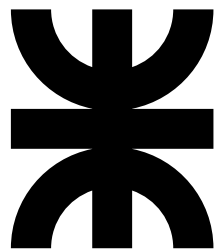
Informática II

Recursión

Gonzalo F. Pérez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2018 –

Introducción

- Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma **disciplinada** y **jerárquica**.

Introducción

- Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma **disciplinada** y **jerárquica**.
- Para la resolución de algunos problemas es útil contar con funciones que se llaman a sí mismas

Introducción

- Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma **disciplinada** y **jerárquica**.
- Para la resolución de algunos problemas es útil contar con funciones que se llaman a sí mismas

Función recursiva

Es una función que se llama a sí misma, ya sea directa o indirectamente, a través de otra función.

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- Existe un **caso base** (la función solo resuelve este caso).

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- Existe un **caso base** (la función solo resuelve este caso).
- Para un problema complejo existe una versión del mismo pero más sencillo. Entonces, se divide el problema:

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- Existe un **caso base** (la función solo resuelve este caso).
- Para un problema complejo existe una versión del mismo pero más sencillo. Entonces, se divide el problema:
  1. una parte que se sabe resolver

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- Existe un **caso base** (la función solo resuelve este caso).
- Para un problema complejo existe una versión del mismo pero más sencillo. Entonces, se divide el problema:
  1. una parte que se sabe resolver
  2. una parte que no se sabe resolver pero parecido al problema original

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- Existe un **caso base** (la función solo resuelve este caso).
- Para un problema complejo existe una versión del mismo pero más sencillo. Entonces, se divide el problema:
  1. una parte que se sabe resolver
  2. una parte que no se sabe resolver pero parecido al problema original
- Terminación de la recursión. Convergencia al caso base.

Recursión

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

con $1! = 0! = 1$.

Recursión

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

con $1! = 0! = 1$.

Ejemplo: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Recursión

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 1$$

con $1! = 0! = 1$.

Ejemplo: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Factorial de un número entero **number** mayor a cero de *forma iterativa*

```
1   factorial = 1;
2   for(counter = number; counter >= 1; counter--)
3     factorial *= counter;
```

Recursión

Ejemplo 1: cálculo de factorial – recursivo

Definición recursiva

$$n! = n \cdot (n-1)!$$

Recursión

Ejemplo 1: cálculo de factorial – recursivo

Definición recursiva

$$n! = n \cdot (n-1)!$$

Ejemplo: 5!

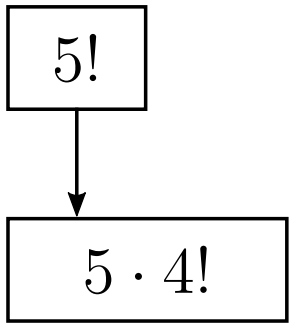
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

