

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 Савенко Я. О.

Прийняла:

Куроп’ятник О. С.

Дніпро, 2022

**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.

**Індивідуальне завдання**

8. Шейкерне сортування

**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 бульбашкою**

У поданому наборі даних (списку чи масиві) порівнюються два сусідні елементи. Якщо один з елементів, не відповідає критерію сортування (є більшим, або ж, навпаки, меншим за свого сусіда), то ці два елементи міняються місцями. Прохід по списку продовжується доти, доки дані не будуть відсортованими.

Обчислювальна складність алгоритму:

Найгірший випадок: O()

Середній випадок: O()

Найкращий випадок: O(n)

**Швидке сортування**

Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної з частин відбувається рекурсивно. Алгоритм швидкого сортування може бути реалізований як у масиві, так і в двозв'язному списку.

Обчислювальна складність алгоритму виглядає:

Найгірший випадок: O()

Середній випадок: O(n log(n))

Найкращий випадок: O(n log(n))

**Шейкерне сортування**

Сортування змішуванням мало чим відрізняється від сортування бульбашкою. Єдина його відмінність у тому, що замість багаторазового проходження через список знизу вгору, він проходить по черзі знизу вгору і згори вниз. Він може досягати трохи вищої ефективності, ніж алгоритм сортування бульбашкою. Причиною цьому є те, що алгоритм сортування бульбашкою проходить по списку лише в одному напрямі, а тому за одну ітерацію елементи списку можна перемістити лише на один крок.

Обчислювальна складність алгоритму виглядає:

Найгірший випадок: O()

Середній випадок: O()

Найкращий випадок: O(n)

**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.shakerSort(arr1);  
 int a = Sort.getShakerSortCount();  
 fw.write("Shaker sort count with size 100: " + a + "\n");  
 Sort.shakerSort(arr2);  
 a = Sort.getShakerSortCount();  
 fw.write("Shaker sort count with size 200: " + a + "\n");  
 Sort.shakerSort(arr3);  
 a = Sort.getShakerSortCount();  
 fw.write("Shaker sort count with size 300: " + a + "\n");  
 Sort.shakerSort(arr4);  
 a = Sort.getShakerSortCount();  
 fw.write("Shaker sort count with size 400: " + a + "\n");  
 Sort.shakerSort(arr5);  
 a = Sort.getShakerSortCount();  
 fw.write("Shaker 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 shakerSortCount = 0;  
  
 public static int getBubbleSortCount()  
 {  
 return bubbleSortCount;  
 }  
 public static int getQuickSortCount()  
 {  
 return quickSortCount;  
 }  
 public static int getShakerSortCount()  
 {  
 return shakerSortCount;  
 }  
  
 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 shakerSort(int [] array) {  
 int buff;  
 int left = 0;  
 int right = array.length - 1;  
 do {  
 for (int i = left; i < right; i++) {  
 if (array[i] > array[i + 1]) {  
 buff = array[i];  
 array[i] = array[i + 1];  
 array[i + 1] = buff;  
 shakerSortCount++;  
 }  
 }  
 right--;  
 for (int i = right; i > left; i--) {  
 if (array[i] < array[i - 1]) {  
 buff = array[i];  
 array[i] = array[i - 1];  
 array[i - 1] = buff;  
 shakerSortCount++;  
 }  
 }  
 left++;  
 } while (left < right);  
 }  
  
 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;  
 }  
}

**Результат роботи складності алгоритмів**

|  |  |  |  |
| --- | --- | --- | --- |
| Size | Shaker Sort | Buble Sort | Quick Sort |
| 100 | 2415 | 2503 | 294 |
| 200 | 11126 | 9600 | 738 |
| 300 | 33724 | 22768 | 1069 |
| 400 | 73315 | 40199 | 1768 |
| 500 | 121103 | 61037 | 2411 |

На таблиці видно, щл найбільш ефективним в дії є швидке сортування. Можна побачити, що шейкерне сортування працює більше ніж сортування бульбашкою, відбувається це із-за того, що в коді рахувалися дії в обох циклах з лічильником, якщо б підрахунки велися в головному циклі, то кількість дій була б приблизно однаковою з сортуванням бульбашкою.

**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.