

5

## Introducción a la Recursión

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Luis Hernández Yáñez  
Facultad de Informática  
Universidad Complutense



# Índice

---

Concepto de recursión	983
Algoritmos recursivos	986
Funciones recursivas	987
Diseño de funciones recursivas	989
Modelo de ejecución	990
La pila del sistema	992
La pila y las llamadas a función	994
Ejecución de la función factorial()	1005
Tipos de recursión	1018
Recursión simple	1019
Recursión múltiple	1020
Recursión anidada	1022
Recursión cruzada	1026
Código del subprograma recursivo	1027
Parámetros y recursión	1032
Ejemplos de algoritmos recursivos	1034
Búsqueda binaria	1035
Torres de Hanoi	1038
Recursión frente a iteración	1043
Estructuras de datos recursivas	1045



## Recursión

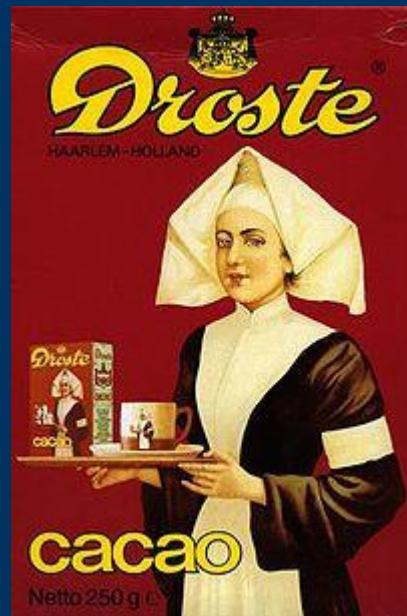


# Concepto de recursión

## *Recursión (recursividad, recurrencia)*

*Definición recursiva: En la definición aparece lo que se define*

$$\text{Factorial}(N) = N \times \text{Factorial}(N-1) \quad (N \geq 0)$$



(wikipedia.org)

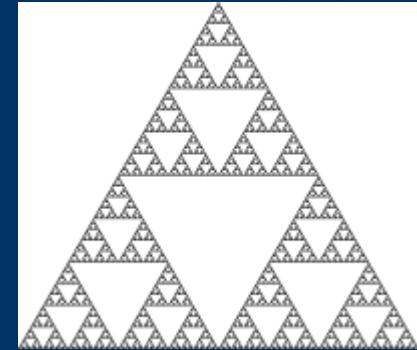
La imagen del paquete  
aparece dentro del propio  
paquete,... ¡hasta el infinito!



La cámara graba lo que graba

([http://farm1.static.flickr.com/83/229219543\\_edf740535b.jpg](http://farm1.static.flickr.com/83/229219543_edf740535b.jpg))

Cada triángulo está  
formado por otros  
triángulos más pequeños



(wikipedia.org)



Las *matrioskas* rusas



# Definiciones recursivas

---

$$\text{Factorial}(N) = N \times \text{Factorial}(N-1)$$

El factorial se define en función de sí mismo

Los programas no pueden manejar la recursión infinita

La definición recursiva debe adjuntar uno o más casos base

*Caso base:* aquel en el que no se utiliza la definición recursiva

Proporcionan puntos finales de cálculo:

$$\text{Factorial}(N) = \begin{cases} N \times \text{Factorial}(N-1) & \text{si } N > 0 \\ 1 & \text{si } N = 0 \end{cases}$$

*Caso recursivo (inducción)*  
*Caso base (o de parada)*

El valor de N se va aproximando al valor del caso base (0)



## Algoritmos recursivos



# Algoritmos recursivos

---

## Funciones recursivas

Una función puede implementar un algoritmo recursivo

La función se llamará a sí misma si no se ha llegado al caso base

$$\text{Factorial}(N) = \begin{cases} 1 & \text{si } N = 0 \\ N \times \text{Factorial}(N-1) & \text{si } N > 0 \end{cases}$$

```
long long int factorial(int n) {
    long long int resultado;
    if (n == 0) { // Caso base
        resultado = 1;
    }
    else {
        resultado = n * factorial(n - 1);
    }
    return resultado;
}
```



## Funciones recursivas

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) { // Caso base  
        resultado = 1;  
    }  
    else {  
        resultado = n * factorial(n - 1);  
    }  
    return resultado;  
}
```

$\text{factorial}(5) \rightarrow 5 \times \text{factorial}(4) \rightarrow 5 \times 4 \times \text{factorial}(3)$   
 $\rightarrow 5 \times 4 \times 3 \times \text{factorial}(2) \rightarrow 5 \times 4 \times 3 \times 2 \times \text{factorial}(1)$   
 $\rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times \underline{\text{factorial}(0)} \rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times 1$   
 $\rightarrow 120$  Caso base

```
1  
1  
2  
6  
24  
120  
720  
5040  
40320  
362880  
3628800  
39916800  
479001600  
6227020800  
87178291200  
1307674368000  
20922789888000  
355687428096000  
6402373705728000  
121645100408832000
```



# Algoritmos recursivos

---

## Diseño de funciones recursivas

Una función recursiva debe satisfacer tres condiciones:

- ✓ Caso(s) base: Debe haber al menos un caso base de parada
- ✓ Inducción: Paso recursivo que provoca una llamada recursiva  
Debe ser correcto para distintos parámetros de entrada
- ✓ Convergencia: Cada paso recursivo debe acercar a un caso base

Se describe el problema en términos de problemas *más sencillos*

$$\text{Factorial}(N) = \begin{cases} 1 & \text{si } N = 0 \\ N \times \text{Factorial}(N-1) & \text{si } N > 0 \end{cases}$$

Función `factorial()`: tiene caso base ( $N = 0$ ), siendo correcta para  $N$  es correcta para  $N+1$  (*inducción*) y se acerca cada vez más al caso base ( $N-1$  está más cerca de 0 que  $N$ )



## Modelo de ejecución



# Modelo de ejecución

---

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) { // Caso base  
        resultado = 1;  
    }  
    else {  
        resultado = n * factorial(n - 1);  
    }  
    return resultado;  
}
```

