Práctica Evaluable 1

Análisis y Diseño de Algoritmos

Nombre: Emilio Gómez Esteban

Grupo: D (Grado Matemáticas + Ingeniería Informática)

CÓDIGO EMPLEADO:

```
public class Analizador {

        private static int numPruebas = 10;
        private static Temporizador t = new Temporizador();

     public static void main(String[] args) {
        long n1 = 10;
        long n2 = 15;
        double ratio = mediaDePruebas(n2)/mediaDePruebas(n1);

        if (ratio>1000){
             if (ratio<3000000){
                 System.out.println("2N");
             } else{
                 System.out.println("NF");
             }
        } else {
             n1 = 10000;
             n2 = 20000;
             ratio = mediaDePruebas(n2)/mediaDePruebas(n1);

             if (6 <= ratio && ratio < 10.0){
                 System.out.println("N3");
             } else if (3.2 <= ratio && ratio < 6.0){
                 System.out.println("N2");
             } else if (1.8<=ratio &&ratio<3.2){
                 System.out.println("NLOGN");
             } else {
                 n1 =1000;
                 n2 =1000000;

                 ratio=mediaDePruebas(n2)/mediaDePruebas(n1);

                 if (700<=ratio&& ratio<1300){
                     System.out.println("N");
                 } else if (ratio<=1.005) {
                     System.out.println("1");
                 }else {
                     System.out.println("LOGN");
```

```
                }
            }
        }
    }

    private static double mediaDePruebas(long n) {
        int mediaTiempos=0;

        for (int i = 0 ; i<numPruebas ; i++){
            t.reiniciar();
            t.iniciar();
            Algoritmo.f(n);
            t.parar();
            mediaTiempos += t.tiempoPasado();

        }
        return mediaTiempos/(double)numPruebas;
    }
}
```

ESTRATEGIA EMPLEADA:

Para tratar de crear un programa que midiese la complejidad de cualquier algoritmo dado existen varias formas de comenzar. En primer lugar, para medir experimentalmente la complejidad de un algoritmo implementado, se usará la clase llamada Algoritmo.class, que contiene la implementación de ese algoritmo mediante el método:

```
public class Algoritmo {
      public static synchronize void f(long n) {
      ...
      }
}
```

en el cual el argumento n es la variable de la que depende el tiempo de ejecución. Además, se echará mano de otra clase denominada Temporizador.java, para medir el tiempo que tarda en ejecutarse un método en Java. Esta clase se encarga de obtener la fecha antes y después de la ejecución de la función y hallar la diferencia. Se observa que, al repetir varias veces la misma prueba, se obtienen resultados diferentes. Para que este problema no tenga mucho impacto en los cálculos, se hará lo siguiente: hallar la media de los tiempos de ejecución para cada valor de n.

Aunque más tarde se tratará de implementar un programa más general, inicialmente, se seguirá una estrategia, para ver como plasmar los resultados y llegar a una solución:

- Ejecutar el mismo algoritmo *f*, cuya complejidad queremos calcular dos veces, una primera para un tamaño de entrada n y la segunda para un tamaño de entrada 2n (además, dependiendo de la especialidad del caso que queramos estudiar, podremos usar tamaños de entrada del cuádruple, potencias de n, etc,). Como se ha adelantado anteriormente, calcularemos la media de tiempos de ejecución para tamaños de entrada n en 10 pruebas del algoritmo. (Para ello se usa el método privado

mediaDePruebas). A continuación, dividiremos la media de los tiempos del algoritmo con entrada 2n entre la media de los tiempos del algoritmo con entrada n. Con esto hallaremos un ratio, que para los tipos de complejidad polinómicas, se estabilizará entorno a una constante cercana (podrá oscilar por izquierda y por derecha de ese punto). Se puede comenzar con este código:

```
public static void main(String arg[]) {
	int n1 = 10000;
	int n2 = 20000;
	Temporizador t = new Temporizador();
	t.iniciar();
	Algoritmo.f(n1);
	t.parar();
	long t1 = t.tiempoPasado();
	t.reiniciar();
	t.iniciar();
	Algoritmo.f(n2);
	t.parar();
	long t2 = t.tiempoPasado();
	double ratio = (double)t2/t1;
	System.out.println(masCercano(ratio));
}
```

