Fundamentos de la programación 2013-2014

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

Fundamentos de la programación

Recursión

Luis Hernández Yáñez

Concepto de recursión

Recursión (recursividad, recurrencia)

Se dice que algo se define de forma recursiva cuando en la definición aparece lo que se está definiendo.

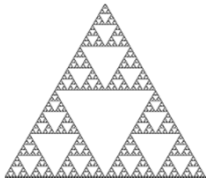
Factorial(N) = N x Factorial(N-1) (N > 0)

(wikipedia.org)

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

La imagen del paquete aparece dentro del propio paquete, que a su vez contiene otra imagen del paquete... *¡hasta el infinito!*

Cada triángulo está formado por otros triángulos más pequeños.



(wikipedia.org)

Las *matrioskas* rusas

Luis Hernández Yáñez

Concepto de recursión

Recursión

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

El factorial se define en función de sí mismo.

Los programas no pueden manejar la recursión infinita, que generaría un bucle infinito en el programa.

La definición recursiva ha de ir acompañada de uno o más casos base.

Un *caso base* es aquel en el que no se utiliza la definición recursiva, sino que se proporciona un punto final de cálculo:

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

Cada vez que se aplica el caso recursivo, el valor de N se va aproximando al valor del caso base (0).

Luis Hernández Yáñez

Fundamentos de la programación

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;
}
```

Luis Hernández Yáñez

Algoritmos recursivos

Funciones recursivas

`factorial.cpp`

