
# Desarrollo Sistemático de Programas

## Unidad 2

## Análisis de Algoritmos

Ing. Carlos A. Afranllie

## Análisis de Algoritmos

• Cuando un programa se va a usar repetidamente resulta importante que los algoritmos implicados sean eficientes.

• Generalmente, asociaremos eficiencia con el tiempo de ejecución del programa, y más raramente con la utilización del espacio de memoria.

• Vamos a concentrarnos en la definición de eficiencia como el tiempo de ejecución de un algoritmo en función del tamaño de su entrada. A la eficiencia de un algoritmo también se le denomina costo, rendimiento o complejidad del algoritmo.

## Costo de un Algoritmo

Para medir el coste de un algoritmo se pueden emplear dos enfoques:

• **Pruebas:** (también llamadas benchmarking en inglés) consiste en elaborar una muestra significativa o típica de los posibles datos de entrada del programa, y tomar medidas cuidadosas del tiempo de ejecución del programa para cada uno de los elementos de la muestra.

• **Análisis:** emplea procedimientos matemáticos para determinar el costo del algoritmo; sus conclusiones son fiables, pero a veces su realización es difícil o imposible a efectos prácticos.

## Costo de un Algoritmo

El análisis de algoritmos mediante el uso de herramientas como por ejemplo la evaluación de costos intenta determinar que tan eficiente es un algoritmo para resolver un determinado problema. En general el aspecto más interesante a analizar de un algoritmo es su costo.

# Análisis de Algoritmos

## Costo de un Algoritmo

• **Costo de tiempo:** Suele ser el más importante, indica cuanto tiempo insume un determinado algoritmo para encontrar la solución a un problema, se mide en función de la cantidad o del tamaño de los datos de entrada.

• **Costo de espacio:** Mide cuanta memoria (espacio) necesita el algoritmo para funcionar correctamente.

***Importante:*** *El análisis del costo de un algoritmo comprende el costo en tiempo y en espacio.*

# Análisis de Algoritmos

## Costo de un Algoritmo

1. El primer paso que debe llevarse a cabo para analizar un algoritmo es definir con precisión lo que entendemos por tamaño de los datos de entrada. Normalmente tal definición resulta natural y no es complicado dar con ella.

   Por ejemplo:

# Análisis de Algoritmos

## Costo de un Algoritmo

• En un algoritmo de ordenación, el tamaño de la entrada es el número de elementos a ordenar.

• En un algoritmo de búsqueda, el tamaño de la entrada es el número de elementos entre los que hay que buscar uno dado.

• En un algoritmo que actúa sobre un conjunto, el tamaño de la entrada es el número de elementos que pertenecen al conjunto.

## Costo de un Algoritmo

2. Una vez definido el tamaño resulta conveniente usar una función **T(n)** para representar el número de unidades de tiempo (segundos, milisegundos,...) que un algoritmo tarda en ejecutarse cuando se le suministran unos datos de entrada de tamaño ***n***.

# Análisis de Algoritmos

## Costo de un Algoritmo

Como el tiempo de ejecución de un programa puede variar sustancialmente según el computador concreto en que se lleve a cabo la ejecución, resulta preferible que **T(n)** sea el número de instrucciones simples (asignaciones, comparaciones, operaciones aritméticas, etc.) que se ejecutan o equivalentemente, el tiempo de ejecución del algoritmo en un computador idealizado, donde cada asignación, comparación, lectura, etc. consume 1 unidad de tiempo.

# Análisis de Algoritmos

## Modelos computacionales

- **Maquina RAM de costo fijo:** La máquina RAM proviene de Random Access Memory. Y es una máquina ideal muy similar a una computadora actual aunque con algunas simplificaciones. Los programas de la máquina RAM se almacenan en memoria. Puede suponerse que todos los accesos a memoria tienen el mismo costo (en tiempo) y que todas las instrucciones tienen un costo constante e idéntico (en tiempo).

## Modelos computacionales

- **Maquina RAM de costo Variable:** En ésta el concepto es similar a la anterior salvo que sus costos no son fijos.