Recursión

Ejemplo 1: cálculo de factorial – recursivo

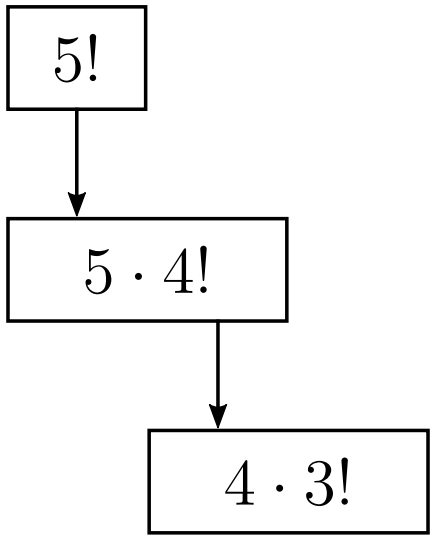
Recursión

Ejemplo 1: cálculo de factorial – recursivo



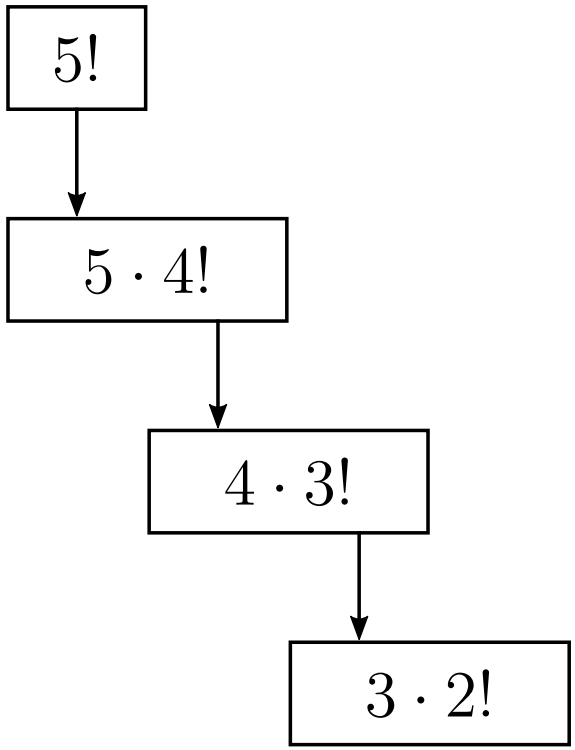
Recursión

Ejemplo 1: cálculo de factorial – recursivo



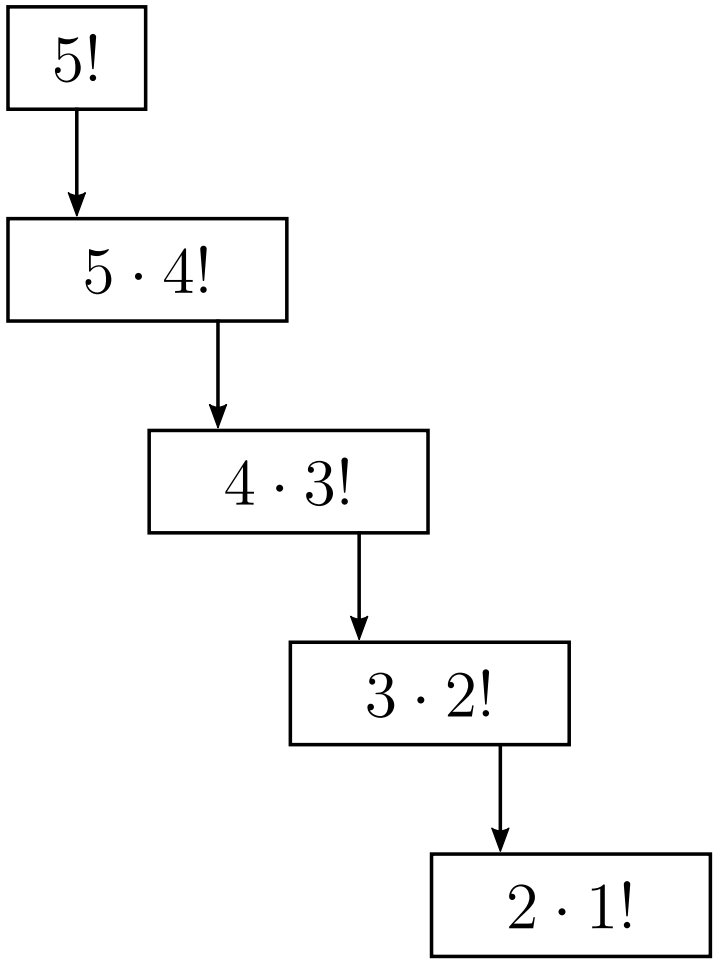
Recursión

Ejemplo 1: cálculo de factorial – recursivo



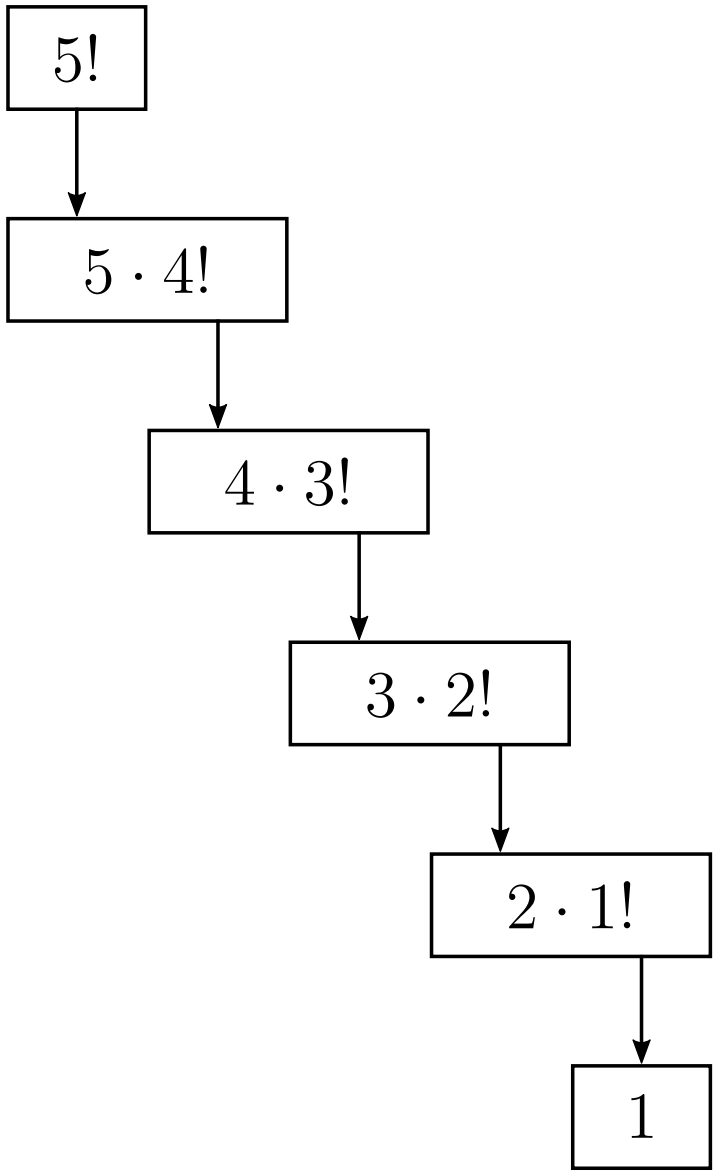
Recursión

Ejemplo 1: cálculo de factorial – recursivo



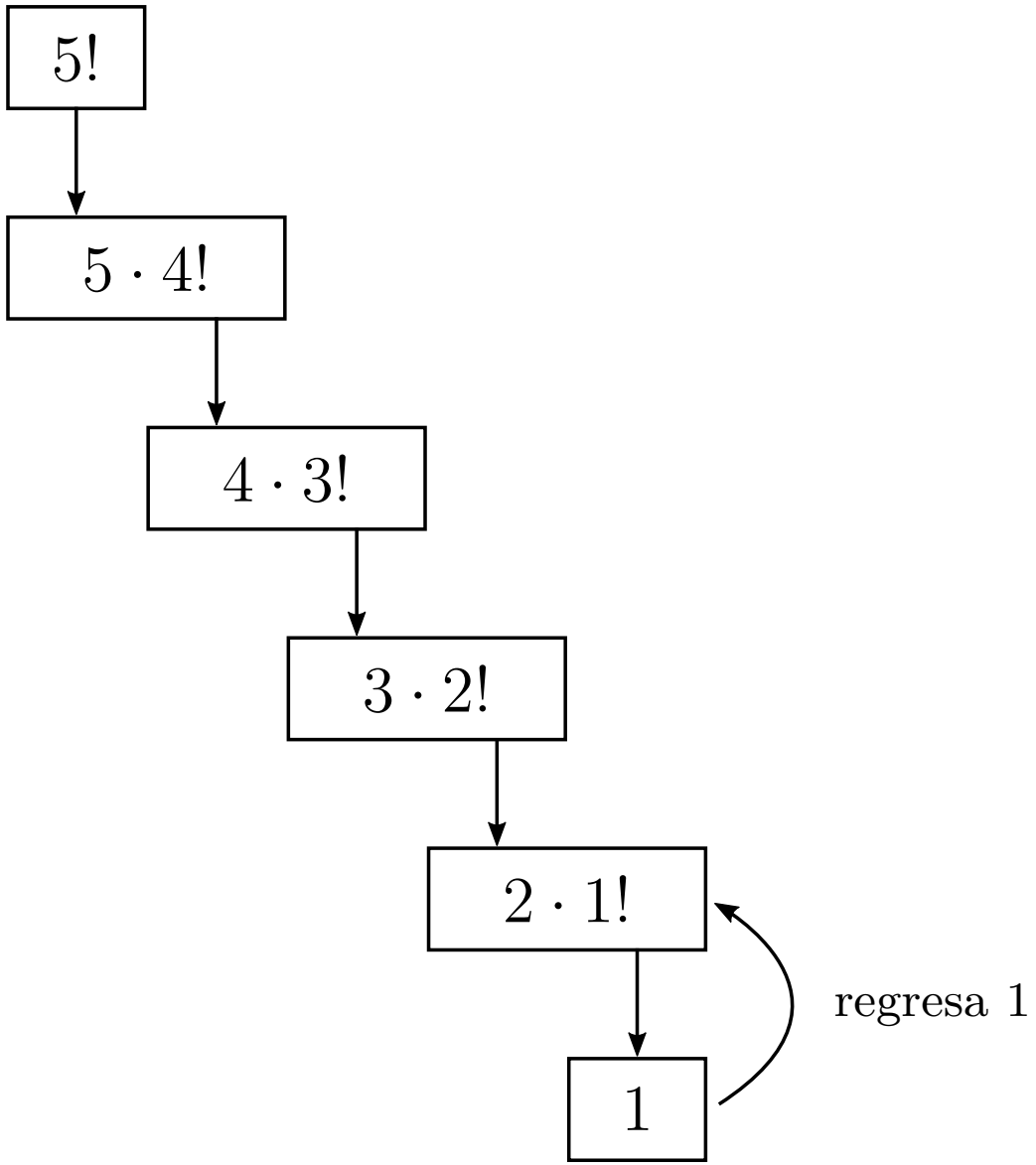
Recursión

Ejemplo 1: cálculo de factorial – recursivo



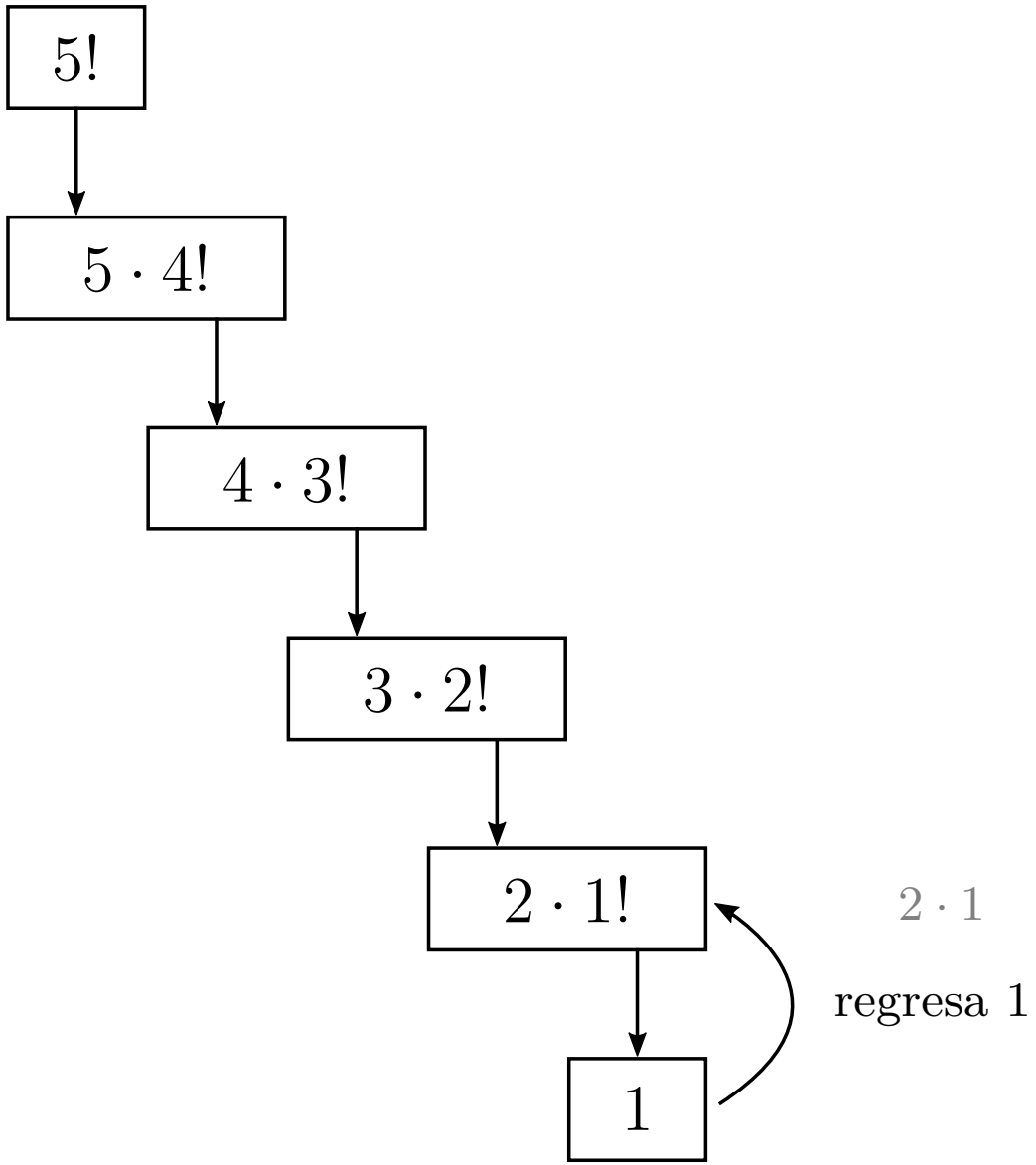
Recursión

Ejemplo 1: cálculo de factorial – recursivo



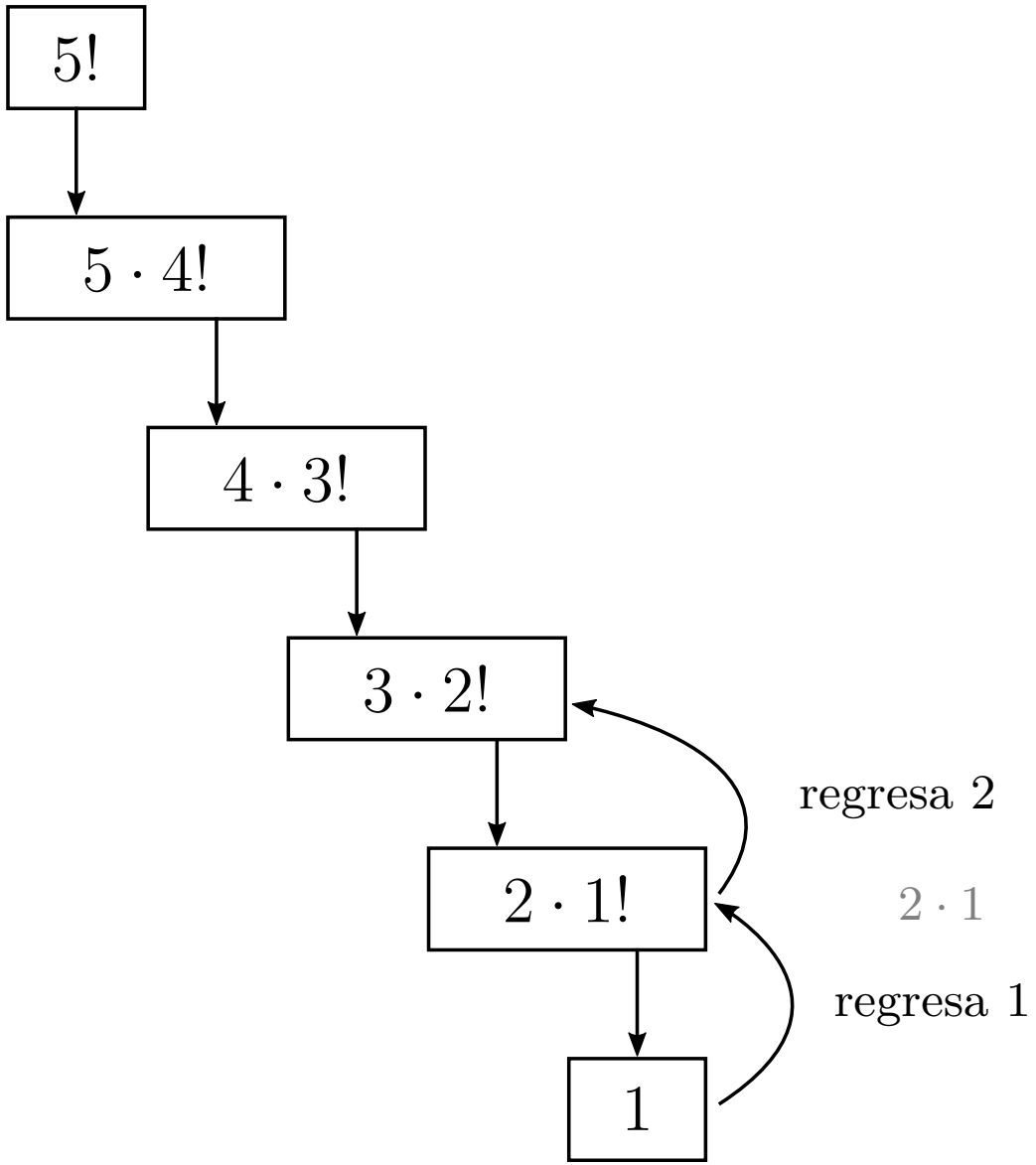
Recursión

Ejemplo 1: cálculo de factorial – recursivo



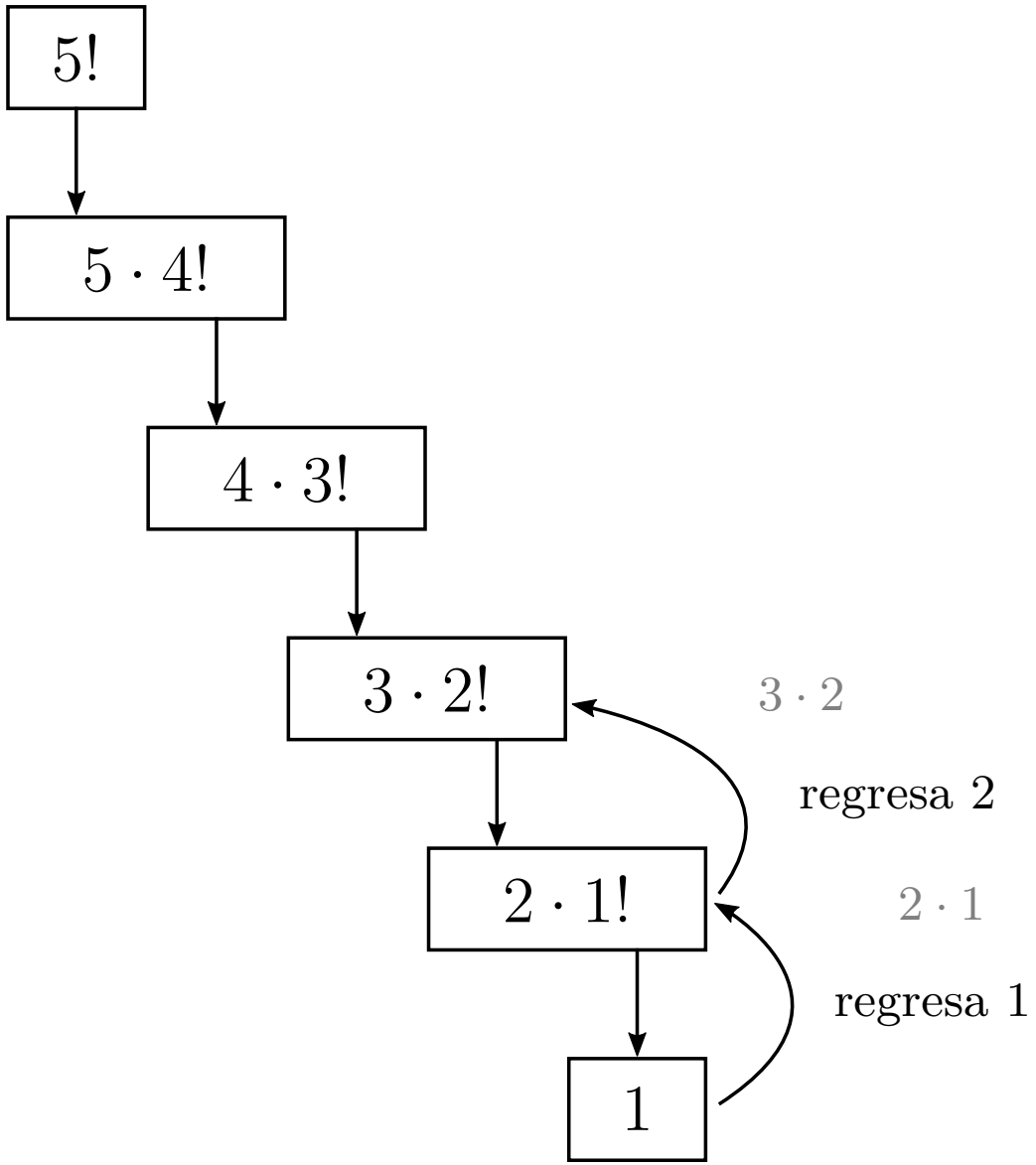
Recursión

Ejemplo 1: cálculo de factorial – recursivo



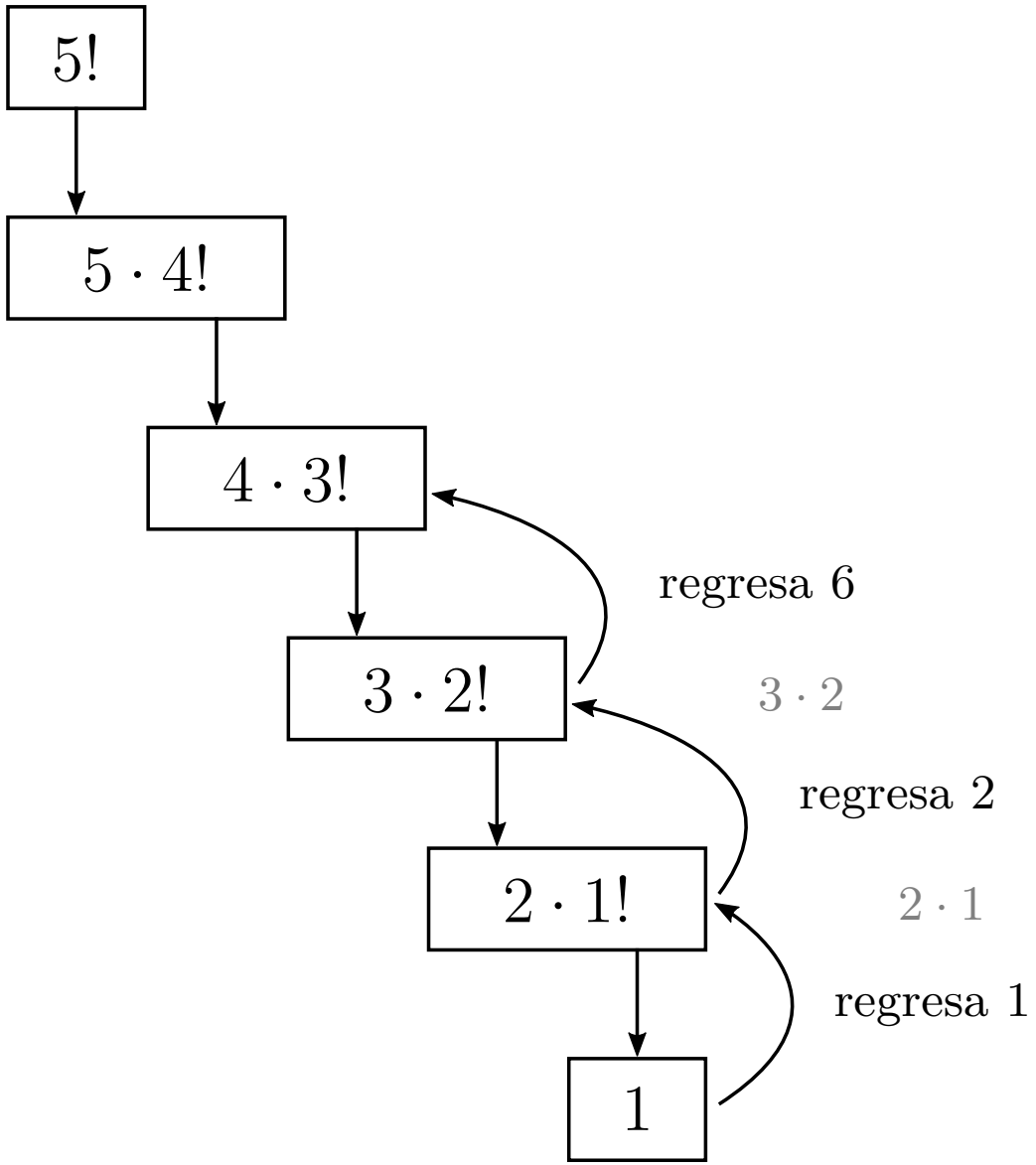
Recursión

