



# Algoritmos Estructuras de Datos I

Facultad de Ciencias Exactas y Tecnología  
Universidad Nacional de Tucumán

2025

# Técnicas algorítmicas(2)

# Recursión-Ejemplo f1

```
FUNCION f1 (n) : entero  $\geq 0 \rightarrow$  entero  $\geq 0$   
  SI (n = 0) ENTONCES  
    retorna (0)  
  SINO  
    retorna ( f1(n-1) + f1(n-1) + n * (n+1) * (n-1) )  
FIN
```

Relación de recurrencia:

$$T(n) = d \quad \text{si } n = 0$$

$$T(n) = c + 2 * T(n-1) \quad \text{si } n > 0$$

c,d son constantes

# Recursión-Ejemplo f1

Sea

si  $n = 0$

$$T(n) = d$$

si  $n > 0$

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

$c, d$  son constantes

Si  $n > 0$

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

Si  $n-1 > 0, n > 1$

$$T(n) = c + 2 * (c + 2 * T(n-2)) = 3 * c + 4 * T(n-2)$$

Si  $n-2 > 0, n > 2$

$$T(n) = 7 * c + 8 * T(n-3)$$

Si  $n-3 > 0, n > 3$

$$T(n) = 15 * c + 16 * T(n-4)$$

$\forall n > k$

$$T(n) = (2^{k+1} - 1) * c + 2^{k+1} * T(n-(k+1))$$

Si  $n = k+1$

$$T(n) = (2^n - 1) * c + 2^n * T(0) = 2^n * (c + d) - c$$

Entonces  $T(n) \in O(2^n)$  para la función f1

# Recursión-Ejemplo f2

```
FUNCION f2 (n) : entero  $\geq 0 \rightarrow$  entero  $\geq 0$   
  SI (n = 0) ENTONCES  
    retorna (0)  
  SINO  
    retorna ( 2 * f2(n-1) + n * (n+1) * (n-1) )  
FIN
```

Relación de recurrencia:

$$T(n) = d \quad \text{si } n = 0$$

$$T(n) = c + T(n-1) \quad \text{si } n > 0$$

c,d son constantes

# Recursión-Ejemplo f2

Sea

si  $n = 0$

$$T(n) = d$$

si  $n > 0$

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

$c, d$  son constantes

Si  $n > 0$

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

Si  $n > 1$

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

Si  $n > 2$

$$T(n) = 3 * c + T(n-3)$$

Si  $n > 3$

$$T(n) = 4 * c + T(n-4)$$

$\forall n > k$

$$T(n) = (k+1) * c + T(n-(k+1))$$

Para  $n = k+1$

$$T(n) = n * c + T(0) = n * c + d$$

Entonces  $T(n) \in O(n)$  para la función f2

# Recursión-Ejemplo Multiplicar

Algoritmo de multiplicación por sumas sucesivas:

```
FUNCION M(A,B) :entero $\geq$ 0 x entero $>$ 0  $\rightarrow$  entero $\geq$ 0  
  si B=1 entonces  
    retorna ( A )  
  sino  
    retorna ( A + M(A,B-1) )  
FIN
```

Relación de recurrencia:

$$T(A,B) = d \quad \text{si } B = 1$$

$$T(A,B) = c + T(A,B-1) \quad \text{si } B > 1, \quad c,d \text{ son constantes}$$

# Recursión-Ejemplo Multiplicar

Sea

$$\text{si } B = 1 \quad T(A,B) = d$$

$$\text{si } B > 1 \quad T(A,B) = c + T(A,B-1) \quad , \quad c,d \text{ son constantes}$$

$$\text{Si } B > 1 \quad T(A,B) = c + T(A,B-1)$$

$$\text{Si } B-1 > 1, B > 2 \quad T(A,B-1) = c + T(A,B-2), \quad T(A,B) = 2*c + T(A,B-2)$$

$$\text{Si } B-2 > 1, B > 3 \quad T(A,B-2) = c + T(A,B-3), \quad T(A,B) = 3*c + T(A,B-3)$$

$$\forall B > k \quad T(A,B) = k*c + T(A,B-k)$$

$$\text{Para } B=k+1 \quad T(A,B) = (B-1)*c + T(A,1) = B*c - c + d$$

Entonces  $T(A,B) \in O(B)$  para la función M



# Recursión-Ejemplo Torres Hanoi

Juego Torres de Hanoi (Édouard Lucas-1883)