Cada llamada recursiva fuerza una nueva ejecución de la función

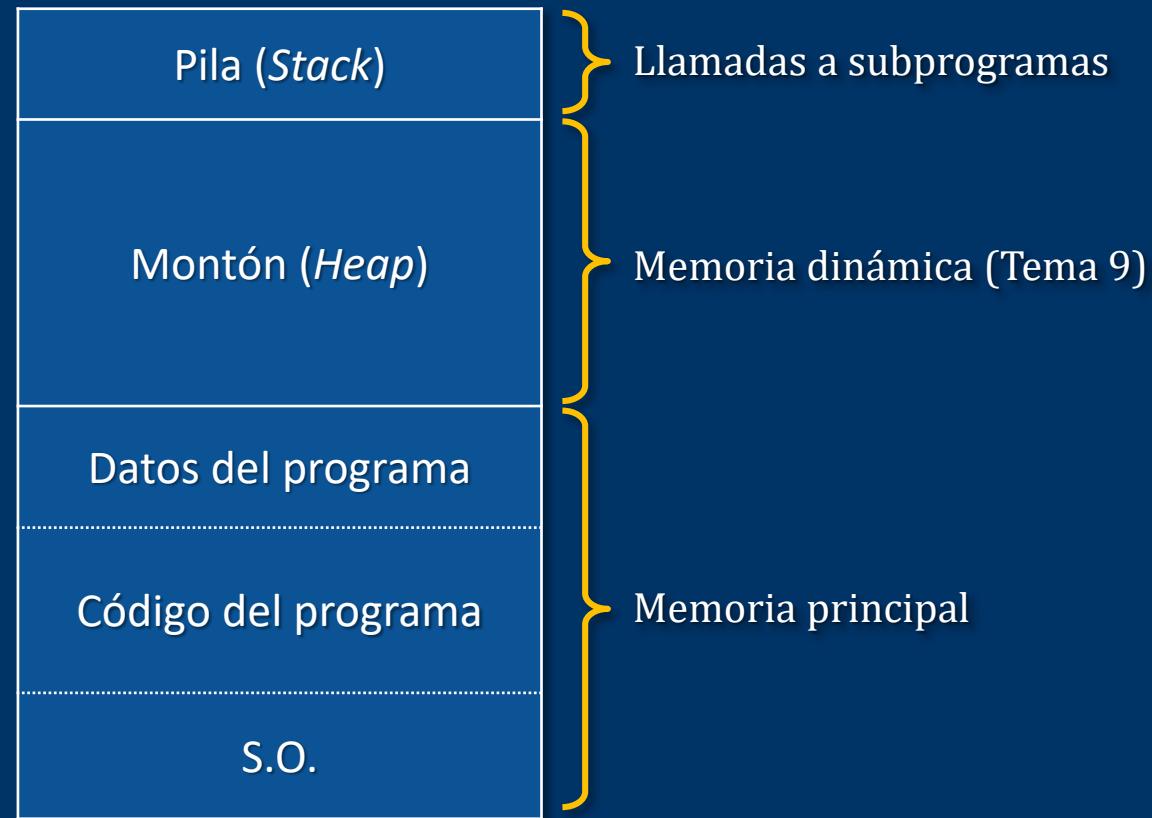
Cada llamada utiliza sus propios parámetros por valor  
y variables locales (n y resultado en este caso)

En las llamadas a la función se utiliza la pila del sistema para  
mantener los datos locales y la dirección de vuelta



# La pila del sistema (*stack*)

Regiones de memoria que distingue el sistema operativo:

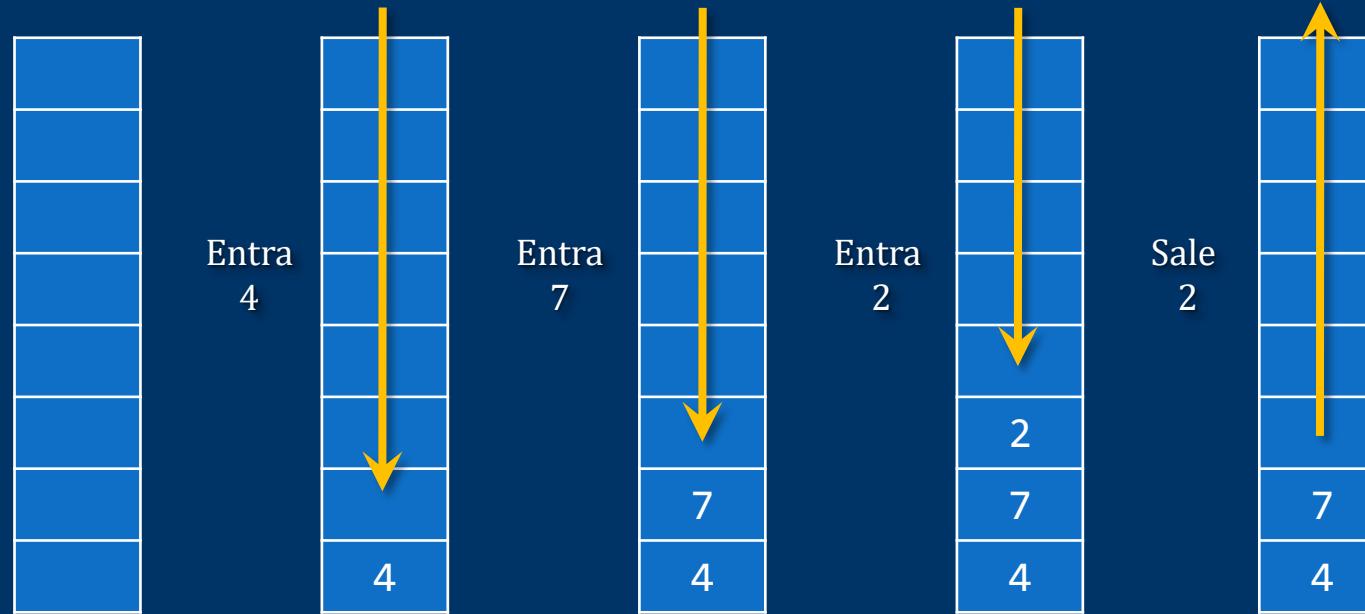


# La pila del sistema (*stack*)

Mantiene los datos locales de la función y la dirección de vuelta

Estructura de tipo *pila*: lista LIFO (*last-in first-out*)

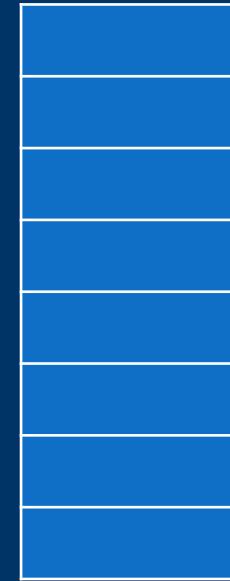
El último que entra es el primero que sale:



# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>   b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>   cout << funcA(4); ←
    ...
}
```



Pila

Llamada a función:  
Entra la dirección de vuelta



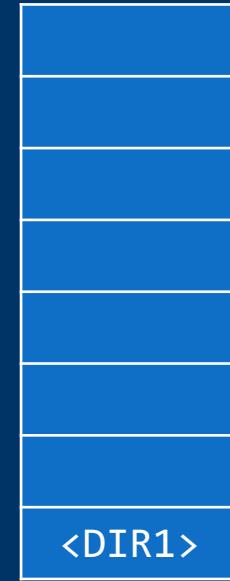
# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    cout << funcA(4);
    ...
}
```



Entrada en la función:  
Se alojan los datos locales



# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    cout << funcA(4);
    ...
}
```

<DIR2>      b = funcB(a);      ← Llamada a función:  
Entra la dirección de vuelta

<DIR1>



Pila



# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>    b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>    cout << funcA(4);
    ...
}
```

Entrada en la función:  
Se alojan los datos locales



Pila



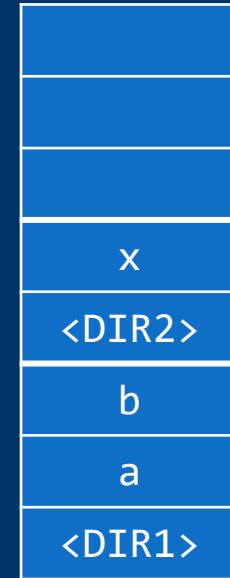
# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2> b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1> cout << funcA(4);
    ...
}
```



Vuelta de la función:  
Se eliminan los datos locales



Pila



# La pila y las llamadas a función

## Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>    b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>    cout << funcA(4);
    ...
}
```

Vuelta de la función:  
Sale la dirección de vuelta



Pila



# La pila y las llamadas a función

Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    b = funcB(a);           ← La ejecución continúa
    ...
    return b;
}
int main() {
    ...
    cout << funcA(4);
    ...
}
```



Pila



# La pila y las llamadas a función

Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>   b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>   cout << funcA(4);
    ...
}
```



Vuelta de la función:  
Se eliminan los datos locales



# La pila y las llamadas a función

Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>   b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>   cout << funcA(4);
    ...
}
```



Vuelta de la función:  
Sale la dirección de vuelta



Pila



# La pila y las llamadas a función

Datos locales y direcciones de vuelta

```
...
int funcB(int x) {
    ...
    return x;
}
int funcA(int a) {
    int b;
    ...
    <DIR2>   b = funcB(a);
    ...
    return b;
}
int main() {
    ...
    <DIR1>   cout << funcA(4); ←
    ...
}
```



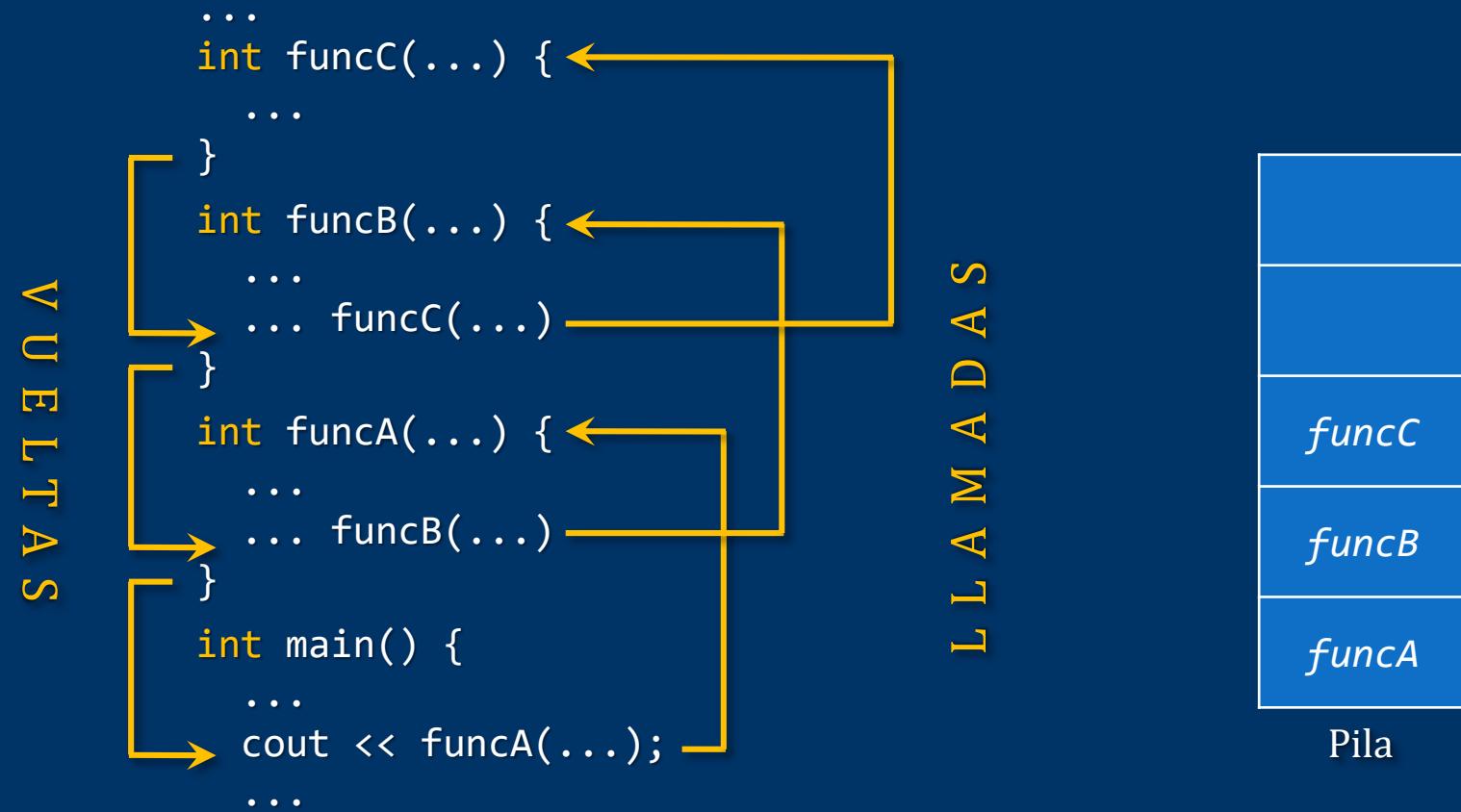
Pila

La ejecución continúa  
en esa dirección



# La pila y las llamadas a función

Mecanismo de pila adecuado para llamadas a funciones anidadas:  
Las llamadas terminan en el orden contrario a como se llaman



# Ejecución de la función factorial()

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) { // Caso base  
        resultado = 1;  
    }  
    else {  
        resultado = n * factorial(n - 1);  
    }  
    return resultado;  
}
```

```
cout << factorial(5) << endl;
```



