**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ТАРАСА ШЕВЧЕНКА**
Факультет комп'ютерних наук та кібернетики
Кафедра теорії та технології програмування

# Кваліфікаційна робота

**на здобуття ступеня бакалавра**
за спеціальністю 122 Комп'ютерні науки
на тему:

# КОРЕНЕВІ ОПТИМІЗАЦІЇ ПРИ РОБОТІ З ВЕЛИКИМИ ОБСЯГАМИ ДАНИХ

Виконав студент 4 курсу
Фекете Іван Іванович _____
(підпис)

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Зубенко Віталій Володимирович _____
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

_____
(підпис)
Студент

Роботу розглянуто й допущено до захисту
на засіданні кафедри теорії
та технології програмування
«____» _____ 20__ р.,
протокол № ___
Завідувач кафедри
М. С. Нікітченко _____
(підпис)

Київ - 2021

# ЗМІСТ

**РЕФЕРАТ**

Обсяг роботи 43 сторінки, 6 ілюстрацій, 2 таблиці, 9 джерел посилань.

АСОЦІАТИВНА ОПЕРАЦІЯ, АЛГОРИТМ, ДЕКОМПОЗИЦІЯ, ДОСЛІДЖЕННЯ, КВАДРАТНИЙ КОРІНЬ, МАСИВ, ОБРОБКА ЗАПИТІВ, ОПТИМІЗАЦІЯ, СТРУКТУРА ДАНИХ, ШВИДКОДІЯ.

Об’єктом роботи є різновид оптимізацій під назвою кореневі оптимізації. Предметом роботи є реалізації кореневих оптимізацій та дослідження ресурсів, які витрачають такі реалізації.

Метою роботи є дослідження ефективних алгоритмів кореневих оптимізацій, визначення класу задач, для яких застосування даних оптимізацій є можливим та оптимальним.

Методи розроблення: алгоритмічне програмування, методи кореневих оптимізацій. Інструменти розроблення: безкоштовне, вільно поширюване в навчальних цілях інтегроване середовище розробки CLion 2019.2.2, мова програмування C++.

Результати роботи: досліджено структуру кореневої декомпозиції, описано алгоритм її роботи, вирахувано асимптотичну складність роботи, досліджено різноманітні можливості структури, вказані переваги та недоліки. Визначено область задач, які є розв’язними за допомогою кореневих оптимізацій та для яких вони становлять теоретичний та практичний інтерес.

# ВСТУП

**Оцінка сучасного стану об'єкта дослідження.** Структури даних, що дозволяють ефективно працювати з великими обсягами даних не є чимось новим для науки. За всю історію існування структур даних було розроблено багато ефективних способів зберігати дані в бінарних деревах, хеш-таблицях, B-деревах тощо. Найпоширеніші з цих структур вже давно є частинами стандартних бібліотек мов C++, Java, Python та багатьох інших, а якщо виникає потреба в модифікації дерев пошуку для певних задач способом, що не є передбаченим в стандартних реалізаціях, то на допомогу можуть прийти додаткові бібліотеки. Проте, існує достатньо великий клас задач, для яких вищенаведені підходи не спрацюють, адже витрачають занадто багато часу і пам'яті на підтримку необхідних даних в актуальному стані. І тут виникає проблема, адже підходи, які засновані на кореневих оптимізаціях, рідко піддаються зведенню до єдиної реалізації. Також поширеною проблемою є те, що стандартні структури даних не завжди дозволяють ефективно розподіляти обчислення між певною кількістю потоків, що може значно пришвидшити роботу програми. Однією із причин може бути складна ієрархічна будова більшості структур даних, що ускладнює розподіл обчислення на незалежні між собою частини, результат яких потім можна буде легко об'єднати.

**Актуальність роботи та підстави для її виконання.** Ефективна обробка даних являється однією з основних задач програмування. Для вирішення цієї проблеми часто використовують різноманітні структури даних. Часто досягнути прийнятної асимптотики для відповіді на запити суми, максимуму, мінімуму чи складніших операцій серед певних елементів динамічного масиву є нетривіальною задачею. Єдиного підходу до вивчення структур даних для обробки масивів ще немає, тому дана проблема є актуальною та багатогранною для подальшого розвитку.

**Метою й завданням роботи** є проведення дослідження з питань, пов’язаних із можливостями швидкої обробки масивів за допомогою кореневих оптимізацій.

Для досягнення поставленої мети в роботі визначені наступні **завдання:**

- окреслити теоретичні відомості застосування кореневих оптимізацій для обробки масивів;
- розглянути роботу з декомпозиціями масивів;
- охарактеризувати базове завдання та побудову кореневої декомпозиції;
- реалізувати запити зміни елемента, розвороту інтервала та запити на знаходження результату асоціативної операції за допомогою мови програмування C++;
- здійснити оцінку ресурсів часу та пам’яті, використовуваних кореневою декомпозицією;
- дослідити нестандартні способи кореневих оптимізацій, окреслити їх переваги та недоліки.

**Об’єкт дослідження:** методи кореневих оптимізацій.

**Предмет досліджень:** реалізації кореневих оптимізацій.

**Можливі сфери застосування.** Практична цінність дослідження полягає в можливості використання одержаних результатів для більш глибокого розуміння сутності структури кореневої декомпозиції, що сприятиме зростанню ефективності управління даними процесами через підвищення вірогідності та реальності оцінки ресурсів часу та пам’яті, використовуваних кореневою декомпозицією. Сама структура даних може широко використовуватися в комерційній розробці у випадках, коли потрібна висока швидкодія роботи з великими інтервалами даних, впорядкованих за певним індексом. Матеріали наведеної роботи мають теоретичне та практичне значення для науковців, які цікавляться даною проблематикою та

інженерів-програмістів, для яких швидка обробка, оновлення та отримання інформації про великі обсяги даних є важливою, зокрема, тих, кого цікавить багатопоточне програмування та структури даних, які ефективно взаємодіють із ним.

# РОЗДІЛ 1.ТЕОРЕТИЧНІ ВІДОМОСТІ ЩОДО ЗАСТОСУВАННЯ КОРЕНЕВИХ ОПТИМІЗАЦІЙ ДЛЯ ОБРОБКИ МАСИВІВ

На практиці часто виникає необхідність в обробці даних у вигляді довільного набору значень, тобто масивів. **Масив** — це кінцева іменована послідовність величин одного типу, які розрізняються за порядковим номером [1].

**Декомпозиція** — це метод, в основі якого лежить заміна виконання об'ємного завдання шляхом виконання ряду простіших завдань, які є пов'язаними між собою відповідно до структури початкового завдання. Декомпозиція, як процес розщеплення, розглядає кожну систему, що підлягає розділенню, як комплекс, що містить окремі підсистеми, пов'язаних між собою, які в свою чергу можуть бути відокремлені в частинах(схематично зображено на рисунку 1.1). Системи в даному випадку — це процеси, явища, концепції та інші абстракції.[2]

При декомпозиції використовуються такі принципи:

- кожне розділення створює новий рівень розгалужень;
- єдине ціле розділюється за однією, спільною для всіх рівнів, умовою;
- відокремлювані частини разом мають описувати всю структуру в цілому;
- глибина ієрархії визначається вимогами осяжності та легкості розуміння отриманої системи.

