Tema 4. Recursión

Estructura de Datos y Algoritmos

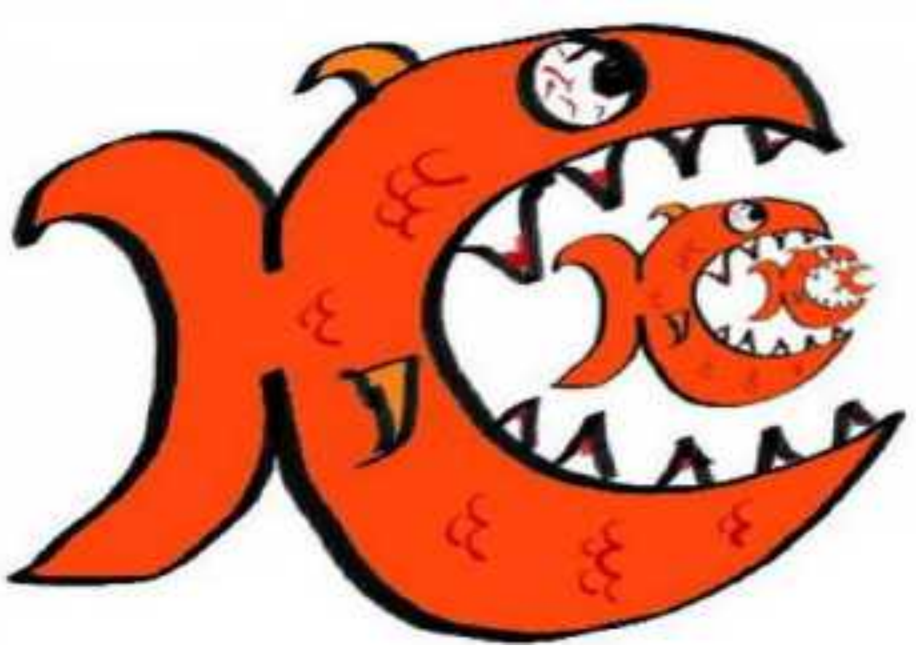
Material elaborado por los profesores de la asignatura (Grupo LABDA)

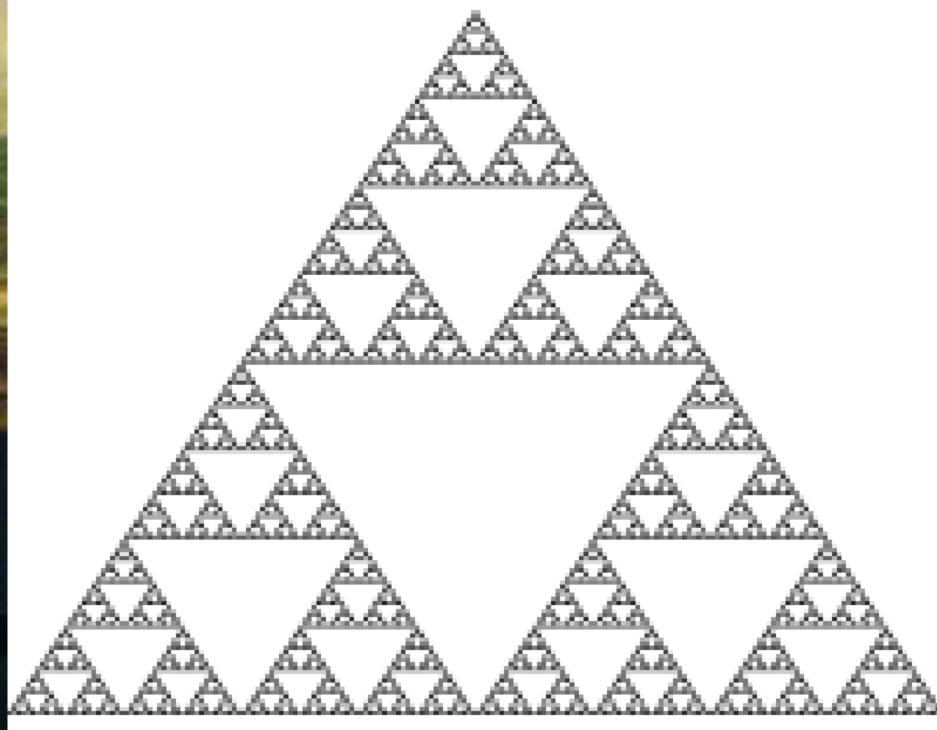
Objetivos

- Al final de la clase, los estudiantes deben ser capaces de:

1) Describir el concepto de recursividad y dar ejemplos de su uso
2) Identificar el caso base y el caso general de una función recursiva
3) Escribir una función recursiva para resolver un problema
4) Comparar las soluciones iterativas y recursivas para los problemas fundamentales


Pistacchi
Design





Qué es la recursión?

- Un método se llama a sí mismo
- Algunas estructuras de datos pueden tener una estructura recursiva (lista o árboles)
- Cercano a la inducción matemática.

Los tres niveles de recursión

1. Un algoritmo recursivo debe tener un caso base
2. Un algoritmo recursivo debe cambiar su estado y avanzar hacia el caso base
3. Un algoritmo recursivo debe llamarse recursivamente

Cómo resolver problemas por recursión?

- Cada método recursivo tiene dos partes:
  - **CASO(S) BASE**: Caso(s) tan simple que se pueden resolver directamente
  - **CASO(S) RECURSIVO**: Caso(s) más complejo y se hace uso de la recursividad para:
    - Dividir el problema en subproblemas más pequeños y,
    - Combinar en una solución el problema más grande

Multiplica 2 números usando la suma

$$5 \times 3 = 15 = 5 + 5 + 5$$

Multiplica 2 números usando la suma

5 x 3 = 15 = 5 + 5 + 5

```
int multiply(int x, int y) {
    int result=0;
    for (int i=1; i<=y;i++) {
        result=result+x;
    }
    return result;
}
```

Multiplica 2 números usando la suma

$5 \times 3 = 15 = 5 + 5 + 5$

```
int multiplyRec(int x, int y) {



}
```

1) Determina el caso base(s)

Multiplica 2 números usando la suma

5 x 3 = 15 = 5 + 5 + 5

```
int multiplyRec(int x, int y) {

        if (y==1) return x;



}
```

1) Determinar el caso(s) base

Multiplica 2 números usando la suma

5 x 3 = 15 = 5 + 5 + 5

```
int multiplyRec(int x, int y) {
        if (y==1) return x;


}
```

2) Determina el caso(s) recursivo

Multiplica 2 números usando la suma

5 x 3 = 15 = 5 + 5 + 5

```
int multiplyRec(int x, int y) {

    if (y==1) return x;
    else
            return x + multiplyRec(x,y-1);
}
```

2) Determina el caso(s) recursivo

Seguimiento de recursión

```
if (y==1) return x;
else      return x + multiplyRec(x,y-1);
```

multiplyRec(5,3)

return 5 + multiplyRec(5,2);

Seguimiento de recursión

multiplyRec(5,3)

```
if (y==1) return x;
else     return x + multiplyRec(x,y-1);
```

return 5 + multiplyRec(5,2);

return 5 + multiplyRec(5,1);

Seguimiento de recursión

multiplyRec(5,3)

```
if (y==1) return x;
else      return x + multiplyRec(x,y-1);
```

return 5 + multiplyRec(5,2);

return 5 + multiplyRec(5,1);

return 5;

Seguimiento de recursión

multiplyRec(5,3)

```
if (y==1) return x;
else      return x + multiplyRec(x,y-1);
```

return 5 + multiplyRec(5,2);

return 5 + multiplyRec(5,1);

return 5;

Combinar desde el caso base

Seguimiento de recursión

multiplyRec(5,3)

```
if (y==1) return x;
else      return x + multiplyRec(x,y-1);
```

return 5 + multiplyRec(5,1);

return 5 + 5;

return 5;

Combinar desde el caso base

Seguimiento de recursión

multiplyRec(5,3) = 15 **Final result**

return 5 + 10;

return 5 + 5;

return 5;