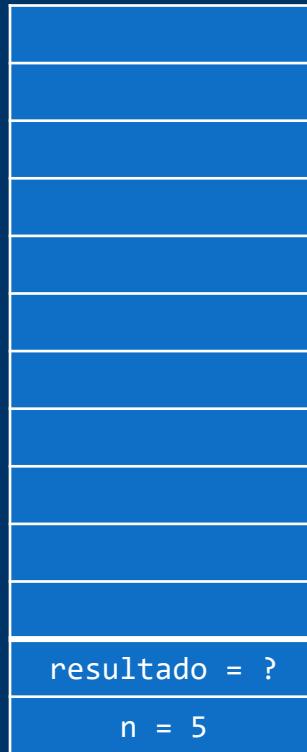
Obviaremos las direcciones de vuelta en la pila



# Ejecución de la función factorial()

---

factorial(5)

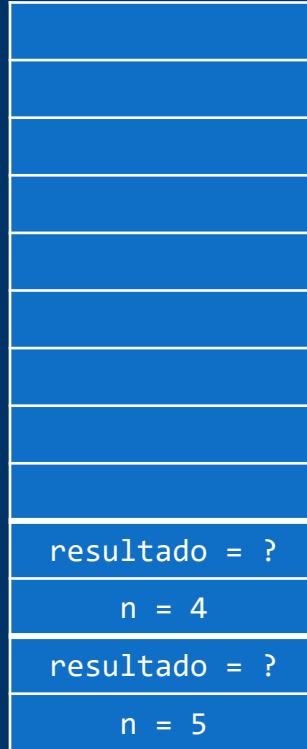


Pila



# Ejecución de la función factorial()

factorial(5)  
↳ factorial(4)

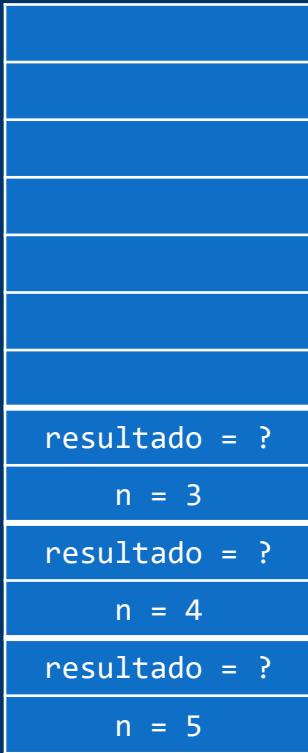


Pila



# Ejecución de la función factorial()

```
factorial(5)
  ↘ factorial(4)
    ↘ factorial(3)
```



Pila



# Ejecución de la función factorial()

```
factorial(5)
  ↘ factorial(4)
    ↘ factorial(3)
      ↘ factorial(2)
```

resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

```
factorial(5)
  ↘ factorial(4)
    ↘ factorial(3)
      ↘ factorial(2)
        ↘ factorial(1)
```

resultado = ?
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

factorial(5)

  ↳ factorial(4)

    ↳ factorial(3)

      ↳ factorial(2)

      ↳ factorial(1)

        ↳ factorial(0)

resultado = 1
n = 0
resultado = ?
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

factorial(5)

  ↳ factorial(4)

    ↳ factorial(3)

      ↳ factorial(2)

      ↳ factorial(1)

      ↳ factorial(0)



resultado = 1
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

factorial(5)

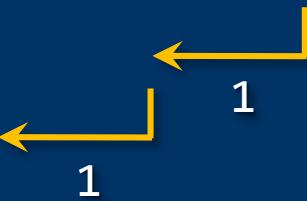
  ↳ factorial(4)

    ↳ factorial(3)

      ↳ factorial(2)

      ↳ factorial(1)

        ↳ factorial(0)



resultado = 2
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

factorial(5)

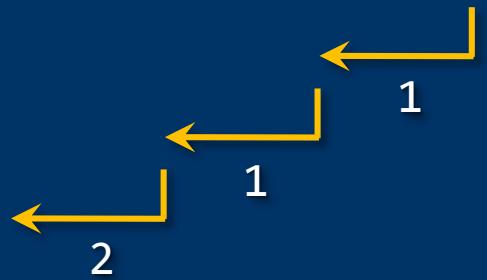
  ↳ factorial(4)

    ↳ factorial(3)

      ↳ factorial(2)

      ↳ factorial(1)

        ↳ factorial(0)



resultado = 6
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

factorial(5)

  ↳ factorial(4)

    ↳ factorial(3)

      ↳ factorial(2)

      ↳ factorial(1)

        ↳ factorial(0)

        1

        1

        2

        6

resultado = 24
n = 4
resultado = ?
n = 5

Pila



# Ejecución de la función factorial()

**factorial(5)**

↳ factorial(4)

↳ factorial(3)

↳ factorial(2)

↳ factorial(1)

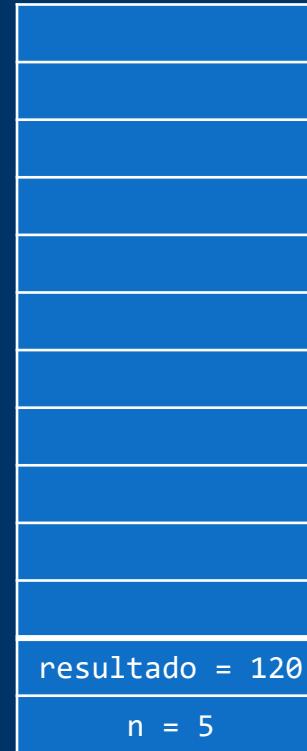
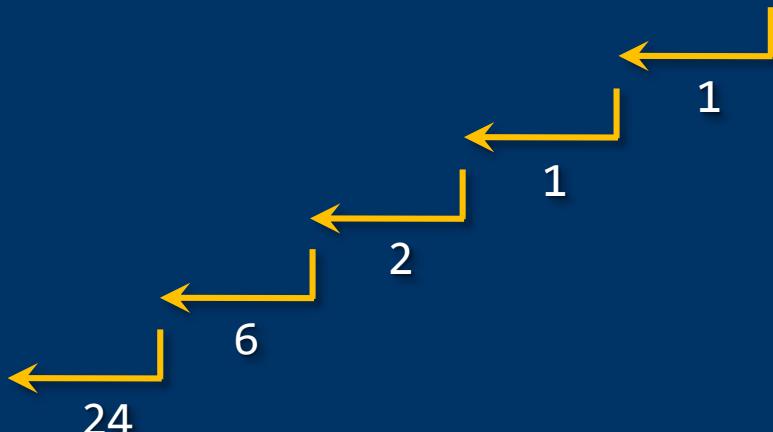
→ factorial(0)