# Máquina RAM de costo fijo

## Ejemplo 1

```
Algoritmo 1 - Este es un ejemplo
        y <- 1              /* Asignación
        A[2] <- 1           /* Asignación a un elemento de un vector
        si n < 1 entonces
                X <- w
        sino
                X <- y
        fin_si
        mientras n >= 0 hacer
                n  <- n -1
        fin_mientras
        para i desde 0 hasta 10 hacer
                A[i] <- 0
        fin_para
Fin_Algoritmo
```

# Máquina RAM de costo fijo

## Ejemplo 2

**Algoritmo 2** - Ejemplo simple

```
        a  <- 3
        b  <-  a * 5
        si b = 5 entonces
                a  <- 2
        sino
                a  <- 1
        fin_si
Fin_Algoritmo
```

El **algoritmo 2** tiene 5 instrucciones de las cuales se ejecutan únicamente 4. Por lo tanto el costo del programa en tiempo es de C*4, siendo C una constante que indica cuanto tiempo tarda en ejecutarse una instrucción. El espacio que necesita el programa es el espacio ocupado por las variables A y B. Es decir 2*Ec siendo Ec el espacio que ocupa una variable.

# Máquina RAM de costo variable

## Ejemplo

**Algoritmo 3** - Ejemplo simple

```
        a  <- 3
        b  <-  a * 5
        si b = 5 entonces
                a  <- 2
        sino
                a  <- 1
        fin_si
Fin_Algoritmo
```

Este programa cuesta ahora 3*C1+1*C2+1*C3.
Donde C1 = costo de una asignación, C2 = costo de una multiplicación y C3 = costo de comparar dos números.

El costo del espacio en la máquina RAM de costo variable es igual al de la máquina RAM de costo fijo.

## Algoritmos iterativos

Los algoritmos iterativos son aquellos que se basan en la ejecución de ciclos; que pueden ser de tipo for, while, repeat, etc.

La gran mayoría de los algoritmos tienen alguna parte iterativa y muchos son puramente iterativos.

Analizar el costo en tiempo de estos algoritmos implica entender cuantas veces se ejecuta cada una de las instrucciones del algoritmo y cual es el costo de cada una de las instrucciones.

## Ordenamiento Uno contra Todos

Uno de los temas importantes del curso es el análisis de algoritmos de ordenamiento, por lo que vamos a empezar con uno de los más simples: el de uno contra todos.

```
Algoritmo Uno_contra_todos (A,n). Ordena el vector A.
        para i desde 1 hasta (n -1) hacer
                para  j desde (i + 1) hasta n hacer
                        si A[j] < A[i] entonces
                                Swap(A[i], A[j])
                        fin_si
                fin_para
        fin_para
Fin_Algoritmo
```

# Ordenamiento Uno contra Todos

| **Ejemplo:** | | | |
|---|---|---|---|
| 3 4 1 5 2 7 6 4 | Compara A[1] con A[2] | 1 2 4 5 3 7 6 4 | Compara A[3] con A[5] |
| 3 4 1 5 2 7 6 4 | Compara A[1] con A[3] | 1 2 3 5 4 7 6 4 | Compara A[3] con A[6] |
| 1 4 3 5 2 7 6 4 | Compara A[1] con A[4] | 1 2 3 5 4 7 6 4 | Compara A[3] con A[7] |
| 1 4 3 5 2 7 6 4 | Compara A[1] con A[5] | 1 2 3 5 4 7 6 4 | Compara A[3] con A[8] |
| 1 4 3 5 2 7 6 4 | Compara A[1] con A[6] | 1 2 3 5 4 7 6 4 | Compara A[4] con A[5] |
| 1 4 3 5 2 7 6 4 | Compara A[1] con A[7] | 1 2 3 4 5 7 6 4 | Compara A[4] con A[6] |
| 1 4 3 5 2 7 6 4 | Compara A[1] con A[8] | 1 2 3 4 5 7 6 4 | Compara A[4] con A[7] |
| 1 4 3 5 2 7 6 4 | Compara A[2] con A[3] | 1 2 3 4 5 7 6 4 | Compara A[4] con A[8] |
| 1 3 4 5 2 7 6 4 | Compara A[2] con A[4] | 1 2 3 4 5 7 6 4 | Compara A[5] con A[6] |
| 1 3 4 5 2 7 6 4 | Compara A[2] con A[5] | 1 2 3 4 5 7 6 4 | Compara A[5] con A[7] |
| 1 2 4 5 3 7 6 4 | Compara A[2] con A[6] | 1 2 3 4 4 7 6 5 | Compara A[5] con A[8] |
| 1 2 4 5 3 7 6 4 | Compara A[2] con A[7] | 1 2 3 4 4 6 7 5 | Compara A[6] con A[7] |
| 1 2 4 5 3 7 6 4 | Compara A[2] con A[8] | 1 2 3 4 4 5 7 6 | Compara A[6] con A[8] |
| 1 2 4 5 3 7 6 4 | Compara A[3] con A[4] | 1 2 3 4 4 5 6 7 | Compara A[7] con A[8] |
| | | | FIN. |

