Análisis de algoritmos

Recursividad

Recursividad

La recursividad es un concepto fundamental en matemáticas y en computación.

- Es una alternativa diferente para implementar estructuras de repetición (ciclos).
- Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de reglas no ambiguas.

Función recursiva

Las funciones recursivas se componen de:

- Caso base: una solución simple para un caso particular (puede haber más de un caso base)
- Ej. Factorial de un numero: 5! = 5* 4!
- La base si es 0 factorial es 1.

Función recursiva

- Caso recursivo: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base.
- Los pasos que sigue el caso recursivo son los siguientes:
  1. El procedimiento se llama a sí mismo
  2. El problema se resuelve, tratando el mismo problema pero de tamaño menor
  3. La manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará

Ejemplo: factorial

Escribe un programa que calcule el factorial (!) de un entero no negativo. He aquí algunos ejemplos de factoriales:

- 1! = 1
- 2! = 2! 2*1
- 3! = 3*2! 3*2*1
- 4! = 4*3! 4*3*2*1
- 5! = 5*4! 5*4*3*2*1

Ejemplo: factorial (iterativo - repetetitivo)

```
int factorial (int n)
comienza
    fact ← 1
    for i ← 1 hasta n
        fact ← i * fact
    return fact
termina
```

```
int factorial (int n)
{ int fact = 1;
  for  (int i = 1; i <= n; i++)
  fact = i * fact;
  return fact;
  }
```

Ejemplo: factorial (recursivo)

```
int factorial (int n)
comienza
   if n = 0
          return 1
   else
          return factorial (n-1)*n
termina
```

```
int factorial (int n)
{ if n == 0
      return 1
  else
     return factorial (n-1) * n
  }
```

Ejemplo:

- A continuación se puede ver la secuencia de factoriales.

0! = 1		
1! = 1	= 1*1	= 1 * 0!
2! = 2	= 2*1	= 2 * 1!
3! = 6	= 3*2*1	= 3 * 2!
4! = 24	= 4*3*2*1	= 4 * 3!
5! = 120	= 5**4*3*2*1	= 5 * 4!
N! =	= N * (N – 1)!	

Solución

Aquí podemos ver la secuencia que toma el factorial

$$N! = \begin{cases} 1 & \text{Si } N = 0 \text{ (base)} \\ N * (N - 1)! & \text{si } N > 0 \text{ (recursión)} \end{cases}$$

Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción.

La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

Solución Recursiva

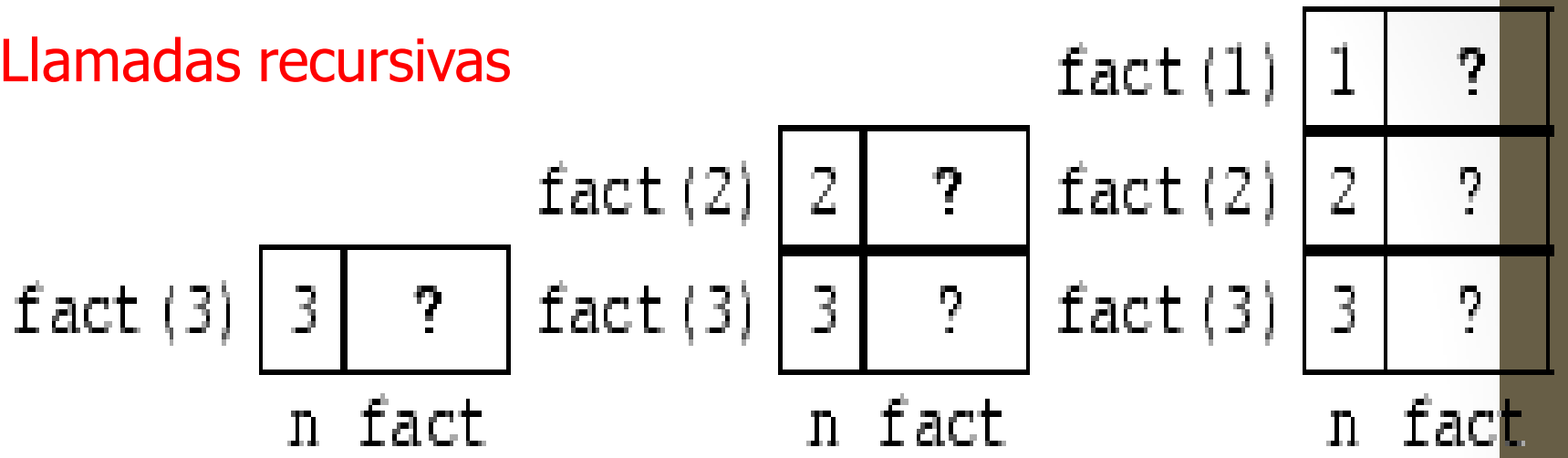
Dado un entero no negativo x, regresar el factorial de x fact:
Entrada n entero no negativo,
Salida:entero.

```
int fact (int n)
{
  if   (n   ==   0)
         return  1;
  else
         return  fact(n – 1) * n ;
}
```

