

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**Український державний університет науки і технологій**

Кафедра «Комп’ютерні інформаційні технології»

**Лабораторна робота №2**

**з дисципліни «Алгоритми та структури даних»**

**на тему: «Аналіз алгоритмів»**

Виконав:

студент гр. ПЗ2011 Грива Я. А.

Прийняла:

Демидович І. М.

Дніпро, 2022

1. **Постановка задачі**

Напишіть програму мовою java для визначення обчислювальної складності алгоритмів сортування бульбашкою, швидкого сортування та методу відповідно до індивідуального завдання. Проведіть експеримент за схемою, описаною в теоретичних відомостях. В ході роботи програма має записувати середні значення обчислювальної складності у файл. За отриманими значеннями побудуйте графіки залежності обчислювальної складності в залежності від кількості (розміру) вхідних даних. Визначте порядок обчислювальної складності.

1. **Індивідуальне завдання**

Написати програму сортування вибором.

1. **Опис схеми експерименту**

Для трьох алгоритмів сортування проводиться аналіз середньої обчислювальної складності. Масиви створюються генератором випадкових чисел. Результати розрахунків повинні друкуватися у відповідний текстовий файл.

За отриманими даними буде побудовано графік залежності середньої кількості обчислень для вказаних вище розмірім масивів для трьох зазначених алгоритмів.

**Сортування вибором**

Алгоритм працює таким чином:

1. Знаходить у списку найменше значення
2. Міняє його місцями із першим значенням у списку
3. Повторює два попередніх кроки, доки список не завершиться

Фактично, таким чином ми поділили список на дві частини: перша (ліва) — повністю відсортована, а друга (права) — ні.

Обчислювальна складність алгоритму:

Найгірший випадок: O(n2)

Середній випадок: О(n2)

Найкращий випадок: O(n)

**Сортування бульбашкою**

Сортування бульбашкою ділиться на кроки. На кожному кроці виконується попарне порівняння. В результаті кожного кроку найбільші елементи починають записуватись з кінця масиву. Тобто після першого кроку найбільший за значенням елемент масиву стоятиме на останньому місці. На другому етапі робота проводиться з усіма елементами крім останнього. Знову знаходиться найбільший елемент і ставиться в кінець масиву, з яким виконується робота. Третій крок повторює другий і так, поки масив не буде відсортований.

Обчислювальна складність алгоритму:

Найгірший випадок: O()

Середній випадок: O()

Найкращий випадок: O(n)

**Швидке сортування**

Вибрати із масиву опорний елемент. Порівняти всі інші елементи з опорним і переставити їх у масиві так, щоб розбити масив на три масиви, що йдуть один за одним: «менші опорного», «опорний» та «більше опорного». Для відрізків «менших» і «більших» значень виконати рекурсивно ту саму послідовність операцій, якщо довжина відрізка більша за одиницю.

Обчислювальна складність алгоритму виглядає:

Найгірший випадок: O()

Середній випадок: O(n log(n))

Найкращий випадок: O(n log(n))

1. **Текст програми**

Main.java

package com.company;  
import java.io.\*;  
  
public class Main {  
 public static void fillArray(int[] array)  
 {  
 for(int i = 0; i < array.length; i++)  
 {  
 array[i] = (int)(Math.*random*()\*100);  
 }  
 }  
  
 public static void main(String[] args)  
 {  
 int []arr1 = new int[100];  
 *fillArray*(arr1);  
 int []arr2 = new int[200];  
 *fillArray*(arr2);  
 int []arr3 = new int[300];  
 *fillArray*(arr3);  
 int []arr4 = new int[400];  
 *fillArray*(arr4);  
 int []arr5 = new int[500];  
 *fillArray*(arr5);  
  
 try(FileWriter fw = new FileWriter("Sort.txt", true))  
 {  
 Sort.*SelectionSort*(arr1);  
 int a = Sort.*getShakerSortCount*();  
 fw.write("Selection sort count with size 100: " + a + "**\n**");  
 Sort.*SelectionSort*(arr2);  
 a = Sort.*getShakerSortCount*();  
 fw.write("Selection sort count with size 200: " + a + "**\n**");  
 Sort.*SelectionSort*(arr3);  
 a = Sort.*getShakerSortCount*();  
 fw.write("Selection sort count with size 300: " + a + "**\n**");  
 Sort.*SelectionSort*(arr4);  
 a = Sort.*getShakerSortCount*();  
 fw.write("Selection sort count with size 400: " + a + "**\n**");  
 Sort.*SelectionSort*(arr5);  
 a = Sort.*getShakerSortCount*();  
 fw.write("Selection sort count with size 500: " + a + "**\n**");  
 }  
 catch (IOException e)  
 {  
 System.out.println(e.getMessage());  
 }  
 }  
}

