RECURSION

Temario

- ✓ Tipos de recursión
- ✓ Eficiencia y recursión

Tipos de RECURSION

Según desde donde se realice el llamado recursivo:

Directa: la función se llama a sí misma.

Indirecta: la función A llama a la función B, y esta última llama a la función A

Tipos de RECURSION

Según el número de llamadas recursivas generadas en tiempo de ejecución:

Lineal o simple: se genera una única llamada interna. Ejemplo función de cálculo de potencia,

No lineal o múltiple: se generan dos o mas llamadas internas. Ejemplo los árboles binarios.

Tipos de RECURSION

Según el punto donde se realice el llamado recursivo:

Final (Tail recursion): se genera una única llamada interna. Ejemplo función de mcd (máximo común divisor)

$$\text{mcd}(a, b) = \text{mcd}(b, a) = \begin{cases} \text{mcd}(a-b, b) & \text{si } a > b \\ \text{mcd}(a, b-a) & \text{si } a < b \\ a & \text{si } a = b \end{cases}$$

No final (Nontail recursive function): se hace alguna operación al volver la llamada recursiva. Ejemplo: impresión inorder.

Ventajas de la RECURSION

Solución de problemas de manera natural, sencilla, comprensible y elegante.

Facilidad de comprobar y verificar que la solución es correcta. **Principio de inducción matemática**.

Desventajas de la RECURSION

En general las soluciones recursivas son mas ineficientes con respecto al tiempo y al espacio que las soluciones iterativas. Esto ocurre debido a las llamadas recursivas a los módulos , la creación de variables dinámicas en la pila de activación, la duplicación de variables por parámetros por valor, etc. (OVERHEAD- sobrecarga)

Algunas soluciones recursivas pueden repetir cálculos innecesariamente. Pro ejemplo: revisar soluciones del número de fibonacci. ¿Qué ocurre en la solución clásica?

ANALISIS de RECURSION

En general cualquier función recursiva se puede transformar en una solución iterativa.

Ventaja de la solución iterativa: más eficiente en tiempo y espacio.
Desventaja de la solución iterativa: en algunos casos puede ser muy complicada y a menudo suelen necesitarse estructuras de datos auxiliares.

Si la eficiencia es un parámetro crítico y la función se va a ejecutar frecuentemente, conviene escribir una solución iterativa

ANALISIS de RECURSION

La recursión puede simular el funcionamiento de una PILA abstracta.

¿En que casos?

¿Como ocurre esta simulación?

EFICIENCIA

REPASO DE EFICIENCIA

EFICIENCIA

Se define T(n) al tiempo de ejecución de un programa con una entrada de tamaño n.

Las unidades de T(n) no se especifican.

NOTACIÓN ASINTÓTICA

Se utiliza para hacer referencia a la velocidad de crecimiento de los valores de una función. Nos interesa estudiar el comportamiento de los algoritmos para volúmenes de datos de gran tamaño.

Por ejemplo: Si el tiempo de ejecución de un programa **T(n)** es **$O(n^2)$** significa que existen constantes **C** y **n_0** tales que

