Análisis de Complejidad Temporal en Algoritmos Recursivos

Algoritmos recursivos

- Ejemplo: Cálculo del factorial de un número natural.

```
int factorial(int n){
    if(n == 1){
        return 1;
    }else{
        return n*factoral(n-1);
    }
}
```

Algoritmos recursivos

- La función que representa el costo de esta función será:

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ T(n-1) + c & \text{sino} \end{cases}$$

Si el valor de entrada es 1 entonces el tiempo es constante y si la entrada es mayor a 1, el tiempo esta dado por el tiempo de ejecutar el algoritmo en el valor de la entrada disminuido en 1 mas un costo constante de la multiplicación y la resta.

Métodos de Análisis

- Para resolver estas recurrencias vamos a ver 2 métodos:
  - Basados en **sustracción**: como el ejemplo Factorial, donde la llamada recursiva se hace con una sustracción sobre n, o sea de la forma $T(n-b)$.
  - Basados en resolución por **división**: donde la llamada se hace con una división sobre n, o sea de la forma $T(n/b)$.

Método basado en sustracción

Para el primer caso en donde tenemos una función de tiempo como la siguiente:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n-b) + p(n) & \text{si } n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $p(n)$ es un polinomio de grado k y $b \in \mathbb{N}$. a es la cantidad de llamadas recursivas que se ejecutan efectivamente para el peor caso, b es la cantidad de unidades en la que se disminuye el tamaño de la entrada y k es el grado del polinomio que representa el tiempo de ejecutar las sentencias fuera de la llamada recursiva, entonces se tiene que el orden es

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

Ejemplo: Factorial de n

```
int factorial(int n){
    if(n == 1){
        return 1;
    }else{
        return n*factoral(n-1);
    }
}
```

El ejemplo del factorial se puede resolver con el primero de los casos en donde $a = 1$, $b = 1$ y $k = 0$, como $a = 1$ podemos utilizar la opción del medio y el resultado es $\Theta(n^{k+1}) = \Theta(n)$. En el siguiente capítulo se verán ejemplos de aplicación de ambos casos.

Método basado en división

Para el caso de resolución por división, tenemos lo siguientes:

$$T(n) = \begin{cases} c & \text{si } 0 \leq n < b \\ aT(n/b) + f(n) & \text{si } n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $f(n) \in \Theta(n^k)$ y $b \in \mathbb{N}$, $b > 1$. a es la cantidad de llamadas recursivas que se ejecutan efectivamente para el peor caso, b es la cantidad de unidades en la que se divide el tamaño de la entrada y k es el grado del polinomio que representa el tiempo de ejecutar las sentencias fuera de la llamada recursiva, entonces se tiene que el orden es

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log(n)) & \text{si } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \end{cases}$$

El ejemplo del factorial se puede resolver con el primero de los casos en donde $a = 1$, $b = 1$ y $k = 0$, como $a = 1$ podemos utilizar la opción del medio y el resultado es $\Theta(n^{k+1}) = \Theta(n)$. En el siguiente capítulo se verán ejemplos de aplicación de ambos casos.

Ejemplo: Búsqueda de un número x

- Indicar si un elemento x dado está presente en una secuencia ordenada de números S también dada. La solución que utiliza esta técnica es la llamada ***Búsqueda binaria.***

ALGORITMO BUSQUEDABIN

Entrada: S: Vector<entero>, x: entero

Salida: **verdadero** o **falso**

```
si longitud(S) = 1
   devolver S[0] = x
sino
   y ← S[longitud(S)/2]
   si x = y
      devolver verdadero
   sino
      mitad ← longitud(S)/2
      si x < y
         devolver BusquedaBin(S[0, mitad − 1], x)
      sino
         devolver BusquedaBin(S[mitad..longitud(S) − 1], x)
      fin si
   fin si
fin si
```

Ejemplo: Búsqueda de un número x

Por ejemplo, para un valor de $x = 3$ la siguiente secuencia

1	3	7	10	25	106	121	130	145

la secuencia de ejecución sería:

1	3	7	10	25

1	3

3

Aplicando método basado en división

Si quisiéramos analizar la complejidad de este algoritmo, en forma intuitiva, podríamos ver que en cada llamada, el tamaño de la entrada se reduce a la mitad, hasta llegar a tamaño 1, con lo que estaríamos pensando en $\mathcal{O}(\log(n))$. Si quisiéramos analizarlo con los métodos de resolución de recurrencias vistos, deberíamos plantear la recurrencia como:

$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 1T(n/2) + c & \text{si } n > 1 \end{cases}$$

y encontrando los valores de a, b y k que son necesarios para resolver la recurrencia con el método de la división, tenemos que $a = 1$, $b = 2$ y $k = 0$, entonces tenemos que $a = b^k$ ya que $1 = 2^0$ con lo que el resultado es $\Theta(n^k \log(n))$ que es $\Theta(n^0 \log(n) = \Theta(\log(n))$ que es lo que habíamos encontrado en forma intuitiva.

Búsqueda binaria en Java

```
public static boolean busquedaBin(VectorTDA<Integer>
    valores, int inicio, int fin, int valor) throws
    Exception{
    if(inicio==fin){
        return (valores.recuperarElemento(inicio) == valor
            );
    }else{
        int mitad = (fin+inicio)/2;
        if(valor <= valores.recuperarElemento(mitad)){
            return busquedaBin(valores,inicio,mitad,valor);
        }else{
            return busquedaBin(valores,mitad+1,fin,valor);
        }
    }
}
```

Ejercicios

1. Implementar y testear en Java el código de Búsqueda binaria.
2. Resolver de la Guía de Trabajos Prácticos de Programación III, los ejercicios ***de Parte 1: Análisis de eficiencia temporal***
3. Hallar la complejidad temporal de los siguientes códigos:

1.
```
unsigned int Funcion1 (unsigned int i){
 if (i <= 1)
      return 1;
 else {
      unsigned int aux = Funcion1 (i-1);
      return 2 * aux;
      }
}
```

2.
```
unsigned int Funcion2 (unsigned int i){
if (i <= 1)
      return 1;
else
      return Funcion2 (i-1)+ Funcion2 (i-1);
}
```

5.
```
void funcion ( int i, int j, unsigned int h) {
     if (h != 0) {
          int m = ( i + j ) / 2 ;
          calculo(m, h);  // calculo pertenece a O(h)
          funcion (i, m, h- 1);
          funcion (m, j, h- 1);
     }
}
```