

Algoritmos y Estructuras de Datos

Cursada 2015

Prof. Alejandra Schiavoni

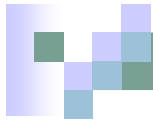
Prof. Catalina Mostaccio

Facultad de Informática – UNLP



Agenda

- Análisis de algoritmos
- Algoritmos recursivos vs.
Iterativos
- Optimizando algoritmos



Agenda

Análisis de algoritmos

- Introducción al concepto $T(n)$
 - ✓ Tiempo, entrada, peor caso, etc.
- Notación Big-Oh
 - ✓ Definición y ejemplos
 - ✓ Reglas (suma, producto)
- Cálculo del $T(n)$
 - ✓ En algoritmos iterativos y recursivos



Análisis de algoritmos

- *Nos permite comparar algoritmos en forma independiente de una plataforma en particular*
- *Mide la eficiencia de un algoritmo, dependiendo del tamaño de la entrada*

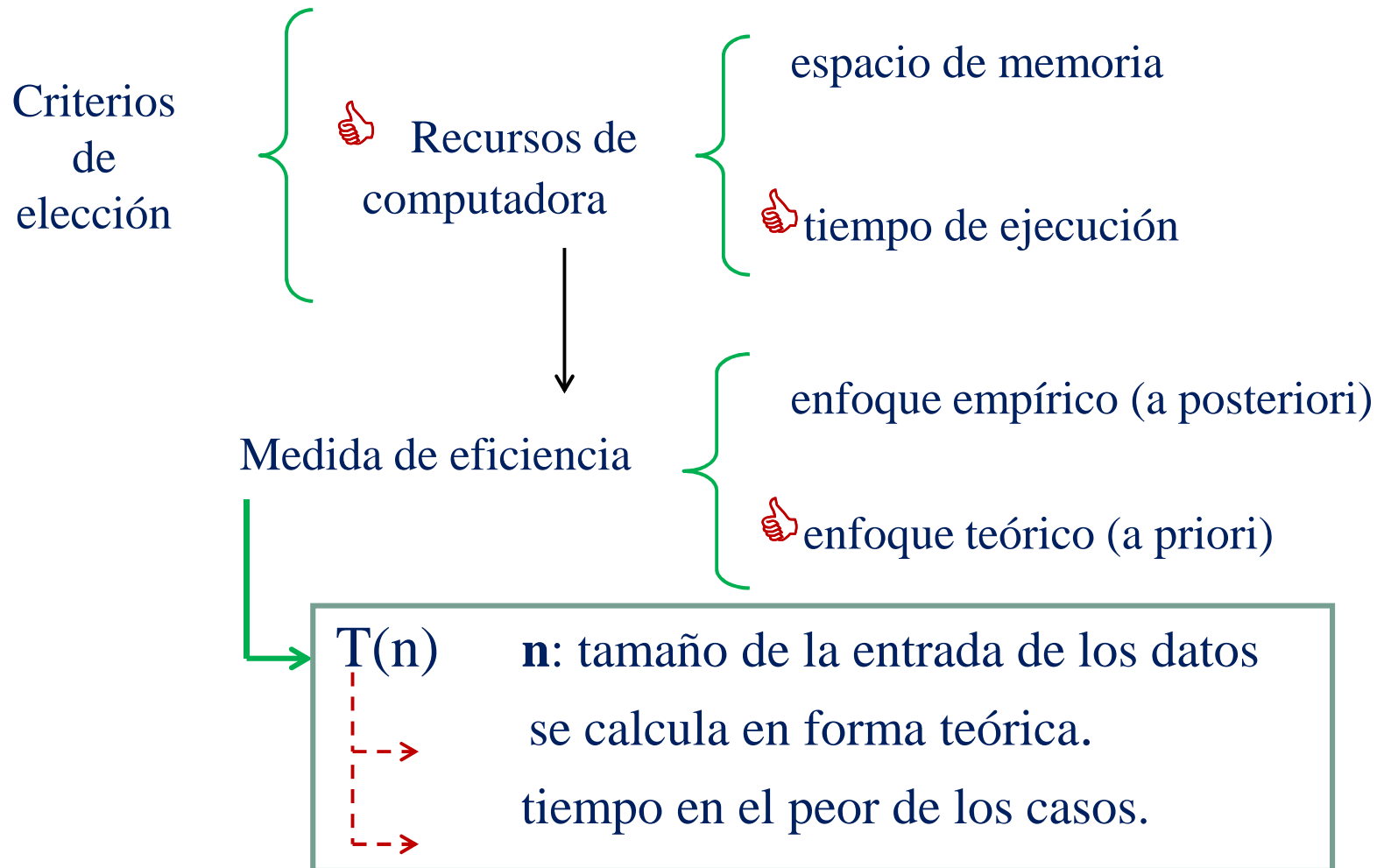


Análisis de algoritmos

Pasos a seguir:

- Caracterizar los datos de entrada del algoritmo
- Identificar las operaciones abstractas, sobre las que se basa el algoritmo
- Realizar un análisis matemático, para encontrar los valores de las cantidades del punto anterior