```
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;
}
```

```
D:\FP\Tema11>factorial
1
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
```

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

Luis Hernández Yáñez

Algoritmos recursivos

Diseño de funciones recursivas

Una función recursiva debe satisfacer tres condiciones para estar bien diseñada:

- ✓ Caso(s) base: Debe haber al menos un caso base de parada.
- ✓ Inducción: Paso recursivo que provoca una llamada recursiva.
  Se podrá demostrar que es correcto para distintos parámetros de entrada.
- ✓ Convergencia: Cada paso recursivo debe acercarse al caso base.

Describimos el problema en términos de problemas *más sencillos*.

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

La 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

Fundamentos de la programación

Modelo de ejecución

Luis Hernández Yáñez

Funciones recursivas

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, quedando interrumpida la actual.

Cada llamada a la función utiliza sus propios parámetros por valor y variables locales (`n` y `resultado` en este caso).

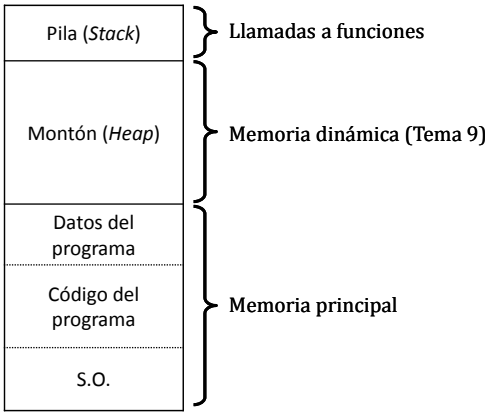
En las llamadas a la función se utiliza la pila del sistema para mantener los datos locales y la dirección de vuelta.

Luis Hernández Yáñez

Funciones recursivas

La pila del sistema (stack)

El SO dispone en la memoria de la computadora varias regiones con diferentes propósitos:



Luis Hernández Yáñez

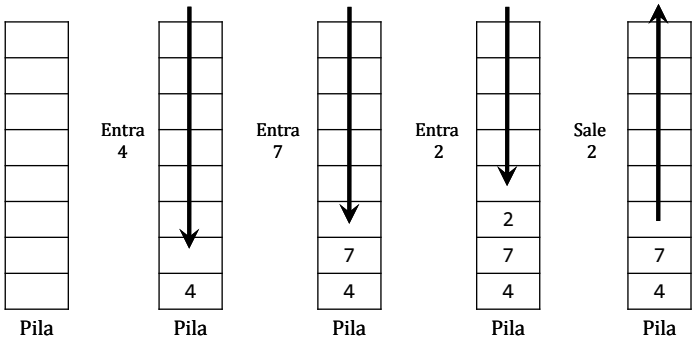
Funciones recursivas

La pila del sistema (stack)

La pila se utiliza para guardar en cada llamada a función los datos locales de la función y la dirección de vuelta.

Es una estructura de tipo *pila*: lista LIFO (*last-in first-out*)

El último que entra es el primero que sale:



Luis Hernández Yáñez

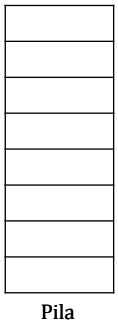
Funciones recursivas

La pila y las llamadas a función

Cada vez que se produce una llamada a una función se alojan en la pila sus datos locales y la dirección 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:
Se guarda la dirección de vuelta



Luis Hernández Yáñez

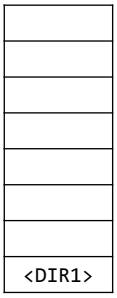
Funciones recursivas

La pila y las llamadas a función

Cada vez que se produce una llamada a una función se alojan en la pila sus datos locales y la dirección 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

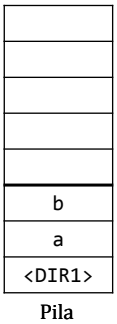
Funciones recursivas

La pila y las llamadas a función

Cada vez que se produce una llamada a una función se alojan en la pila sus datos locales y la dirección 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:
Se guarda la dirección de vuelta



Pila

Luis Hernández Yáñez

Funciones recursivas

La pila y las llamadas a función

Cada vez que se produce una llamada a una función se alojan en la pila sus datos locales y la dirección 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

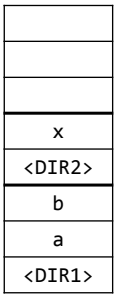
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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

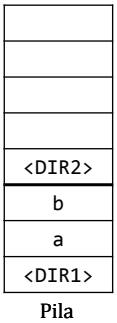
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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 obtiene la dirección de vuelta



Pila

Luis Hernández Yáñez

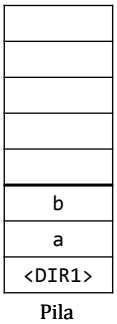
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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

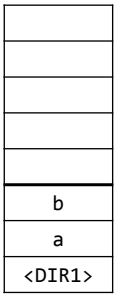
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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

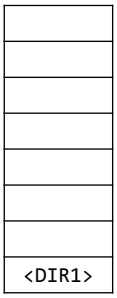
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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 obtiene la dirección de vuelta



Pila

Luis Hernández Yáñez

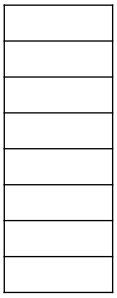
Funciones recursivas

La pila y las llamadas a función

Cada vez que se vuelve de una llamada a una función se recupera de la pila la dirección de vuelta de la función anterior.

