Juan Carlos Alcausa Luque

Memoria Práctica 1 ADA

1. Descripción del problema

Implementar un programa Analizador.java capaz de determinar de forma automática la complejidad experimental de la función f(long n) de distintas clases denominadas Algoritmo.class que se proporcionan ya compiladas. A priori se sabe que la complejidad de las funciones es siempre una de las ocho que se muestran en la siguiente tabla. La ejecución del programa java Analizador dará como resultado una de las siguientes etiquetas:

Complejidad	Salida de `java Analizador`
$\Theta(1)$	**1**
$\Theta(log(n))$	**LOGN**
$\Theta(n)$	**N**
$\Theta(nlog(n))$	**NLOGN**
$\Theta(n^2)$	**N2**
$\Theta(n^3)$	**N3**
$\Theta(2^n)$	**2N**
$\Theta(n!)$	**NF**

Para la realización de esta práctica se ha seguido esencialmente el primer enfoque de los que se muestran en el campus virtual. Aquí se muestra el código de Analizador.class y posteriormente se analiza parte por parte:

2. Código implementado

```
public class Analizador {

	private static Temporizador temporizador = new
Temporizador();

	public static void main(String[] args) {
		int [] n1 = {10, 12, 15};
		int [] n2 = {20, 24, 30};
		double ratio = ratio(n1,n2);
		if(ratio>1000) {
			if(ratio<35000) {
				System.out.println("2N");
			}else {
				System.out.println("NF");
			}
		}else {
			int [] m1= {10000,12000,15000};
			int [] m2= {20000,22000,30000};
			ratio = ratio(m1,m2);
			if(6.0<=ratio && ratio<=10.0) {
				System.out.println("N3");
			}else if(2.3<=ratio && ratio<=6) {
				System.out.println("N2");
```

```
                }else if (1.6<ratio && ratio<2.3) {
                        System.out.println("NLOGN");
                }else{
                        int [] p1 = {1000, 2000, 3000};
                        int [] p2 = {1000000, 2000000, 3000000};
                        ratio= ratio(p1,p2);
                        if(ratio>100) {
                                System.out.println("N");
                        }else if(ratio<1 || (ratio>1 &&
ratio<1.05)) {
                                System.out.println("LOGN");
                        }else {
                                System.out.println("1");
                        }
                }
        }

    }
    private static double ratio (int [] n1, int[] n2) {
        long [] t1 = new long[n1.length];
        long [] t2 = new long [n1.length];

        double [] ratio = new double [n1.length];

        for (int nTam=0; nTam<n1.length; ++nTam) {
            temporizador.reiniciar();
            temporizador.iniciar();
            Algoritmo.f(n1[nTam]);
            temporizador.parar();
            t1[nTam] = temporizador.tiempoPasado();

            temporizador.reiniciar();
            temporizador.iniciar();
            Algoritmo.f(n2[nTam]);
            temporizador.parar();
            t2[nTam] = temporizador.tiempoPasado();
            ratio[nTam] = t2[nTam]/t1[nTam];
        }
        return mediaRatio(ratio);
    }
    private static double mediaRatio (double [] ratio) {
        double ratioMedio
        for (int i=0; i<ratio.length; ++i) {
            ratioMedio+=ratio[i];
        }
        ratioMedio= ratioMedio/ratio.length;
        return ratioMedio;
    }
}
```

3. Decisiones de diseño

La idea principal de mi enfoque ha sido calcular en primer lugar y de forma manual las ratios que producen teóricamente las funciones a estudiar cuando duplicábamos su entrada, separando cada caso en función de la entrada más adecuada. Como en nuestro caso no se trata de una función teórica sino de una medición de tiempo estos valores no son exactos y se ha tomado un intervalo cercano a los valores obtenidos.

Para reducir los posibles errores en las mediciones de tiempo, se han tomado además distintas entradas con distintos tamaños y se ha realizado la media de todas ellas a través del método auxiliar mediaRatio.

En primer lugar, se utilizan las entradas {10, 12, 15} para distinguir entre la complejidad exponencial y la factorial. Conocemos que $2^n \in O(n!)$ y por tanto podemos así determinar cual de las dos se trata.

A continuación, en caso de no tratarse de ninguna de las anteriores, se toman nuevas entradas, {10000,12000,15000}, más adecuadas para los otros órdenes de complejidad y que nos permiten distinguir entre n^3, n^2 y nlog(n). De nuevo, para determinar cuál de ellos se trata se estudian las ratios que generarían teóricamente al duplicar dichas entradas y se toma un intervalo centrado en esos puntos como referencia, teniendo en cuenta los posibles errores en los tiempos producidos por otras tareas que se están realizando en ese momento.

Por último, si tampoco corresponde el algoritmo con ninguno de estos órdenes de complejidad se toman de nuevo otras entradas, {1000, 2000, 3000}, que en este caso no se duplicarán, sino que se multiplicarán por 1000 para así obtener una mayor precisión. En este caso se han determinado las ratios tanto teórica como experimentalmente, probando con funciones de la complejidad indicada y observando los tiempos de ejecución para determinar las ratios.

En cuanto a la clase *ratio*, toma dos arrays de enteros con las entradas, el original y el que ha sido multiplicado. A continuación, se mide el tiempo de ejecución de cada uno de los elementos de los arrays y se incorporan las ratios que generan a otro array. Finalmente, se realiza la media de dichas ratios mediante la clase auxiliar *mediaRatio*.

4. Análisis de la complejidad de *Analizador*

Consideremos como operación elemental del algoritmo Analizador a cada llamada a cualquier método de la clase Temporizador. Tomaremos como entrada dos array de tamaño n. Distinguiremos entre mejor caso, peor caso y caso medio.

El mejor caso sería que el algoritmo a analizar fuese 2n o n! en cuyo caso la complejidad sería **8n**, ya que se invoca el método *ratio* una única vez y se ejecuta el bucle for n veces con 8 llamadas a métodos de *Temporizador* en cada ejecución.

El caso medio sería si la complejidad fuera n^3, n^2 o nlogn. En este caso la complejidad sería 8n + 8n= **16n** ya que se llama a la función ratio en dos ocasiones, una por cada array de valores. La función ratio ejecuta un bucle for n veces con 8 llamadas a métodos de *Temporizador*.

El peor caso se daría si la complejidad a analizar fuese n, logn o 1. El método ratio se invocaría en tres ocasiones por lo que la complejidad sería 8n + 8n + 8n = **24n.**