Індивідуальна робота №4
Наденко Олексій МФ-21

Постановка задачі:
1. Оберіть і програмно реалізуйте один з методів вдосконаленого сортування з обчисленням кількості порівнянь та переміщень.
2. Програмно реалізуйте один з методів зовнішнього сортування з обчисленням кількості порівнянь та переміщень.
3. Протестуйте ці реалізацію в найкращому, найгіршому та середньому випадках для даних розміром 100, 1000, 10000, 100000, 1000000 елементів. Збережіть отримані кількості порівнянь та переміщень у підготовленому середовищі та проаналізуйте отримані графіки. Порівняйте ці методи із вже реалізованими. Зробіть свої висновки та запишіть їх. Перевірте, чи відображують отримані вами результати загальні формули трудомісткості та очікувану поведінку методу для різних початкових послідовностей.

№1

Візьмемо алгоритм **швидкого сортування (QuickSort)**, оскільки він один із найпопулярніших методів вдосконаленого сортування з оцінками трудомісткості:

Найкращий випадок: O(n*logn)
Середній випадок: O(n/logn)
Найгірший випадок: O(n^2) (наприклад, для вже відсортованих або зворотно відсортованих даних при виборі поганого опорного елемента).

Код программы:
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Глобальні змінні для підрахунку порівнянь та переміщень
long long comparisons = 0;
long long swaps = 0;

// Функція для обміну двох елементів
void swap(int *a, int *b) {
    swaps++;
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Розділення масиву для QuickSort
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Опорний елемент
    int i = low - 1;
```

```
    for (int j = low; j < high; j++) {
        comparisons++;
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// Реалізація алгоритму QuickSort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Функція для заповнення масиву випадковими числами
void fillRandom(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand();
    }
}

// Функція для заповнення масиву у найкращому випадку (вже відсортований)
void fillSorted(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;
    }
}

// Функція для заповнення масиву у найгіршому випадку (зворотний порядок)
void fillReversed(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i;
    }
}

int main() {
    // Розміри тестових масивів
    int sizes[] = {100, 1000, 10000, 100000, 1000000};
    int numTests = 5;

    // Результати
    for (int t = 0; t < numTests; t++) {
```

```
        int size = sizes[t];
        int *arr = (int *)malloc(size * sizeof(int));

        // Тест для випадкових даних
        fillRandom(arr, size);
        comparisons = 0;
        swaps = 0;
        quickSort(arr, 0, size - 1);
        printf("Random - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons,
swaps);

        // Тест для найкращого випадку
        fillSorted(arr, size);
        comparisons = 0;
        swaps = 0;
        quickSort(arr, 0, size - 1);
        printf("Best - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons, swaps);

        // Тест для найгіршого випадку
        fillReversed(arr, size);
        comparisons = 0;
        swaps = 0;
        quickSort(arr, 0, size - 1);
        printf("Worst - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons,
swaps);