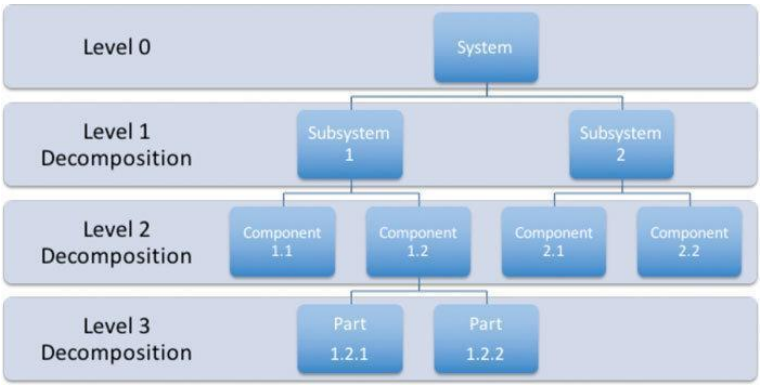


Рисунок 1.1 — Ієрархічна структура декомпозиції

На практиці будь-яка складна система для свого представлення використовує декомпозиції. Це зумовлено потребою структуризації інформації, розбиття її на менші частини, легші для сприйняття та обробки. В інформатиці втілення основних принципів декомпозиції також можна спостерігати у всіх її сферах: розбиття програм на незалежні модулі, структурне програмування, об'єктно-орієнтоване програмування, тощо.

**Декомпозиція даних** — це розбиття певного структурованого набору даних на рівноправні частини. Декомпозиція даних є одним із етапів розробки паралельних програм, який дозволяє процесам одночасно працювати із виділеними їм блоками пам'яті без створення критичних секцій та блокувань.

Також декомпозиція даних відбувається, якщо через великі обсяги вони фізично не можуть зберігатися в одному датацентрі. В такому випадку дані об'єднуються в єдине ціле на рівні абстракцій, а всі процеси, які їх обробляють, можуть бути розділені на окремі підпроцеси відповідно до сховища, в якому ці дані зберігаються. Це дозволяє виконувати ці процеси асинхронно на різних кластерах, що пришвидшує процес обробки порівняно із послідовним процесом.

**Препроцесинг** — попередня обробка даних з ціллю подати їх у формі, зручній для роботи наступного модуля.[3] Препроцесинг широко використовується веб-браузерами, компіляторами та багатьма іншими програмами, результат яких може бути повторно використаний.

Препроцесинг широко використовується в структурах даних, оскільки вони побудовані таким чином, щоб модифікація застосовувалася до якомога меншої кількості елементів структури, що означає, що після модифікації переважна частина інформації залишається без змін, тобто в тому вигляді, в якому вона була подана на етапі препроцесингу.

**Кореневі оптимізації** — це узагальнена назва різних методів і структур даних, які базуються на тому факті, що при розділенні множини із $n$ елементів на блоки по $\sqrt{n}$ елементів отримаємо приблизно $\sqrt{n}$ блоків. [4]

До найпоширеніших кореневих оптимізацій відносять:

- кореневу декомпозицію на масивах;
- кореневі евристики на запитах;
- алгоритм Мо;
- поділ на “легкі” і “важкі” об’єкти.

**Коренева декомпозиція** — це один із видів оптимізацій, який полягає у поділі масиву на частини, розміри яких приблизно дорівнюють $\sqrt{n}$. Такі частини прийнято називати **блоками**. В кожному із блоків може бути виконаний препроцесинг, який дозволяє швидко отримувати потрібні дані про масив на виділених відрізках масиву. При оновленні даних масиву потрібно перезапустит препроцесинг лише для блоків, які входять в зону впливу даного оновлення. Схема декомпозиції подана на рисунку 1.2.

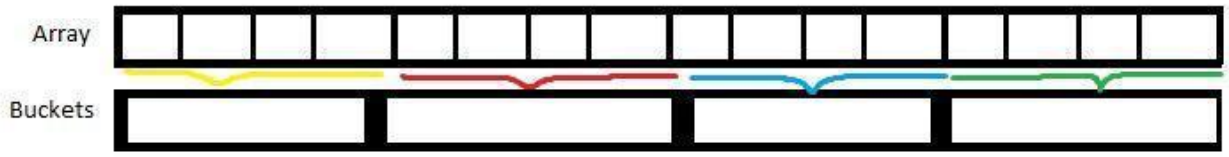


Рисунок 1.2 — Схема кореневої декомпозиції

# РОЗДІЛ 2. РОБОТА З ДЕКОМПОЗИЦІЯМИ МАСИВІВ

## 2.1. Базове завдання декомпозиції

Щоб розібратись у ідеї і особливостях роботи декомпозиції масивів найкраще перейти до конкретної задачі.

Розглянемо масив $a$ із $n$ елементів і $m$ запитів. Запити поділяються на три типи:

- зміна значення певного елементу масиву
- виведення кількості таких чисел в масиві $a_i$, що $l \le i \le r$ та $x \le a_i \le y$
- розворот елементів масиву з позиціями $l$, $l+1$, … $r$

*Формат вхідних даних:*

Перший рядок містить два цілі числа: $n$ і $m$ ($1 \le n, m \le 50\ 000$). В наступному рядку задано $n$ елементів масиву $a$ ($1 \le a[i] \le 10^9$). В наступних $m$ рядках задані запити у форматах:

"1 $x$ $y$" для запитів першого типу ($1 \le x \le n$, $1 \le y \le 10^9$)

"2 $l$ $r$ $x$ $y$" для запитів другого типу ($1 \le l \le r \le n$, $1 \le x \le y \le 10^9$)

"3 $l$ $r$" для запитів третього типу ($1 \le l \le r \le n$)

*Формат вихідних даних:*

Для кожного запиту другого типу потрібно з нового рядка вивести одне ціле число – відповідь на запит.

Обмеження часу роботи програми: 1 с.

Обмеження пам'яті: 256 МіБ.

*Підходи до розв'язку*

Реалізація поставленої задачі спонукає спробувати виконати всі запити на зміни, а для запитів другого типу проходитись по масиву від лівого елемента до правого. Однак, слід зауважити, що такий розв'язок не є оптимальним, адже в гіршому випадку він зробить $n^2$ операцій (коли всі

запити другого і третього типів будуть містити відрізок від першого до останнього елемента). Отже, його асимптотична складність $O(n^2)$.

Іншою ідеєю є побудувати дерево відрізків, кожна із вершин якого містить збалансоване бінарне дерево пошуку(червоно-чорне дерево, АВЛ-дерево, декартове дерево тощо).[5] Таким чином, на запити першого і другого типів можна буде відповідати зі складністю $O(log^2 n)$, але для запитів третього типу потрібно перебудовувати всю структуру дерева відрізків і його дерев-вузлів, що знову сповільнює алгоритм.

В усіх вищенаведених випадках випадках, при найбільших тестах програма буде працювати значно довше однієї секунди.

Скористаємося ідеєю про те, що масив можна розбивати на блоки однакової довжини. Позначимо цю довжину як $L$. В кожному блоці будемо зберігати, крім самих даних в порядку, в якому вони йдуть в масиві, ці ж дані, тільки впорядковані за неспаданням їх значень. Це дозволить відповідати нам на запит другого типу для конкретного блока за $O(log(L))$ операцій за допомогою бінарного пошуку[5].

Таким чином, щоб відповісти на запит другого типу, потрібно пройтися по всіх блоках, які входять в даний відрізок, і для кожного блока порахувати відповідь методом, що описаний вище, а для елементів, які не належать до блоків, які входять в проміжок цілком, виконаємо просту ітерацію по всіх значеннях таких елементів з подальшою перевіркою на відповідність умовам. Таких елементів буде не більше ніж $L-1$ з лівого кінця проміжку і не більше ніж $L-1$ з правого кінця. Отже, сумарна складність відповіді на запит другого типу складає $O(\frac{N}{L} log(L))$.

