МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Індивідуальне завдання № 4

з курсу «Алгоритми і структури даних»

на тему: «Методи сортування»

Виконала: студентка 2 курсу групи МФ-22

напряму підготовки (спеціальності)

Комп’ютерні науки

122 Комп’ютерні науки

Мухачова В.В.

Харків – 2024

Завдання:

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.

Аналіз коду:

Аналіз функцій:

```
void merge(int arr[], int left, int mid, int right, int *comparisons, int *moves)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
    {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++)
    {
        R[i] = arr[mid + 1 + i];
    }

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
        (*comparisons)++;
        (*moves)++;
    }
```

```
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
        (*moves)++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
        (*moves)++;
    }
}
```

Функція merge виконує злиття двох відсортованих підмасивів в один масив і є ключовою частиною алгоритму Merge Sort.

```
void mergeSort(int arr[], int left, int right, int *comparisons, int *moves)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid, comparisons, moves);
        mergeSort(arr, mid + 1, right, comparisons, moves);

        merge(arr, left, mid, right, comparisons, moves);
    }
}
```

Функція mergeSort реалізує алгоритм сортування злиттям (Merge Sort). Рекурсивно сортує ліву і праву половини і зліває відсортовані половини.

```c
void shellSort(int arr[], int n, int *comparisons, int *moves)
{
    int gap, i, j, temp;

    for (gap = n / 2; gap > 0; gap /= 2)
    {
        for (i = gap; i < n; i++)
        {
            temp = arr[i];
            j = i;
            while (j >= gap)
            {
                (*comparisons)++;
                if (arr[j - gap] > temp)
                {
                    arr[j] = arr[j - gap];
                    (*moves)++;
                    j -= gap;
                }
                else
                {
                    break;
                }
            }
            arr[j] = temp;
            (*moves)++;
        }
    }
}
```

Функція shellSort реалізує алгоритм сортування Шелла (Shell Sort), який є оптимізованою версією сортування вставками. Цей алгоритм використовує проміжки (gaps) для порівняння та переміщення елементів, що дозволяє швидше впорядковувати масив.

```c
//масив вже відсортований
void generate_best_case(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 1;
    }
}
```

Функція generate_best_case генерує масив в найкращому випадку для сортування, тобто масив вже відсортований.

```
//масив відсортований у зворотному порядку
void generate_worst_case(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = n - i;
    }
}
```

Функція generate_worst_case генерує масив в найгіршому випадку для сортування, тобто масив відсортований в зворотньому порядку.

```
//випадковий масив
void generate_random_case(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 1000;
    }
}
```

Функція generate_random_case генерує рандомний масив.

```c
int main()
{
    srand(time(NULL));

    int sizes[] = {100, 1000, 10000, 100000, 1000000};
    int num_sizes = 5;


    for (int i = 0; i < num_sizes; i++)
    {
        int n = sizes[i];
        int arr_best[n], arr_worst[n], arr_random[n];
        int comparisons, moves;

        //тест на найкращому випадку
        generate_best_case(arr_best, n);
        comparisons = 0; moves = 0;
        shellSort(arr_best, n, &comparisons, &moves);
        printf("Shell Sort (Best):   Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);

        //тест на найгіршому випадку
        generate_worst_case(arr_worst, n);
        comparisons = 0; moves = 0;
        shellSort(arr_worst, n, &comparisons, &moves);
        printf("Shell Sort (Worst):  Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);


        //тест на випадковому випадку
        generate_random_case(arr_random, n);
        comparisons = 0; moves = 0;
        shellSort(arr_random, n, &comparisons, &moves);
        printf("Shell Sort (Random): Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);
```

```c
        //тест на найкращому випадку
        generate_best_case(arr_best, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_best, 0, n, &comparisons, &moves);
        printf("Merge Sort (Best):   Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);

        //тест на найгіршому випадку
        generate_worst_case(arr_worst, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_worst, 0, n, &comparisons, &moves);
        printf("Merge Sort (Worst):  Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);

        //тест на випадковому випадку
        generate_random_case(arr_random, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_random, 0, n, &comparisons, &moves);
        printf("Merge Sort (Random): Size of the array=%d: Comparisons = %d, Moves = %d\n", n, comparisons, moves);
    }

    return 0;
}
```

Функція main генерує масиви, сортує їх і виводить результати.

