Fundamentos de la programación

10 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

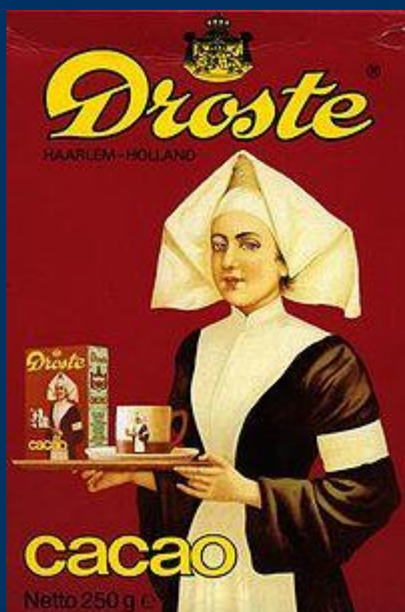
Recursión

Concepto de recursión

Recursión (recursividad, recurrencia)

Definición recursiva: En la definición aparece lo que se define

Factorial(N) = N x Factorial(N-1)   (N >= 0)



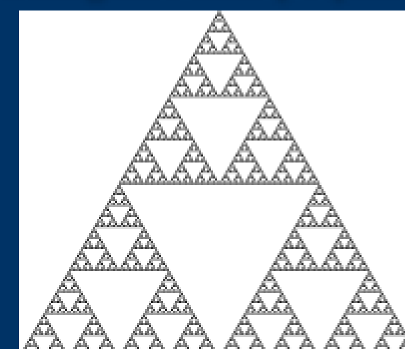
(wikipedia.org)

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)

Cada triángulo está formado por otros triángulos más pequeños



(wikipedia.org)

Las *matrioskas* rusas

Luis Hernández Yáñez

Definiciones recursivas

Factorial(N) = N x 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:

$$
\text{Factorial}(N) = \begin{cases} N \times \text{Factorial}(N-1) & \text{si } N > 0 \quad \textit{Caso recursivo (inducción)} \\ 1 & \text{si } N = 0 \quad \textit{Caso base (o de parada)} \end{cases}
$$

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

Luis Hernández Yáñez

Algoritmos recursivos

Luis Hernández Yáñez

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;
}
```

Algoritmos recursivos

factorial.cpp

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;
}
```

factorial(5) → 5 x factorial(4) → 5 x 4 x factorial(3)

→ 5 x 4 x 3 x factorial(2) → 5 x 4 x 3 x 2 x factorial(1)

→ 5 x 4 x 3 x 2 x 1 x factorial(0) → 5 x 4 x 3 x 2 x 1 x 1

→ 120 Caso base

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

Luis Hernández Yáñez

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*

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

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)