Introducción al concepto $T(n)$





Definiciones

➤ **Big-Oh**

➤ **Omega**

➤ **Theta**



Notación Big-Oh

- **Definición y ejemplos**
- **Regla de la suma y regla del producto**



Notación Big-Oh

Definición

Decimos que

$$T(n) = O(f(n))$$

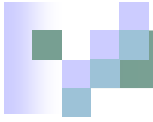
si existen constantes $c > 0$ y n_0 tales que:

$$T(n) \leq c f(n) \quad \text{para todo } n \geq n_0$$

Se lee: $T(n)$ es de orden de $f(n)$

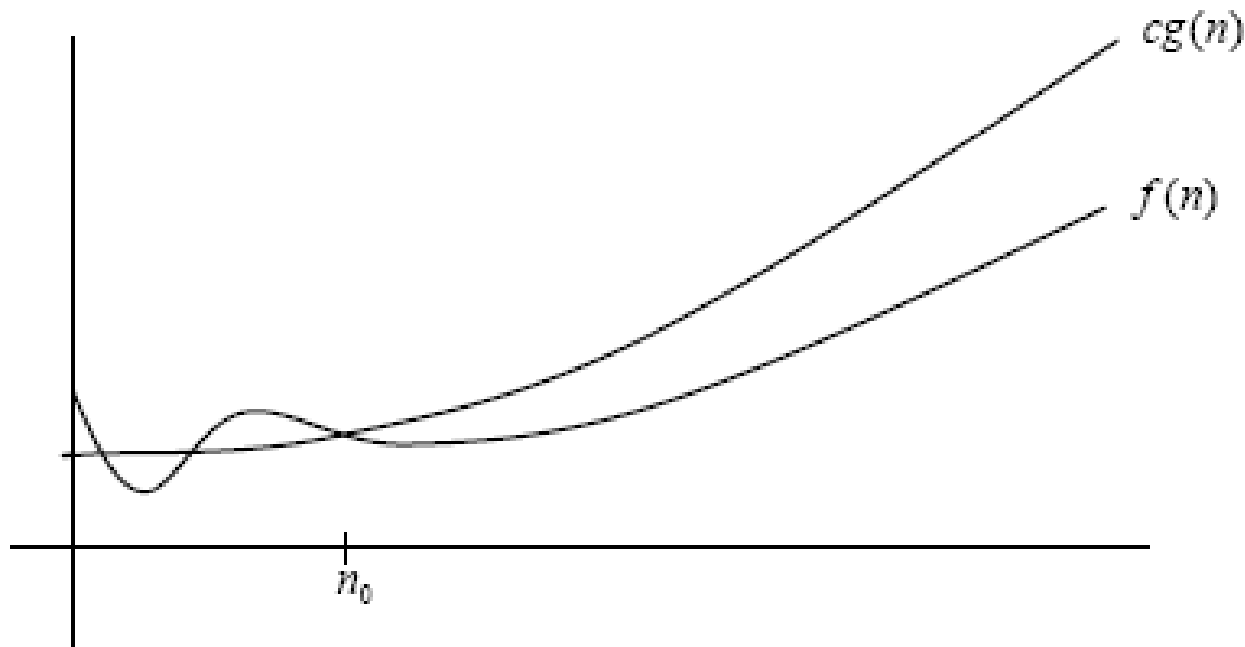
$f(n)$ representa una cota superior de $T(n)$

La tasa de crecimiento de $T(n)$ es menor o igual que la de $f(n)$



Notación Big-Oh

Geométricamente $f(n) = O(g(n))$ es:





Notación Big-Oh

Ejemplos

1.- $T(n) = 3n^3 + 2n^2$ es $O(n^3)$? **Verdadero**

2.- $T(n) = 3n^3 + 2n^2$ es $O(n^4)$? **Verdadero**

3.- $T(n) = 1000$ es $O(1)$? **Verdadero**

4.- $T(n) = 3^n$ es $O(2^n)$? **Falso**



Notación Big-Oh

- Regla de la suma y regla del producto

Si $T_1(n)=O(f(n))$ y $T_2(n)=O(g(n))$, entonces:

1. $T_1(n)+T_2(n)=\max(O(f(n)),O(g(n)))$
2. $T_1(n) \cdot T_2(n)=O(f(n) \cdot g(n))$



Notación Big-Oh

➤ Otras reglas :

- $T(n)$ es un polinomio de grado $k \Rightarrow T(n) = O(n^k)$
- $T(n) = \log^k(n) \Rightarrow O(n)$ para cualquier k
 n siempre crece más rápido que cualquier potencia de $\log(n)$
- $T(n) = \text{cte} \Rightarrow O(1)$
- $T(n) = \text{cte} * f(n) \Rightarrow T(n) = O(f(n))$



Omega

Definición

Decimos que

$$T(n) = \Omega(g(n))$$

si existen constantes $c > 0$ y n_0 tales que:

$$T(n) \geq c g(n) \quad \text{para todo } n \geq n_0$$

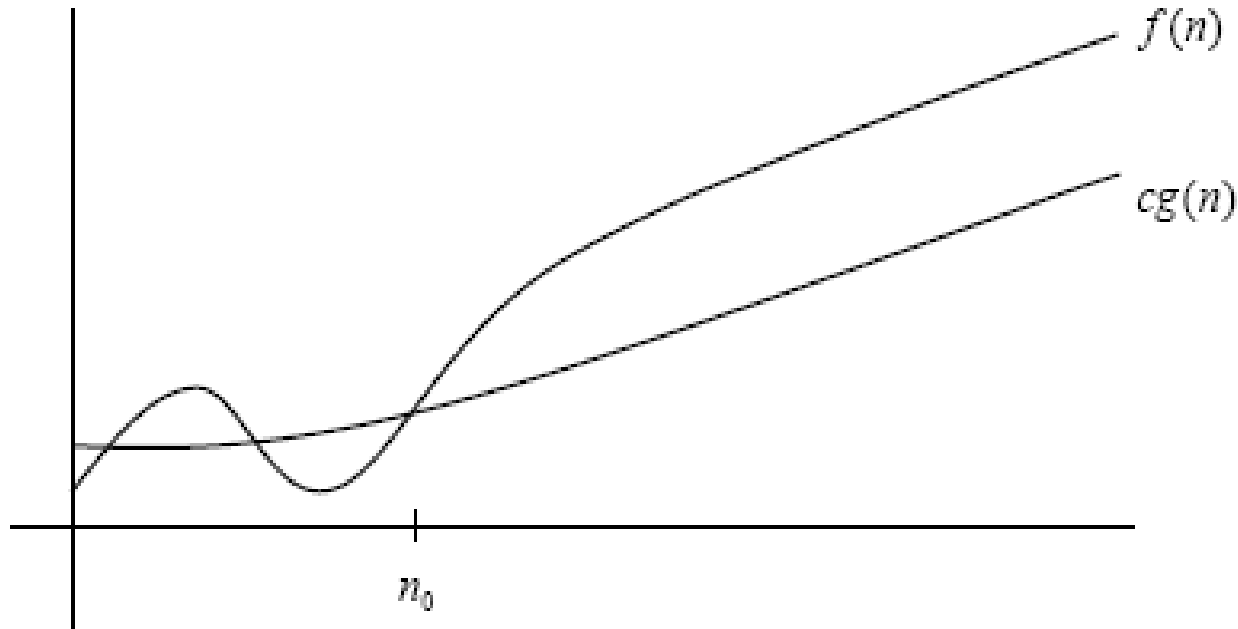
Se lee: $T(n)$ es omega de $g(n)$

$g(n)$ representa una cota inferior de $T(n)$

La tasa de crecimiento de $T(n)$ es mayor o igual que la de $g(n)$

Omega

Geométricamente $f(n) = \Omega(g(n))$ es:





Theta

Definición

Decimos que

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

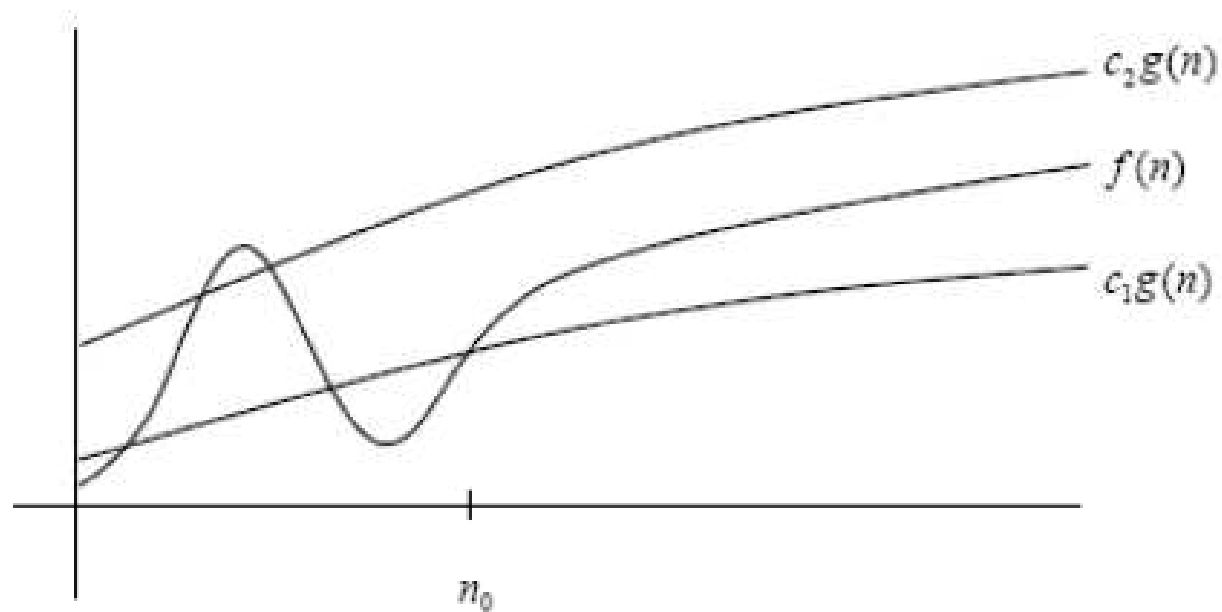
$$\leftrightarrow T(n) = O(h(n)) \text{ y } T(n) = \Omega(h(n))$$

Se lee: $T(n)$ es theta de $h(n)$

$T(n)$ y $h(n)$ tienen la misma tasa de crecimiento

Theta

Geométricamente $f(n) = \Theta(g(n))$ es:

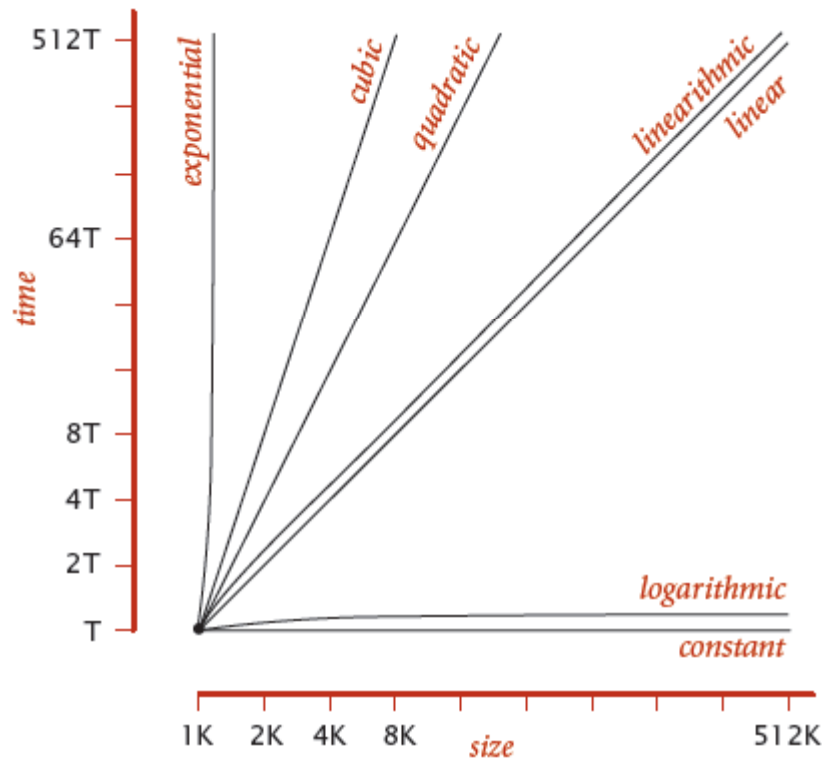




Algunas funciones

Ordenadas en forma creciente	Nombre
1	Constante
$\log n$	Logaritmo
n	Lineal
$n \log n$	$n \log n$
n^2	Cuadrática
n^3	Cúbica
$c^n \ c > 1$	Exponencial

Algunas funciones



Este conjunto de funciones es suficiente para describir la tasa de crecimiento de los algoritmos típicos

Cuadro comparativo del tiempo para diferentes funciones

Costo		$n=10^3$	Tiempo	$n=10^6$	Tiempo
Logarítmico	$\log_2(n)$	10	10 segundos	20	20 segundos
Lineal	n	10^3	16 minutos	10^6	11 días
Cuadrático	n^2	10^6	11 días	10^{12}	30.000 años

Orden de ejecución del algoritmo

Cantidad de operaciones

Tiempo total del algoritmo

Cantidad de operaciones

Tiempo total del algoritmo

$n = 10^3$


Algoritmos y Estructuras de Datos
2015

$n = 10^6$



Cálculo del Tiempo de Ejecución

Estructuras
de
Control

- 
- Secuencia
 - Condicional :
 - *if/else*
 - *switch*
 - Iteración :
 - *for*
 - *while*
 - *do-while*



Cálculo del Tiempo de Ejecución

➤ Condicional :

a) *if (boolean expression) {*
 statement(s)
 }

b) *if (boolean expression) {*
 statement(s)
 } else {
 statement(s)
 }



Cálculo del Tiempo de Ejecución

➤ Condicional :

c) **switch** (*integer expression*) {
 case *integer expression* : *statement(s)* ; **break**;
 ...
 case *integer expression* : *statement(s)* ; **break**;
 default : *statement(s)* ; **break**;
}



Cálculo del Tiempo de Ejecución

➤ Iteración :

a) **for** (*initialization; termination; increment*) {
 statement(s)
}

b) **while** (*boolean expression*) {
 statement(s)
}

c) **do** {
 statement(s)
} **while** (*boolean expression*);

Cálculo del Tiempo de Ejecución

➤ Iteración :

a) **For**

```
int sum = 0;  
int [] a = new int [n];  
for (int i = 1; i <= n ; i++ )  
    sum += a[i];
```

Viene como
parámetro



$$\begin{aligned} T(n) &= cte_1 + \sum_{i=1}^n cte_2 = \\ &= cte_1 + n * cte_2 \\ &\Rightarrow O(n) \end{aligned}$$

Cálculo del Tiempo de Ejecución

a) **For**

```
int sum = 0;  
int [] a = new int [n][n];  
for (int i = 1; i <= n ; i++) {  
    for (int j = 1; j <= n ; j++)  
        sum += a[i][j];  
}
```

Viene como
parámetro



$$T(n) = cte_1 + \sum_{i=1}^n \sum_{j=1}^n cte_2 =$$
$$= cte_1 + n * n * cte_2$$
$$\Rightarrow O(n^2)$$

Cálculo del Tiempo de Ejecución

a) For

```
int [] a = new int [n];  
int [] s = new int [n];  
for ( int i = 1; i <= n ; i++ )  
    s[i] = 0;  
for ( int i = 1; i <= n ; i++ ) {  
    for (int j = 1; j <= i ; j++)  
        s[i] += a[j];  
}
```

Viene como
parámetro

$$\begin{aligned} T(n) &= cte_1 + \sum_{i=1}^n cte_2 + \\ &+ \sum_{i=1}^n \sum_{j=1}^i cte_3 = \\ &= cte_1 + n * cte_2 + \\ &cte_3 * \sum_{i=1}^n i = \dots \end{aligned}$$

Cálculo del Tiempo de Ejecución

➤ Iteración :

b) While

int x= 0;

int i = 1;

while (i <= n) {

x = x + 1;

i = i + 2;

}

$$T(n) = cte_1 + \sum_{i=1}^{(n+1)/2} cte_2 =$$

$$= cte_1 + cte_2/2 * (n+1)$$

$$\Rightarrow O(n)$$



Cálculo del Tiempo de Ejecución

➤ Iteración :

b) While

int x= 1;

while (x < n)

*x = 2 *x;*

$$T(n) = cte_1 + cte_2 * \log(n)$$

$$\Rightarrow O(\log(n))$$



Ejercicios para analizar (I)

Considerando que un algoritmo requiere $f(n)$ operaciones para resolver un problema y la computadora procesa 100 operaciones por segundo.

Si $f(n)$ es:

a.- $\log_{10} n$

b.- \sqrt{n}

Determine el tiempo en segundos requerido por el algoritmo para resolver un problema de tamaño $n=10000$.



Ejercicios para analizar (II)

Suponga que Ud. tiene un algoritmo ALGO-1 con un tiempo de ejecución exacto de $10n^2$. ¿En cuánto se hace más lento ALGO-1 cuando el tamaño de la entrada n aumenta:.....?

