Міністерство освіти і науки
України Львівський національний університет імені Івана Франка
Факультет електроніки та комп'ютерних технологій

Звіт

про виконання лабораторної роботи №2

“Швидке сортування. Порядкові статистики”

Виконав:

студент 2 курсу

групи ФЕП-23с

Чепара Станіслав

Викладач:

доц. Середницька Христина Ігорівна

Львів - 2025

Мета роботи

Дослідити алгоритм швидкого сортування (Quicksort), його переваги та недоліки, а також зрозуміти важливість рандомізації у виборі опорного елемента. Здобути практичні навички з реалізації процедури Partition та рекурсивної функції Quicksort. Вивчити поняття порядкових статистик та реалізувати алгоритм RandomizedSelect для ефективного знаходження мінімального, максимального значень та медіани у масиві без його повного сортування.

Теоретичні відомості:

Зміст роботи

Частина 1. Швидке сортування.

1. У бібліотеці Sort (створеній раніше), згідно описаних в теоретичній частині алгоритмів, створити функції QuickSort(…), Partition(…) та RandomizedPartition(…) для реалізації швидкого сортування одномірного масиву даних.

2. Створити новий проект Lab_02_01 та підключити до нього бібліотеку Sort. У функції main() проекту реалізувати можливість введення одномірного масиву даних та відображення результатів сортування. Відкомпілювати проект та продемонструвати його роботу для одномірного масиву даних, отриманого від викладача.

main.cpp

```cpp
#include <cstdio>
#include <cstdlib>
#include "Sort/Sort.h"

static void print_arr(const int* a, int n) {
    for (int i = 0; i < n; ++i) {
        if (i) std::printf(" ");
        std::printf("%d", a[i]);
    }
    std::printf("\n");
}

int main() {
    int n;
    std::printf("Введіть к-сть елементів:\n");
    if (std::scanf("%d", &n) != 1 || n < 0) {
        std::fprintf(stderr, "Некоректний розмір масиву.\n");
        return 1;
    }
    std::printf("Введіть масив:\n");
    int* a = (int*)std::malloc(sizeof(int) * (n > 0 ? n : 1));
    if (!a) { std::fprintf(stderr, "Немає пам'яті.\n"); return 1; }

    for (int i = 0; i < n; ++i) std::scanf("%d", &a[i]);

    if (n > 1) Sort::QuickSort(a, 0, n - 1);
    // Якщо треба випадковий опорний:
    // if (n > 1) Sort::QuickSortRandomized(a, 0, n - 1);

    std::printf("Відсортований масив:\n");
    print_arr(a, n);

    std::free(a);
    return 0;
}
```

Sort.cpp

Sort > Sort.cpp > {} Sort

```cpp
#include "Sort.h"
#include <cstdlib>
#include <ctime>

namespace Sort {
    static bool seeded = false;
    static void ensure_seeded() {
        if (!seeded) { std::srand((unsigned)std::time(nullptr)); seeded = true; }
    }

    int Partition(int* a, int l, int r) {
        int x = a[r];              // опорний
        int i = l - 1;             // позиція для елементів ≤ x
        for (int j = l; j <= r - 1; ++j) {
            if (a[j] <= x) {
                ++i;
                iswap(a[i], a[j]);
            }
        }
        iswap(a[i + 1], a[r]);
        return i + 1;              // індекс опорного після розбиття
    }

    int RandomizedPartition(int* a, int l, int r) {
        ensure_seeded();
        int pivotIndex = l + std::rand() % (r - l + 1);
        iswap(a[pivotIndex], a[r]);
        return Partition(a, l, r);
    }

    void QuickSort(int* a, int l, int r) {
        if (l < r) {
            int q = Partition(a, l, r);
            QuickSort(a, l, q - 1);
            QuickSort(a, q + 1, r);
        }
    }

    void QuickSortRandomized(int* a, int l, int r) {
        if (l < r) {
            int q = RandomizedPartition(a, l, r);
            QuickSortRandomized(a, l, q - 1);
            QuickSortRandomized(a, q + 1, r);
        }
    }

    int RandomizedSelect(int* a, int l, int r, int i) {
        // i — 1-базований номер всередині a[l..r]
        if (i <= 0 || i > (r - l + 1)) return 0; // перевірка меж
        ensure_seeded();
        while (true) {
            if (l == r) return a[l];
            int pivotIndex = l + std::rand() % (r - l + 1);
            iswap(a[pivotIndex], a[r]);
            int q = Partition(a, l, r);
            int k = q - l + 1; // кількість елементів у лівій частині + опорний
            if (i == k) return a[q];
            else if (i < k) r = q - 1;      // шукаємо зліва
            else { l = q + 1; i -= k; }     // справа
        }
    }
}
```