Sort.java

package com.company;  
  
public class Sort  
{  
 private static int bubbleSortCount = 0;  
 private static int quickSortCount = 0;  
 private static int selectiomSortCount = 0;  
  
 public static int getBubbleSortCount()  
 {  
 return bubbleSortCount;  
 }  
 public static int getQuickSortCount()  
 {  
 return quickSortCount;  
 }  
 public static int getShakerSortCount()  
 {  
 return selectiomSortCount;  
 }  
  
 public static void BubbleSort(int[] array)  
 {  
 bubbleSortCount = 0;  
 for (int i = 0; i < array.length; i++)  
 {  
 for (int j = 0; j < array.length - i - 1; j++)  
 {  
 if (array[j]> array[j+1])  
 {  
 int temp = array[j];  
 array[j] = array[j+1];  
 array[j+1] = temp;  
 bubbleSortCount++;  
 }  
 }  
 }  
 }  
  
 public static void SelectionSort(int [] array) {  
 for (int i = 0; i < array.length; i++) {  
 int pos = i;  
 int min = array[i];  
 for (int j = i + 1; j < array.length; j++) {  
 if (array[j] < min) {  
 pos = j;  
 min = array[j];  
 selectiomSortCount++;  
 }  
 }  
 array[pos] = array[i];  
 array[i] = min;  
 selectiomSortCount++;  
 }  
 }  
  
 public static void QuickSort(int[] array)  
 {  
 quickSortCount = 0;  
 *QuickSort*(array, 0, array.length - 1);  
 System.out.println("Количество перестановок: " + quickSortCount);  
 }  
  
 private static int[] QuickSort(int[] array, int minIndex, int maxIndex)  
 {  
 if(minIndex >= maxIndex)  
 {  
 return array;  
 }  
 var pivotIndex = *Partition*(array, minIndex, maxIndex);  
  
 *QuickSort*(array, minIndex, pivotIndex - 1);  
 *QuickSort*(array, pivotIndex + 1, maxIndex);  
 return array;  
 }  
 private static int Partition(int[] array, int minIndex, int maxIndex)  
 {  
 var pivot = minIndex - 1;  
 for (var i = minIndex; i < maxIndex; i++)  
 {  
 if (array[i] < array[maxIndex])  
 {  
 pivot++;  
  
 var temp = array[pivot];  
 array[pivot] = array[i];  
 array[i] = temp;  
 quickSortCount++;  
  
 }  
 }  
  
 pivot++;  
 var temp = array[pivot];  
 array[pivot] = array[maxIndex];  
 array[maxIndex] = temp;  
 quickSortCount++;  
 return pivot;  
 }  
}

1. **Результати аналізу складності алгоритмів**

|  |  |  |  |
| --- | --- | --- | --- |
| Size | QuickSort | BubleSort | SelectionSort |
| 100 | 294 | 2503 | 331 |
| 200 | 738 | 9600 | 1057 |
| 300 | 1069 | 22768 | 2055 |
| 400 | 1768 | 40199 | 3522 |
| 500 | 2411 | 61037 | 5493 |

Як видно з таблиці та діаграми, найбільш ефективним методом сортування є швидке сортування. Алгоритм використовує менше пам’яті, виконується за менший проміжок часу. Сортування бульбашкою та вибором є одними з самих простих алгоритмів, але не є ефективними. Алгоритми виконують дуже велику кількість операцій.

1. **Висновок**

У ході виконання лабораторної роботи був ознайомлений з алгоритмами сортування, та обчислюваною складністю алгоритмів. Був проведений експеримент, який на практиці показує швидкість роботи різних алгоритмів сортування. Усі значення були записані в таблицю та виведені на графік.