# Práctica 1. Analizador

**Autor:** Marcos Hidalgo Baños **Fecha:** 7 de octubre del 2022

## Descripción del problema.

El objetivo de esta práctica es la implementación de un programa que llamaremos **Analizador** en el lenguaje de programación JAVA que sea capaz de clasificar el orden de complejidad temporal de un algoritmo cualquiera proporcionado mediante una clase **Algoritmo**.

Como herramienta auxiliar, se nos proporciona una tercera clase **Temporizador** con la implementación de un cronómetro para realizar las medidas del tiempo transcurrido durante la ejecución del algoritmo en cuestión. Cabe destacar que para cumplir dicho cometido nos bastará con calcular la diferencia entre el tiempo final e inicial después de ejecutar la función.

## Código implementado y decisiones de diseño.

La clase **Analizador.java** propuesta como solución está compuesta por una serie de funciones auxiliares a la principal (main) que permitirán clasificar el algoritmo a estudiar en alguna de las clases de complejidad conocidas en base a la similitud de sus tiempos de ejecución.

Las principales características que diferencian mi solución de otras posibles son:

- Es una **mezcla de las dos estrategias** propuestas, ya que incluye la idea de realizar dos tipos de estudios según el tamaño de entrada (n y 2n) además de comparar los resultados obtenidos con otras funciones de complejidad temporal conocida.
- La toma de datos se realiza un número de veces fijo (variable 'iteraciones' en el código), obtenido de forma heurística por cada tamaño de entrada, pero este es un proceso que se repetirá todas las veces posibles mientras que no se agote el tiempo disponible.

## Funciones Auxiliares.

El siguiente conjunto de funciones representan a los diferentes órdenes de complejidad. Esto se consigue mediante la implementación de una operación conocida, la cual sabemos pertenece exactamente a dicha complejidad. La utilidad de estas funciones en nuestro programa será tangible a la hora de determinar si pertenece a una clase u otra, puesto que el tiempo de ejecución del algoritmo deberá ser similar al de una de estas funciones.

```java
public static void N (long n) {
    int j = 0;
    for (int i = 0; i < n; i++) {
        j=i;
    }
}

public static void N2 (long n) {
    int j = 0;
    for (int i = 0; i < n*n; i++) {
        j=i;
    }
}

public static void N3 (long n) {
    int j = 0;
    for (int i = 0; i < n*n*n; i++) {
        j=i;
    }
}

public static int fact (long n) {
    int f = 1;
    for (int i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}

public static void FN(long n) {
    int j=0;
    for (int i = 0; i<fact(n);i++) {
        j=i;
    }
}
```

```java
public static void _1(long n) {
    int j=0;
}

public static long log2 (long n) {
    double num = Math.log(n);
    double den = Math.log(2);
    double div = num/den;
    return ((long)div);

}

public static void LOGN (long n) {
    int j=0;
    for (int i = 0; i < log2(n); i++) {
        j = i;
    }
}

public static void NLOGN (long n) {
    int j = 0;

    for (int i = 0; i < n * log2(n); i++) {
        j=1;
    }
}

public static void _2N (long n) {
    int j = 0;
    for (int i = 0; i < Math.pow(2, n); i++) {
        j=i;
    }
}
```

## Función masCercano.

A partir del **ratio** calculado en el programa principal, este método nos permite distinguir cuál es la clase de complejidad que más se parece a la del algoritmo.

Notar que las funciones auxiliares descritas anteriormente son llamadas desde este método.

Su estructura general es la de un **condicional** con un total de cinco ramas, las cuales agrupan a las principales clases de complejidad según el ratio.

