МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД

«УЖГОРОДСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГIЙ

Кафедра інформаційних управляючих систем та технологій

ПАРАСКА БОГДАН ВОЛОДИМИРОВИЧ

# Дослідження методів навчання нейромереж у задачах прогнозування цін

122 «Комп’ютерні науки»

Дипломна робота на здобуття освітнього ступеня

бакалавра

Науковий керівник:

Ніколенко Володимир Володимирович доктор технічних наук, професор

Ужгород – 2024

# Реєстрація

(номер)

« » червня 2025 р. Копча-Горячкіна Г. Е.

(підпис)

# Дипломна робота допущена до захисту

Завідувач кафедри

Міца О.В.

(підпис)

доктор технічних наук, професор

« » червня 2024 р.

Рецензент Машталір С.В.

(підпис)

доктор технічних наук,

професор Харківського національного

університету радіоелектроніки

**ПІБ:** Меренич Василь Васильович

**Назва:** Оптимізації в динамічному програмуванні та їх реалізації

**Факультет:** Інформаційних технологій

**Спеціальність:** 122 «Комп’ютерні науки»

**Науковий керівник:** д.т.н., проф. Міца О.В.

За темою роботи опубліковано 1 статтю.

## Анотація

У цій дипломній роботі було досліджено розв’язки, які базуються на методі динамічного програмування та проведено аналіз його можливих оптимізацій. Було реалізовано розв’язки задач мовою C++ з використанням цих оптимізацій.

Робота складається з двох розділів. У першому розділі описано загальні відомості про метод динамічного програмування, наведено деякі класичні задачі, які вирішуються цим методом, та їх розв’язки. У другому розділі описані оптимізація Кнута-Яо, [“Розділяй та володарюй”](#_mpwkk8edro9p), [Convex hull trick](#_f4vi4ckuoxwk) та дерево Лі-Чао, та оптимізація за допомогою використання певних структур даних. Наведено задачі з ресурсів Codeforces, AtCoder і CSES, та їх рішення, які використовують ці методи для покращення швидкості роботи. Проведено порівняльний аналіз їх часової складності.

Дана дипломна робота містить: сторінок — 52, розділів — 2, рисунків — 43, кількість використаних джерел — 8.

Ключові слова: динамічне програмування, часова складність алгоритму, оптимізація Кнута-Яо, [“Розділяй та володарюй”](#_mpwkk8edro9p), [Convex hull trick](#_f4vi4ckuoxwk), дерево Лі-Чао, дерево відрізків.

ЗМІСТ

[ВСТУП 3](#_3lgk1rprkjvf)

[РОЗДІЛ 1](#_m139y5ua5nkr). [ОГЛЯД МЕТОДУ ДИНАМІЧНОГО ПРОГРАМУВАННЯ 5](#_yv7ogtolvu12)

[1.1. Визначення та основні концепції 5](#_vnzg5msdqf7e)

[1.2. Класичні задачі, які вирішуються за допомогою динамічного програмування 7](#_keuaos8r8gh3)

[РОЗДІЛ 2](#_yvzprrmsayc5). [МЕТОДИ ОПТИМІЗАЦІЙ ДИНАМІЧНОГО ПРОГРАМУВАННЯ 11](#_hyd59c40ljms)

[2.1. Оптимізація за допомогою використання певних структур даних 11](#_kugszjb8465)

[2.2. Оптимізація Кнута - Яо 16](#_eh6z1a1ssw62)

[2.3. Оптимізація “Розділяй та володарюй” 23](#_mpwkk8edro9p)

[2.4. Нерівність чотирикутника 28](#_lvjn5nlhrvxq)

[2.5. Convex hull trick 32](#_f4vi4ckuoxwk)

[ВИСНОВКИ 51](#_256vmtg8cshc)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 52](#_efsemex4vva8)

# 

# 

# ВСТУП

З розвитком технологій та зростанням обсягів даних виникає необхідність у більш ефективних методах обчислень, що робить оптимізацію алгоритмів динамічного програмування надзвичайно актуальною. Традиційні методи динамічного програмування дозволяють знаходити оптимальні рішення для складних задач, але часто вимагають значних обчислювальних ресурсів. Використання оптимізацій дозволяє значно зменшити час виконання і обсяг пам'яті, необхідної для розв'язання цих задач. Ці оптимізації сприяють підвищенню ефективності алгоритмів, роблячи їх придатними для реального застосування в умовах обмежених ресурсів. Дослідження в галузі оптимізації динамічного програмування є актуальним і необхідним для забезпечення ефективних і швидких рішень у різних прикладних задачах, що робить цю тему важливою і перспективною для подальшого наукового і практичного розвитку.

Динамічне програмування – це метод розв'язання складних задач шляхом розбиття їх на простіші підзадачі і запам'ятовування результатів цих підзадач, щоб уникнути їх повторного обчислення. Головна ідея полягає в тому, щоб розв'язати кожну підзадачу лише один раз, зберегти її результат і використовувати його повторно при необхідності. За допомогою динамічного програмування можна розв'язувати комбінаторні задачі, задачі з мінімізацією або максимізацією оцінки (наприклад, задача про рюкзак, задача про розбиття на підвідрізки), а також задачі теорії ігор. До розв’язків певних задач можна застосовувати оптимізаційні методи та техніки, які зменшують часову складність їх роботи. Основні типи оптимізацій динамічного програмування включають метод "Розділяй та володарюй", Кнута-Яо, Convex Hull Trick та використання структур даних.

***Метою дипломної роботи*** є дослідження розв’язків, які базуються на методі динамічного програмування, та аналіз його можливих оптимізацій.

Для досягнення мети потрібно виконати наступні ***завдання***:

1. Розглянути базові принципи та методи динамічного програмування. Проаналізувати основні задачі, які вирішуються за допомогою динамічного програмування.
2. Описати та проаналізувати існуючі методи оптимізації динамічного програмування, а саме метод "Розділяй та володарюй", Кнута-Яо, Convex Hull Trick та пришвидшення за допомогою використання структур даних.
3. Розглянути та розв'язати задачі за допомогою цих методів. Порівняти складність цих рішень із розв'язками без їх використання.
4. Реалізувати дані розв’язки мовою C++, та оцінити швидкість їх роботи на ресурсах для тестування.

***Об’єкт дослідження*** ‒ оптимізації методу динамічного програмування.

***Предмет дослідження*** ‒ аналіз оптимізацій методу динамічного програмування з метою зниження обчислювальної складності та покращення ефективності використання пам'яті.

Ця робота складається з двох розділів. У першому розділі зроблено огляд методу динамічного програмування. У цьому розділі розглядається метод та класичні задачі для розв’язку яких він використовується. У другому розділі описано методи "Розділяй та володарюй", Кнута-Яо, Convex Hull Trick та пришвидшення за допомогою використання структур даних. Розглянуто нерівність чотирикутника. Представлено розв’язки задач та їх реалізацію мовою C++.

# 

# РОЗДІЛ 1

# ОГЛЯД МЕТОДУ ДИНАМІЧНОГО ПРОГРАМУВАННЯ

## 1.1. Визначення та основні концепції

Динамічне програмування — це одночасно і метод математичної оптимізації і парадигма розв’язання алгоритмічних задач.

Термін "динамічне програмування" був вперше використаний у 1940-х роках Річардом Беллманом для опису процесу розв'язання проблем, де потрібно знайти найкращі рішення одне за одним. У 1953 році він удосконалив його до сучасного значення, маючи на увазі, зокрема, вкладання менших проблем у більші рішення, після чого ця галузь була визнана IEEE як тема системного аналізу та інженерії.

В обидвох контекстах суть методу полягає в розбитті однієї задачі на декілька менших підзадач у рекурсивній формі. Зазвичай він використовується для вирішення оптимізаційних задач, тобто для знаходження найкращого (за певним критерієм) розв’язку з множини всіх можливих розв’язків. Такі завдання можуть мати декілька оптимальних розв’язків.

Існує дві ключові характеристики, які повинна мати задача, щоб до неї можна було застосувати динамічне програмування: оптимальна підструктура та підзадачі, що перетинаються. Якщо задачу можна розв'язати, об'єднавши оптимальні розв'язки підзадач, що не перетинаються, стратегія називається "розділяй і володарюй"[1]. Саме тому сортування злиттям і швидке сортування не класифікуються як задачі динамічного програмування.

Оптимальна підструктура означає, що розв'язок задачі оптимізації може бути отриманий шляхом комбінації оптимальних розв'язків її підзадач. Такі оптимальні підструктури зазвичай описуються за допомогою рекурсії. Наприклад, у випадку чисел Фібоначчі це , або для певної задачі максимізації

Перекриття підзадач означає, що простір підзадач повинен бути невеликим, тобто будь-який рекурсивний алгоритм, що розв'язує задачу, повинен розв'язувати ті самі підзадачі знову і знову, а не генерувати нові підзадачі. Наприклад, розглянемо рекурсивне формулювання для генерування ряду Фібоначчі: при та базовим випадком . Тоді та . Тепер розв'язується в рекурсивних піддеревах як , так і .

Незважаючи на те, що загальна кількість підзадач може бути невелика а саме , ми будемо розв'язувати одні й ті ж проблеми знову і знову, якщо приймемо такий наївний рекурсивний спосіб розв'язання. Динамічне програмування враховує цей факт і розв'язує кожну підзадачу лише один раз.

## 

Рис. 1.1. Дерево рекурсії для підрахунку (однакові стани позначені одними кольорами)

Цього можна досягти одним із двох способів:

* Підхід зверху вниз: Це прямий наслідок рекурсивного формулювання будь-якої проблеми. Якщо розв'язок будь-якої проблеми можна сформулювати рекурсивно, використовуючи розв'язок її підпроблем, і якщо підпроблеми перетинаються, то можна легко запам'ятати або зберегти розв'язки підпроблем у вигляді таблиці. Кожного разу, коли ми намагаємося вирішити нову підпроблему, ми спочатку перевіряємо таблицю, щоб побачити, чи вона вже вирішена. Якщо рішення було записано, ми можемо використовувати його безпосередньо, в іншому випадку ми вирішуємо підзадачу і додаємо її рішення в таблицю.

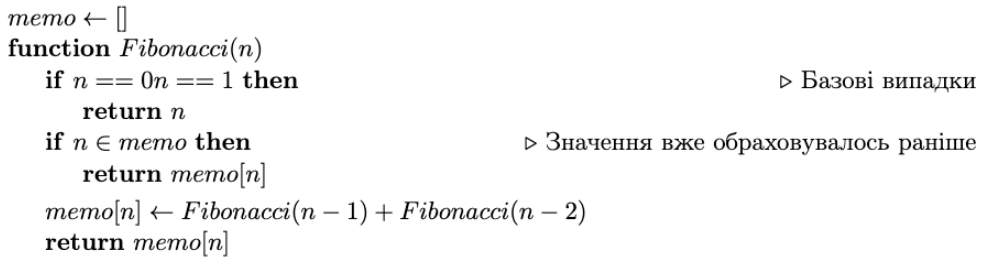


Рис. 1.2. Псевдокод знаходження числа Фібоначчі методом “зверху вниз”

* Висхідний підхід: Після того, як ми рекурсивно сформулювали рішення проблеми в термінах її підпроблем, ми можемо спробувати переформулювати проблему за принципом "знизу вгору": спробувати вирішити спочатку підпроблеми, а потім використати їхні рішення для побудови та отримання рішень більших підпроблем. Це також зазвичай робиться в табличній формі шляхом ітеративного знаходження рішень все більших і більших підпроблем, використовуючи рішення малих підпроблем.



Рис. 1.3. Псевдокод знаходження числа Фібоначчі висхідним підходом

## 1.2. Класичні задачі, які вирішуються за допомогою динамічного програмування

### Задача про рюкзак

*Джерело: https://atcoder.jp/contests/dp/tasks/dp\_d*

Дано предметів, пронумерованих числами . Для кожного задано елемент має вагу та вартість Таро вирішив вибрати деякі з предметів і понести їх додому у рюкзаку. Місткість рюкзака дорівнює , а це означає, що сума ваг взятих предметів повинна бути не більше .

Знайдіть максимально можливу суму вартостей предметів, які Таро візьме з собою додому.

Обмеження: кількість речей , ваги та , ціни

Дана задача розв’язується за допомогою динаміки , де — індекс останнього розглянутого елементу, — загальна вага набраних елементів, — максимальна сумарна вартість елементів, вибраних середе , загальна вага яких рівна .

Перебираючи значення у порядку зростання, та значення можна оновлювати значення розглядаючи два випадки:

* З використання елементу , тоді вартість всіх вибраних елементів рівна . Тобто попередній стан мав значення ваги і з додаванням елементу вона змінюється на .
* Без використання елементу , тоді вартість всіх вибраних елементів рівна . Тобто попередній стан мав значення ваги і оскільки новий елемент не додався то він таким і залишиться.

Тоді значення обраховується за формулою , для та , для .

Базовий стан цієї динаміки, це . Відповідь знаходиться, як максимум серед всіх .

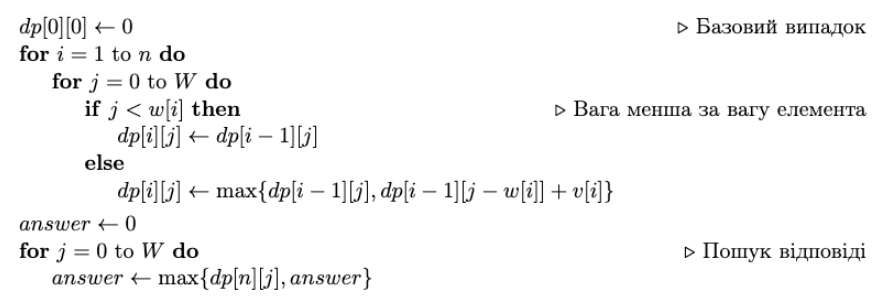


Рис. 1.4. Псевдокод алгоритму розвʼязання задачі про рюкзак

Складність цього алгоритму рівна .

### Найдовша спільна підпослідовність

*Джерело: https://atcoder.jp/contests/dp/tasks/dp\_f*

Дано два рядки та . Потрібно знайти найдовший рядок який є підпослідовністю обидвох рядків та Підпослідовність рядка – це рядок, отриманий шляхом видалення нуля або більше символів з і конкатенації символів, що залишилися, без зміни порядку.

Обмеження. Рядки складаються з малих латинських літер, та довжина кожного з них не перевищує .

Ідея розв’язку даної задачі полягає у знаходженні довжини найдовшої спільної підпослідовності за допомогою динамічного програмування, та на основі обчислених значень знаходження найдовшої.

Нехай, — довжина найдовшої спільної підпослідовності підрядків та . Значення перераховуються наступним чином:

* або , хоча б один рядок є пустим, отже довжина найдовшої спільної підпослідовності рівна .
* , тоді можливий перехід , тобто елемент додається до найдовшої підпослідовності підрядків та
* , максимальна відповідь серед менших підзадач.

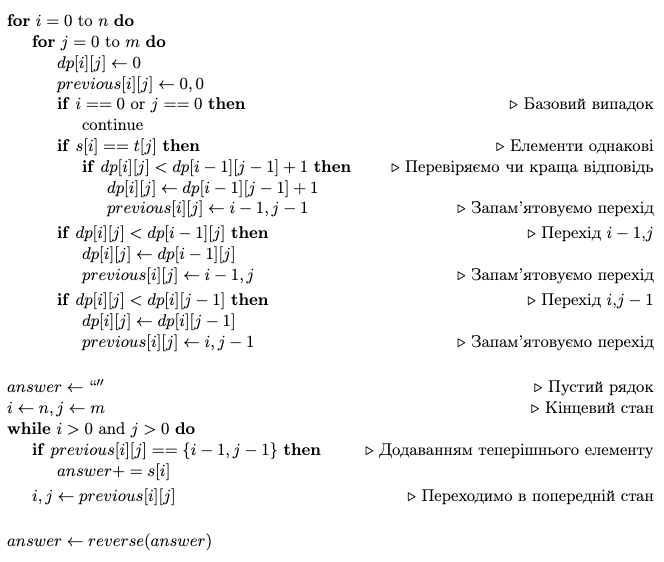
Для побудови послідовності в наявний алгоритм для кожного стану динаміки запам'ятовується стан через який вона набуває оптимального значення. Тоді, починаючи зі стану , де n та це довжини рядків та відповідно, розглядається перехід до менших станів.