Problema Factorial

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4) → *First Call* → 4 * factorial(3)

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)

First Call

4 * factorial(3)

3 * factorial(2)

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)

First Call

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)

First Call

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1

Base case

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)

First Call ⇨

4 * factorial(3)

↓

3 * factorial(2)

↓

2 * factorial(1)

↓

1

1

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)

First Call

4 * factorial(3)

3 * factorial(2)

2*1

1

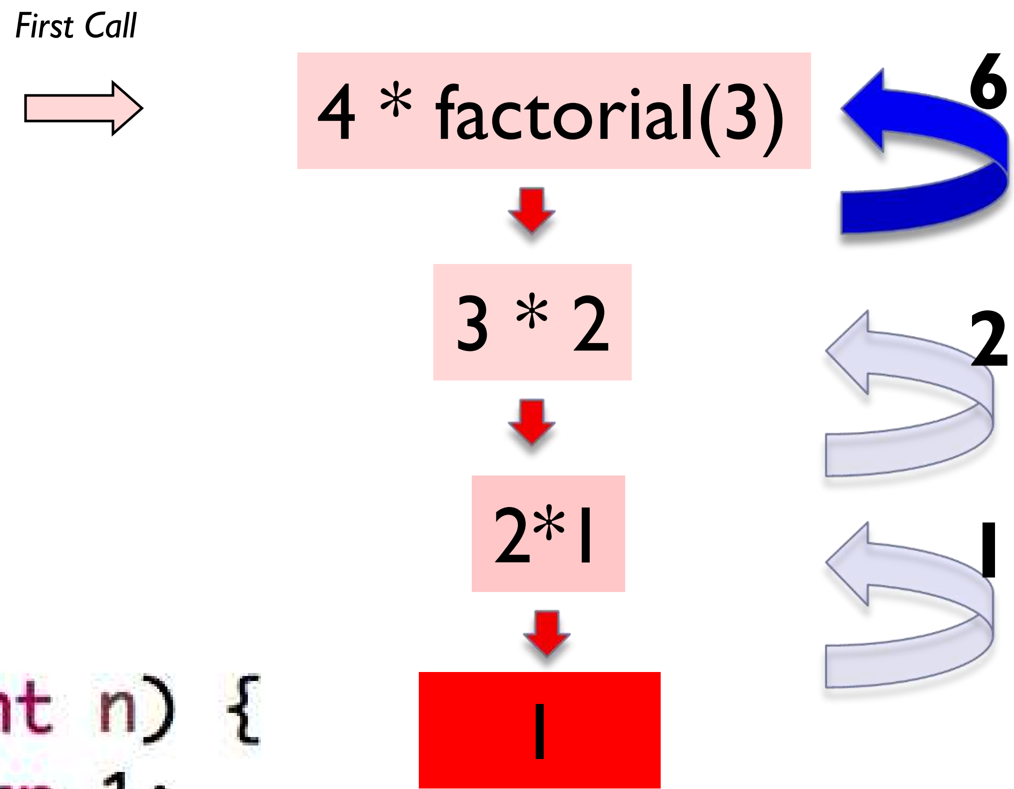