Приклади роботи програми:

```
Shell Sort (Best):   Size of the array=100: Comparisons = 503, Moves = 503
Shell Sort (Worst):  Size of the array=100: Comparisons = 668, Moves = 763
Shell Sort (Random): Size of the array=100: Comparisons = 849, Moves = 891
Merge Sort (Best):   Size of the array=100: Comparisons = 361, Moves = 680
Merge Sort (Worst):  Size of the array=100: Comparisons = 413, Moves = 680
Merge Sort (Random): Size of the array=100: Comparisons = 555, Moves = 680
Shell Sort (Best):   Size of the array=1000: Comparisons = 8006, Moves = 8006
Shell Sort (Worst):  Size of the array=1000: Comparisons = 11716, Moves = 12706
Shell Sort (Random): Size of the array=1000: Comparisons = 14476, Moves = 15004
Merge Sort (Best):   Size of the array=1000: Comparisons = 5049, Moves = 9987
Merge Sort (Worst):  Size of the array=1000: Comparisons = 4945, Moves = 9987
Merge Sort (Random): Size of the array=1000: Comparisons = 8716, Moves = 9987
Shell Sort (Best):   Size of the array=10000: Comparisons = 120005, Moves = 120005
Shell Sort (Worst):  Size of the array=10000: Comparisons = 172578, Moves = 182565
Shell Sort (Random): Size of the array=10000: Comparisons = 264605, Moves = 269658
Merge Sort (Best):   Size of the array=10000: Comparisons = 69018, Moves = 133631
Merge Sort (Worst):  Size of the array=10000: Comparisons = 64613, Moves = 133631
Merge Sort (Random): Size of the array=10000: Comparisons = 120525, Moves = 133631
Shell Sort (Best):   Size of the array=100000: Comparisons = 1500006, Moves = 1500006
Shell Sort (Worst):  Size of the array=100000: Comparisons = 2244585, Moves = 2344566
Shell Sort (Random): Size of the array=100000: Comparisons = 3526831, Moves = 3577266
Merge Sort (Best):   Size of the array=100000: Comparisons = 853916, Moves = 1668946
Merge Sort (Worst):  Size of the array=100000: Comparisons = 815030, Moves = 1668946
Merge Sort (Random): Size of the array=100000: Comparisons = 1536031, Moves = 1668946
```

Висновки:

1. Порівняння Shell Sort та Merge Sort

- Shell Sort: Ефективність сортування значною мірою залежить від початкової послідовності даних. У найкращому випадку кількість порівнянь і переміщень менша, ніж у найгіршому. Для випадкових даних витрати часто наближаються до найгіршого випадку, але залишаються трохи нижчими. Зі збільшенням розміру масиву (n) кількість порівнянь і переміщень зростає значно швидше порівняно з Merge Sort. Часова складність Shell Sort наближається до O(n^2) для найгірших сценаріїв залежно від вибраної послідовності gap.
- Merge Sort: Кількість переміщень залишається сталою для всіх типів початкових послідовностей, оскільки алгоритм завжди створює тимчасові масиви однакової довжини. Кількість порівнянь варіюється залежно від початкової послідовності: Найменша для впорядкованих даних (найкращий випадок). Найбільша для випадкових даних. Часова складність залишається близькою до O(n \log n), що відображається в повільнішому зростанні кількості операцій при збільшенні n.

2. Відповідність трудомісткості теоретичним оцінкам

- Shell Sort: У найкращому випадку трудомісткість близька до O(n \log n), але за рахунок додаткових переміщень у циклах витрати можуть бути ближчими до O(n^(3/2)) або навіть O(n^2). Виявлено зростання кількості порівнянь і переміщень при збільшенні розміру масиву, особливо для випадкових даних. Для впорядкованих даних алгоритм виконує менше операцій, що підтверджує його ефективність у найкращому випадку.
- Merge Sort: Теоретична трудомісткість O(n\logn) підтверджується результатами. Незмінна кількість переміщень у всіх випадках пов'язана зі структурою алгоритму (завжди створюються нові масиви). Залежність кількості порівнянь від структури даних відповідає очікуваній поведінці.

3. Порівняння методів

- Merge Sort демонструє стабільну поведінку та очікувану трудомісткість навіть для великих масивів, що робить його кращим вибором для впорядковування великих даних.
- Shell Sort може бути більш ефективним для невеликих масивів або в найкращому випадку, але не є надійним для великих даних через зростання трудомісткості у найгіршому випадку.