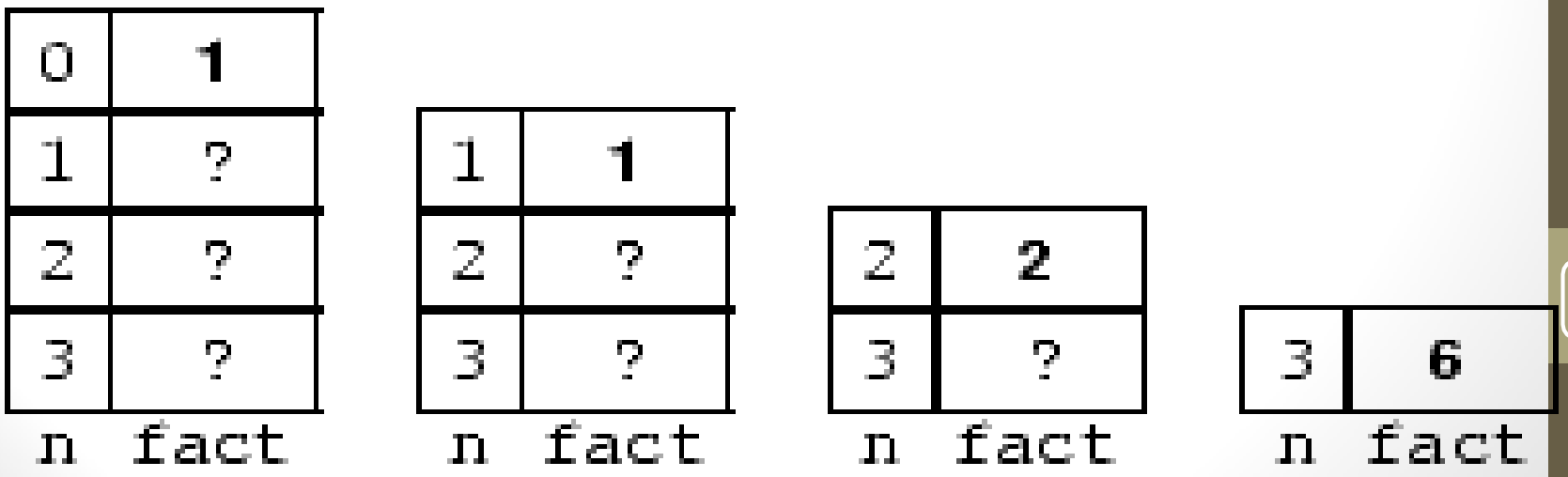
Es importante determinar un caso base, es decir un punto en el cual existe una condición por la cual no se requiera volver a llamar a la misma función.

¿Cómo funciona la recursividad?