Luis Hernández Yáñez

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;
}
```

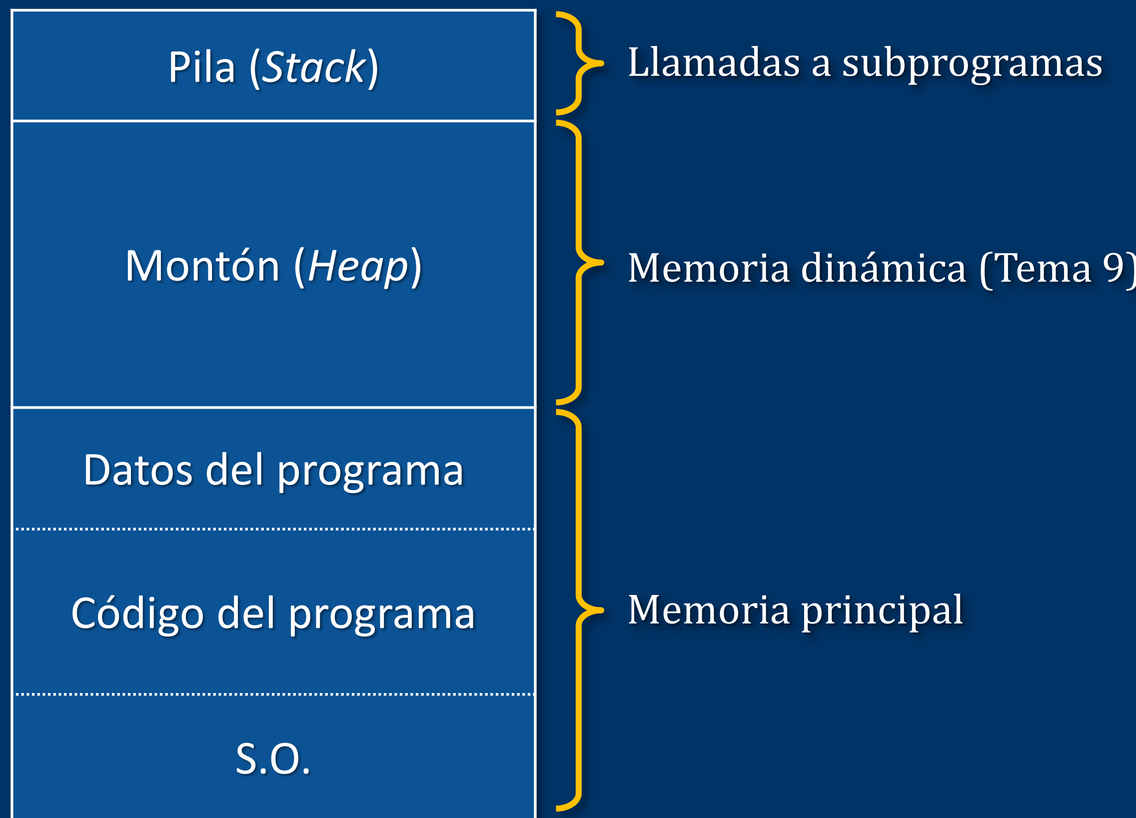
Cada llamada recursiva fuerza una nueva ejecución de la función

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:



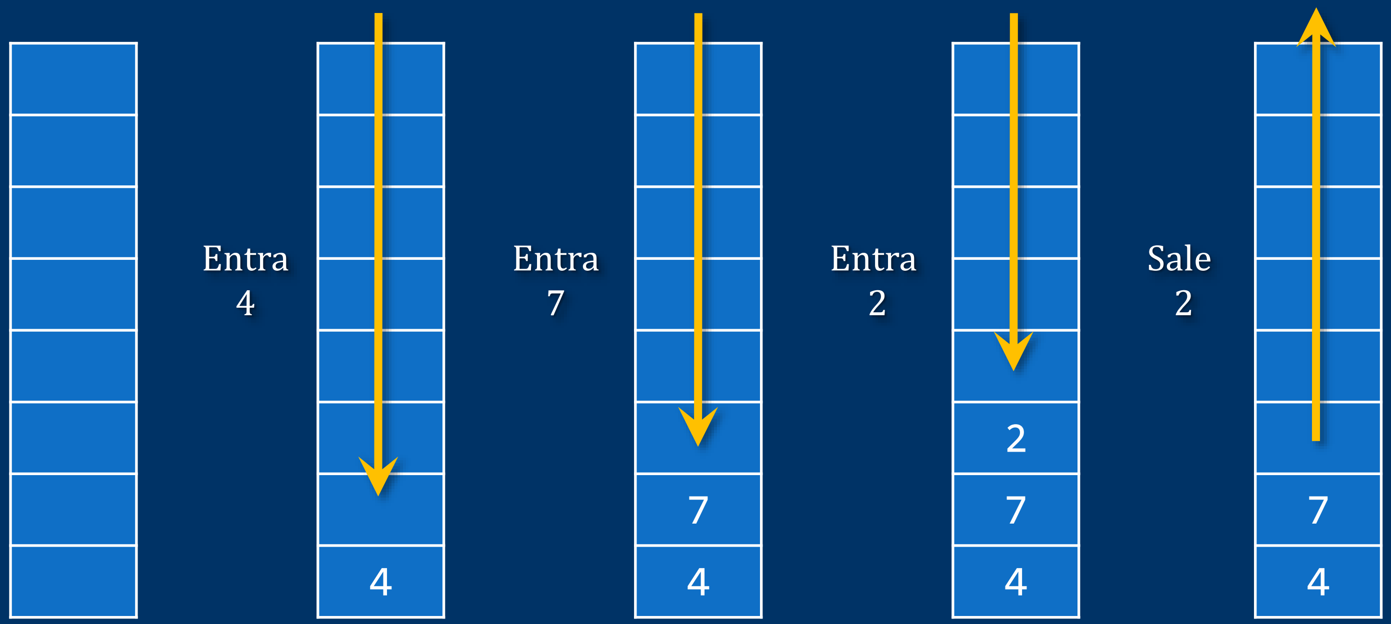
Luis Hernández Yáñez

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:



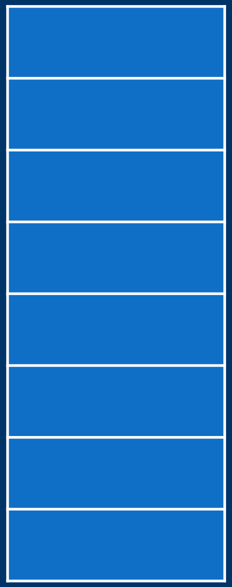
Luis Hernández Yáñez

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);
           ...
```