Для запиту першого типу тепер, окрім оновлення даних безпосередньо в масиві, потрібно оновити інформацію ще і у відповідному блоці. Для того, щоб видалити або вставити у відсортований масив один елемент, знадобиться $O(L)$ операцій.

Тепер для того, щоб обробити запит третього типу, потрібно зробити зміни в самій структурі декомпозиції. Базуючись на попередній спробі використати структури даних, засновані на бінарних деревах, це може бути досить трудомісткою процедурою, тому потрібно зробити так, щоб структура декомпозиції зазнала мінімальних змін при виконанні модифікацій для запиту третього типу. Правильним підходом буде можливість розвернути частину списку блоків, а розворот даних всередині конкретного блоку оптимізувати таким чином, щоб не виконувати явних змін порядку елементів всередині блоку і уникнути лінійної складності при обробці запитів.

Проблемою на шляху до можливості розвороту потрібних блоків є те, що їх структура поділу не відповідає межам відрізка більшості запитів. Тому правильним виходом була би можливість поділу того чи іншого блока за індексом, після чого розворот блоків стає тривіальною задачею. Складність поділу блока при цьому становитиме $O(L\,log(L))$ для кожного блока відповідно. Розворот даних всередині масиву матиме складність$O(\frac{N}{L})$.

Для розвороту даних всередині блока використаємо такий підхід: не будемо робити явних розворотів даних, а замість цього будемо зберігати прапорець, який і буде запам’ятовувати, в якому порядку потрібно ітеруватися по елементах блоку: в прямому чи зворотному. Це не вплине на складність жодного із методів блоку, оскільки один із них взагалі не аналізує, в якому порядку елементи знаходяться в блоці, а інший легко модифікувати згідно з описаними нами вище оптимізаціями, і позбавить нас потреби явно змінювати порядок елементів всередині блоку. Враховуючи цю а також попередні оптимізації, асимптотика обробки запиту третього типу складатиме $O(L\,log(L)\,+\frac{N}{L})$ на один запит.

Алгоритм для обробки запиту породив нову проблему: неконтрольований поділ блоків. З кожним таким поділом їх кількість буде рости, а розміри зменшуватися, що ускладнить прогнозованість часу обробки кожного із запитів. Для того, щоб уникнути цієї проблеми, рекомендується

кожні $L$ запитів перебудовувати декомпозицію з нуля. Таким чином, складність одного етапу перебудови становить $O(N\ log(L))$.

Надалі буде розглянуто детальніший опис реалізації окремих класів і функцій, потрібних для декомпозиції а також наведено приклади програм, написаних на C++.

## 2.2 Побудова декомпозиції

Наша декомпозиція складається з певної кількості структурних одиниць, які ми назвемо блоками. Блок, відповідно, складається із певної кількості елементів початкового масиву, які, до того ж, розміщені в ньому послідовно один за одним. В кожному блоці, окрім елементів, розміщених у тому порядку, в якому вони йдуть в масиві, ми будемо зберігати список таких же елементів, тільки впорядкованих за неспаданням, і прапорець, який буде позначати, як саме потрібно ітеруватися елементами блоку: в прямому чи зворотному порядку.

Отже, при створенні декомпозиції масиву нам, окрім самих елементів, потрібно знати, блоки яких розмірів у нас повинні бути(щоб розбити дані на блоки) і максимальну дозволену кількість блоків для декомпозиції(цей параметр потрібен для запиту 3 типу, його значення буде обговорено пізніше. На рисунку 2.1 зображена блок-схема побудови декомпозиції.

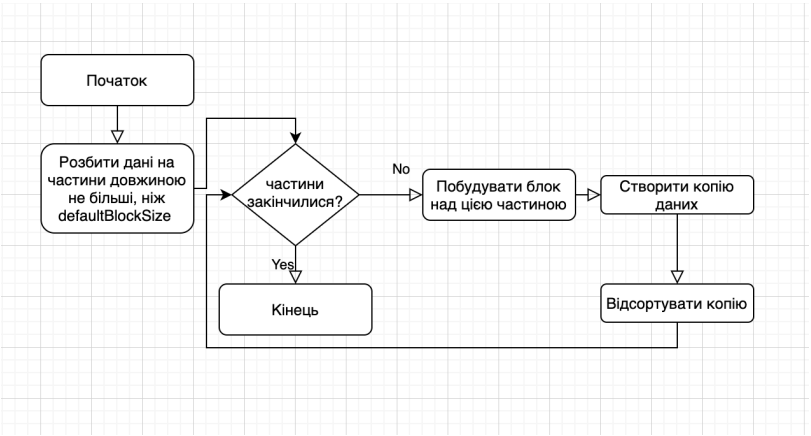


Рисунок 2.1 — Блок-схема побудови декомпозиції

Код реалізації побудови декомпозиції на C++:

```
Decomposition(vector<T> data, size_t defaultBlockSize, size_t capacity):
defaultBlockSize(defaultBlockSize), capacity(capacity) {
    decompose(data);
}
void decompose(vector<T> data) {
    vector<vector<T>> blocksData((data.size() + defaultBlockSize - 1) /
defaultBlockSize);
    for (size_t i = 0; i < data.size(); i++) {
        blocksData[i / defaultBlockSize].push_back(data[i]);
    }
    for (const auto& blockData: blocksData) {
        blocks.push_back(new Block(blockData));
    }
}
explicit Block(const vector<T>& data)
    : data(data), sorted(data), reversed(false) {
    sort(sorted.begin(), sorted.end());
}
```

### 2.3. Реалізація запиту зміни елемента

Даний запит стосується лише одного елемента, тому зачепить лише один блок, що означає, що цей запит можна реалізувати в класі блоку, а в класі, відповідальному за декомпозицію лише створити для нього метод-обгортку.

Щоб легко змінити елемент масиву, потрібно знайти його блок, в цьому блоці виявити його позицію і оновити дані як на позиції, так і у відсортованому масиві.

Блок-схема алгоритму зміни елементу наведено на рис. 2.2.

Код реалізації зміни одного елемента у декомпозиції на C++:

```
void updateElement(size_t position, T value) {
    position = getCorrectPosition(position);
    for (auto &item: sorted) {
        if (item == data[position]) {
            item = value;
            break;
        }
    }
    sort(sorted.begin(), sorted.end());
    data[position] = value;
}
```


Рисунок 2.2 — Блок-схема алгоритму зміни елементу

## 2.4. Реалізація запиту на знаходження кількості елементів у заданих межах на відрізку

Для цього запиту в нас буде два різновиди отримати відповідь: результат із блоку і ітерація по елементах. Як ми вже говорили раніше, перший спосіб швидший, тому за можливості ми будемо використовувати саме його.

Визначимо, в яких блоках знаходяться наші елементи. Для цих блоків ми зробимо пряму ітерацію по елементах, оскільки вони входять у наш відрізок лише частково, а для блоків, які знаходяться між ними, ми використаємо відсортовані масиви і знайдемо бінарним пошуком відповідь на запит, так як ці блоки входять у відрізок повністю.

Блок-схема алгоритму знаходження кількості елементів у заданих межах на відрізку наведено на рис. 2.3.

