ESTRUCTURA DE DATOS

PROYECTO 1

JONATHAN ARIAS BUSTO

UO283586

ESCUELA INGENIERIA INFORMATICA

29/09/2021

> GRAFICOS A

La primera parte de gráficos viene formada por la representación de las siguientes complejidades: lineal, cuadrática, cubica y logarítmica.

Antes de nada, para poder ejecutar los métodos que representen estas complejidades necesitamos un método que nos pare el hilo principal de ejecución durante unos milisegundos, siendo el método:

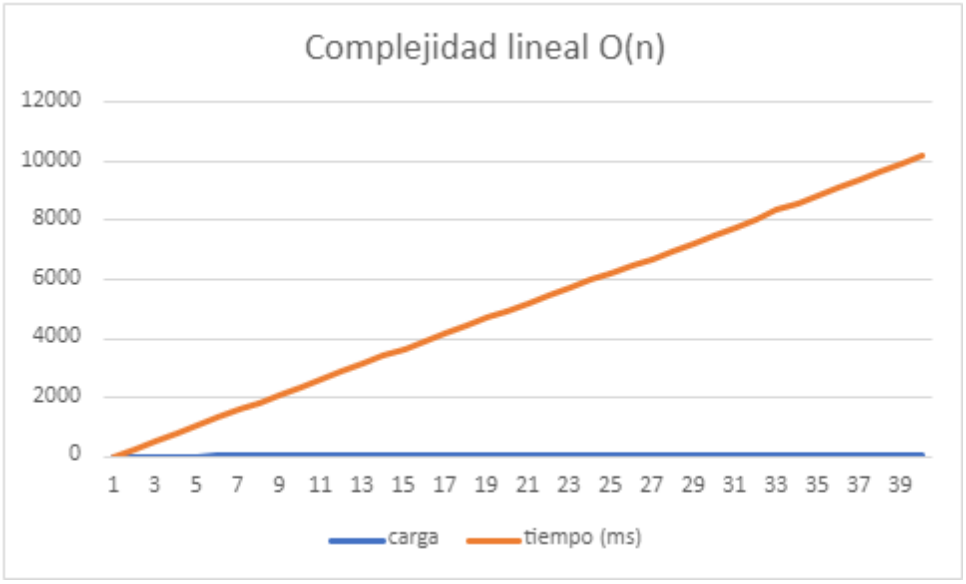
```java
/**
 * Metodo que para el hilo principal de ejecucion 250 ms para poder hacer calculo
 * de tiempos en diferentes algoritmos
 */
private static void doNothing() {
    try {
        Thread.sleep(250);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Ahora que tenemos este método que nos permite parar la ejecución en el hilo principal se puede hacer métodos para representar complejidades de forma muy sencilla.

Comenzaremos con la complejidad lineal que es la más sencilla al tener un simple bucle for con carga desde 0 hasta n, siendo en este caso n = 40.

```java
/**
 * Metodo que representa un algoritmo de complejidad lineal O(n)
 *
 * @param n Carga del metodo
 */
public static void linealDoNothing(int n) {
        for (int i=0;i<n;i++) {
            doNothing();
        }
}
```

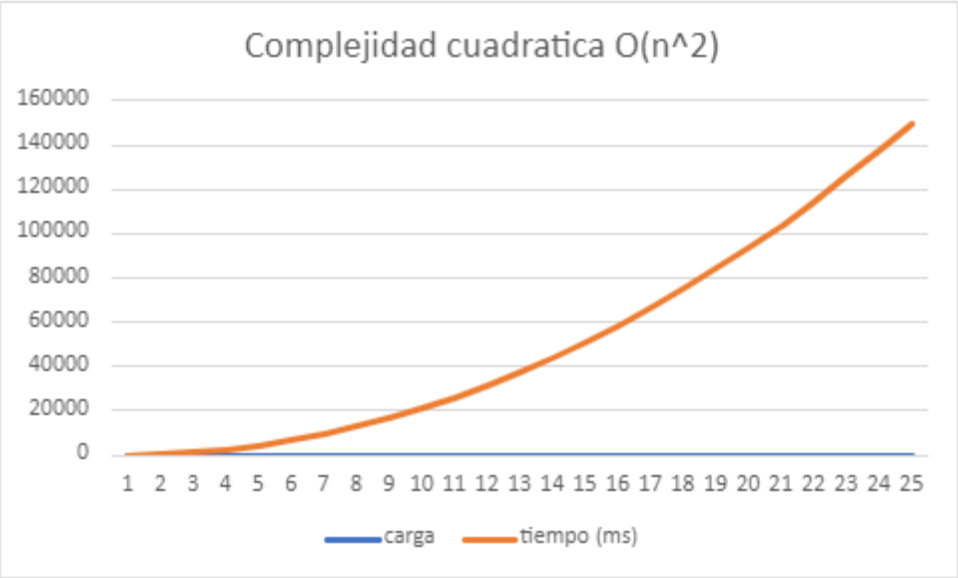
La grafica después de ejecutar este método queda tal que así:



La siguiente complejidad será la cuadrática, que es similar al caso anterior en cuanto a código solo que se anida un bucle for dentro de otro, quedando el código tal que así:

```java
/**
 * Metodo que cenresenta un algoritmo de complejidad cuadcatica O(n^2)
 *
 * @param n Carga del metodo
 */
public static void cuadraticaDoNothing(int n) {
    for (int i=0;i<n;i++) {
        for (int j=0; j<n;j++) {
            doNothing();
        }
    }
}
```

Ejecutando este código con una carga de 0 a 25 nos quedara una gráfica como la siguiente:



Se puede apreciar como la tendencia de la curva ha cambiado a una curva y no a una recta con pendiente constante. Esto se debe al cambio de complejidad a O(n^2) por lo que se tarda más en ejecutar el código para una misma carga.

El siguiente caso es la complejidad cúbica que siguiendo la tendencia de los anteriores casos se puede representar con 3 bucles for anidados, siendo el código el siguiente:

```java
/**
 * Metodo que representa un algoritmo de complejidad cubica O(n^3)
 *
 * @param n Carga del metodo
 */
public static void cubicaDoNothing(int n) {
    for (int i=0;i<n;i++) {
        for (int j=0;j<n;j++) {
            for (int k=0;k<n;k++) {
                doNothing();
            }
        }
    }
}
```

Al ejecutar este metodo con una carga de 0 a 10 la gráfica queda así:

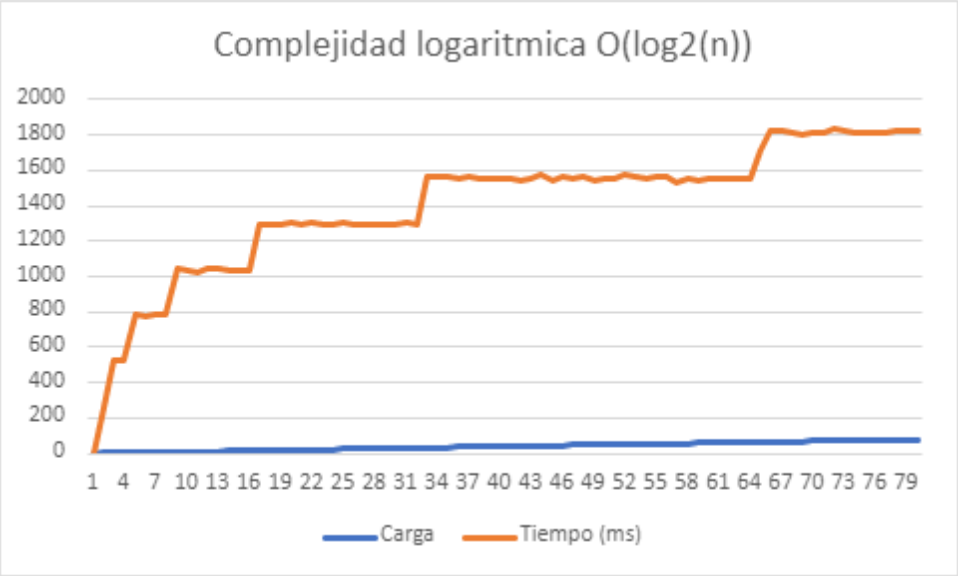
Se puede apreciar que la tendencia de la curva cada vez es más exponencial, por lo que el tiempo de ejecución es aún mayor que en el caso anterior. Pudiendo comparar el tiempo de ejecución para una carga fija como puede ser el caso 9:

- Para la complejidad cuadrática: 21109 ms
- Para la complejidad cúbica: 120000ms

La siguiente y ultima complejidad que trataremos es la logarítmica que viene dada por el siguiente código:

