Practica1:AnalisisExperimental

Alumno: Jose Luis Obiang Ela Nanguang

Profesor: Luis Vicente Calderita Estévez

Asignatura: Análisis y Diseño de Algoritmos (ADA)

Fecha Entrega: 19/03/2024 23:59

Índice

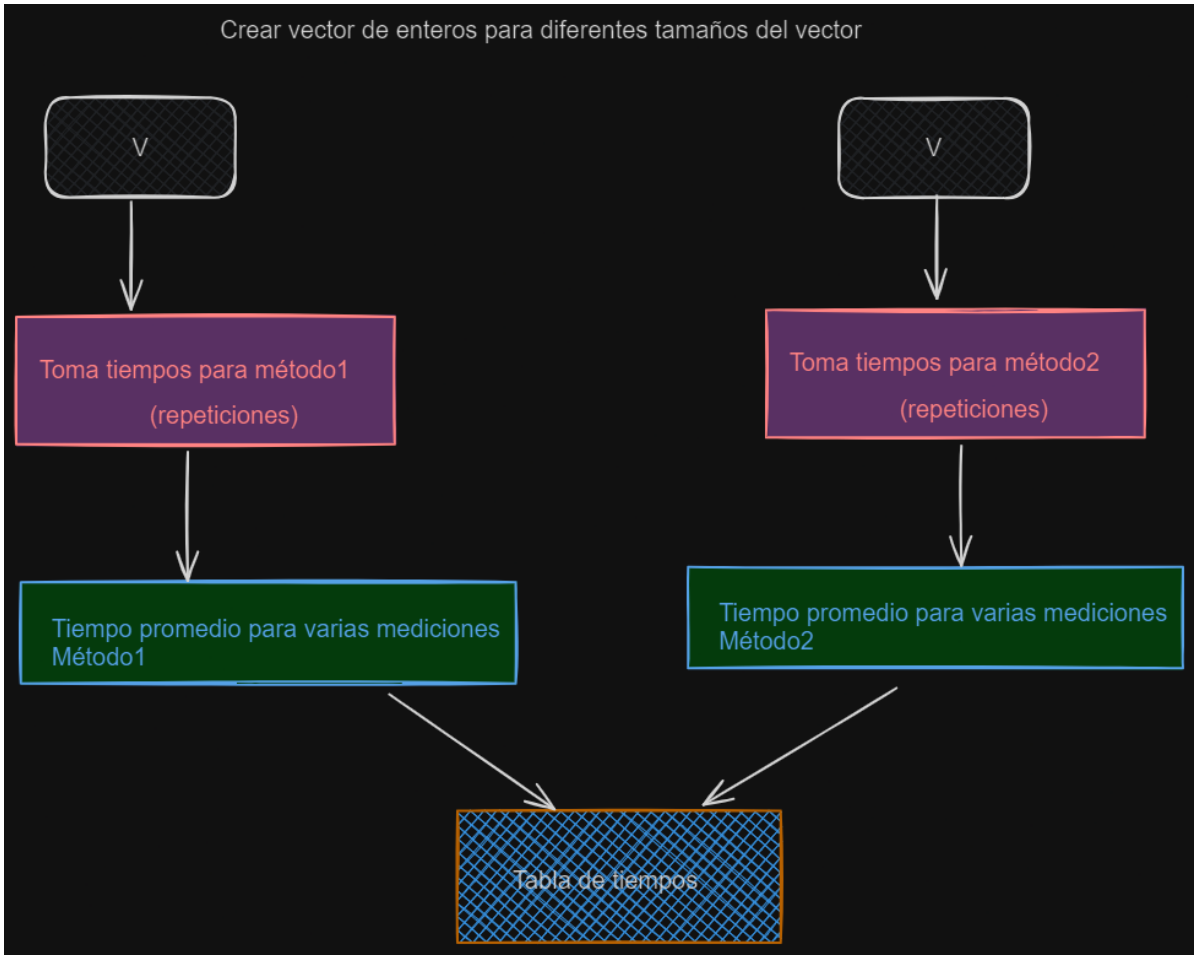
Análisis

Enunciado

1. Analizar el comportamiento temporal del
   algoritmo ante diferentes configuraciones de datos de entrada:
   a. Caso Mejor.
   b. Caso Peor.
   c. Datos de entrada aleatorios.
2. Se deben realizar pruebas temporales con
   tamaños de problema (talla) crecientes.
3. Realizar varias mediciones para cada talla, para reducir el efecto de las perturbaciones en el sistema en la toma de tiempos.