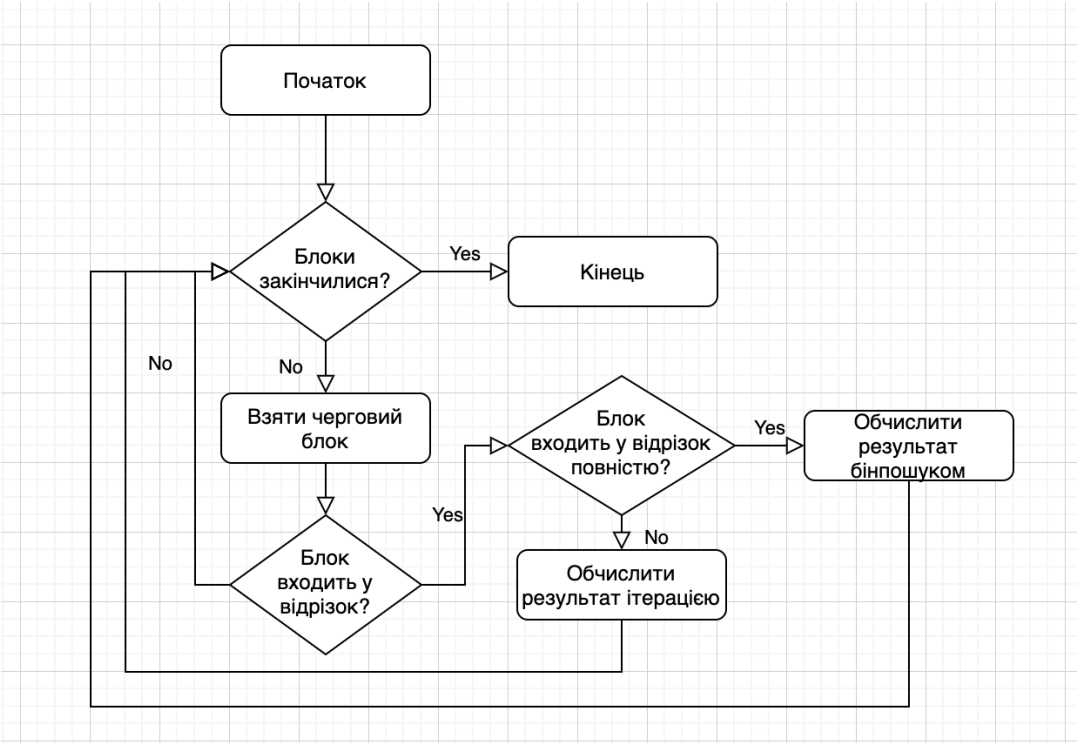


Рисунок 2.3 — Блок-схема алгоритму знаходження кількості елементів у заданих межах на відрізку

Код реалізації запиту знаходження кількості елементів у заданих межах на відрізку на C++:

```
int getElementsBetweenAtRange(size_t leftPosition, size_t rightPosition, T leftBound, T rightBound) {
        size_t leftPositionBlock = getBlockPosition(leftPosition);
        size_t rightPositionBlock = getBlockPosition(rightPosition);
        size_t leftBlockPosition = getPositionInTheBlock(leftPosition);
        size_t rightBlockPosition = getPositionInTheBlock(rightPosition);
        if (leftPositionBlock == rightPositionBlock) {
        return blocks[leftPositionBlock]->
    getElementsBetweenAtRange(leftBlockPosition, rightBlockPosition,
    leftBound, rightBound);
        } else {
                int total = 0;
                for (size_t i = leftPositionBlock + 1; i < rightPositionBlock;
        i++) {
                        total +=
                blocks[i]->getNumberOfValuesBetween(leftBound,
                rightBound);
                }
                total += blocks[leftPositionBlock]->
                getElementsBetweenAtRange(leftBlockPosition,
                blocks[leftPositionBlock]->size() - 1, leftBound, rightBound);
                total +=
                blocks[rightPositionBlock]->getElementsBetweenAtRange(0,
                rightBlockPosition, leftBound, rightBound);
                return total;
        }
    }
```

### 2.5. Реалізація запиту розвороту підвідрізка масиву

Для того, щоб правильно розвернути підвідрізок масиву, нам потрібно перебудувати структуру декомпозиції так, щоб існувала множина блоків, яка покриває цей відрізок і не покриває при цьому жодного іншого елемента, що не належить цьому відрізку. Для цього ми визначаємо блоки, в яких знаходяться наші елементи, після чого розділяємо кожен із цих блоків на 2 частини так, щоб задовольнити вищенаведену умову. Після цього ми можемо змінити порядок блоків, і для кожного блока змінити прапорець, який позначає порядок ітерації цими елементами. Окремо варто розглянути випадок, коли обидва кінці відрізка знаходяться в одному блоці: тоді доведеться наївно розвернути елементи всередині блоку.

Для того, щоб кількість блоків не росла неконтрольовано, ми ввели параметр *capacity*, що позначає, яка максимальна кількість блоків може бути утворена. Якщо кількість блоків на якомусь етапі більша ніж дозволена, то вся декомпозиція буде побудована заново.

Блок-схема алгоритму розвороту підвідрізка масиву наведена на рисунку 2.4.

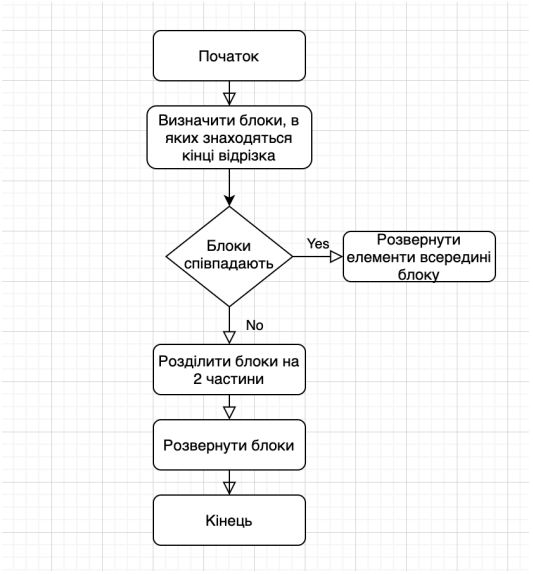


Рисунок 2.4 — Блок-схема алгоритму розвороту підвідрізка масиву

Код реалізації запиту розвороту підвідрізка масиву на C++

```
void reverseRange(size_t left, size_t right) {
    size_t leftPositionBlock = getBlockPosition(left);
    size_t rightPositionBlock = getBlockPosition(right);

    size_t leftBlockPosition = getPositionInTheBlock(left);
    size_t rightBlockPosition = getPositionInTheBlock(right);
    if (leftPositionBlock == rightPositionBlock){
        blocks[leftPositionBlock]->reverseRange(leftBlockPosition,
    rightBlockPosition);
    } else {
        replaceAndSplit(rightPositionBlock, rightBlockPosition);
        replaceAndSplit(leftPositionBlock, leftBlockPosition);
        size_t leftBlockRange = leftBlockPosition;
        size_t rightBlockRange = rightBlockPosition + 2;
        for (size_t i = leftBlockRange; i <= rightBlockRange; i++){
            blocks[i]->reverseBlock();
        }
        reverse(blocks.begin() + leftBlockRange, blocks.begin() +
        rightBlockRange + 1);
        if (blocks.size() > capacity) {
            rebuild();
        }
    }
}
```

У додатках 1 і 2 наведено повний код реалізації декомпозиції для поставленої задачі.

