МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Індивідуальна робота № 4

з курсу «Алгоритми і структури даних»
(назва дисципліни)

Виконала: студентка 2 курсу групи мф-21
напряму підготовки (спеціальності)
Комп'ютерні науки
122 Комп'ютерні науки
(шифр і назва напряму підготовки (спеціальності))
Лобанова Д.В.
(прізвище й ініціали студента)

Харків – 2024

Мета програми:
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.

Вхідні дані:

1. Масиви розміром 100, 1000, 10000, 100000, 1000000 елементів.
2. Тип початкової послідовності:
    - "best" (кращий випадок)
    - "worst" (найгірший випадок)
    - "average" (середній випадок)

Вихідні дані:

1. Кількість порівнянь і переміщень для кожного методу сортування в залежності від розміру масиву та типу початкової послідовності.
2. Результати тестування записані у файл sort_statistics.txt.

Обґрунтування вибору методів сортування:

1. **Метод Шелла.** Це вдосконалена версія сортування вставками, яка працює швидше, особливо з великими масивами даних. Замість того щоб переміщати елементи по одному, метод спочатку сортує елементи на певній відстані один від одного. Потім ці відстані зменшуються, і масив поступово стає впорядкованим. Це зменшує кількість переміщень і прискорює сортування.
2. **Метод природного злиття.** Цей метод добре підходить для дуже великих обсягів даних, навіть якщо вони не поміщаються в оперативну пам’ять. Він знаходить уже впорядковані частини масиву і об'єднує їх, поступово збираючи весь масив у потрібному порядку. Тобто, він особливо підходить, якщо дані спочатку частково впорядковані або якщо вони не поміщаються в оперативну пам'ять.

Опис програми

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdlib>

using namespace std;
```

Клас для сортування методом Шелла

```
class ShellSort {
public:
   int comp = 0; // Лічильник порівнянь
   int mov = 0; // Лічильник переміщень

   Функція для сортування масиву методом Шелла

   void sort(int arr[], int n) {
      int gap = n / 2; // Початковий інтервал
      while (gap > 0) {
         for (int i = gap; i < n; ++i) {
            int temp = arr[i];
            int j = i;

            // Переміщення елементів за умовою
            while (j >= gap && arr[j - gap] > temp) {
               arr[j] = arr[j - gap];
               j -= gap;
               comp++;
               mov++;
            }
            arr[j] = temp;
            mov++;
         }
         gap /= 2; // Зменшуємо інтервал
      }
   }
};
```

Клас для сортування методом сортування природним злиттям

```
class MergeSort {
public:
   int comp = 0; // Лічильник порівнянь
   int mov = 0;  // Лічильник переміщень

   Функція для злиття двох відсортованих частин масиву

   void merge(int arr[], int left, int mid, int right) {
      int n1 = mid - left + 1; // Розмір лівої частини
      int n2 = right - mid; // Розмір правої частини

      // Виділяємо пам'ять для тимчасових масивів
      int* L = new int[n1];
      int* R = new int[n2];
```

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

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

    // Злиття елементів з двох масивів у вихідний
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
        comp++;
        mov++;
    }

    // Копіюємо залишки лівої частини
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
        mov++;
    }

    // Копіюємо залишки правої частини
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
        mov++;
    }

    // Звільняємо пам'ять
    delete[] L;
    delete[] R;
}
```

Функція природного злиття

```
    void natSort(int arr[], int left, int right) {
        int* temp = new int[right + 1]; // Тимчасовий масив для копій
        while (true) {
            int start = left;
            bool sorted = true;

            while (start < right) {
                int mid = start;

                // Знаходимо першу відсортовану послідовність
                while (mid < right && arr[mid] <= arr[mid + 1]) {
                    mid++;
                    comp++;
                }

                if (mid == right) break;

                int end = mid + 1;

                // Знаходимо другу відсортовану послідовність
                while (end < right && arr[end] <= arr[end + 1]) {
                    end++;
                    comp++;
                }

                // Зливаємо ці послідовності
                merge(arr, start, mid, end);
                sorted = false;
                start = end + 1;
            }

            if (sorted) break; // Якщо весь масив відсортований
        }
        delete[] temp;
    }
};
```

Функція для генерації масивів

```
void generateArr(int* arr, int n, const string& caseType) {
    if (caseType == "best") {
        // Кращий випадок (масив вже відсортований)
        for (int i = 0; i < n; i++) {
            arr[i] = i + 1;
        }
    } else if (caseType == "worst") {
        // Найгірший випадок (масив відсортований у зворотному порядку)
        for (int i = 0; i < n; i++) {
```

```
            arr[i] = n - i;
        }
    } else if (caseType == "average") {
        // Середній випадок (масив заповнений випадковими числами)
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 1000000;
        }
    }
}
```

