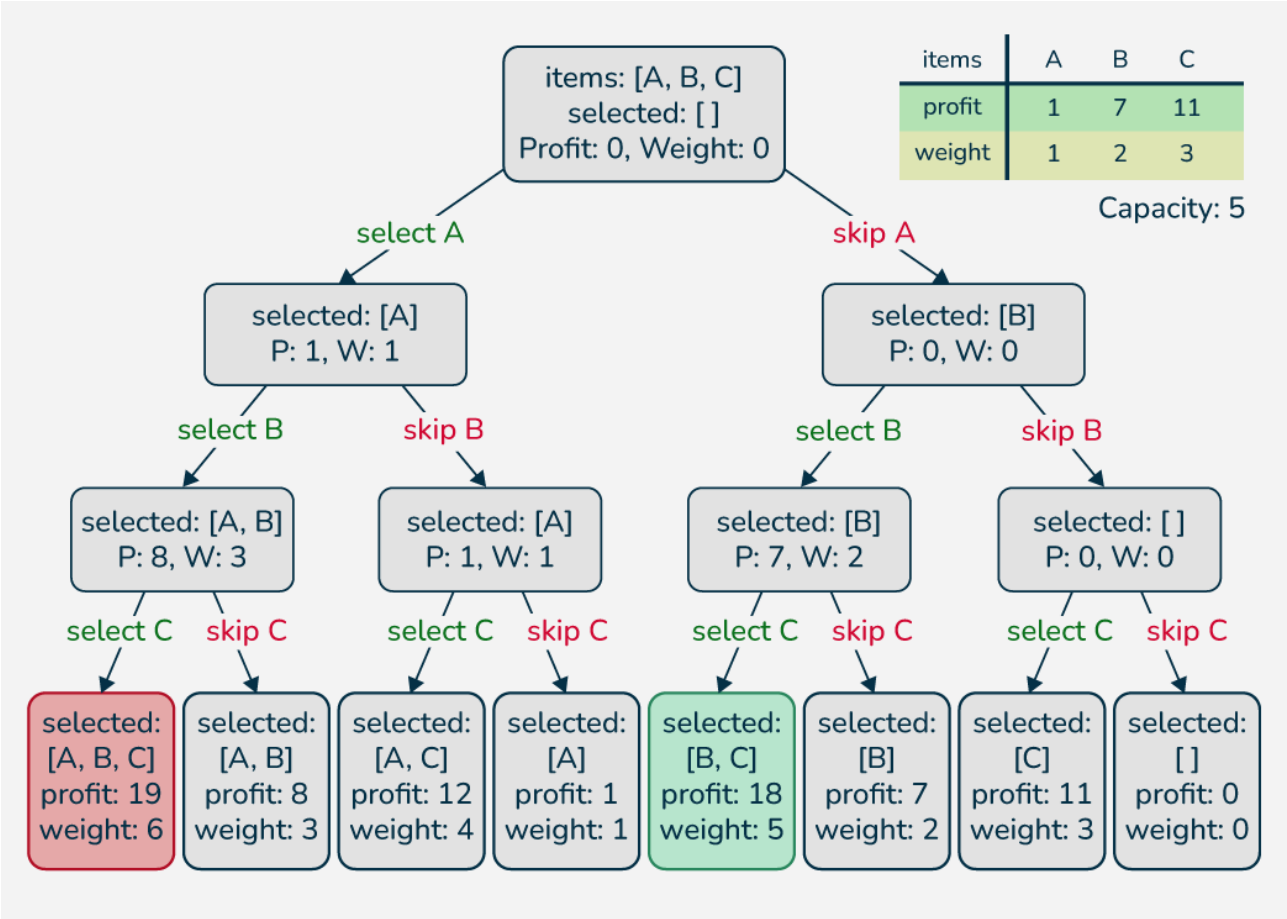


Рис. 10.9. Дерево рекурсії для `0-1` задачі пакування рюкзака

Якщо подивитися на дерево рекурсії, то можна помітити, що одні й ті ж самі підзадачі розв'язуються повторно по декілька разів. Для наочності візьмемо інший набір предметів і покажемо дерево рекурсії для нього (рис. 10.10).