# РОЗДІЛ 3. ПОЄДНАННЯ ДЕКОМПОЗИЦІЇ МАСИВІВ ТА КОРЕНЕВИХ ОПТИМІЗАЦІЙ ДЛЯ ЕФЕКТИВНОЇ ОБРОБКИ ЗАПИТІВ НА ІНТЕРВАЛАХ

## 3.1. Принцип підбору оптимальних розмірів блоків для ефективної роботи декомпозиції

Враховуючи складності, які були описані в пункті 2.1 розділу 2, можемо зробити висновок, що середня асимптотика запиту буде становити $O((L + \frac{N}{L}) \cdot log(L))$, де $N$ - розмір масиву, $L$ - розмір блоку. Отже, щоб зробити складність мінімальною, нам потрібно мінімізувати функцію $f(L) = (L + \frac{N}{L}) \cdot log(L)$.

Оскільки функція логарифму зростає достатньо повільно, можемо нею знехтувати при підборі оптимального значення. Тоді потрібно мінімізувати значення $L + \frac{N}{L}$. З рівняння видно, що зі збільшенням одного із доданків інший зменшується і навпаки. З цього випливає, що $L \approx \sqrt{N}$.

Саме з виведеної вище властивості і випливає те, що декомпозиція, яка розв’яже задачу оптимально, є кореневою. Проте, як показує практика, визначення розміру блоку є окремою, не менш цікавою задачею. Це пов’язано з тим, що різні операції, які ми звикли вважати умовно атомарними і надавати їм складність $O(1)$, мають складнішу структуру і відповідно різний час виконання. Тому інколи неоптимальний, на перший погляд, блок, може давати набагато кращий результат, ніж поділ на блоки стандартного розміру. Кожна задача в такому випадку потребує окремого дослідження на предмет того, який блок є оптимальним в даному випадку.

Щоб не сповільнювати роботу програми алгоритмами підбору блоку і зробити поведінку декомпозиції більш передбачуваною, рекомендується

зробити підбір оптимальної довжини самостійно, і зробити розмір блока фіксованим.

### 3.2. Оцінка ресурсів часу та пам’яті, використовуваних кореневною декомпозицією

Всі дані, які ми зберігаємо всередині декомпозиції, поділені на блоки, сумарний розмір яких дорівнює $n$. В кожному блоці зберігаються власне дані та їх відсортована копія. Наша реалізація може працювати з будь-яким типом даних, але для тесту використаємо стандартний тип int в C++, який зазвичай займає 4 байти пам’яті. Кількість блоків, які ми зберігаємо, приблизно рівна $\sqrt{N}$, для швидкого розвороту ми використовуємо вказівники на об’єкти, розмір яких складає 4 або 8 байт(залежно від розрядності системи). Складність пам’яті може бути оцінена як $O((N + \sqrt{N})) = O(N)$.

При побудові дерева ми запустимо в кожному блоці сортування, отже сумарна складність складе $O(\sqrt{N} \cdot \sqrt{N} \cdot log(\sqrt{N})) = O(N \cdot log(\sqrt{N})) = = O(N \cdot log(N))$.

При обробці запиту першого типу ми оновимо відсортований масив і початковий масив, кожна із операцій має лінійну складність відносно довжини масиву, отже, наш алгоритм матиме складність $O(\sqrt{N})$.

При відповіді на запити другого типу ми будемо обходити блоки і в них запускати бінарний пошук, що матиме складність $O(\sqrt{N} \cdot log(\sqrt{N})) = O(\sqrt{N} \cdot log(N))$ на запит.

При обробці запиту третього типу ми будемо ітеруватися по масиву блоків і поділимо два блоки, що також матиме складність $O(\sqrt{N} \cdot log(N))$ на запит. Також, приблизно кожні $O(\sqrt{N})$ запитів буде відбуватися перебудова структури декомпозиції, що матиме сумарну складність $O(N \cdot \sqrt{N} \cdot log(N))$.

Асимптотичну складність алгоритму в цілому можна оцінити як $O(N \cdot \sqrt{N} \cdot log(N))$. Мова програмування C++ виконує за секунду приблизно $2 \cdot 10^8$ арифметичних операцій, що означає, що час виконання нашої програми складатиме приблизно декілька секунд. залежно від підібраних параметрів декомпозиції.

За допомогою випадково згенерованих тестів було перевірено точний час роботи програми для різних конфігурацій параметрів декомпозиції. Результати подано в таблиці 1. В цій таблиці $b$ - це розмір блока, а $c$ - максимальна дозволена кількість блоків. Якщо час не вказаний, це означає, що для даної конфігурація параметрів неможлива коректна робота декомпозиції.

### 3.3. Узагальнення декомпозиції для різних типів задач

У нашому конкретному випадку ми розглядали декомпозицію для операції кількості елементів, що знаходяться в певних числових межах, але ця структура даних підходить і для більш загальних задач. Декомпозиція може використовуватися для зберігання довільних типів даних і результатів операцій над ними, для яких виконується умова асоціативності: *x*(y*z) = (x*y)*z.* Зокрема, найпоширенішими з таких операцій є:

- сума значень на відрізку(зокрема сума за модулем, xor-сума);
- знаходження кількості значень на відрізку, що задовольняють певний критерій(зводиться до знаходження суми значень на відрізку);
- множення(цілих чисел за модулем, кватерніонів, матриць);
- знаходження найбільшого/найменшого значення в масиві.

Для можливості обчислення результату важливим є існування алгоритму швидкого об'єднання результатів обраної операції над частинами масиву. Не для всіх асоціативних операцій існує такий алгоритм, зокрема, для

операції сортування об’єднання двох відсортованих послідовностей даних матиме складність *O(l)*, де $l$ — це сумарна довжина двох послідовностей.

Таблиця 1. Результати тестування на різних параметрах

| | b=50 | b=100 | b=150 | b=200 |
|---|---|---|---|---|
| c=500 | - | - | 3519 мс | 2759 мс |
| c=750 | - | 3123 мс | 2481 мс | 2246 мс |
| c=1000 | - | 2633 мс | 2380 мс | 2243 мс |
| c=1250 | 4224 мс | 2666 мс | 2428 мс | 2386 мс |
| c=1500 | 3680 мс | 2757 мс | 2646 мс | 2550 мс |

Тому складність об’єднання даних при знаходженні відповіді на запит навіть на одній позиції дорівнюватиме сумарній довжині відрізків, що об’єднуються. В найгіршому випадку складність однієї операції буде становити $O(n)$. З цього можна зробити висновок, що декомпозиція даних найкраще підходить для вирішення поставленої задачі, якщо результат запиту на знаходження інформації є якоюсь статистичною функцією від відрізка масиву (наприклад, сума, кількість, середнє, найбільше чи найменше значення).

# РОЗДІЛ 4. ПРИКЛАДИ ЗАСТОСУВАННЯ КОРЕНЕВОЇ ДЕКОМПОЗИЦІЇ НА ПРАКТИЦІ

Ідея розбиття даних на блоки не є чимось новим для програмування. Такий підхід виправданий, якщо потрібно знайти баланс між тим, щоб обробити дані з певної множини по одному, чи взяти їх в обробку всі за один раз. Як правило, такі проблеми відбуваються, якщо потрібно щось завантажити зі стороннього сервера і для цього потрібно приєднатися до нього: завантаження даних по одному - це забагато приєднань-від'єднань, які впливають на час, завантаження всього разом - завеликий об'єм, що також негативно впливає на час. Тому в даному випадку оптимальним виходом буде обрати оптимальний розмір блока даних, що будуть завантажуватися, виходячи з природи даних, що завантажуються.