```
        ...
        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

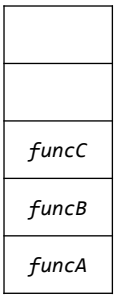
La pila y las llamadas a función

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

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

VUELTAS

LLAMADAS



Pila

Funciones recursivas

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;
}
```

Cada vez que se realiza una llamada recursiva se guardan en la pila los datos locales (`n` y `resultado`) y la dirección de vuelta.

Veamos cómo se ejecuta `factorial(5)`, obviando las direcciones de vuelta, que son datos de bajo nivel y no afectan a la explicación.

Funciones recursivas

Ejecución de la función `factorial()`

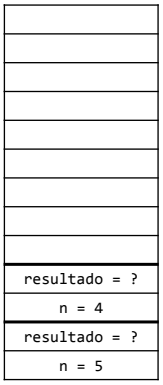
```
factorial(5)
```

Pila

Funciones recursivas

Ejecución de la función `factorial()`

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

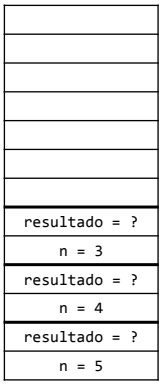


Pila

Funciones recursivas

Ejecución de la función `factorial()`

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

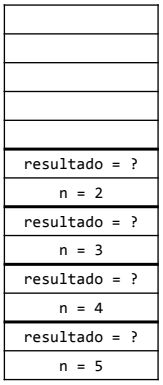


Pila

Funciones recursivas

Ejecución de la función `factorial()`

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



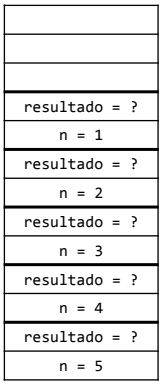
Pila

Luis Hernández Yáñez

Funciones recursivas

Ejecución de la función `factorial()`

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



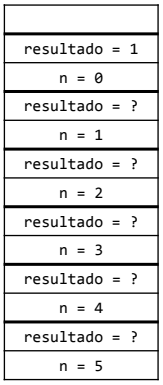
Pila

Luis Hernández Yáñez

Funciones recursivas

Ejecución de la función `factorial()`

```
factorial(5)
  ↳ factorial(4)
      ↳ factorial(3)
          ↳ factorial(2)
              ↳ factorial(1)
                  ↳ factorial(0)
```



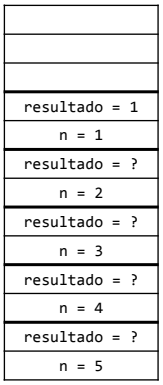
Pila

Luis Hernández Yáñez

Funciones recursivas

Ejecución de la función `factorial()`