1

1

2

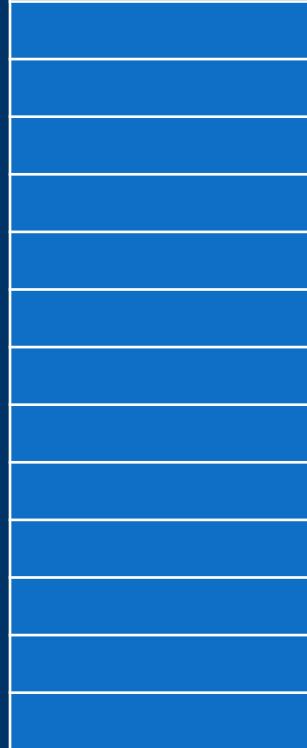
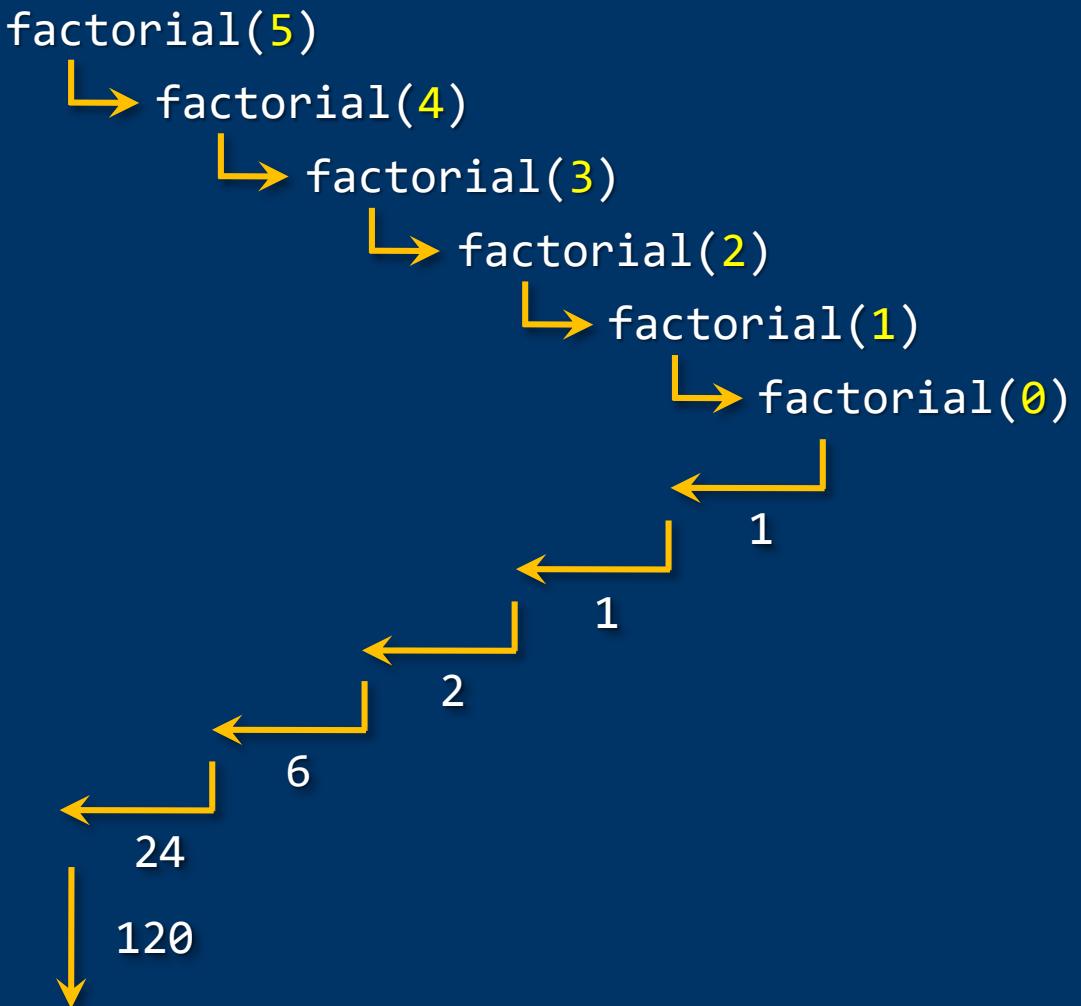
6



Pila



# Ejecución de la función factorial()



Pila



## Tipos de recursión



# Recursión simple

---

Sólo hay una llamada recursiva

Ejemplo: Cálculo del factorial de un número entero positivo

```
long long int factorial(int n) {  
    long long int resultado;  
    if (n == 0) { // Caso base  
        resultado = 1;  
    }  
    else {  
        resultado = n * factorial(n - 1);  
    }  
    return resultado;  
}
```

Una sola llamada recursiva



# Recursión múltiple

---

Varias llamadas recursivas

Ejemplo: Cálculo de los números de *Fibonacci*

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

↑      ↑  
Dos llamadas recursivas



# Recursión múltiple

fibonacci.cpp

```
...
int main() {
    for (int i = 0; i < 20; i++) {
        cout << fibonacci(i) << endl;
    }
    return 0;
}

int fibonacci(int n) {
    int resultado;
    if (n == 0) {
        resultado = 0;
    } else if (n == 1) {
        resultado = 1;
    } else {
        resultado = fibonacci(n - 1) + fibonacci(n - 2);
    }
    return resultado;
}
```

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

```
0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
55  
89  
144  
233  
377  
610  
987  
1597  
2584  
4181
```



# Recursión anidada

---

En una llamada recursiva alguno de los argumentos es otra llamada  
Ejemplo: Cálculo de los números de Ackermann:

$$\text{Ack}(m, n) \left\{ \begin{array}{ll} n + 1 & \text{si } m = 0 \\ \text{Ack}(m-1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{array} \right.$$



Argumento que es una llamada recursiva



## Números de Ackermann

...

```
int ackermann(int m, int n) {
    int resultado;
    if (m == 0) {
        resultado = n + 1;
    }
    else if (n == 0) {
        resultado = ackermann(m - 1, 1);
    }
    else {
        resultado = ackermann(m - 1, ackermann(m, n - 1));
    }
    return resultado;
}
```