Рис. 1.5. Псевдокод алгоритму розвʼязання задачі пошуку найдовшої спільної підпослідовності

Складність даного рішення рівна

# 

# РОЗДІЛ 2

# МЕТОДИ ОПТИМІЗАЦІЙ ДИНАМІЧНОГО ПРОГРАМУВАННЯ

## 2.1. Оптимізація за допомогою використання певних структур даних

Розв’язки певних задач, що використовують динамічне програмування та оновлюють відповідь для певного стану через не константну множину розв’язків менших станів, можна оптимізувати за допомогою певних структур даних, що підтримують потрібні запити на множинах даних. Розглянемо такі оптимізації на прикладах конкретних задач.

**Максимальна зростаюча підпослідовність**

*Джерело: https://cses.fi/problemset/task/1145*

Вам задано масив з цілих чисел. Ваша задача - знайти найдовшу зростаючу підпослідовність у масиві, тобто найдовшу підпослідовність, у якій кожен наступний елемент більший за попередній.

Підпослідовністю називається послідовність, яку можна отримати з масиву, видаливши деякі елементи без зміни порядку елементів, що залишились.

Обмеження. Довжина масиву , елементи .

Розглянемо алгоритм знаходження відповіді зі складністю та використанням пам’яті . Нехай — це довжина найдовшої зростаючої підпослідовності яка закінчується в елементі Тоді, значення відповідей можна перераховувати за формулою . Відповіддю на задачу буде максимальне значення серед всіх .

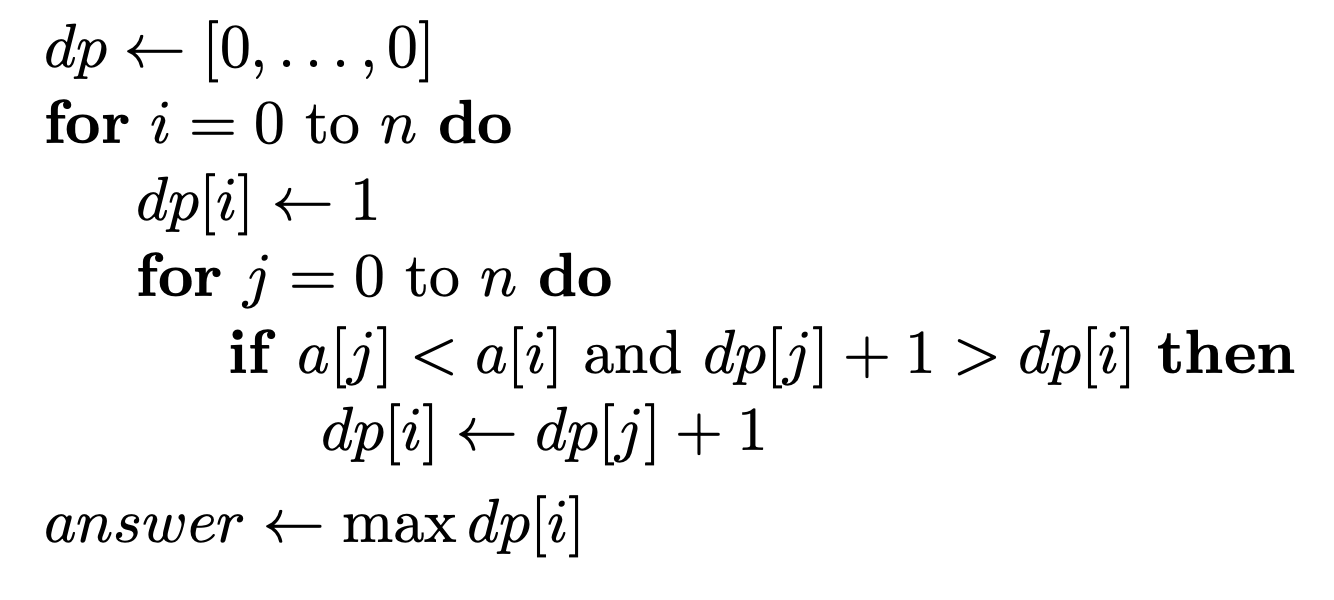


Рис. 2.1. Псевдокод алгоритму пошуку найдовшої зростаючої послідовності за

Щоб оптимізувати цей розв'язок, можна зберігати масив , де — найбільше значення для та які вже є обрахованими. Тобто, це довжина максимальної зростаючої підпослідовності, що закінчується в елементі значення якого рівне .

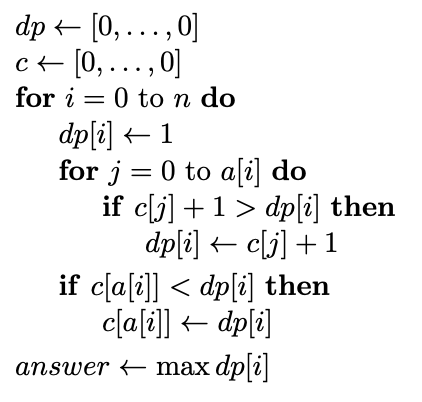


Рис. 2.2. Псевдокод алгоритму пошуку найдовшої зростаючої послідовності за де – це максимальне значення

В даному алгоритмі ми беремо максимум на відрізку та оновлюємо значення в позиції Для виконання цих операцій можна використати дерево відрізків[2], яка підтримує обидві ці операції з часовою складністю та використовує пам’яті, якщо перед обрахунком провести нормалізацію масиву , так щоб кожен елемент був цілим числом у межах .

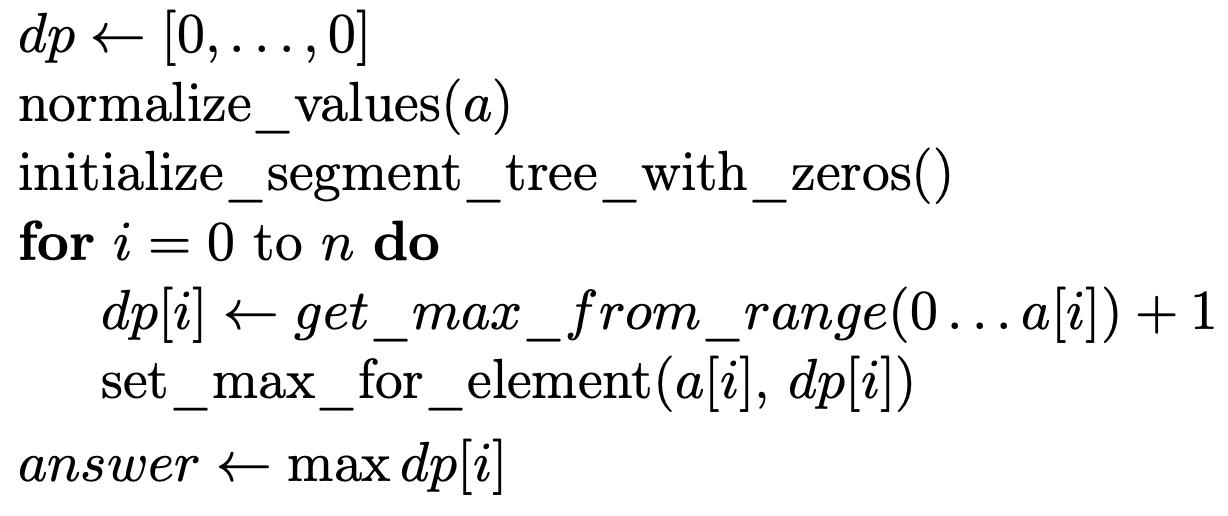


Рис. 2.3. Псевдокод алгоритму пошуку найдовшої зростаючої послідовності за з використанням дерева відрізків

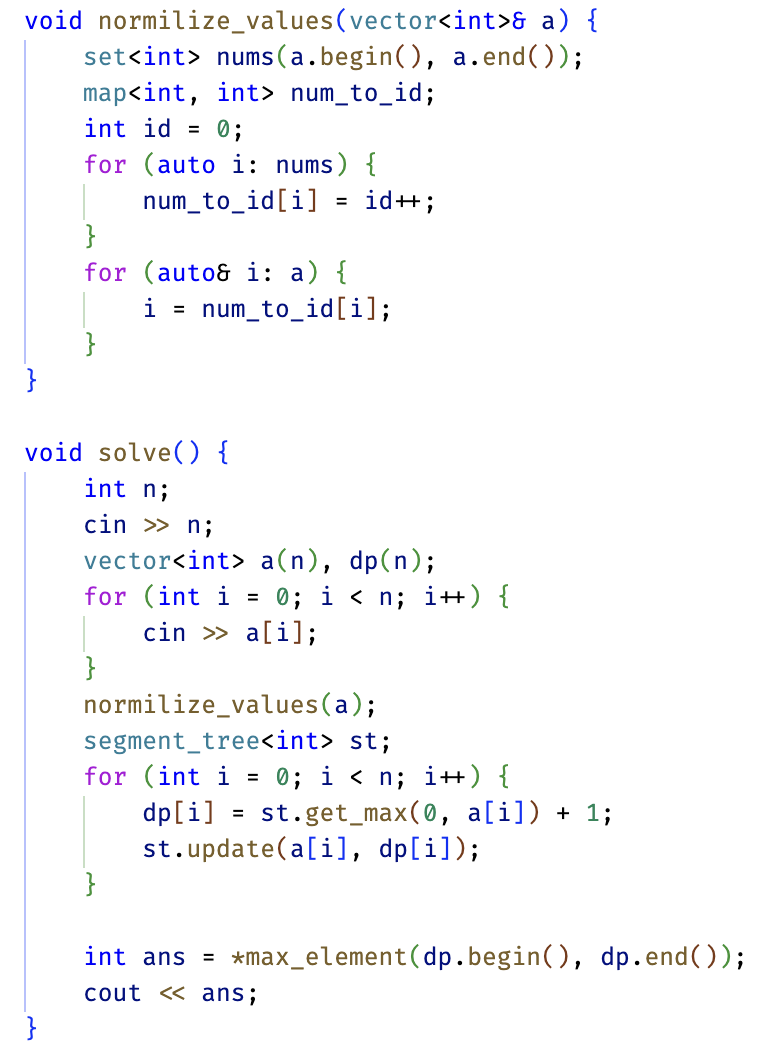


Рис. 2.4. Реалізація мовою C++

Це рішення, має складність для нормалізації початкового масиву та для знаходження відповіді, тож у загальному воно працює зі складністю . Завдяки використанню дерева відрізків ми змогли перейти від рішення за до швидшого не змінивши ні стан, ні переходи в початковому алгоритмі. У системі тестування рішення проходить всі тести з максимальним часом 0.34 секунди.

**Гарячий запуск (складна версія)**

*Джерело: https://codeforces.com/contest/1799/problem/D2*

У вас є пристрій з двома процесорами. У вас також є програм, пронумерованих від до які можна запустити на цих процесорах, кожна з яких може бути запущена на рівно одному з процесорів.

Кожна програма 𝑖 (1≤𝑖≤𝑘) займає секунд для виконання на одному з процесорів. Однак, якщо остання програма, яку запускали на цьому процесорі, також була програмою то вона виконується лише за секунд (). Зауважте, що це справедливо лише у тому випадку, якщо ми запускаємо програму декілька разів поспіль — якщо ми запускаємо програму , потім якусь іншу програму, потім знову програму то вона знову займе секунд за.

Вам задано послідовність довжиною , яка складається з цілих чисел від до . Вам потрібно на своєму пристрої виконати програми у цьому порядку. Для всіх ви не можете розпочати виконання програми

доки не завершиться програма .

Знайдіть мінімальний час, необхідний для виконання всіх програм по черзі.

Обмеження. Довжина послідовності , кількість різних програм . Значення .

У даній задачі можна зберігати наївну динаміку — мінімальна сумарна вартість роботи всіх програм за умови, що остання виконана програма на першому процесорі є , а на другому . Але можна зробити спостереження, що хоча б одне з цих значень завжди буде рівне , оскільки всі програми запускаються по черзі, а отже перед запуском програми відбувся запуск .

Тоді можемо перейти до — мінімальна сумарна вартість роботи всіх програм за умови, що остання виконана програма на одному з процесорів є , а на іншому (ситуація при якій також можлива, але ніколи не є оптимальною оскільки ). Тоді можемо перераховувати значення за допомогою таких переходів:

* ) — запуск програми на процесорі на якому виконувалась програма
* — запуск програми на процесорі на якому останньою виконувалась програма .

Тут функція повертає ціну запуску програми на процесорі, де останньою запускалась програма . Часова складність цього розв’язку рівна

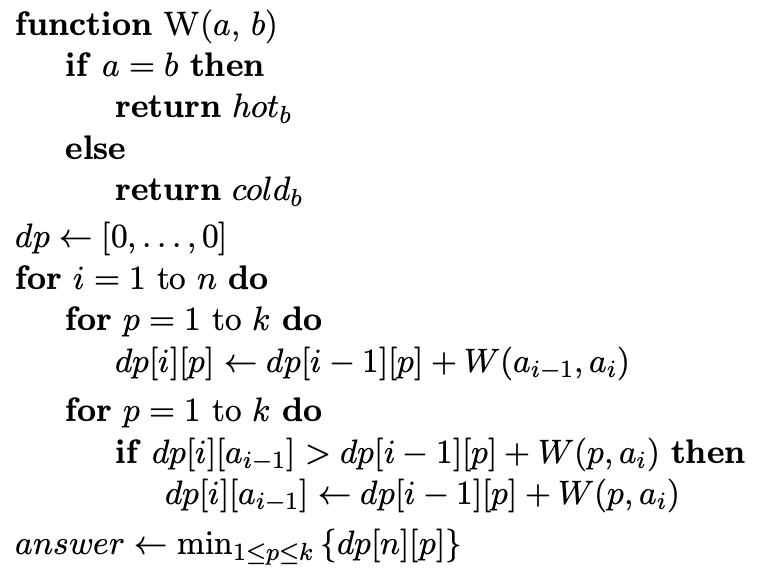


Рис. 2.5. Псевдокод розв'язку

Зараз в першому циклі, ми оновлюємо всі значення додаючи до них однакове число , а в другому знаходження мінімуму серед буде еквівалентно знаходженням мінімумів серед та . Ці операції можна оптимізувати за допомогою дерева відрізків з відкладеним поширенням[3] зберігаю стан в листках дерева. Тоді отримаємо наступний алгоритм.

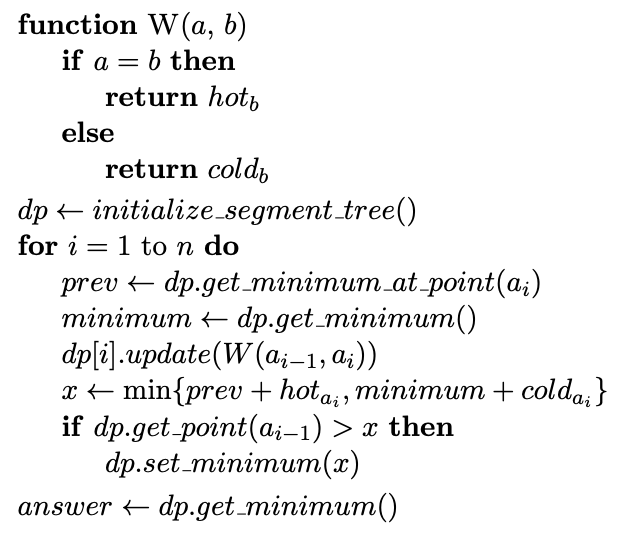


Рис. 2.6. Псевдокод розв'язку з використанням дерева відрізків

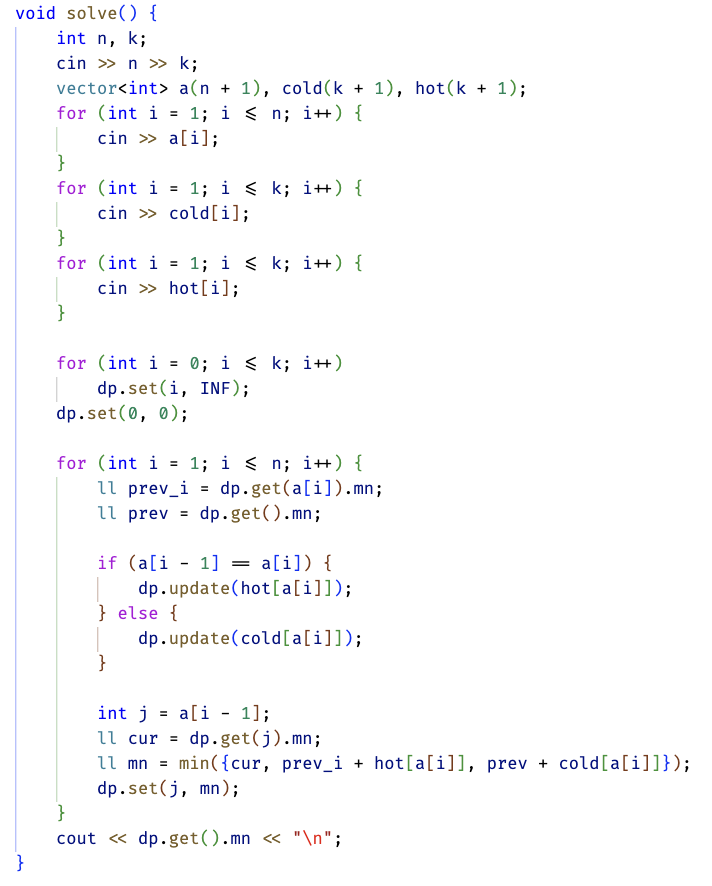


Рис. 2.7. Код розв'язку з використанням дерева відрізків мовою C++

Це рішення, має складність . Завдяки використанню дерева відрізків ми змогли перейти від рішення за до швидшого не змінивши ні стан ні переходи в початковому алгоритмі. У системі тестування рішення проходить всі тести з максимальним часом 358 мілісекунд.