```
factorial(5)
  ↳ factorial(4)
      ↳ factorial(3)
          ↳ factorial(2)
              ↳ factorial(1)
                  ↳ factorial(0)
                  ←
                  1
```

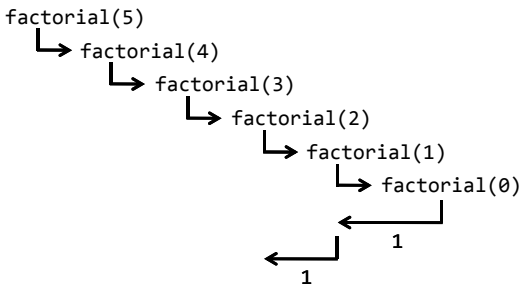


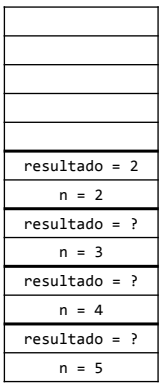
Pila

Luis Hernández Yáñez

Funciones recursivas

Ejecución de la función `factorial()`

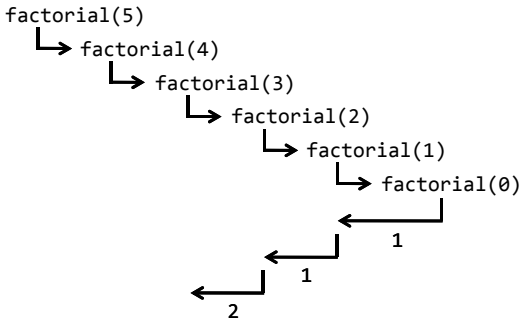


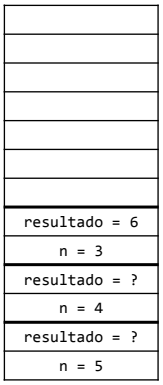


Pila

Funciones recursivas

Ejecución de la función `factorial()`

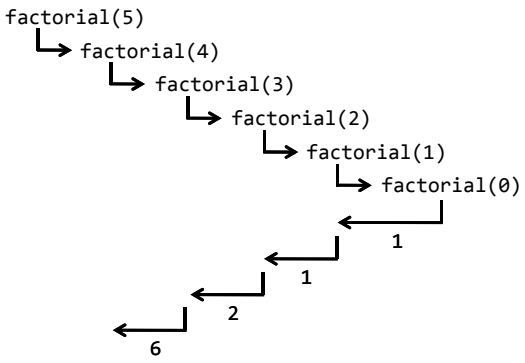


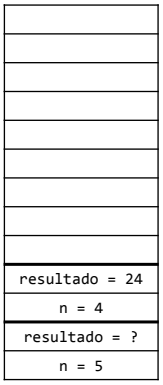


Pila

Funciones recursivas

Ejecución de la función `factorial()`

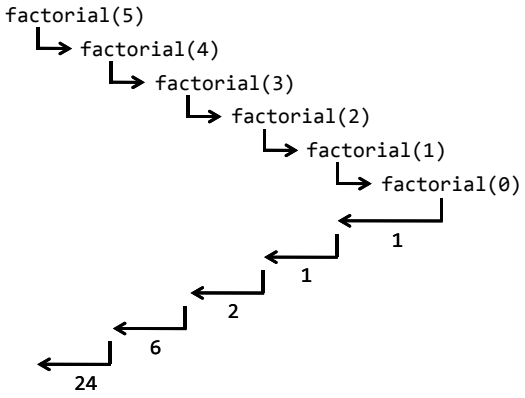


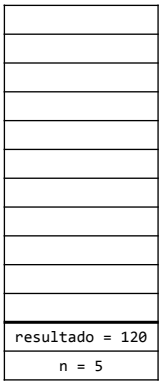


Pila

Funciones recursivas

Ejecución de la función `factorial()`

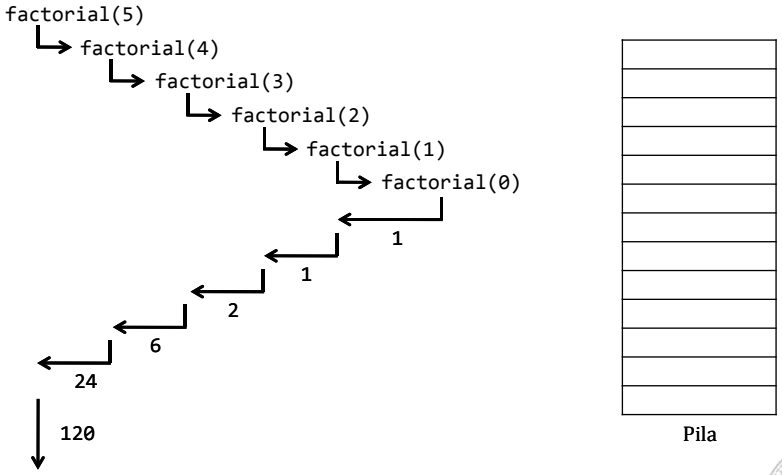




Pila

Funciones recursivas

Ejecución de la función `factorial()`



Luis Hernández Yáñez

Fundamentos de la programación

Tipos de recursión

Luis Hernández Yáñez

Tipos de recursión

Recursión simple

Decimos que la recursión es simple cuando en la función sólo hay una llamada recursiva.

Por ejemplo, el 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 llamada recursiva

Luis Hernández Yáñez

Tipos de recursión

Recursión múltiple

Cuando en la función hay varias llamadas recursivas a la propia función, decimos que tenemos recursión múltiple.

Por ejemplo, el cálculo de los números de *Fibonacci*:

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

Dos llamadas recursivas

Luis Hernández Yáñez

Tipos de recursión

Recursión múltiple: los números de Fibonacci

fibonacci.cpp

```
...
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;
}

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

   return 0;
}
```

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

```
D:\FP\Tema11>fibonacci
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
```

Luis Hernández Yáñez

Tipos de recursión

Recursión anidada

La recursión anidada la tenemos cuando en una llamada recursiva alguno de los argumentos consiste en otra llamada recursiva.

Por ejemplo, el 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

Tipos de recursión

Recursión anidada: los números de Ackermann

ackermann.cpp