$T(n) <= Cn^2$ para **$n>=n_0$**

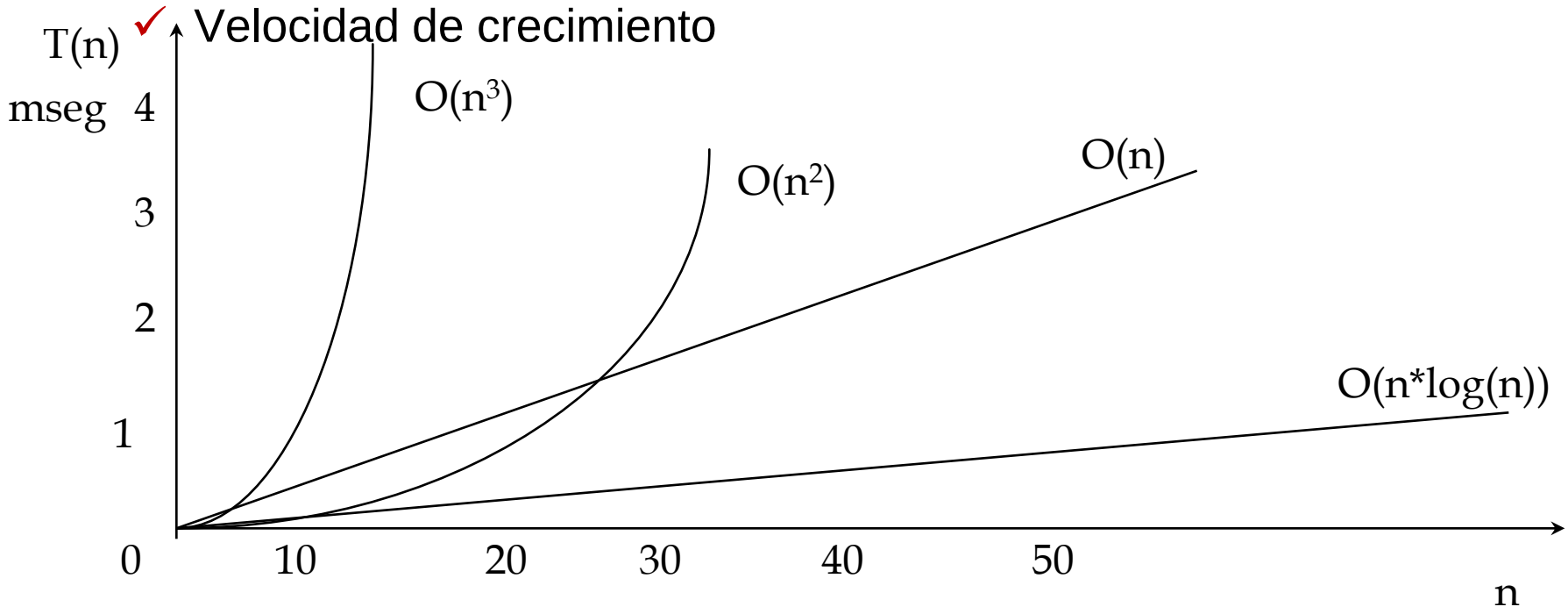
EFICIENCIA

- Se dice que **T(n)** es **O(f(n))** si existen constantes positivas **C** y $\mathbf{n_0}$ tales que **T(n) <= C f(n)** cuando **n>=** $\mathbf{n_0}$.

- Cuando el tiempo de ejecución de un programa es de **O(f(n))**, se dice que tiene velocidad de crecimiento **f(n).**

- **f(n)** es una cota superior para la velocidad de crecimiento de **T(n).**

EFICIENCIA


Velocidad de crecimiento
T(n)
mseg
4
3
2
1
0
10
20
30
40
50
n
O(n³)
O(n²)
O(n)
O(n*log(n))

EFICIENCIA

Algunas condiciones en las cuales el tiempo de ejecución de un programa se puede ignorar en favor de otros factores:

- ✓ Si un programa se va a utilizar algunas veces, el costo de su escritura y depuración es el dominante.
- ✓ Si un programa se va a ejecutar sólo con entradas “pequeñas”, la velocidad de crecimiento puede ser menos importante que el factor constante.

EFICIENCIA

- Se tendrá en cuenta para una medida de eficiencia es el número de operaciones elementales que emplea el algoritmo.
- Una operación elemental utiliza un tiempo constante para su ejecución, independientemente del tipo de dato con el que trabaje.
- Se considera que cada operación elemental se ejecuta en una unidad de tiempo.
- Una operación elemental es una asignación, una comparación o bien una operación aritmética simple

Eficiencia

Reglas Generales

- ✓ **Regla 1**: For
- ✓ **Regla 2**: For anidados
- ✓ **Regla 3**: sentencias consecutivas
- ✓ **Regla 4**: If / else

Los comentarios y declaraciones no se consideran para el cálculo

IMPORTANTE:

Puede darse el caso que el algoritmo mas adecuado en cada momento dependa del tamaño de la entrada:

Algoritmo1: $f(n) = 4n - 1$

Algoritmo2: $h(n) = 2n + 10^6$

En estas situaciones hay que considerar cual es el algoritmo a aplicar teniendo en cuenta el tamaño de la entrada.

Eficiencia – Cálculo del Tiempo de Ejecución

- ✓ Ejemplo: cálculo de $\sum_{i=1}^{n} i^3$
- ✓

```
Function sum ( n: integer ) : integer;
    var i, s_parcial : integer;

    begin
{1}   s_parcial := 0;
{2}   for i := 1 to n do
{3}       s_parcial := s_parcial + i * i * i;
{4}   sum := s_parcial;
     end;
```

Eficiencia

Las líneas {1} y {4} cuentan una unidad cada una, entonces tenemos 2

La línea {3} cuenta 4 unidades cada vez que se ejecuta, entonces **4 n**
(1 asignación, 1 suma y 2 productos)

La línea {2} tiene una inicialización, testeo de i<=n e incremento de i, entonces 1 de la asignación, más (n+1) para todos los test y n para el incremento, entonces **2n + 2**

T(N) ==> total = 2 + 4 n + 2 n + 2

==> 6 n + 4 = **O(n)**

Eficiencia - Ejemplos

- ✓ Consideremos el cálculo del mínimo m de tres números a, b y c.