2

1

```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Tracing factorial

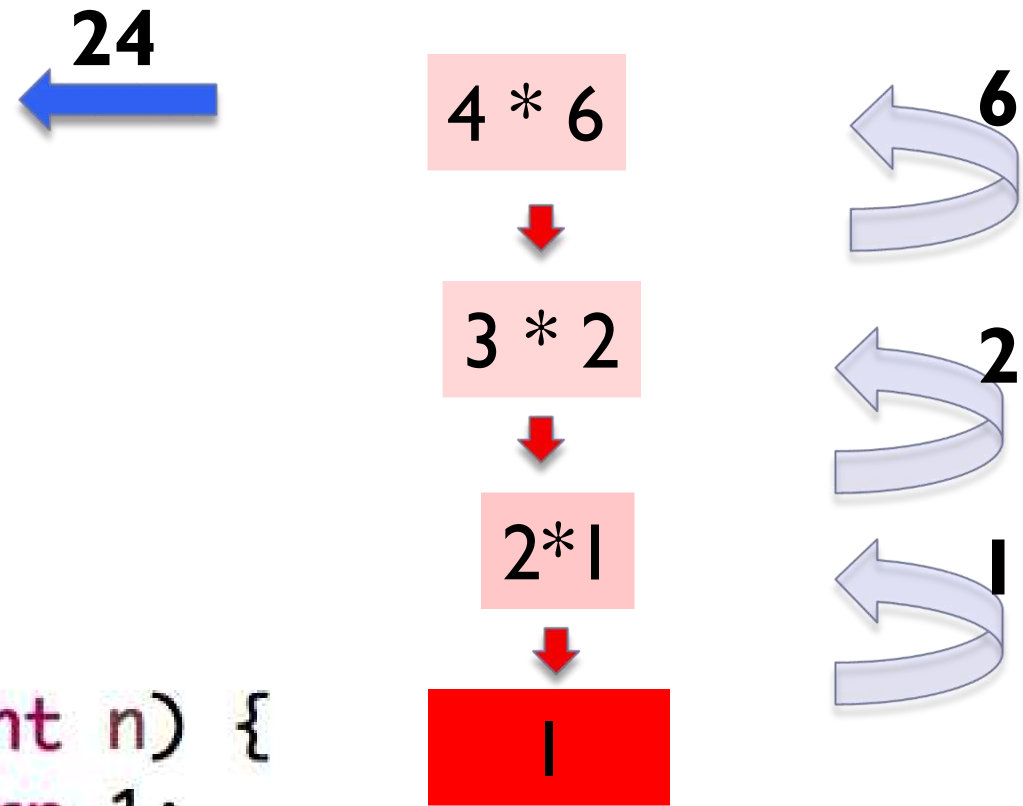
- factorialRec(4)




```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Seguimiento factorial

- factorialRec(4)




```
int factorialRec(int n) {
    if (n==1) return 1;
    else return n * factorialRec(n-1);
}
```

Suma de una lista de números

```
public int sumArray(int[] data) {
    int sum=0;
    for (int i=0; i<data.length;i++) {
        sum=sum + data[i];
    }
    return sum;
}
```

Suma de una lista de números

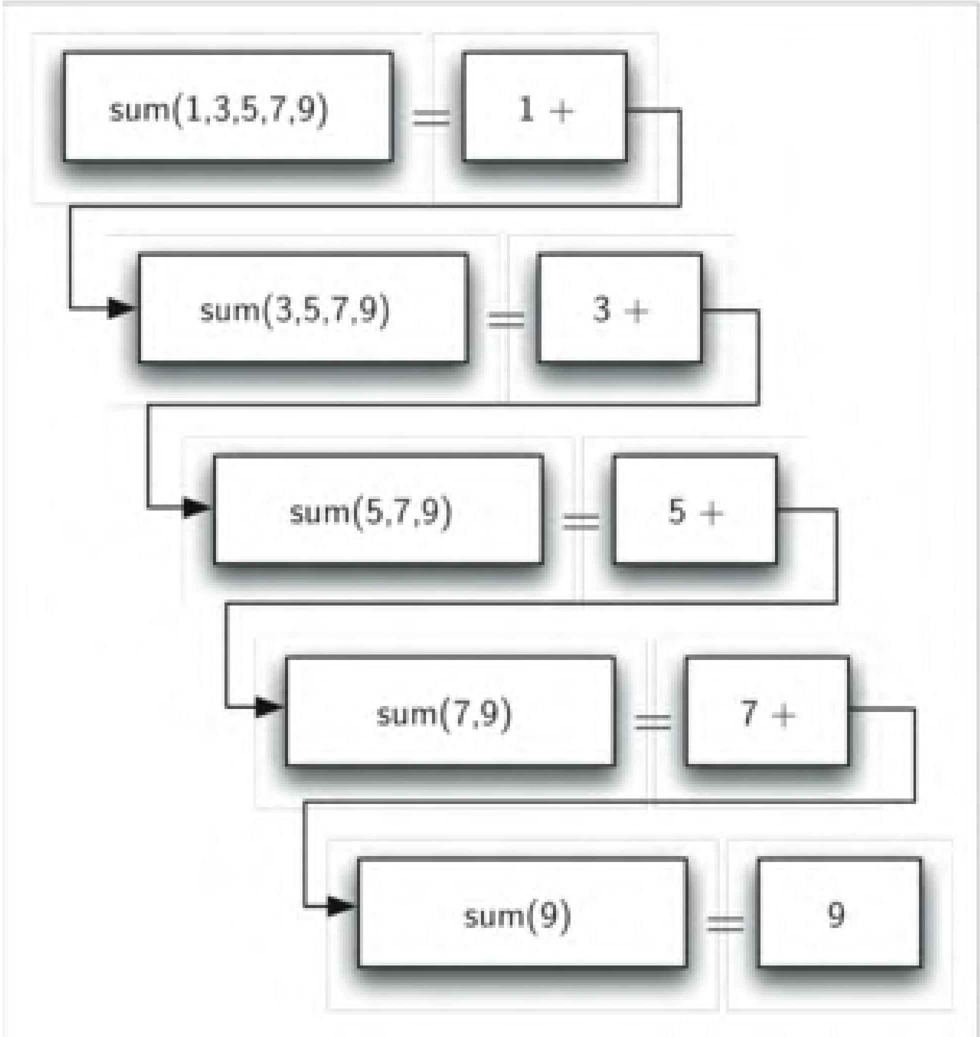
- Imaginar que no hay bucles.
- ¿Cómo se obtiene la suma de una lista de números?