Ejemplo 1: cálculo de factorial – recursivo



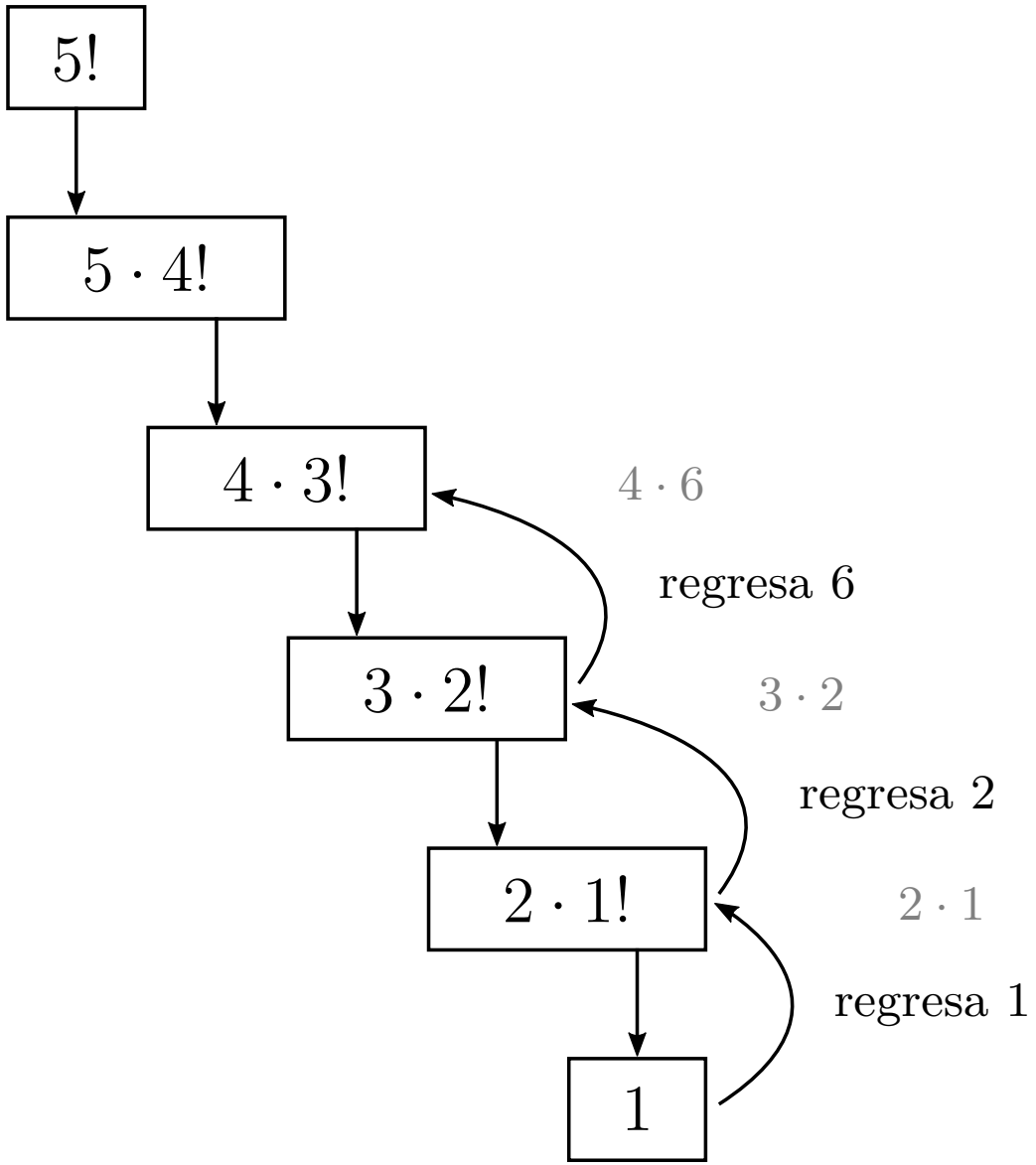
Recursión

Ejemplo 1: cálculo de factorial – recursivo



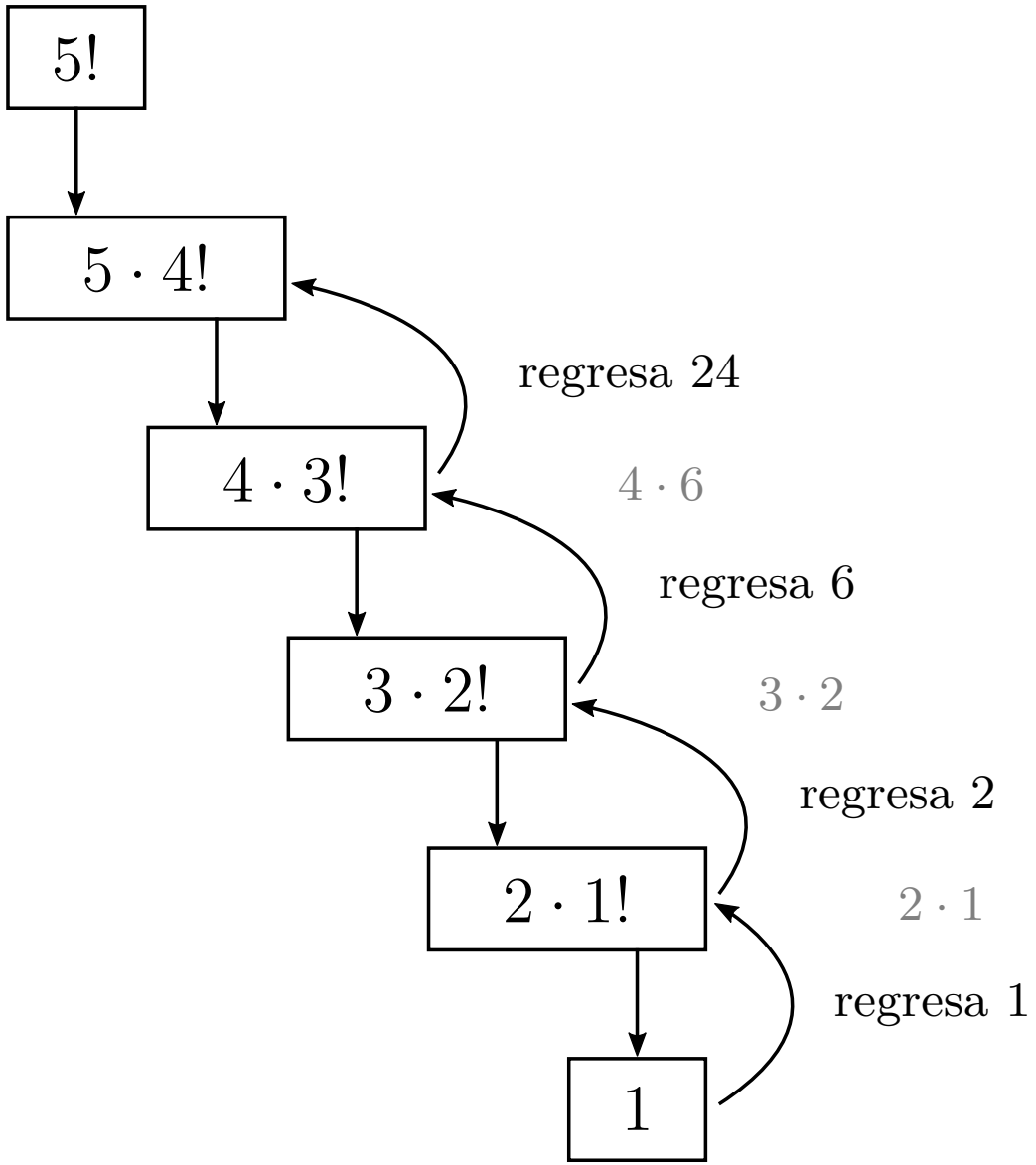
Recursión

Ejemplo 1: cálculo de factorial – recursivo



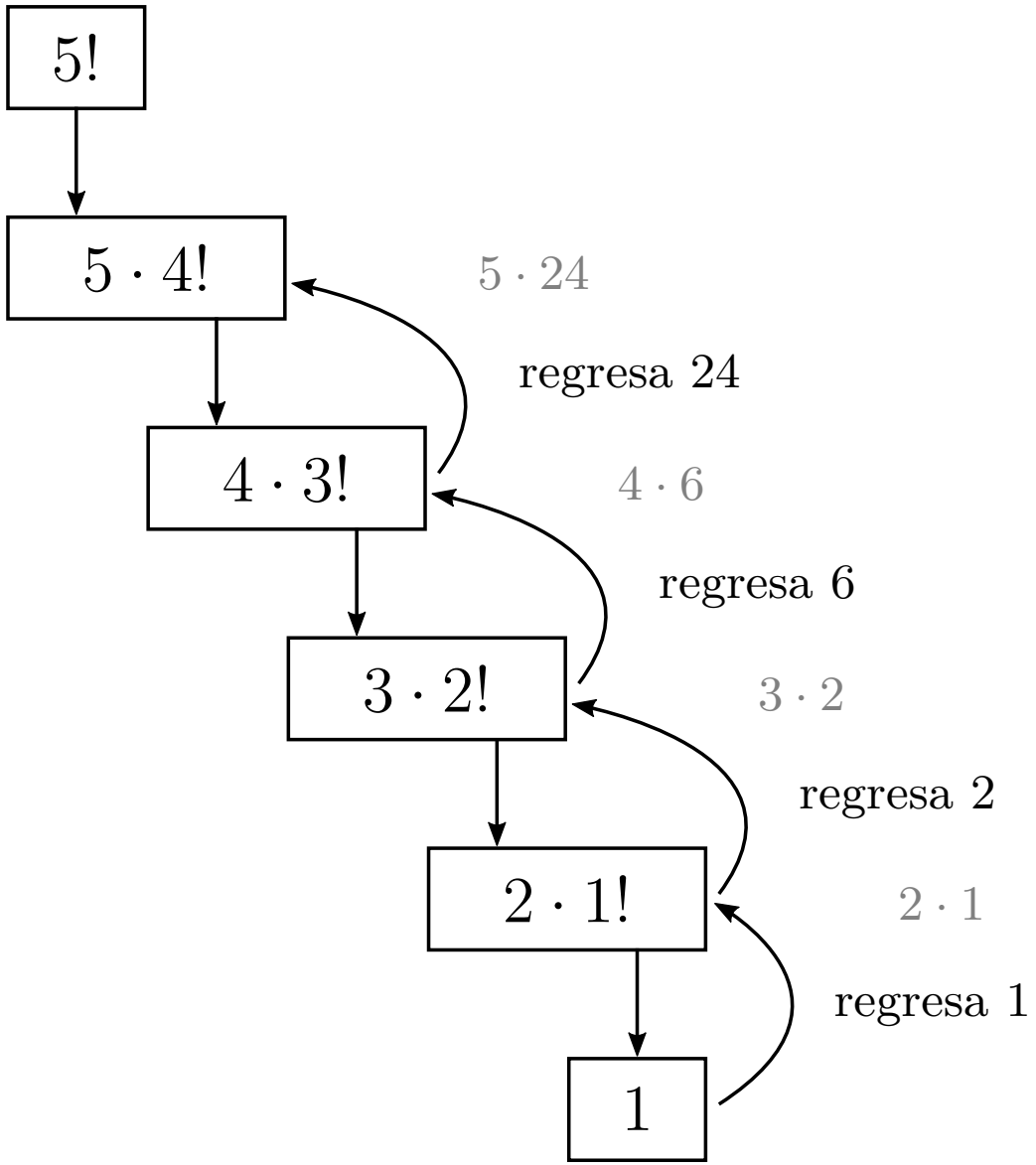
Recursión

Ejemplo 1: cálculo de factorial – recursivo



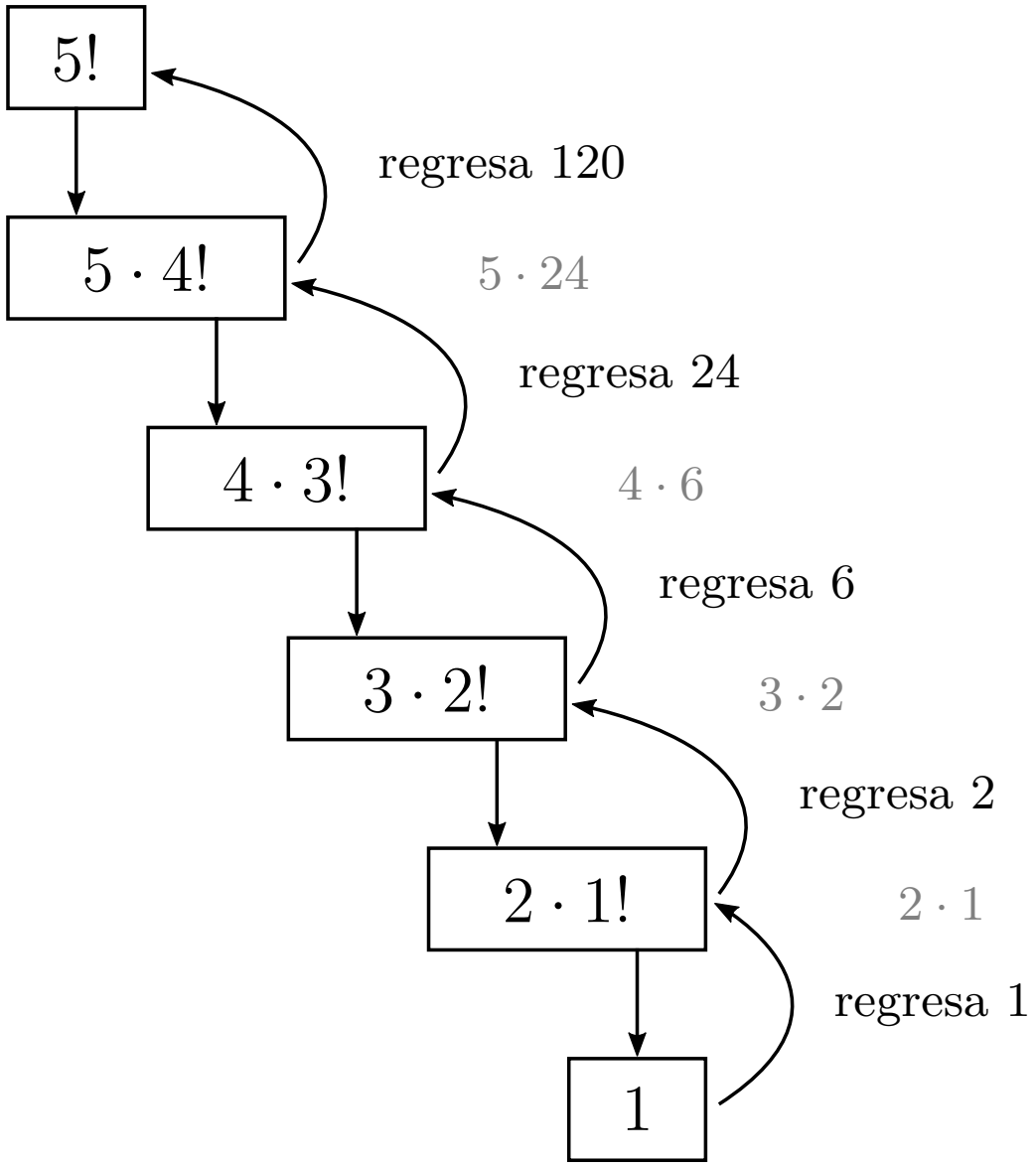
Recursión

Ejemplo 1: cálculo de factorial – recursivo



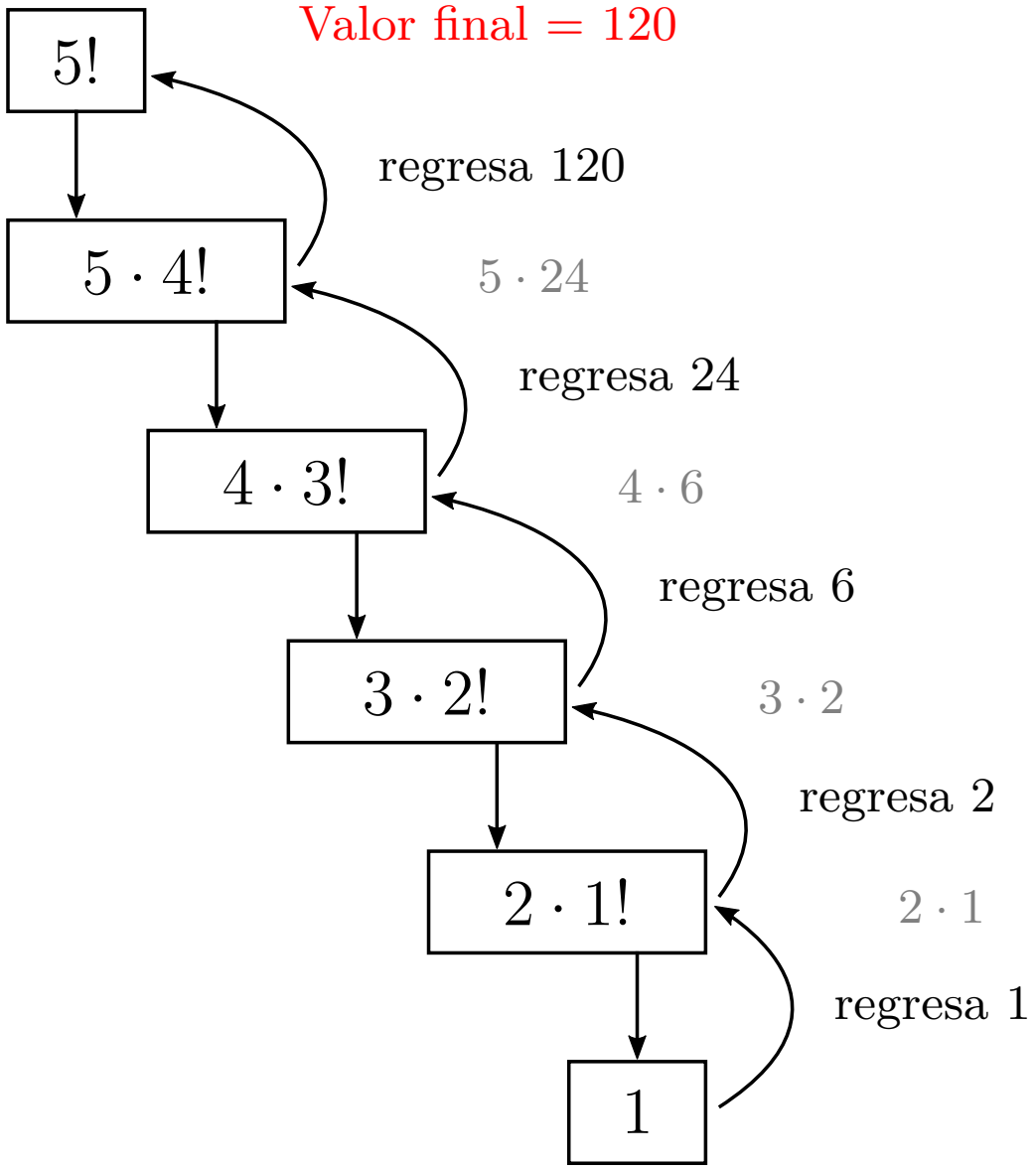
Recursión

Ejemplo 1: cálculo de factorial – recursivo



Recursión

Ejemplo 1: cálculo de factorial – recursivo



Recursión

Ejemplo 1: cálculo de factorial – recursivo

```c
/* Recursive factorial function (D&D, Fig.5.14) */
#include <stdio.h>

long factorial(long);

int main(void)
{
  int i;

  for(i = 1; i <= 10; i++)
    printf("%2d! = %ld\n", i, factorial(i));

  return 0;
}

/* Recursive definition of function factorial */
