# Programación 3

## Cursada 2020

**Prof. Alejandra Schiavoni (ales@info.unlp.edu.ar)**

**Prof. Pablo Iuliano (piuliano@info.unlp.edu.ar)**

# Agenda
## Análisis de algoritmos

- Introducción al concepto T(n)
  - Tiempo, entrada, peor caso, etc.

- Cálculo del T(n)
  - En algoritmos iterativos
  - En algoritmos recursivos

- Notación Big-Oh
  - Definición y ejemplos
  - Reglas (suma, producto)

- Ejemplo de optimización de algoritmos

# 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
# Función de recurrencia

*Factorial (n)* - Desarrollo de la recurrencia

$$T(n) = T(n-1) + cte_2 \qquad n > 1$$

$$T(n-2) + cte_2$$

$$T(n-3) + cte_2$$

$$T(n-4) + cte_2$$

# Cálculo del Tiempo de Ejecución
# Función de recurrencia

*Factorial (n)* - Desarrollo de la recurrencia

$T(n) = T(n - 1) + cte_2 = (T(n - 2) + cte_2) + cte_2 =$

$= T(n - 2) + 2\ cte_2 = (T(n - 3) + cte_2) + 2\ cte_2 =$

$= T(n - 3) + 3\ cte_2 = \ldots\ldots\ldots$

Paso i:

$T(n) = T(n - i) + i\ {*}cte_2$

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

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

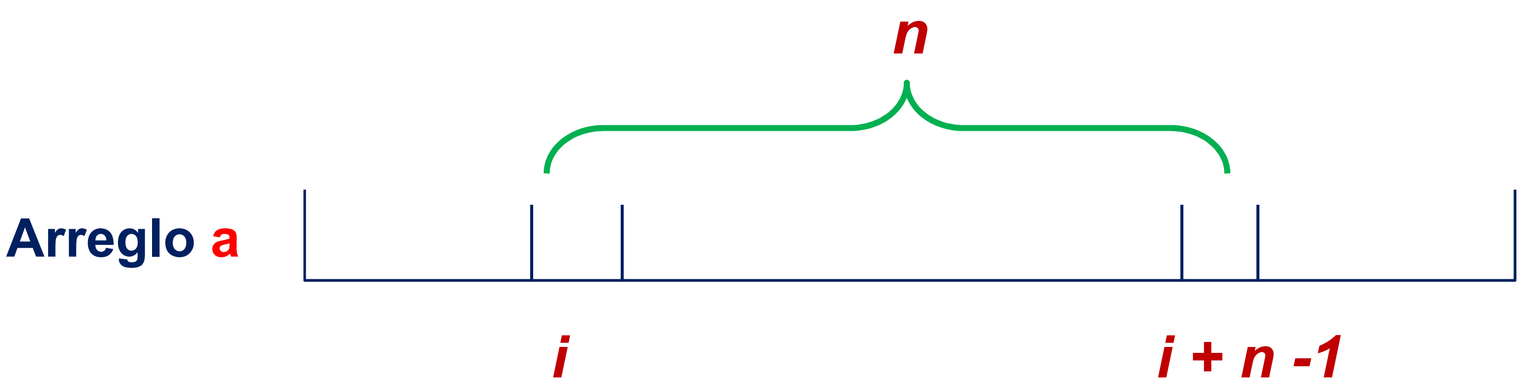
Cuando **n-i = 1** $\longrightarrow$ **i = n -1**, reemplazamos **i** en la expresión de T(n)

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

# 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*

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

# Cálculo del Tiempo de Ejecución
# Función de recurrencia

*Máximo en un arreglo* – Desarrollo de la recurrencia

$$T(n) = 2 * T(n/2) + cte_2 \qquad n > 1$$

$$2 * T(n/4) + cte_2$$

$$2 * T(n/8) + cte_2$$

$$2 * T(n/16) + cte_2$$

# Cálculo del Tiempo de Ejecución
# Función de recurrencia

## *Máximo en un arreglo* – Desarrollo de la recurrencia

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

$= 4 * T(n/4) + 3\ \text{cte}_2 = 4 * [2 * T(n/8) + \text{cte}_2 ] + 3\ \text{cte}_2 =$

$= 8 * T(n/8) + 7\ \text{cte}_2 = 8 * [2 * T(n/16) + \text{cte}_2 ] + 7\ \text{cte}_2 =$

$= 16 * T(n/16) + 15\ \text{cte}_2 = \ldots\ldots..$

Paso i:

$T(n) = 2^i * T(n/2^i) + (2^i - 1) * \text{cte}_2$

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

# Cálculo del Tiempo de Ejecución
# Función de recurrencia

## *Máximo en un arreglo* – Desarrollo de la recurrencia

$$T(n) = 2^i * T(n/2^i) + (2^i - 1) * \mathrm{cte}_2$$

Cuando **$n/2^i = 1$** $\longrightarrow$ **$n = 2^i$** $\longrightarrow$ **$i = \log_2 n$,**

reemplazamos **i** en la expresión de T(n)

$$T(n) = n * \underbrace{T(n/n)}_{T(1) = \mathrm{cte}_1} + (n - 1) * \mathrm{cte}_2 = n * \mathrm{cte}_1 + (n - 1) * \mathrm{cte}_2 = O(n)$$

# Notación Big-Oh

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

# Notación Big-Oh
## Definición

Decimos que

$$\mathcal{T}(n) = O(f(n))$$

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

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

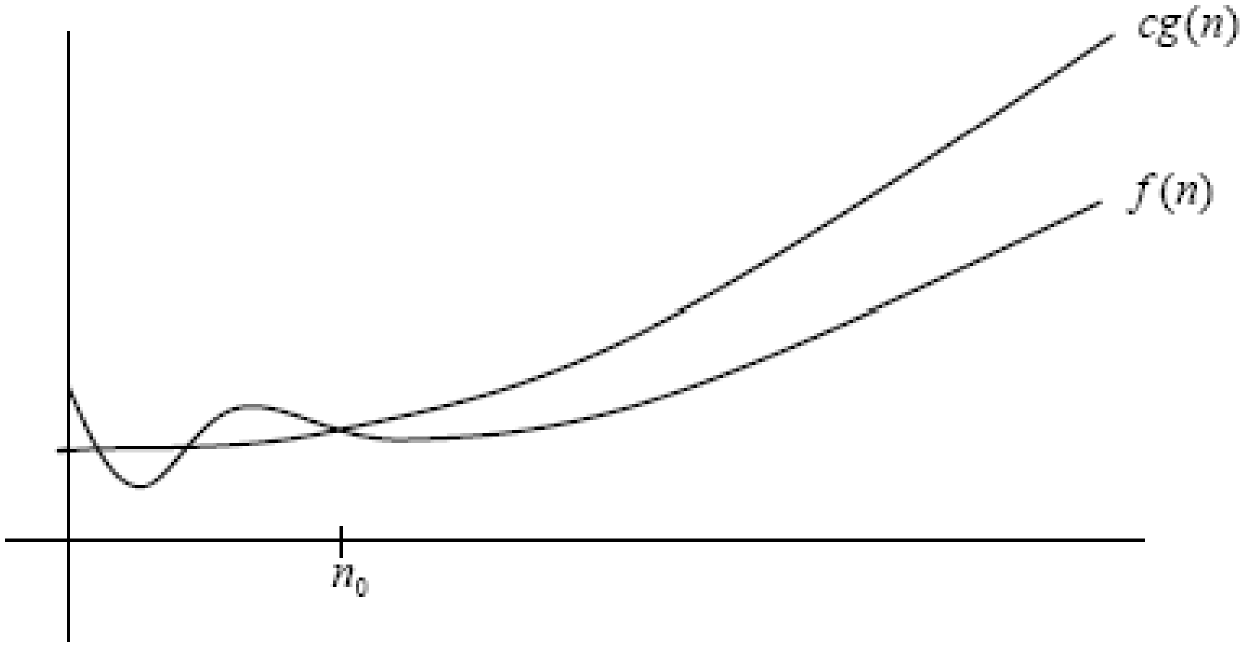
Se lee: T(n) es de orden de f(n)

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

Como la notación O grande solamente da una cota asintótica superior, y no una cota asintóticamente ajustada, podemos hacer declaraciones que en primera instancia parecen incorrectas, pero que son técnicamente correctas.

Por ejemplo, es absolutamente correcto decir que la búsqueda binaria se ejecuta en un tiempo **O(n)**. Eso es porque el tiempo de ejecución crece no más rápido que una constante multiplicada por **n**. De hecho, crece más despacio.

# Notación Big-Oh

Pensémoslo de esta manera: tenés 10 pesos en el bolsillo, vas con un amigo y le decís:

*"Tengo una cantidad de dinero en mi bolsillo, y te aseguro que no es más de un millón de pesos*"

Tu afirmación es absolutamente cierta, aunque no terriblemente precisa. Un millón de pesos es una cota superior de los 10 pesos, del mismo modo que O(n) es una cota superior del tiempo de ejecución de la búsqueda binaria. Otras cotas superiores, imprecisas, sobre la búsqueda binaria serían $O(n^2)$, $O(n^3)$ y $O(2^n)$.