## 2.2. Оптимізація Кнута - Яо

Оптимізація Кнута - Яо — стандартна оптимізація динаміки, яка дозволяє вирішувати два типи задач (за умови, що вони відповідають певній умові):

* Задача про оптимальне розбиття масиву з елементів на підвідрізків. Цю задачу можна вирішити підтримуючи стан динаміки — мінімальна ціна розбиття перших елементів масиву на підвідрізків. Порахувати значення можна за допомогою переходу , де — індекс елементу на якому закінчилась підпослідовність та — функція, що повертає ціну підвідрізку, що починається в елементі з індексом та закінчується в елементі Складність такого розвʼязку .
* Динаміка по підвідрізкам — задачі які вирішуються за допомогою знаходження відповідей для підвідрізків меншого розміру та комбінація їх для знаходження відповіді для більшого відрізку. Такі задачі вирішуються за допомогою динаміки — відповідь для відрізку та переходу . Де — ціна використання відрізку. Складність такої динаміки рівна

При використанні оптимізації Кнута - Яо можна досягнути складності в обидвох випадках. Розглянемо використання оптимізації на задачах першого типу.

### Задача про оптимальне розбиття масиву на підвідрізки

Дано масив невід'ємних чисел довжини . Назвемо ціною підвідрізку квадрат суми його елементів. Необхідно розбити масив на підвідрізків таким чином, щоб мінімізувати суму потужностей розбиття. Іншими словами, виділити такі чисел (межі підвідрізків розбиття) , щоб значення

було мінімальним.

Цю задачу можна вирішити за допомогою динаміки , де — індекс елементу на якому закінчилась підпослідовність та — функція, що повертає ціну підвідрізку, що починається в елементі з індексом та закінчується в елементі , саме значення . Тоді відповіддю на задачу є , тобто мінімальна ціна розділити перші елементів на підвідрізків. Значення можна обраховувати за , як квадрат різниці префіксних сум його кінців в позиціях та , якщо попередньо порахувати всі префіксні суми за . Складність даного алгоритму рівна .

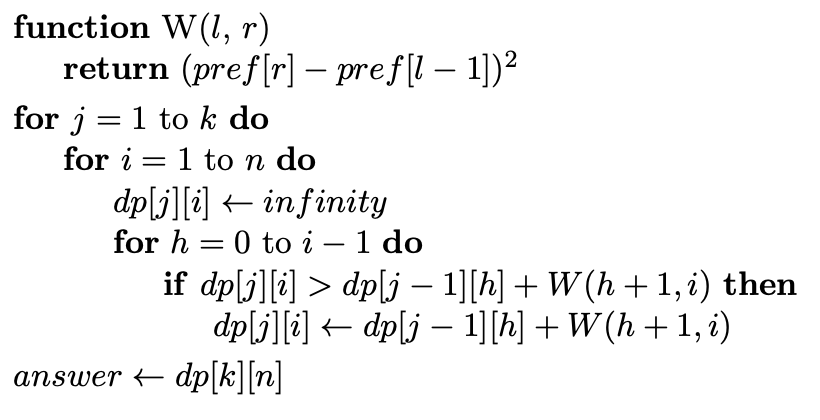


Рис. 2.8. Псевдокод рішення зі складністю

Позначимо значення при якому досягається мінімальна вартість , як . Якщо таких значень декілька, то візьмемо найлівіше з них. Тоді знаючи ми могли б знаходити за

Оптимізацію Кнута-Яо можна використовувати за умови, монотонності точки розрізу по обидвом координатах[3], а саме: , для будь-яких та .

Інакше кажучи, для фіксованої кількості підрядків , оптимальні значення повинні не спадати по , тобто чим правіше буде закінчення останнього підвідрізку, тим правіше має бути його оптимальний початок. Також для фіксованого кінця , оптимальне значення повинно не зменшуватись по , тобто чим більше відрізків ми можемо використати, тим коротшим повинен бути найправіший з них. Більш формально, то елементи матриці повинні не спадати як по рядках, так і по стовпцях.

Використовуючи цю умову, ми можемо перебирати значення від менших до більших та значення від більших до менших, можна розглядати значення тільки в проміжку , оскільки в момент обрахунку значення вони обидві межі цього проміжку вже будуть знайдені.

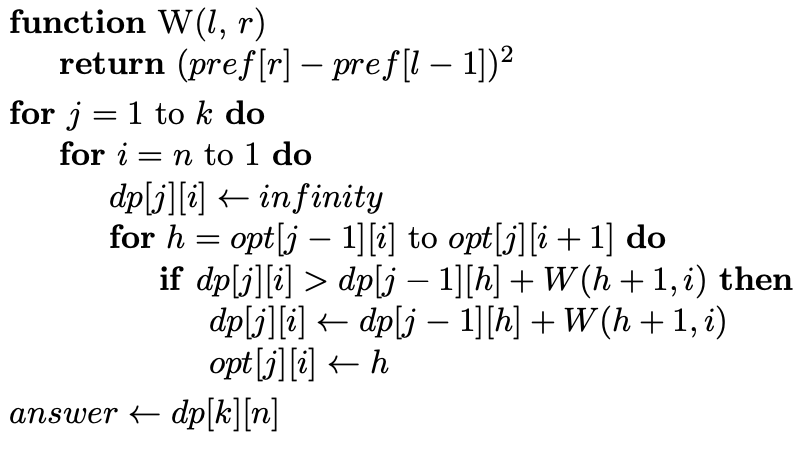


Рис. 2.9. Псевдокод рішення задачі зі складністю

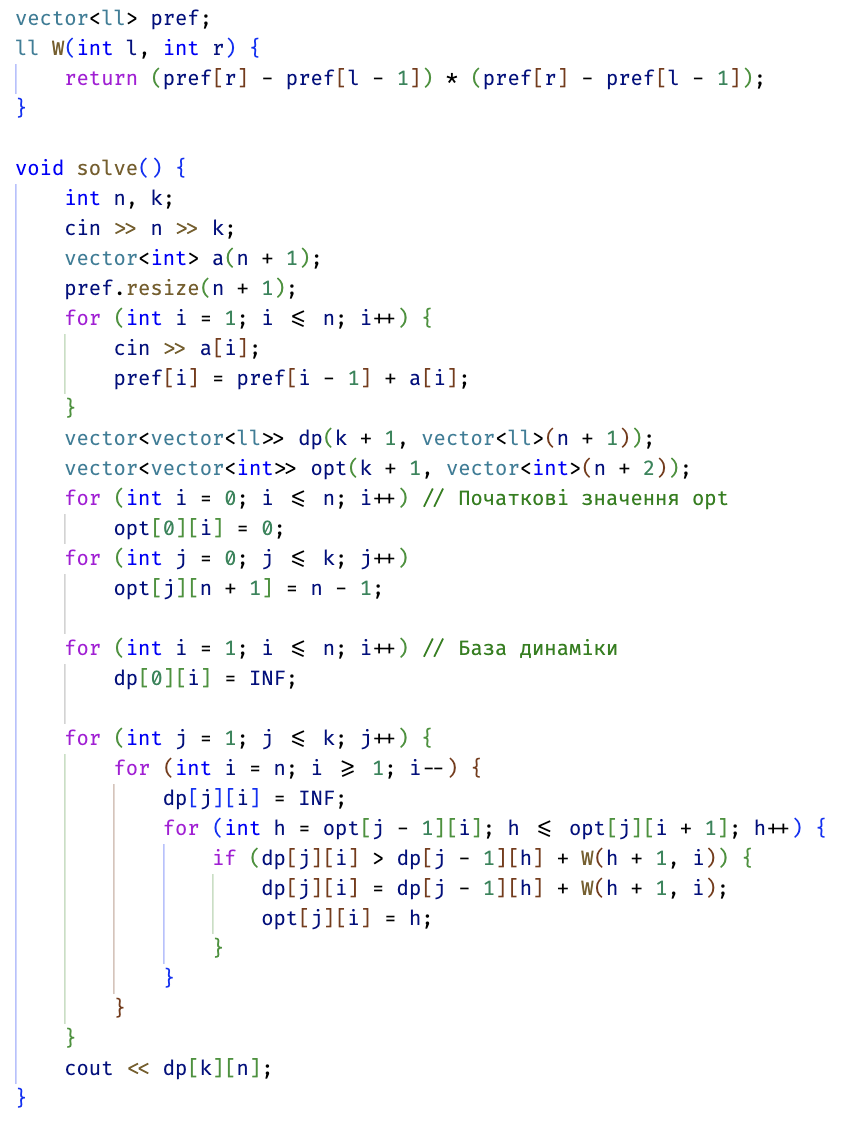


Рис. 2.10. Рішення задачі зі складністю мовою C++

Також, варто звернути увагу на те, що ми знаходимо всі значення , а отже, у разі необхідності ми можемо їх використати для відновлення розбиття масиву, при якому досягається оптимальне значення.

Щоб дізнатись складність алгоритму, проаналізуємо кількість операцій які ми робимо. Для кожної пари , ми робимо рівно операцій (якщо вважати, що та ). Тоді:

Значення можна розписати як , значення, як . Тоді отримаємо кількість операцій рівну:

Отже складність рішення рівна .

### Динаміка по підвідрізкам

### Knuth Division

*Джерело: https://cses.fi/problemset/task/2088*

Дано масив з чисел, ви повинні розбити його на підмасивів, кожен з яких містить один елемент.

На кожному кроці ви можете вибрати будь-який підмасив і розбити його на два підмасиви. Вартість такого розбиття дорівнює сумі значень у вибраному підмасиві.

Потрібно знайти мінімальну сумарну вартість.

Обмеження. Довжина масиву , елементи масиву .

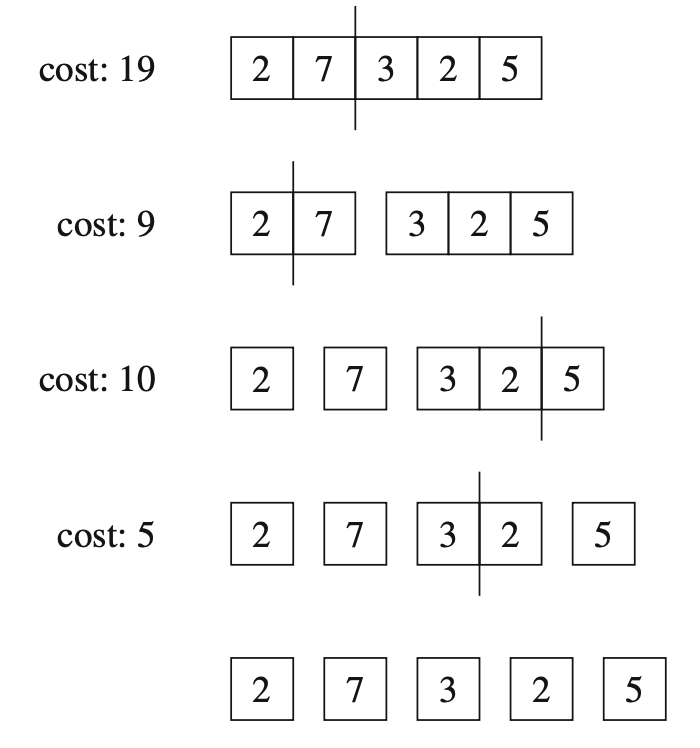


Рис. 2.11. Оптимального розбиття масиву

Для вирішення цієї задачі, використаємо динаміку — мінімальна вартість об'єднання всіх купок від -тої до -тої. Її можна обраховувати за формулою:

Тобто, ми обєднуємо дві купки, які містять у собі всі цукерки з купок та створення яких коштує та відповідно і платимо за цю операцію. Тут – це ціна об’єднання двох купок, а саме сумарна кількість цукерок яку можна обраховувати за , якщо попередньо знайти всі префікс суми за .

Таку динаміку потрібно раховувати від менших відрізків розміру , до більших. Відповіддю буде вартість об'єднання всіх купок від до , тобто .

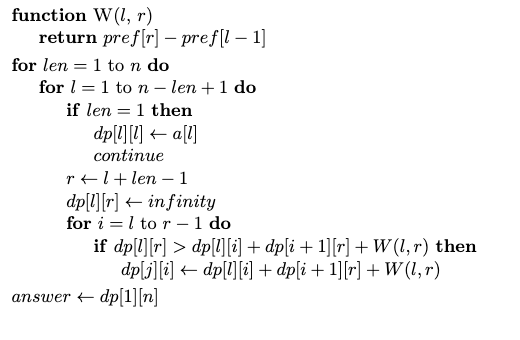


Рис. 2.12. Псевдокод рішення за

Складність наївного обрахування динаміки рівна сумі довжин всіх підвідрізків, а саме .

Позначимо оптимальне , при якому набувається мінімальне значення динаміки , як . Тоді стверджується, що знову виконується умова монотонності точки розрізу по обидвох координатах (), як і в попередній задачі. Отже можна застосувати оптимізацію Кнута-Яо.

