

Fundamentos de la programación

Introducción a la recursión

Facultad de Informática
Universidad Complutense

Ana Gil Luezas
(Adaptadas del original de Luis Hernández Yáñez)



Índice

Algoritmos recursivos	2
Tipos de recursión	
Recursión simple	18
recursión múltiple	19
Recursión anidada	21
Recursión cruzada	25
Recursión simple	
Código anterior y posterior	26
Recursión final y	
Parámetros acumuladores	30
Ejemplos de algoritmos recursivos	
Búsqueda binaria	34
Torres de Hanoi	37



Concepto de recursión

$$\text{Factorial}(N) = N \times \underbrace{N-1 \times \dots \times 1}_{\downarrow}$$

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

El factorial se define en función de sí mismo

Para calcular el factorial de N hay que calcular antes el de $N-1$

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

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



Algoritmos recursivos

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

```
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;  
}
```

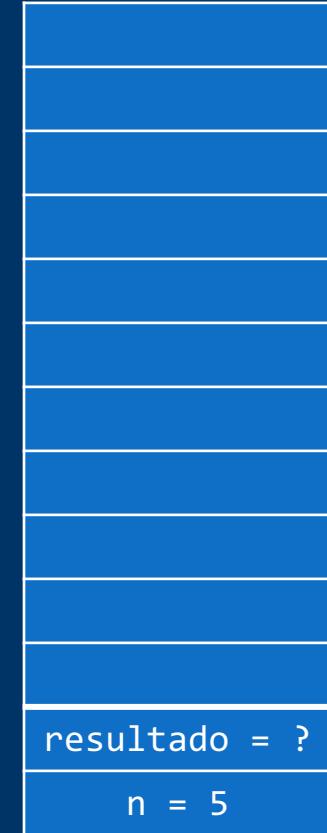
$\text{factorial}(5) \rightarrow 5 \times \text{factorial}(4) \rightarrow 5 \times 4 \times \text{factorial}(3)$
 $\rightarrow 5 \times 4 \times 3 \times \text{factorial}(2) \rightarrow 5 \times 4 \times 3 \times 2 \times \text{factorial}(1)$
 $\rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times \underline{\text{factorial}(0)} \rightarrow 5 \times 4 \times 3 \times 2 \times 1 \times 1 \rightarrow 5 \times 4 \times 3 \times 2 \times 1$
Caso base
 $\rightarrow 5 \times 4 \times 3 \times 2 \rightarrow 5 \times 4 \times 6 \rightarrow 5 \times 24 \rightarrow 120$

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



Ejecución de la función factorial()

factorial(5)



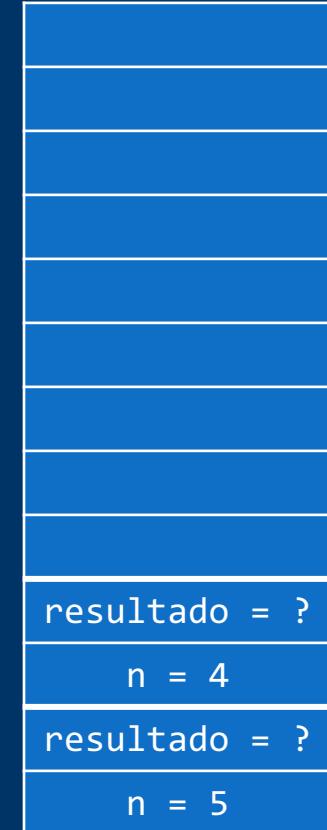
Pila



Ejecución de la función factorial()

factorial(5)

↳ factorial(4)

