Estructura de Datos y Algoritmos 1

Teórico #4:

Análisis de tiempo ejecución de algoritmos recursivos

Ejemplo: Factorial de un número natural

```
int Fact(int n){
  if (n == 0) return 1;
  else
    return n*Fact(n-1);
}
```

$$T(n) = \begin{cases} c_1 & \text{Si } n = 0 \\ c_2 + T(n-1) & \text{Si } n \geq 1 \end{cases}$$

¿Cómo calcular $T(n)$?

T(n) para algoritmos recursivos

Expansión de recurrencias

$$T(n) = c_2 + T(n-1) \quad \text{Si } n \geq 1$$

$$T(n-1) = c_2 + T(n-2) \quad \text{Si } n-1 \geq 1 \; (n \geq 2)$$

$$T(n) = 2 \cdot c_2 + T(n-2) \quad \text{Si } n \geq 2$$

$$T(n-2) = c_2 + T(n-3) \quad \text{Si } n-2 \geq 1 \; (n \geq 3)$$

$$T(n) = 3 \cdot c_2 + T(n-3) \quad \text{Si } n \geq 3$$

$$\cdots (k \text{ veces})$$

$$T(n) = k \cdot c_2 + T(n-k) \quad \text{Si } n \geq k$$

$$T(n) = (n-1) \cdot c_2 + T(0) \quad \text{Si } k = n-1$$

$$= n \cdot c_2 - c_2 + c_1 \quad \Rightarrow \boldsymbol{O(n)}$$

Otro Ejemplo: Búsqueda binaria

```
int BusquedaBinaria(int* L, int inicio,
                                int fin, int x){
  if (fin > inicio) return -1;
  int medio = (inicio + fin)/2;
  if (x == L[medio])
    return medio;
  else if (x > L[medio])
    return BusquedaBinaria(L, medio+1, fin, x);
  else
    return BusquedaBinaria(L, inicio, medio-1, x);
}
```

$$T(n) = \begin{cases} c_1 & \text{Si } n \leq 1 \\ c_2 + T(n/2) & \text{Si } n > 1 \end{cases}$$

T(n) para algoritmos recursivos

Expansión de recurrencias

$$T(n) = c_2 + T(n/2) \quad \text{Si } n \geq 1$$

$$T(n/2) = c_2 + T(n/2^2) \quad \text{Si } n/2 > 1 \; (n > 2^1)$$

$$T(n) = 2 \cdot c_2 + T(n/2^2) \quad \text{Si } n > 2^1$$

$$T(n/2^2) = c_2 + T(n/2^3) \quad \text{Si } n/2^2 > 1 \; (n > 2^2)$$

$$T(n) = 3 \cdot c_2 + T(n/2^3) \quad \text{Si } n > 2^2$$

$$\cdots (k \text{ veces})$$

$$T(n) = k \cdot c_2 + T(n/2^k) \quad \text{Si } n > 2^{k-1}$$

$$\text{Si } \frac{n}{2^k} = 1 \;\Rightarrow k = \log(n) \Rightarrow T(n) = \log(n) \cdot c_2 + T(1)$$

$$\Rightarrow \boldsymbol{O(log(n))}$$

Ecuaciones de Recurrencia

Como hemos visto hasta ahora, el tiempo $\mathrm{T(n)}$ se expresa en función del tiempo para valores inferiores del tamaño de las instancias: $\mathrm{T(n - 1)}$, $\mathrm{T(n - 2)}$, $\mathrm{T(n/2)}$, etc

En general las ecuaciones de recurrencia tienen la forma:

$$T(n) = b \qquad \text{para } 0 \leq n \leq n_0 \text{ (condiciones iniciales)}$$

$$T(n) = f(T(n-1), T(n-2), \ldots, T(n-k), n) \qquad \text{en otro caso}$$

Las ecuaciones recurrentes según su forma se pueden clasificar en:

- Ecuaciones lineales **homogéneas**
- Ecuaciones lineales **no homogéneas**

Ecuaciones Lineales homogéneas

Tienen la forma: $a_0T(n) + a_1T(n-1) + \cdots + a_kT(n-k) = 0$

Suponiendo que las soluciones tienen la forma $T(n) = x^n$

$$a_0x^n + a_1x^{n-1} + \cdots + a_kx^{n-k} = 0$$

$$\Big\Downarrow \text{ Dividiendo por } x^n$$

$$a_0x^k + a_1x^{k-1} + \cdots + a_k = 0$$

Lo que recibe el nombre de **Ecuación Característica** de la ecuación recurrente lineal homogénea, con:

- k : conocida
- a_i: conocidas
- x : desconocidas

Ecuaciones Lineales homogéneas

Suponiendo que la ecuación: $a_0x^k + a_1x^{k-1} + \cdots + a_k = 0$

tiene las soluciones $s_1, s_2, \dots, s_k$

Entonces la solución de la ecuación recurrente es:

$$\boxed{T(n) = c_1s_1^n + c_2s_2^n + \cdots + c_ks_k^n = \sum_{i=1}^{k} c_is_i^n}$$

Siendo c_i constantes cuyos valores dependen de las condiciones iniciales

Ejemplo

Calcular el orden de complejidad de un algoritmo cuyo tiempo de ejecución es:

$$T(n) = \begin{cases} n & \text{Si } n \leq 1 \\ 3T(n-1) + 4T(n-2) & \text{Si } n > 1 \end{cases}$$

$$T(n) = 3T(n-1) + 4T(n-2)$$

$$\Updownarrow$$

$$T(n) - 3T(n-1) - 4T(n-2) = 0$$

de donde: $a_0 = 1, a_1 = -3, a_2 = 4, k = 2$

Ecuación característica $x^2 - 3x - 4 = 0$

donde: $s_1 = -1, s_2 = 4$

Solución General: $T(n) = c_1(-1)^n + c_2 4^n$

Ejemplo (cont)

Solución General: $T(n) = c_1(-1)^n + c_2 4^n$

Luego, hay que calcular los valores de las constantes. Para ello se usarían los valores iniciales $T(0) = 0, T(1) = 1$

$$\begin{aligned} c_1 + c_2 &= 0 \\ -c_1 + 4c_2 &= 1 \end{aligned} \quad \Longrightarrow \quad c_1 = -\frac{1}{5}, \qquad c_2 = \frac{1}{5}$$

$$T(n) = (-\frac{1}{5})(-1)^n + (\frac{1}{5})4^n \in O(4^n)$$

Ecuaciones Lineales homogéneas

Dadas las soluciones $s_1, s_2, \ldots, s_k$ siendo s_k de multiplicidad m la solución será

$$T(n) = c_1 s_1^n + c_2 s_2^n + \cdots + c_k s_k^n + c_{k+1} n^1 s_k^n + \cdots + c_{k+2} n^2 s_k^n + \cdots + c_{k+m-1} n^{m-1} s_k^n$$

Ejemplo:

$$T(n) = \begin{cases} n & \text{Si } n \leq 2 \\ 5T(n-1) - 8T(n-2) + 4T(n-3) & \text{Si } n > 2 \end{cases}$$

Ecuación característica $x^3 - 5x^2 + 8x - 4 = 0$

donde: $s_1 = 1$ (multiplicidad 1)

$s_2 = 2$ (multiplicidad 2)

Ecuaciones Lineales homogéneas

Ejemplo:

$$T(n) = \begin{cases} n & \text{Si } n \leq 2 \\ 5T(n-1) - 8T(n-2) + 4T(n-3) & \text{Si } n > 2 \end{cases}$$

Ecuación característica $x^3 - 5x^2 + 8x - 4 = 0$

donde: $s_1 = 1$ (multiplicidad 1) $\quad s_2 = 2$ (multiplicidad 2)

Solución: $T(n) = c_{11}1^n + c_{21}2^n + c_{22}n2^n$

Según condiciones iniciales

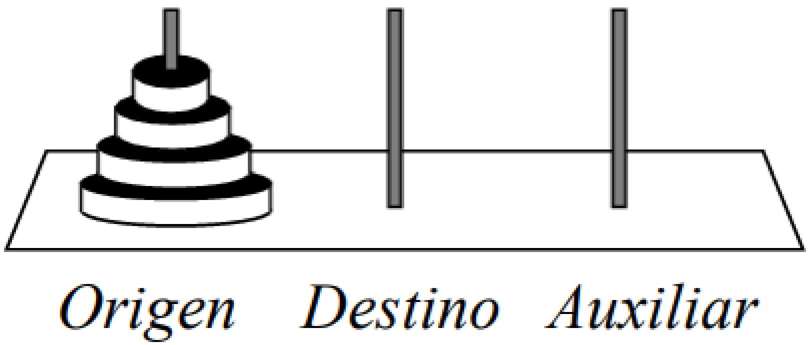
$$\begin{aligned} c_{11} + c_{21} &= T(0) = 0 \\ c_{11} + 2c_{21} + 2c_{22} &= T(1) = 1 \\ c_{11} + 4c_{21} + 8c_{22} &= T(2) = 2 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} c_{11} &= -2, \\ c_{21} &= 2, \\ c_{22} &= -1/2, \end{aligned}$$

Por tanto: $T(n) = 2^{n+1} - n2^{n-1} - 2 \in O(n2^n)$

Ejemplo: Algoritmo Torres de Hanoi

Se tienen **n** discos de diferentes tamaños ubicados ordenadamente en la torre A (torre origen) y se quieren pasar para la torre B (torre destino) quedando los discos en el mismo orden.

