uao

04

ESTRUCTURA DE DATOS Y ALGORITMOS 1

Recursividad

Profesor
Orlando Arboleda Molina

uao

Recursión

- Un **método recursivo** es aquel que **se invoca a si mismo**, ya sea **directamente** o **indirectamente** a través de otro método.

- Un **método recursivo** para resolver un problema:
  - Debe tener **uno** o **mas casos simples** de resolver, denominado caso **base**.
  - Para **casos mas complejos**, realiza una **invocación al mismo método** para una **versión mas sencilla o pequeña**. Este es denominado el **caso recursivo**.

Ejemplo: método recursivo para el computo del factorial de un valor n.

$$0! = 1$$
$$1! = 1$$
$$2! = 2 * 1 = 2 * 1!$$
$$3! = 3 * 2 * 1 = 3 * 2!$$
$$4! = 4 * 3 * 2 * 1 = 4 * 3!$$
$$\ldots\ldots$$
$$n! = n * \left((n-1) * \cdots * 2 * 1\right) = n * (n-1)!$$

```
function factorialRecursivo(n){
    if (n<=1) {
        return 1;          } caso base
    }else{
        return n*factorialRecursivo(n-1);   } caso recursivo
    }
}
```

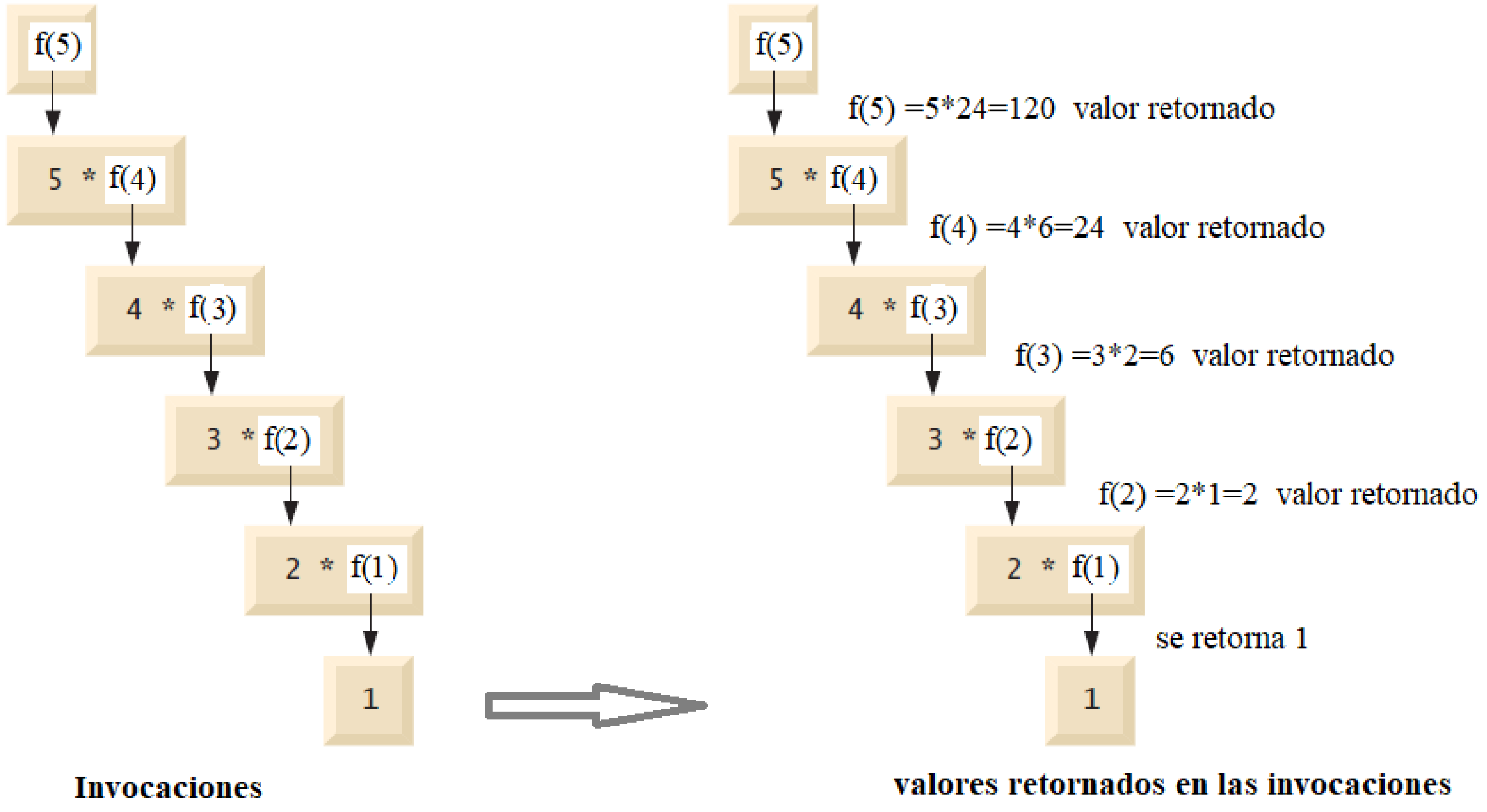
- Usando **métodos recursivos** se pueden **generar soluciones** más **elegantes** y **sencillas**.