Функція для копіювання масиву

```
void copyArr(int* arr, int* arrCopy, int size) {
    for (int i = 0; i < size; i++) {
        arrCopy[i] = arr[i];
    }
}
```

Функція для виконання тестів та запису результатів у файл

```
void Tests(int n, const string& caseType, ofstream& outputFile) {
    // Генерація масиву для тесту
    int* arr = new int[n];
    generateArr(arr, n, caseType);

    // Тестування ShellSort
    ShellSort shellSort;
    int* arrCopy = new int[n];
    copyArr(arr, arrCopy, n);
    shellSort.sort(arrCopy, n);
    outputFile << "ShellSort (" << caseType << "): Comparisons = "
               << shellSort.comp << ", Movements = " << shellSort.mov << endl;

    // Тестування MergeSort
    MergeSort mergeSort;
    copyArr(arr, arrCopy, n);
    mergeSort.natSort(arrCopy, 0, n - 1);
    outputFile << "MergeSort (" << caseType << "): Comparisons = "
               << mergeSort.comp << ", Movements = " << mergeSort.mov << endl;

    delete[] arr;
    delete[] arrCopy;
}
```

Основна функція програми

```
int main() {
    srand(time(0)); // Ініціалізація генератора випадкових чисел
```

```
    // Створення вихідного файлу
    ofstream outputFile("sort_statistics.txt");
    if (!outputFile) {
        cerr << "Error opening output file." << endl;
        return 1;
    }

    // Масив розмірів масивів для тестування
    int sizes[] = {100, 1000, 10000, 100000, 1000000};
    string caseTypes[] = {"best", "worst", "average"};

    // Виконання тестів для кожного розміру та випадку
    for (int size : sizes) {
        outputFile << "\nTesting with array size: " << size << "\n";
        for (const string& caseType : caseTypes) {
            Tests(size, caseType, outputFile);
        }
    }

    outputFile.close(); // Закриваємо файл
    cout << "Results have been written to sort_statistics.txt" << endl;

    return 0;
}
```

Код програми

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdlib>

using namespace std;

// Клас для сортування методом Шелла
class ShellSort {
public:
    int comp = 0; // Лічильник порівнянь
    int mov = 0; // Лічильник переміщень

    // Функція для сортування масиву методом Шелла
    void sort(int arr[], int n) {
        int gap = n / 2; // Початковий інтервал
        while (gap > 0) {
            for (int i = gap; i < n; ++i) {
                int temp = arr[i];
                int j = i;

                // Переміщення елементів за умовою
                while (j >= gap && arr[j - gap] > temp) {
                    arr[j] = arr[j - gap];
                    j -= gap;
                    comp++;
                    mov++;
                }
                arr[j] = temp;
                mov++;
```

```
            }
            gap /= 2; // Зменшуємо інтервал
        }
    }
};

// Клас для сортування методом сортування природним злиттям
class MergeSort {
public:
    int comp = 0; // Лічильник порівнянь
    int mov = 0;  // Лічильник переміщень

    // Функція для злиття двох відсортованих частин масиву
    void merge(int arr[], int left, int mid, int right) {
        int n1 = mid - left + 1; // Розмір лівої частини
        int n2 = right - mid; // Розмір правої частини

        // Виділяємо пам'ять для тимчасових масивів
        int* L = new int[n1];
        int* R = new int[n2];

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

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

        // Злиття елементів з двох масивів у вихідний
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
            comp++;
            mov++;
        }

        // Копіюємо залишки лівої частини
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
            mov++;
        }

        // Копіюємо залишки правої частини
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
            mov++;
        }

        // Звільняємо пам'ять
        delete[] L;
        delete[] R;
    }

    // Функція природного злиття
    void natSort(int arr[], int left, int right) {
        int* temp = new int[right + 1]; // Тимчасовий масив для копій
        while (true) {
            int start = left;
            bool sorted = true;

            while (start < right) {
                int mid = start;
```

```
            // Знаходимо першу відсортовану послідовність
            while (mid < right && arr[mid] <= arr[mid + 1]) {
                mid++;
                comp++;
            }

            if (mid == right) break;

            int end = mid + 1;

            // Знаходимо другу відсортовану послідовність
            while (end < right && arr[end] <= arr[end + 1]) {
                end++;
                comp++;
            }

            // Зливаємо ці послідовності
            merge(arr, start, mid, end);
            sorted = false;
            start = end + 1;
        }

        if (sorted) break; // Якщо весь масив відсортований
    }
    delete[] temp;
  }
};