Llamada a función:
Entra la dirección de vuelta



Pila

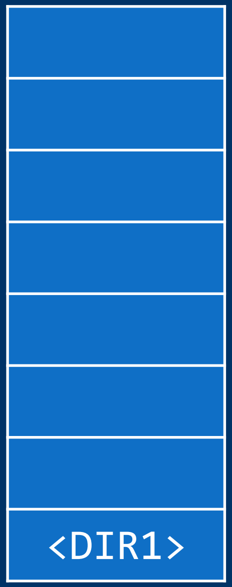
Luis Hernández Yáñez

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

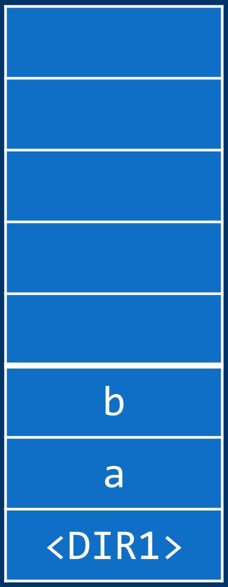
Luis Hernández Yáñez

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);
              ...
```

Llamada a función:
Entra la dirección de vuelta



Pila

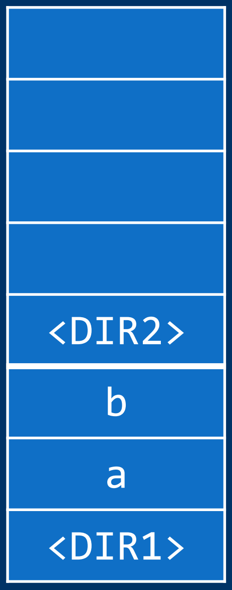
Luis Hernández Yáñez

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

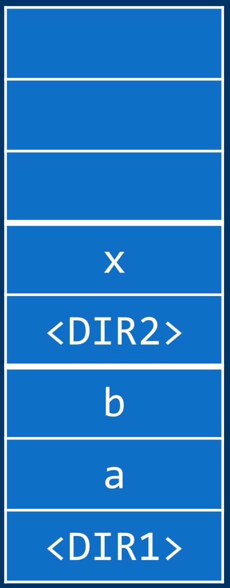
Luis Hernández Yáñez

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

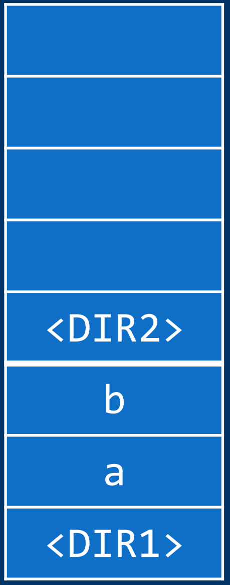
Luis Hernández Yáñez

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

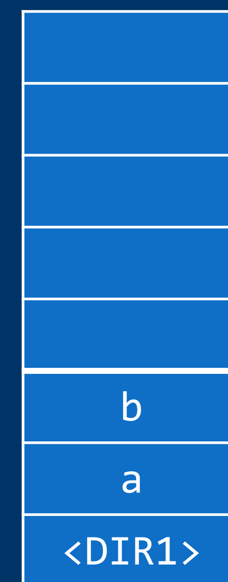
Luis Hernández Yáñez

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);
           ...
```