Durante las pruebas, se ha visto como las complejidades de tipo polinómicas son fáciles de tratar. El ratio calculado para incluir estas funciones en un intervalo es claro y no requiere de muchas pruebas. Sin embargo, a la hora de ver otro tipo de funciones no se ve de esta forma. Para ello, una de las soluciones que se buscan es dividir el programa en subprogramas dependiendo de la tratabilidad de las funciones. Como se ve al principio del código final, para los algoritmos, de los cuales queramos comprobar su tiempo, que tengan unos tiempos de ejecución del orden de (2^n) ó del orden de (n!), creamos un caso aparte de todos los demás. Esto se debe a que cuando se quiere comparar su tiempo de ejecución no podemos darle valores demasiado altos, lo que supondría un problema para la finalización de la ejecución. Es suficiente con crear un tamaño de entrada n y otro cuyo valor sea n+c, donde c es una constante "pequeña". En esos casos el cociente entre funciones del orden de (2^n) tiende a estabilizarse, sin embargo, el de (n!) tiende a infinito.

A continuación, para algoritmos bastante eficientes, hay veces que los ratios podían coincidir y es complicado clasificarlos mientras se tomen los mismos tamaños de entrada (creamos un caso que los reúna a todos y más tarde lo subdividiremos de nuevo). Es el caso de (n) y (nlongn). No obstante, para algoritmos que tienen tiempos de ejecución constantes (marcados en las pruebas con "1"), es sencillo ver que, para cualquier tamaño de entrada, la salida no podría estar muy lejos de 1, ya que al calcular los ratios, el tiempo no cambiará. Por tanto, resolvemos el problema sin complicaciones, y lo colocamos también aparte de las polinómicas, para evitar que los ratios se puedan "pisar".

Volviendo al caso comentado anteriormente, toca resolver el dilema para las funciones de tipo (n) y (nlogn). Aunque haciendo pruebas no se ha resuelto con el 100% de efectividad, la idea es la siguiente:

Los ratios de dos funciones con el mismo tipo de estos órdenes, tienden a acercarse a 2, cuando un tamaño de entrada es n y el siguiente es 2n. Haciendo cálculos siempre es más engorroso trabajar con logaritmos, por tanto, para intentar facilitar el estudio, se intenta crear un caso para las polinómicas (n^2,que se acerca a 4) y (n^3, que se acerca a 8), y en vez de incluir las de tipo (n), incluimos las de tipo (nlogn). Así sale el intervalo que se observa en el código. Más tarde, meteremos en el caso donde dejábamos las funciones constantes, a las lineales, y en este caso, los tamaños de entrada no tienen que coincidir con los anteriores, pues la función constante no supone un problema frente a esto. De esta forma para tamaños de entrada grandes, el límite del cociente de funciones lineales tenderá al puro cociente entre ellas (que podrá oscilar por izquierda y por derecha).

Por último, nos quedaría incluir las funciones logarítmicas. Para ello, se puede echar mano de la segunda estrategia posible e ir calculando tiempos de ejecución de orden logarítmicos y comparando con otras funciones. La cuestión es que este tipo de funciones tiene un crecimiento “lento”, es decir, que el ratio entre log de valores grandes es bajo. Para no buscar un intervalo de manera engorrosa, la solución que se ha intentado usar es la de dejarla como último caso posible e incluirla en el subprograma junto a las constantes y a las lineales, para evitar confusión.

Para terminar, es cierto, que los intervalos podían no estar del todo bien definidos, así que se ha intentado ir retocando y refinando, para que las oscilaciones que pueda haber estén recogidas en todos los casos. Con esto y con todo se trata de ir haciendo pruebas con nuestro código e ir cambiando nuestro programa hasta ver buenos resultados a la hora de clasificar un algoritmo.