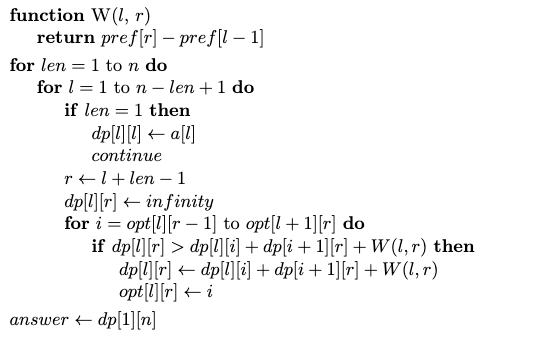


Рис. 2.13. Псевдокод рішення за

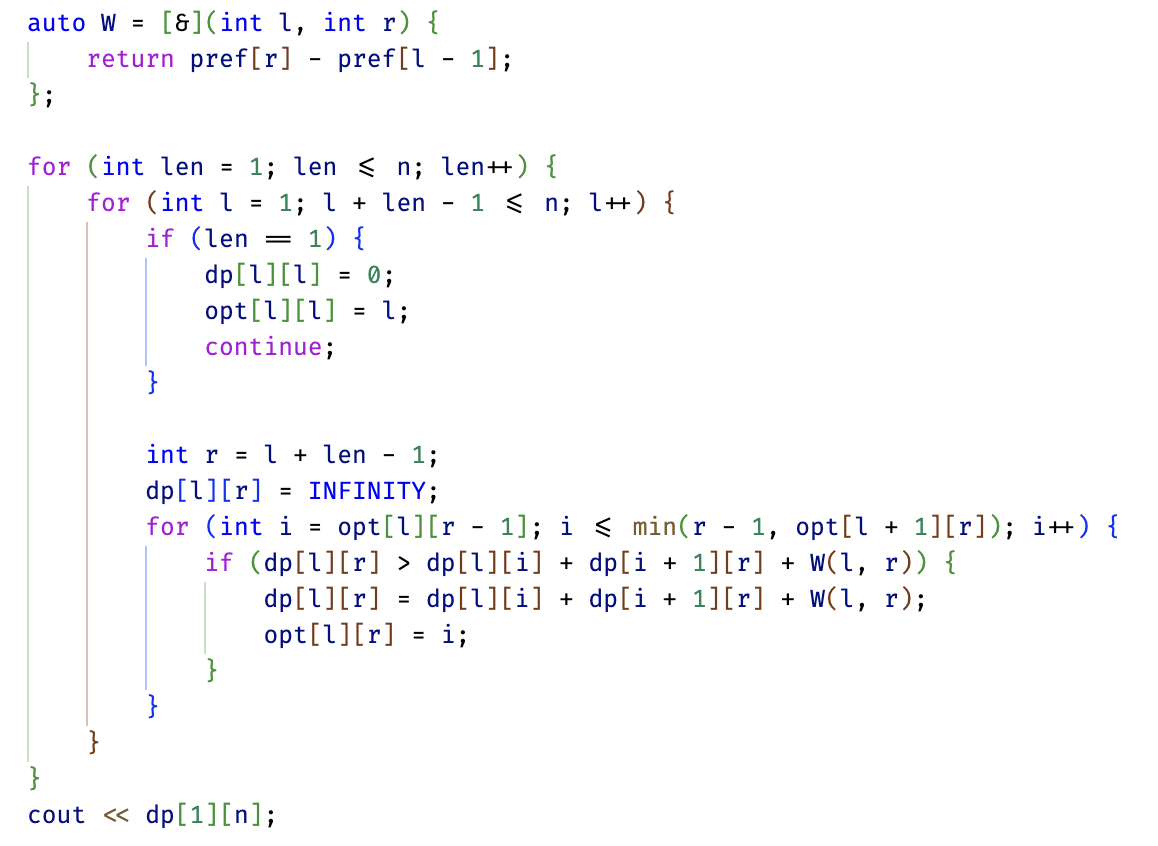


Рис. 2.14. Псевдокод рішення задачі за мовою C++

Кількість виконаних операцій, аналогічна кількості операцій у розв'язку попередньої задачі, а тоже складність також рівна . У системі тестування рішення проходить всі тести з максимальним часом 0.69 секунди.

## 2.3. Оптимізація “Розділяй та володарюй”

Розділяй та володарюй (divide-and-conquer) — це стандартна оптимізація динаміки, що дає змогу розв'язувати задачі про оптимальне розбиття масиву з елементів на підвідрізків за (якщо ця динаміка задовольняє деякій умові, про яку ми поговоримо пізніше), тоді як наївна динаміка розв'язує цю задачу за . Іншими словами, ми оптимізуємо динаміку такого виду:

де — мінімальна вартість розбиття перших елементів на підвідрізків, а — ціна підвідрізку .

Розглянемо цю оптимізацію на прикладі наступної задачі.

**Ciel і кабінки**

*Джерело: https://codeforces.com/contest/321/problem/E*

Лисиця Cielу парку розваг. Зараз вона стоїть у черзі перед колесом огляду. У черзі стоїть людей (точніше, лисиць): першою людиною ми називаємо ту, що стоїть на чолі черги, а -ю - останню в черзі.

Є k кабінок, і спосіб розподілу людей виглядає наступним чином:

* Коли приїжджає перша кабінка, людей, що стоять на початку черги, заходять у кабіну.
* Потім, коли приїжджає другий вагони, людей, які стоять на початку черги, заходять у кабіну.

…

* Решта людей заходять в останній (-й) вагон.

Зверніть увагу, що повинні бути додатними. З твердження випливає, що та .

Ви знаєте, що люди не хочуть сидіти з незнайомцями у кабінах, тому ваша задача — знайти оптимальний спосіб розподілу (тобто знайти оптимальну послідовність ), щоб зробити людей щасливими. Для кожної пари людей та існує значення , яке позначає рівень незнайомства. Можна вважати, що для всіх та для всіх . Тоді незнайоме значення кабіни — це сума рівнів незнайомства між будь-якою парою людей, які знаходяться в кабінці.

Загальне невідоме значення — це сума невідомих значень для всіх гондол. Допоможіть Лисичці Ciel знайти мінімально можливу сумарну невідому вартість для деякого оптимального розподілу.

Обмеження: кількість людей , кількість кабін , значення незнайомств .

Розглянемо наступну динаміку. Нехай — мінімальна вартість поділити перших людей між кабінками. Формула переходу буде наступною:

де – людина, яка була останньою в групі , – подвоєна ціна підвідрізку . Функцію оцінку можна обраховувати за , якщо підрахувати значення масиву префіксних сум матриці , . Щоб підраховувати всі значення за , будемо використовувати попередньо знайдені значення, тоді: . Тоді .

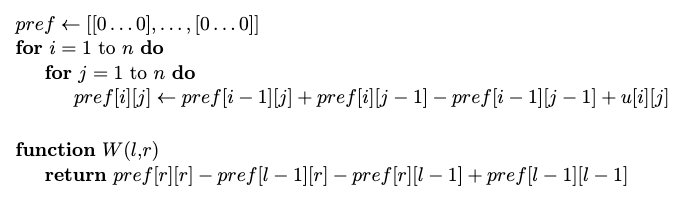


Рис. 2.15. Псевдокод генерації префіксних сум для матриці

Наївний підрахунок цієї динаміки має складність . Для її покращити використаємо оптимізацію “Розділяй та володарюй”. Позначимо , як таке що набуває мінімального значення, якщо таких є декілька, то візьмемо найменше серед них. Щоб оптимізацію можна було використовувати, повинна виконуватись умова [5], для будь яких та . Інакше кажучи чим більше елементів ми використовуємо, тим правіше повинна початись остання група.

Повернемося до алгоритму, який обчислював значення послідовно. За замовчуванням ми знаємо, що – це якесь число з інтервалу . Однак якщо ми вже порахували , то оскільки , то ми можемо скоротити інтервал пошуку до . З першого погляду може здатися, що це вже допомагає нам знаходити всі значення за лінійний час, але це не так. За умови, що всі значення дорівнюють нулю. Це ніяк не суперечить умові монотонності точки розрізу, однак у цьому випадку наша оптимізація ніяк нам не допомагає: ми все ще для кожного індексу перебираємо всі індекси від до .

Давайте спробуємо знайти оптимальну точку розрізу для індексу . Тоді якщо ми знайшли , то ми знаємо, що для всіх індексів , менших , вірно, що , а для всіх індексів, більших за , вірно, що . Тоді ми розбили область пошуку оптимальної точки розрізу для лівої і правої половин масиву на дві множини, що перетинаються тільки в .

Ми знайшли оптимальну точку розрізу для індексу , після чого на межі пошуку оптимальної точки розрізу для лівої та правої половин масиву з'явилися обмеження. Тоді давайте рекурсивно запустимо пошук для лівої і правої половин. Там ми будемо шукати оптимальні точки розрізу для індексу від 0 до , і для індексу від до , і так далі.

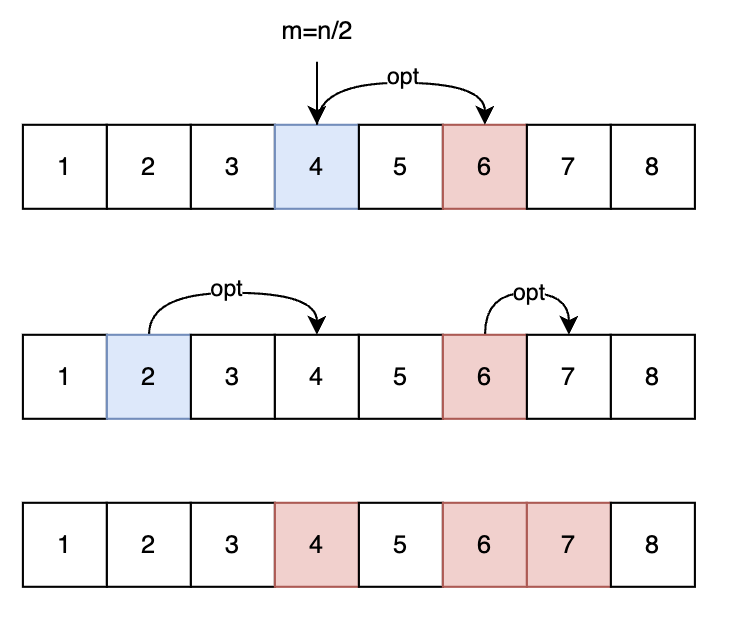


Рис. 2.16. Приклад рекурсивної роботи динаміки. Синій колір – елементи для яких обчислюється значення , червоний – межі обчислення

Щоразу ми маємо деякий відрізок індексів масиву, для яких треба знайти оптимальні точки розрізу, а також з вищих рівнів рекурсії в нас уже є деякі обмеження на ці точки розрізу: ми точно знаємо, що вони не менші за і менше . Тоді ми вибираємо індекс , знаходимо для нього оптимальну точку розрізу на інтервалі , після чого запускаємося рекурсивно для відрізка на якому нижнє обмеження - це все ще , а верхнє обмеження це (адже права межа рахується невключно), бо всі індекси в лівій половині відрізка менші за ; а також запускаємося рекурсивно з відрізка

на якому верхнє обмеження залишається рівним , а нижнє обмеження стає рівним , тому що всі індекси правої половини відрізка більші за .

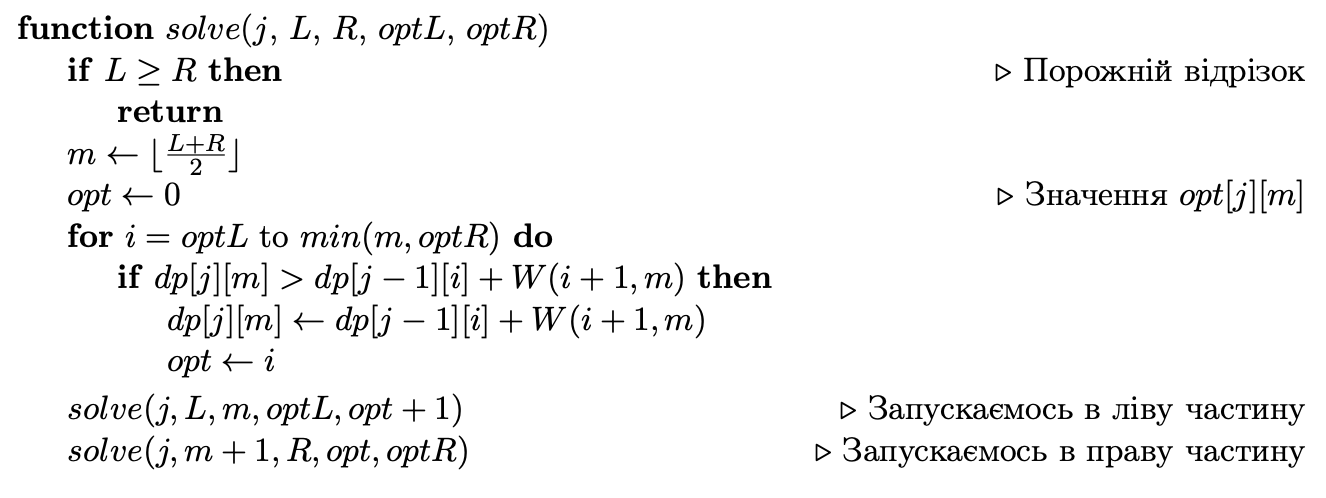


Рис. 2.17. Псевдокод реалізації оптимізації

Кожного разу, довжина розглянутих відрізків зменшується у два рази , тому глибина рекурсії дорівнюватиме . Давайте доведемо, що всі виклики з одного рівня рекурсії сумарно відпрацюють за .

В одному конкретному виклику ми робимо константну кількість операцій, а також запускаємо цикл від до . Можна вважати, що один рекурсивний запуск працює за . На верхньому рівні рекурсії це рівне . Після чого на наступному рівні рекурсії півінтервал

ділиться на два: та , і так далі. На кожному рівні рекурсії початковий півінтервал розбивається на декілька відрізків . Сусідні відрізки перетинаються рівно в одному елементі, отже на одному рівні рекурсії кожен індекс буде використано для перерахунку щонайбільше двічі, тому всього ми зробимо операцій на одному рівні рекурсії, що в підсумку дасть нам для всіх рівнів. Таким чином, підсумкова асимптотика алгоритму - , що й потрібно було довести.

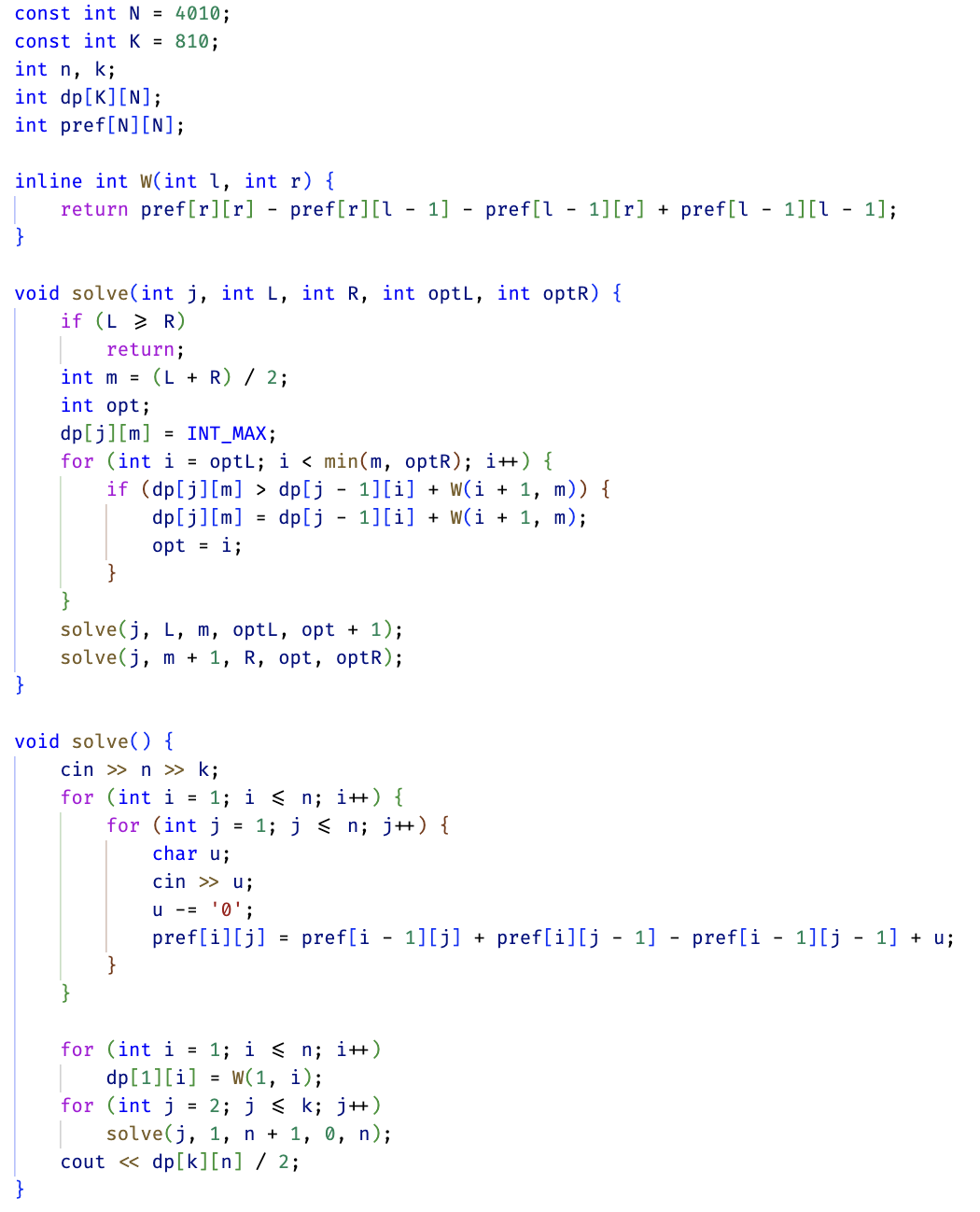


Рис. 2.18. Код рішення задачі мовою C++

Дане рішення проходить всі тести з максимальним часом 1280 мілісекунд.