Objetivos

- Estudiar el coste temporal real de algunos algoritmos de ordenación.
- Los algoritmos ordenarán ascendentemente un vector de enteros de diferente tamaño.
- Implementar los algoritmos de ordenación: Selection sort y Bubble sort.
- Comparar el coste entre los algoritmos implementados y los algoritmos ya disponibles (Cocktail Sort y Quicksort).



Desarrollo

Código

▼ Clases

Main.java:

```java
package es.uex.cum.ada;

import javax.swing.*;
import java.awt.*;
import java.util.Arrays;

/**
 * Práctica 1: Análisis experimental de la Eficiencia de algunos Algoritmos de Ordenación
 * <ul>
 *     <li>Estudiar el coste temporal real de algunos algoritmos de ordenación</li>
 *     <li>Los algoritmos ordenaran ascendentemente un vector de enteros de diferente tamaño</li>
 *     <li>Implementar los algoritmos de ordenación Selection sort y Bubble sort</li>
 *     <li>Comparar el coste entre los algoritmos implementados y los algoritmos ya disponibles(Cocktail Sort y Bubble Sort)</li>
 *     <ul>Analizar el comportamiento temporal del algoritmo ante diferentes configuraciones de datos de entrada
 *          <li>Caso Mejor</li>
 *          <li>Caso Peor</li>
 *          <li>Datos de entrada aleatorios</li>
 *     </ul>
 *     <li>Se deben realizar pruebas temporales con tamaños de problema (talla) crecientes.</li>
 *     <li>Realizar varias mediciones para cada talla para reducir el efecto de perturbaciones en el sistema en la toma de tiempos.</li>
 * </ul>
 *
 * @author José Luis Obiang Ela Nanguang
 * @version 1.0 2024-03-18
 * {@link JFrame}, {@link JPanel}, {@link JTextArea}, {@link JTable}, {@link JScrollPane}, {@link GridLayout}, {@link BorderLayout}, {@link Algoritmos}, {@link InterfazAlgoritmos}
 */


public class Main {

    public static void main(String[] args) {
        // Crear una instancia de la clase Algoritmos
        Algoritmos al = new Algoritmos();

        // Definir los tamaños de los arrays a ordenar
        int[] tallas = {2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000};

        // Definir el número de repeticiones por cada tamaño
        int repeticiones = 10;

        // Definir los casos
        String[] casos = {"Caso Mejor", "Caso Peor", "Caso Aleatorio"};

        // Array para almacenar los casos, las tallas y tiempos medios de ejecución de cada algoritmo
        String[][][] tiempos = new String[casos.length][tallas.length][al.algoritmos.length];

        // Iterar sobre los casos
        for (int i = 0; i < casos.length; i++) {
            JPanel casoPanel = new JPanel(new BorderLayout());
            String caso = casos[i];
            System.out.println("========================================================== " + caso + "
==========================================================");

            // Iterar sobre los tamaños de los arrays
            for (int j = 0; j < tallas.length; j++) {
                // Crear una ventana Swing
                JFrame frame = new JFrame("Arrays - " + caso + " - Talla: " + tallas[j]);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                // Crear un panel principal para la ventana
                JPanel panel = new JPanel(new GridLayout(2, 1));
                JTextArea originalTextArea = new JTextArea();
                JTextArea ordenadoTextArea = new JTextArea();

                int talla = tallas[j];
                long[] tiemposTotales = new long[al.algoritmos.length];
                System.out.println("Talla del array: " + talla);

                // Generar el array según el caso
                int[] arrayOriginal = al.generarArray(talla, caso);
                // Mostrar el array original (solo para comprobar que funciona)
                System.out.println("Array original: " + Arrays.toString(arrayOriginal));

                int[] arrayOrdenado = new int[talla];
                // Iterar sobre las repeticiones
                for (int l = 0; l < al.algoritmos.length; l++) {
                    // Medir el tiempo de ejecución de cada algoritmo de ordenación
                    long inicio = System.currentTimeMillis();
                    for (int k = 0; k < repeticiones; k++) {
                        // Ordenar el array con el algoritmo correspondiente
                        arrayOrdenado = switch (l) {
                            case 0 -> al.bubbleSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
                            case 1 -> al.selectionSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
                            case 2 -> al.quickSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length), 0, arrayOriginal.length - 1);
                            case 3 -> al.cocktailSort(Arrays.copyOf(arrayOriginal, arrayOriginal.length));
                            default -> arrayOriginal;
                        };

                        long fin = System.currentTimeMillis();
                        tiemposTotales[l] += (fin - inicio);
                    }
                }

                // Calcular los tiempos medios
                for (int l = 0; l < al.algoritmos.length; l++) {
                    tiempos[i][j][l] = String.valueOf(tiemposTotales[l] / (double) repeticiones);
                    // Mostrar el array ordenado con el algoritmo correspondiente (solo para comprobar que funciona)
                    System.out.println("Array ordenado con algoritmo " + al.algoritmos[l] + ": " + Arrays.toString(arrayOrdenado));
                }

                // Mostrar el array original y el array ordenado
                originalTextArea.setEditable(false);
                originalTextArea.append("Array Original (" + caso + " - Talla: " + talla + "):\n" + Arrays.toString(arrayOriginal));
                ordenadoTextArea.setEditable(false);
                ordenadoTextArea.append("Array Ordenado (" + caso + " - Talla: " + talla + "):\n" + Arrays.toString(arrayOrdenado));

                panel.add(originalTextArea);
                panel.add(ordenadoTextArea);

                frame.add(panel);
                frame.setSize(400, 300);
                frame.setLocationRelativeTo(null); // Centrar la ventana en la pantalla
                frame.setVisible(true);


            }

System.out.println("==========================================================================================================================================\n")
;
        }

        // Mostrar los resultados finales en una tabla
        al.mostrarResultados(casos, tallas, tiempos);

        JFrame frameFinal = new JFrame("Análisis experimental");
        frameFinal.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Crear un panel principal para la ventana
        JPanel panelFinal = new JPanel(new GridLayout(1, casos.length));

        // Crear y agregar tablas para cada caso
        for (int i = 0; i < casos.length; i++) {
            String caso = casos[i];
            JTable table = al.createTable(caso, tallas, tiempos[i]);
            JScrollPane scrollPane = new JScrollPane(table);
            panelFinal.add(scrollPane);
        }

        // Agregar el panel principal a la ventana
        frameFinal.add(panelFinal);

        // Configurar la ventana y hacerla visible
        frameFinal.setSize(800, 600);
        frameFinal.setLocationRelativeTo(null); // Centrar la ventana en la pantalla
        frameFinal.setVisible(true);
    }


}