Оригинальний код програми:

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

void generate_best_case(int arr[], int n);
void generate_worst_case(int arr[], int n);
void generate_random_case(int arr[], int n);
void merge(int arr[], int left, int mid, int right, int *comparisons, int
*moves);
void mergeSort(int arr[], int left, int right, int *comparisons, int *moves);
void shellSort(int arr[], int n, int *comparisons, int *moves);
```

```c
int main()
{
    srand(time(NULL));
    int sizes[] = {100, 1000, 10000, 100000, 1000000};
    int num_sizes = 5;
    for (int i = 0; i < num_sizes; i++)
    {
        int n = sizes[i];
        int arr_best[n], arr_worst[n], arr_random[n];
        int comparisons, moves;

        //тест на найкращому випадку
        generate_best_case(arr_best, n);
        comparisons = 0; moves = 0;
        shellSort(arr_best, n, &comparisons, &moves);
        printf("Shell Sort (Best):   Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);

        //тест на найгіршому випадку
        generate_worst_case(arr_worst, n);
        comparisons = 0; moves = 0;
        shellSort(arr_worst, n, &comparisons, &moves);
        printf("Shell Sort (Worst):  Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);


        //тест на випадковому випадку
        generate_random_case(arr_random, n);
        comparisons = 0; moves = 0;
        shellSort(arr_random, n, &comparisons, &moves);
        printf("Shell Sort (Random): Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);

        //тест на найкращому випадку
        generate_best_case(arr_best, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_best, 0, n, &comparisons, &moves);
```

```
        printf("Merge Sort (Best):   Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);

        //тест на найгіршому випадку
        generate_worst_case(arr_worst, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_worst, 0, n, &comparisons, &moves);
        printf("Merge Sort (Worst):  Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);

        //тест на випадковому випадку
        generate_random_case(arr_random, n);
        comparisons = 0; moves = 0;
        mergeSort(arr_random, 0, n, &comparisons, &moves);
        printf("Merge Sort (Random): Size of the array=%d: Comparisons = %d,
Moves = %d\n", n, comparisons, moves);
    }

    return 0;
}
//масив вже відсортований
void generate_best_case(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = i + 1;
    }
}
//масив відсортований у зворотному порядку
void generate_worst_case(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = n - i;
    }
}
//випадковий масив
void generate_random_case(int arr[], int n)
{
```

```
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 1000;
    }
}

//отримує два відсортованих масива зливає їх в один
void merge(int arr[], int left, int mid, int right, int *comparisons, int *moves)
{
    //розрахунок кількості елементів в лівому і правому масивах
    int n1 = mid - left + 1;
    int n2 = right - mid;

    //створення тимчасових масивів
    int L[n1], R[n2];

    //копіювання масивів в тимчасові масиви
    for (int i = 0; i < n1; i++)
    {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++)
    {
        R[i] = arr[mid + 1 + i];
    }

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

    //іде по підмасивам
    while (i < n1 && j < n2)
    {
        //ящко елемент з лівого меньше то він копіюється в масив
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        //інакше копіюється правий
```

```
        {
            arr[k] = R[j];
            j++;
        }
        k++;
        (*comparisons)++;
        (*moves)++;
    }

    //якщо залишаються елементи в одному з масивів, то копіює їх в основний масив
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
        (*moves)++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
        (*moves)++;
    }
}

//рекурсивно сортує половини масива і зливає відсортовані половини
void mergeSort(int arr[], int left, int right, int *comparisons, int *moves)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid, comparisons, moves);
        mergeSort(arr, mid + 1, right, comparisons, moves);

        merge(arr, left, mid, right, comparisons, moves);
    }
```

```
}
//сортування Шелла
void shellSort(int arr[], int n, int *comparisons, int *moves)
{
    int gap, i, j, temp;
    //починаємо з gap=n/2 зменьшуємо по gap=1
    for (gap = n / 2; gap > 0; gap /= 2)
    {
        for (i = gap; i < n; i++)
        {
            temp = arr[i];
            j = i;
            //цикл для переміщення елементів
            while (j >= gap)
            {
                (*comparisons)++;
                //порівнює елменти на відстані gap
                if (arr[j - gap] > temp)
                {
                    //якщо більше переміщує
                    arr[j] = arr[j - gap];
                    (*moves)++;
                    //зменьшує j
                    j -= gap;
                }
                else
                {
                    break;
                }
            }
            //після того як знайдена правильна позиція j для елемента temp
вставляємо в масив
            arr[j] = temp;
            (*moves)++;
        }
    }
}
```