Informática II
Recursión

Gonzalo F. Perez Paina

Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

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.

Ejemplo 1: cálculo de factorial

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

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 `num` mayor a cero de *forma iterativa*

```
factorial = 1;
for(cont = num; cont >= 1; cont--)
  factorial *= cont;
```

Ejemplo 1: cálculo de factorial

Definición recursiva

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

Ejemplo 1: cálculo de factorial

Definición recursiva

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

Ejemplo: 5!

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

Ejemplo 1: cálculo de factorial

Definición recursiva

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

Ejemplo: 5!

$$\begin{aligned} 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\ 5! &= 5 \cdot (4!) \end{aligned}$$

El problema se dividie en: 1) una parte que se sabe resolver (caso base) y 2) una parte similar al problema original

Ejemplo 1: cálculo de factorial

Definición recursiva

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

Ejemplo: 5!

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

El problema se dividie en: 1) una parte que se sabe resolver (caso base) y 2) una parte similar al problema original

☞ Caso base: $0! = 1! = 1$

Ejemplo 1: cálculo de factorial

$\boxed{5!}$

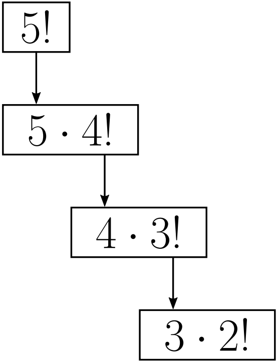
Ejemplo 1: cálculo de factorial

$$\boxed{5!}$$
$$\downarrow$$
$$\boxed{5 \cdot 4!}$$

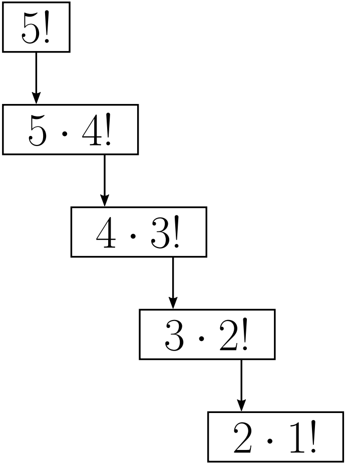
Ejemplo 1: cálculo de factorial

$$\boxed{5!}$$
$$\downarrow$$
$$\boxed{5 \cdot 4!}$$
$$\downarrow$$
$$\boxed{4 \cdot 3!}$$