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



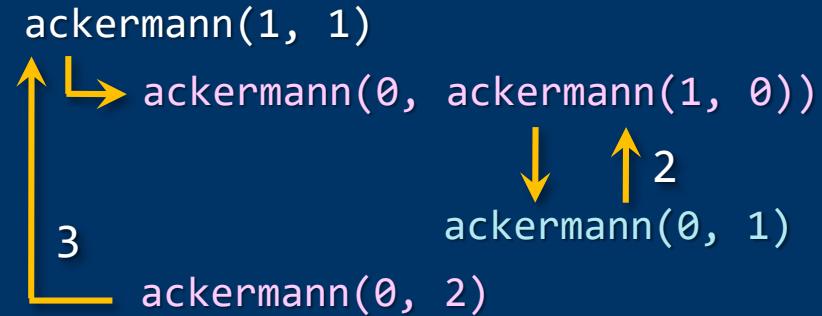
Pruébalo con números muy bajos:  
Se generan MUCHAS llamadas recursivas



# Recursión anidada

## Números de Ackermann

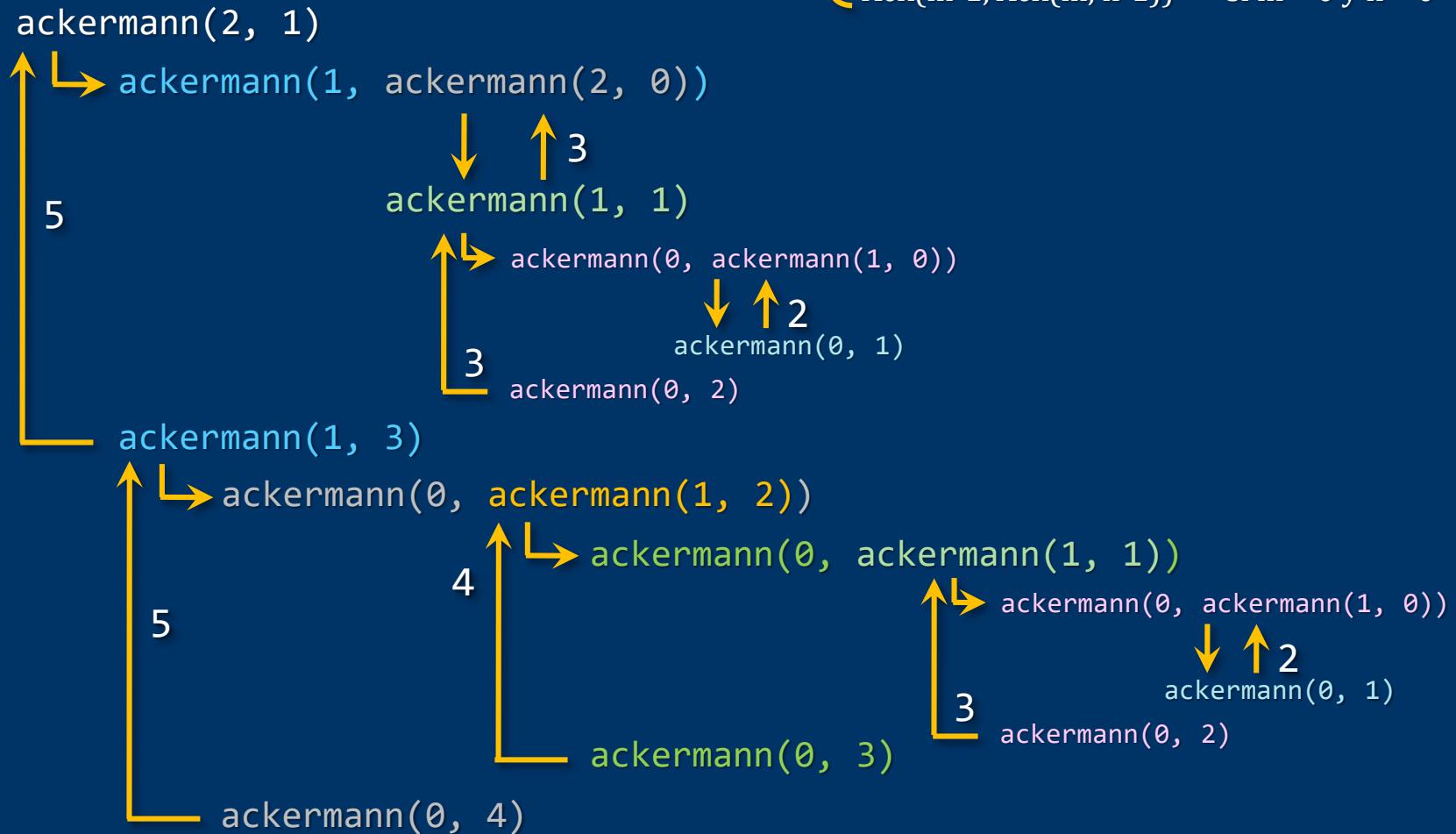
$$\text{Ack}(m, n) \begin{cases} n + 1 & \text{si } m = 0 \\ \text{Ack}(m-1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$



# Recursión anidada

## Números de Ackermann

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



## Código del subprograma recursivo



# Código del subprograma recursivo

---

*Código anterior y posterior a la llamada recursiva*

```
{  
    Código anterior  
    Llamada recursiva  
    Código posterior  
}
```

*Código anterior*

Se ejecuta para las distintas entradas antes que el *código posterior*

*Código posterior*

Se ejecuta para las distintas entradas tras llegarse al caso base

El código anterior se ejecuta en orden directo para las distintas entradas, mientras que el código posterior lo hace en orden inverso

Si no hay código anterior: *recursión por delante*

Si no hay código posterior: *recursión por detrás*



# Código del subprograma recursivo

*Código anterior y posterior a la llamada recursiva*

```
void func(int n) {  
    if (n > 0) { // Caso base: n == 0  
        cout << "Entrando (" << n << ")" << endl; // Código anterior  
        func(n - 1); // Llamada recursiva  
        cout << "Saliendo (" << n << ")" << endl; // Código posterior  
    }  
}
```

→ func(5);

El código anterior se ejecuta para los sucesivos valores de n (5, 4, 3, ...)

El código posterior al revés (1, 2, 3, ...)