Para un vector de 8 elementos el algoritmo insumió 28 comparaciones. Antes de analizar en forma genérica el costo en tiempo del algoritmo veamos ejemplos más simples.

## Costo de una instrucción en ciclos simples

**para** i desde 1 hasta n **hacer**
    Instrucción
**fin_para**

El cálculo en costo de una instrucción en un ciclo simple puede hacerse utilizando una sumatoria.

$$\sum_{i=1}^{n} C1$$

Donde C1 es el costo de la instrucción que se ejecuta en el ciclo. El resultado de la sumatoria es n * C1, lo cual es evidente porque la instrucción dentro del ciclo se ejecuta n veces.

## Costo de una instrucción en ciclos anidados independientes

```
para i desde 1 hasta n hacer
        para j desde 3 hasata m hacer
                instrucción
        fin_para
fin_para
```

Podemos calcular el costo de la instrucción nuevamente usando sumatorias de la forma:

$$\sum_{i=1}^{n}\sum_{j=3}^{m} C1 = \sum_{i=1}^{n} (m-2) * C1 = n * (m-2) * C1$$

# Algoritmos iterativos

## Costo de una instrucción en ciclos anidados dependientes

```
para i desde 1 hasta n hacer
        para j desde i hasata n hacer
                instrucción
        fin_para
fin_para
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C1 = \sum_{i=1}^{n}(n-i+1)*C1 = C1*\sum_{i=1}^{n}(n-i+1) =$$

$$= C1*(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1) = C1*(n*n - \frac{n*(n+1)}{2} + n) = C1*(n^2 - \frac{n^2+n}{2} + n) =$$

$$= C1*(\frac{2n^2 - n^2 - n + 2n}{2}) = C1*\frac{n^2+n}{2}$$

## Sumatorias Útiles

Antes de continuar vamos a recordar algunas sumatorias útiles.

$$\sum_{i=1}^{n} c = c * n$$

$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + 1|$$

## Análisis del algoritmo Uno contra Todos