Яскрава ілюстрація цієї оптимізації - це стрічка новин будь-якої соцмережі: Facebook, Instagram, LinkedIn тощо. Особливо помітним є даний ефект, якщо користуватися даними соцмережами за допомогою пристрою із низькою продуктивністю, оскільки там можна буде помітити простій при завантаженні чергового блока, після чого користувачу стане доступна певна кількість новин, завантажена разом із новим блоком.

Для подібного роду оптимізацій є відомий шаблон користувацького інтерфейсу Pagination(Нумерування сторінок).[6] Його суть полягає в тому, щоб розбивати велику кількість даних на блоки, зручні як для завантаження, так і для сприйняття користувачем. Цим прийомом користуються більшість сайтів-каталогів, як от інтернет-магазини, онлайн-бібліотеки та інші. Як правило, тут користувачу можуть надати можливість обрати самому розмір однієї сторінки(тут під розміром мається на увазі кількість елементів, що відображаються на одній сторінці або розмір блока в термінології декомпозиції.

Подібний до декомпозиції прийом застосовується і в мові програмування C++ при роботі з потоками введення/виведення. Там ця особливість потоків називається буферизацією.[7] Суть її полягає в тому, щоб не передавати символи в потік виведення по одному, чи маленькими частинами, а накопичувати їх, поки не назбирається якась критична кількість символів, яку і буде виведено. Це робиться для того, щоб не оновлювати потік після кожної операції виведення в потік, адже оновлення є доволі трудомісткою операцією. Якщо є потреба підтримувати дані в потоці актуальними в режимі реального часу, оновити потік можна самостійно, для цього є спеціальний метод *flush()* або спеціальні символи, виведення яких призведе до автоматичного оновлення потоку.

# РОЗДІЛ 5. НЕСТАНДАРТНІ СПОСОБИ ЗАСТОСУВАННЯ КОРЕНЕВОЇ ОПТИМІЗАЦІЇ ПРИ РОЗВ'ЯЗУВАННІ ОЛІМПІАДНИХ ЗАДАЧ

## 5.1 Алгоритм Мо

Розглянемо наступну алгоритмічну задачу, запропоновану на платформі з проведення онлайн-змагань з програмування Codeforces.[8]

**Потужний масив**

Задано масив натуральних чисел $a_1, a_2, ..., a_n$. Розглянемо деякий його підмасив $a_l, a_{l+1}, ..., a_r$, де $1 \le l \le r \le$ n, і для кожного натурального числа s позначимо через $K_s$ число входжень числа $s$ в цей підмасив. Назвемо потужністю підмасиву суму добутків $K_s \cdot K_s \cdot s$ по всіх різних натуральних $s$. Так як кількість різних чисел в масиві скінченна, сума містить лише скінченне число ненульових доданків.

Необхідно обчислити потужності кожного з $t$ заданих підмасивів.

*Вхідні дані*

Перший рядок містить два цілих числа $n$ і $t$ ($1 \le n, t \le 200000$) - довжина масиву і кількість запитів відповідно. Другий рядок містить n натуральних чисел $a_i$ ($1 \le a_i \le 10^6$) - елементи масиву. Наступні $t$ рядків містять по два натуральних числа $l$ і $r$ ($1 \le l \le r \le n$) - індекси лівого і правого кінців відповідного підмасиву.

*Вихідні дані*

Виведіть $t$ рядків, де $i$-й рядок містить єдине натуральне число - потужність підмасиву $i$-го запиту.

Приклади вхідних та вихідних даних наведені в таблиці 2.

*Розв'язок*

Очевидно, що ідею повного перебору варто відкинути одразу, адже її швидкодія не підходить нам при заданих обмеженнях на розмір масиву і

кількість запитів. Відомі евристики із частковими результатами на префіксах чи іншого роду відрізках не є застосовними до даної задачі, адже через специфіку значення, яке потрібно обчислити на відрізку, процес об'єднання результатів, обчислених на суміжних відрізках є занадто трудомістким для практичного використання.

Діятимемо таким чином: обрахуємо відповідь на перший запит наївним методом. Для цього для кожного числового значення потрібно дізнатися кількість його входжень в заданий відрізок. Після того, як ми знайшли відповідь на цей запит, ми перейдемо до наступного запиту так: переміщуватимемо ліву і праву межу відрізка попереднього запиту, поки вони не співпадуть з межами для другого запиту, підтримуючи при цьому актуальну кількість входжень кожного із значень в відрізок. Таким чином, ми побудували коректний алгоритм, але складність якого в найгіршому випадку складе $O(t \cdot n \cdot log(n))$. Щоб пришвидшити алгоритм, скористаємося кореневою евристикою.

Таблиця 2. Вхідні та Вихідні дані для задачі "Потужний масив"

| **Вхідні дані** | **Вихідні дані** |
|---|---|
| 3 2<br>1 2 1<br>1 2<br>1 3 | 3<br>6 |
| 8 3<br>1 1 2 2 1 3 1 1<br>2 7<br>1 6<br>2 7 | 20<br>20<br>20 |

В умові задачі не сказано, що ми повинні обробляти запити саме в такому порядку, в якому вони нам подані у вхідних даних. Це означає, що дані можна впорядкувати таким чином, щоб сумарна кількість пересувань меж відрізків зменшилася. Для цього ми згрупуємо запити у кореневі блоки по лівій межі відрізка, і всередині кожного блоку відсортуємо запити за неспаданням правої межі. Тепер для запитів, що знаходяться всередині одного блоку, буде витрачено не більше ніж $\sqrt{n}$ операцій при переміщенні лівої межі для одного запиту і не більше ніж *O(n)* операцій переміщення правої межі сумарно для всього блоку(адже значення в ньому відсортовані за неспаданням). При переході із блоку в блок буде витрачено в найгіршому випадку $O(n)$ операцій. Таким чином, підсумкова складність отриманого розв'язку складає $O((t + n)\sqrt{n}\, log(n))$.

Даний підхід в алгоритмічному програмуванні прийнято називати алгоритмом Мо, але хто саме його винайшов і коли - науці невідомо. Загальноприйнята назва була вперше зафіксована серед китайських учасників олімпіад, і від них поширилася і серед інших ентузіастів.[8]

### 5.2 Розбиття на "легкі" та "важкі" частини

Часто використовувати евристики, які не користуються типовим для кореневих оптимізації "розбиттям на блоки", але мають такий самий ефект оптимізації, як і інші оптимізації цього виду. Однією із таких евристик є поділ на "легкі" та "важкі" частини певної множини однотипних об'єктів із метою розділення їх подальшої обробки.

Нехай задана множина *D*, на самому початку вона порожня. В нас є 3 види запитів, пов'язаних з цією множиною, а саме:

- додати рядок *s* в множину *D*;
- видалити рядок *s* із множини *D*;

- обчислити сумарну кількість входжень рядка *t* у кожен із рядків із множини *D* в якості підрядка.

Важливою особливість заданих запитів є те, що відповідати на них потрібно онлайн, тобто новий запит не буде отримано, поки не буде надано відповідь на попередній запит. Всього є *m* таких запитів. Також, сумарна довжина рядків, як і кількість запитів, не перевищує 300000.