Suma una lista de números

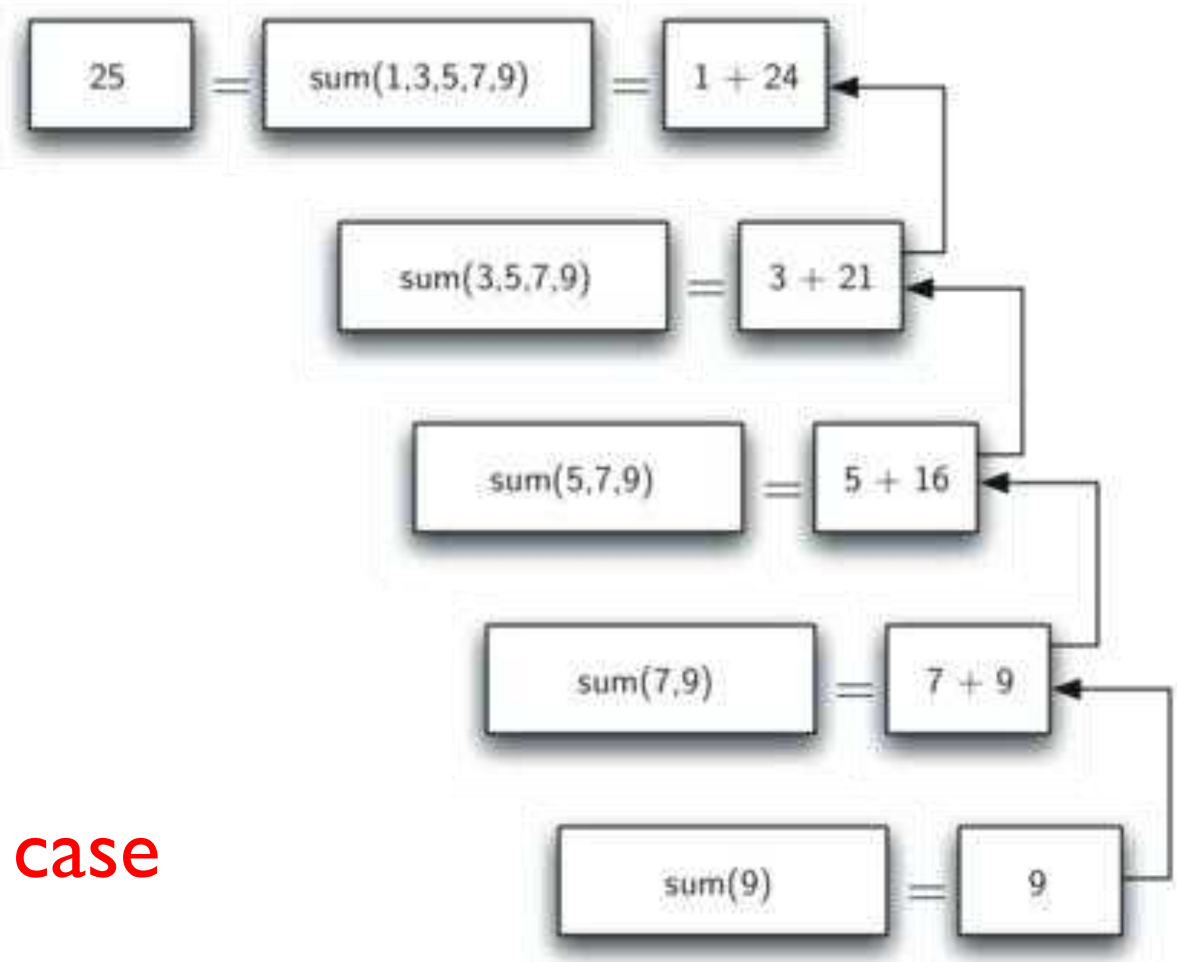
- Dada una lista [1,3,5,7,9]
- La suma de sus números puede ser expresada:

- (1 +(3 + (5 + (7 + 9))))
- Sum =(1 +(3 + (5 + 16)))
- Sum = (1 + (3 + 21))
- Sum = (1+24)
- Sum = 25

Suma de una lista de números



Suma una lista de números (solución recursiva)



Base case

Suma de una lista de números (solución recursiva)

```
public int sumArrayRec(int[] data) {
    return sumArrayRec(data,0);
}

public int sumArrayRec(int[] data, int i) {
    if (i==data.length-1) return data[i];
    else return data[i]+sumArrayRec(data,i+1);
}
```

Una pregunta

- ¿Cuántas llamadas recursivas se necesita para obtener la suma de estos números? 2,4,6,8,10?

Máximo común divisor (mcd)

- El mcd de dos enteros p y q es el mayor entero d que divide a p y q.

Máximo común divisor (mcd)

mcd(4032,1272)

$4032 = 2^6 * 3^2 * 7$

$1272 = 2^3 * 3^1 * 53$

=> $mcd(4032,1272) = 2^3 * 3^1 = 24$

Máximo común divisor (mcd)

- Encontrar el entero mayor d que divide a p y q
- Euclid's algorithm
- $$mcd(a,b) = \begin{cases} a & \text{if } b=0 \\ mcd(b, a\%b) & \text{otherwise} \end{cases}$$

Máximo común divisor (mcd)

$$
\begin{aligned}
mcd(4032, 1272) &= mcd(1272, 216) \\
&= mcd(216, 192) \\
&= mcd(192, 24) \\
&= mcd(24, 0) = 0
\end{aligned}
$$

Máximo común divisor (mcd)

```
//assume a>b, y a,b>=0
int gcd(int a, int b) {
    if (b==0) return a;
    else return gcd(b,a%b);
}
```

Números Fibonacci

1,1,2,3,5,8,13,21,34,55,89,144,233,377 ...

$$Fib(n) = \begin{cases} 1 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ Fib(n-1) + Fib(n-2) & \text{if } n>1 \end{cases}$$