        free(arr);
    }
    return 0;
}
```

```
Random - Size: 100, Comparisons: 633, Swaps: 452
Best - Size: 100, Comparisons: 4950, Swaps: 5049
Worst - Size: 100, Comparisons: 4950, Swaps: 2549
Random - Size: 1000, Comparisons: 10870, Swaps: 6155
Best - Size: 1000, Comparisons: 499500, Swaps: 500499
Worst - Size: 1000, Comparisons: 499500, Swaps: 250499
Random - Size: 10000, Comparisons: 158165, Swaps: 78569
Best - Size: 10000, Comparisons: 49995000, Swaps: 50004999
Worst - Size: 10000, Comparisons: 49995000, Swaps: 25004999
Random - Size: 100000, Comparisons: 1980765, Swaps: 1149594
Best - Size: 100000, Comparisons: 4999950000, Swaps: 5000049999
Worst - Size: 100000, Comparisons: 4999950000, Swaps: 2500049999
Random - Size: 1000000, Comparisons: 24970335, Swaps: 13827148
```

Зовнішнє сортування:

Метод сортування природним злиттям (Natural Merge Sort) є вдосконаленням класичного методу злиття. Він використовує природні впорядковані підпослідовності (runs), які вже є у вхідних даних, замість розбиття на фіксовані блоки. Цей підхід скорочує кількість етапів сортування та злиття.

Алгоритм природного злиття:
Пошук підпослідовностей (runs): Пройдіть через масив (або файл) і визначте природні впорядковані сегменти.
Злиття: Об’єднуйте ці підпослідовності в більші відсортовані частини доти, поки не отримаєте один відсортований масив.
Кількість порівнянь і переміщень: Підраховуємо порівняння під час визначення runs та під час злиття.

Очікувані результати:

Найкращий випадок (вже відсортований масив): Кількість runs дорівнює 1. Сортування завершується за O(n).
Найгірший випадок (зворотний порядок): Кількість runs дорівнює розміру масиву.
Складність O(n*logn)
Середній випадок: Залежить від довжини вже впорядкованих runs.

Код програми:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Глобальні змінні для підрахунку порівнянь і переміщень
long long comparisons = 0;
long long swaps = 0;

// Функція для створення випадкового масиву
void fillRandom(int *arr, int size) {
    srand(time(NULL));
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 10000;  // Випадкові числа до 10000
    }
}

// Функція для створення відсортованого масиву (найкращий випадок)
void fillSorted(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;
    }
}

// Функція для створення зворотно відсортованого масиву (найгірший випадок)
void fillReversed(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i;
    }
}
```

```
// Функція для копіювання підмасиву
void copyArray(int *source, int start, int end, int *destination) {
    for (int i = start; i <= end; i++) {
        destination[i - start] = source[i];
    }
}

// Функція для злиття двох підпослідовностей
void merge(int *arr, int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int leftSize = leftEnd - leftStart + 1;
    int rightSize = rightEnd - rightStart + 1;

    int *left = (int *)malloc(leftSize * sizeof(int));
    int *right = (int *)malloc(rightSize * sizeof(int));

    // Копіюємо підпослідовності
    copyArray(arr, leftStart, leftEnd, left);
    copyArray(arr, rightStart, rightEnd, right);

    int i = 0, j = 0, k = leftStart;

    // Злиття
    while (i < leftSize && j < rightSize) {
        comparisons++; // Порівняння
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
            swaps++; // Переміщення
        } else {
            arr[k++] = right[j++];
            swaps++; // Переміщення
        }
    }

    // Додаємо залишки
    while (i < leftSize) {
        arr[k++] = left[i++];
        swaps++; // Переміщення
    }
    while (j < rightSize) {
        arr[k++] = right[j++];
        swaps++; // Переміщення
    }

    free(left);
    free(right);
}

// Функція для пошуку природних підпослідовностей (runs)
```

```
int findRuns(int *arr, int size, int *runStarts) {
    int numRuns = 0;
    runStarts[numRuns++] = 0;

    for (int i = 1; i < size; i++) {
        comparisons++; // Порівняння для визначення меж runs
        if (arr[i] < arr[i - 1]) {
            runStarts[numRuns++] = i;
        }
    }

    return numRuns;
}

// Природне сортування злиттям
void naturalMergeSort(int *arr, int size) {
    int *runStarts = (int *)malloc((size + 1) * sizeof(int)); // Для зберігання стартів runs

    while (1) {
        // Знаходимо runs
        int numRuns = findRuns(arr, size, runStarts);
        if (numRuns == 1) break; // Масив відсортовано

        // По черзі зливаємо runs
        for (int i = 0; i < numRuns - 1; i += 2) {
            int leftStart = runStarts[i];
            int leftEnd = runStarts[i + 1] - 1;
            int rightStart = runStarts[i + 1];
            int rightEnd = (i + 2 < numRuns) ? runStarts[i + 2] - 1 : size - 1;

            merge(arr, leftStart, leftEnd, rightStart, rightEnd);
        }
    }

    free(runStarts);
}

// Основна функція
int main() {
    // Розміри тестових масивів
    int sizes[] = {100, 1000, 10000};
    int numTests = 3;

    for (int t = 0; t < numTests; t++) {
        int size = sizes[t];
        int *arr = (int *)malloc(size * sizeof(int));

        // Тест для випадкових даних
```

```
        fillRandom(arr, size);
        comparisons = 0;
        swaps = 0;
        naturalMergeSort(arr, size);
        printf("Random - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons,
swaps);

        // Тест для найкращого випадку
        fillSorted(arr, size);
        comparisons = 0;
        swaps = 0;
        naturalMergeSort(arr, size);
        printf("Best - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons, swaps);

        // Тест для найгіршого випадку
        fillReversed(arr, size);
        comparisons = 0;
        swaps = 0;
        naturalMergeSort(arr, size);
        printf("Worst - Size: %d, Comparisons: %lld, Swaps: %lld\n", size, comparisons,
swaps);

        free(arr);
    }

    return 0;
}
```

```
Random - Size: 100, Comparisons: 1183, Swaps: 564
Best - Size: 100, Comparisons: 99, Swaps: 0
Worst - Size: 100, Comparisons: 1108, Swaps: 688
Random - Size: 1000, Comparisons: 18197, Swaps: 8985
Best - Size: 1000, Comparisons: 999, Swaps: 0
Worst - Size: 1000, Comparisons: 15921, Swaps: 9984
Random - Size: 10000, Comparisons: 258635, Swaps: 126255
Best - Size: 10000, Comparisons: 9999, Swaps: 0
Worst - Size: 10000, Comparisons: 214593, Swaps: 136320
```