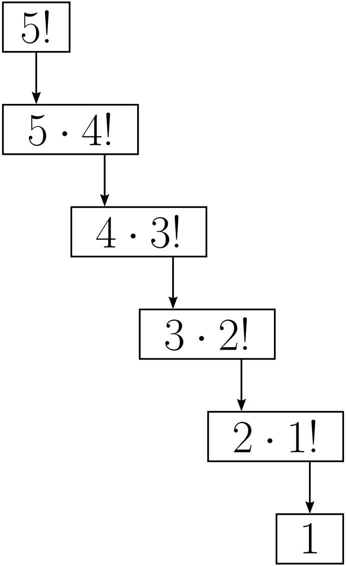
Ejemplo 1: cálculo de factorial



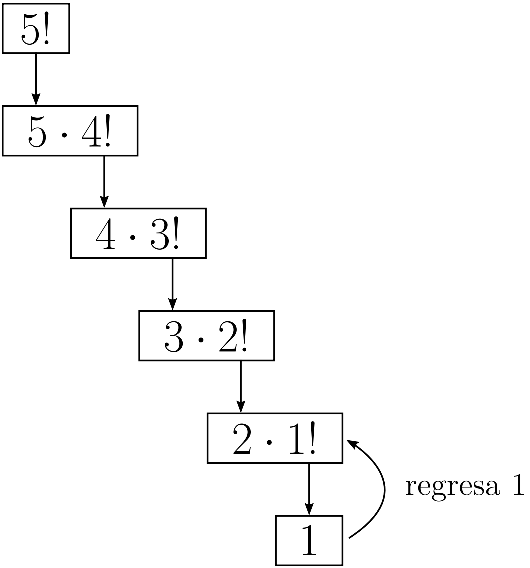
Ejemplo 1: cálculo de factorial



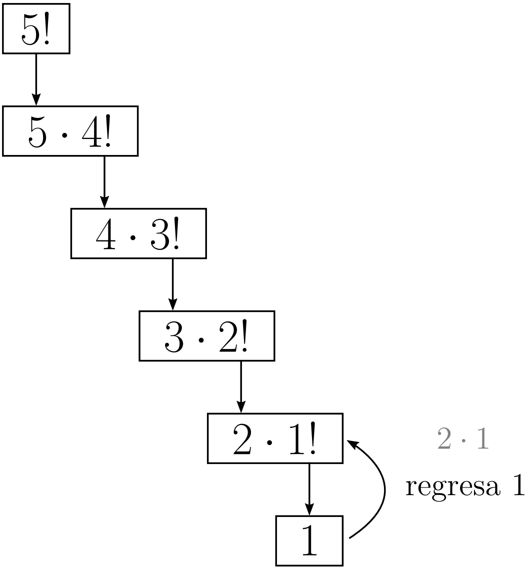
Ejemplo 1: cálculo de factorial



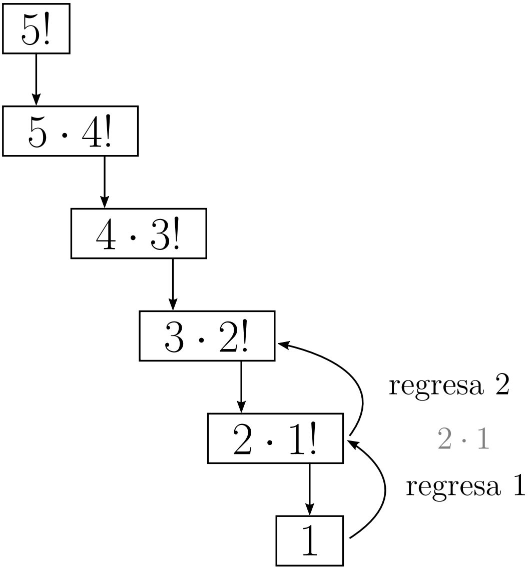
Ejemplo 1: cálculo de factorial



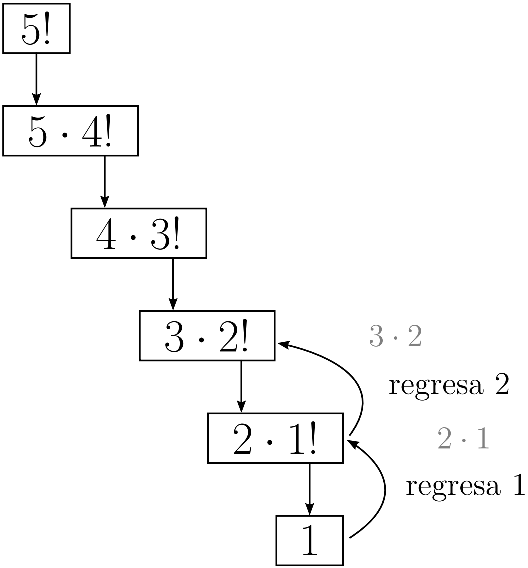
Ejemplo 1: cálculo de factorial



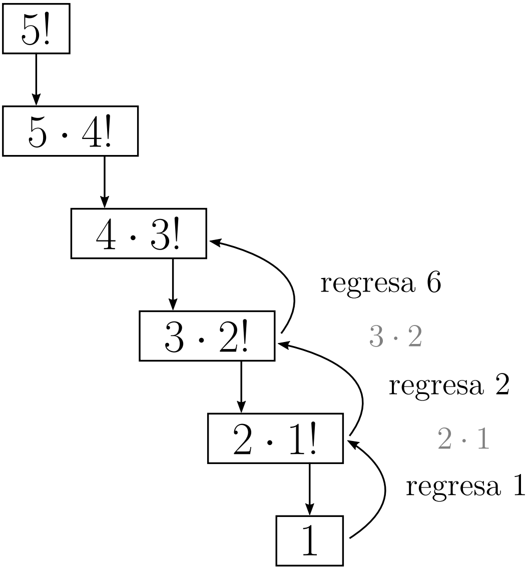
Ejemplo 1: cálculo de factorial



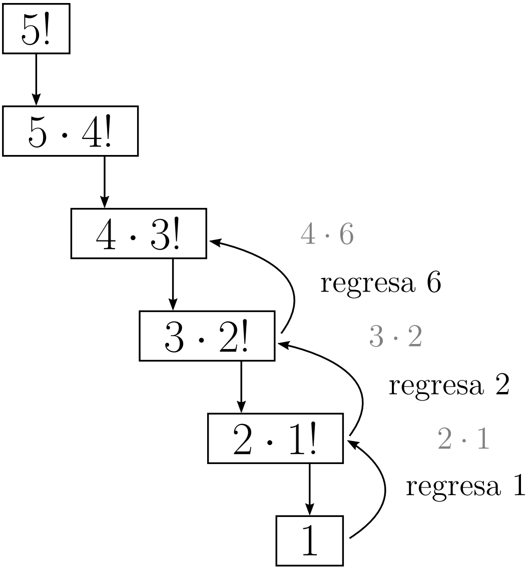
Ejemplo 1: cálculo de factorial



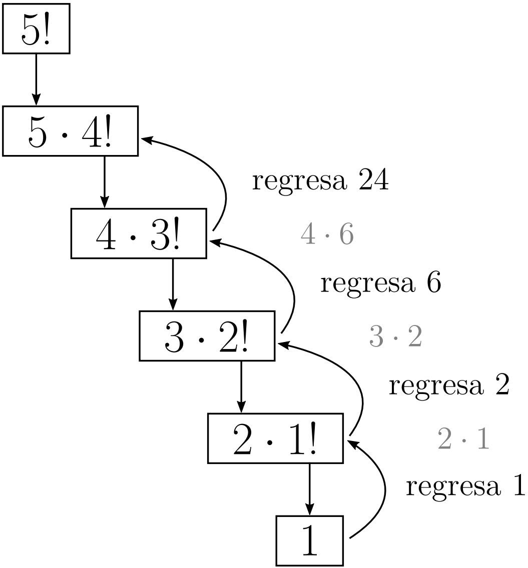
Ejemplo 1: cálculo de factorial



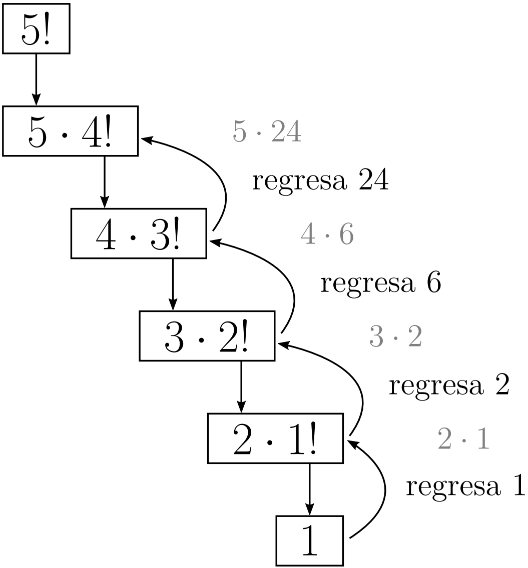
Ejemplo 1: cálculo de factorial



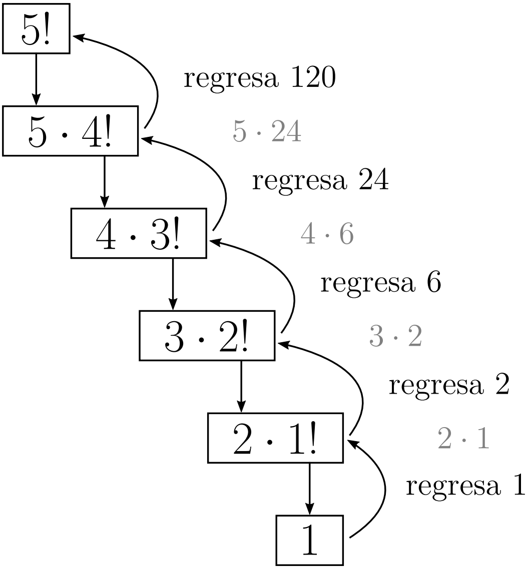
Ejemplo 1: cálculo de factorial



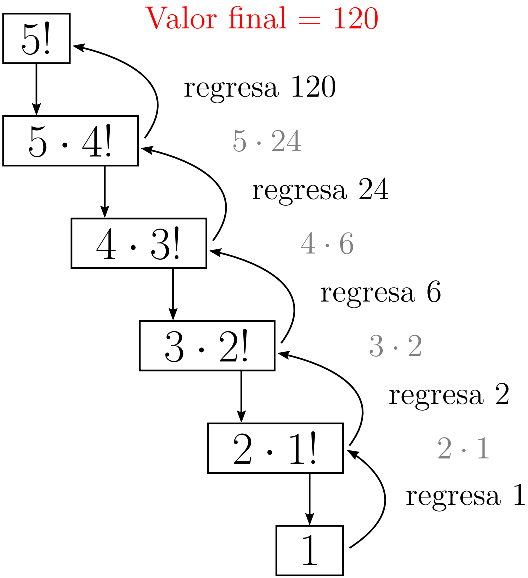
Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial

```c
#include <stdio.h>

long factorial(long ); /* Prototipo de función */

int main(void)
{
  int i;

  for(i = 0; i <= 10; i++)
    printf("%2d! = %ld\n", i, factorial(i));

  return 0;
}

/* Función recursiva 'factorial' */
long factorial(long numero)
{




}
```

Ejemplo 1: cálculo de factorial

```c
#include <stdio.h>

long factorial(long ); /* Prototipo de función */

int main(void)
{
  int i;

  for(i = 0; i <= 10; i++)
    printf("%2d! = %ld\n", i, factorial(i));

  return 0;
}

/* Función recursiva 'factorial' */
long factorial(long numero)
{
  if(numero <= 1) /* Caso base */
    return 1;


}
```

Ejemplo 1: cálculo de factorial

```c
#include <stdio.h>