Recursión

Ejemplo: las siguientes serán las invocaciones y valores retornados en la invocación al método ***factorialRecursivo*** para un valor de n=5.

```
function factorialRecursivo(n){
    if (n<=1) {
        return 1;
    }else{
        return n*factorialRecursivo(n-1);
    }
}
```

$$f(n) = \begin{cases} 1, & si \;\; n \leq 1 \\ n * f(n-1), & si \; n > 1 \end{cases}$$



- La **recursión** es **costosa** por el **espacio en memoria** (en cada llamada se crea una nueva copia) y **el tiempo del procesador** (las invocaciones realizadas).

Recursión

- Para obtener las funciones matemáticas recursivas, se debe identificar:
  - **Caso recursivo** - como computar el resultado generalmente de un **caso n**, a partir de las **soluciones previas** (generalmente de n-1).
  - **Caso base** – el **valor inicial** a partir del cual se pueden obtener los otros resultados.

Ejemplo: obtención de la función recursiva para la potencia positiva de una base b

Teniendo en cuenta que:

$$b^0 = 1$$
$$b^1 = b$$
$$b^2 = b.b$$
$$b^3 = b.b.b$$
$$b^4 = b.b.b.b$$
$$\ldots\ldots\ldots$$
$$b^n = \underbrace{b.b.b.\ \ldots\ .b}_{\textbf{\textit{n} veces}}$$

$$b^0 = 1$$
$$b^1 = b.\boldsymbol{b^0}$$
$$b^2 = b.\boldsymbol{b^1}$$
$$b^3 = b.\boldsymbol{b^2}$$
$$b^4 = b.\boldsymbol{b^3}$$
$$\ldots..$$
$$b^n = b.\boldsymbol{b^{n-1}}$$

$$\boldsymbol{f(b,n)} = \begin{cases} \mathbf{1}, & \boldsymbol{si\ \ n = 0} \\ \boldsymbol{b * f(b, n-1)}, & \boldsymbol{si\ n > 1} \end{cases}$$

Ejercicios

- Obtener la función recursiva denominada ***calcularDeuda*** que permite computar: el valor adeudado por un préstamo de *A* pesos, al cabo de *n* meses, si en cada mes se aplica el porcentaje *i* (ej: i puede ser 0.05 correspondiente al 5%) sobre el valor adeudado hasta la fecha (y no se han realizado pagos previos).