Особливістю підходу поділу об'єктів на "легкі" та "важкі" частини є те, що ми можемо поділити об'єкти за певною ознакою таким чином, щоб можна було розділити способи опрацювання "легкої" та "важкої" множин. Давайте визначимо *l* як сумарну довжину всіх уведених рядків. Тоді за ознаку, за якою відбувається поділ на дві неперетинні множини, візьмемо таку умову: $|s| \leq \sqrt{l}$. Якщо умова виконується, то, відповідно, об'єкт будемо вважати "легким", а рядки назвемо *короткими*, в протилежному випадку об'єкт будемо вважати "важким", а рядки назвемо *довгими*. Особливість множини коротких рядків виписана в якості умови поділу, що ми і використаємо в майбутньому, а для довгих рядків можемо підмітити, що оцінка їх кількості буде складати $O(\sqrt{l})$. Обробка перших двох запитів - річ доволі проста, оскільки контейнери для формування множин давно стали частиною стандартних бібліотек більшості мов програмувань. Щодо третього запиту - визначимо такі підходи для обробки рядків:

- якщо рядок, який приходить на обробку - довгий, то ми можемо собі дозволити пройтися по множині довгих рядків і знайти кількість входжень за лінійний час алгоритмом Кнута-Моріса-Пратта, за допомогою Z-функції чи однієї із численних модифікацій алгоритма Боєра-Мура;[5]
- якщо ж рядок, який приходить на обробку - короткий, то потрібно його захешувати і порахувати кількість підрядків, в яких хеш співпадає. Оскільки таких підрдяків буде $O(|s|\sqrt{l})$, то можна при додаванні рядка пройтися по всіх можливих підрядках довжини меншої ніж $\sqrt{l}$, і

порахувати кількості потрібних нам хешів із занесенням у хеш-таблицю. Для уникнення колізій рекомендується розділити хеш-таблиці відповідно до довжини рядків, які в них зберігаються

Таким чином, розділивши дані, що обробляються, на блоки, ми змогли досягти сумарної складності $O((m + l)\sqrt{l})$ .

### 5.3 Коренева декомпозиція по запитах або метод часткової актуалізації

Давайте підійдемо до базової задачі кореневої декомпозиції іншим чином: будемо розбивати не масив на блоки, а запити. Метою цього розбиття є те, що реальне оновлення даних буде відбуватися лише на початку нового блоку. Оскільки кількість таких блоків можна оцінити як $O(\sqrt{m})$, то це означає, що ми можемо собі дозволити повністю перебудовувати всю структуру, яка дозволяє відповідати нам на запити. Якою ця буде структура, в даному випадку не так важливо: це можуть бути масиви префіксних результатів, дерево відрізків, розріджена таблиця чи інша структура даних, яка допомагає відповідати на задані запити.

Тепер розглянемо, яким чином ми можемо відповідати на запити. Однією з проблем є те, що дані, які ми зберігаємо, не є актуальними. Обійти цю проблему можна легко, адже ми знаємо, що для того, щоб зробити цю структуру актуальною, потрібно не більше ніж $\sqrt{m}$ модифікацій. Знаючи інтервал, ми можемо легко відкинути запити оновлення, які на нього жодним чином не впливають, а знаючи природу запитів, ми можемо модифікувати не саму структуру, яка допомагає відповідати на запити, а результат, який ми отримали. Це може дозволити відповідати на запити, маючи не до кінця актуальні дані, доповнюючи їх даними про модифікації, які повинні бути внесені, але не вносяться через високу часову складність їх внесення. Тому ми частково актуалізуємо дані, і доповнюємо їх під час відповіді на запит.

Період оновлення даних в даному прикладі обраний $\sqrt{m}$, щоб спростити розуміння того, якою має бути принцип підбору цієї величини, але на практиці потрібно враховувати час, що витрачається на обробку кожного із етапів програми, і з урахуванням всіх величини проводити підбір цієї константи. Якщо ці дії є занадто складними в силу непрогнозованості алгоритмів, можна спробувати делегувати підбір величини одному з оптимізаційних алгоритмів, таким як градієнтний спуск, метод Нелдера-Міда, метод імітації відпалу тощо.[9]

# ВИСНОВКИ

В даній роботі нами розглянуто різновиди кореневих оптимізацій, в тому числі кореневої декомпозиції, описано алгоритми її роботи, вирахувано асимптотичну складність роботи, досліджено різноманітні можливості структури, вказані переваги та недоліки.

Коренева декомпозиція – достатньо гнучка структура, і число завдань, що вирішуються нею, теоретично необмежене і набагато більше, ніж у інших стандартних структур даних. Проте якщо для більшості операцій стандартних структур даних достатньо, то кореневі оптимізації поводять себе набагато краще з операціями, для яких збереження даних у дереві відрізків є досить трудомісткою операцією.

Важливою особливістю кореневої декомпозиції є те, що вона споживають лінійний обсяг пам'яті: для масиву з $n$ елементів у випадку задачі, яку ми розглядали, сумарний розмір масивів складав $2n$; в деяких випадках обсяги додаткової пам’яті можуть бути меншими за лінійну.

Кореневі оптимізації дозволяють помітно пришвидшити операції та запити на відрізку масиву. Вони є чудовою оптимізацією інших алгоритмів. Асимптотична складність побудови декомпозиції в нашому випадку складає $O(n \cdot log(n))$, а відповіді на запити здійснюються за $O(\sqrt{n} \cdot log(n))$. Вона є зручною у використанні та нескладною у реалізації. За допомогою простоти ідеї розбиття масиву на блоки її можна всебічно вдосконалювати та підлаштовувати для розв’язання поставленої задачі.

Ця структура поступається по швидкодії іншим структурам даних, але має певні переваги порівняно з іншими(наприклад, займає менше пам’яті, ніж дерево відрізків), і клас задач, до якого можуть бути застосовані кореневі оптимізації, є значно ширшим та досі вивчається.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Акимов О. Е. Дискретная математика: логика, группы, графы / О. Е. Акимов. – М.: Лаборатория базовых знаний, 2001. – 376 с.
2. Хорошев А. Н. Введение в управление проектированием механических систем: Учебное пособие / А. Н. Хорошев. – Белгород, 1999. – 372 с.
3. ОСНОВИ ТВОРЕННЯ МАШИН / М. Я.Бучинський, О. В. Горик, А. М. Чернявський, С. В. Яхін. – Харків: НМТМ, 2017. – 448 с.
4. CP Algorithms [Електронний ресурс] – Режим доступу до ресурсу: https://cp-algorithms.com.
5. Вступ до алгоритмів / Т. Г.Кормен, Ч. Е. Лейзерсон, Р. Л. Рівест, К. Стайн. – Київ: К.І.С., 2018. – (3).
6. UI Patterns [Електронний ресурс] – Режим доступу до ресурсу: http://ui-patterns.com.
7. cplusplus.com [Електронний ресурс] – Режим доступу до ресурсу: https://www.cplusplus.com/.
8. Codeforces [Електронний ресурс] – Режим доступу до ресурсу: https://codeforces.com.
9. Левитин А. В. Алгоритмы: введение в разработку и анализ / А. В. Левитин. – М., 2006. – 160 с.