- ✓ Hay varios caminos para obtenerlo:

{camino 1}

```
m := a;
if b < m then m := b;
if c < m then m := c;
```

Peor de los casos, sumamos 5

Eficiencia - Ejemplos

```
{camino 2}
   if a <= b then
                  if a <= c then
                                 m := a
                               else
                                 m := c
               else
                  if b <= c then
                                 m := b
                               else
                                 m := c
```

Siempre suma 3

Eficiencia - Ejemplos

{camino 3}

```
if (a <= b ) and ( a<= c) then m := a;
if (b <= a ) and ( b<= c) then m := b;
if (c <= a ) and ( c<= b) then m := c;
```

Se evalúan todas las condiciones, si no es short circuit, se suma 2 testeos por cada if (6 testeos) y 1 asignación.

El peor de los casos es que sean todos iguales y se suma 3 de las 3 asignaciones, totaliza 9.

Eficiencia - Ejemplos

{camino 4}

```
if (a <= b) and (a<= c ) then
                        m := a
                else if  b <= c then
                                m := b
                        else
                                m := c;
```

ANALIZAR RELACION EFICIENCIA – LINEAS DE CODIGO

En el peor de los casos suma 4 y en el mejor 3.

Eficiencia

Se puede mejorar la eficiencia de los programas, observando algunas guías simples relacionadas con la eficiencia del código.

Un punto importante a tener en cuenta es no repetir cálculos innecesarios. Es decir, podemos escribir:

```
t := x * x * x;
y := 1/(t-1) + 1/(t-2) + 1/(t-3) + 1/(t-4)
```

en lugar de

```
y:=1/(x*x*x-1) + 1/(x*x*x-2) +1/(x*x*x-3) + 1/(x*x*x-4)
```

Eficiencia

Cálculo del tiempo de ejecución en una solución modularizada

- ✓ Si se tiene un programa con módulos que no son recursivos, es posible calcular el tiempo de ejecución de los distintos procesos, uno a la vez, partiendo de aquellos que no llaman a otros. Debe haber al menos un módulo con esa característica.

- ✓ Después puede evaluarse el tiempo de ejecución de los procesos que llamaron a los módulos anteriores y así sucesivamente.

Cálculo de TIEMPOS de la RECURSION

•En el caso de los ***procesos recursivos*** se debe asociar a cada proceso/modulo recursivo una **función desconocida de tiempo T (n)**, donde *n* mide el tamaño de los argumentos del proceso.

•Luego se puede obtener una **recurrencia para T(n),** es decir una ecuación para T(n) en función de T(k) para varios valores de k.

•Si la recursión es utilizada como reemplazo de un lazo iterativo el análisis de tiempo es relativamente fácil

Métodos para obtener la función de recurrencia

1. Suponer una función f(n) y usar la recurrencia para demostrar que $T(n) = f(n)$. La prueba se hace sobre inducción sobre n.

2. Sustituir las recurrencias por su igualdad hasta llegar a cierta $T(n_0)$ conocida

3. Usar soluciones generales para ciertas ecuaciones de recurrencia conocidas: Resolución de la ecuación característica.

Cálculo de TIEMPOS de la RECURSION

```
Function Factorial (n: integer): integer;
begin
 {1}    if (n=0) or (n=1)

 {2}      then Factorial := 1

 {3}      else Factoral:= n * Factorial (n-1);

end;
```

Para ciertas constantes c y d:

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

T(n) es el tiempo de ejecución de Factorial(n)

El tiempo de las líneas {1} y {2} es O(1) y para la línea {3} es O(1) + T(n-1). Se está suponiendo que la multiplicación de dos enteros es una operación de orden 1

Cálculo de TIEMPOS de la RECURSION

(A)
Para ciertas constantes c y d:

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

2) Reemplazamos a n por (n-1)

1) Si n > 2, desarrollamos T(n-1) porque hay un llamado recursivo (ver programa de factorial).

3) Queda: $T(n-1) = c + T(n - 1 - 1)$

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

4) Con lo cual puedo ahora reemplazar a T(n -1) con c + T(n-2) en (A), como c es una constante quedaría que

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

Métodos para obtener la función de recurrencia

Usaremos el método (2) para sustituir las recurrencias por su igualdad hasta llegar a cierta $T(n_0)$ conocida, buscando siempre una **cota superior**

$$
\begin{aligned}
T(n) &= T(n-1) + c_2 \\
&= (T(n-2) + c_2) + c_2 && = T(n-2) + 2*c_2 = \\
&= (T(n-3) + c_2) + 2*c_2 && = T(n-3) + 3*c_2 = \\
&\ldots \\
&= T(n-k) + k*c_2
\end{aligned}
$$

Cuando $k = n-1$, tenemos que $T(n) = T(1) + c_2*(n-1)$, y es $O(n)$.

RECURSION

Link de consulta:

http://www.unlu.edu.ar/~program1/biblioteca/progr-pascal.pdf

Ver capítulo 10. Para recursión