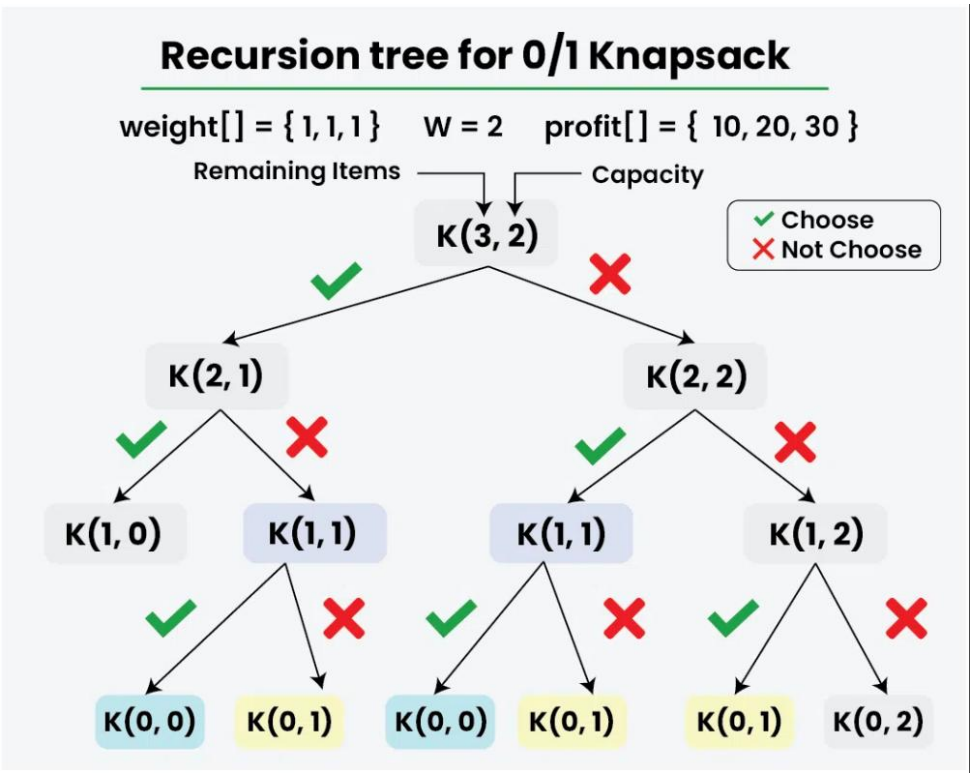


Рис. 10.10. Дерево рекурсії з повторюваними підзадачами

Оскільки в нас є повторювані підзадачі, то ми маємо другу ознаку задачі, що може бути розв'язана методом динамічного програмування, а саме *задачі, що перекриваються*.

Тепер ми можемо застосувати будь-який з двох підходів динамічного програмування: *memoization* або *tabulation*. Обидва методи мають однакову часову складність і майже однакову просторову складність. Проте, метод «знизу до гори» більш наочно демонструє метод динамічного програмування, а також має перспективу до оптимізації використаної пам'яті. Саме тому ми розглянемо цей підхід для вирішення задачі пакування рюкзака.

10.4.2. Метод «знизу до гори»

Метод «знизу до гори» вимагає послідовного вирішення всіх підзадач, що виникають в основній задачі. В нашому випадку, ми спочатку повинні вирішити задачу для нульової ємності рюкзака. Далі для одиничної ємності, потім для ємності 2 і так далі до максимальної ємності W. Для кожної ємності ми послідовно розглядаємо можливість додавання нуля предметів, тоді першого, другого і так далі до останнього предмета. Всі розрахунки зручно тримати в двовимірному масиві, що описує таблицю розв'язків (T).

Розглянемо приклад, де N=3, W=6, w={1, 2, 3}, v={10, 15, 40}. Будемо йти крок за кроком й заповнювати таблицю. Очевидно, що для нульової кількості елементів всі розв'язки будуть нулями (рис. 10.11).

		Вага						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Предмети	**0**	0	0	0	0	0	0	0
	1							
	2							
	3							

Рис. 10.11. Початкова таблиця розв'язків

Тепер розглянемо перший предмет і вирішимо задачі для рюкзаків різної ємності від 0 до 6. Цінність першого предмета складає 10, а вага 1. В рюкзак з ємністю нуль ми не можемо покласти предмет вагою 1, а отже загальна цінність предметів для рюкзака з ємністю 0 залишається нульовою. В рюкзаки з ємностями від 1 до 6 включно ми можемо покласти перший предмет, а отже загальна цінність для таких рюкзаків буде дорівнювати цінності першого предмета, тобто 10 (рис. 10.12).