## ДОДАТОК А Код реалізації блоку декомпозиції

```
#ifndef UNTITLED_BLOCK_H
#define UNTITLED_BLOCK_H

#include <vector>
#include <algorithm>
using namespace std;

template<typename T> class Block {
public:
   Block() = default;
   explicit Block(const vector<T>& data) : data(data), sorted(data), reversed(false)
{
      sort(sorted.begin(), sorted.end());
   }

   int getNumberOfValuesBetween(T left, T right) {
      auto it1 = lower_bound(sorted.begin(), sorted.end(), left);
      auto it2 = upper_bound(sorted.begin(), sorted.end(), right);
      return it1 >= it2 ? 0 : it2 - it1;
   }
   int getElementsBetweenAtRange(size_t leftPosition, size_t rightPosition, T
leftBound, T rightBound) {

      leftPosition = getCorrectPosition(leftPosition);
      rightPosition = getCorrectPosition(rightPosition);

      if (leftPosition > rightPosition) {
         swap(leftPosition, rightPosition);
      }


      int total = 0;
      for (size_t i = leftPosition; i <= rightPosition; i++) {
         if (leftBound <= data[i] && data[i] <= rightBound) {
            total++;
         }
      }

      return total;
   }
```

```
void updateElement(size_t position, T value) {
    position = getCorrectPosition(position);
    for (auto &item: sorted) {
        if (item == data[position]) {
            item = value;
            break;
        }
    }
    sort(sorted.begin(), sorted.end());
    data[position] = value;
}

T get(size_t position) const {
    return data[getCorrectPosition(position)];
}

vector<T> getData() const {
    return data;
}

void reverseBlock() {
    reversed = !reversed;
}

void reverseRange(size_t left, size_t right) {
    left = getCorrectPosition(left);
    right = getCorrectPosition(right);
    if (left > right) {
        swap(left, right);
    }

    reverse(data.begin() + left, data.begin() + right + 1);
}

pair<Block<T>*, Block<T>*> split(size_t newBlockIndex) {
    if (reversed) {
        reverseBlock();
        reverse(data.begin(), data.end());
    }

    vector<T> dataLeft(data.begin(), data.begin() + newBlockIndex);
    vector<T> dataRight(data.begin() + newBlockIndex, data.end());

    return make_pair(new Block(dataLeft), new Block(dataRight));
```

```
    }

    size_t size() const {
        return data.size();
    }

    void printData() {
        if (reversed) {
            for (auto it = data.rbegin(); it != data.rend(); it++) {
                cout << *it << " ";
            }
        } else {
            for (auto item: data) {
                cout << item << " ";
            }
        }
    }
private:
    bool reversed;
    vector<T> data;
    vector<T> sorted;

    size_t getCorrectPosition(size_t givenPosition) {
        if (reversed) {
            return data.size() - 1 - givenPosition;
        } else {
            return givenPosition;
        }
    }
};


#endif //UNTITLED_BLOCK_H
```

## ДОДАТОК Б Код реалізації декомпозиції

```
#ifndef UNTITLED_DECOMPOSITION_H
#define UNTITLED_DECOMPOSITION_H

#include "block.h"
#include <vector>
#include <algorithm>

using namespace std;

template <typename T> class Decomposition {
public:
    Decomposition(vector<T> data, size_t defaultBlockSize, size_t capacity)
        : defaultBlockSize(defaultBlockSize), capacity(capacity) {
        decompose(data);
    }

    void updateElement(size_t position, T value) {
        for (Block<T>* block: blocks) {
            if (block->size() > position) {
                block->updateElement(position, value);
                return;
            } else {
                position -= block->size();
            }
        }
    }
```

```
    int getElementsBetweenAtRange(size_t leftPosition, size_t rightPosition, T
leftBound, T rightBound) {
        size_t leftPositionBlock = getBlockPosition(leftPosition);
        size_t rightPositionBlock = getBlockPosition(rightPosition);

        size_t leftBlockPosition = getPositionInTheBlock(leftPosition);
        size_t rightBlockPosition = getPositionInTheBlock(rightPosition);

        if (leftPositionBlock == rightPositionBlock) {
            return blocks[leftPositionBlock]->getElementsBetweenAtRange(
                    leftBlockPosition, rightBlockPosition, leftBound, rightBound);
        } else {
            int total = 0;
            for (size_t i = leftPositionBlock + 1; i < rightPositionBlock; i++) {
                total += blocks[i]->getNumberOfValuesBetween(leftBound,
rightBound);
            }
            total += blocks[leftPositionBlock]->getElementsBetweenAtRange(
                    leftBlockPosition, blocks[leftPositionBlock]->size() - 1, leftBound,
rightBound);

            total += blocks[rightPositionBlock]->getElementsBetweenAtRange(
                    0, rightBlockPosition, leftBound, rightBound);

            return total;
        }
    }
```

```
    void reverseRange(size_t left, size_t right) {
        size_t leftPositionBlock = getBlockPosition(left);
        size_t rightPositionBlock = getBlockPosition(right);

        size_t leftBlockPosition = getPositionInTheBlock(left);
        size_t rightBlockPosition = getPositionInTheBlock(right);

        if (leftPositionBlock == rightPositionBlock) {
            blocks[leftPositionBlock]->reverseRange(leftBlockPosition,
rightBlockPosition);
        } else {
            replaceAndSplit(rightPositionBlock, rightBlockPosition);
            replaceAndSplit(leftPositionBlock, leftBlockPosition);

            size_t leftBlockRange = leftBlockPosition;
            size_t rightBlockRange = rightBlockPosition + 2;
            for (size_t i = leftBlockRange; i <= rightBlockRange; i++) {
                blocks[i]->reverseBlock();
            }
            reverse(blocks.begin() + leftBlockRange, blocks.begin() +
rightBlockRange + 1);

            if (blocks.size() > capacity) {
                rebuild();
            }
        }
    }
```

```
    void printData() {
        cout << "========" << endl;
        for (auto block: blocks) {
            cout << "Block " << block << ":\n";
            block->printData();
            cout << "\n\n";
        }
        cout << endl;
    }

private:
    vector<Block<T>*> blocks;
    size_t defaultBlockSize;
    size_t capacity;

    void decompose(vector<T> data) {
        vector<vector<T>> blocksData((data.size() + defaultBlockSize - 1) /
defaultBlockSize);
        for (size_t i = 0; i < data.size(); i++) {
            blocksData[i / defaultBlockSize].push_back(data[i]);
        }

        for (const auto& blockData: blocksData) {
            blocks.push_back(new Block(blockData));
        }
    }
```

```
void rebuild() {
    vector<T> data;
    for (auto block: blocks) {
        vector<T> blockData = block->getData();
        for (auto item : blockData) {
            data.push_back(item);
        }
    }

    for (auto block: blocks) {
        delete block;
    }
    blocks.clear();
    decompose(data);
}

size_t getBlockPosition(size_t position) const {
    for (size_t i = 0; i < blocks.size(); i++) {
        if (blocks[i]->size() <= position) {
            position -= blocks[i]->size();
        } else {
            return i;
        }
    }
    return -1;
}

size_t getPositionInTheBlock(size_t position) const {
```

```
        for (size_t i = 0; i < blocks.size(); i++) {
            if (blocks[i]->size() <= position) {
                position -= blocks[i]->size();
            } else {
                return position;
            }
        }
        return -1;
    }

    pair<size_t, size_t> getBlockRanges(size_t blockPosition) const {
        size_t leftPosition = 0;
        for (size_t i = 0; i < blockPosition; i++) {
            leftPosition += blocks[i]->size();
        }
        return {leftPosition, leftPosition + blocks[blockPosition]->size() - 1};
    }

    void replaceAndSplit(size_t blockPosition, size_t splitPosition) {
        auto newBlocks = blocks[blockPosition]->split(splitPosition);
        delete (blocks[blockPosition]);
        blocks[blockPosition] = newBlocks.first;
        blocks.insert(blocks.begin() + blockPosition + 1, newBlocks.second);
    }
};

#endif //UNTITLED_DECOMPOSITION_H
```