# Recursión-Ejemplo Torres Hanoi

## Juego Torres de Hanoi

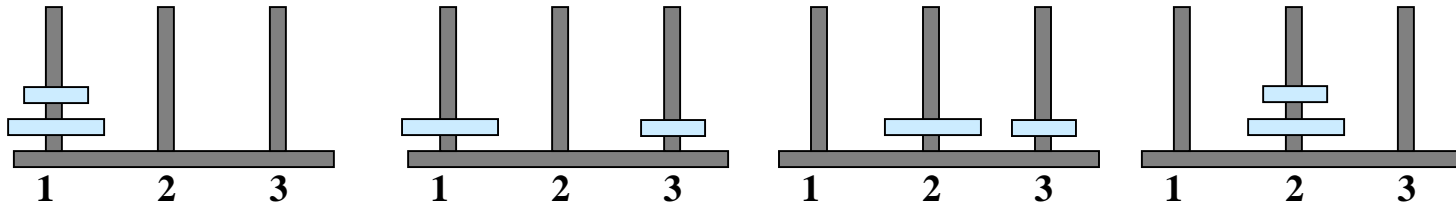
- Tres varillas verticales.
- En una de las varillas se apila un número determinado de discos en tamaño decreciente (No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio en una de las varillas)
- El juego consiste en pasar todos los discos de la varilla ocupada a una de las otras varillas vacías siguiendo tres reglas:
  - Sólo se puede mover un disco cada vez.
  - Sólo se puede sacar el disco que se encuentre arriba en cada varilla.
  - Un disco de mayor radio no puede colocarse sobre uno más chico.



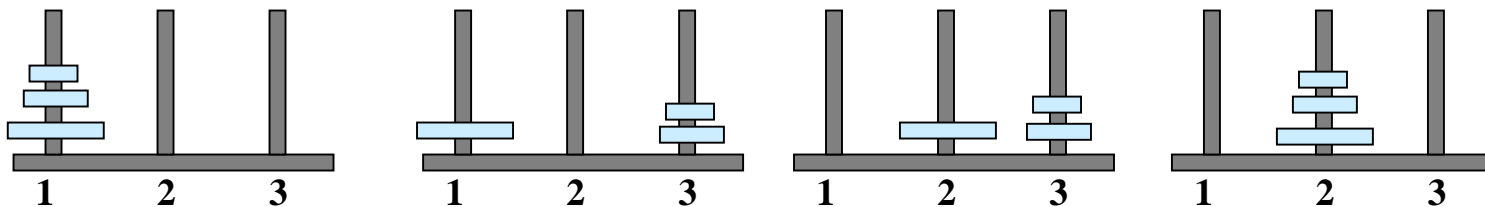
# Recursión-Ejemplo Torres Hanoi

## Problema de las Torres de Hanoi.

*2 anillos: 3 movimientos*



*3 anillos: 7 movimientos*



# Recursión-Ejemplo Torres Hanoi

FUNCION **Hanoi** (  $n$  ,  $i$  ,  $j$  )

Entrada:

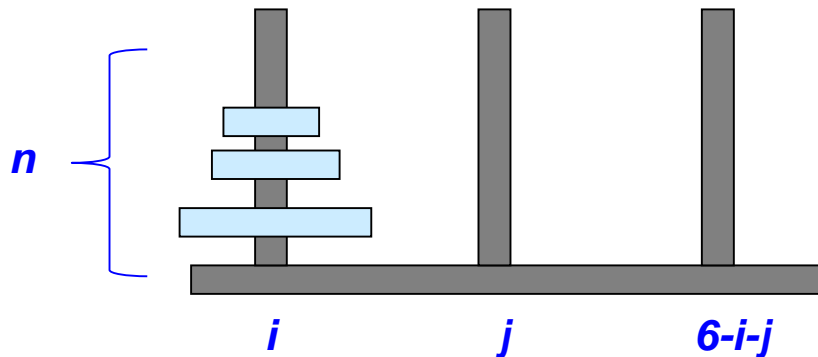
$n$  : cantidad de anillos a mover

$i$  : torre origen con valores : 1,2,3

$j$  : torre destino con valores : 1,2,3

Auxiliar:

$6-i-j$  : torre auxiliar con valores : 1,2,3



# Recursión-Ejemplo Torres Hanoi

```
FUNCION Hanoi (  $n$  ,  $i$  ,  $j$  )
```

```
// mueve  $n$  anillos de la torre origen  $i$  al destino  $j$ 
```

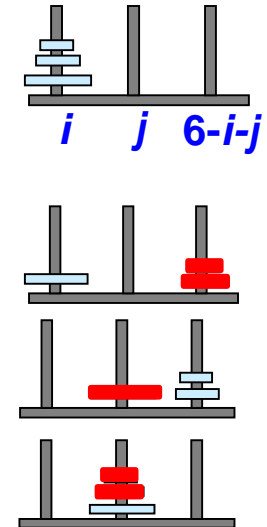
```
SI  $n > 0$  ENTONCES
```

```
    Hanoi (  $n-1$  ,  $i$  ,  $6-i-j$  )
```

```
    Escribir ( se mueve un disco de  $i \rightarrow j$  )
```

```
    Hanoi (  $n-1$  ,  $6-i-j$  ,  $j$  )
```

```
FIN
```



Cuántos movimientos hace este algoritmo para  $n$  discos?

# Recursión-Ejemplo Torres Hanoi

Sea  $M(n)$  el numero de movimiento que se realizan en el algoritmo:

$$M(n) = 0 \quad \text{si } n=0$$

$$M(n) = 2 M(n-1) + 1 \quad \text{si } n>0$$

Desarrollando se puede obtener que:  $M(n) = 2^n - 1$

Entonces el algoritmo HANOI tiene un tiempo  $\in O(2^n)$ .

Si  $n=64$  discos:

numero de movimientos =  $2^{64} - 1 = 18.446.744.073.709.551.615 \approx 1.84 \times 10^{19}$

si se hace 1 movimiento / segundo durante las 24 hs del día,

1 año = 31536000 segundos

Tiempo Total = 585.000.000.000 =  **$5.8 \cdot 10^{11}$  años**



# Recursión

Planteos:

- Se puede remover la recursión de un algoritmo y transformarlo en iterativo?
- Es una técnica eficiente la recursión ?
- Cuando conviene usarla ?
- Cuando no se debe usar ?

# Divide & Conquer

Es una metodología ***top-down*** ya que resuelve los problemas de una manera *descendente* en 2 etapas:

1. ***Divide:*** división en problemas mas chicos que se resuelven en forma independiente
  2. ***Conquer:*** la solución del problema original se forma combinando la solución de los problemas más chicos.
- Generalmente se usa la ***recursión*** en las implementaciones de estos algoritmos, aunque puede aplicarse también de manera ***iterativa***.



# Divide & Conquer

**ALGORITMO : D&C**

**ENTRADA:**     x (problema)

**SALIDA:**     y (solución)

**P1.**     **SI** x es “suficientemente chico” **ENTONCES**  
          aplicar un algoritmo básico para x

**SINO**

**Descomponer** x en instancias mas chicas ( $x_1, x_2, \dots, x_k$ )

**PARA**   i=1,k     **HACER**

$y_i \leftarrow \text{D\&C}(x_i)$

**Combinar** los ( $y_1, y_2, \dots, y_k$ ) para obtener la solucion y.

**P2.**     **FIN**

# Divide & Conquer-Ejemplo $x^n$

Ejemplo: calcular  $x^n$ , cuando  $n$  es un entero positivo o nulo y  $x$  es un numero real.

- 1) pot1 : algoritmo usando iteración:  $x * x * x \dots x$  ( $n$  veces)
- 2) Pot2 : Algoritmo usando recursión:  $x^0=1$ ,  $x^n=x * x^{n-1}$ ,  $n \geq 1$
- 3) Algoritmo usando: *divide & conquer + recursión + balance*

Se define  $x^n$  en función de  $x^{n/2}$

$$x^0 = 1$$

$$x^n = x^{n/2} * x^{n/2}$$

si  $n \geq 2$  es par

$$x^n = x * x^{(n-1)/2} * x^{(n-1)/2}$$

si  $n \geq 1$  es impar

# Divide & Conquer-Ejemplo $x^n$

Algoritmo usando: recursion + divide & conquer + balance.

