Universidad Internacional de Valencia (VIU)

15GIIN – Estructuras de Datos y Algoritmos

Segundo Portafolio – ACT2

Alumno: Gagliardo Miguel Angel

Ejercicio 1

Un programa tarda 0,5 milisegundos para el tamaño de entrada 100. ¿Cuántas veces más tardará para un tamaño de entrada 1000 si el tiempo de ejecución es el siguiente?:

a) Lineal

Para N = 100 sabemos que tarda 0,5ms
Para N = 1000, usando regla de 3 simple:
((1000 / 100) * 0,5 ms) = **5ms; o bien, 10 veces más.**

b) O(N log N) (suponga el logaritmo en base 10). La respuesta puede ser una estimación

[log(1000) / log(100)] * 0,5ms = 1,5 * 0,5ms = **0,75ms; o bien 1,5 veces mas**

c) Cuadrático

Dado que **N = 100**
En una función cuadratica tendremos $N^2 = 100^2$ = **10.000 instrucciones** ejecutadas en **0,5ms**

Entonces para **N = 1000**. $N^2 = 1000^2$ = 1.000.000 instrucciones

El tiempo en total será entonces:
0
$(1000^2) / (100^2) * 0,5$ = **50ms; o bien, 100 veces más.**

d) Cúbico

Dado que **N = 100**
En una función cúbica tendremos $N^3 = 100^3$ = **1.000.000 instrucciones** ejecutadas en **0,5ms**

Entonces para **N = 1000**. $N^3 = 1000^3$ = **1.000.000.000 instrucciones**

El tiempo en total será entonces:

$(1000^3) / (100^3) * 0,5$ = **500ms o 0,5 segundos. Finalmente: 1000 veces más.**

Ejercicio 3

Dar la complejidad exacta en función de n (el grado del polinomio) del tiempo en peor caso de los métodos “**calculaHorner**”, “**calculaConPotencia**” y “**calculaConPotencia1**”. Debe en cada caso justificar detalladamente sus respuestas.

---

calculaHorner

```
1 private static double calculaHorner(double [ ] pol, double x, int n) {
2   /* Aplica la formula de Horner para evaluar el polinomio
3   pol de grado n en el valor de la variable x */
4   double resultado = 0;
5   for (int i = 0; i <= n; i++) {
6     resultado = (resultado * x) + pol[i];
7   }
8   return resultado;
9 }
```

Linea 4: 1 operacion elemental de inicializar resultado en 0
Linea 5: 1 (inicializar I en cero) + 2*N (comparacion de i con n y suma de 1)
Linea 6: 2 Operaciones: Multiplicacion y suma

El peor caso es cuando **i** es menor que **n**, dado que entraremos en el for loop y sumaremos i+1 hasta que sea igual a **n**.

En total: 4 + 2*N = O(N). O sea que la complejidad de la funcion calculaHorner es Lineal.

---

calculaConPotencia

```
private static double calculaConPotencia(double [ ] pol,
   double x, int n) {
 /* Aplica la formula del polinomio para evaluar
 pol de grado n en el valor de la variable x
 las potencias i-esimas de x se calculan multiplicando
  i veces x */

 int i=n; double suma=0.0;
    while (i >= 0) {
         suma += pol[n-i]*potencia(x, i);
         i--;
      }
     return suma;
 }

public static double potencia(double x,int i) {
 double resultado = 1.0;
 for(int j=0;j<i;j++)
   resultado*=x;
 return resultado;
}
```

Para el metodo **calculaConPotencia**, tenemos:

Linea 8: 2 operaciones elementales de inicializacion de **i** y de **suma**
Linea 9: 2*N Entro en el while loop (Comparacion de i con 0 y posterior resta de i-1)
Linea 10: 3 Operaciones: La suma de suma con el valor que devuelva pol[n-i] y la posterior multiplicacion con lo que devuelva la llamada a la funcion **potencia** (que analizaremos abajo).

El peor caso para esta funcion es cuando **i** es menor que **n**, dado que entraremos en el while loop y restaremos i-1 hasta que sea menor a 0.