La ejecución continúa en esa dirección



Pila

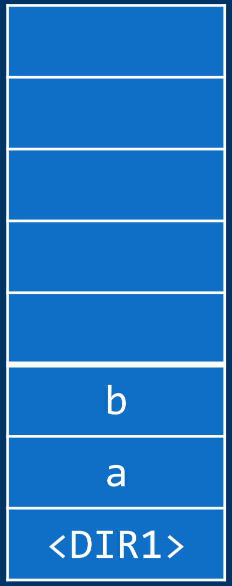
Luis Hernández Yáñez

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

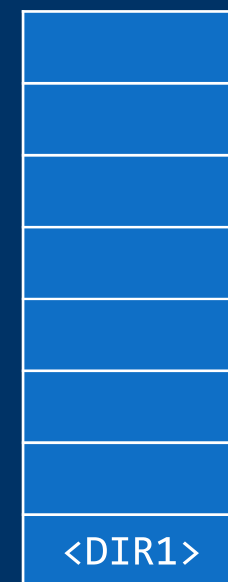
Luis Hernández Yáñez

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

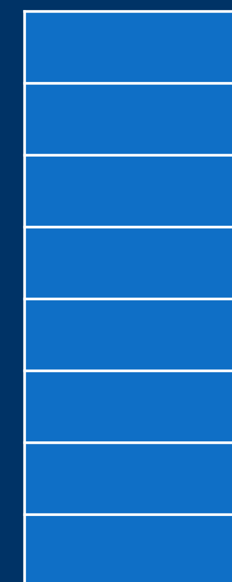
Luis Hernández Yáñez

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);
           ...
```

La ejecución continúa en esa dirección



Pila

Luis Hernández Yáñez

La pila y las llamadas a función

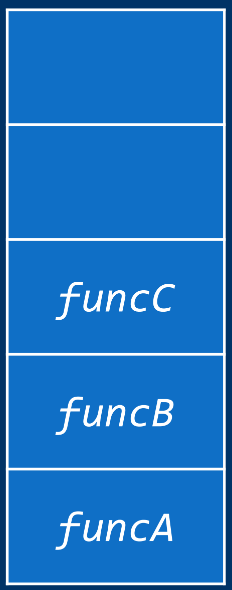
Mecanismo de pila adecuado para llamadas a funciones anidadas:

Las llamadas terminan en el orden contrario a como se llaman

VUELTAS

```
...
int funcC(...) {
  ...
}
int funcB(...) {
  ...
  ... funcC(...)
}
int funcA(...) {
  ...
  ... funcB(...)
}
int main() {
  ...
  cout << funcA(...);
  ...
```

LLAMADAS



Pila

Luis Hernández Yáñez

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

Luis Hernández Yáñez

Ejecución de la función `factorial()`

```
factorial(5)
```

Pila

Luis Hernández Yáñez

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
```

Pila
resultado = ?
n = 4
resultado = ?
n = 5

Pila

Luis Hernández Yáñez

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
       └→ factorial(3)
```

Pila
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila

Luis Hernández Yáñez

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

Luis Hernández Yáñez

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

Luis Hernández Yáñez

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

Luis Hernández Yáñez

Ejecución de la función factorial()

factorial(5)

→ factorial(4)

→ factorial(3)

→ factorial(2)

→ factorial(1)

→ factorial(0)

1

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

Pila

Luis Hernández Yáñez

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
       └→ factorial(3)
            └→ factorial(2)
                 └→ factorial(1)
                      └→ factorial(0)
                   ←┘
                      1
              ←┘
                 1
```

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)
                      ←┘
                       1
                 ←┘
                  1
            ←┘
             2
```

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

Pila

Luis Hernández Yáñez

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
       └→ factorial(3)
            └→ factorial(2)
                 └→ factorial(1)
                      └→ factorial(0)
                      ←┘
                        1
                 ←┘
                   1
            ←┘
              2
       ←┘
         6
```

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

Pila

Luis Hernández Yáñez

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
       └→ factorial(3)
            └→ factorial(2)
                 └→ factorial(1)
                      └→ factorial(0)
                      ←┘
                      1
                 ←┘
                 1
            ←┘
            2
       ←┘
       6
  ←┘
  24
```

Pila
resultado = 120
n = 5

Pila

Ejecución de la función factorial()