- a.- El doble
- b.- El triple



Ejercicios para analizar (III)

```
private int ejercicio3(int n) {  
    int p=0; int j=1;  
  
    for (int i=1; i<=n; i++)  
        if (esImpar(i))  
            j:=j*2;  
        else  
            for (int k=1; k<=j; k++)  
                p:= p+1;  
  
    return p;  
}
```

```
public boolean esImpar(int unNumero){  
    if (unNumero%2 != 0)  
        return true;  
    else  
        return false;  
}
```




Cálculo del Tiempo de Ejecución

Ejemplo:

```
private void imparesypares(int n){
    int x=0; int y=0;

    for (int i=1;i<=n;i++)
        if (esImpar(i))
            for (int j=i;j<=n;j++)
                x++;
        else
            for (int j=1;j<=i;j++)
                y++;
}
```

```
public boolean esImpar(int unNumero){
    if (unNumero%2 != 0)
        return true;
    else
        return false;
}
```

Cálculo del Tiempo de Ejecución

Ejemplo (cont.):

Desarrollo de la función $T(n)$ del método *imparesypares*

- Asumiendo valor de “n” par.
- El método *esImpar* tiene todas sentencias constantes

$$T_{esImpar}(n) = cte1$$

- El método *imparesypares* tiene un loop en el que: en cada iteración se llama al método *esImpar* y la mitad de las veces se ejecuta uno de los *for* (para valores de “i” impares) y la mitad restante el otro *for* (para valores de “i” pares)

$$T(n) = \sum_{i=1}^n cte1 + \sum_{i=1}^{n[paso\ 2]} \left(\sum_{j=i}^n cte2 + \sum_{j=1}^{i+1} cte2 \right)$$

Valores pares dados por el siguiente a los impares “i”

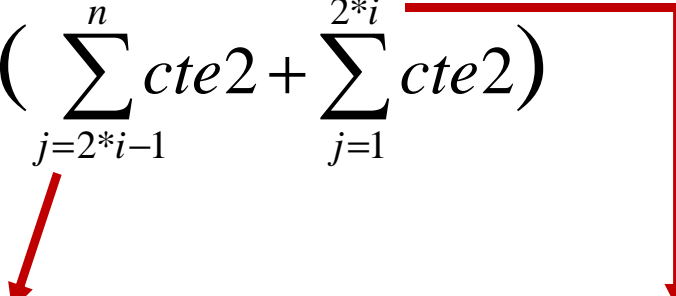
Es la llamada al método *esImpar*, que se ejecuta para todos los valores de “i”

Valores de “j” impares

Cálculo del Tiempo de Ejecución

Ejemplo (cont.):

Desarrollo de la función $T(n)$ del método *imparesypares*

$$T(n) = \sum_{i=1}^n cte1 + \sum_{i=1}^{n/2} \left(\sum_{j=2*i-1}^n cte2 + \sum_{j=1}^{2*i} cte2 \right)$$


Como “ i ” ahora toma valores consecutivos entre 1 y $n/2$, entonces se hace un cambio de variable para seguir tomándose valores impares y pares en cada loop



Cálculo del Tiempo de Ejecución

Ejemplo (cont.):

Resolviendo la función $T(n)$

$$T(n) = \sum_{i=1}^n cte1 + \sum_{i=1}^{n/2} \left(\sum_{j=2*i-1}^n cte2 + \sum_{j=1}^{2*i} cte2 \right)$$

$$\begin{aligned} T(n) &= cte1 * n + \sum_{i=1}^{n/2} cte2 * (n - 2*i + 1 + 1 + 2*i - 1 + 1) = \\ &= cte1 * n + cte2 * (n + 2) * n / 2 \\ &= cte1 * n + cte2 / 2 * n^2 + cte2 * n \end{aligned}$$

$$T(n) = O(n^2)$$



Ejercicio para analizar

```
0      i = 0; j = 0;
1      while(i<1000)
2          for( int k = i; k <= n; k++ ) {
3              i++;
4              j++; }
5      for( int p = 0; p < n*n; p++ )
6          for( int q = 0; q < p; q++ )
7              j--;
```

1. ¿Con qué valor termina la variable i ?
2. ¿Cuántas veces se ejecuta la sentencia 3?
 - a. $O(n)$
 - b. $O(n^2)$
 - c. $O(n^3)$
 - d. $O(n^4)$
 - e. Ninguna de las anteriores



Cálculo del Tiempo de Ejecución Algoritmos Recursivos

*/***

Calcula el Factorial.

**/*

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



Cálculo del Tiempo de Ejecución

Función de recurrencia

Factorial (n)

$$T(n) = \begin{cases} \text{cte}_1 & n = 1 \\ \text{cte}_2 + T(n - 1) & n > 1 \end{cases}$$



Cálculo del Tiempo de Ejecución

Función de recurrencia

Factorial (n) - Desarrollo de la recurrencia

$$T(n) = \underbrace{T(n - 1)} + \text{cte}_2 \quad n > 1$$

$$\underbrace{T(n - 2)} + \text{cte}_2$$

$$\underbrace{T(n - 3)} + \text{cte}_2$$

$$T(n - 4) + \text{cte}_2$$



Cálculo del Tiempo de Ejecución

Función de recurrencia

Factorial (n) - Desarrollo de la recurrencia

$$\begin{aligned} T(n) &= T(n - 1) + \text{cte}_2 = (T(n - 2) + \text{cte}_2) + \text{cte}_2 = \\ &= T(n - 2) + 2 \text{cte}_2 = (T(n - 3) + \text{cte}_2) + 2 \text{cte}_2 = \\ &= T(n - 3) + 3 \text{cte}_2 = \dots\dots\dots \end{aligned}$$

Paso i:

$$T(n) = T(n - i) + i * \text{cte}_2$$

El desarrollo va terminar en el caso base de la recurrencia,
cuando **$n-i = 1$**