```java
/**
 * Metodo que representa un algoritmo de complejidad logaritmica O(log2 n)
 *
 * @param n Carga del metodo
 */
public static void logaritmicaDoNothing(int n) {
    int i = n;
    while (i > 0) {
        doNothing();
        i = i/2;
    }
}
```

Este método es bastante más eficiente que los dos anteriores casos, puesto que sigue la tendencia de una gráfica logarítmica. Ejecutando este método para una carga de 80 la gráfica queda así:



> GRAFICOS B

La segunda parte de los gráficos viene determinada por los siguientes métodos: potenciaRecursiva_v0doNothing(), potenciaRecursiva_v1doNothing(), potenciaRecursiva_v2doNothing() y potenciaRecursiva_v3doNothing().

Cada una de estos métodos tiene su propia complejidad y por ello vamos a representarlos gráficamente.

El primer caso de estos métodos viene dado por potenciaRecursiva_v0doNothing(), que tiene el siguiente código:

```
public static int potenciaRecursiva_v0doNothing(int n) {
    doNothing();
    if (n == 0) {
        return 1;
    }
    return potenciaRecursiva_v0doNothing(n-1)*2;
}
```

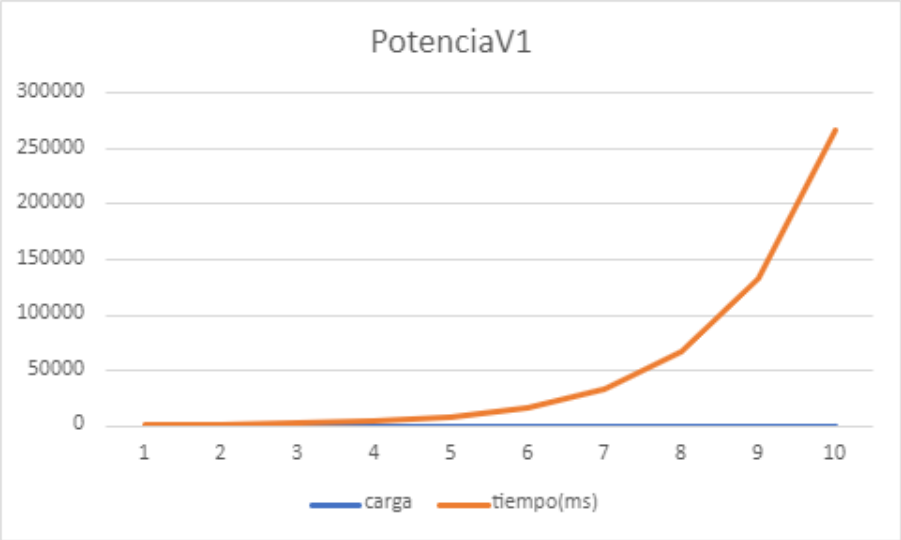
La grafica tras ejecutar este método para una carga de 0 a 50 es la siguiente:

Como se puede apreciar es una tendencia de complejidad lineal O(n).

En el segundo de los casos (potenciaRecursiva_v1doNothing()), tenemos el siguiente código:

```java
public static int potenciaRecursiva_v1doNothing(int n) {
    doNothing();
    if (n == 0) {
        return 1;
    }
    return (potenciaRecursiva_v1doNothing(n-1) + potenciaRecursiva_v1doNothing(n-1));
}
```

Tras ejecutar este método para una carga de 0 a 10 la gráfica queda de la siguiente manera:

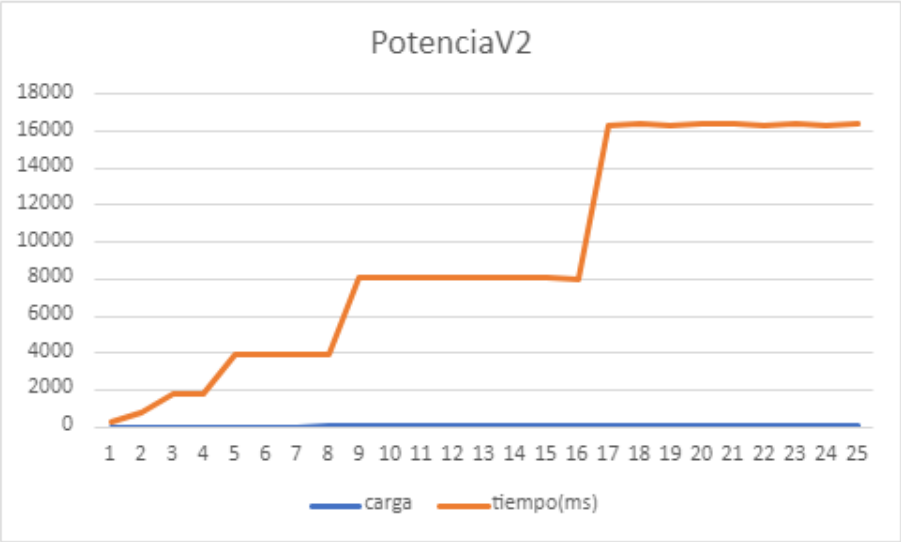


En este caso vemos que los tiempos de ejecución se asemejan al de una complejidad cubica O(n^3), por lo tanto, podemos afirmar que este método no es el más eficiente de todos.

El siguiente de los casos es el método potenciaRecursiva_v2doNothing(), que tiene como código:

```java
public static int potenciaRecursiva_v2doNothing(int n) {
    doNothing();
    if (n == 0) {
        return 1;
    }
    if (n % 2 == 0) {
        return (potenciaRecursiva_v2doNothing(n/2) * potenciaRecursiva_v2doNothing(n/2));
    } else {
        return (potenciaRecursiva_v2doNothing(n/2) * potenciaRecursiva_v2doNothing(n/2)*2);
    }
}
```

Ejecutamos este código con una carga de 0 a 25 y el resultado de la gráfica es el siguiente:

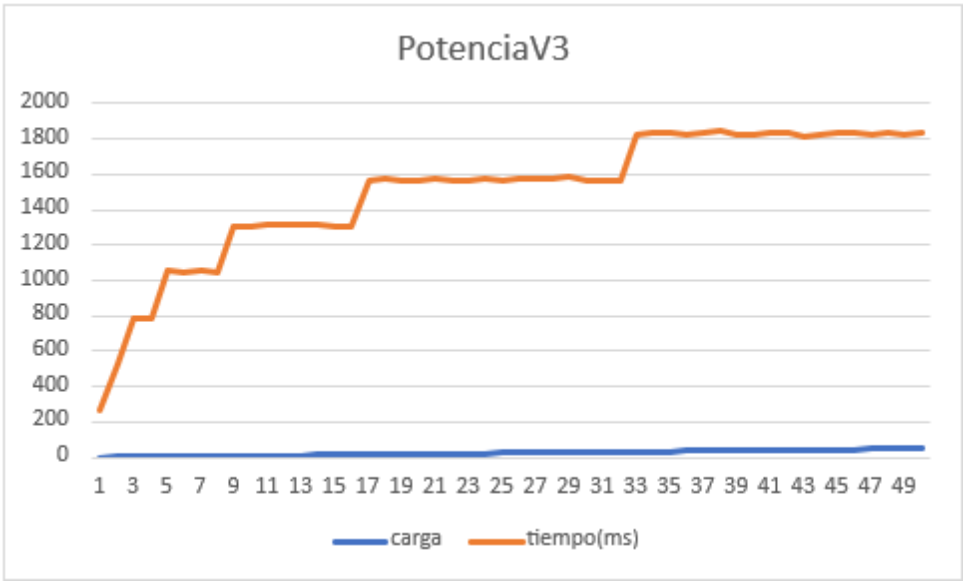


En este caso la tendencia está bastante clara, viendo como para cierta cantidad de carga el tiempo de ejecución es el mismo, pero sin embargo hay ciertas cargas que aumenta el tiempo exponencialmente para volver a tener esta tendencia constante.

El último caso de estos métodos es el potenciaRecursiva_v3doNothing(), que tiene el siguiente código:

```java
public static int potenciaRecursiva_v3doNothing(int n) {
    doNothing();
    if (n == 0) {
        return 1;
    }
    int result = potenciaRecursiva_v3doNothing(n/2);
    if (n % 2 == 0) {
        return (result*result);
    } else {
        return (result*result*2);
    }
}
```

El grafico de este método se realizó con una carga de 0 a 50, siendo el resultado el siguiente:



Este grafico se asemeja mucho a una complejidad de tipo logarítmica O(log^2 n). Se puede aprecia su tendencia asintótica en todo su trazo.

En resumen, podemos decir que la versión de potencia2 más eficiente es la versión 3 (potenciaRecursiva_v3doNothing) por tener una complejidad de tipo logarítmica.