Programación 2

Introducción al análisis de algoritmos recursivos

T(n) para programas recursivos

- **<u>Un ejemplo</u>**: “factorial”

```
int fact (int n)
   { if (n>1) return n*fact(n-1);
     else return 1; }
```

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$\Rightarrow$ **T(n) es O(?)**

T(n) para programas recursivos

- **<u>Expansión de recurrencias:</u>**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n > 1$) c es una constante

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ $(n>2)$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ $(n>2)$

$T(n) = 2.c + T(n-2)$, Si $n>2$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ $(n>2)$

$T(n) = 2.c + T(n-2)$, Si $n>2$

$T(n-2) = c + T(n-3)$, Si $n-2>1$ $(n>3)$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ $(n>2)$

$T(n) = 2.c + T(n-2)$, Si $n>2$

$T(n-2) = c + T(n-3)$, Si $n-2>1$ $(n>3)$

$T(n) = 3.c + T(n-3)$, Si $n>3$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ $(n>2)$

$T(n) = 2.c + T(n-2)$, Si $n>2$

$T(n-2) = c + T(n-3)$, Si $n-2>1$ $(n>3)$

$T(n) = 3.c + T(n-3)$, Si $n>3$

… (*i veces*)

$T(n) = i.c + T(n-i)$, Si $n>i$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n-1)$ (Si $n>1$) c es una constante

$T(n-1) = c + T(n-2)$, Si $n-1>1$ ($n>2$)

$T(n) = 2.c + T(n-2)$, Si $n>2$

$T(n-2) = c + T(n-3)$, Si $n-2>1$ ($n>3$)

$T(n) = 3.c + T(n-3)$, Si $n>3$

… (*i veces*)

$T(n) = i.c + T(n-i)$, Si $n>i$

Si $i=n-1$: $T(n) = (n-1).c + T(1)$, Si $\underline{n>n-1}$ Ok

$T(n) = n.c - c + d \Rightarrow$ **O(n)**

T(n) para programas recursivos

- **Otro ejemplo**: Ver que el orden O de tiempo de ejecución de un algoritmo de búsqueda binaria sobre un vector ordenado de *n* elementos es $O(\log_2(n))$

Reflexionar sobre la eficiencia comparativa de la búsqueda secuencial y la binaria sobre un arreglo (vector) ordenado.

3	**5**	**8**	**10**	**14**	**22**	**29**	**45**	**77**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n > 1$) c es una constante

Podemos asumir *n* es potencia de 2 para realizar la expansión de recurrencia:

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \le 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n > 1$) c es una constante

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n>1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2>1$ ($n>2^1$)

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n>1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2>1$ $(n>2^1)$

$T(n) = 2.c + T(n/2^2)$, Si $n>2^1$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \le 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n > 1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2 > 1$ ($n > 2^1$)

$T(n) = 2.c + T(n/2^2)$, Si $n > 2^1$

$T(n/2^2) = c + T(n/2^3)$, Si $n/2^2 > 1$ ($n > 2^2$)

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n>1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2>1$ $(n>2^1)$

$T(n) = 2.c + T(n/2^2)$, Si $n>2^1$

$T(n/2^2) = c + T(n/2^3)$, Si $n/2^2>1$ $(n>2^2)$

$T(n) = 3.c + T(n/2^3)$, Si $n>2^2$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n>1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2>1$ $(n>2^1)$

$T(n) = 2.c + T(n/2^2)$, Si $n>2^1$

$T(n/2^2) = c + T(n/2^3)$, Si $n/2^2>1$ $(n>2^2)$

$T(n) = 3.c + T(n/2^3)$, Si $n>2^2$

… (*i veces*)

$T(n) = i.c + T(n/2^i)$, Si $n>2^{i-1}$

T(n) para programas recursivos

- **Expansión de recurrencias:**

$T(n) = d$ (Si $n \leq 1$) d es una constante

$T(n) = c + T(n/2)$ (Si $n>1$) c es una constante

$T(n/2) = c + T(n/2^2)$, Si $n/2>1$ $(n>2^1)$

$T(n) = 2.c + T(n/2^2)$, Si $n>2^1$

$T(n/2^2) = c + T(n/2^3)$, Si $n/2^2>1$ $(n>2^2)$

$T(n) = 3.c + T(n/2^3)$, Si $n>2^2$

… (*i veces*)

$T(n) = i.c + T(n/2^i)$, Si $n>2^{i-1}$

Si $n/2^i=1 \Rightarrow i=\log(n) \Rightarrow T(n) = \log(n).c + T(1) \Rightarrow$ **O(log(n))**

Ejemplo sobre un AB

Considere el siguiente procedimiento sobre un AB:

```
void proc (AB t){
1.      if (t==NULL) accionBase
2.      else if (condicion1(t)) proc(t->izq);
3.      else if (condicion2(t)) proc(t->der);
4.      else accionNodo;
} //Donde: condicion1, condicion2 y accionNodo tienen O(1)
```

Peor caso:

T(n) = d, Si n=0 (n es la cantidad de nodos del árbol *t*) **– 1 –**

T(n) = c + T(n-1), Si n>0 **– 2, 3 –**

=> Es O(n)

Ejemplo sobre un AB (cont.)