```
factorial(5)
  └→ factorial(4)
       └→ factorial(3)
            └→ factorial(2)
                 └→ factorial(1)
                      └→ factorial(0)
```

1

1

2

6

24

120

Pila

Luis Hernández Yáñez

Tipos de recursión

Luis Hernández Yáñez

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*

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

Dos llamadas recursivas

Luis Hernández Yáñez

Recursión múltiple

fibonacci.cpp

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

```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;
}
```

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

Luis Hernández Yáñez

Recursión anidada

En una llamada recursiva alguno de los argumentos es otra llamada

Ejemplo: Cálculo de los números de Ackermann:

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

Argumento que es una llamada recursiva

Luis Hernández Yáñez

Recursión anidada

ackermann.cpp

Números de Ackermann

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

```
...
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;
}
```

Pruébalo con números muy bajos:
Se generan MUCHAS llamadas recursivas

Luis Hernández Yáñez

Recursión anidada

Números de Ackermann

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

```
ackermann(1, 1)
   ackermann(0, ackermann(1, 0))
                                 2
                 ackermann(0, 1)
 3
   ackermann(0, 2)
```

Luis Hernández Yáñez

Recursión anidada

Números de Ackermann

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

```
ackermann(2, 1)
    ackermann(1, ackermann(2, 0))
                                 3
              ackermann(1, 1)
 5
                   ackermann(0, ackermann(1, 0))
                                                 2
                              ackermann(0, 1)
               3
                   ackermann(0, 2)
    ackermann(1, 3)
       ackermann(0, ackermann(1, 2))
                ackermann(0, ackermann(1, 1))
              4
                              ackermann(0, ackermann(1, 0))
      5
                                                           2
                                         ackermann(0, 1)
                              3
                              ackermann(0, 2)
                ackermann(0, 3)
       ackermann(0, 4)
```

Luis Hernández Yáñez

Código del subprograma recursivo

Luis Hernández Yáñez

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*

Luis Hernández Yáñez

Código del subprograma recursivo

Código anterior y posterior a la llamada recursiva

```cpp
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);`

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

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

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

Luis Hernández Yáñez

Código del subprograma recursivo

directo.cpp

Recorrido de los elementos de una lista (directo)

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

```cpp
...
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
```

Luis Hernández Yáñez

Código del subprograma recursivo `inverso.cpp`

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
```

Luis Hernández Yáñez

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

Luis Hernández Yáñez

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)

Luis Hernández Yáñez

Búsqueda binaria

binaria.cpp

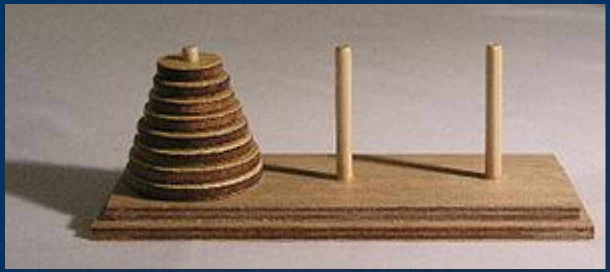
```
int buscar(tLista lista, int buscado, int ini, int fin) {
   int pos = -1;
   if (ini <= fin) {
      int mitad = (ini + fin) / 2;
      if (buscado == lista.elementos[mitad]) {
         pos = mitad;
      }
      else if (buscado < lista.elementos[mitad]) {
         pos = buscar(lista, buscado,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

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

Luis Hernández Yáñez

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)

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]