```
...
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;
}

int main() {
   int m, n;
   cout << "M = "; cin >> m;
   cout << "N = "; cin >> n;
   cout << ackermann(m, n) << endl;
   return 0;
}
```

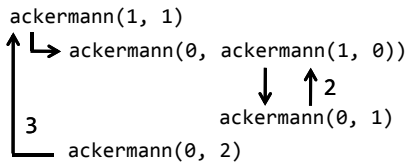
$$Ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ack(m-1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Pruébalo con números muy bajos:
Se generan MUCHAS llamadas recursivas

Luis Hernández Yáñez

Tipos de recursión

Recursión anidada: los números de Ackermann

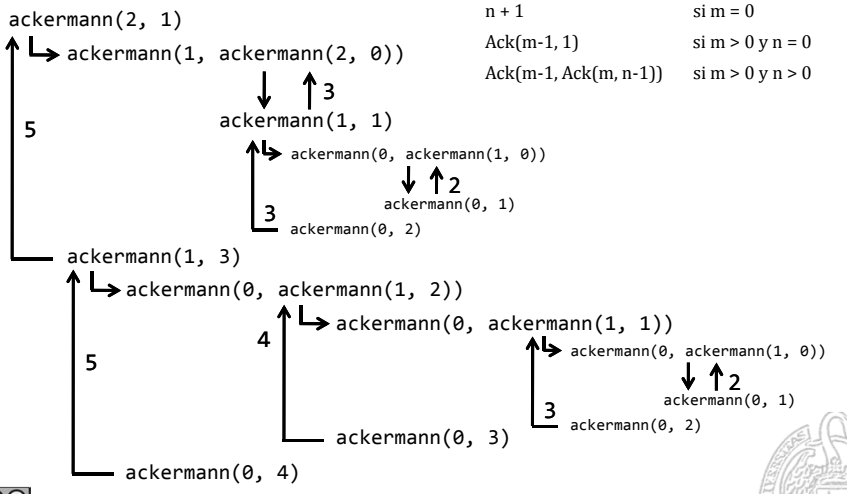


$$Ack(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ Ack(m-1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ Ack(m-1, Ack(m, n-1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Luis Hernández Yáñez

Tipos de recursión

Recursión anidada: los números de Ackermann



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

Luis Hernández Yáñez

Tipos de recursión

Recursión cruzada o indirecta

par.cpp

La recursión cruzada o indirecta se da cuando una función llama a otra y ésta a su vez llama a la primera.

Por ejemplo, saber si un número es par o impar:

```
bool par(int n) {
   if (n == 0) return true;
   else return impar(n - 1);
}
bool impar(int n) {
   if (n == 0) return false;
   else return par(n - 1);
}
int main() {
   int x;
   cout << "Num: "; cin >> x;
   if (par(x)) cout << "Es par";
   else cout << "Es impar";
   ...
```

Luis Hernández Yáñez

Fundamentos de la programación

Código del subprograma recursivo

Luis Hernández Yáñez

Código del subprograma recursivo

Código anterior y posterior a la llamada recursiva

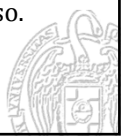
En general, en un subprograma recursivo tendremos un código anterior a la llamada recursiva y un código posterior:

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

El *código anterior* se ejecutará repetidas veces para las distintas entradas antes de que se llegue a ejecutar cualquier *código posterior*. Éste se ejecutará repetidas veces para las distintas entradas tras llegar al caso base. Si no hay código anterior, tenemos *recursión por delante*.

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 posterior, tenemos *recursión por detrás*.

Luis Hernández Yáñez



Código del subprograma recursivo

Código anterior y posterior a la llamada recursiva

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

Si llamamos a la función con el argumento 5, la salida será:

```
D:\FP\Tema11>prueba
Entrando (5)
Entrando (4)
Entrando (3)
Entrando (2)
Entrando (1)
Saliendo (1)
Saliendo (2)
Saliendo (3)
Saliendo (4)
Saliendo (5)
```

El código anterior a la llamada se ejecuta para los distintos valores que va tomando n.
El código posterior a la llamada, al revés.

Luis Hernández Yáñez

Código del subprograma recursivo

Recorrido de los elementos de una lista (directo) `directo.cpp`

Con el código anterior a la llamada procesamos los elementos en el mismo orden en que se encuentran:

```
...
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);
   }
}
```

```
D:\FP\Tema11>directo
1
3
8
13
17
22
23
39
52
55
```

Luis Hernández Yáñez

Código del subprograma recursivo

Recorrido de los elementos de una lista (inverso) `inverso.cpp`

Con el código posterior a la llamada procesamos los elementos en el orden inverso en que se encuentran:

```
...
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;
   }
}
```

```
D:\FP\Tema11>inverso
55
52
39
23
22
17
13
8
3
1
```


Fundamentos de la programación

Parámetros y recursión

Parámetros y recursión

Parámetros por valor y por referencia

En los subprogramas recursivos los parámetros por valor son datos que el subprograma necesita para realizar sus operaciones.
(En cada llamada recursiva pueden ser distintos.)

Los parámetros por referencia son resultados que se van construyendo a lo largo de las distintas llamadas (misma variable en las llamadas):

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

En la llamada a `factorial(0)`, `fact` toma el valor 1. A la vuelta, se le multiplica por el `n` de la llamada anterior (1). Y a la vuelta, por el `n` de la llamada anterior (2). Y así sucesivamente...

Fundamentos de la programación

Ejemplos de algoritmos recursivos

Ejemplos de algoritmos recursivos

Búsqueda binaria

La búsqueda binaria es un ejemplo de algoritmo que trabaja partiendo el problema en subproblemas más pequeños y aplicando el mismo proceso a cada subproblema.

Esa *repetición* del proceso para cada subproblema muestra una naturaleza recursiva que nos permite implementarla así.

Partimos de la lista completa en la que hay que buscar.
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...
            Quedarse con la primera mitad de la lista y repetir el proceso
        Si el buscado es mayor que el elemento mitad...
            Quedarse con la segunda mitad de la lista y repetir el proceso

Ejemplos de algoritmos recursivos

Búsqueda binaria

A cada ejecución de la función se le pasan dos índices que indiquen las posiciones inicial y final de la sublista a procesar dentro de la lista:

```
int buscar(tLista lista, int buscado, int ini, int fin)
// Devuelve la posición donde está (0, ...) 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 cómo funciona y cómo se implementó la búsqueda binaria en el Tema 7 (implementación iterativa).

Ejemplos de algoritmos recursivos

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;
}
```

1ª llamada:

```
pos = buscar(lista, buscado, 0, lista.cont - 1);
```

Luis Hernández Yáñez

Ejemplos de algoritmos recursivos

Búsqueda binaria: implementación con parámetros de salida

