Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра ІПІ

Звіт

з лабораторної роботи №2 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

„ **Проектування і аналіз алгоритмів внутрішнього сортування**”

Виконав ІП-45 Янов Богдан Євгенійович

Перевірив Соколовський Владислав Володимирович

Київ 2025

Лабораторна робота №2

Проектування і аналіз алгоритмів внутрішнього сортування

Мета: дослідити метод декомпозиції, обчислити складність алгоритму і оцінити поріг його ефективності.

За допомогою методу декомпозиції розробити алгоритм, який буде розв’язувати наступну, задачу. Вхідні дані. Матриця D натуральних чисел розмірності u*m, де u — це кількість користувачів, m — кількість фільмів. Кожний елемент матриці D[i, i] вказує на позицію фільму і в списку вподобань користувача і. Іншим вхідним елементом є x — номер користувача, з яким будуть порівнюватись всі інші користувачі. Вихідні дані. Список з впорядкованих за зростанням другого елементу пар (i, c), де і — номер користувача, c — число, яке вказує на степінь схожості вподобань користувачів x та c (кількість інверсій).

Критерій	**Merge Sort**
Стійкість	**Стійкий** - однакові елементи не змінюють відносний порядок
Природність	**Не природний** - стандартна реалізація враховує початковий порядок
Базування на порівняннях	**Так** - заснований на порівнянні елементів
Необхідність у додатковій пам'яті	**Так** - потребує додаткової пам'яті n для злиття підмасивів.
Потреба в знаннях про структури даних	**Середня** - базова реалізація не вимагає складних структур, але потрібно розуміння рекурсії та розділення масивів.

Псевдокод алгоритмів

```
merge_sort(arr, left, right):
     if left >= right return 0
     mid = (left + right) / 2
     inv_count = merge_sort(arr, left, mid)
     inv_count += merge_sort(arr, mid + 1, right)
     inv_count += merge(arr, left, mid, right)
     return inv_count
```

```
merge(arr, left, mid, right)
     left_arr = arr[left: mid+1]
     right_arr = arr[mid+1: right+1]
     i = 0
     j = 0
     k = left
     inv_count = 0
     while i < len(left_arr) and j < len(right_arr):
          if left_arr[i] <= right_arr[j]
               arr[k] = left_arr[i]
               i += 1
          else
               arr[k] = right_arr[j]
               inv_count += len(left_arr) - i
               j += 1
          k += 1
          while i < len(left_arr):
               arr[k] = left_arr[i]
               i += 1
               k += 1
          while j < len(right_arr):
               arr[k] = right_arr[j]
               j += 1
               k += 1

     return inv_count
```

Рекурентне рівняння Merge Sort

T(n) - часова складність сортування злиттям для масиву з n елементів.
Алгоритм виконує 3 основні дії:

1. Ділення масиву на дві половини - O(1)
2. Рекурсивне сортування обох половин - 2T(n/2)
3. Злиття двох відсортованих підмасивів - O(n)

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2T\left(\frac{n}{2}\right) + O(n), & n > 1 \end{cases}$$

a = 2 (дві підзадачі)

b = 2 (кожна з розміром n/2)
f(n) = O(n) (злиття)
Тоді n^log(b, a) = n^1 = n

Це 2 випадок Master Theorem: f(n) = O(n^log(b,a)) => T(n) = O(n log n)

Остаточна асимптотика:

- Найгірший випадок - O(n log n)
- Середній випадок - O(n log n)
- Найкращий випадок - O(n log n)

Пояснення

- Merge Sort не залежить від впорядкованості даних
- Кожен рівень рекурсії має лінійний обсяг роботи, а рівнів всього log n
- Загальна складність - n log n

Програмна реалізація

```
import sys

if len(sys.argv) != 3:
   print("Usage: python lab.py <user_id> <filename>")
   sys.exit(1)

selected_user = int(sys.argv[1]) - 1
filename = sys.argv[2]


def count_inversions(arr: list[int]) -> int:
   return merge_sort(arr, 0, len(arr) - 1)


def merge_sort(arr: list[int], left: int, right: int) -> int:
   if left >= right:
       return 0

   mid = (left + right) // 2
   inv_count = merge_sort(arr, left, mid)
   inv_count += merge_sort(arr, mid + 1, right)
```

```
    inv_count += merge(arr, left, mid, right)

    return inv_count


def merge(arr: list[int], left: int, mid: int, right: int) -> int:
    left_arr = arr[left : mid + 1]
    right_arr = arr[mid + 1 : right + 1]

    i = 0
    j = 0
    k = left
    inv_count = 0

    while i < len(left_arr) and j < len(right_arr):
        if left_arr[i] <= right_arr[j]:
            arr[k] = left_arr[i]
            i += 1
        else:
            arr[k] = right_arr[j]
            inv_count += len(left_arr) - i
            j += 1
        k += 1

    while i < len(left_arr):
        arr[k] = left_arr[i]
        i += 1
        k += 1

    while j < len(right_arr):
        arr[k] = right_arr[j]
        j += 1
        k += 1

    return inv_count
```

```
num_users = 0
num_items = 0

matrix: list[list[int]] = []

with open(filename, "r") as f:
   num_users, num_items = map(int, f.readline().split())

   matrix = [[] for _ in range(num_users)]

   for i in range(num_users):
       user_id, *rating = map(int, f.readline().split())
       matrix[user_id - 1] = rating

selected_ranking = matrix[selected_user]

order = sorted(range(num_items), key=lambda i: selected_ranking[i])

results = {
   i: count_inversions([matrix[i][j] for j in order])
   for i in range(num_users)
   if i != selected_user
}

for user_i, inv in sorted(results.items(), key=lambda x: x[1]):
   print(user_i + 1, inv)
```

Приклад роботи

Вхідний файл	Вихідний файл
5 5	4
1 4 2 1 3 5	5 3
2 3 1 2 4 5	1 4
3 4 5 3 2 1	2 4
4 5 1 3 4 2	3 5
5 3 2 4 5 1	

Висновок: в результаті виконання лабораторної роботи я проаналізував алгоритм сортування злиттям, який реалізує принцип роботи "Розділяй та володарюй" (метод розбиття на підзадачі). Алгоритм рекурсивно розділяє масив на дві рівні частини, сортує кожну частину окремо, а потім об’єднує їх у відсортований масив. Завдяки такій структурі його часова складність дорівняє O(n log n), що забезпечує стабільно хорошу ефективність при роботі з великими обсягами даних. Цей метод сортування вимагає додаткову пам’ять при для тимчасових масивів, і його реалізація є складнішою, ніж у простіших алгоритмів сортування. Для малих масивів витрати на розбиття та злиття можуть бути більшими за виграш у швидкості, тому у таких випадках доцільніше використовувати простіші методи. Проте для середніх і великих обсягів даних сортування злиттям є оптимальним вибором.