## 2.4. Нерівність чотирикутника

Нерівність чотирикутника — це умова на вартість підвідрізків під час перерахунку динаміки, що використовується в багатьох оптимізаціях динаміки (оптимізація Кнута - Яо, оптимізація розділяй-і-владарюй, 1D/1D-оптимізація, лямбда-оптимізація).

Щоб використовувати в різні оптимізації динаміки, ми хочемо, щоб виконувалися певні властивості для динаміки (наприклад, монотонність точки розрізу або опуклість), проте найчастіше доводити ці властивості складно. У цьому нам допомагає нерівність чотирикутника. Це деяка умова на функцію ціни , з якої випливають різні умови на , які нам потрібні. Тому, нерівність чотирикутника для функції вартості - простий спосіб доводити коректність свого рішення яке використовує якусь з оптимізацій динаміки.

Нерівність має наступний вигляд:

де . Ця нерівність використовується для задач в яких потрібно мінімізувати відповідь, для задач максимізації потрібно розглядати обернену, тобто змінити знак на .

Розглянемо використання цієї нерівності для вище розглянутих оптимізацій:

* Задача про поділ на відрізків

Оптимізація Кнута-Яо для задачі поділу на відрізків. Необхідними та достатніми умовами для коректної роботи оптимізації Кнута-Яо є умови монотонності точки розрізу за обома координатами, тобто , для будь-яких та .

Для роботи оптимізації “Розділяй та володарюй” необхідною та достатньою умовою є .

Якщо для функції оцінки виконується нерівність чотирикутника, то виконуються і ці умови [6].

* Задача динаміки по підвідрізкам

Для використання оптимізації Кнута-Яо необхідною та достатньою є умовою є . Достатньою умовою є виконання нерівності чотирикутника для функції та її монотонность ( для ) [7].

Доведення коректності використання цих оптимізацій для попередньо розглянутих задач:

Задача про оптимальне розбиття масиву на підвідрізків. У даній задачі функція оцінки підвідрізку визначається, як . Для можливості використання оптимізації Кнута-Яо потрібно, щоб для

Нерівність після відкриття дужок має наступний вигляд:

Тоді:

Оскільки всі є невід'ємними, то . Тоді для виконується, що . Для , . Отже виконується, що , а відповідно і нерівність чотирикутника для функції .

Knuth Division. У цій задачі функція оцінки визначається, як Для коректності застосування оптимізації Кнута-Яо для динаміки по підвідрізкам, потрібно довести, що виконуються наступні нерівності для : та .

Доведення виконання нерівності чотирикутника:

Оскільки кожен доданок з лівої частини зустрічається і в правій, та навпаки, нерівність після спрощення матиме вигляд:

Доведення монотонності функції :

Відрізок містить у собі всі елементи відрізку та елементи з від відрізків та . Сума елементів з відрізків та є невід'ємною, оскільки всі ( у випадку коли відрізом не містить жодного елементу). Отже нерівність завжди виконується. Тоді для цієї задачі можна використовувати оптимізацію Кнута-Яо.

Ciel і кабінки. У даній задачі для можливості використання оптимізації “Розділяй та володарюй” потрібно довести, що виконується нерівність чотирикутника для функції , тобто сума елементів квадратної матриці по всім .

Якщо позначити суми на певних частинах матриці наступним чином:

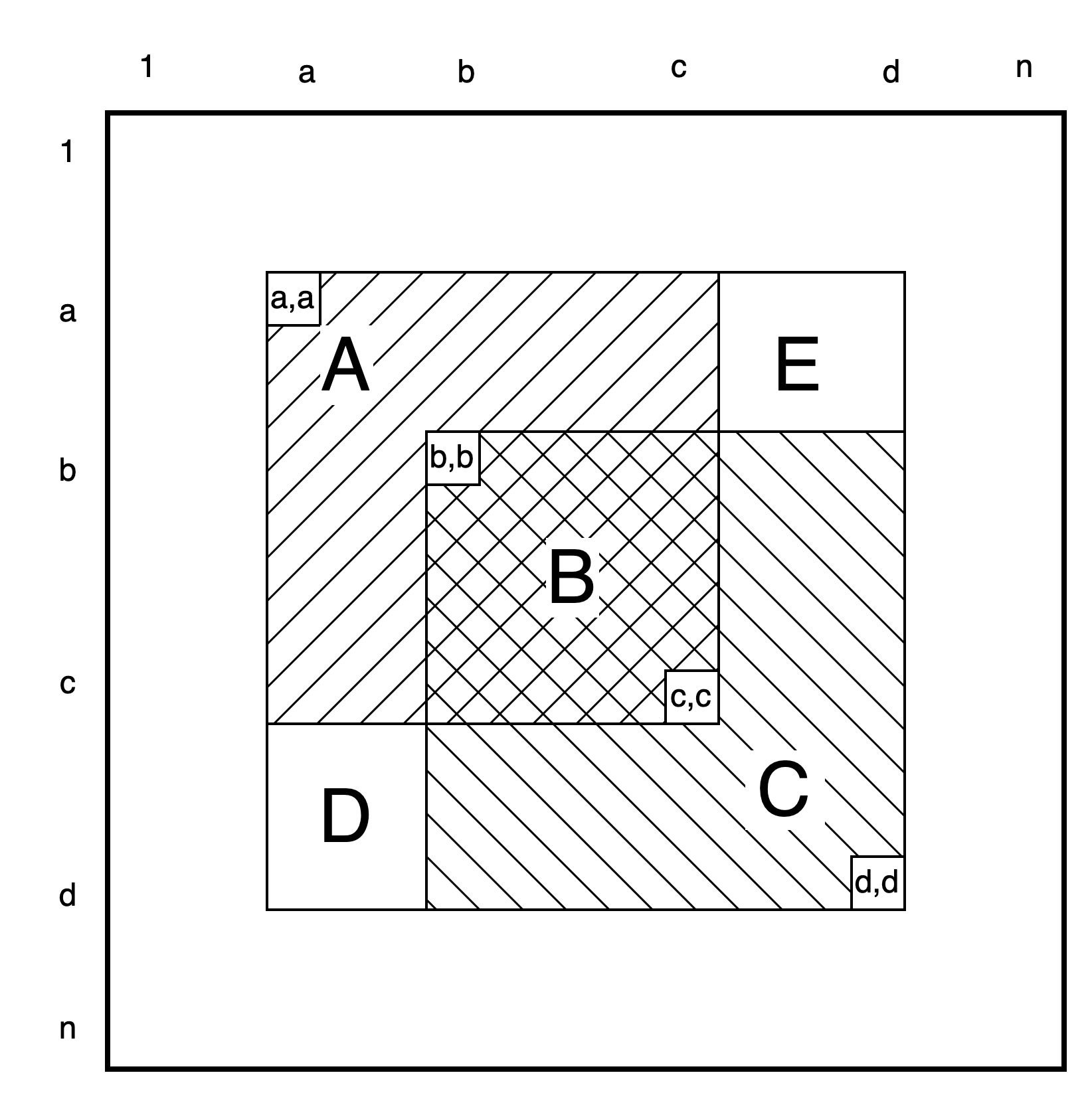


Рис. 2.19. Розбиття матриці

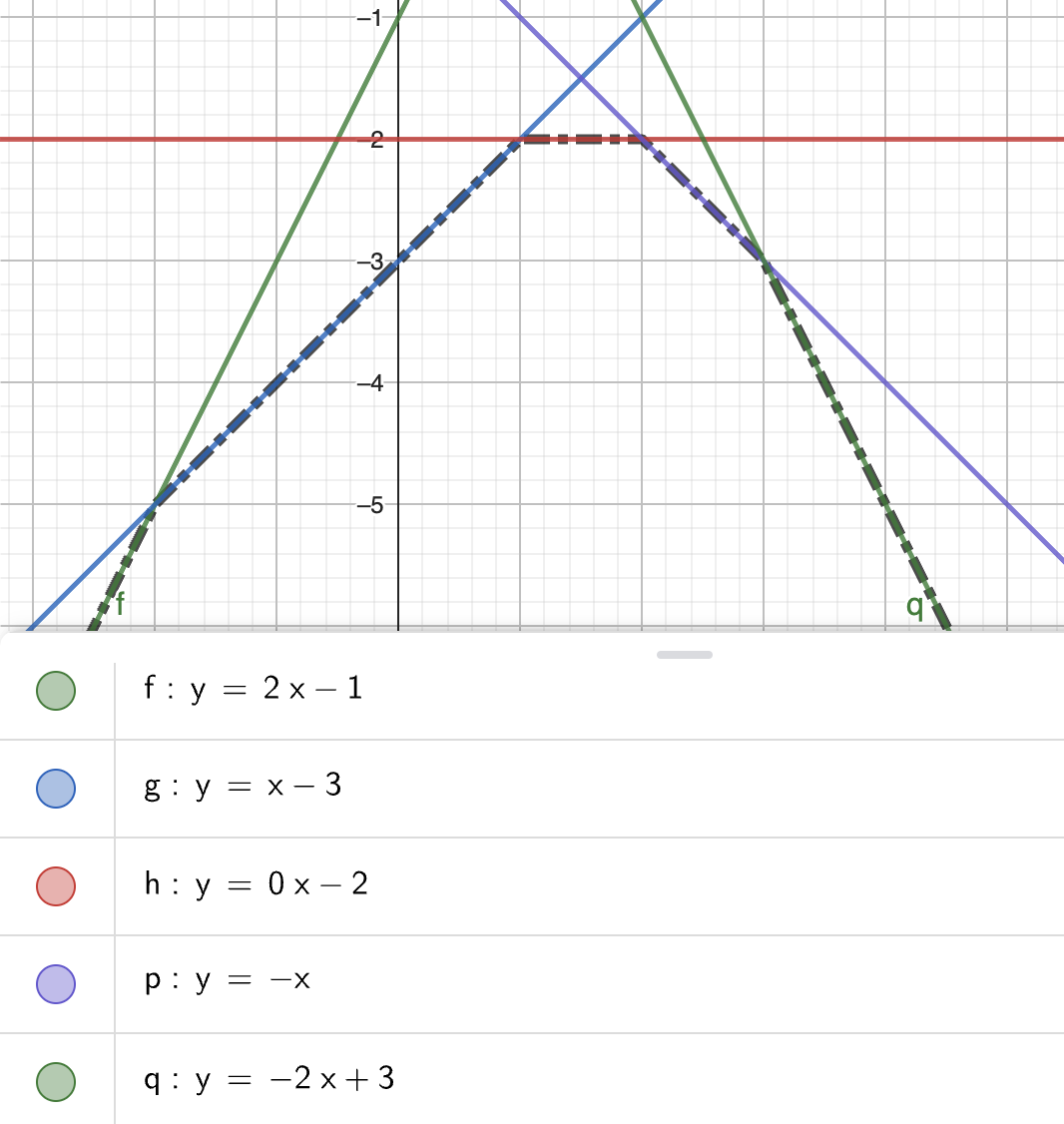
* — сума на всіх елементів для яких виконується , тобто
* — сума на всіх елементів , для яких , тобто ).
* — сума на всіх елементів для яких виконується , тобто
* — сума всіх елементів для яких виконується та .
* — сума всіх елементів для яких виконується та .

Тоді . Нерівність набуває наступного виду:

Оскільки, всі елементи матриці є невід'ємними, то значення . Отже нерівність чотирикутника завжди виконується.

## 2.5. Convex hull trick

Convex hull trick — це один з методів оптимізації динамічного програмування, який використовує ідею опуклої оболонки. За його допомогою можна покращити складність знаходження значення одного стану динамік, що обчислюються за формулою або ж , з лінійної на логарифмічну. Основний принцип оптимізації полягає в конструюванні та підтримці опуклої оболонки для множини прямих виду та знаходженні оптимального значення для заданого яке належатиме цій оболонці. Розглянемо використання цієї оптимізації на прикладі задачі.



Риc. 2.20. Приклад опуклої оболонки 5 прямих для знаходження мінімуму

### Frog 3

*Джерело: https://atcoder.jp/contests/dp/tasks/dp\_z*

Дано камінців, які пронумеровані числами . Для кожного , висота камінця рівна . Гарантується, що .

Жабка початково знаходиться на камені номер . Вона буде повторювати наступну дію певну кількість разів, доки не досягне каменю .

* Якщо жабка знаходиться на камені , вона може стрибнути на один з каменів . Ціна цього стрибка рівна .

Потрібно знайти мінімальну ціну яку потрібно сплатити, щоб дістатись від каменю до .

Обмеження. Кількість каменів , висоти , .

Розв’яжемо цю задачу за допомогою наступної динаміки. Нехай, — мінімальна ціна яку потрібно сплатити, щоб дістатись від каменю до . Його можна обраховувати за фомулою , де це камінь з якого відбувся стрибок на камінь . Наївне обрахування динаміки має складність . Якщо відкрити дужки у формулі то отримаємо

.

Можна побачити, що вираз є рівнянням прямої , де та . Тоді, потрібно вміти підтримувати структуру, яка може знаходити мінімум у певній точці серед всіх прямих та додавати нову пряму.

## 

Рис. 2.20. Псевдокод підрахунку

Варто звернути увагу, що гарантовано дані в зростаючому порядку, отже прямі додаються в порядку спадання коефіцієнтів та всі , тобто . Використовуючи цю властивість доведемо, що для набору прямих , які утворюють опуклу оболонку існує послідовність розбиттів () така, що для всіх з відрізку мінімум досягається на прямій , для всіх з відрізку мінімум досягається на прямій , і так далі до з відрізку мінімум досягається на прямій .

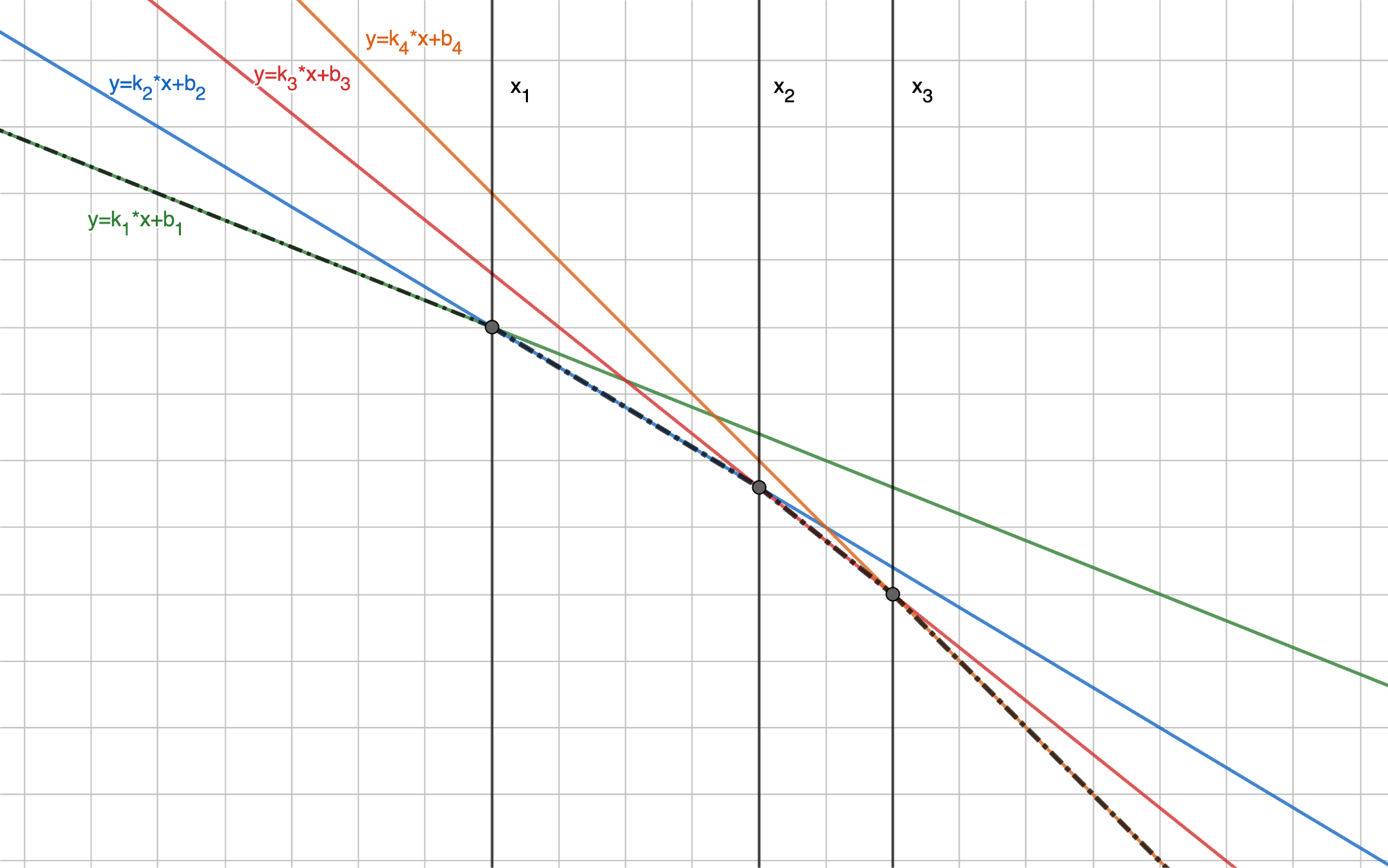


Рис. 2.21. Приклад розміщення точок перетину сусідніх прямих при побудові опуклої оболонки

На рисунку 2.21. на проміжку найнищою прямою є перша, на проміжку друга, третя, четверта.

