Algoritmos
y
Estructuras de Datos

Cursada 2011

Prof. Catalina Mostaccio
Prof. Alejandra Schiavoni

Facultad de Informática - UNLP

2011

Objetivos de la materia

- Analizar algoritmos y evaluar su eficiencia

- Estudiar estructuras de datos dinámicas, tales como árboles y grafos: su implementación y aplicaciones

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)

Criterios de elección

- Recursos de computadora
  - espacio de memoria
  - tiempo de ejecución

Medida de eficiencia

- enfoque empírico (a posteriori)
- enfoque teórico (a priori)

T(n) **n**: tamaño de la entrada de los datos

se calcula en forma teórica.

tiempo en el peor de los casos.

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:

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

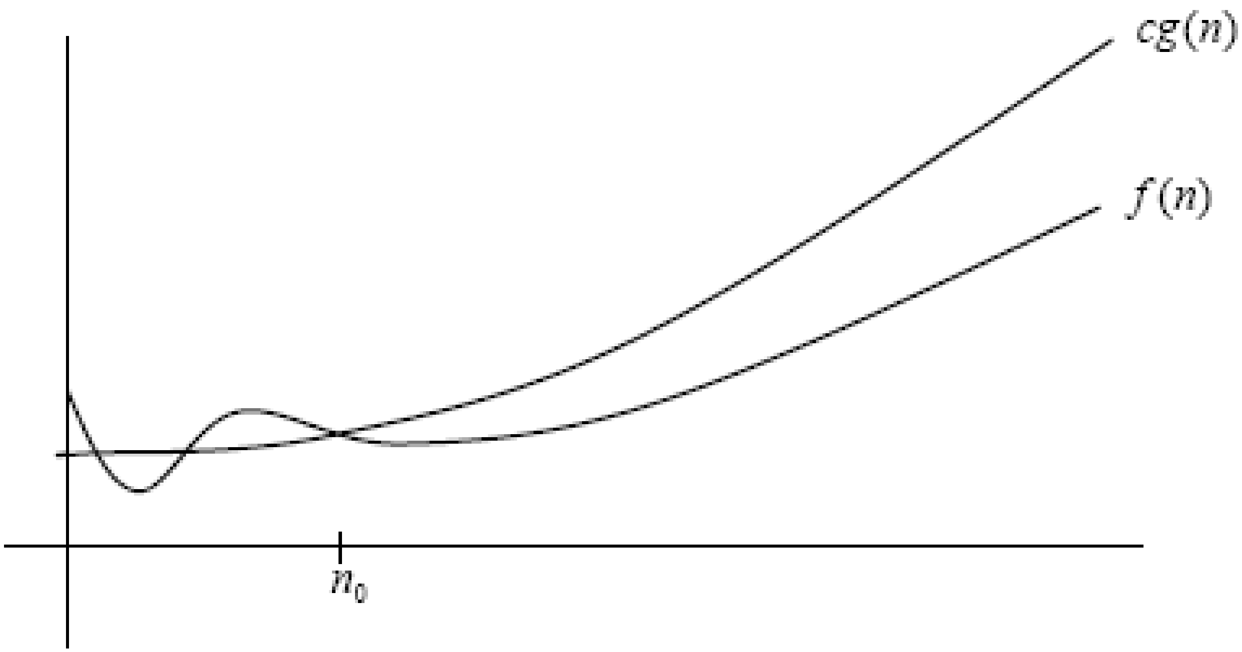
Se lee: T(n) es de orden de f(n)

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 $\boldsymbol{k} \Rightarrow T(n) = O(n^k)$
  - $T(n) = \log^k n \Rightarrow O(n)$ para cualquier k

    n siempre crece más rápido que cualquier potencia de log(n)

  - $T(n) = cte \quad \Rightarrow \quad O(1)$
  - $T(n) = cte * f(n) \Rightarrow T(n) = O(\, f(n)\,)$