La similitud de este valor entre las distintas clases hacen que, en la mayoría de casos no seamos capaces de distinguir exactamente a cuál pertenece, por lo que deberemos realizar un análisis más exhaustivo.

```java
public static String masCercano(double ratio) {
    Temporizador temporizador = new Temporizador(2);
    int itConstanteLog = 10;
    int itLinearNlogn = 10;
    int itExpFact = 5;

    if (ratio < 1.5) { // Constante o logaritmico
        long n = 200000;
        long tn = 0, t1 = 0, tlogn = 0;

        for (int i = 0; i < itConstanteLog; i++) {
            // Probamos con el algoritmo
            temporizador.reiniciar();
            temporizador.iniciar();
            Algoritmo.f(n);
            temporizador.parar();
            tn += temporizador.tiempoPasado();

            // Probamos con uno constante
            temporizador.reiniciar();
            temporizador.iniciar();
            _1(n);
            temporizador.parar();
            t1 += temporizador.tiempoPasado();

            // Probamos con uno logaritmico
            temporizador.reiniciar();
            temporizador.iniciar();
            LOGN(n);
            temporizador.parar();
            tlogn += temporizador.tiempoPasado();

            // Incrementamos el tamaño de entrada
            n *= 10;

        }
        tn /= itConstanteLog; // Nos quedamos con la media
        t1 /= itConstanteLog; // Nos quedamos con la media
        tlogn /= itConstanteLog; // Nos quedamos con la media

        if (((double)tn)/tlogn < 0.1) return "1";
        else return "LOGN";
```

```
    } else if (ratio < 3.0) { // Linear o NlogN
        long n = 100000;
        long tn = 0, tnlogn = 0;

        for (int i = 0; i < itLinearNlogn; i++) {
            // Probamos con el algoritmo
            temporizador.reiniciar();
            temporizador.iniciar();
            Algoritmo.f(n);
            temporizador.parar();
            tn += temporizador.tiempoPasado();

            // Probamos con uno NLOGN
            temporizador.reiniciar();
            temporizador.iniciar();
            NLOGN(n);
            temporizador.parar();
            tnlogn += temporizador.tiempoPasado();

            // Incrementamos el tamaño de entrada
            n *= 110;
            n /= 100;
        }
        tn /= itLinearNlogn; // Nos quedamos con la media
        tnlogn /= itLinearNlogn; // Nos quedamos con la media

        if (((double)tn)/tnlogn < 0.1) return "N";
        else return "NLOGN";

    } else if (ratio < 6.0) { // Cuadratico
        return "N2";

    } else if (ratio < 10.0) { // Cubico
        return "N3";

    }  else {   // Exponencial o factorial
        long tn = 0, tf = 0;

        for (int i = 0; i < itExpFact; i++) {
            // Probamos con el algoritmo
            temporizador.reiniciar();
            temporizador.iniciar();
            Algoritmo.f(10);
            temporizador.parar();
            tn += temporizador.tiempoPasado();

            // Probamos con uno factorial
            temporizador.reiniciar();
            temporizador.iniciar();
            FN(10);
            temporizador.parar();
            tf += temporizador.tiempoPasado();
        }
        tn /= itExpFact;
        tf /= itExpFact;

        if (((double)tn) / ((double) tf) < 0.1) return "2N";
        else return "NF";
    }
}
```

(continuación)

Aunque en casos como las complejidades cuadrática o cúbica no sea necesario, en el resto de casos deberemos realizar una comparativa entre el tiempo de ejecución del algoritmo y alguno de los candidatos de la rama del condicional.

## NOTA

La decisión final se toma en base a una división real entre dichos tiempos de ejecución.

El valor 0.1 ha sido tomado de forma heurística, siendo lo suficientemente pequeño como para distinguir que el tiempo de ejecución del denominador es mayor que la del numerador.