**FUNCION** **pot3**(x,n) : real x entero  $\geq 0 \rightarrow$  real

    SI  $n=0$  ENTONCES

        RETORNA (1)

    SINO

        SI n es par ENTONCES

            RETORNA ( **pot3**( $x*x$ ,  $n/2$ ))

        SINO

            RETORNA (  $x * \text{pot3}(x*x, (n-1)/2)$  )

FIN

- La complejidad de **pot3**  $\in O(\log_2 n)$

# Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado  $A[1..n]$  buscar un dado  $x$  por el método de Búsqueda Binaria.

La **búsqueda binaria**, también conocida como **búsqueda dicotómica**, o **búsqueda logarítmica**, determina la pertenencia y/o encuentra la posición de un dado valor en un arreglo ordenado.

Durante este proceso va comparando el valor dado con el elemento  $x$  en la posición del medio del arreglo,

- Si  *$x$  coincide con el elemento del medio*: listo, lo encontré.
- Si  *$x$  es menor* que el del medio: continua con la parte izquierda del arreglo.
- Si  *$x$  es mayor* que el del medio: continua con la mitad derecha.
- Así sigue hasta que lo encuentre o se crucen los índices.

## Ejemplo de D&C: **búsqueda binaria iterativa** en A ordenado

**FUNCION BBinariaITER**(buscado,A,n) : entero x arreglo x entero  $\rightarrow$  bool

P1. inferior  $\leftarrow$  1

P2. superior  $\leftarrow$  n

P3. MIENTRAS (inferior  $\leq$  superior) HACER

    medio  $\leftarrow$  ( inferior + superior ) / 2

    SI buscado = A[medio] ENTONCES

        Retorna Verdadero

    SINO SI buscado < A[medio] ENTONCES

        superior  $\leftarrow$  medio - 1

    SINO // buscado > A[medio]

        inferior  $\leftarrow$  medio + 1

P4. Retorna Falso

P5. FIN

# Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado  $A[1..n]$  buscar un dato buscado por el método de Búsqueda Binaria.

Ejemplo:           $n=8$           buscado=46

**A=**    **[5      11      23      35      46      62      79      94]**

inferior=1          superior=8          medio=4           $A[4]=35 < \text{buscado}$

**A=**    [5      11      23      35      46      62      79      94]

inferior=5          superior=8          medio=6           $A[6]=62 > \text{buscado}$

**A=**    [5      11      23      35      46      62      79      94]

inferior=5          superior=5          medio=5           $A[5]=46 = \text{buscado}$

→ Encontrado=Verdadero, índice=5

# Divide & Conquer-Ejemplo **BB**

Dado un arreglo ordenado  $A[1..n]$  buscar un dato buscado por el método de Búsqueda Binaria.

Ejemplo:  $n=8$   $\text{buscado}=13$

**A=** [5    11    23    35    46    62    79    94]

inferior=1    superior=8    medio=4     $A[4]=35 > \text{buscado}$

**A=** [5    11    23    35    46    62    79    94]

inferior=1    superior=3    medio=2     $A[2]=11 < \text{buscado}$

**A=** [5    11    23    35    46    62    79    94]

inferior=3    superior=3    medio=3     $A[3]=23 > \text{buscado}$

inferior=3    superior=2     $\rightarrow$  se cruzan los índices

$\rightarrow$  encontrado=Falso, índice= -1

## Ejemplo de D&C: **búsqueda binaria recursiva** en A ordenado

**FUNCION BBinariaR**(buscado,A,inferior,superior)

: entero x arreglo x entero x entero  $\rightarrow$  bool

P1.      medio  $\leftarrow$  (inferior + superior) / 2

P2.      SI inferior > superior ENTONCES

            Retorna Falso

            SINO SI buscado = A[medio] ENTONCES

                    Retorna Verdadero

            SINO SI buscado < A[medio] ENTONCES

                    Retorna **BBinariaR**(buscado,A,inferior,medio-1)