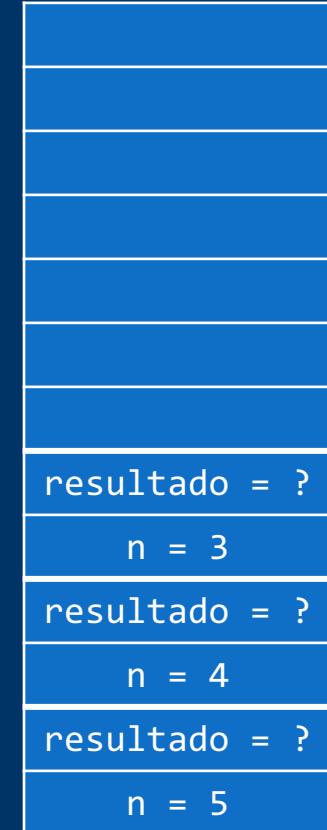


Pila



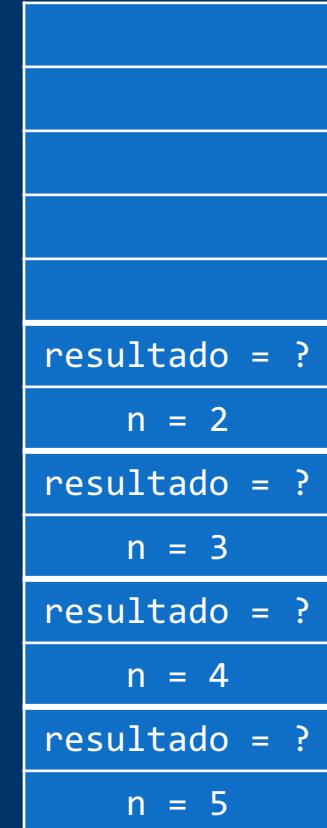
Ejecución de la función factorial()

```
factorial(5)
  ↳ factorial(4)
    ↳ factorial(3)
```



Ejecución de la función factorial()

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



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)
```

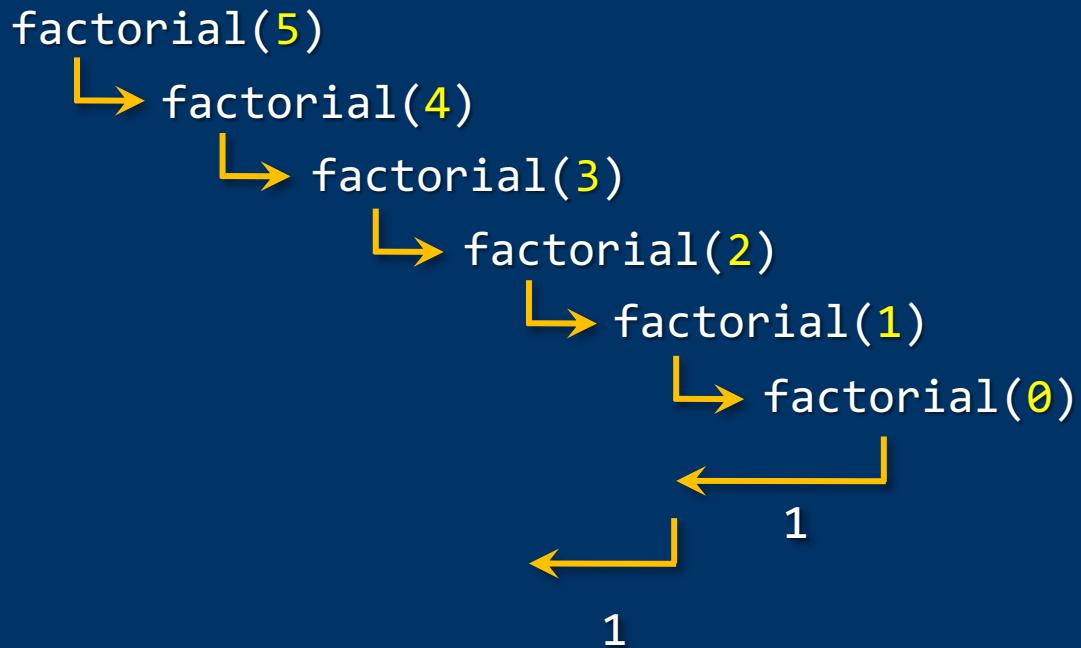


resultado = 1
n = 1
resultado = ?
n = 2
resultado = ?
n = 3
resultado = ?
n = 4
resultado = ?
n = 5

Pila



Ejecución de la función factorial()



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



Ejecución de la función factorial()