		Вага						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Предмети	**0**	0	0	0	0	0	0	0
	1	0	10	10	10	10	10	10
	2							
	3							

Рис. 10.12. Таблиця після обробки першого предмета

Наступним кроком розглянемо предмет номер 2. Його вага w=2, а цінність v=15. Для рюкзака ємністю 0, цінність залишається нульовою. Для рюкзака ємністю 1 ми не можемо додати другий предмет, оскільки його вага більша за ємність рюкзака. Отже ми не додаємо другий предмет до рюкзака ємністю 1 і розглядаємо максимальну цінність рюкзака коли в нього додається перший предмет. Ця задача в нас вже вирішена і ми знаємо, що відповідь 10.

Тепер пробуємо вирішити задачу для рюкзака ємністю 2. Оскільки другий предмет може бути доданий в такий рюкзак, то нам необхідно обрати максимальне значення між двома варіантами: якщо додати предмет 2 до рюкзака і якщо не додати. Якщо не додати, то розв'язок буде 10, це ми вже знаємо. Якщо ж додати, то розв'язком буде цінність предмета 2, що дорівнює 15 плюс розв'язок для залишкової ємності рюкзака в яку намагаються додати попередні предмети. Залишкова ємність рюкзака після додавання предмета 2 буде (2-2=0), а розв'язок для нульового рюкзака і першого предмету теж нуль. Якщо записати вираз в використанням таблиці, то отримаємо наступне:

$$\max(T[1][2], 15+T[1][2-2])=\max(10, 15+0)=15$$

Отже розв'язком задачі для рюкзака ємністю 2 і двома предметами буде 15.

Тепер розв'яжемо задачу для рюкзака ємністю 3. Знову у нас два варіанти: додавати предмет 2 або не додавати. Якщо додамо, то залишкова вага рюкзака буде 1 і для цієї ваги в нас є розв'язок для предмета 1. Вираз для розв'язку з використанням таблиці буде наступним:

$$\max(T[1][3], 15+T[1][3-2])=\max(10, 15+10)=25$$

Отже розв'язком задачі для рюкзака ємністю 3 і двома предметами буде 25. Для рюкзаків більшої ємності і тих самих двох перших предметів розв'язок залишиться незмінним (рис. 10.13).

		Вага						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Предмети	**0**	0	0	0	0	0	0	0
	1	0	10	10	10	10	10	10
	2	0	10	15	25	25	25	25
	3							

Рис. 10.13. Таблиця після обробки другого предмета

Останнім кроком розглянемо предмет номер 3. Його вага w=3, а цінність v=40. Для рюкзаків вагою менше за 3 розв'язки залишаться незмінними, оскільки третій предмет в такі рюкзаки додати неможливо. Для рюкзака з ємністю 3 необхідно перевірити обидва варіанти і обрати максимальний. Вираз для розв'язку з використанням таблиці буде наступним:

$$max(T[2][3], 40+T[2][3-3])=max(25, 40+0)=40$$

Отже розв'язком буде додавання лише третього предмета до рюкзака, що зробить його цінність максимальною.

Тепер розв'яжемо задачу для рюкзака з ємністю 4. Вираз для розв'язку з використанням таблиці буде наступним:

$$max(T[2][4], 40+T[2][4-3])=max(25, 40+10)=50$$

Як бачимо, нам варто взяти перший і третій предмет, щоб цінність рюкзака була максимальною.

Розв'язок для рюкзака з ємністю 5 буде наступним:

$$max(T[2][5], 40+T[2][5-3])=max(25, 40+15)=55$$

Для такого рюкзака оптимальним вибором будуть предмети 2 і 3, що дасть нам максимальну цінність 55.

Останній розв'язок буде для рюкзака ємністю 6, як і написано в умові завдання:

$$max(T[2][6], 40+T[2][6-3])=max(25, 40+25)=65$$

Остаточний вигляд таблиці після проведення всіх розрахунків буде наступним (рис. 10.14).

Часова складність цього методу складає O(N*W). Це вже значно краще, ніж в методу прямого перебору. Просторова складність також складає O(N*W), але це гірше, ніж в методі прямого перебору. І, хоча, час виконання є важливішим, все ж хотілося б мати просторову складність близьку до методу прямого перебору.