            SINO // buscado > A[medio]

                    Retorna **BBinariaR**(buscado,A,medio+1,superior)

P3.      FIN



# Divide & Conquer

## Tiempo de ejecución:

$$T(n) = a T(n/b) + c n^k$$

**Teorema:** la solución de la ecuación:

$$T(n) = a T(n/b) + c n^k$$

donde  $c \geq 0$  real,  $a \geq 1$  real,  $b \geq 2$  entero y  $k \geq 0$  entero,  
es:

- $T(n) = O(n^{\log_b a})$  si  $a > b^k$
- $T(n) = O(n^k \log_b n)$  si  $a = b^k$
- $T(n) = O(n^k)$  si  $a < b^k$

# Divide & Conquer

Se aplica el **Teorema**:  $T(n) = a T(n/b) + c n^k$

- $T(n) = O(n^{\log_b a})$  si  $a > b^k$
- $T(n) = O(n^k \log_b n)$  si  $a = b^k$
- $T(n) = O(n^k)$  si  $a < b^k$

Para estimar el **tiempo de ejecución de Búsqueda binaria**:

$a=1$  ,  $b=2$  ,  $k=0$  entonces:  $a = b^k$

la solución de la ecuación según el teorema es entonces:

$$T(n) = O(n^k \log_b n)$$

→  $T(n) \in O(\log_2 n)$  para la búsqueda binaria

# Programación Dinámica

- Es una **técnica ascendente** (*bottom up*)
- Comienza con los casos más pequeños.
- Combinando soluciones, va obteniendo soluciones para los casos cada vez mayores, hasta que finalmente llega a la solución del problema original.
- Forma una tabla de resultados de subproblemas para alcanzar la solución del problema.
- No recalcula.

# Programación Dinámica

- La mayor aplicación de la Programación Dinámica es en la resolución de **problemas de optimización**.
- En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se busca es encontrar la solución de valor óptimo (máximo o mínimo).
- La solución de problemas mediante esta técnica se basa en el llamado **principio de optimalidad** enunciado por *Bellman en 1957* que dice:

**“En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima”.**

# Programación Dinámica - Ejemplo

**Leonardo Fibonacci** (1170-1240), llamado Leonardo Pisano, matemático italiano que realizó importantes aportes en los campos matemáticos del álgebra y la teoría de números.

Descubrió la sucesión llamada de Fibonacci : 0, 1, 1, 2, 3, 5, 8, 13...

Los términos se pueden definir por la siguiente recurrencia:

$$\begin{aligned} f_0 &= 0 \quad , \quad f_1 = 1 \quad , \\ f_n &= f_{n-1} + f_{n-2} \quad \text{para } n \geq 2 \end{aligned}$$

A cada término de esta sucesión se le denomina número de Fibonacci.

# Programación Dinámica – Ej. Fib1

El algoritmo obtenido directamente de la definición es **recursivo**:

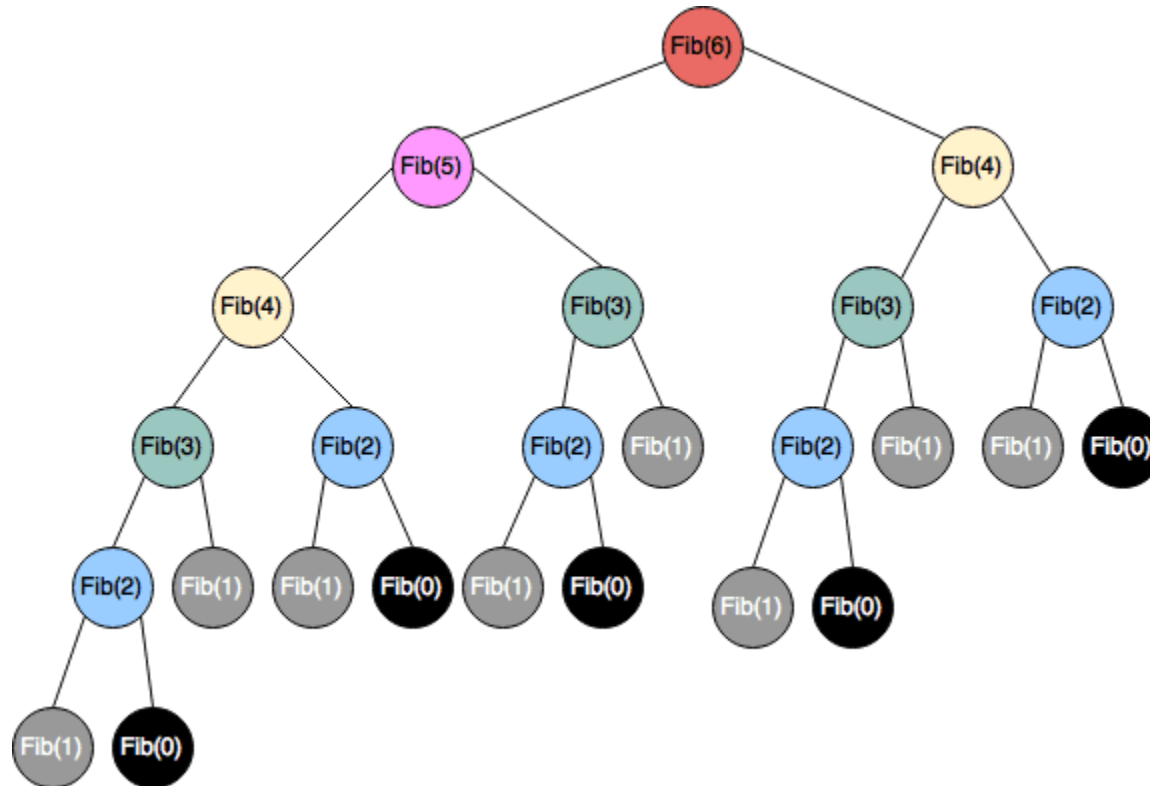
```
FUNCION Fib1 (n) :entero $\geq$ 0  $\rightarrow$  entero $\geq$ 0
  SI n < 2 ENTONCES
    retorna(n)
  SINO
    retorna Fib1(n-1) + Fib1(n-2)
FIN
```

$$T(n) = \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1-\sqrt{5}}{2} \right)^n \quad T(n) \in O \left( \left( \frac{1+\sqrt{5}}{2} \right)^n \right) = O(1.618^n)$$

- **Fib1** es un algoritmo exponencial en tiempo de ejecución.

# Programación Dinámica – Ej. Fib1

Por ejemplo para  $\text{Fib}(6)=8$  se genera el siguiente árbol de llamadas:



Cabe destacar que se puede plantear otros algoritmos mucho mas eficientes que Fib1.

# Programación Dinámica – Ej. Fib2

El planteo de ***programación dinámica*** para el calculo directo de la secuencia de Fibonacci hace el almacenamiento en solo 2 variables auxiliares : i, j.

```
FUNCION Fib2 (n) : entero $\geq$ 0  $\rightarrow$  entero $\geq$ 0
  i  $\leftarrow$  1
  j  $\leftarrow$  0
  PARA k = 1,n HACER
    j  $\leftarrow$  i + j;
    i  $\leftarrow$  j - i;
  retorna(j)
FIN
```

- El algoritmo Fib2 tiene tiempo de orden lineal: **O(n)**.



# Programación Dinámica – Ej. Fib3

El tercer algoritmo un poco más sofisticado, usa unas cuantas variables auxiliares mas.

```
FUNCION Fib3(n) : entero $\geq$ 0  $\rightarrow$  entero $\geq$ 0
  i  $\leftarrow$  1; j  $\leftarrow$  0; k  $\leftarrow$  0; h  $\leftarrow$  1;
  MIENTRAS n>0 HACER
    SI n es impar ENTONCES
      t  $\leftarrow$  j*h;
      j  $\leftarrow$  i*h+j*k+t;
      i  $\leftarrow$  i*k+t;
      t  $\leftarrow$  h*h;
      h  $\leftarrow$  2*k*h+t;
      k  $\leftarrow$  k*k+t;
      n  $\leftarrow$  n / 2;
  Retorna(j)
FIN
```

- El algoritmo Fib3 tiene tiempo de orden  $O(\log_2 n)$