```
void buscar(tLista lista, int buscado, int ini, int fin,
            bool & encontrado, int & pos) {
   if (ini > fin)  encontrado = false;
   else {
      int mitad = (ini + fin) / 2;
      if (buscado == lista.elementos[mitad]) {
         encontrado= true;
         pos = mitad;
      }//if
      else {
         if (buscado < lista.elementos[mitad])
            buscar(lista, buscado, ini, mitad - 1, encontrado, pos);
         else
            buscar(lista, buscado, mitad + 1, fin, encontrado, pos);
      }//else
   }//else
}
```

1ª llamada:

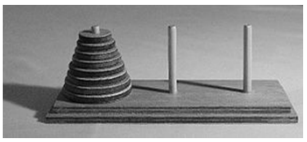
```
buscar(lista, buscado, 0, lista.cont - 1, enc, i);
```

Luis Hernández Yáñez
Pedro J. Martín de la Calle

Ejemplos de algoritmos recursivos

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 radios 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, de forma que entre todos consigan con el tiempo llevar la torre del pilar inicial a uno de los otros dos. Debían respetar una 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.]

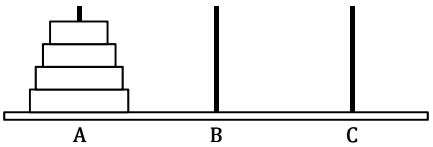
Ejemplos de algoritmos recursivos

Las torres de Hanoi

Queremos intentar resolver el *juego* en el menor número de movimientos posible.

¿Qué disco hay que mover en cada paso y a dónde hay que moverlo?

Primero, identifiquemos los elementos (torre de cuatro discos):



Cada pilar se identifica con una letra.

Mover del pilar X al pilar Y significará coger el disco superior de X y ponerlo encima de los que haya en Y.

Ejemplos de algoritmos recursivos

Las torres de Hanoi

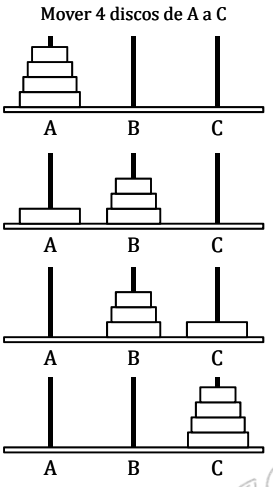
Debemos enfocar la resolución del problema en base a problemas más pequeños.

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 pilar *destino* se utiliza el tercer pilar 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*



Luis Hernández Yáñez

Ejemplos de algoritmos recursivos

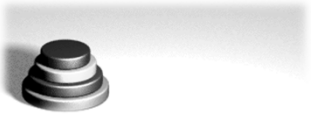
Las torres de Hanoi

Mover N-1 discos de un pilar a otro se realiza de la misma forma, pero utilizando ahora otros pilares como origen y destino.

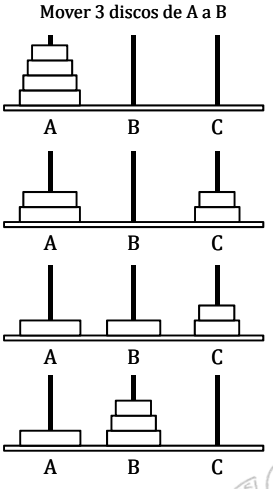
Mover N-1 discos del pilar A al pilar B:

- 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

Claramente se puede resolver fácilmente con una implementación recursiva.



Simulación para 4 discos (wikipedia.org)



Luis Hernández Yáñez

Ejemplos de algoritmos recursivos

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) return;
   else {
      hanoi(n - 1, origen, auxiliar, destino);
      cout << origen << " --> " << destino << endl;
      hanoi(n - 1, auxiliar, destino, origen);
   }
}

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

   return 0;
}
```

```
D:\FP\Tema11>hanoi
N. 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

Fundamentos de la programación

Recursión frente a iteración

Luis Hernández Yáñez

Recursión frente a iteración

¿Qué es preferible?

Cualquier algoritmo recursivo se puede traducir a un algoritmo iterativo que realice el mismo trabajo.

Los algoritmos recursivos son menos eficientes que sus alternativas iterativas: sobrecarga de las llamadas a los subprogramas.

Siempre que resulte sencillo desarrollar una versión iterativa, ésta será preferible a su versión recursiva.

Pero en ocasiones no resulta sencillo obtener esa versión iterativa equivalente, siendo la versión recursiva mucho más simple.
En estos casos será preferible la versión recursiva
(si el rendimiento no es un factor clave).

Desarrolla versiones iterativas del factorial o de los números de Fibonacci. *¿Y qué tal con los números de Ackermann?*

Luis Hernández Yáñez

Fundamentos de la programación

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 también se pueden definir de forma similar:

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 (como los árboles) que estudiarás en posteriores cursos.

Luis Hernández Yáñez

Estructuras de datos recursivas

Procesamiento de estructuras recursivas

La naturaleza recursiva de algunas estructuras de datos lleva de forma natural a un procesamiento recursivo de las mismas:

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

Donde *resto(lista)* devuelve una lista igual a la que se proporciona sin su primer elemento.

Luis Hernández Yáñez

Referencias bibliográficas

- ✓ *C++: An Introduction to Computing* (2ª edición)
  J. Adams, S. Leestma, L. Nyhoff. Prentice Hall, 1998
- ✓ *El lenguaje de programación C++* (Edición especial)
  B. Stroustrup. Addison-Wesley, 2002
- ✓ *Programación en C++ para ingenieros*
  F. Xhafa et al. Thomson, 2006

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.