		Вага						
		0	**1**	**2**	**3**	**4**	**5**	**6**
Предмети	**0**	0	0	0	0	0	0	0
	1	0	10	10	10	10	10	10
	2	0	10	15	25	25	25	25
	3	0	10	15	40	50	55	65

Рис. 10.14. Таблиця після обробки третього предмета

Якщо уважно проаналізувати вирази, що використовуються для вирішення задачі пакування рюкзаків для кожного наступного предмету, то можна помітити, що у формулах використовується лише один попередній рядок таблиці. Подібне ми вже бачили під час розгляду задачі LCS. Там ми побачили, що потрібно додатково пам'ятати лише одне число з попереднього рядка і тоді можна використовувати лише одновимірний масив. В задачі про пакування рюкзака індекс числа з попереднього рядка залежить від ваги поточного предмета, що розглядається і він завжди менший за поточний індекс стовпчика, що позначає ємність рюкзака. Отже ми можемо змінити напрямок заповнення таблиці та використовувати всього лише один масив довжиною W+1. Таким чином просторова складність для методу «знизу до гори» складе O(W).

Розглянемо крок за кроком, як це відбувається. Почнемо з розв'язків для предмета 2. Початковий масив значень буде містити розв'язки для предмета 1. Далі будемо рухатися від ємності рюкзака 6 до ємності рюкзака 0 (рис. 10.15).

0	**1**	**2**	**3**	**4**	**5**	**6**
0	10	10	10	10	10	10
0	10	10	10	***10***	10	**25**
0	10	10	***10***	10	**25**	25
0	10	***10***	10	**25**	25	25
0	***10***	10	**25**	25	25	25
0	10	**15**	25	25	25	25

Рис. 10.15. Вектор в процесі обробки другого предмета

Тепер розглянемо формування вектора розв'язків для предмета 3. Початковий масив значень буде містити розв'язки для предмета 2 (рис. 10.16).

0	**1**	**2**	**3**	**4**	**5**	**6**
0	10	15	25	25	25	25
0	10	15	***25***	25	25	**65**
0	10	***15***	25	25	**55**	65
0	***10***	15	25	**50**	55	65
0	10	15	**40**	50	55	65

Рис. 10.16. Вектор в процесі обробки третього предмета

В переліку посилань на використані джерела ви знайдете візуалізацію заповнення таблиці за посиланням під номером 12.

10.5. Задачі з числами

Існує клас задач з цілими числами, що вирішуються методом динамічного програмування. В англомовній літературі метод динамічного програмування для їх вирішення називається Digit Dynamic Programing (Digit DP). До цього класу задач відносяться задачі підрахунку суми цифр у числах між двома вказаними цілими числами, а також підрахунку кількості чисел, що менші за задане і мають певну ознаку (наприклад визначену суму цифр, визначений добуток цифр або визначені цифри).

Ключова ідея цього методу в перетворенні числа на масив з N цифр, де найбільш значуща цифра буде мати індекс N-1, а цифра в позиції одиниць — індекс 0. Після цього ми генеруємо числа з кількістю цифр 1, потім з кількістю цифр 2 і так до кількості цифр N.

Розглянемо приклад де нам потрібно знайти суму всіх цифр чисел, що менші або дорівнюють визначеному числу. Число лежить в межах $[1; 10^{18}]$. Якщо число буде, наприклад 11, то сума всіх цифр у числах, що менші або дорівнюють йому буде розрахована наступним чином (1+2+3+4+5+6+7+8+9+1+0+1+1=48). Щоб порахувати розв'язок для трицифрового числа, нам знадобиться порахувати суми цифр у всіх двоцифрових і одноцифрових числах. Це означає, що в нас є *задачі, що перекриваються* і замість того, щоб заново їх розв'язувати, ми можемо використати результати попередніх розрахунків. Для цього найкраще підійде метод динамічного програмування «згори до низу» (memoization). Основне, що залишилося, це визначити структуру масиву для збереження результатів підзадач.

Щоб порахувати суму цифр в числі нам потрібно знати зі скількох цифр складається власне це число. Оскільки ми будемо розглядати всі числа, що складаються з кількості цифр від 1 до N (де N це кількість цифр в максимальному числі), то для нас важливо знати поточну розрядність числа. Оскільки ми перетворюємо число на масив, то нам потрібно знати індекс цифри — idx. Для одноцифрових чисел індекс буде 0, для двоцифрових — 1 і так далі. Отже індекс це одне зі значень для якого потрібно зберігати розрахунки.

Тепер поміркуємо, яку суму цифр може мати число, що складається з однієї цифри. Очевидно, що це може бути 0 або 1 або 2 і так до 9. Тобто 10 варіантів. Тепер подумаємо, які суми цифр може мати двоцифрове число. Якщо ми розглядаємо двоцифрове число як фрагмент трицифрового або більшого числа, то в нас можуть бути такі числа: 0+0=0, 0+1=1, 0+2=2, … , 1+0=1, 1+1=2, … , 9+8=17, 9+9=18. Як бачимо в нас є 19 варіантів сум — від 0 до 18. Варто відзначити, що деякі варіанти сум повторюються декілька разів. Наприклад варіанти 0+3, 3+0, 1+2, 2+1 всі дають суму 3. Для числа з 18 цифр максимальне значення суми цифр буде 18*9=162, а це означає, що для зберігання кількості всіх можливих сум нам потрібен масив на 163 елементи.