```
Algoritmo Uno_contra_todos (A,n). Ordena el vector A.
        para i desde 1 hasta (n -1) hacer
                para  j desde (i + 1) hasta n hacer
                        si A[j] < A[i] entonces
                                Swap(A[i], A[j])
                        fin_si
                fin_para
        fin_para
Fin_Algoritmo
```

El costo del algoritmo lo vamos a calcular suponiendo que C1 es el costo de efectuar la comparación y que C2 es el costo de efectuar el SWAP. Sin embargo el SWAP no se hace siempre sino que se hace únicamente cuando la comparación da A[j] > A[i], por ello debemos particionar el análisis en tres casos: el peor caso, el mejor caso y el caso medio.

# Algoritmos iterativos

## Peor caso

En el peor caso el algoritmo siempre hace el Swap. Esto ocurre por ejemplo cuando el vector viene ordenado en el orden inverso al que utilizamos.

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (C1 + C2)$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - (i+1) + 1)$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - i)$$

$$T(n) = (C1 + C2) * (\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

# Algoritmos iterativos

$$T(n) = (C1 + C2) * ((n-1) * n - \frac{(n-1) * n}{2})$$

$$T(n) = (C1 + C2) * (n^2 - n - \frac{n^2 - n}{2})$$

$$T(n) = (C1 + C2) * (\frac{2n^2 - 2n - n^2 + n}{2})$$

$$T(n) = (C1 + C2) * (\frac{n^2 - n}{2})$$

$$T(n) = (C1 + C2) * (\frac{1}{2}n^2 - \frac{1}{2}n)$$

Por ejemplo para N = 8 el peor caso nos da: (C1+C2)*(32–4)=(C1+C2)*28 un total de 28 comparaciones y 28 Swaps.

## Mejor caso

En el mejor caso el vector ya viene ordenado por lo que nunca efectúa ningún Swap, el costo total es entonces el costo de las comparaciones que es igual a lo calculado antes.

$$T(n) = C1 * (\frac{1}{2} n^2 - \frac{1}{2} n)$$

## Caso medio

En el caso medio el algoritmo efectúa todas las comparaciones y solo la mitad de los Swaps.

$$T(n) = C1 * (\frac{1}{2} n^2 - \frac{1}{2} n) + C2 * \frac{1}{2} (\frac{1}{2} n^2 - \frac{1}{2} n)$$

***Importante:*** *Al analizar el tiempo de un algoritmo se debe analizar el mejor caso, el peor caso y el caso medio. Habitualmente el que más interesa es el peor caso.*

# Orden de Algoritmos

## Introducción:

Supongamos que para cierto problema hemos hallado dos posibles algoritmos que lo resuelven, A y B.

Evaluamos con cuidado sus costos respectivos y obtenemos que $TA(n)=100n$ y $TB(n)=2n^2$.

¿Cuál será conveniente usar, basándose en el criterio de eficiencia

# Orden de Algoritmos

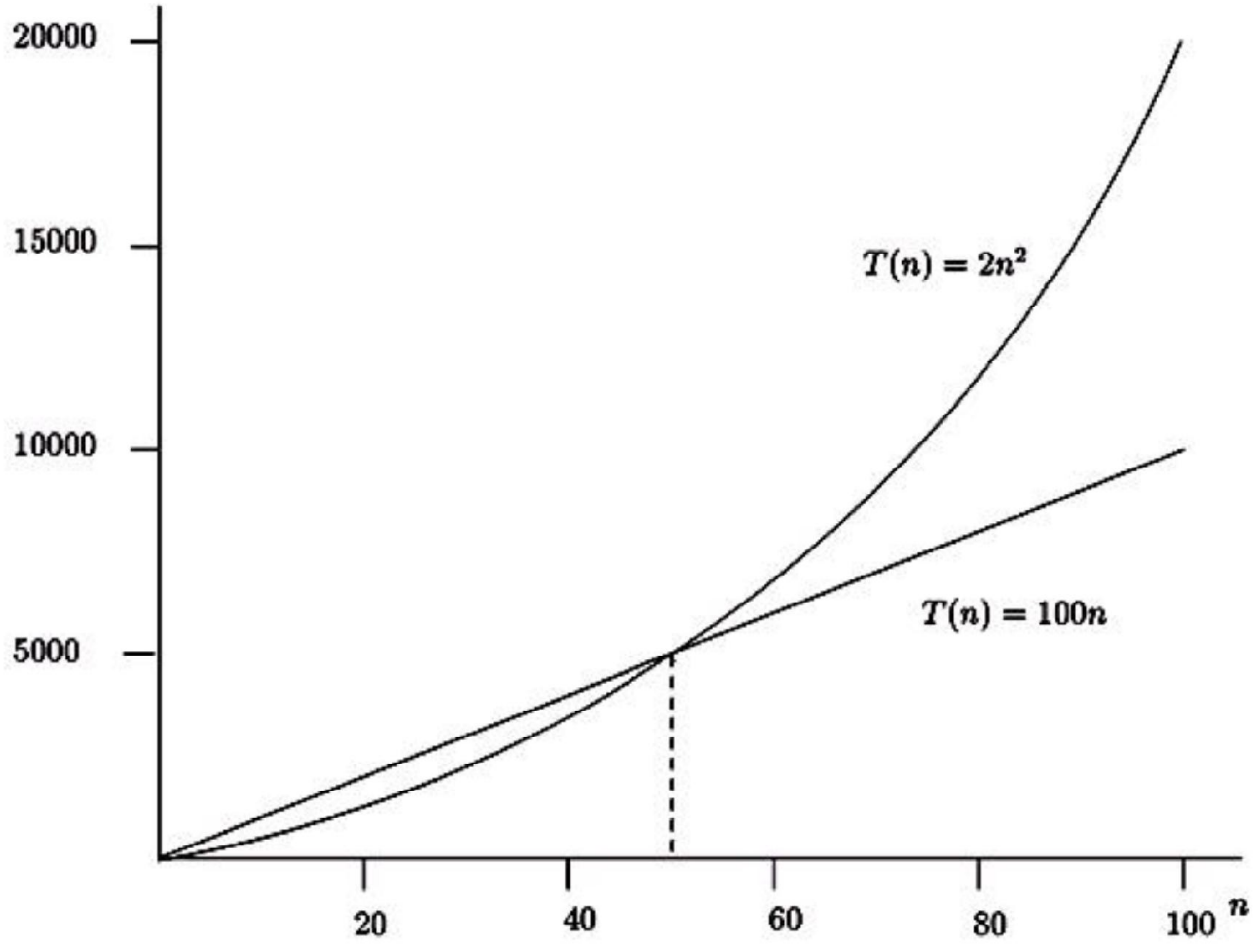


Figura 1: Comparación de los costes $100n$ y $2n^2$