```

InterfazAlgoritmos.java

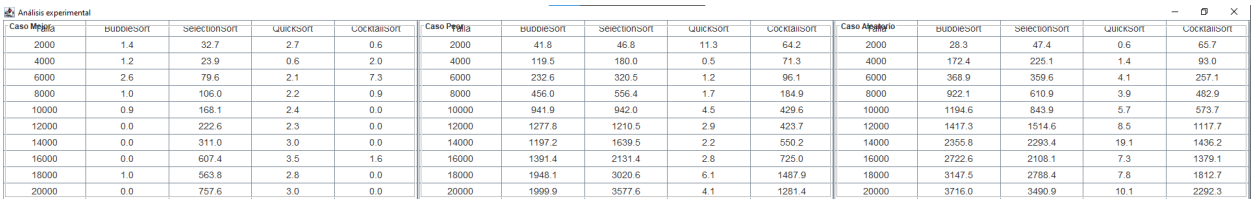
```java
package es.uex.cum.ada;

import javax.swing.*;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.util.Arrays;
import java.util.Random;

/**
 * Interfaz que contiene todas las funciones a usar en el programa
 * @author José Luis Obiang Ela Nanguang
 * @version 1.0 2024-03-18
 */

public interface InterfazAlgoritmos {
    int[] generarArray(int talla, String caso);
    void mostrarResultados(String[] casos, int[] tallas, String[][][] tiempos);
    int[] bubbleSort(int arr[]);
    int[] selectionSort(int arr[]);
    int[] cocktailSort(int arr[]);
    int[] quickSort(int a[], int izq, int der);
    void intercambiar(int a[], int ind1, int ind2);
    int mediana3(int a[], int izq, int der);
    int[] generateRandomArray(int size);
    int[] generateSortedArray(int size);
    int[] generateReverseSortedArray(int size);
    JTable createTable(String caso, int[] tallas, String[][] tiempos);


}

```

Algoritmos.java

Github

https://github.com/Jloen1999/ADA

Ejecución


Análisis experimental

Caso Mejor

Talla	BubbleSort	SelectionSort	QuickSort	CocktailSort
2000	1.4	32.7	2.7	0.6
4000	1.2	23.9	0.6	2.0
6000	2.6	79.6	2.1	7.3
8000	1.0	106.0	2.2	0.9
10000	0.9	168.1	2.4	0.0
12000	0.0	222.6	2.3	0.0
14000	0.0	311.0	3.0	0.0
16000	0.0	607.4	3.5	1.6
18000	1.0	563.8	2.8	0.0
20000	0.0	757.6	3.0	0.0