Таким чином ми зрозуміли, що для кожної цифри в числі, яких може бути 18 в нашому прикладі, ми можемо отримати суму, що не буде перевищувати число 162. Насправді для кожного індексу цифри буде свій максимум, але для цифри з індексом 17 максимальна сума буде 162, а тому ми просто можемо створити матрицю розмірністю 18x163, щоб гарантовано мати можливість зберігати кількості всіх можливих сум.

Та насправді індекс цифри та кількість сум цифр це ще не вся достатня інформація для опису підзадачі. Коли ми формуємо число з молодших цифр в поточній позиції цифри може бути дві ситуації: в цій можна використовувати всі цифри від 0 до 9, або лише цифри від 0 до digit[idx], тобто до значення цифри в цій позиції. Наприклад для числа 324, коли ми генеруємо числа, що починаються з цифр 0**, 1** або 2** в сотнях, то значення десятків і одиниць може бути від 0 до 9. Але коли ми формуємо числа, що починаються з цифри 3** в сотнях, тоді в десятках у нас є обмеження на можливі цифри в діапазоні від 0 до 2. Так само якщо число починається з цифр 32* в сотнях і десятках, тоді в одиницях в нас є обмеження на використання цифр в діапазоні від 0 до 4. Таким чином для кожного індексу числа ми можемо порахувати кількість сум цифр в менших числах для двох випадків: коли цифра має обмеження або коли не має. Англійською мовою це обмеження називають tight і це булеве значення що набуває двох станів 0 або 1. Зважаючи на присутність tight нам потрібно мати дві матриці 18x163 для варіанту з обмеженням (tight=1) і варіанту без обмеження (tight=0).

Тепер розглянемо, яким чином пов'язані вирішення підзадач з вирішенням всієї задачі. Щоб знайти суму цифр чисел менших за задане число, що складається з N цифр, нам потрібно вирішити задачу для числа з N-1 цифр. Але таких задач

буде не одна. Їхня кількість залежить від найбільш значущого числа. Наприклад для трицифрового числа максимальний індекс буде 2. Якщо це число 324, то ми повинні порахувати всі варіанти для цифр 0, 1, 2, 3. Тобто спочатку зафіксувати цифру 0, вирішити підзадачу для всіх двозначних чисел та додати цифру 0 до цього вирішення. Потім зафіксувати цифру 1 і вирішити задачу для всіх двозначних чисел і додати цифру 1. Потім зафіксувати цифру 2 і знову вирішити для всіх двозначних чисел та додати 2. Так само зробити і для цифри 3. Як бачите ми чотири рази вирішували одну й ту ж підзадачу, але оскільки ми використовуємо memoization, то попередньо пораховані результати вже будуть в таблиці і не будуть перераховуватися заново. Приблизний код буде таким:

```
int ans = 0;
for (int i=0; i<=k; i++)
{
   ans += state(idx-1, newTight, sum+i)
}
state(idx,tight,sum) = ans;
```

В цьому прикладі для числа 324, для індексу `idx=2` значення `tight=1` оскільки не всі цифри можна використовувати. В той же час, коли ми чотири рази вирішували задачу для двоцифрового числа, то у випадку цифр 0, 1, 2 в сотнях, значення `tight` для `idx=1` було 0, оскільки в десятках ми могли перебирати всі цифри. А от коли значення сотень стало 3, то для десятків значення `tight` теж стає 1, оскільки лише цифри 0, 1, 2 дозволені. Тобто `tight` для числа з `N-1` цифр залежить від попереднього значення і значення поточної цифри:

```
newTight = previousTight & (digitTake == digit[idx])
```

Повний код реалізації цього методу ви можете знайти за посиланням з пункті 14 списку використаних джерел.

Для засвоєння цього методу розглянемо приклад з покроковою візуалізацією результатів.

Приклад. Необхідно порахувати суму всіх цифр у числах, що менші або дорівнюють 24.