long factorial(long ); /* Prototipo de función */

int main(void)
{
  int i;

  for(i = 0; i <= 10; i++)
    printf("%2d! = %ld\n", i, factorial(i));

  return 0;
}

/* Función recursiva 'factorial' */
long factorial(long numero)
{
  if(numero <= 1) /* Caso base */
    return 1;
  else /* Recursión */
    return (numero * factorial(numero - 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

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 número es la suma de los dos números previos.

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 número es la suma de los dos números previos.

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

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 número es la suma de los dos números 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)

Ejemplo 2: la serie Fibonacci

```c
#include <stdio.h>
long fibonacci(long ); /* Prototipo de función */

int main(void)
{
  long resultado, numero;

  printf("Ingrese un entero: ");
  scanf("%ld", &numero);

  resultado = fibonacci(numero);
  printf("fibonacci(%ld) = %ld\n", numero, resultado);
  return 0;
}

/* Función recursiva 'fibonacci' */
long fibonacci(long n)
{




}
```

Ejemplo 2: la serie Fibonacci

```c
#include <stdio.h>
long fibonacci(long ); /* Prototipo de función */

int main(void)
{
  long resultado, numero;

  printf("Ingrese un entero: ");
  scanf("%ld", &numero);

  resultado = fibonacci(numero);
  printf("fibonacci(%ld) = %ld\n", numero, resultado);
  return 0;
}

/* Función recursiva 'fibonacci' */
long fibonacci(long n)
{