Si n<50, resulta que B es más eficiente que A, pero no mucho más; para n>50 el algoritmo A es mejor que B, y la ventaja de A sobre B se hace mucho mayor cuánto mayor es n. Para entradas de tamaño n=100, A es el doble de rápido que B, pero para entradas de tamaño n=1000, A es veinte veces más rápido que B.

# Orden de Algoritmos

## Notación O

Esta notación sirve para clasificar el crecimiento de las funciones. Así en vez de decir que el costo T(k) del algoritmo de localización del mínimo es T(k)=4k, diremos que es O(k), lo que informalmente significa que el costo es "alguna constate multiplicada por k" para entradas de tamaño k.

# Orden de Algoritmos

## Notación O

Habitualmente no interesa cuánto tarda exactamente un algoritmo sino que nos interesa saber cuál es la tasa de crecimiento del algoritmo en función de los datos de entrada. Para el estudio de algoritmos existe una notación muy práctica y utilizada universalmente en este tipo de problemas conocida como la gran "O" (Big-Oh notation) que define el orden de un algoritmo.

# Orden de Algoritmos

## Definición

***"El orden de un algoritmo se define como la tasa de crecimiento del tiempo que insume el algoritmo en función de la cantidad o tamaño de los datos de entrada".***

# Orden de Algoritmos

## Ordenes más comunes

| Orden | Nombre informal |
|---|---|
| O(log n) | logarítmico |
| O(n) | lineal |
| O(n2) | cuadrático |
| O(n3) | cúbico |
| O(nk) | polinómico |
| O(2n) | exponencial |

# Orden de Algoritmos

## Reglas

• **Regla de las sumas:** Consideremos un algoritmo que consta de k partes compuestas secuencialmente, cada una con un costo $T_i(n)$ y donde $T_i(n) = O(f_i(n))$ para i=1, .., k. Entonces

$$T(n) = T_1(n) + \ldots + T_k(n) = O(max\{ f_1(n), \ldots. f_k(n)\}).$$

• **Reglas de los productos:** Si el costo del cuerpo de una iteración es O(f(n)) y el número de iteraciones es O(g(n)) entonces el costo total de la iteración es O(f(n)*g(n)).

# Orden de Algoritmos

## Regla de las sumas

Supóngase que un algoritmo consta de dos partes. Si la primera tiene coste $O(n^2)$ y la segunda tiene coste $O(n^3)$, entonces el coste del algoritmo sería

$$T(n) = O(n^2) + O(n^3) = O(n^2 + n^3) = O(\max\{n^2, n^3\}) = O(n^3)$$

Para las estructuras iterativas debemos evaluar el costo del cuerpo y el número de iteraciones que se producirán en función del tamaño de la entrada.

# Orden de Algoritmos

## Regla de los productos

Si en un algoritmo aparecen bucles anidados se analizan uno por uno desde el más interno al más externo. Por ejemplo, el siguiente segmento de un algoritmo inicializa las componentes de una matriz n x n a 0.