long factorial(long number)
{
  if(number <= 1)
    return 1;
  else
    return (number * factorial(number-1));
}
```

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes
- Sintetizan en poco espacio lo que sería complicado en la forma iterativa

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes
- Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- Forma natural a los problemas inherentemente recursivos

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes
- Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- Forma natural a los problemas inherentemente recursivos

Desventajas

- Consumen más memoria, pudiendo agotarse

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes
- Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- Forma natural a los problemas inherentemente recursivos

Desventajas

- Consumen más memoria, pudiendo agotarse
- Más lentas que las versiones iterativas debido a la cantidad de llamadas a funciones

Recursión

Ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos
- Las funciones recursivas son más elegantes
- Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- Forma natural a los problemas inherentemente recursivos

Desventajas

- Consumen más memoria, pudiendo agotarse
- Más lentas que las versiones iterativas debido a la cantidad de llamadas a funciones
- Más difíciles de comprender

Recursión

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$$

empieza con 0 y 1, y tiene la propiedad de que cada nro es la suma de los dos nros previos.

Recursión

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$$

empieza con 0 y 1, y tiene la propiedad de que cada nro es la suma de los dos nros previos.

La relación de números sucesivos de Fibonacci converge al valor constante 1,618, conocido como *relación áurea*.

Recursión

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \cdots$$

empieza con 0 y 1, y tiene la propiedad de que cada nro es la suma de los dos nros previos.

La relación de números sucesivos de Fibonacci converge al valor constante 1,618, conocido como *relación áurea*.

Definición recursiva

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = fibonacci(n-1) + fibonacci(n-2)

Recursión

Ejemplo 2: la serie Fibonacci

```c
/* Recursive fibonacci function (D\&D, Fig. 5-15) */
#include <stdio.h>

long fibonacci(long);

int main(void)
{
  long result, number;

  printf("Enter an integer: ");
  scanf("%ld", &number);
  result = fibonacci(number);
  printf("Fibonacci(%ld) = %ld\n", number, result);
  return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci(long n)
{
  if(n == 0 || n == 1)
    return n;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursión

Comparación con iteración