Ejercicios de implementación

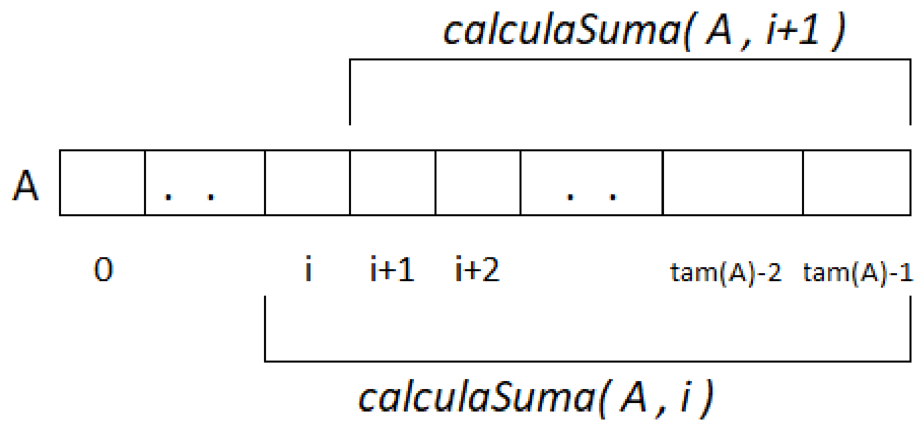
- En la **practica de Recursión** incluir las siguientes implementaciones:
  1. La implementación de la función recursiva que denominaremos ***getGCD***, que según el libro **How To Program in Java**, es definido de la siguiente manera:
     "The Greatest Common Divisor of integers x and y is the largest integer that evenly divides into both x and y. The gcd is defined recursively as follows: If y is equal to 0, then gcd(x,y) is x; otherwise, gcd(x,y) is gcd(y,x%y), where % is the remainder operator".

  2. Las funciones recursivas obtenidas hasta el momento.

Recursión

- Se pueden obtener **funciones recursivas** para **datos almacenados** en estructuras.

Ejemplo: se indica la función recursiva ***calculaSuma(A,i)*** que retorna la sumatoria de los valores almacenados en el arreglo A, desde la posición i hasta la posición final.



$$calculaSuma(A, i) = \begin{cases} 0 & \text{si } i>tam(A)-1 \\ A[i]+calculaSuma(A,i+1) & \text{si } i<=tam(A)-1 \end{cases}$$

Ejercicios de implementación: tomando como referencia un arreglo de valores numéricos, implementar de forma recursiva las siguiente funciones:

1. La función ***calcularSuma***
2. La función ***obtenerMayor***, que debe retornar el numero mas grande que existe desde la posición *i*
3. La función ***contarMayores*** que debe retornar cuantos valores son mayores al *valorDado* desde la posición *i*
4. La función ***esPalindromo***, que debe determinar si teniendo en cuenta los valores almacenados, es o no palíndromo desde la posición *i* hasta la posición *j*

Tiempo de Ejecución y Orden de Complejidad

- El **tiempo de ejecución** de un **método recursivo** es una **función de recurrencia**, a partir de la cual se obtendrá su **orden de complejidad**.

Ejemplo: a continuación se presentan tiempos de ejecución ***T(n)*** que se pueden obtener del caso base (indicado con el color verde) y el caso recursivo (indicado con el color rojo) .

O(1) + T(n-1)

```
function factorialRecursivo(n){
    if (n<=1) {
        return 1;
    }else{
        return n*factorialRecursivo(n-1);
    }
}
```

O(1)

formula general, siendo **a** y **b** constantes

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{a}, & \boldsymbol{si\ \ n \le 1} \\ \boldsymbol{b + T(n-1)}, & \boldsymbol{si\ n > 1} \end{cases}$$

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{O(1)}, & \boldsymbol{si\ \ n \le 1} \\ \boldsymbol{O(1) + T(n-1)}, & \boldsymbol{si\ n > 1} \end{cases}$$

Tiempo de Ejecución y Orden de Complejidad

Ejemplo: A continuación se presenta la complejidad O del peor caso, obtenido a partir de los tiempos de ejecución T(n) para el método que computa el factorial

formula general, siendo **a** y **b** constantes

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{a}, & \boldsymbol{si\ \ n \le 1} \\ \boldsymbol{b + T(n-1)}, & \boldsymbol{si\ n > 1} \end{cases}$$

$$\boldsymbol{T(n)} = \begin{cases} \boldsymbol{O(1)}, & \boldsymbol{si\ \ n \le 1} \\ \boldsymbol{O(1) + T(n-1)}, & \boldsymbol{si\ n > 1} \end{cases}$$