## Programa principal.

Una vez analizados los métodos auxiliares (nunca mejor dicho) nos encontramos en posición de estudiar el Main de la clase Analizador. En primer lugar, nos encontramos con aquellas variables que nos servirán a lo largo del programa, entre las que destacan variables de control del tiempo y del tamaño de entrada, además de los correspondientes temporizadores.

```
public class Analizador {

    public static void main(String arg[]) {

        /*
         * VARIABLES DE CONTROL DEL TIEMPO
         */

        // Maximo de segundos en ejecucion del main
        int tiempoTotal = 8;
        // Total de milisegundos transcurridos
        long tiempoEjecucion = 0;
        // Numero estimado de iteraciones (puede ser mayor si da tiempo)
        int maxSimulaciones = 50;
        // Numero de veces que vamos a probar el algoritmo con entrada n o n2
        int iteraciones = 5;


        /*
         * VARIABLES DE CONTROL DEL TAMAÑO DE ENTRADA
         */

        // Tamaños de entrada al algoritmo (n2 es el doble de n)
        long n = 2, n2 = 2;
        // Tiempo transcurrido para cierto tamaño de entrada
        long tn, tn2;
        // Arrays de tiempos de ejecucion para entrada n y n2
        ArrayList<Long>tiemposN = new ArrayList<Long>();
        ArrayList<Long>tiemposN2 = new ArrayList<Long>();

        // Cronometro general, para tamaño de entrada n y n2
        Temporizador crono = new Temporizador(1); //Cronometro en milisegs
        Temporizador temporizador= new Temporizador(2); //Cronometro en nanosegs
        Temporizador temporizador2 = new Temporizador(2); //Cronometro en nanosegs
```

```
    while (tiempoEjecucion/1000 < tiempoTotal) {

        // Aumentamos el tamaño de entrada segun la iteracion
        if (tiemposN.size() < maxSimulaciones) {n += 1;}
        else {n *= 10; maxSimulaciones *= 2;}
        // n2 siempre es el doble de n
        n2 = n*2;

        /*
         * CALCULO DEL TIEMPO EMPLEADO PARA ENTRADA N
         */

        tn = 0;
        for (int i = 0; i < iteraciones; i++) {
            crono.iniciar();

            temporizador.iniciar();
            Algoritmo.f(n);
            temporizador.parar();
            tn += temporizador.tiempoPasado();
            temporizador.reiniciar();

            crono.parar();
            tiempoEjecucion = crono.tiempoPasado();
            if (tiempoEjecucion/1000 > tiempoTotal) break;
        }
        tn /= iteraciones; // Nos quedamos con la media
        tiemposN.add(tn);

        /*
         * CALCULO DEL TIEMPO EMPLEADO PARA ENTRADA N2
         */

        tn2 = 0;
        for (int i = 0; i < iteraciones; i++) {
            crono.iniciar();

            temporizador2.iniciar();
            Algoritmo.f(n2);
            temporizador2.parar();
            tn2 += temporizador2.tiempoPasado();
            temporizador2.reiniciar();

            crono.parar();
            tiempoEjecucion = crono.tiempoPasado();
            if (tiempoEjecucion/1000 > tiempoTotal) break;
        }
        tn2 /= iteraciones; // Nos quedamos con la media
        tiemposN2.add(tn2);

        tiempoEjecucion = crono.tiempoPasado();

    }

    double ratio = ((double) tiemposN2.get(tiemposN2.size()-1))
                    / ((double) tiemposN.get(tiemposN.size()-1));
    System.out.println(masCercano(ratio));
}
```

(continuación)

El programa general se compone de un bucle 'while' que comprueba que no hayamos excedido el tiempo total.

Una vez hecha la comprobación, deberemos realizar alguna operación para aumentar el tamaño de entrada. Una forma de hacerlo es aumentar de uno en uno hasta llegar a un determinado número de iteraciones, para después avanzar a pasos cada vez más agigantados.

A continuación estudiaremos el algoritmo con el tamaño de entrada estipulado n y con el doble 2n.

El ratio será obtenido una vez rellenado el array de tiempos mientras que no se sobrepase el límite.

## Análisis de la complejidad del programa.

Debido a la naturaleza del problema, en la que el número de veces que el programa puede ser ejecutado varía en función de lo complejo que sea el algoritmo a estudiar, no podremos dar una expresión exacta para la complejidad del programa que no contenga un parámetro.

Si tenemos en cuenta como operación principal aquellos bucles (normalmente bucles for) empleados para obtener los valores de tiempo de ejecución, bien sean los del algoritmo a estudiar o de las funciones auxiliares descritas anteriormente, en el peor de los casos:

- Realizaríamos un **número $\alpha$ de iteraciones** del bucle while, mientras que no sobrepasemos el límite de tiempo total.
- Para el cálculo del tiempo empleado por el algoritmo para la entrada de tamaño n y n2, deberemos parar completamente por **dos bucles for** con 'iteraciones' número de iteraciones (cinco en el caso de la solución propuesta).
- La función masCercano se ejecutará una única vez, pero en el peor de los casos deberá distinguir entre dos clases de complejidad similares. Para ello, se deberá ejecutar **otro bucle for** que compare los tiempos de ejecución un número de veces determinado.

Notar que si el algoritmo es complejo, este proceso llevará más tiempo para ser realizado.

En conclusión, el programa en el peor de los casos realizará un total de $\alpha$ *** 3 bucles for** de comparación de tiempos de ejecución, que a su vez contendrán las operaciones intrínsecas del algoritmo / función auxiliar para el tamaño de entrada correspondiente.

Enlace al repositorio de Github con el código fuente.

github.com/MarkosHB/Analisis-y-diseno-de-Algoritmos/blob/main/Analizador/Analizador.java

- Marcos Hidalgo Baños -