Спочатку визначимо максимальні значення для основних параметрів задачі. Кількість цифр в числі N=2. Теоретично максимально можлива сума цифр в числі 9*N=18, але практично для цього прикладу це буде число 10. Отже нам потрібно створити дві таблиці розмірністю 2x11 — одна для `tight=1` та інша для `tight=0`. Спочатку обидві таблиці заповнені значеннями `-1`. Початковий індекс цифри буде `idx=1`, початкова сума `sum=0` і початкове обмеження `tight=1`. Запускаємо рекурсивну функцію з цими початковими параметрами і вона починає перебирати всі можливі значення цифр для поточного індексу: `digitSum(idx=1, sum=0, tight=1)`.

Індекс цифри 1. Число (0X**).** Рахуємо обмеження для передачі в наступну ітерацію рекурсії. Оскільки поточне значення цифри 0 не дорівнює оригінальному значенню 2, то tight=0. Викликаємо рекурсивну функцію для індексу 0, передаємо туди поточну суму плюс 0, та нове пораховане обмеження: digitSum(idx=0, sum=0, tight=0).

Індекс цифри 0. Число (00**).** Оскільки обмеження для цифри з індексом 1 дорівнює нулю, то і для поточної цифри воно буде нульовим. Викликаємо рекурсивну функцію для індексу -1, передаємо туди поточну суму плюс число 0, та нове пораховане обмеження: digitSum(idx=-1, sum=0, tight=0).

Індекс цифри -1. Число (00**).** Індекс менше нуля, отже просто повертаємо передану суму і виходимо з рекурсії на попередній рівень. Поки ніяких змін в таблицях не відбулося.

Індекс цифри 0. Число (01**).** Зараз ми знову на рівні рекурсії, де перевіряємо цифру в одиницях числа. Тобто всі вхідні значення аналогічні, коли ми перевіряли цифру нуль в цьому індексі. Знову викликаємо рекурсивну функцію для індексу -1 та передаємо туди поточну суму плюс 1: digitSum(idx=-1, sum=1, tight=0).

Індекс цифри -1. Число (01**).** Індекс менше нуля, отже просто повертаємо передану суму і виходимо з рекурсії на попередній рівень. Поточний результат функції стає 1.

Індекс цифри 0. Число (02**).** Тепер ми рахуємо число з цифрою 2 в одиницях. Поточна сума зараз 0, а тому нова сума буде 2. Інші параметри залишаються незмінними. Викликаємо функцію для попереднього індексу -1: digitSum(idx=-1, sum=2, tight=0).

Індекс цифри -1. Число (02**).** Індекс менше нуля, отже просто повертаємо передану суму і виходимо з рекурсії на попередній рівень. Поточний результат функції стає 1+2=3.

Таким чином виконуються рекурсивні виклики для всіх десяти цифр в позиції одиниць та отримується результат суми всіх чисел, що менші 10. Він дорівнює 45. Оскільки вхідна сума була 0, індекс цифри 0, та обмеження 0, то записуємо результат роботи функції у відповідну комірку таблиці (рис. 10.17).

1	0	0	0	0	0	0	0	0	0	0	0	**tight=0**
0	45	0	0	0	0	0	0	0	0	0	0	
idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**sum**
0	0	0	0	0	0	0	0	0	0	0	0	**tight=1**
1	0	0	0	0	0	0	0	0	0	0	0	

Рис. 10.17. Таблиці результатів після виконання всіх ітерацій для чисел 0X

Індекс цифри 1. Число (1X**).** Тепер ми повертаємося ще на один рівень рекурсії вище. Поточне значення результату функції на цьому рівні рекурсії в нас стає 45.

Знову змінюємо цифру вже в позиції десятків. Наступна цифра в нас 1, а це менше за 2, а тому `tight=0`. Викликаємо рекурсивну функцію для індексу `0`, передаємо туди поточну суму плюс 1, та нове пораховане обмеження: `digitSum(idx=0, sum=1, tight=0)`.

Індекс цифри 0. Число (10**).** Оскільки обмеження для цифри з індексом 1 дорівнює нулю, то і для поточної цифри воно буде нульовим. Викликаємо рекурсивну функцію для індексу `-1`, передаємо туди поточну суму плюс число `0`, та нове пораховане обмеження: `digitSum(idx=-1, sum=1, tight=0)`.

Індекс цифри -1. Число (10**).** Індекс менше нуля, отже просто повертаємо передану суму і виходимо з рекурсії на попередній рівень. Поточний результат функції стає 1.