T(n) = b+T(n-1)	vez 1
T(n) = b+(b+T(n-2)) = 2b+T(n-2)	vez 2
T(n) = 2b+(b+T(n-3)) = 3b+T(n-3)	vez 3
…..	
T(n) = kb+T(n-k)	vez k

T(n) = *O*(1)+T(n-1)	vez 1
T(n) = *O*(1)+(*O*(1)+T(n-2)) = 2*O*(1)+T(n-2)	vez 2
T(n) = 2*O*(1)+(*O*(1)+T(n-3)) = 3*O*(1)+T(n-3)	vez 3
…..	
T(n) = k*O*(1)+T(n-k)	vez k

Se tiene en cuenta que el caso base se da en T(1), para despejar k
Como n-k = 1 entonces **k = n-1**

T(n) = (n-1)b+T(n-(n-1))=(n-1)b+T(1)=(n-1)b+a
T(n) = *O*(n)

T(n) = (n-1)O(1)+T(n-(n-1))=(n-1)O(1)+T(1)=O(n)+O(1)
T(n) = *O*(n)

- La **complejidad obtenida se esperaba**, porque el método recursivo se invoca para cada uno de los n_i valores **inferiores** a *n* y que el **caso base** es de complejidad **O(1)**.

Tiempo de Ejecución y Orden de Complejidad

Ejemplo: obtener *T(n)* del siguiente algoritmo, siendo *A* un arreglo con *n* datos, *p* y *r* son posiciones dentro del arreglo y la complejidad del método OPERA es O(n).

- Con la invocación *OPERA_ALGO(A,0,A.length-1)* se opera con los n datos

$O(1) + 2T(n/2) + O(n)$

$O(n) + 2T(n/2)$

```
OPERA_ALGO(A, p, r)          } O(1)
if p < r
    then q ← ⌊(p + r)/2⌋
         OPERA_ALGO(A, p, q)
         OPERA_ALGO(A, q + 1, r)
         OPERA(A, p, q, r)
```

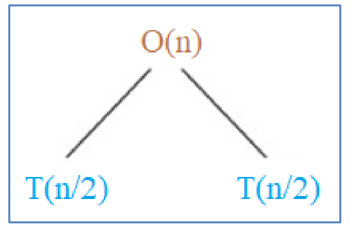
$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(n/2) & \text{if } n > 1 \end{cases}$$

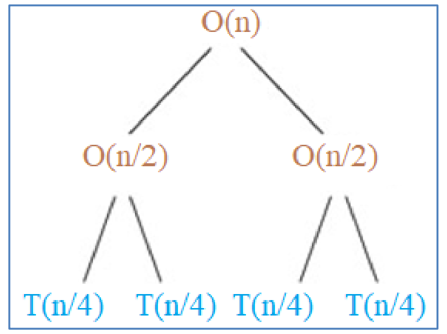
Tiempo de Ejecución y Orden de Complejidad

- La **complejidad** también puede calcularse realizando una **sustitución** de **forma grafica** (**en forma de árbol**). En este caso, la **raíz** de cada subárbol se corresponde a la **función o complejidad** que acompaña la **recursión** y las **ramas** cada uno de los **llamados recursivos**.