Cálculo del Tiempo de Ejecución

Función de recurrencia

Factorial (n) - Desarrollo de la recurrencia

$$T(n) = T(n - i) + i * cte_2$$

Cuando $n-i = 1$ \longrightarrow $i = n - 1$, reemplazamos i en la expresión de $T(n)$

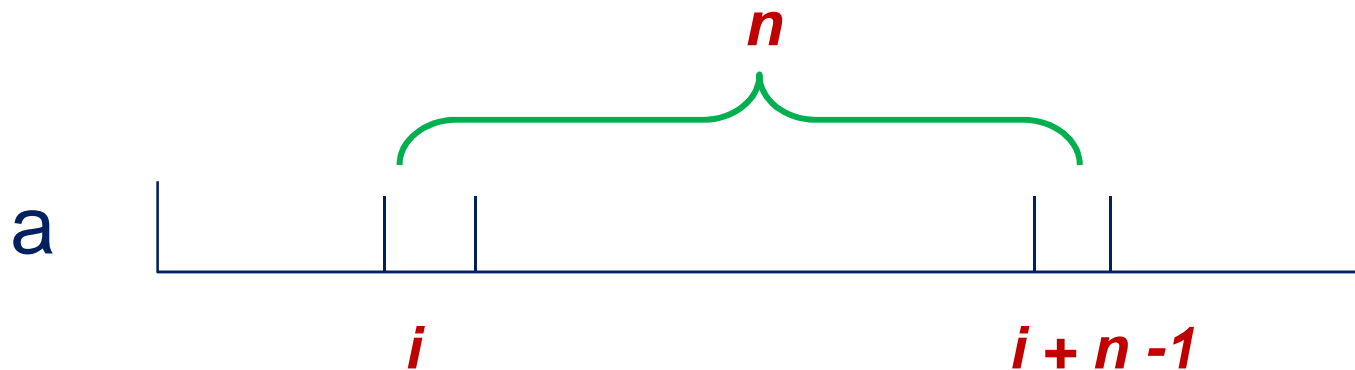
$$T(n) = \underbrace{T(n - (n-1))}_{T(1) = cte_1} + (n-1) * cte_2 = cte_1 + (n-1) * cte_2 = O(n)$$

Cálculo del Tiempo de Ejecución

Algoritmos Recursivos

➤ Ejemplo :

Encontrar el máximo elemento en un arreglo de enteros tomando n posiciones a partir de la posición i





Cálculo del Tiempo de Ejecución Algoritmos Recursivos

```
/** Calcula el Máximo en un arreglo. */  
public static int max( int [] a, int i, int n ) {  
    int m1; int m2;  
    if (n == 1)  
        return a[i];  
    else {    m1 = max (a, i, n/2);  
             m2 = max (a, i + (n/2), n/2);  
             if (m1 < m2)  
                 return m2;  
             else return m1;  
    }  
}
```



Cálculo del Tiempo de Ejecución

Función de recurrencia

Máximo en un arreglo

$$T(n) = \begin{cases} \text{cte}_1 & n = 1 \\ 2 * T(n/2) + \text{cte}_2 & n > 1 \end{cases}$$



Cálculo del Tiempo de Ejecución

Función de recurrencia

Máximo en un arreglo – Desarrollo de la recurrencia

$$T(n) = 2 * \underbrace{T(n/2)} + \text{cte}_2 \quad n > 1$$

$$2 * \underbrace{T(n/4)} + \text{cte}_2$$

$$2 * \underbrace{T(n/8)} + \text{cte}_2$$

$$2 * T(n/16) + \text{cte}_2$$



Cálculo del Tiempo de Ejecución

Función de recurrencia

Máximo en un arreglo – Desarrollo de la recurrencia

$$\begin{aligned}T(n) &= 2 * T(n/2) + cte_2 = 2 * [2 * T(n/4) + cte_2] + cte_2 = \\&= 4 * T(n/4) + 3 cte_2 = 4 * [2 * T(n/8) + cte_2] + 3 cte_2 = \\&= 8 * T(n/8) + 7 cte_2 = 8 * [2 * T(n/16) + cte_2] + 7 cte_2 = \\&= 16 * T(n/16) + 15 cte_2 = \dots\dots\end{aligned}$$

Paso i:

$$T(n) = 2^i * T(n/2^i) + (2^i - 1) * cte_2$$

El desarrollo va terminar en el caso base de la recurrencia, cuando $n/2^i = 1$



Cálculo del Tiempo de Ejecución

Función de recurrencia

Máximo en un arreglo – Desarrollo de la recurrencia

$$T(n) = 2^i * T(n/2^i) + (2^i - 1) * cte_2$$

Cuando $n/2^i = 1 \longrightarrow n = 2^i \longrightarrow i = \log_2 n$,
reemplazamos i en la expresión de $T(n)$

$$T(n) = n * \underbrace{T(n/n)}_{T(1)} + (n - 1) * cte_2 = n * cte_1 + (n - 1) * cte_2 = O(n)$$