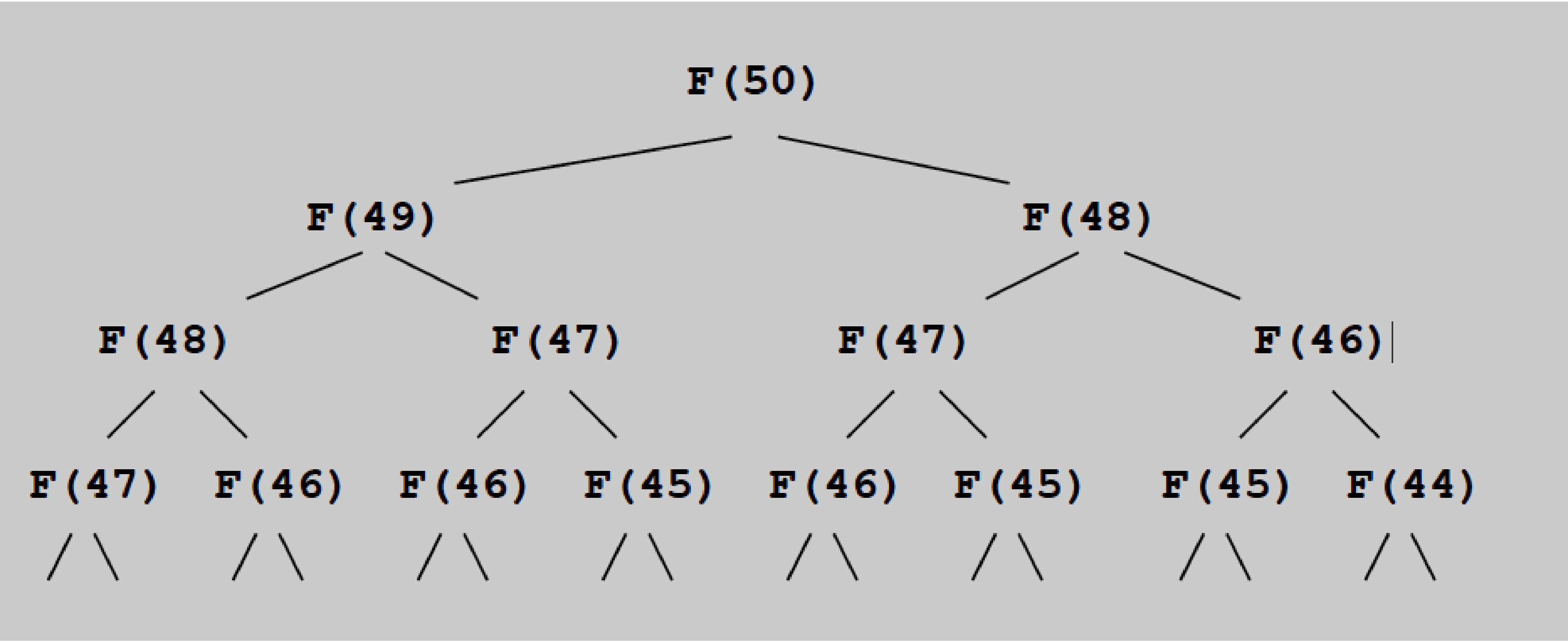
Números Fibonacci

```
Int fib(int n)
{
  if ((n == 0) || (n == 1))
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

¿Es esta una forma eficiente de calcular F(50)?

Números Fibonacci

No!! Este código es muy ineficiente



Una forma más eficiente de calcular los números de Fibonacci

```
public static long fibo(int n) {
    if (n == 0) return 0;

    long[] F = new long[n+1];
    F[0] = 0;
    F[1] = 1;

    for (int i = 2; i <= n; i++)
        F[i] = F[i-1] + F[i-2];

    return F[n];
}
```

Programación dinámica

Iteración vs Recursión

- Un bucle también es un proceso iterativo
- Un método recursivo es más matemáticamente elegante que usar un bucle.
- Los bucles son más eficientes que los métodos recursivos
- Todos los métodos recursivos se pueden resolver usando una solución iterativa
- No todos los problemas se pueden resolver usando recursión

Recursión

- **Ventaja:** enfoque fácil y ordenado => paradigma de programación de gran alcance
- **Desventaja:** tiene peor complejidad de tiempo que los bucles (porque cada llamada de función requiere memoria múltiple para almacenar la dirección interna del método)

- *Iterar es humano, hacer recursividad, divino*