- Basados en una estructura de control

  iteración: estructura de repetición

  recursión: estructura de selección

Recursión
Comparación con iteración

- Basados en una estructura de control

  iteración: estructura de repetición
  recursión: estructura de selección

- Implican repetición

  iteración: utiliza la estructura de repetición de forma explícita
  recursión: repetición con llamadas de función repetidas

Recursión
Comparación con iteración

- Basados en una estructura de control
  - iteración: estructura de repetición
  - recursión: estructura de selección
- Implican repetición
  - iteración: utiliza la estructura de repetición de forma explícita
  - recursión: repetición con llamadas de función repetidas
- Prueba de terminación
  - iterativa: falla la condición de continuación del ciclo
  - recursión: se reconoce un caso base

Recursión
Comparación con iteración

- Basados en una estructura de control
  - iteración: estructura de repetición
  - recursión: estructura de selección
- Implican repetición
  - iteración: utiliza la estructura de repetición de forma explícita
  - recursión: repetición con llamadas de función repetidas
- Prueba de terminación
  - iterativa: falla la condición de continuación del ciclo
  - recursión: se reconoce un caso base
- Pueden ocurrir de forma indefinida (ciclo infinito)
  - iteración: si la condición de continuación del ciclo nunca es falsa
  - recursión: si la recursión no reduce el problema en cada ocasión

Recursión
Complejidad computacional

- En cada llamada recursiva de `fibonacci` se llama dos veces a la misma función
- La cantidad de llamadas recursivas es por lo tanto 2^n
- Este número crece muy rápidamente; ejemplos: $2^{20} \approx$ millón, $2^{30} \approx$ mil millones
- En ciencia de la computación a esto se le llama *complejidad exponencial*