```
para i desde 1 hasta n hacer
        para j desde 1 hasta n hacer
                C[i, j] : = 0
        fin_para
fin_para
```

Si tomamos como tamaño de la entrada la dimensión de la matriz, n, el costo del segmento completo es $O(n^2)$.

# Orden de Algoritmos

## Algoritmos Recursivos

***"Un algoritmo recursivo es aquel que en algún momento durante su ejecución se invoca a si mismo"***.

Por ejemplo el siguiente programa sirve para calcular el factorial de un número.

**Algoritmo Fact**(n). Calcula el factorial de n en forma recursiva.

```
si n < 2 entonces
        devolver 1
sino
        devolver n*Fact(n-1)
fin_si
```

# Orden de Algoritmos

## Algoritmos Recursivos

• En general el programar en forma recursiva permite escribir menos código y algunas veces nos permite simplificar un problema difícil de solucionar en forma iterativa.

• En cuanto a la eficiencia un programa recursivo puede ser mas, menos o igual de eficiente que un programa iterativo.

# Orden de Algoritmos

## Algoritmos Recursivos

• En primer lugar hay que señalar que los programas recursivos consumen espacio en memoria para la pila ya que cada una de las llamadas recursivas al algoritmo son apiladas una sobre otras para preservar el valor de las variables locales y los parámetros utilizados en cada invocación.

• Este aspecto lo vamos a considerar un costo en espacio ya que lo que estamos consumiendo es memoria.

# Orden de Algoritmos

## Algoritmos Recursivos

• En cuanto al tiempo de ejecución del programa debemos calcularlo de alguna forma para poder obtener el orden del algoritmo.

• Para ello se recurre a plantear el tiempo que insume un programa recursivo de la siguiente forma:

Tiempo de ejecución del factorial

$$T(n) = \begin{cases} 1 & \text{si n < 2} \\ 1 + T(n-1) & \text{sino} \end{cases}$$

# Orden de Algoritmos

## Algoritmos Recursivos

• Las ecuaciones planteadas definen una recurrencia.

• Una recurrencia es un sistema de ecuaciones en donde una o más de las ecuaciones del sistema están definidas en función de si mismas.

• Para calcular el orden de un algoritmo recursivo es necesario resolver la recurrencia que define el algoritmo.

## Recurrencias

Tomemos por ejemplo la siguiente recurrencia muy común en algoritmos recursivos.

$$T(n) = \begin{cases} 1 \\ 2T(\frac{n}{2}) + n \end{cases}$$

Estudiaremos tres técnicas que suelen utilizarse para resolver recurrencias.

- El método de sustitución
- El método iterativo
- El teorema maestro de las recurrencias.

## El método de sustitución

En método de sustitución se utiliza cuando estamos en condiciones de suponer que el resultado de la recurrencia es conocido.

En estos casos lo que hacemos es sustituir la solución en la recurrencia y luego demostrar que el resultado es valido por inducción.

Para nuestro ejemplo sabemos que

$$T(n) = (n \log n) + n$$

## El método de sustitución

Veamos que pasa cuando n=1

Si n=1 => T(1)=1

1 log 1 + 1 = 0 + 1 = 1 Verifica

Hipótesis inductiva

si n>1 => T(n')=(n' log n' )+n' si n'<n

## El método de sustitución

Demostración

$$T(n) = 2T(\frac{n}{2}) + n \qquad (1)$$

Como $\frac{n}{2} < n$

Por inducción

$$T(\frac{n}{2}) = (\frac{n}{2}) * \log(\frac{n}{2}) + \frac{n}{2} \qquad (2)$$

## El método de sustitución

Reemplazando (2) en (1)