Pero solo se puede tomar un solo disco cada vez y nunca puede estar un disco encima de otro más pequeño. Se tiene una torre C que puede ser usada de intermediaria (torre auxiliar).



Ejemplo: Algoritmo Torres de Hanoi

```
void Hanoi(int n, char origen, char destino, char auxiliar){
  if (n == 1)
    cout << “Mover el disco de base ” << origen
         << “ para la base ” << destino << endl
  else
  {
    /* Mover los n-1 discos de “origen“ a “auxiliar“ usando
       “destino“ como auxiliar */
    Hanoi(n-1, origen, destino, auxiliar);
    /* Mover disco n de “origen” para “destino” */
    cout << “Mover disco ” << n << “ de base ” << origen
         << “ para la base ” << destino << endl
    /* Mover los n-1 discos de “auxiliar” a “destino” usando
       “origen” como auxiliar */
    Hanoi(n-1, auxiliar, destino, origen);
  }
}
```

Ejemplo: Algoritmo Torres de Hanoi

$$T(n) = \begin{cases} 1 & \text{Si } n = 1 \\ 2T(n-1) + 1 & \text{Si } n > 1 \end{cases}$$

$$T(n) = 2T(n-1) + 1$$

$$T(n-1) = 2T(n-2) + 1$$

$$T(n) - T(n-1) = 2T(n-1) - 2T(n-2)$$

$$T(n) - 3(n-1) + 2T(n-2) = 0$$

Ecuación característica $x^2 - 3x + 2 = 0$ $\quad s_1 = 1, s_2 = 2$

Solución: $T(n) = c_{11}1^n + c_{21}2^n$

Según condiciones iniciales $c_{11} = -1, c_{21} = 1,$

$$T(n) = 2^n - 1 \in O(2^n)$$

Cambio de variable

Se utiliza en expresiones del tipo:

$$\boldsymbol{T(n) = aT(n/2) + bT(n/4) + \cdots}$$

Para resolverlas se procede a:

1. Convertir las ecuaciones anteriores en algo de la forma:

$$\boldsymbol{S(k) = aS(k - c_1) + bS(k - c_2) + \cdots}$$

2. Resolver el problema en ***k***
3. Deshacer el cambio y obtener el resultado en ***n***

Cambio de variable: Ejemplo

$$T(n) = 4T(n/2) + n$$

Reemplazar n por 2^k , o sea $S(k) = T\left(2^k\right) = T(n)$

$$S(k) = 4T\left(2^{k-1}\right) + 2^k$$

$$S(k) = 4S(k-1) + 2^k$$

$$2S(k-1) = 8T(k-2) + 2^k$$

$$S(k) - 2S(k-1) = 4S(k-1) - 8S(k-2)$$

$$S(k) - 6S(k-1) + 8S(k-2) = 0$$

Cambio de variable: Ejemplo

$$T(n) = 4T(n/2) + n$$

Reemplazar n por 2^k , o sea $S(k) = T\left(2^k\right) = T(n)$

$$S(k) - 6S(k-1) + 8S(k-2) = 0$$

Ecuación característica $x^2 - 6x + 8 = 0 \quad s_1 = 4, s_2 = 2$

$$S(k) = c_1 4^k + c_2 2^k$$

$$\Downarrow$$

$$T(n) = c_1 n^2 + c_2 n$$

Ejercicios

Calcular el Orden de Complejidad de los siguientes algoritmos

1) Probar que el algoritmo InsertionSort es $\boldsymbol{O(n^2)}$

$$\boldsymbol{T(n) = \begin{cases} d & \text{Si } n \leq 1 \\ T(n-1) + cn & \text{Si } n > 1 \end{cases}}$$

2) Probar que el algoritmo MergeSort es $\boldsymbol{O(n \log n)}$

$$\boldsymbol{T(n) = \begin{cases} d & \text{Si } n \leq 1 \\ 2T(n/2) + cn & \text{Si } n > 1 \end{cases}}$$

Complejidad de Divide y Vencerás

Teorema que permite calcular la complejidad para tiempos de ejecución con la forma

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + O(n^k) \quad a \geq 1, b > 1}$$

$$\boldsymbol{T(n) = \begin{cases} O(n^{\log_b a}) & \text{Si } a > b^k \\ O(n^k \log_2 n) & \text{Si } a = b^k \\ O(n^k) & \text{Si } a < b^k \end{cases}}$$

Observar que MergeSort es $\boldsymbol{O(n \log_2 n)}$, $\boldsymbol{a = b = 2, k = 1}$

Regla práctica: es mejor que los subproblemas tengan tamaños aproximadamente iguales para que el rendimiento del algoritmo sea “bueno”. Por ejemplo, comparar InsertSort y MergeSort.