Sort.h

```
#ifndef SORT_SORT_H
#define SORT_SORT_H

#include <cstddef>

namespace Sort {
    inline void iswap(int& a, int& b) {
        int t = a; a = b; b = t;
    }
    int Partition(int* a, int l, int r);
    int RandomizedPartition(int* a, int l, int r);
    void QuickSort(int* a, int l, int r);
    void QuickSortRandomized(int* a, int l, int r);
    int RandomizedSelect(int* a, int l, int r, int i);
}

#endif // SORT_SORT_H
```

Результати

```
Введіть к-сть елементів:
12
Введіть масив:
78 21 14 97 87 62 74 85 76 45 84 22
Відсортований масив:
14 21 22 45 62 74 76 78 84 85 87 97
stanislav@fedora:~/Desktop/Algoritms&Data
```

Частина 2. Порядкові статистики.

Хід роботи:

1. Створити новий проект Lab_02_02. Згідно описаного в теоретичній частині алгоритму, створити функцію RandomizedSelect(…) для пошуку порядкових статистик. У функції main() проекту реалізувати можливість введення одномірного масиву даних та відображення результатів пошуку порядкових статистик у ньому.

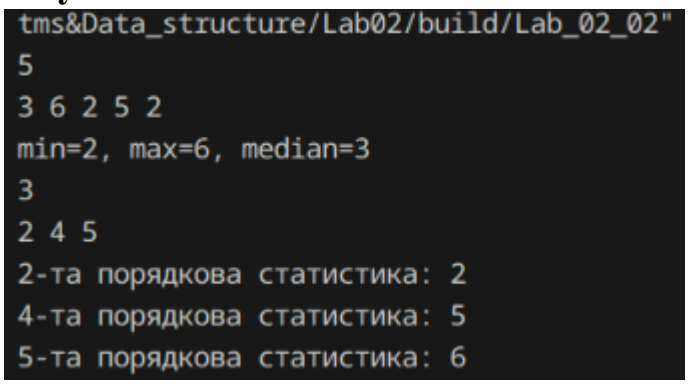
2. Підключити до проекту Lab_02_02 бібліотеку Sort з метою використання функції RandomizedPartition(…).

3. Відкомпілювати проект та продемонструвати його роботу для одномірного масиву даних, отриманого від викладача: знайти i-ту порядкову статистику (i = 1…n-1), максимальне, мінімальне значення та медіану

main.cpp

```cpp
int main() {
    int n;
    if (std::scanf("%d", &n) != 1 || n <= 0) {
        std::fprintf(stderr, "Некоректний розмір масиву.\n");
        return 1;
    }

    int* a = (int*)std::malloc(sizeof(int) * n);
    if (!a) { std::fprintf(stderr, "Немає пам'яті.\n"); return 1; }

    for (int i = 0; i < n; ++i) std::scanf("%d", &a[i]);

    // Мін/макс/медіана(и) через RandomizedSelect (без повного сортування)
    int mn = Sort::RandomizedSelect(a, 0, n - 1, 1);
    int mx = Sort::RandomizedSelect(a, 0, n - 1, n);

    if (n % 2 == 1) {
        int med = Sort::RandomizedSelect(a, 0, n - 1, (n + 1) / 2);
        std::printf("min=%d, max=%d, median=%d\n", mn, mx, med);
    } else {
        int m1 = Sort::RandomizedSelect(a, 0, n - 1, n / 2);
        int m2 = Sort::RandomizedSelect(a, 0, n - 1, n / 2 + 1);
        std::printf("min=%d, max=%d, lower_median=%d, upper_median=%d\n", mn, mx, m1, m2);
    }

    int q; // кількість запитів i-тої порядкової статистики
    if (std::scanf("%d", &q) != 1 || q < 0) {
        std::fprintf(stderr, "Некоректна кількість запитів.\n");
        std::free(a); return 1;
    }

    for (int t = 0; t < q; ++t) {
        int i;
        if (std::scanf("%d", &i) != 1) { std::fprintf(stderr, "Очікувалось i.\n"); std::free(a); return 1; }
        if (i <= 0 || i > n) {
            std::printf("i=%d поза межами [1..%d]\n", i, n);
        } else {
            int val = Sort::RandomizedSelect(a, 0, n - 1, i);
            std::printf("%d-та порядкова статистика: %d\n", i, val);
        }
    }

    std::free(a);
    return 0;
}
```

Результат



Висновок

У ході виконання лабораторної роботи було розроблено та реалізовано алгоритми швидкого сортування та пошуку порядкових статистик. Для сортування використано метод **QuickSort** із функціями `Partition` та `RandomizedPartition`, що забезпечує ефективний розподіл елементів відносно опорного. Реалізовано також алгоритм **RandomizedSelect** для знаходження мінімального, максимального, медіанного та довільного і-го елемента без повного сортування масиву..