// Функція для генерації масивів
void generateArr(int* arr, int n, const string& caseType) {
  if (caseType == "best") {
      // Кращий випадок (масив вже відсортований)
      for (int i = 0; i < n; i++) {
          arr[i] = i + 1;
      }
  } else if (caseType == "worst") {
      // Найгірший випадок (масив відсортований у зворотному порядку)
      for (int i = 0; i < n; i++) {
          arr[i] = n - i;
      }
  } else if (caseType == "average") {
      // Середній випадок (масив заповнений випадковими числами)
      for (int i = 0; i < n; i++) {
          arr[i] = rand() % 1000000;
      }
  }
}

// Функція для копіювання масиву
void copyArr(int* arr, int* arrCopy, int size) {
  for (int i = 0; i < size; i++) {
      arrCopy[i] = arr[i];
  }
}

// Функція для виконання тестів та запису результатів у файл
void Tests(int n, const string& caseType, ofstream& outputFile) {
  // Генерація масиву для тесту
  int* arr = new int[n];
  generateArr(arr, n, caseType);

  // Тестування ShellSort
  ShellSort shellSort;
  int* arrCopy = new int[n];
  copyArr(arr, arrCopy, n);
  shellSort.sort(arrCopy, n);
  outputFile << "ShellSort (" << caseType << "): Comparisons = "
             << shellSort.comp << ", Movements = " << shellSort.mov << endl;

  // Тестування MergeSort
  MergeSort mergeSort;
  copyArr(arr, arrCopy, n);
  mergeSort.natSort(arrCopy, 0, n - 1);
  outputFile << "MergeSort (" << caseType << "): Comparisons = "
             << mergeSort.comp << ", Movements = " << mergeSort.mov << endl;
```

```cpp
    delete[] arr;
    delete[] arrCopy;
}

int main() {
    srand(time(0)); // Ініціалізація генератора випадкових чисел

    // Створення вихідного файлу
    ofstream outputFile("sort_statistics.txt");
    if (!outputFile) {
        cerr << "Error opening output file." << endl;
        return 1;
    }

    // Масив розмірів масивів для тестування
    int sizes[] = {100, 1000, 10000, 100000, 1000000};
    string caseTypes[] = {"best", "worst", "average"};

    // Виконання тестів для кожного розміру та випадку
    for (int size : sizes) {
        outputFile << "\nTesting with array size: " << size << "\n";
        for (const string& caseType : caseTypes) {
            Tests(size, caseType, outputFile);
        }
    }

    outputFile.close(); // Закриваємо файл
    cout << "Results have been written to sort_statistics.txt" << endl;

    return 0;
}
```

Результати програми

```
Results have been written to sort_statistics.txt


Process finished with exit code 0
```

sort_statistics.txt

File Edit View

```
Testing with array size: 100
ShellSort (best): Comparisons = 0, Movements = 503
MergeSort (best): Comparisons = 99, Movements = 0
ShellSort (worst): Comparisons = 260, Movements = 763
MergeSort (worst): Comparisons = 914, Movements = 688
ShellSort (average): Comparisons = 388, Movements = 891
MergeSort (average): Comparisons = 1107, Movements = 586

Testing with array size: 1000
ShellSort (best): Comparisons = 0, Movements = 8006
MergeSort (best): Comparisons = 999, Movements = 0
ShellSort (worst): Comparisons = 4700, Movements = 12706
MergeSort (worst): Comparisons = 13931, Movements = 9984
ShellSort (average): Comparisons = 7738, Movements = 15744
MergeSort (average): Comparisons = 17237, Movements = 8982

Testing with array size: 10000
ShellSort (best): Comparisons = 0, Movements = 120005
MergeSort (best): Comparisons = 9999, Movements = 0
ShellSort (worst): Comparisons = 62560, Movements = 182565
MergeSort (worst): Comparisons = 194603, Movements = 136320
ShellSort (average): Comparisons = 150385, Movements = 270390
MergeSort (average): Comparisons = 248479, Movements = 126142

Testing with array size: 100000
ShellSort (best): Comparisons = 0, Movements = 1500006
MergeSort (best): Comparisons = 99999, Movements = 0
ShellSort (worst): Comparisons = 844560, Movements = 2344566
MergeSort (worst): Comparisons = 2415018, Movements = 1692992
ShellSort (average): Comparisons = 2883845, Movements = 4383851
MergeSort (average): Comparisons = 3116027, Movements = 1592224

Testing with array size: 1000000
ShellSort (best): Comparisons = 0, Movements = 18000007
MergeSort (best): Comparisons = 999999, Movements = 0
ShellSort (worst): Comparisons = 9357504, Movements = 27357511
MergeSort (worst): Comparisons = 28884985, Movements = 19980544
ShellSort (average): Comparisons = 48654815, Movements = 66654822
MergeSort (average): Comparisons = 37218139, Movements = 18978384
```