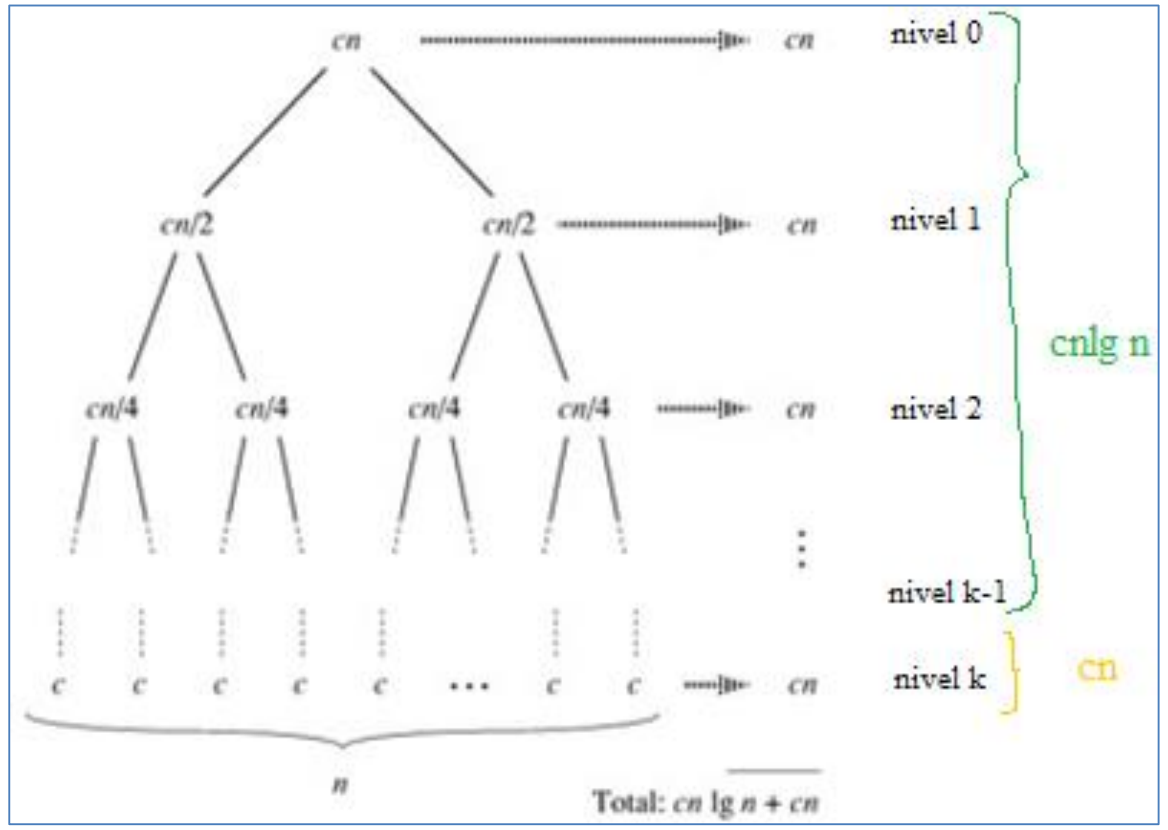
Ejemplo: se muestran las sustituciones iniciales y final, para calcular la complejidad de la recurrencia.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n) + 2T(n/2) & \text{if } n > 1 \end{cases}$$





Se tiene en cuenta que el caso base se da en T(1), para despejar k
Como $n / 2^k = 1$, luego $n = 2^k$, por lo cual **$k = \lg_2 n$**



- Como T(n) = cn lg n + cn entonces es O(nlg n)

Tiempo de Ejecución y Orden de Complejidad

Ejercicio1: determinar la complejidad ***O*** de los siguientes tiempos de ejecución de métodos recursivos.

$$T(n) = \begin{cases} O(\log n) & \text{si } n = 0 \\ O(n) + T(n-1) & \text{si } n > 0 \end{cases}$$

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ O(n) + 3T(\frac{n}{3}) & \text{si } n > 1 \end{cases}$$

$$T(n) = \begin{cases} O(\log n) & \text{si } n = 0 \\ O(1) + T(n-1) & \text{si } n > 0 \end{cases}$$

$$T(n) = \begin{cases} O(1) & \text{si } n = 1 \\ O(1) + 2*T(\frac{n}{2}) & \text{si } n > 1 \end{cases}$$

Ejercicio2: determinar el computo ***T(n)*** y su correspondiente complejidad ***O*** del peor caso, de los ejercicios propuestos con antelación y los ejercicios propuestos en el taller del caso.