Caso Peor

Talla	BubbleSort	SelectionSort	QuickSort	CocktailSort
2000	41.8	46.8	11.3	64.2
4000	119.5	180.0	0.5	71.3
6000	232.6	320.5	1.2	96.1
8000	456.0	556.4	1.7	184.9
10000	941.9	942.0	4.5	429.6
12000	1277.8	1210.5	2.9	423.7
14000	1197.2	1639.5	2.2	550.2
16000	1391.4	2131.4	2.8	725.0
18000	1948.1	3020.6	6.1	1487.9
20000	1999.9	3577.6	4.1	1281.4

Caso Aleatorio

Talla	BubbleSort	SelectionSort	QuickSort	CocktailSort
2000	28.3	47.4	0.6	65.7
4000	172.4	225.1	1.4	93.0
6000	368.9	359.6	4.1	257.1
8000	922.1	610.9	3.9	482.9
10000	1194.6	843.9	5.7	573.7
12000	1417.3	1514.6	8.5	1117.7
14000	2355.8	2293.4	19.1	1436.2
16000	2722.6	2108.1	7.3	1379.1
18000	3147.5	2788.4	7.8	1812.7
20000	3716.0	3490.9	10.1	2292.3

Análisis del comportamiento temporal.

▼ Caso Mejor

- En el caso mejor, donde el array ya está ordenado, se observa que los algoritmos de ordenación **Bubble Sort** y **Cocktail Sort** son los más eficientes en términos de tiempo.
- **Bubble Sort** y **Cocktail Sort** tienen tiempos muy bajos en comparación con los otros algoritmos en este caso. Y por lo que se observa el tiempo de ordenación es inversamente proporcional al tamaño del array, es decir, el tiempo de ordenación se achica cuanto mayor es el tamaño.
- **Selection Sort** muestra tiempos más altos en comparación con los otros algoritmos. El tiempo de ordenación aumenta conforme aumenta el tamaño del array.
- **Quick Sort** también muestra tiempos bajos, aunque ligeramente más altos que **Bubble Sort** y **Cocktail Sort**.
- En general, los tiempos son relativamente bajos y aumentan gradualmente a medida que aumenta el tamaño del arreglo, pero los algoritmos más eficientes (**Bubble Sort, Cocktail Sort** y **Quick Sort** respectivamente) mantienen sus ventajas.

https://grid.is/@jloen/caso-mejor-aSZarToDRZ29pD7WdPEeTw

▼ Caso Peor

- En el caso peor, donde el arreglo está invertido, se observa que los tiempos de ejecución de los algoritmos son significativamente más altos en comparación con el caso mejor.
- **Bubble Sort** y **Selection Sort** tienen tiempos muy altos. **Selection Sort** muestra el peor rendimiento entre los cuatro algoritmos con valores gradualmente mayores que en el caso mejor.
- Los tiempos de ejecución aumentan drásticamente a medida que aumenta el tamaño del arreglo, con **Quick Sort** muestra la menor tasa de crecimiento entre los algoritmos. Sigue siendo rápido. Seguido de él tenemos el **Cocktail Sort, Bubble Sort** y por último **Selection Sort**(el más lento en este caso).

https://grid.is/@jloen/caso-peor-4OlBlWlpQxCcWccnI_EzbA

▼ Caso Aleatorio

- En el caso aleatorio o medio, donde los datos de entrada son aleatorios, se observa un comportamiento intermedio en términos de tiempos de ejecución.
- **Quick Sort** sigue siendo el algoritmo más eficiente en este caso, seguido de **Selection Sort**.
- En este orden, tanto el **Bubble Sort** como el **Selection Sort** y **Cocktail Sort** muestran un rendimiento muy bajo especialmente a medida que aumenta el tamaño del arreglo.
- Los tiempos de ejecución de todos los algoritmos aumentan de manera significativa a medida que aumenta el tamaño del array, con **Quick Sort**

mostrando la menor tasa de crecimiento.

- **Bubble Sort y Selection Sort** muestran un comportamiento similar en términos de tiempos de ejecución en este caso, aunque **Bubble Sort** tiende a ser un poco más lento.

> https://grid.is/@jloen/caso-aleatorio-YmwKctI7Q3GSG6u:OlJmiQ

Conclusión

Al finalizar el análisis temporal del coste de ejecución de los diferentes algoritmos de ordenación mencionados he podido llegar a la conclusión de que el **Quick Sort** muestra consistentemente el mejor rendimiento en todos los casos. El **Bubble Sort** en comparación a los demás algoritmos tiene un rendimiento pésimo tanto en el caso peor como en aleatorio seguido del **Selection Sort** y **Cocktail Sort** respectivamente, mientras que en el mejor caso tiene un rendimiento similar al **Quick Sort** y **Cocktail Sort**.

Selection Sort en el mejor caso es el más lento de todos.

Herramientas

- Gráficas: grid
- Diagrama: excalidraw
- Code Snapshot: carbon.now.sh