Omega
Definición

Decimos que

$$\mathcal{T}(n) = \Omega(g(n))$$

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

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

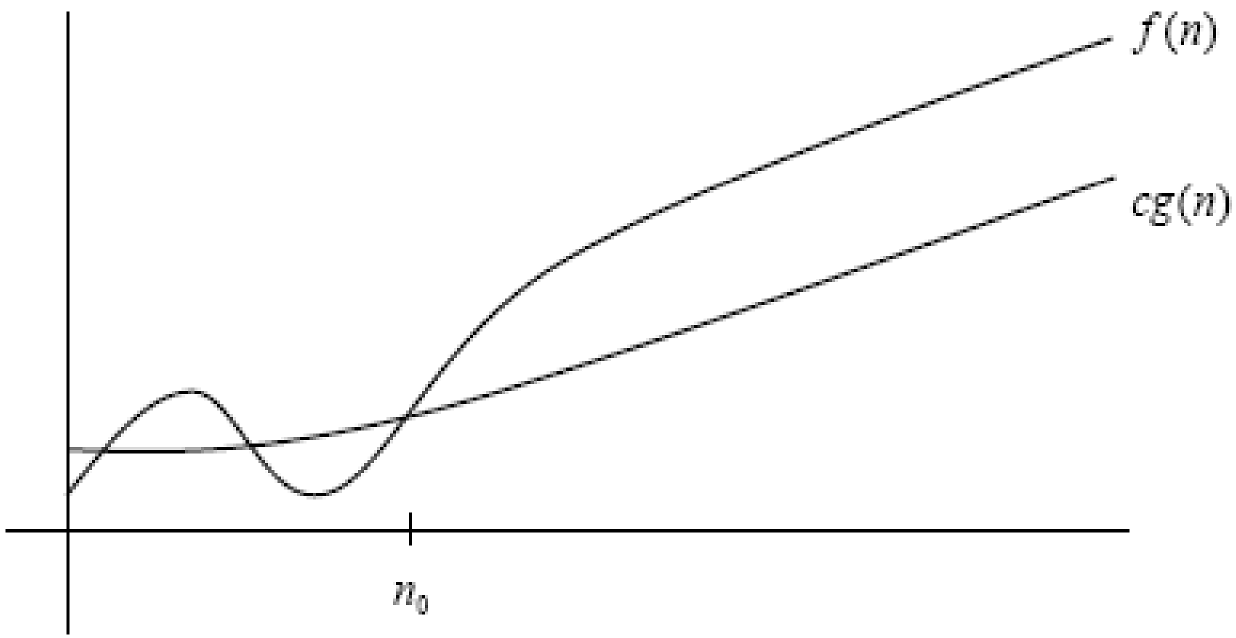
Se lee: T(n) es omega de g(n)

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) = Ω(g(n)) es:



Theta
Definición

Decimos que

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

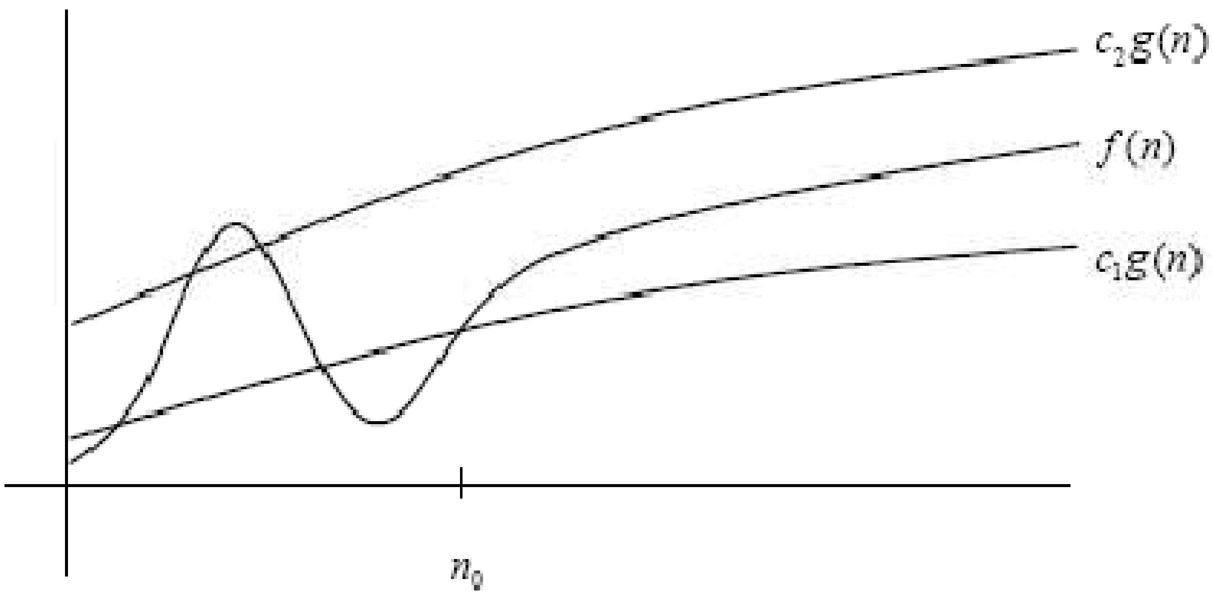
$$\leftrightarrow \; T(n) = O(h(n)) \text{ y } T(n) = \Omega(h(n))$$

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

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

Theta

Geométricamente f(n) = Θ(g(n)) es:



Algunas funciones

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

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

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 expession) {*

  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 [] a;
int sum = 0;
a = new int [20];
for (int i =1; i<= n ; i++ )
    sum += a[i];
```

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

$$= n * cte_2$$

$$\Rightarrow O(n)$$

Cálculo del Tiempo de Ejecución

a) ***For***

```
int [] [] a;
int sum = 0;
a = new int [20][20];
for (int i =1; i<= n ; i++) {
  for (int j =1; j<= n ; j++)
      sum += a[i][j];
}
```

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

$$= cte_1 + n*n*cte_2$$

$$\Rightarrow O(n^2)$$

Cálculo del Tiempo de Ejecución

a) ***For***

```
int [] a = new int [20];
int [] s = new int [20];
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];
}
```

$$T(n) = cte_1 + \sum_{i=1}^{n} cte_2 + \sum_{i=1}^{n} \sum_{j=1}^{i} cte_3 =$$

$$= cte_1 + n * cte_2 + cte_3 * \sum_{i=1}^{n} i$$

$$\Rightarrow O(n^2)$$

Cálculo del Tiempo de Ejecución

- Iteración :

b) ***While***

```
int x= 0;
int i = 1;
while ( i <= n) {
        x = x + 1;
        i = i + 2;
}
```

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

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

$$\Rightarrow O(n)$$

Cálculo del Tiempo de Ejecución

- Iteración :

b) *While*

int x= 1;

while (x < n)

*x = 2 *x;*

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

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

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)

$$T(n) = \sum_{i=1}^{n} cte1 + \sum_{i=1}^{n[paso2]} \left(\sum_{j=i}^{n} cte2 + \sum_{j=1}^{i+1} cte2 \right)$$

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 "*i*" impares

Cálculo del Tiempo de Ejecución

Ejemplo (cont.):

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

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

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)

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

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

$$= cte1 * n + cte2 * (n + 2) * n / 2$$

$$= cte1 * n + cte2 / 2 * n^2 + cte2 * n$$

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

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

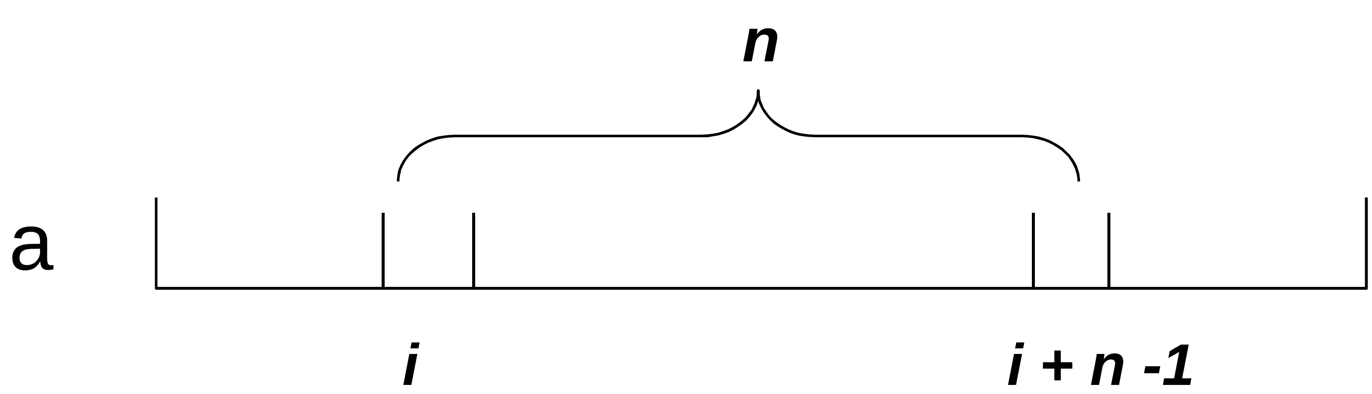
Factorial (n)

$$T(n) = \begin{cases} cte_1 & n = 1 \\ cte_2 + T(n-1) & n > 1 \end{cases}$$

Cálculo del Tiempo de Ejecución Algoritmos Recursivos

- Ejemplo :

Encontrar el máximo elemento en un arreglo de enteros tomando <u>n</u> posiciones a partir de la posición <u>i</u> .



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

$$T(n) = \begin{cases} cte_1 & n = 1 \\ 2 * T(n/2) + cte_2 & n > 1 \end{cases}$$

Cálculo del Tiempo de Ejecución

Algoritmos recursivos vs. iterativos

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;
   }
```

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:

$$a_1, a_2, a_3,........a_n$$

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, $\underbrace{11, -4, 13}$, -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*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*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*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

$$T(n) = \begin{cases} cte_1 & n = 1 \\ 2 * T(n/2) + n + cte_2 & n > 1 \end{cases}$$

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 */
```