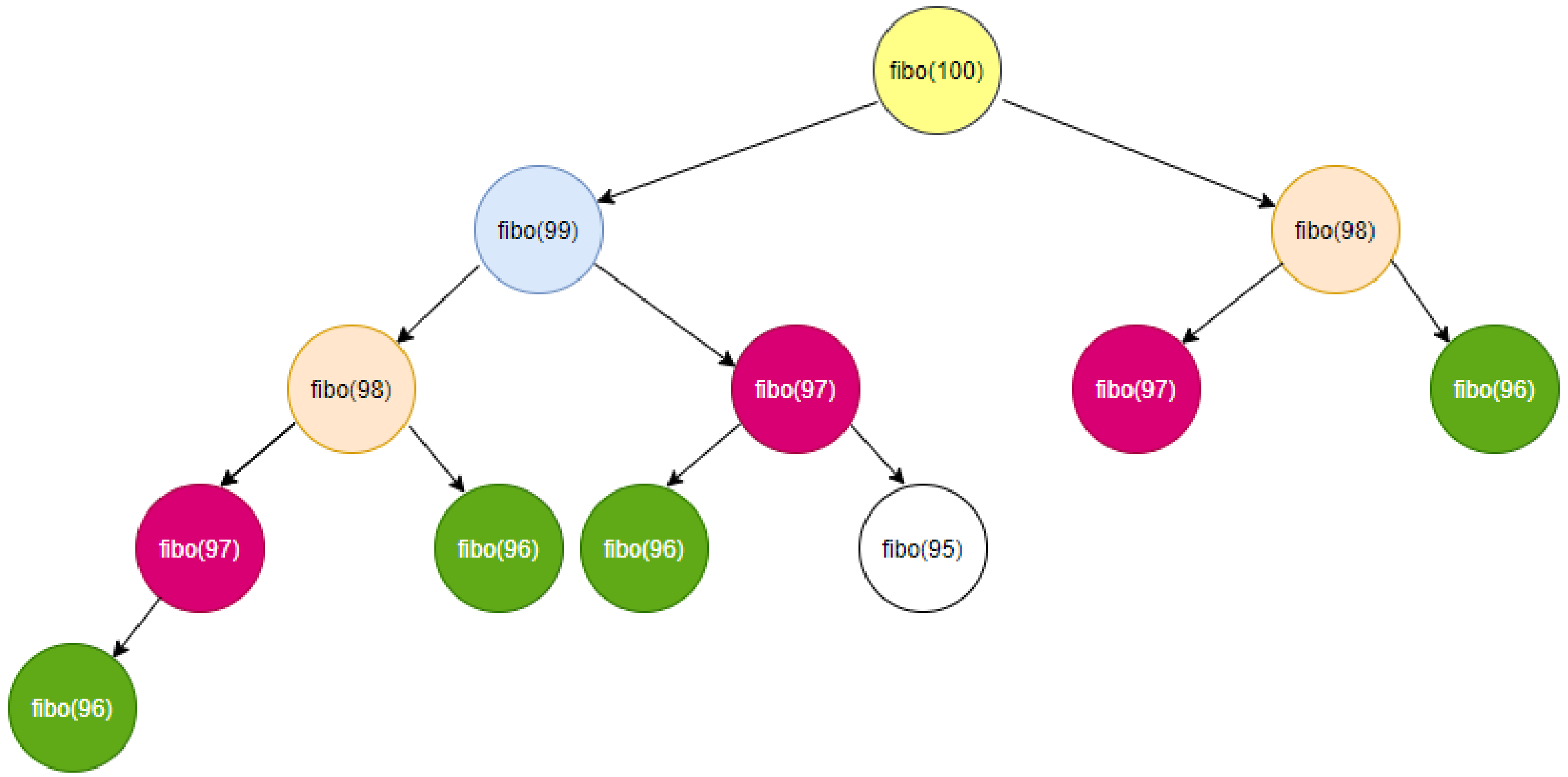
Inconvenientes en las Recurrencias

- Hay **recurrencias** en las que se **solapan los cómputos**.