$$T(1) = cte_1$$



Cálculo del Tiempo de Ejecución

Algoritmos recursivos vs. iterativos



Números de Fibonacci – Introducción

Describir el Problema de los Conejos

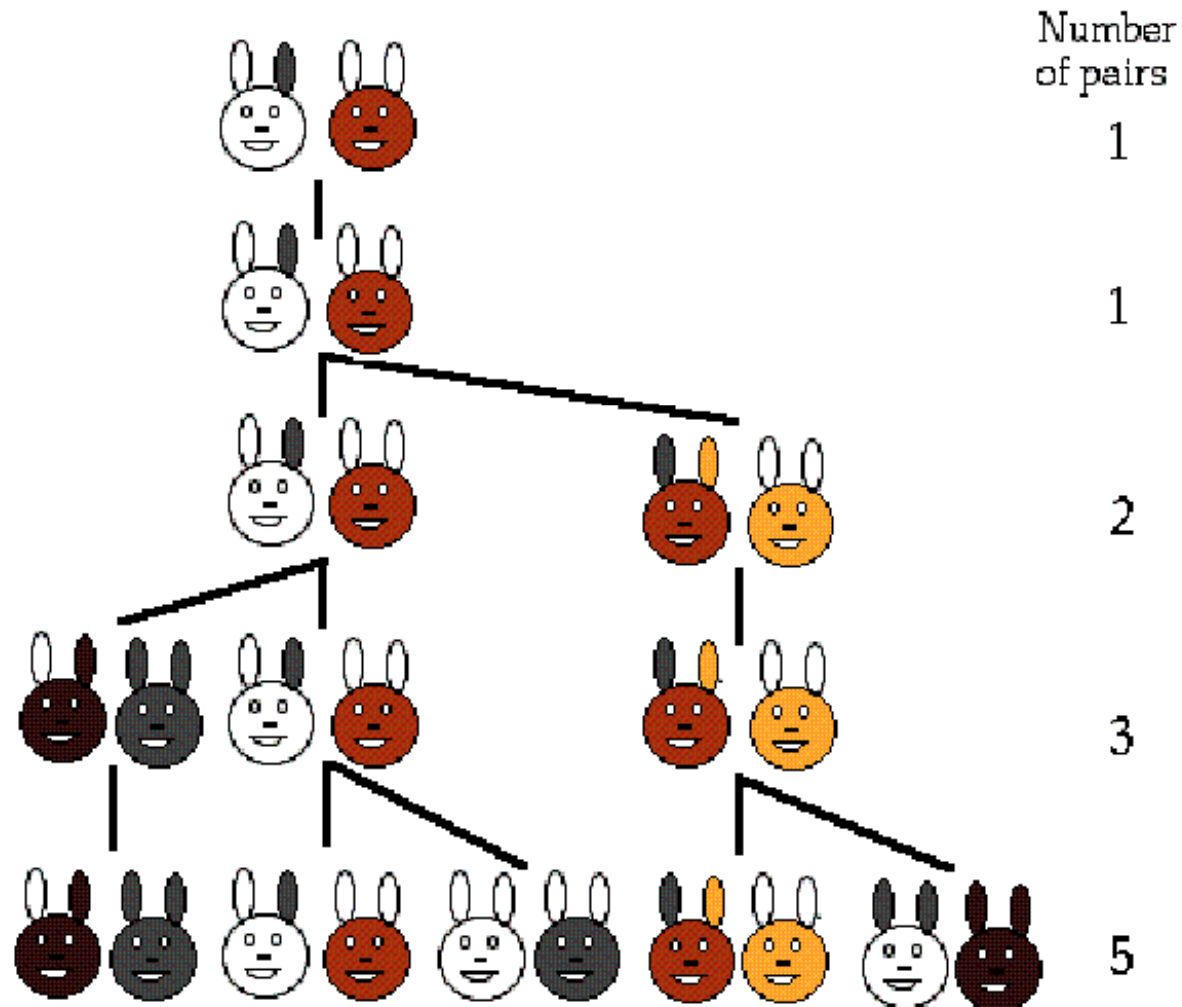
El ejercicio de Fibonacci (1202) pregunta cuántas parejas de conejos habrá en una granja luego de 12 meses, si se coloca inicialmente una sola pareja y se parte de las siguientes premisas:

1. Los conejos alcanzan la madurez sexual a la edad de un mes.
2. En cuanto alcanzan la madurez sexual los conejos se aparean y siempre resulta preñada la hembra.
3. El periodo de gestación de los conejos es de un mes.
4. Los conejos no mueren.
5. La hembra siempre da a luz una pareja de conejos de sexos opuestos.
6. Los conejos tienen una moral y un instinto de variedad genética muy relajados y se aparean entre parientes.

El proceso de crecimiento de la población de conejos es mejor descrito con la siguiente ilustración.

Números de Fibonacci – Introducción

Problema de los Conejos





Números de Fibonacci – Introducción

Problema de los Conejos

Como se puede observar el número de parejas de conejos por mes está determinado por la sucesión de Fibonacci. Así que la respuesta al ejercicio del *Liber Abaci* (*libro acerca del Ábaco*), acerca de cuántas parejas de conejos habrá luego de un año, resulta ser el doceavo término de la sucesión: 144.



Números de Fibonacci – versión recursiva

```
/**  Cálculo de los números de Fibonacci
 */

public static int fib( int n )
{
    if (n <= 1)
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
/* End */
```



Números de Fibonacci – versión iterativa

```
/**    Cálculo de los números de Fibonacci    */  
  
public static int fibonacci( int n )    {  
    if (n <= 1)  
        return 1;  
  
    int ultimo = 1;  
    int anteUltimo = 1;  
    int resul= 1;  
    for( int i = 2; i <= n; i++ )  
    {  
        resul = ultimo + anteUltimo;  
        anteUltimo = ultimo;  
        ultimo = resul;  
    }  
    return resul;  
}
```



Ejercicio recursivo para analizar

```
private int recursivo(int n) {  
    final int m=....;  
    int suma;  
  
    if (n <= 1)  
        return (1);  
    else {  
        suma=0;  
        for (int i=1; i<=m; i++)  
            suma=suma + i;  
        for (i=1; i<=m; i++)  
            suma=suma + recursivo(n-1);  
        return suma;  
    }  
}
```



Función de recurrencia

$$T(n) = \begin{cases} \text{cte}_1 & n \leq 1 \\ \text{cte}_2 + m + m * T(n-1) & n > 1 \end{cases}$$



Cálculo del Tiempo de Ejecución

Optimizando algoritmos

Problema: encontrar el valor de la suma de la sub-secuencia de suma máxima



Problema de la subsecuencia de suma máxima


Dada una secuencia de números enteros, algunos negativos:

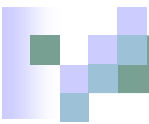
$$a_1, a_2, a_3, \dots, a_n$$

encontrar el valor máximo de la $\sum_{k=i}^j a_k$

Por convención, la suma es cero cuando todos los enteros son negativos.