Finalmente, solo para el while de calculaConPotencia (<u>obviando</u> la llamada a la funcion potencia): 5 + 2*N = O(N) o bien, una lineal.

Luego la funcion **potencia**, tenemos:

Linea 17: 1 Operacion elemental de inicializacion de resultado
Linea 18: 1 (inicializar j en cero) + 2*N (comparacion de j con i y suma de j+1)
Linea 19: 1 Operacion fundamental (multiplicacion de resultado con x)

Por tanto el peor caso es donde j sea menor que i, dado que entraremos en el for loop y sumaremos j+1 hasta que sea igual a i.

Finalmente, para el metodo potencia: 3 + 2*N = **O(N)**

Dado que la llamada a **potencia** se hace **desde** el while loop de **calculaConPotencia**, para el peor escenario siempre, seria identico lo mismo que poner loops uno dentro del otro, por tanto podemos decir que la **complejidad** de **calculaConPotencia es: O(N) * O(N) = $O(N^2)$ o bien, una cuadratica.**

<u>**calculaConPotencia1**</u>

```
 1  private static double calculaConPotencia1(double [ ] pol,
 2    double x, int n) {
 3   /*   Aplica la formula del polinomio para evaluar
 4    * el polinomio pol de grado n en el valor de
 5    * la variable x
 6    * las potencias i-esimas de x se calculan con
 7    * un algoritmo mejorado potencia1*/
 8
 9     int i=n; double suma=0.0;
10     while (i >= 0) {
11       suma += pol[n-i]*potencia1(x, i);
12       i--;
13      }
14      return suma;
15  }
16
17  public static double potencia1(double x,int i) {
18    if( i == 0 ) return 1;
19    if( i == 1 ) return x;
20    if( i%2 == 0 )return potencia1( x * x, i / 2 );
21    else return potencia1( x * x, i / 2 ) * x;
22  }
```

Para **calculaConPotencia1:**

Linea 9: 2 operaciones elementales de inicializacion de **i** y de **suma**
Linea 10: 2*N Entro en el while loop (Comparacion de i con 0 y posterior resta de i-1)
Linea 11: 3 Operaciones: La suma de suma con el valor que devuelva pol[n-i] y la posterior multiplicacion con lo que devuelva la llamada a la funcion **potencia1** (que analizaremos abajo).

Esta funcion es identica a **calcularConPotencia**, por tanto el peor caso para esta funcion es cuando **i** es menor que **n**, dado que entraremos en el while loop y restaremos i-1 hasta que sea menor a 0.

Finalmente, solo para el while de calculaConPotencia1 (<u>obviando</u> la llamada a la funcion potencia1): 5 + 2*N = **O(N) o bien, una lineal.**

Para la funcion **potencia1**, tenemos:

Linea 18: Se evalua la condicion de i == 0
Linea 19: Se evalua la condicion de i == 1
Linea 20: Todos los if que contiene evaluan si **"i"** es divisible con resto 0 por 2 (o sea si es **par**). De ser **verdadero,** devuelve una llamada **recursiva**, en caso contrario (o sea, si es **impar**) **tambien** va a realizar una llamada **recursiva** (independientemente de los parametros). Dado que en dichas llamadas recursivas **se divide el valor de i por la mitad**, quiere decir que se va a ejecutar la cantidad de veces que podamos dividir **i a la mitad**. Esto podemos calcularlo usando la funcion **logaritmica (log)**.

Por ejemplo, si a **calcularConPotencia1** le pasaramos un valor **n = 8,** independientemente del valor de x, se ejecutara **potencia1** solo 3 veces dado que **$\log_2$ (8) = 3.**

Dado que **potencia1** son "if" cases con recursividad, podemos decir que para los mejores casos (0 y 1) es O(1), pero para los demas (los peores) va a ser O(Log N), entonces tenemos **O(Log N) para potencia1.**

Dado que la llamada a **potencia1** se hace **desde** el while loop de **calculaConPotencia1**, para el peor escenario siempre, podemos podemos decir que tenemos **O(N) * O(Log N)**.
Finalmente para **calculaConPotencia1, la complejidad es O(N Log N).**