Ejemplo: a continuación se presentan algunos solapamientos de la función ***Fibonacci*** operando con n=100.

$$fibo(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ fibo(n-1) + fibo(n-2) & \text{si } n > 1 \end{cases}$$

```java
public long fibo(int n)
{
    if (n<=1)
        return 1;
    else
        return fibo(n-1)+fibo(n-2);
}
```

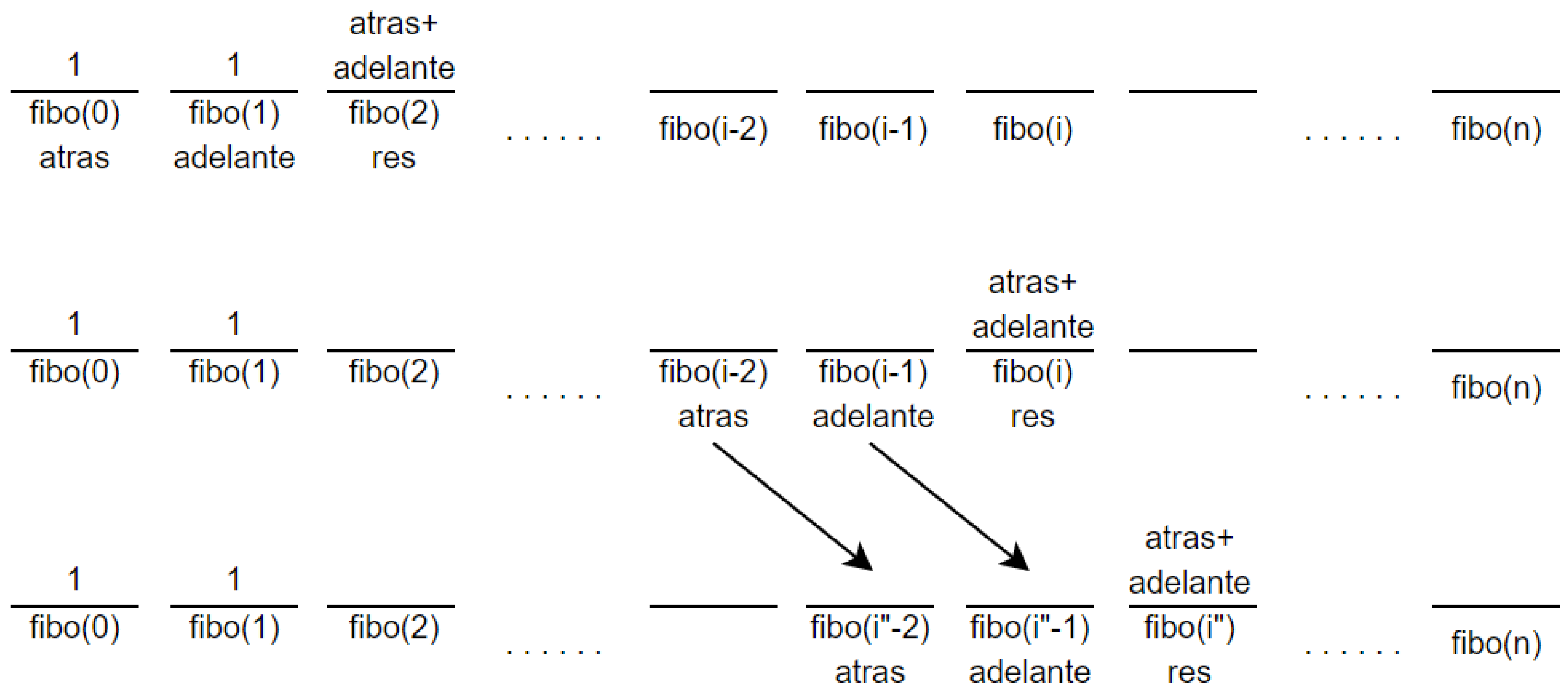


- En estos casos se propone una **implementación** equivalente que sea **iterativa** de **abajo-arriba** (que calcule **desde** los **casos bases** hasta el **valor solicitado**).

Inconvenientes en las Recurrencias

Ejemplo: se presenta la idea y posterior implementación de una versión iterativa de la función ***Fibonacci***.

$$fibo(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ fibo(n-1) + fibo(n-2) & \text{si } n > 1 \end{cases}$$




```
function fiboIterativo(n){
    if (n<=1) {
        return 1;
    }else{
        let atras = 1;
        let adelante = 1;
        let res = 0;
        for (let i=2; i<=n; i++){
            res = atras + adelante;
            atras = adelante;
            adelante = res;
        }
        return res;
    }
}
```

04

BUEN VIENTO Y BUENA MAR !!!

uao