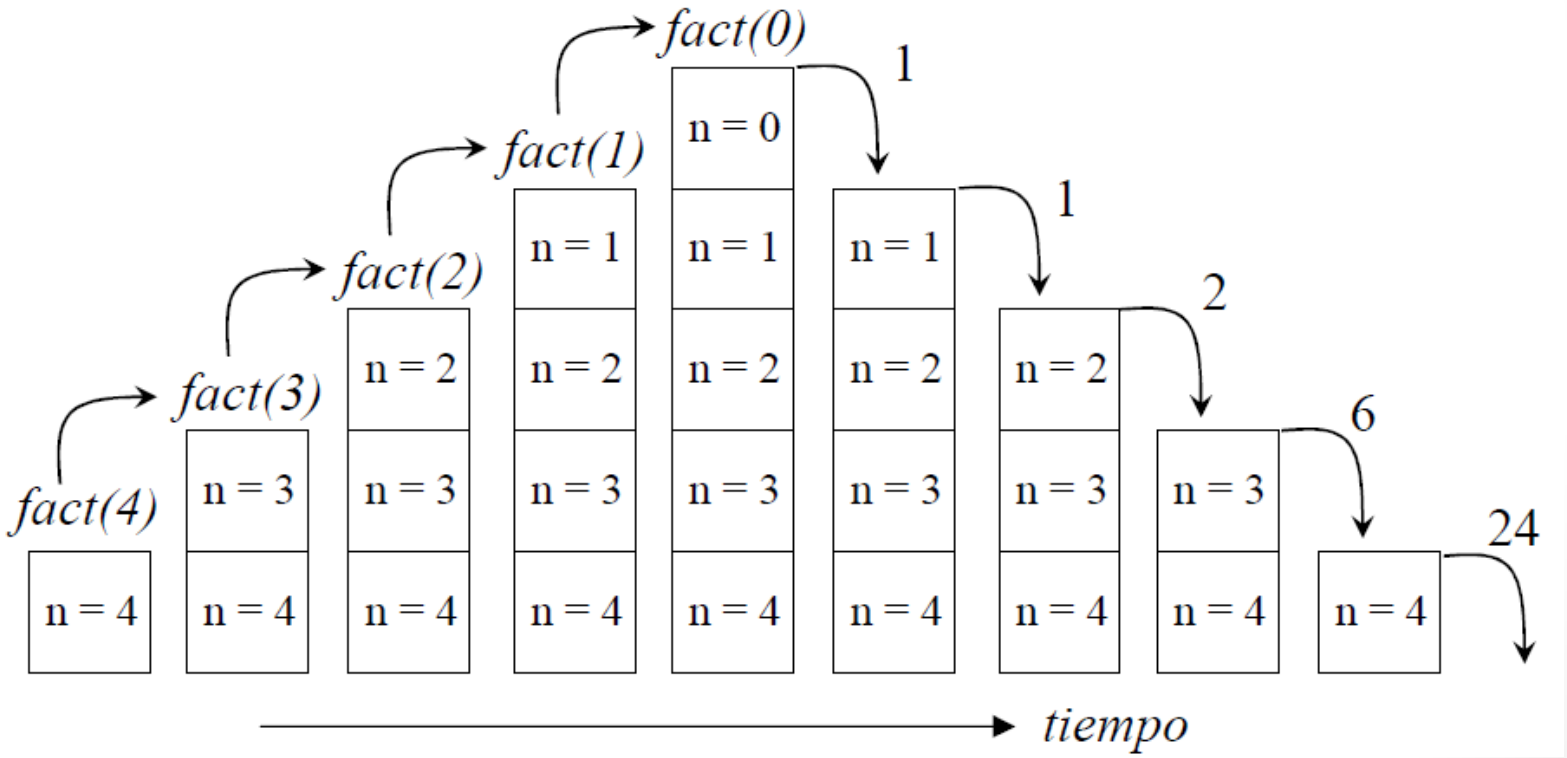
Llamadas recursivas



Resultados de las llamadas recursivas



¿Cómo funciona la recursividad?



¿Por qué escribir programas recursivos?

- Son mas cercanos a la descripción matemática.
- Generalmente mas fáciles de analizar
- Se adaptan mejor a las estructuras de datos recursivas.
- Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

Factible de utilizar recursividad

- Para simplificar el código.
- Cuando la estructura de datos es recursiva ejemplo : árboles.

No factible utilizar recursividad

- Cuando los métodos usen arreglos largos.
- Cuando el método cambia de manera impredecible de campos.
- Cuando las iteraciones sean la mejor opción.

Otros conceptos

- Cuando un procedimiento incluye una llamada a sí mismo se conoce como **recursión directa.**

- Cuando un procedimiento llama a otro procedimiento y éste causa que el procedimiento original sea invocado, se conoce como **recursión indirecta.**

Ejemplo: Serie de Fibonacci

Valores: 0, 1, 1, 2, 3, 5, 8...

Cada término de la serie suma los 2 anteriores. Fórmula recursiva

fib(n) = fib (n - 1) + fib (n - 2)

Caso base: Fib (0)=0; Fib (1)=1

Caso recursivo: Fib (i) = Fib (i -1) + Fib(i -2)

```
int fib(int n)
{
  if (n <= 1) return n                    //condición base
  else
   return fib(n-1)+fib(n-2)               //condición recursiva
  }
```