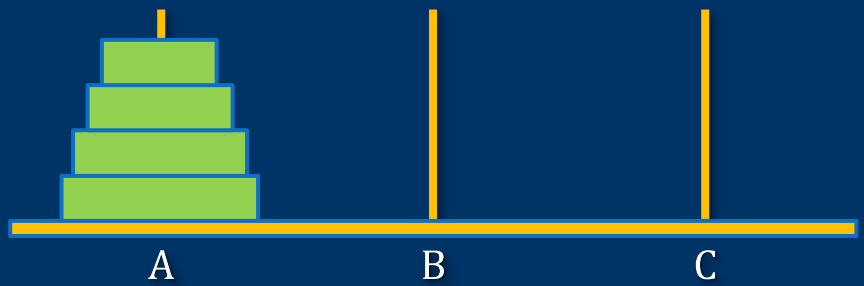
Luis Hernández Yáñez

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

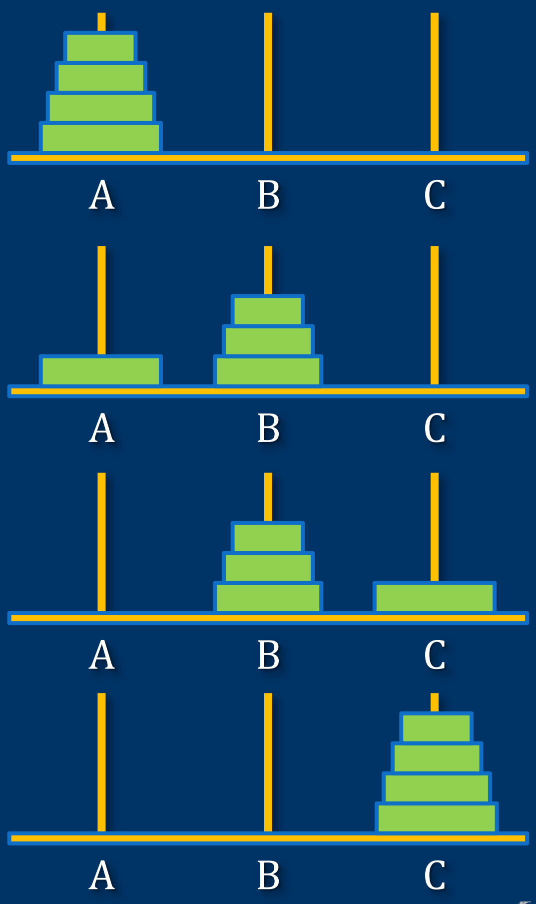
Mover N discos del pilar A al pilar C:

- 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



Luis Hernández Yáñez

Las *Torres de Hanoi*

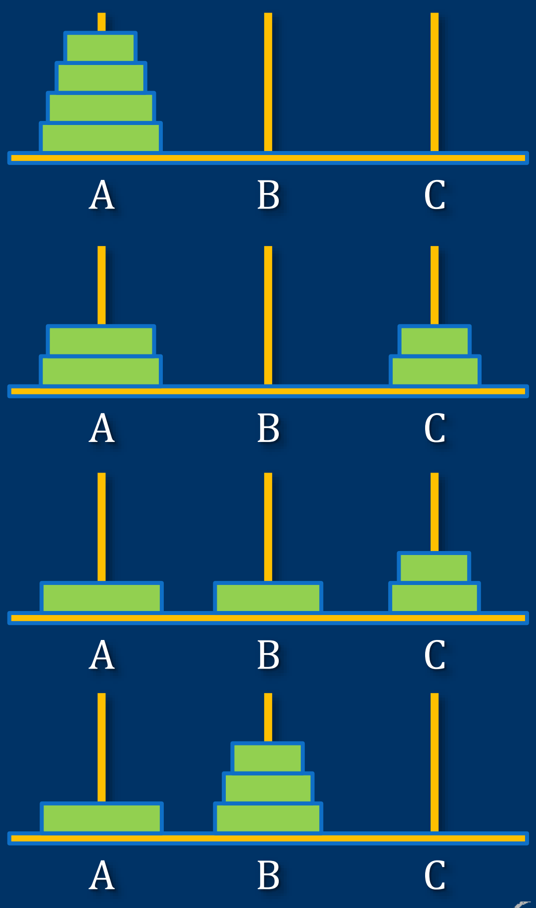
Mover N-1 discos se hace igual, pero usando ahora otros origen y destino

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

Mover 3 discos de A a B



Simulación para 4 discos (wikipedia.org)

Luis Hernández Yáñez

Las *Torres de Hanoi*

hanoi.cpp

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
```

Luis Hernández Yáñez

Recursión frente a iteración

Luis Hernández Yáñez

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;
}
```

Luis Hernández Yáñez

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?*

Luis Hernández Yáñez

Estructuras de datos recursivas

Luis Hernández Yáñez

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

Luis Hernández Yáñez

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.

Luis Hernández Yáñez