  if(n == 0 || n == 1) /* Caso base */
    return n;
  else /* Recursión */
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Comparación entre recursión e iteración

- Basados en una estructura de control

  iteración: estructura de repetición
  recursión: estructura de selección

Comparación entre recursión e 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

Comparación entre recursión e 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

Comparación entre recursión e 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

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*

Actividad práctica

1. Escribir un programa que defina una función `factorial()` donde el cálculo se realiza de forma iterativa. Evaluar dicha función de forma similar al programa ejemplo de la función factorial recursiva.

2. Escribir dos programas que imprima la serie de Fibonacci con la siguiente interacción con el usuario:

```
> ./a.out
--- Programa de serie de Fibonacci ---
Ingrese un entero: 12
Serie: 0 1 1 2 3 5 8 13 21 34 55 89 144
```

   Uno de ellos debe implementar la función `fibonacci(n)` de forma iterativa y el otro de forma recursiva.

Actividad práctica

3. Modificar el programa ejemplo de la función recursiva del cálculo del factorial para visualizar la recursión. El programa debe tener la siguiente interacción con el usuario

```
> ./a.out
Ingrese un entero: 4

4! = 4 * 3!
   3! = 3 * 2!
      2! = 2 * 1!

--> Resultado: 24

> ./a.out
Ingrese un entero: 5

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

--> Resultado: 120
```