Як бачимо, для цифри `0` в позиції одиниць ми тепер отримали суму цифр 1. Це сталося тому, що врахована цифра в десятках, а вона зараз 1. Кожна ітерація рекурсії буде повертати суму, що буде на одиницю більшою, за аналогічні виклики функції коли в десятках стояв `0`. В результаті викликів `10` ітерацій ми отримаємо суму цифр у всіх числах, що більші за `10` але менші за `20`. Ця сума буде дорівнювати 55. Оскільки вхідна сума була 1, індекс цифри `0`, та обмеження `0`, то записуємо результат роботи функції у відповідну комірку (рис. 10.18).

1	0	0	0	0	0	0	0	0	0	0	0	**tight=0**
0	45	55	0	0	0	0	0	0	0	0	0	
idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**sum**
0	0	0	0	0	0	0	0	0	0	0	0	**tight=1**
1	0	0	0	0	0	0	0	0	0	0	0	

Рис. 10.18. Таблиці результатів після виконання всіх ітерацій для чисел 1X

Індекс цифри 1. Число (2X**).** Знову повертаємося ще на один рівень рекурсії вище. Поточне значення результату функції на цьому рівні рекурсії в нас стає `45+55=100`. Знову змінюємо цифру вже в позиції десятків. Наступна цифра в нас 2, а це дорівнює поточній цифрі в числі, а тому `tight=1`. Викликаємо рекурсивну функцію для індексу `0`, передаємо туди поточну суму плюс 2, та нове пораховане обмеження: `digitSum(idx=0, sum=2, tight=1)`.

Індекс цифри 0. Число (20**).** Оскільки обмеження для цифри з індексом 1 дорівнює один, то для поточної цифри потрібно зважати на нього. Ми не можемо викликати нижній рівень рекурсії для цифр, що більші за 4, адже 4 це максимальне допустиме значення одиниць для початкового числа, коли в десятках стоїть 2. Наступне обмеження для цифр менше 4 в одиницях буде `0`, а для цифри 4 воно буде 1. Викликаємо рекурсивну функцію для індексу `-1`, передаємо туди поточну суму плюс число `0`, та нове пораховане обмеження: `digitSum(idx=-1, sum=2, tight=0)`.

Індекс цифри -1. Число (20**).** Індекс менше нуля, отже просто повертаємо передану суму і виходимо з рекурсії на попередній рівень. Поточний результат функції стає 2.

Повторюємо аналогічні виклики для цифр 1, 2, 3 і 4. Для останньої обмеження буде 1, але в цьому випадку це не впливає на загальний результат. В результаті викликів 5 ітерацій ми отримаємо суму цифр у всіх числах, що більші за 20 але менші за 25. Ця сума буде дорівнювати 10. Оскільки вхідна сума була 2, індекс цифри 0, та обмеження 1, то записуємо результат роботи функції у відповідну комірку таблиці (рис. 10.19).

1	0	0	0	0	0	0	0	0	0	0	0	**tight=0**
0	45	55	0	0	0	0	0	0	0	0	0	
idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**sum**
0	0	0	10	0	0	0	0	0	0	0	0	**tight=1**
1	0	0	0	0	0	0	0	0	0	0	0	

Рис. 10.19. Таблиці результатів після виконання всіх ітерацій для чисел 2X

Індекс цифри 1. Число (3X**).** Знову повертаємося ще на один рівень рекурсії вище. Поточне значення результату функції на цьому рівні рекурсії в нас стає 45+55+10=110. Оскільки наступна цифра для десятків має бути 3, а вона більша за початкову цифру 2, то ми закінчуємо роботу функції на цьому рівні та записуємо результат роботи функції в таблицю (рис. 10.20). Нагадаємо, що початковий виклик функції був з такими параметрами: `digitSum(idx=1, sum=0, tight=1)`.

1	0	0	0	0	0	0	0	0	0	0	0	**tight=0**
0	45	55	0	0	0	0	0	0	0	0	0	
idx	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**sum**
0	0	0	10	0	0	0	0	0	0	0	0	**tight=1**
1	110	0	0	0	0	0	0	0	0	0	0	

Рис. 10.20. Таблиці результатів після виконання всіх ітерацій для числа 24

Звичайно в такому короткому прикладі в нас не виникала ситуація, коли ми використовували попередньо розраховані результати. Це буде помітно вже для трицифрових і більших чисел. Також у випадку необхідності порахувати суму цифр для декількох різних чисел, попередньо розраховані результати будуть використовуватися. Наприклад, якщо буде потрібно порахувати суму цифр в числі 32, ми вже матимемо розв'язок для рекурсій 0X і 1X. Залишиться порахувати 2X та 30, 31, 32. Зверніть увагу, що 2X в нас не пораховано. В нас є лише варіант з обмеженням коли ми порахували числа 20, 21, 22, 23 і 24. Саме тому в нас є дві таблиці — одна для повного набору цифр від 0 до 9, а інша для обмеженого набору цифр, який визначається конкретним значенням початкового числа.