# 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. $T_1(n)*T_2\ (n)=O(f(n)*g(n))$

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

# Notación Big-Oh

Ejemplos

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

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

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

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

# 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**

# Cálculo del Tiempo de Ejecución

## Algoritmos recursivos vs. iterativos

# Números de Fibonacci – Introducción

## Descripción del 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.

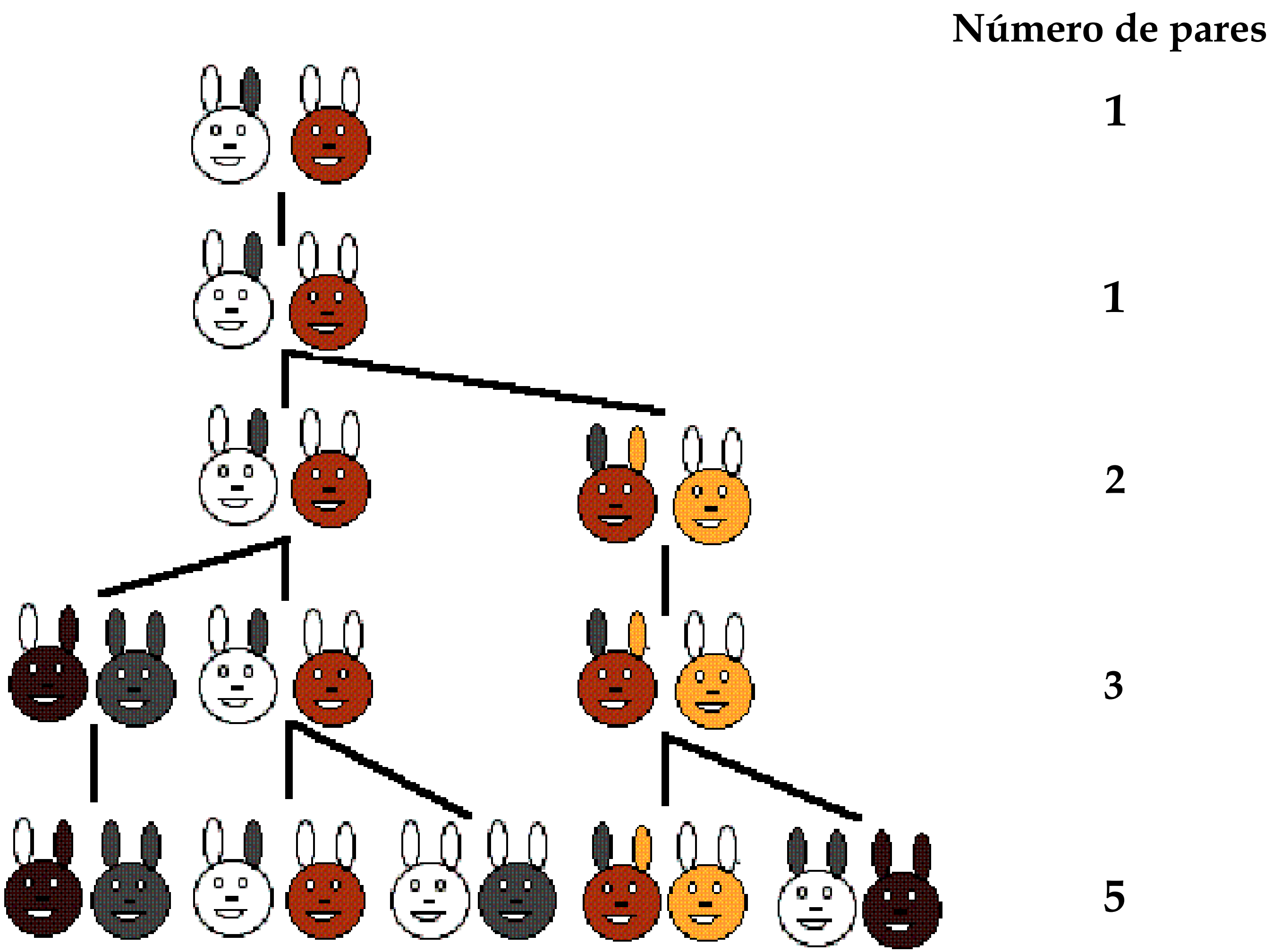
5. La hembra siempre da a luz una pareja de conejos de sexos opuestos.

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 );
     }
```

# 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 de recurrencia

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

# 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

$$T(n) = \begin{cases} cte_1 & n <= 1 \\ cte_2 + m + m^* \, T(n-1) & n > 1 \end{cases}$$

# 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:

$$a_1, a_2, a_3, \ldots\ldots a_n$$

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

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

Ej.: $\underbrace{-2,\ 11,\ -4,\ 13}$, -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;
     }
```

# 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;
       }
```

```
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;
}
```

# Suma de la sub-secuencia de suma máxima
# Versión 3: O(n*log n)

```
/** Solución recursiva:
  *  Explicar la resolución detallada y graficamente          */
```

# 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;
 }
```