Щоб це довести, доведемо, що , де — індекс прямої на якій досягається мінімум в точці та . Нехай та . Тоді виконується, що та . З першої нерівності: . З другої нерівності: , . Тоді , отже . Тобто виконується умова, що .

При побудові опуклої оболонки додаватимемо прямі одна за одною та перебудовувати суфікс послідовності розбиттів знаходячи точку розбиття між останньою прямою та тією яку додаємо. Тоді можливі 2 випадки:

* Точка розбиття , отже остання пряма в оболонці, не є мінімумом для жодного аргументу, то цю пряму потрібно видалити з оболонки.

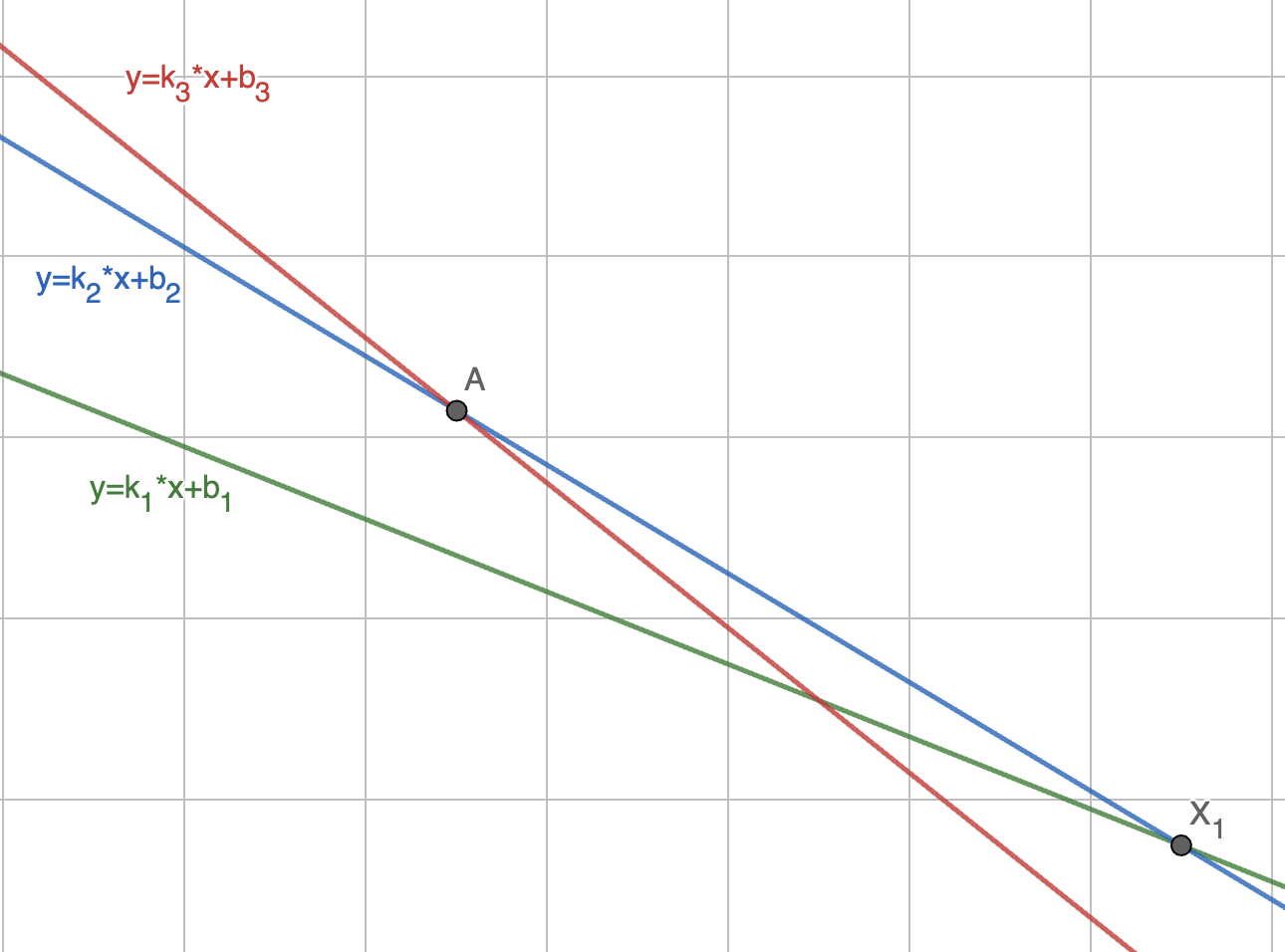


Рис. 2.22. Точка перетину першої та другої прямих знаходиться лівіше за точку перетину другої та третьої . Тут друга пряма ніколи не є найнищою

* Точка розбиття , тоді пряма яку ми додаємо є останньою в оболонці.

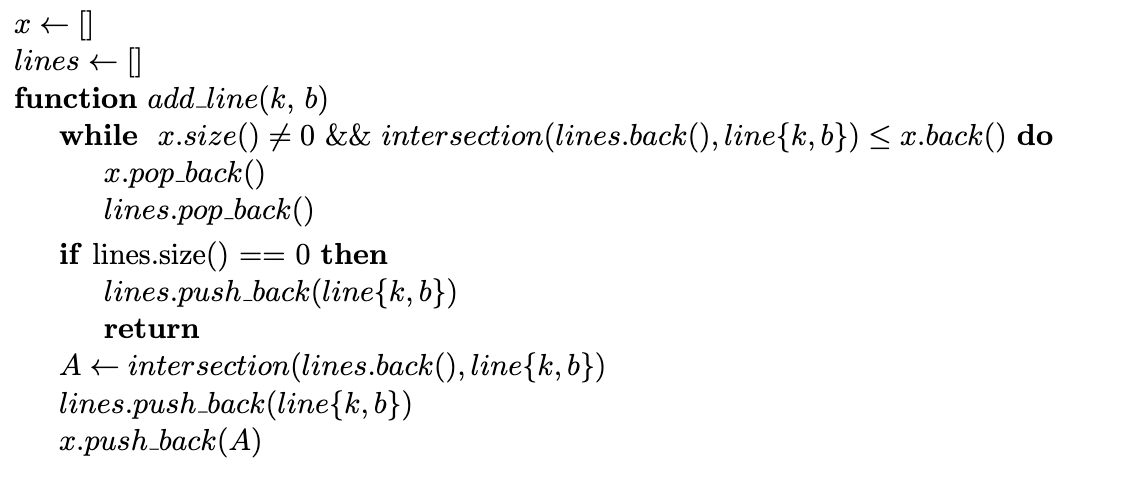


Рис. 2.23. Псевдокод функції додавання прямих до опуклої оболонки

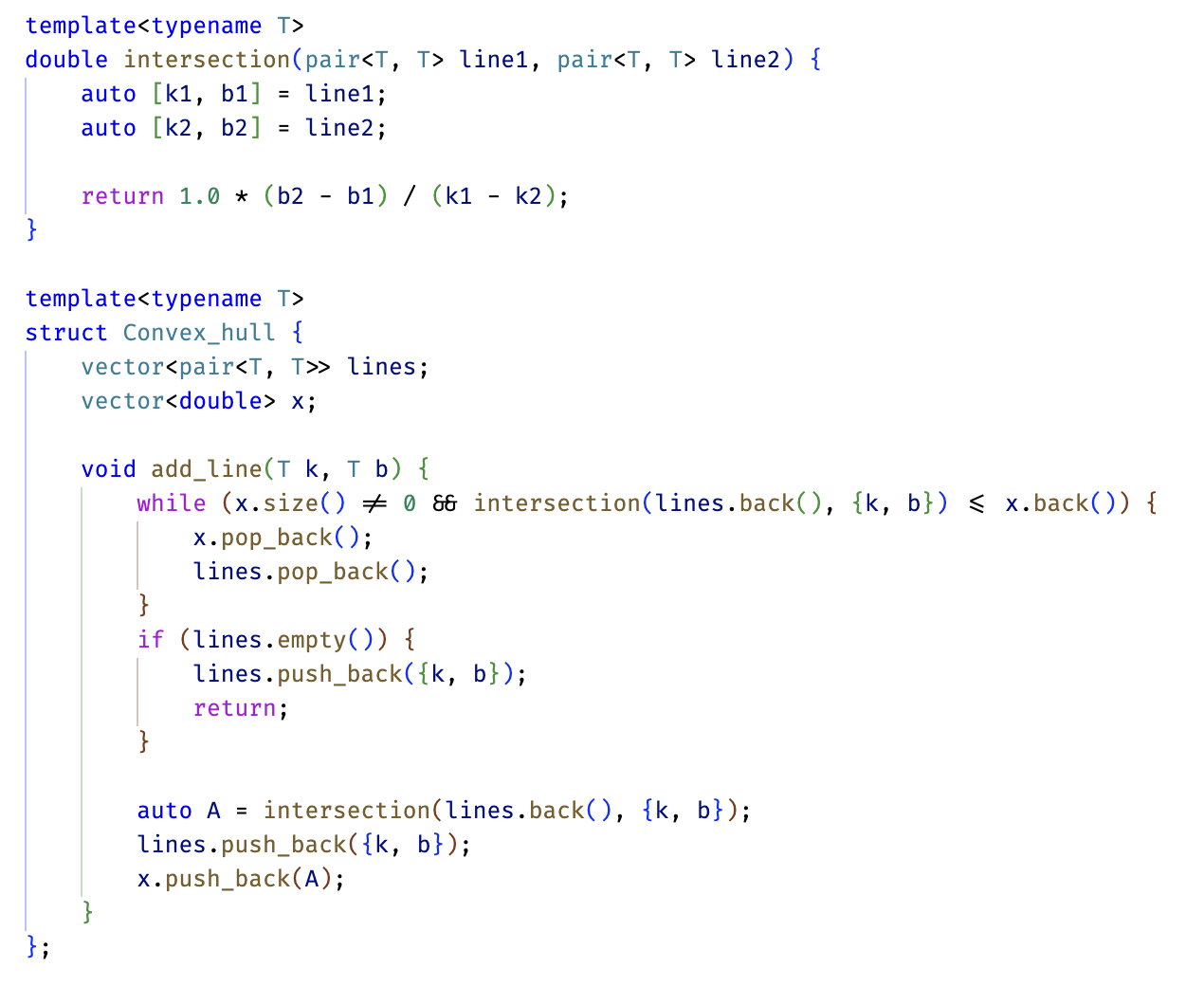


Рис. 2.24. Реалізація функції додавання прямих до опуклої оболонки мовою C++

Загальна складність роботи функції add\_line рівна кількості видалень з вектору . Оскільки кожна пряма додається рівно один раз, отже видаляється не більше одного разу. Тоді складність рівна .

Маючи всі точки розбиття відсортовані в порядку зростання, можна знаходити мінімум для заданої координати знайшовши індекс найлівішого для якого виконується, що . Це можна зробити за допомогою бінарного пошуку.

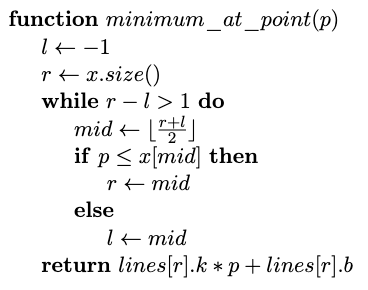


Рис. 2.25. Псевдокод функції знаходження мінімального значення

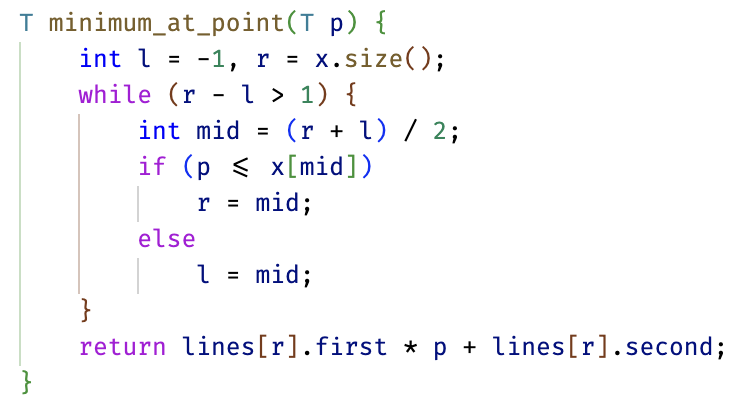


Рис. 2.26. Реалізації функції знаходження мінімального значення

мовою C++

Оскільки, ми зберігаємо масив , то замість написання бінарного пошуку можна скористатись можливостями STL(Standard Template Library) мови C++.

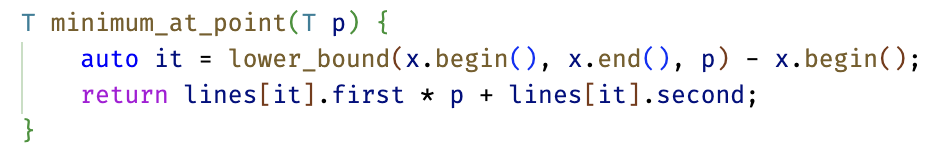


Рис. 2.27. Реалізації функції знаходження мінімального значення мовою C++ з використанням функції lower\_bound з STL

Складність порівняння двох чисел рівна , отже складність кожного виклику

Рішення задачі матиме такий вигляд:

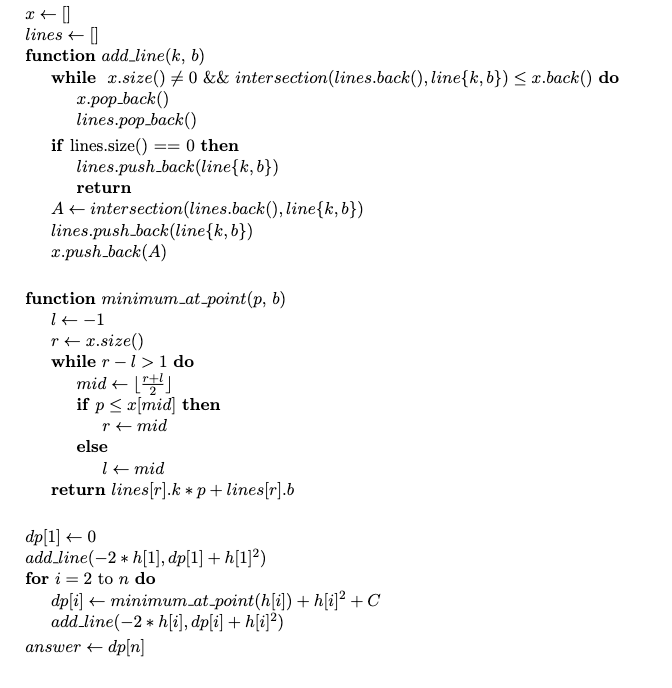


Рис. 2.28. Псевдокод повного рішення задачі

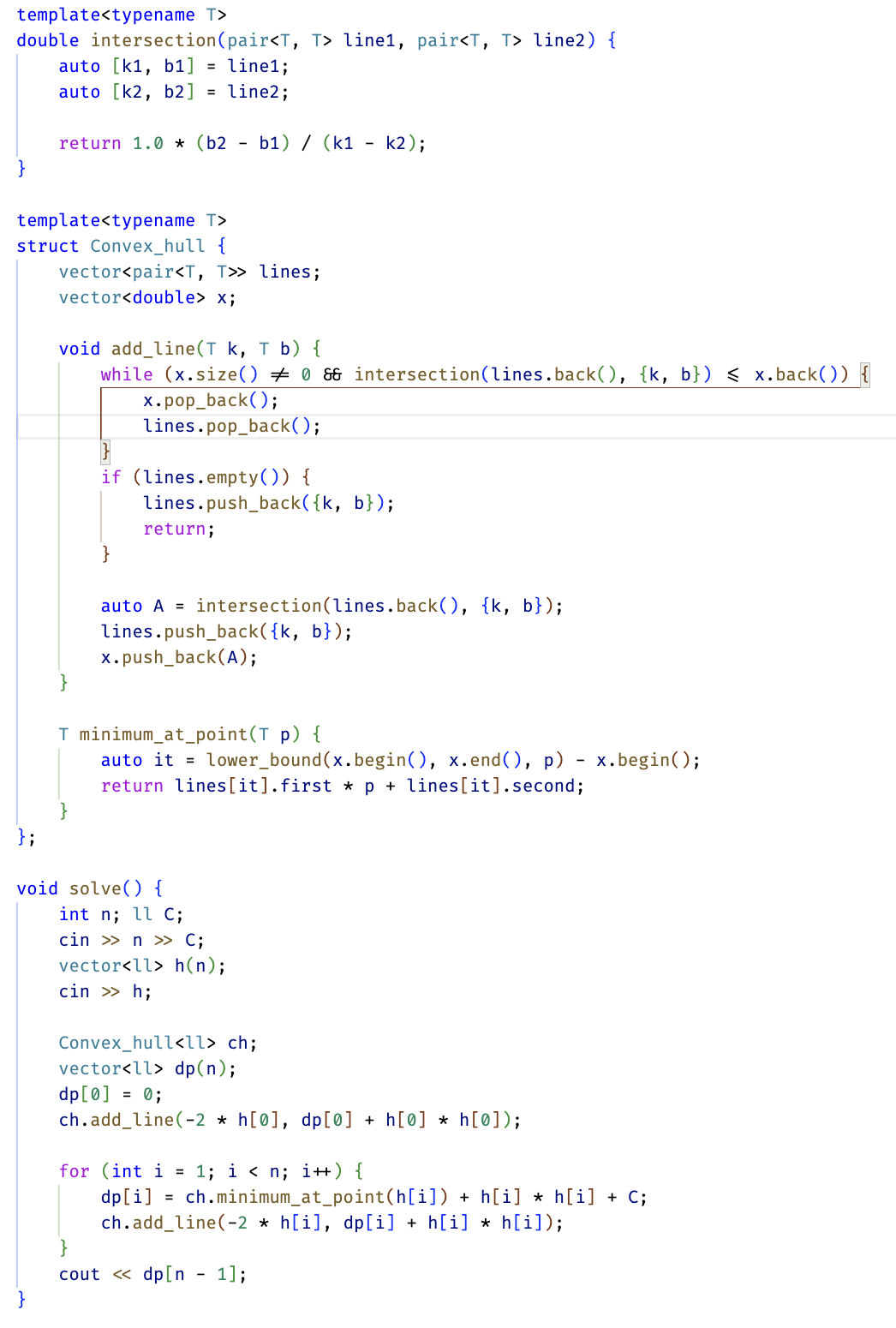


Рис. 2.29. Реалізація повного рішення задачі мовою C++

Дане рішення має складність . У системі тестування рішення проходить всі тести з максимальним часом 30 мілісекунд.

### Дерево Лі-Чао

Попередній метод побудови опуклої оболонки є простим та доволі швидким, проте вимагає монотонного впорядкування за кутом нахилу . У деяких задачах ця умова не виконується і потрібно використовувати інші методи побудови.

Одним з таких методів є структура даних — дерево Лі-Чао. Ця структура заснована на звичайному дереві відрізків, тому всі операції працюватимуть за неамортизований час, і це дерево можна робити персистентним, як і будь-яке інше дерево відрізків.

Дерево відрізків — це структура даних, яка в кожній вершині дерева зберігає інформацію для відповідного відрізка. Висота цього дерева рівна , де — кількість листків.

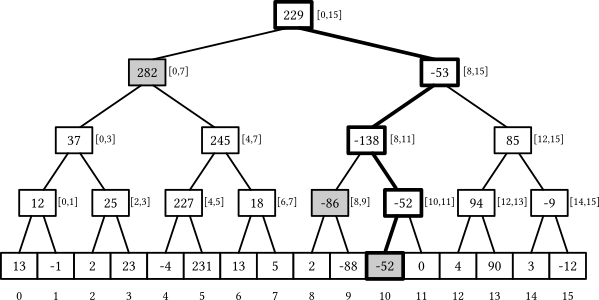


Рис. 2.30. Побудова дерева відрізків для 16 вершин

Будемо розв’язувати задачу, у якій є початково пуста множина прямих та потрібно виконувати два типи запитів:

* Додати лінійну функцію до множини
* Для заданого цілого () знайти мінімальне по всім прямим з множини .

Побудуємо дерево відрізків на координатах, тобто так, щоб кожен листок дерева відповідав одному значенню з відрізку . Тоді складність обробки одного запиту до структури рівна .

У кожній вершині дерева будемо зберігати одну лінійну функцію (для початку це), таким чином, щоб гарантувати, що на простому шляху від кореня до листка, одна з функцій на шляху, буде тією, що дає мінімальне значення у цьому листку. Тоді мінімум для заданого буде знаходитись, як мінімальне значення серед всіх прямих які зберігаються у вершинах дерева на простому шляху від кореня до листка який відповідає за цю точку.

Розглянемо спосіб за яким ми можемо це досягти. Ми хочемо додати нову пряму в деяку вершину дерева і перемістити інші значення в дереві так, щоб умова на те, що оптимальна пряма для кожної точки лежить на шляху від цієї точки до кореня, не зіпсувалася. Припустимо, що ми знаходимось у деякій вершині, яка відповідає за піввідрізок , — це функція яка зберігається зараз в цій вершині та — це функція яку ми додаємо.  Тоді точка перетину цих функцій буде знаходитись або в півінтервалі або , де . Оскільки дві прямі перетинаються тільки в одній точці, то точка перетину є єдиною. Якщо точка перетину міститься в лівій половині, то одна з функцій та є нижче іншої на всьому інтервалі , та навпаки, якщо перетин в правій половині відповідно.

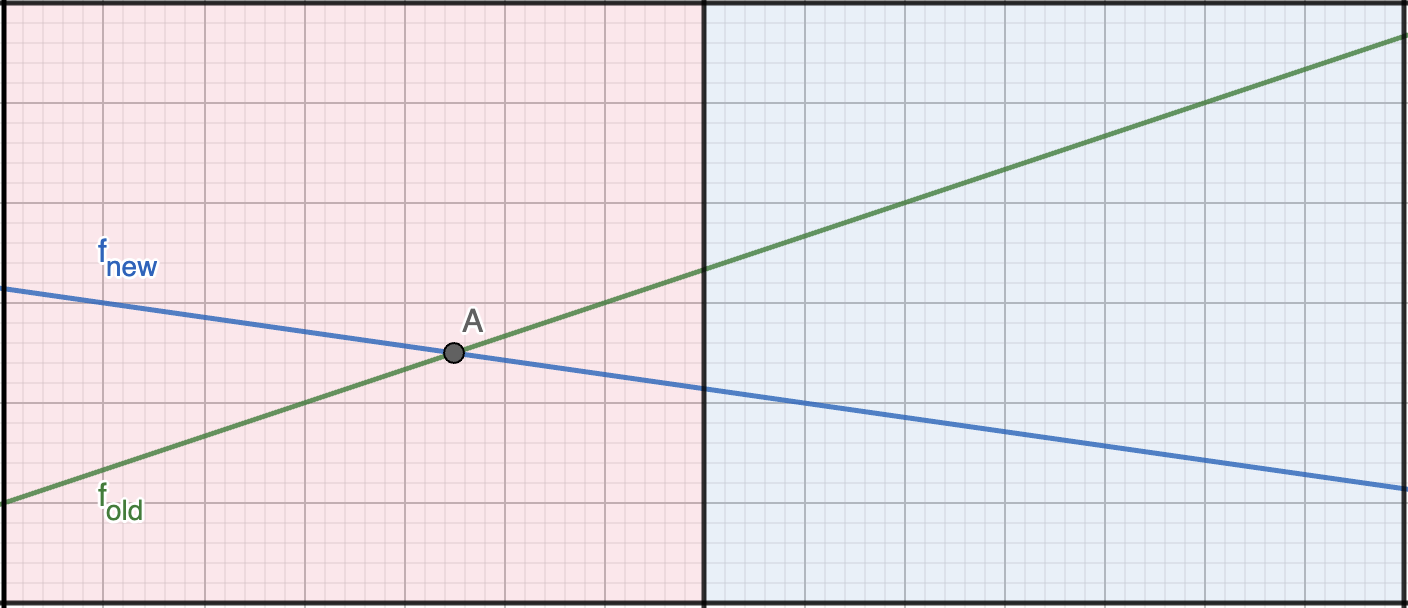


Рис. 2.31. Точка перетину функцій знаходиться в лівій половині заданої частини відрізку

Тепер, щоб коректність зберігалась для всіх точок які знаходяться на половини відрізка, що не містить точки перетину, ми виберемо нижчу функцію і запишемо її в поточну вершину. Можна побачити, що це завжди буде та функція, яка є нижчою в точці Після цього ми рекурсивно перейдемо до іншої половини відрізка з функцією, яка була верхньою. Це збереже коректність на першій половині відрізка, а на другій половині коректність буде збережено за рахунок рекурсивного виклику.

Щоб зрозуміти в якій половині знаходиться точка перетину(без використання типів з плаваючою комою, щоб зберегти точність) достатньо розглянути наступні випадки:

* Одна з функцій є нижчою, як в точці так в точці , тоді точка перетину в правій половині
* Інакше точка перетину в лівій половині.

Тоді реалізація додавання функції:

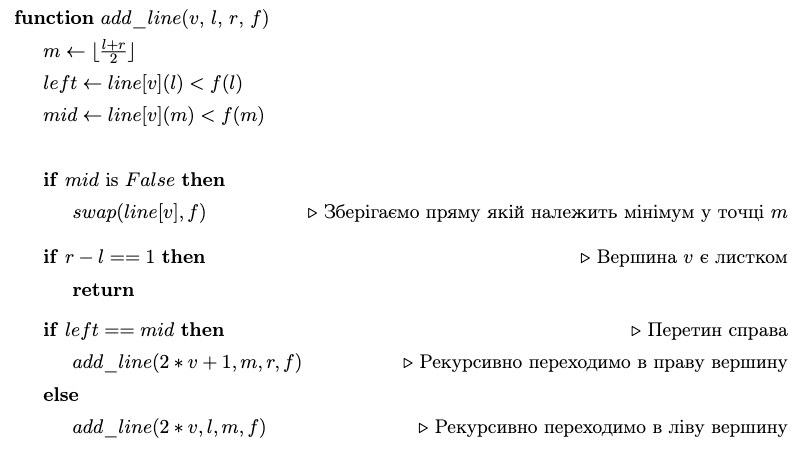


Рис. 2.32. Псевдокод додавання функції до дерева Лі-Чао

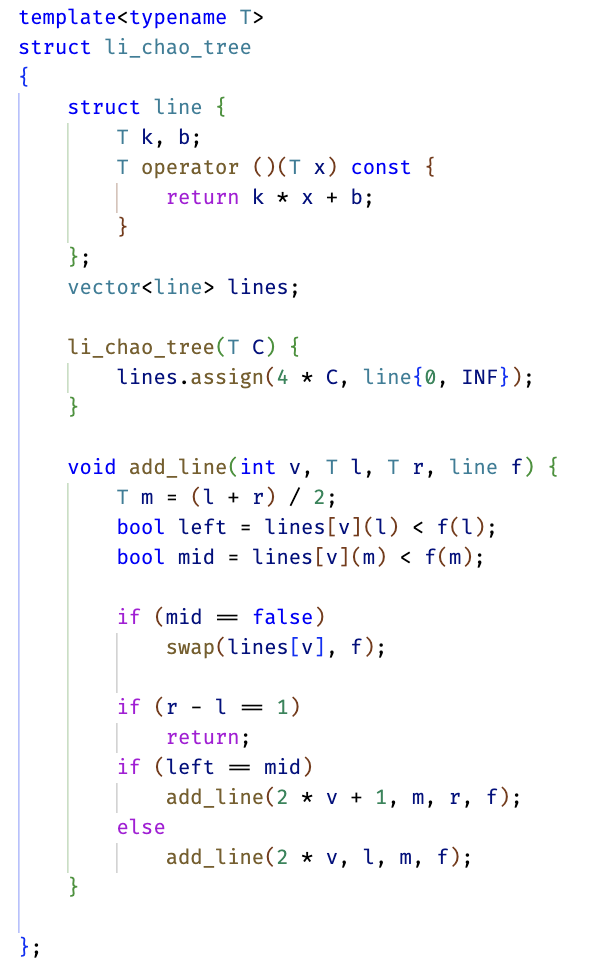


Рис. 2.33. Реалізація додавання функції до дерева Лі-Чао мовою C++

Щоб додати пряму до дерева, потрібно викликати функцію add\_line для кореневої вершини.

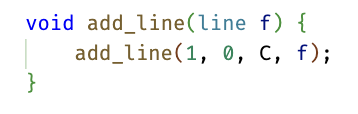


Рис. 2.34. Виклик функції додавання прямої мовою C++

Складність додавання однієї прямої рівна висоті дерева, а саме . Щоб, знайти мінімальне , пройдемось по всіх вершинам на шляху від кореня до потрібного листка.

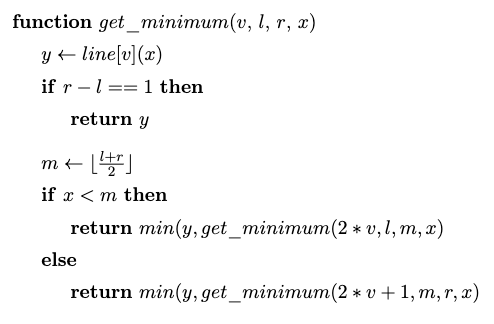


Рис. 2.35. Псевдокод знаходження мінімального значення для точки у дереві Лі-Чао

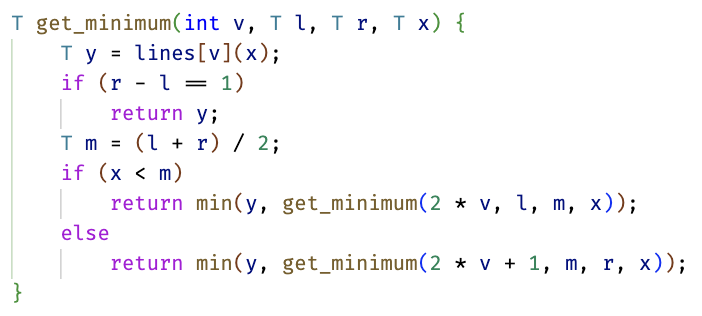


Рис. 2.36. Реалізації функції знаходження мінімального значення для точки у дереві Лі-Чао мовою C++

Викликати цю функцію потрібно для кореня дерева.

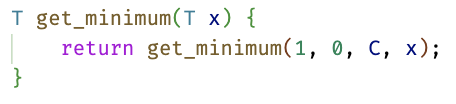


Рис. 2.37. Виклик функції знаходження мінімального значення для точки у дереві Лі-Чао мовою C++

Складність кожного запиту також рівна , оскільки такою є висота дерева.

Розв’язок задачі Frog 3 за допомогою дерева Лі-Чао має наступний вигляд.