Entrando (5)
Entrando (4)
Entrando (3)
Entrando (2)
Entrando (1)
Saliendo (1)
Saliendo (2)
Saliendo (3)
Saliendo (4)
Saliendo (5)



## Recorrido de los elementos de una lista (*directo*)

El código anterior a la llamada procesa la lista en su orden:

```
...
void mostrar(tLista lista, int pos);

int main() {
    tLista lista;
    lista.cont = 0;
    // Carga del array...
    mostrar(lista, 0);

    return 0;
}

void mostrar(tLista lista, int pos) {
    if (pos < lista.cont) {
        cout << lista.elementos[pos] << endl;
        mostrar(lista, pos + 1);
    }
}
```

1  
3  
8  
13  
17  
22  
23  
39  
52  
55



## Recorrido de los elementos de una lista (inverso)

El código posterior procesa la lista en el orden inverso:

```
...
void mostrar(tLista lista, int pos);

int main() {
    tLista lista;
    lista.cont = 0;
    // Carga del array...
    mostrar(lista, 0);

    return 0;
}

void mostrar(tLista lista, int pos) {
    if (pos < lista.cont) {
        mostrar(lista, pos + 1);
        cout << lista.elementos[pos] << endl;
    }
}
```

55  
52  
39  
23  
22  
17  
13  
8  
3  
1



## Parámetros y recursión



# Parámetros y recursión

---

## *Parámetros por valor y por referencia*

Parámetros por valor: cada llamada usa los suyos propios

Parámetros por referencia: misma variable en todas las llamadas

Recogen resultados que transmiten entre las llamadas

```
void factorial(int n, int &fact) {  
    if (n == 0) {  
        fact = 1;  
    }  
    else {  
        factorial(n - 1, fact);  
        fact = n * fact;  
    }  
}
```

Cuando n es 0, el argumento de fact toma el valor 1

Al volver se le va multiplicando por los demás n (distintos)



## Ejemplos de algoritmos recursivos



# Búsqueda binaria

Parte el problema en subproblemas más pequeños

Aplica el mismo proceso a cada subproblema

Naturaleza recursiva (casos base: encontrado o no queda lista)

*Partimos de la lista completa*

*Si no queda lista... terminar (lista vacía: no encontrado)*

*En caso contrario...*

*Comprobar si el elemento en la mitad es el buscado*

*Si es el buscado... terminar (encontrado)*

*Si no...*

*Si el buscado es menor que el elemento mitad...*

*Repetir con la primera mitad de la lista*

*Si el buscado es mayor que el elemento mitad...*

*Repetir con la segunda mitad de la lista*

→ La repetición se consigue con las llamadas recursivas



# Búsqueda binaria

---

Dos índices que indiquen el inicio y el final de la sublista:

```
int buscar(tLista lista, int buscado, int ini, int fin)  
// Devuelve el índice (0, 1, ...) o -1 si no está
```

¿Cuáles son los casos base?

- ✓ Que ya no quede sublista (`ini > fin`) → No encontrado
- ✓ Que se encuentre el elemento



Repasa en el Tema 7 cómo funciona y cómo se implementó iterativamente la búsqueda binaria (compárala con esta)



```
int buscar(tLista lista, int buscado, int ini, int fin) {  
    int pos = -1;  
    if (ini <= fin) {  
        int mitad = (ini + fin) / 2;  
        if (buscado == lista.elementos[mitad]) {  
            pos = mitad;  
        }  
        else if (buscado < lista.elementos[mitad]) {  
            pos = buscar(lista, buscado, ini, mitad - 1);  
        }  
        else {  
            pos = buscar(lista, buscado, mitad + 1, fin);  
        }  
    }  
    return pos;  
}
```

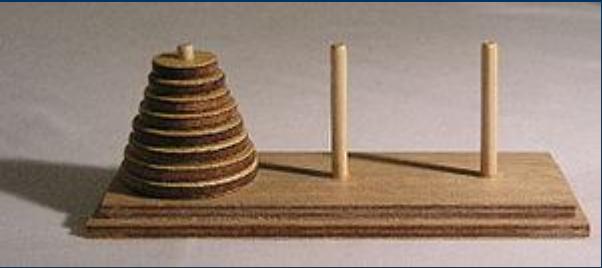
Llamada: pos = buscar(lista, valor, 0, lista.cont - 1);



# Las Torres de Hanoi

---

*Cuenta una leyenda que en un templo de Hanoi se dispusieron tres pilares de diamante y en uno de ellos 64 discos de oro, de distintos tamaños y colocados por orden de tamaño con el mayor debajo*



Torre de ocho discos ([wikipedia.org](https://en.wikipedia.org))

*Cada monje, en su turno, debía mover un único disco de un pilar a otro, para con el tiempo conseguir entre todos llevar la torre del pilar inicial a uno de los otros dos; respetando una única regla: nunca poner un disco sobre otro de menor tamaño*

*Cuando lo hayan conseguido, ¡se acabará el mundo!*

[Se requieren al menos  $2^{64}-1$  movimientos; si se hiciera uno por segundo, se terminaría en más de 500 mil millones de años]

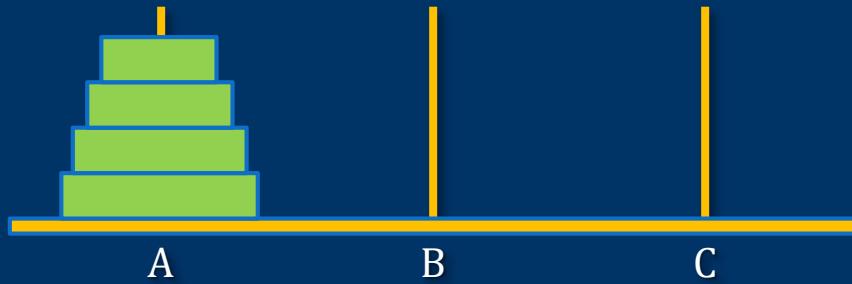


# Las Torres de Hanoi

---

Queremos resolver el *juego* en el menor número de pasos posible  
¿Qué disco hay que mover en cada paso y a dónde?

Identifiquemos los elementos (torre de cuatro discos):



Cada pilar se identifica con una letra

*Mover del pilar X al pilar Y:*

Coger el disco superior de X y ponerlo encima de los que haya en Y



# Las Torres de Hanoi

Resolución del problema en base  
a problemas más pequeños

Mover N discos del pilar A al pilar C:

Mover N-1 discos del pilar A al pilar B

Mover el disco del pilar A al pilar C

Mover N-1 discos del pilar B al pilar C

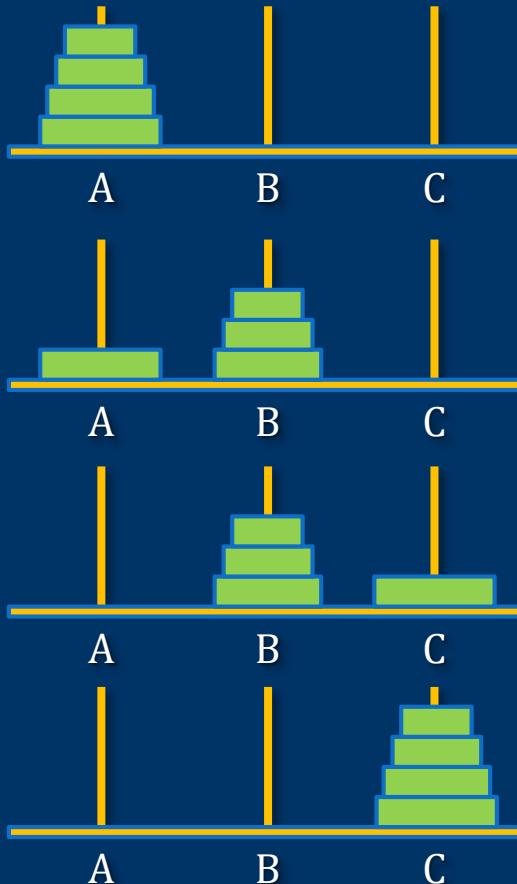
Para llevar N discos de un pilar *origen* a  
otro *destino* se usa el tercero como *auxiliar*

Mover N-1 discos del *origen* al *auxiliar*

Mover el disco del *origen* al *destino*

Mover N-1 discos del *auxiliar* al *destino*

Mover 4 discos de A a C



# Las Torres de Hanoi

Mover N-1 discos se hace igual, pero usando ahora otros origen y destino

Mover N-1 discos del pilar A al pilar B:

Mover N-2 discos del pilar A al pilar C

Mover el disco del pilar A al pilar B

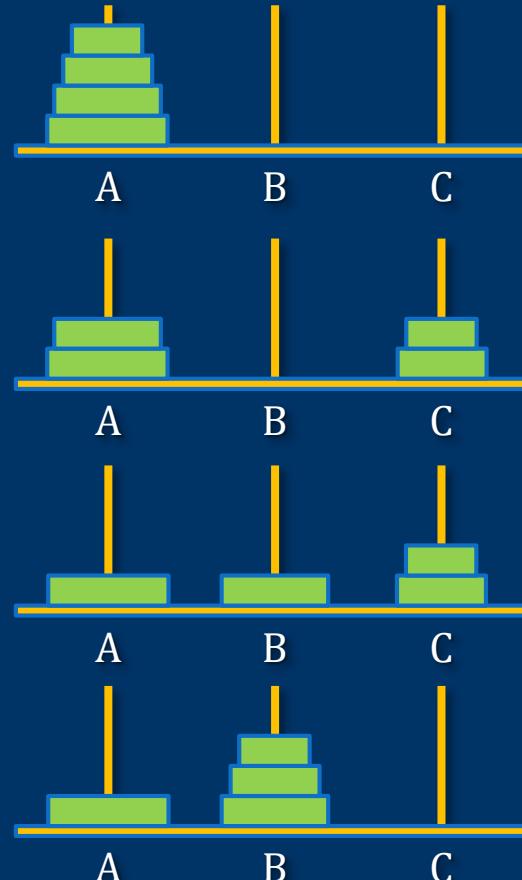
Mover N-2 discos del pilar C al pilar B

Naturaleza recursiva de la solución



Simulación para 4 discos ([wikipedia.org](https://en.wikipedia.org))

Mover 3 discos de A a B



Caso base: no quedan discos que mover

```
...
void hanoi(int n, char origen, char destino, char auxiliar) {
    if (n > 0) {
        hanoi(n - 1, origen, auxiliar, destino);
        cout << origen << " --> " << destino << endl;
        hanoi(n - 1, auxiliar, destino, origen);
    }
}

int main() {
    int n;
    cout << "Número de torres: ";
    cin >> n;
    hanoi(n, 'A', 'C', 'B');

    return 0;
}
```

```
Número de torres: 4
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
A --> B
C --> B
C --> A
B --> A
C --> B
A --> C
A --> B
C --> B
```



## Recursión frente a iteración



# Recursión frente a iteración

```
long long int factorial(int n) {  
    long long int fact;  
  
    assert(n >= 0);  
  
    if (n == 0) {  
        fact = 1;  
    }  
    else {  
        fact = n * factorial(n - 1);  
    }  
  
    return fact;  
}
```

```
long long int factorial(int n) {  
    long long int fact = 1;  
  
    assert(n >= 0);  
  
    for (int i = 1; i <= n; i++) {  
        fact = fact * i;  
    }  
  
    return fact;  
}
```



# Recursión frente a iteración

---

*¿Qué es preferible?*

Cualquier algoritmo recursivo tiene uno iterativo equivalente

Los recursivos son menos eficientes que los iterativos:

- Sobrecarga de las llamadas a subprograma

- Si hay una versión iterativa sencilla, será preferible a la recursiva

- En ocasiones la versión recursiva es mucho más simple

- Será preferible si no hay requisitos de rendimiento

Compara las versiones recursivas del factorial o de los números de Fibonacci con sus equivalentes iterativas

*¿Y qué tal una versión iterativa para los números de Ackermann?*



## Estructuras de datos recursivas



# Estructuras de datos recursivas

---

## *Definición recursiva de listas*

Ya hemos definido de forma recursiva alguna estructura de datos:

Secuencia { elemento seguido de una secuencia  
secuencia vacía (ningún elemento)

Las listas son secuencias:

Lista { elemento seguido de una lista  
lista vacía (ningún elemento) (Caso base)

La lista 1, 2, 3 consiste en el elemento 1 seguido de la lista 2, 3, que, a su vez, consiste en el elemento 2 seguido de la lista 3, que, a su vez, consiste en el elemento 3 seguido de la lista vacía (caso base)

Hay otras estructuras con naturaleza recursiva (p.e., los árboles) que estudiarás en posteriores cursos



# Estructuras de datos recursivas

---

## *Procesamiento de estructuras de datos recursivas*

Naturaleza recursiva de las estructuras: procesamiento recursivo

*Procesar (lista):*

*Si lista no vacía (caso base):*

*Procesar el primer elemento de la lista // Código anterior*

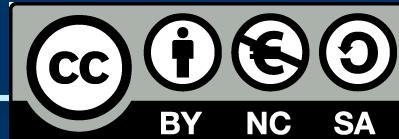
*Procesar (resto(lista))*

*Procesar el primer elemento de la lista // Código posterior*

*resto(lista): sublista tras quitar el primer elemento*



# Acerca de *Creative Commons*



## Licencia CC (*Creative Commons*)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:



Reconocimiento (*Attribution*):

En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.



No comercial (*Non commercial*):

La explotación de la obra queda limitada a usos no comerciales.



Compartir igual (*Share alike*):

La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Pulsa en la imagen de arriba a la derecha para saber más.