10.6. Приклади завдань для тренування

Для тренування практичних навичок програмування потрібно розв'язати декілька різних задач, що використовують метод динамічного програмування.

Ось перелік задач на платформі LeetCode різного рівня складності, які можуть бути розв'язані з використанням динамічного програмування.

Низький рівень складності

1. Climbing Stairs https://leetcode.com/problems/climbing-stairs.
2. Pascal's Triangle https://leetcode.com/problems/pascals-triangle.
3. Counting Bits https://leetcode.com/problems/counting-bits.
4. Min Cost Climbing Stairs https://leetcode.com/problems/longest-increasing-subsequence.
5. Divisor Game https://leetcode.com/problems/divisor-game.

Середній рівень складності

1. Jump Game https://leetcode.com/problems/jump-game.
2. Decode Ways https://leetcode.com/problems/decode-ways.
3. Triangle https://leetcode.com/problems/triangle.
4. Longest Increasing Subsequence https://leetcode.com/problems/longest-increasing-subsequence.
5. Coin Change https://leetcode.com/problems/coin-change.

Високий рівень складності

1. Frog Jump https://leetcode.com/problems/frog-jump.
2. Race Car https://leetcode.com/problems/race-car.
3. Maximum Value of K Coins from Piles https://leetcode.com/problems/maximum-value-of-k-coins-from-piles.
4. Numbers at Most N Given Digit Set https://leetcode.com/problems/numbers-at-most-n-given-digit-set.
5. Number of Digit one https://leetcode.com/problems/number-of-digit-one.

Запитання для самоконтролю

1. Дайте визначення поняттю «динамічне програмування».
2. Яким умовам повинна відповідати задача, щоб її можна було розв'язати за допомогою динамічного методу?
3. На які два підходи поділяється динамічний метод?
4. Які переваги динамічного підходу над рекурсивним?
5. Які класичні задачі, що можуть бути розв'язані за допомогою динамічного програмування, ви знаєте?

ПОСИЛАННЯ НА ВИКОРИСТАНІ ДЖЕРЕЛА

1. https://uk.wikipedia.org/wiki/Динамічне_програмування.
2. https://foxminded.ua/metod-dynamichnoho-prohramuvannia.
3. https://www.geeksforgeeks.org/dynamic-programming.
4. https://www.programiz.com/dsa/dynamic-programming.
5. https://www.programiz.com/dsa/longest-common-subsequence.
6. https://www.geeksforgeeks.org/longest-common-subsequence-dp-4.
7. https://uk.wikipedia.org/wiki/Пошук_найбільшої_спільної_підпослідовності.
8. https://www.baeldung.com/cs/knapsack-problem-np-completeness.
9. https://www.javatpoint.com/0-1-knapsack-problem.
10. https://en.wikipedia.org/wiki/Knapsack_problem.
11. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10.
12. https://www.w3schools.com/dsa/dsa_ref_knapsack.php.
13. https://uk.wikipedia.org/wiki/Задача_пакування_рюкзака.
14. https://www.geeksforgeeks.org/digit-dp-introduction.

СПИСОК ДЖЕРЕЛ

1. Бхаргава А. Ґрокаємо алгоритми: Ілюстрований посібник для програмістів і допитливих. Київ: ArtHuss, 2023. 256 с. ISBN 978-617-8025-57-1.
2. Кучма М.І. Математичне програмування: приклади і задачі. Львів: Новий світ-2000, 2020. 244 с. ISBN 966-7827-98-4.
3. Городня Т.А., Щебрак А.Ф., Бех О.В. Математичне програмування. Львів: Магнолія 2006, 2021. 200 с.
4. https://www.geeksforgeeks.org
5. https://leetcode.com
6. https://www.hackerearth.com
7. https://uk.wikipedia.org
8. https://wikipedia.org
9. https://devzone.org.ua
10. https://www.programiz.com
11. https://algoua.com
12. https://www.mathros.net.ua
13. https://www.tutorialspoint.com

Р

С

Т

Ф

Х

Ч

Ш

Навчальне видання

Бородавка Євгеній Володимирович

Терентьєв Олександр Олександрович

Гончаренко Тетяна Андріївна

ДИНАМІЧНЕ ПРОГРАМУВАННЯ

Підручник