Ej.: -2, 11, -4, 13, -5, -2 la respuesta es 20





Suma de la sub-secuencia de suma máxima

Versión 1 : $O(n^3)$

```
public final class MaxSumTest
{
    /* Cubic maximum contiguous subsequence sum algorithm.
    */
    public static int maxSubSum1( int [ ] a )
    {
        int maxSum = 0;
        for( int i = 0; i < a.length; i++ )
            for( int j = i; j < a.length; j++ )
            {
                int thisSum = 0;
                for( int k = i; k <= j; k++ )
                    thisSum += a[ k ];
                if( thisSum > maxSum )
                    maxSum = thisSum;
            }
        return maxSum;
    } /* END */
}
```

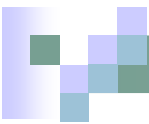
Suma de la sub-secuencia de suma máxima

Versión 2 : $O(n^2)$

```
public final class MaxSumTest
{
    /* Quadratic maximum contiguous subsequence sum algorithm.
    */
    public static int maxSubSum1( int [ ] a )
    {
        int maxSum = 0;
        for( int i = 0; i < a.length; i++ )
            for( int j = i; j < a.length; j++ )
            {
                int thisSum = 0;
                for( int k = i; k <= j; k++ )
                    thisSum += a[ k ];
                if( thisSum > maxSum )
                    maxSum = thisSum;
            }
        return maxSum;
    } /* END */
}
```

int thisSum = 0;

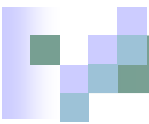
thisSum += a[j];



Suma de la sub-secuencia de suma máxima

Versión 2 : $O(n^2)$

```
public final class MaxSumTest
{
    /* Quadratic maximum contiguous subsequence sum
    algorithm.  */
    public static int maxSubSum2( int [ ] a )
    {
        int maxSum = 0;
        for( int i = 0; i < a.length; i++ )
            int thisSum = 0;
            for( int j = i; j < a.length; j++ )
            {
                thisSum += a[ j ];
                if( thisSum > maxSum )
                    maxSum = thisSum;
            }
        return maxSum;
    } /* END */
}
```

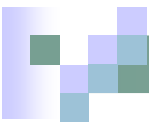


Suma de la sub-secuencia de suma máxima

Versión 3 : $O(n \cdot \log n)$

/ Solución recursiva:**

*** Explicar la resolución detallada y graficamente */**

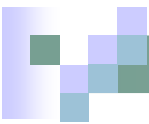


Suma de la sub-secuencia de suma máxima

Versión 3 : $O(n \cdot \log n)$

```
/** Recursive maximum contiguous subsequence sum algorithm.  
 * Finds maximum sum in subarray spanning a[left..right].  
 * Does not attempt to maintain actual best sequence. */
```

```
private static int maxSumRec( int [ ] a, int left, int right )  
{  
    if( left == right ) // Base case  
        if( a[ left ] > 0 )  
            return a[ left ];  
    else  
        return 0;  
    int center = ( left + right ) / 2;  
    int maxLeftSum = maxSumRec( a, left, center );  
    int maxRightSum = maxSumRec( a, center + 1, right );
```

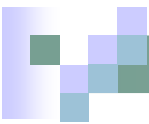


Suma de la sub-secuencia de suma máxima

Versión 3 : $O(n \cdot \log n)$

```
int maxLeftBorderSum = 0, leftBorderSum = 0;
for( int i = center; i >= left; i-- )
{
    leftBorderSum += a[ i ];
    if( leftBorderSum > maxLeftBorderSum )
        maxLeftBorderSum = leftBorderSum;
}

int maxRightBorderSum = 0, rightBorderSum = 0;
for( int i = center + 1; i <= right; i++ )
{
    rightBorderSum += a[ i ];
    if( rightBorderSum > maxRightBorderSum )
        maxRightBorderSum = rightBorderSum;
}
```

Suma de la sub-secuencia de suma máxima

Versión 3 : $O(n \cdot \log n)$

```
        return max3( maxLeftSum, maxRightSum,
                     maxLeftBorderSum + maxRightBorderSum );
    }


    /**
     * Driver for divide-and-conquer maximum contiguous
     * subsequence sum algorithm.          */
    public static int maxSubSum3( int [ ] a ) {
        return maxSumRec( a, 0, a.length - 1 );
    }

    /* END */

    /**      * Return maximum of three integers.      */
    private static int max3( int a, int b, int c ) {
        return a > b ? a > c ? a : c : b > c ? b : c;
    }
```

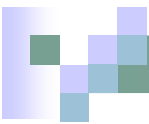
Versión 3 : Función de Tiempo de Ejecución

$$T(n) = \begin{cases} \text{cte}_1 & n = 1 \\ 2 * T(n/2) + n + \text{cte}_2 & n > 1 \end{cases}$$



2 llamadas recursivas

Buscar la sub-secuencia de suma máxima de cada mitad que incluye los elementos centrales



Suma de la sub-secuencia de suma máxima

Versión 4 : $O(n)$

```
/** Linear-time maximum contiguous subsequence sum
    algorithm. */
public static int maxSubSum4( int [ ] a )
{
    int maxSum = 0, thisSum = 0;
    for( int j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];
        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }
    return maxSum;
}
/* END */
```