$$T(n) = 2((\frac{n}{2}) * \log(\frac{n}{2}) + \frac{n}{2}) + n$$

$$T(n) = (n\log(\frac{n}{2}) + n) + n$$

$$T(n) = n(\log n - \log 2) + 2n$$

$$T(n) = n\log n - n + 2n$$

$$T(n) = n\log n + n$$

$$T(n) = \Theta(n\log n)$$

Por lo que queda demostrado.

# Algoritmos Recursivos

## El método de sustitución

- El método de sustitución es simple.

- Requiere que uno ya conozca cual es la solución de la recurrencia.

- A veces sin embargo en recurrencias muy complejas para resolver por otros métodos puede llegar a ser conveniente intentar adivinar la solución y luego demostrarla usando este método.

## El método iterativo

• El método iterativo es un forma bastante poderosa de resolver una recurrencia sin conocer previamente el resultado.

• En este método se itera sobre la recurrencia hasta que se deduce un cierto patrón que permite escribir la recurrencia usando sumatorias.

• Luego realizando una sustitución apropiada y resolviendo las sumatorias se llega al resultado de la recurrencia.

## El método iterativo

Utilizando el método iterativo en nuestro ejemplo observamos lo siguiente:

$$T(1) = 1$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$T(n) = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + n + n$$

$$T(n) = 4(2T(\frac{n}{8}) + \frac{n}{4}) + n + n = 8T(\frac{n}{8}) + n + n + n$$

$$T(n) = 2^k * T(\frac{n}{2^k}) + kn$$

## El método iterativo

Como T(1)=1 hacemos $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

Reemplazando

$$T(n) = 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + n \log_2 n$$

$$Sabemos\ que: 2^{\log_2 n} = n^{\log_2 2} = n^1 = n$$

$$T(n) = nT(\frac{n}{n}) + n \log_2 n = nT(1) + n \log_2 n$$

$$T(n) = n + n \log_2 n$$

$$T(n) = \Theta(n \log n)$$

## El teorema maestro de las recurrencias

Mediante este teorema podemos contar con una poderosa herramienta para resolver algunas recurrencias. El teorema dice lo siguiente.

**Teorema:** Sea:

$$T(n) = aT(\frac{n}{b}) + n^k$$

Si $a \geq 1$ y $b > 1$

Entonces

- Caso 1 Si $a > b^k \Rightarrow T(n) = \in \Theta(n^{\log_b a})$
- Caso 2 Si $a = b^k \Rightarrow T(n) = \in \Theta(n^k \log n)$
- Caso 3 Si $a < b^k \Rightarrow T(n) = \in \Theta(n^k)$

## El teorema maestro de las recurrencias

Por ejemplo para

$$T(n) = 2T(\frac{n}{2}) + n$$

tenemos que:
a = 2, b = 2, k = 1

luego $a=b^k$ y estamos en el caso 2. Por lo que el algoritmo es $O(n^1 \log n)$

Lamentablemente no todas las recurrencias tienen la forma que requiere el teorema maestro por lo que a veces no queda más remedio que aplicar el método iterativo.

## Cambio de variables

En algunas recurrencias puede ser conveniente realizar un cambio de variables de forma tal de resolver una recurrencia ya conocida, luego aplicando la inversa del cambio podemos obtener la solución de la recurrencia original.

Supongamos que tenemos la siguiente recurrencia:

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

## Cambio de variables

Sea m log n. Entonces $n=2^m$. Reemplazando $2^m$ por n en la recurrencia de T(n) tenemos:

$$T(2^m) = T(2^{m/2}) + 1$$

Llamamos $S(m) = T(2^m)$. Entonces la recurrencia queda:

$$S(m) = S(m/2) + 1$$

Usando el teorema maestro sabemos que S(m)=O(log m). Por lo tanto

$$T(n) = T(2^m) = S(m) = O(lgm)$$
$$T(n) = O(\log \log n)$$