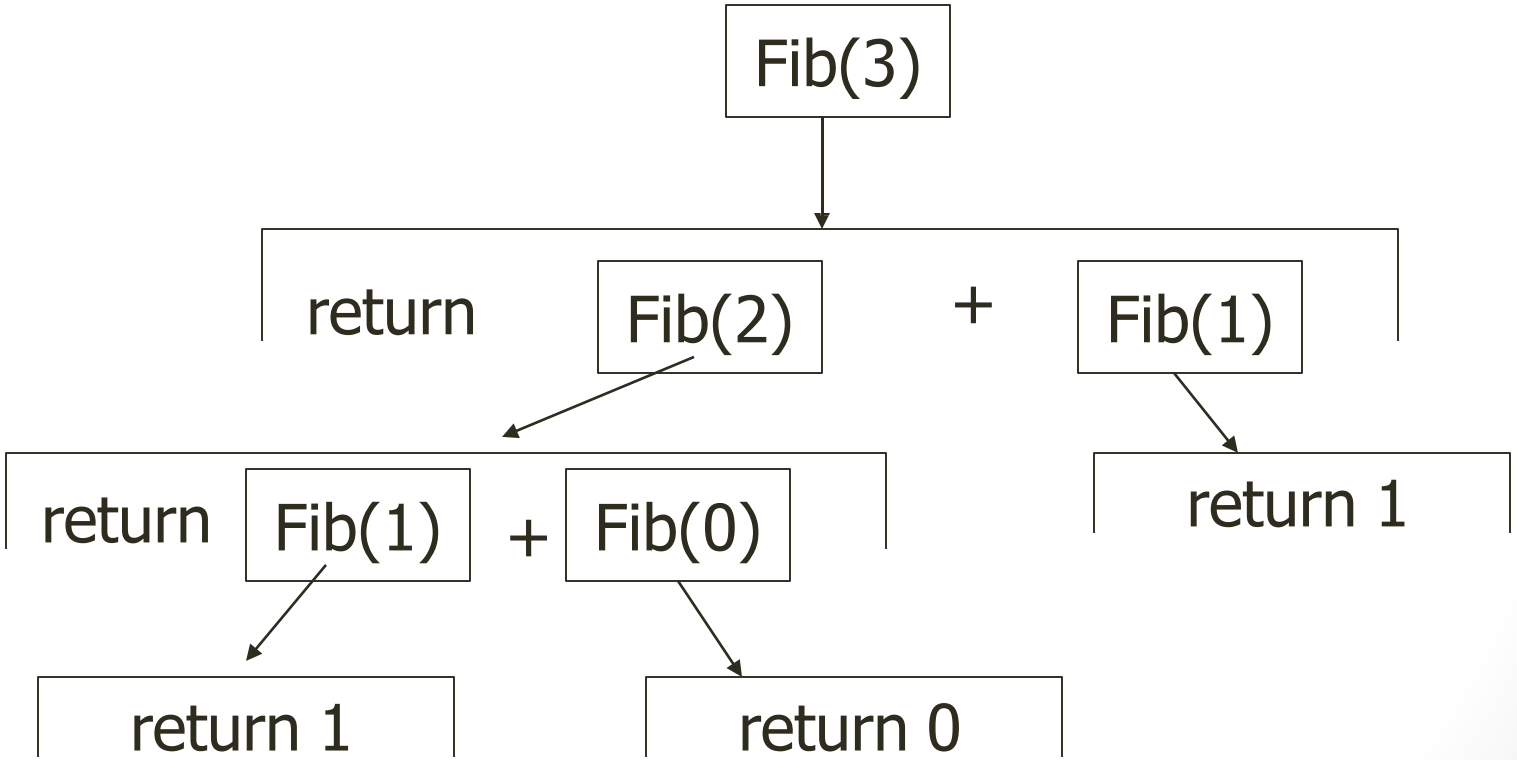
fibonacci

```
#fibonnacci
def fib(n):
    if n<=1:
        return n
    else:
        return fib(n-1) + fib(n-2)

# PRINCIPAL
m=int(input("numero:"))
print(f'Fibonacci n-simo de {m} es {fib(m)}')
```

Ejemplo: Serie de Fibonacci

Traza del cálculo recursivo



Recursión vs iteración

Repetición

Iteración: ciclo explícito (se expresa claramente)

Recursión: repetidas invocaciones a método

Terminación

Iteración: el ciclo termina o la condición del ciclo falla

Recursión: se reconoce el caso base

En ambos casos podemos tener ciclos infinitos

Considerar que resulta más positivo para cada problema

- LA RECURSIVIDAD SE DEBE USAR CUANDO SEA REALMENTE NECESARIA, ES DECIR, CUANDO NO EXISTA UNA SOLUCIÓN ITERATIVA SIMPLE.

Dividir para vencer

- Muchas veces es posible dividir un problema en subproblemas más pequeños, generalmente del mismo tamaño, resolver los subproblemas y entonces combinar sus soluciones para obtener la solución del problema original.

- Dividir para vencer es una técnica natural para las estructuras de datos, ya que por definición están compuestas por piezas. Cuando una estructura de tamaño finito se divide, las últimas piezas ya no podrán ser divididas.