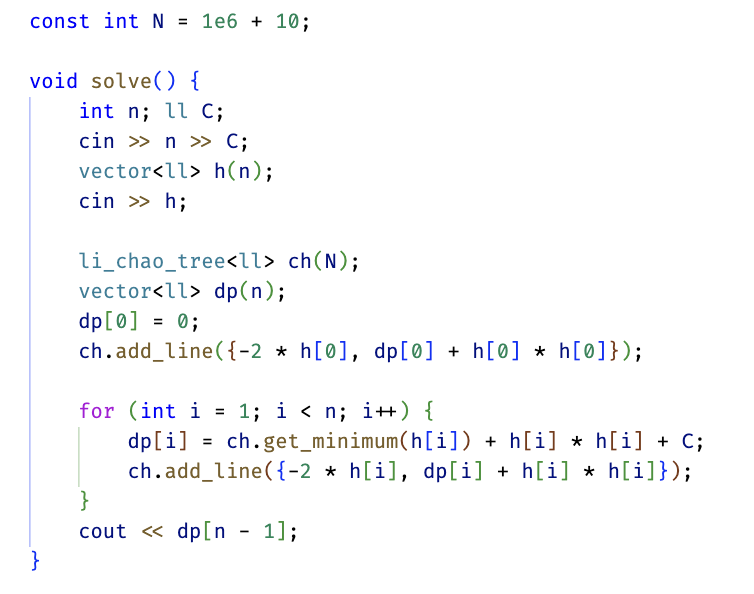


Рис 2.37. Розв’язок задачі Frog 3 за допомогою дерева Лі-Чао мовою C++

Дерево відрізків є дуже гнучкою структурою та підходить під різні адаптації. Так, у випадку коли значення є дуже великим числом, то структуру можна реалізовувати використовуючи ідею неявного дерева відрізків, тоді складність виконання запиту не зміниться, а кількість використаної пам’яті можна оцінити як , де — кількість запитів. Якщо точки є дійсними, а не тільки цілими, можна зробити спуск у дереві до потрібної точності . У такому випадку складність обробки одного запиту рівна Також структуру можна модифікувати, так, щоб була можливість виконувати запити пошуку для будь-якого з її попередніх станів за , тобто зробити її персистентною.

Варто зауважити, що дерево Лі-Чао можна використовувати не тільки для лінійних функцій, а для будь-яких функцій для яких гарантується, що вони перетинаються не більше ніж в одній точці.

Розглянемо використання неявної реалізації дерева на прикладі наступної задачі.

### Втеча через листки

*Джерело: https://codeforces.com/contest/932/problem/F*

Дано дерево з вершин (пронумерованими від до ) з коренем у вершині . З кожною вершиною пов'язано два значення. Значеннями для -ої вершини є та .

Ви можете перейти з вершини до будь-якої вершини у її піддереві. Вартість одного переходу з вершини до вершини дорівнює добутку на . Загальна вартість шляху, утвореного одним або декількома переходами, дорівнює сумі вартостей окремих переходів. Для кожної вершини обчисліть мінімальну сумарну вартість досягнення будь-якого листка з цієї вершини. Зверніть увагу, що корінь ніколи не може бути листком, навіть якщо він має степінь .

Зверніть увагу, що ви не можете переходити від вузла до нього самого.

Обмеження: кількість вершин значення у вершинах та .

Розглянемо розв’язок цієї задачі методом динамічного програмування. Нехай, — мінімальна ціна дістатись з вершини до будь-якого з листків, які знаходяться у піддереві вершини . Якщо вершина є листком то . Інакше, можна зробити стрибок в одну з вершин піддерева, а оптимальний шлях з неї є меншою підзадачею.

Тоді отримуємо перехід . Щоб порахувати цю динаміку, потрібно гарантувати, що для всіх вершин у піддереві є обрахованим, перед знаходженням значення для вершини. Щоб це виконувалось, проходитимось по всіх вершинах дерева за допомогою алгоритму DFS (Depth-first search — алгоритм пошуку у глибину).

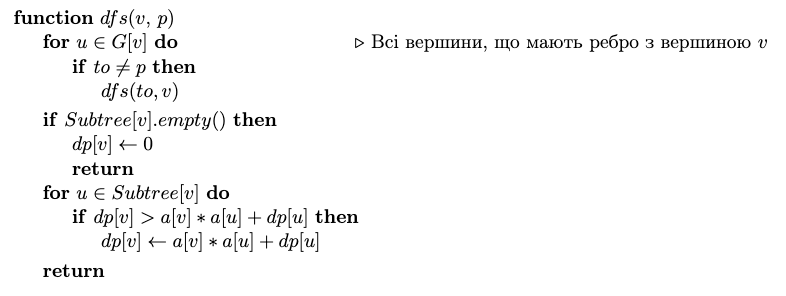


Рис 2.38. Псевдокод наївного розв'язку

Складність такого наївного підрахунку рівна . Для її покращення використаємо дерево Лі-Чао та техніку “Small to large” (“З меншого в більше”) [8]. Для кожної вершини , будемо зберігати дерево, яке містить всі прямі де — вершина з піддерева та повертатимемо його з виклику функції. Позначимо як дерево Лі-Чао для вершини . Щоб побудувати , пройдемось по всіх синах вершини , та додамо до прямі з кожної вершини . Оскільки, через алгоритм побудови структури, кожна пряма зберігається рівно в одній вершині, для побудови ми використаємо не більше прямих.

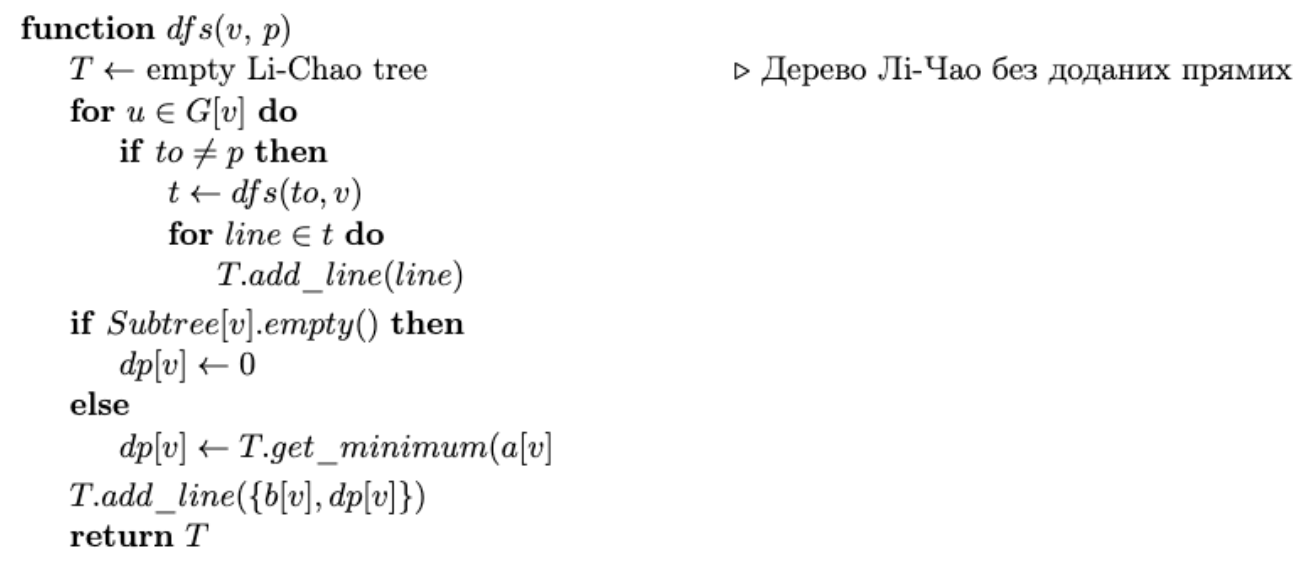


Рис 2.39. Псевдокод розв’язку з використанням дерева Лі-Чао

Складність такої реалізації рівна для побудови всіх дерев та для знаходження усіх мінімальних значень . Щоб оптимізувати побудову, використаємо техніку “Small to large”. Суть цієї техніки заключається в тому, що якщо додавати прямі з меншої структури до більшої структури, то загальну кількість додавань можна оцінити як

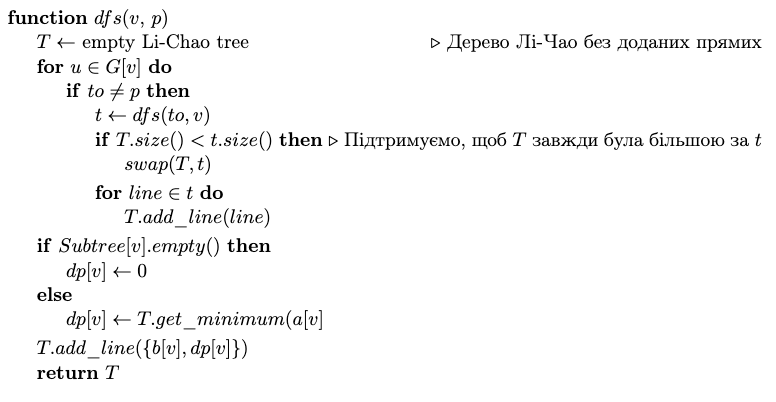


Рис 2.40. Псевдокод розв’язку з використанням дерева Лі-Чао та оптимізації “Small to large”

Доведемо, що кількість операцій можна оцінити як При об'єднанні двох структур ми передаємо елементи від меншої до більшої множини. Якщо розмір меншої множини дорівнює , то розмір результуючої множини буде не менше . Таким чином, елемент, який було переміщено разів, опиниться у множині розміром не менше , а оскільки максимальний розмір множини рівний (для кореня дерева), то кожен елемент буде переміщено не більше разів.

Неявна реалізація дерева Лі-Чао буде мати наступний вигляд.

# 

Рис 2.41. Реалізація неявного дерева Лі-Чао мовою C++

Щоб зберігати найбільшу структуру без перевірки розмірів, можемо почати з найбільшого, за розміром піддерева, сина. Отримаємо наступний розв’язок задачі.

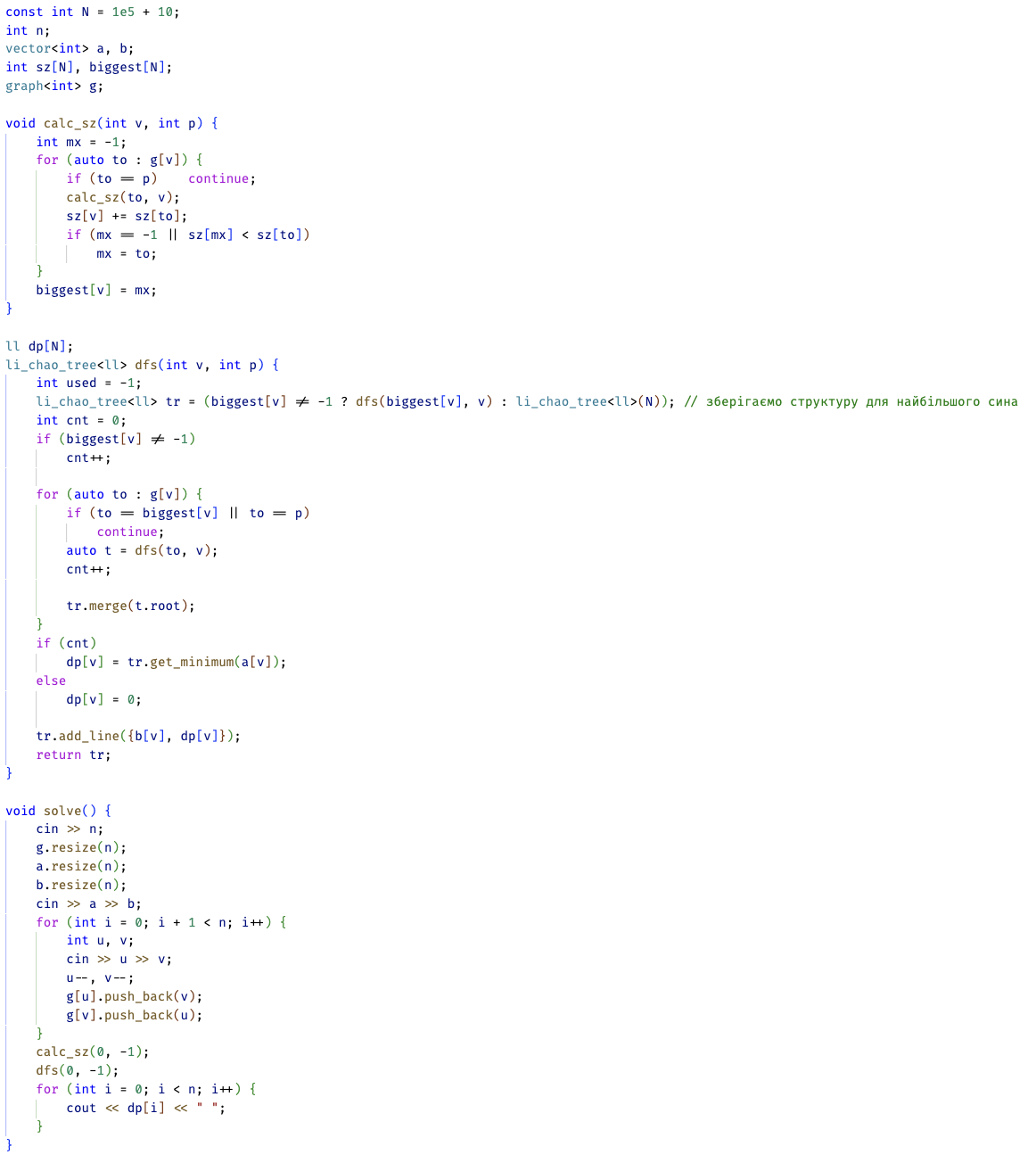


Рис. 2.38. Повний розв’язок з використанням дерева Лі-Чао та оптимізації “Small to large” мовою C++

Загальна складність такого рішення рівно , оскільки кожне з має складність .

# 

# ВИСНОВКИ

У даній роботі було проведено детальний огляд методів оптимізації динамічного програмування, включаючи такі техніки як оптимізація Кнута-Яо, метод "Розділяй та володарюй", Convex Hull Trick, дерево Лі-Чао, а також різноманітні методи оптимізації за допомогою спеціалізованих структур даних. Також розглянута нерівность чотирикутника, яка слугує одним з методів для доведення коректності використання оптимізації Кнута-Яо і методу "Розділяй та володарюй".

Було розв’язано кілька прикладних задач із використанням зазначених оптимізаційних методів, і надано їх реалізації на мові програмування C++. Ці реалізації були перевірені за допомогою сучасних систем тестування, таких як Codeforces, AtCoder та CSES, що дозволило забезпечити їхню коректність та ефективність.

Крім того, було проведено порівняння і аналіз обчислювальної складності розв'язків, отриманих із застосуванням оптимізацій, у порівнянні з базовими алгоритмами, які не використовують ці оптимізації. Результати показали значне покращення у швидкодії, що підтверджує ефективність цих методів.

Таким чином, було продемонстровано важливість та корисність оптимізацій у динамічному програмуванні, а також їхню здатність значно підвищувати продуктивність алгоритмів у реальних умовах.

# 

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C., Introduction to Algorithms (2nd ed.), MIT Press & McGraw–Hill, 2001. 344 с. Дата звернення: 10 березня, 2024.
2. Laaksonen, A. Competitive Programmer’s Handbook [Електронний ресурс]. Доступно: <https://cses.fi/book/book.pdf>. 89 c. Дата звернення: 14 березня, 2024.
3. Segment Tree [Електронний ресурс]. Доступно: <https://cp-algorithms.com/data_structures/segment_tree.html>. Дата звернення: 29 березня, 2024.
4. Оптимізація Кнута-Яо [Електронний ресурс]. Доступно: <https://peltorator.org/posts/knuth_yao_optimization/>. Дата звернення: 3 квітня, 2024.
5. Divide and Conquer DP [Електронний ресурс]. Доступно: <https://cp-algorithms.com/dynamic_programming/divide-and-conquer-dp.html>. Дата звернення: 7 квітня, 2024.
6. Нерівність чотирикутника [Електронний ресурс]. Доступно: <https://peltorator.org/posts/quadrangle_inequality/>. Дата звернення: 16 квітня, 2024.
7. F. Frances Yao, Efficient Dynamic Programming Using Quadrangle Inequalities [Електронний ресурс]. Доступно: <https://cse.hkust.edu.hk/mjg_lib/bibs/DPSu/DPSu.Files/p429-yao.pdf>. Дата звернення: 16 квітня, 2024.
8. Laaksonen, A. Competitive Programmer’s Handbook [Електронний ресурс]. Доступно: <https://cses.fi/book/book.pdf>. 170 c. Дата звернення: 2 травня, 2024.