Considere el siguiente procedimiento sobre un AB:

```
void proc (AB t){
1.      if (t==NULL) accionBase
2.      else if (condicion1(t)) proc(t->izq);
3.      else if (condicion2(t)) proc(t->der);
4.      else accionNodo;
} //Donde: condicion1, condicion2 y accionNodo tienen O(1)
```

Si para cada nodo de *t* la cantidad de nodos a la izquierda y derecha fuera igual, +/- 1:

T(n) = d, Si n=0 (n es la cantidad de nodos del árbol *t*) **– 1 –**

T(n) = c + T(n/2), Si n>0 **– 2, 3 –**

=> Es $O(\log_2(n))$

T(n) para programas recursivos Métodos generales de resolución

Resolución de ecuaciones de recurrencia:

- Suposición de una solución (*guess*)
- <u>Expansión de recurrencias</u>
- Soluciones generales para clases de recurrencias
  - Soluciones homogéneas y particulares
  - Funciones Multiplicativas

  ………….

Ejercicio - Hanoi

Calcular el Orden O del algoritmo:

```
void hanoi(int n, char origen, char destino, char auxiliar){
      if(n > 0){

          /* Mover los n-1 discos de "origen" a "auxiliar" usando "destino" como auxiliar */
          hanoi(n-1, origen, auxiliar, destino);

          /* Mover disco n de "origen" para "destino" */
          printf("\n Mover disco %d de base %c para a base %c", n, origen, destino);

          /* Mover los n-1 discos de "auxiliar" a "destino" usando "origen" como auxiliar */
          hanoi(n-1, auxiliar, destino, origen);
}

main(){
          int n;
          printf("Digite el número de discos: ");
          scanf("%d",&n);
          hanoi(n, 'A', 'C', 'B');
          return 0;
}
```

$$T(n) = d \quad (\text{Si } n \leq 0)$$
$$T(n) = c + 2.T(n-1) \quad (\text{Si } n > 0)$$

Ejercicio: Potencia

Cuánto tiempo se requiere para calcular f(x) = $\sum_{i=0}^{n} a_i x^i$ en los dos siguientes casos para n potencia de 2?

- Usando una rutina simple para realizar la exponenciación.
- Usando la siguiente rutina

```
int potencia(int x,int n) {
    int resultado;
    if (n==0) resultado=1;
    else if (n==1) resultado=x;
        else if (n%2==0) resultado=potencia(x*x, n/2);
            else resultado=potencia(x*x, n/2)*x;
    return resultado;
}
```

Insert Sort y Merge Sort

Calcular el orden (O) de los algoritmos de ordenación: *Insert sort* y *Merge sort*.

InsertSort:

$\mathbf{T(n) = \mathit{d} \;\; (Si\ n \leq 1)}$

$\mathbf{T(n) = T(n-1) + \mathit{c}{*}n \; (Si\ n>1)}$

Probar que es $O(n^2)$

MergeSort:

$\mathbf{T(n) = \mathit{d} \;\; (Si\ n \leq 1)}$

$\mathbf{T(n) = 2.T(n/2) + \mathit{c}{*}n \; (Si\ n>1)}$

Probar que es $O(n{*}\log_2(n))$

Algoritmos *Divide and Conquer*

En la sección 10.2.1 del Weiss se presenta un formato genérico de tiempo de ejecución para algoritmos *divide and conquer* (con las correspondientes soluciones):

$T(n) = a*T(n/b) + O(n^k)$, donde $a>=1$ y $b>1$

Algoritmos *Divide and Conquer*

Si $T(n) = a*T(n/b) + O(n^k)$, donde $a>=1$ y $b>1$

Entonces:

- **$T(n)$ es $O(n^{\log_b(a)})$ Si $a > b^k$**
- **$T(n)$ es $O(n^k * \log_2(n))$ Si $a = b^k$**
- **$T(n)$ es $O(n^k)$ Si $a < b^k$**

Observar que MergeSort es $O(n*\log_2(n))$, $a=b=2$ y $k=1$

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*.

Bibliografía

- **Estructuras de Datos y Análisis de Algoritmos**
  Mark Allen Weiss

  (Capítulo 2)

- **Estructuras de Datos y Algoritmos.**
  A. Aho, J. E. Hopcroft & J. D. Ullman

  (Capítulo 1, secciones 1.4 y 1.5**)**

  (Capítulo 9: recurrencias**)**