```
factorial(5)
```

```
  ↘ factorial(4)
```

```
    ↘ factorial(3)
```

```
      ↘ factorial(2)
```

```
        ↘ factorial(1)
```

```
          ↘ factorial(0)
```

```
            ↙ 1
```

```
            ↙ 1
```

```
            ↙ 2
```

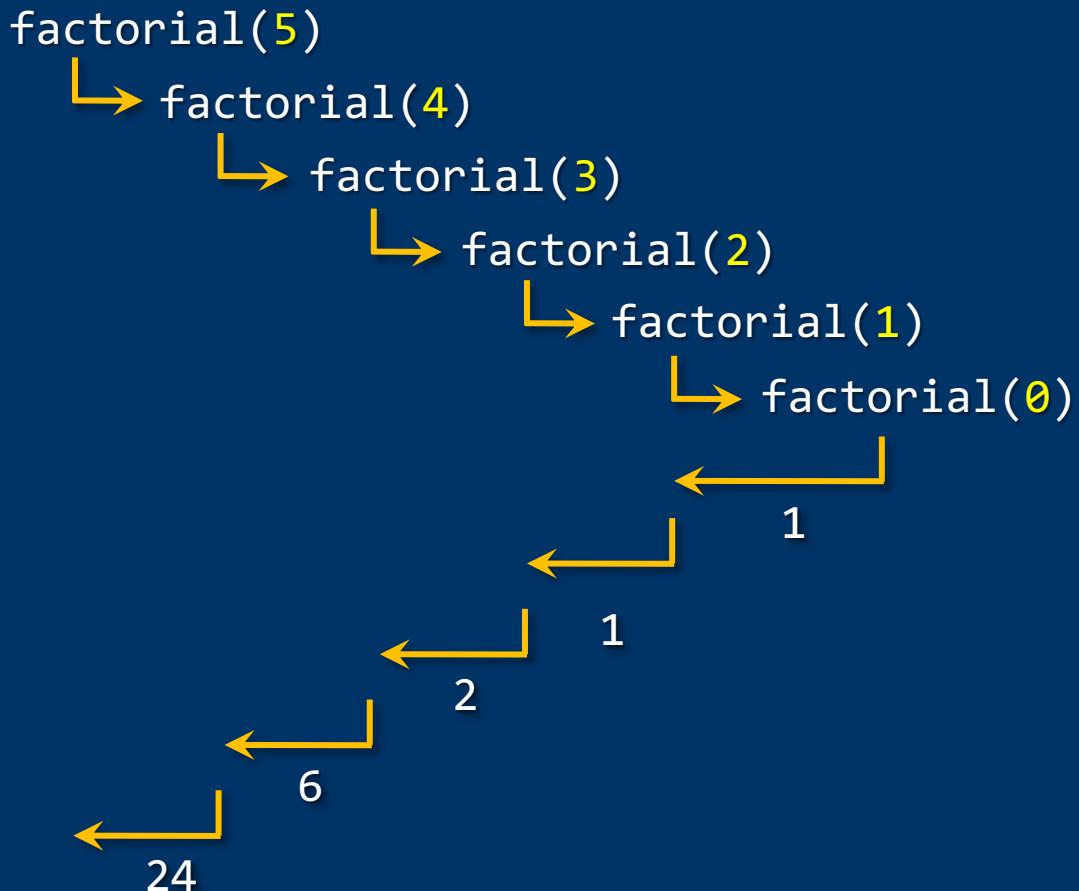
```
            ↙ 6
```

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

Pila



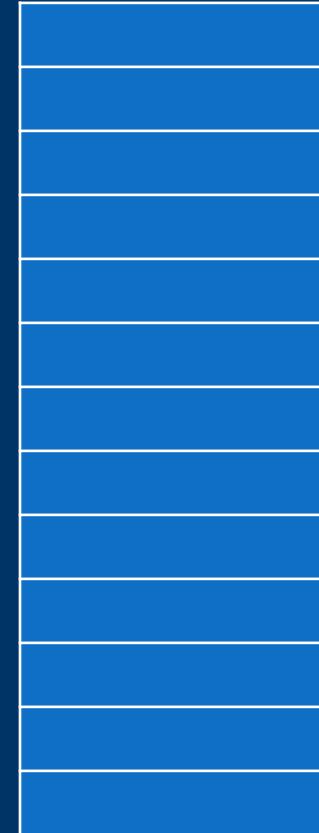
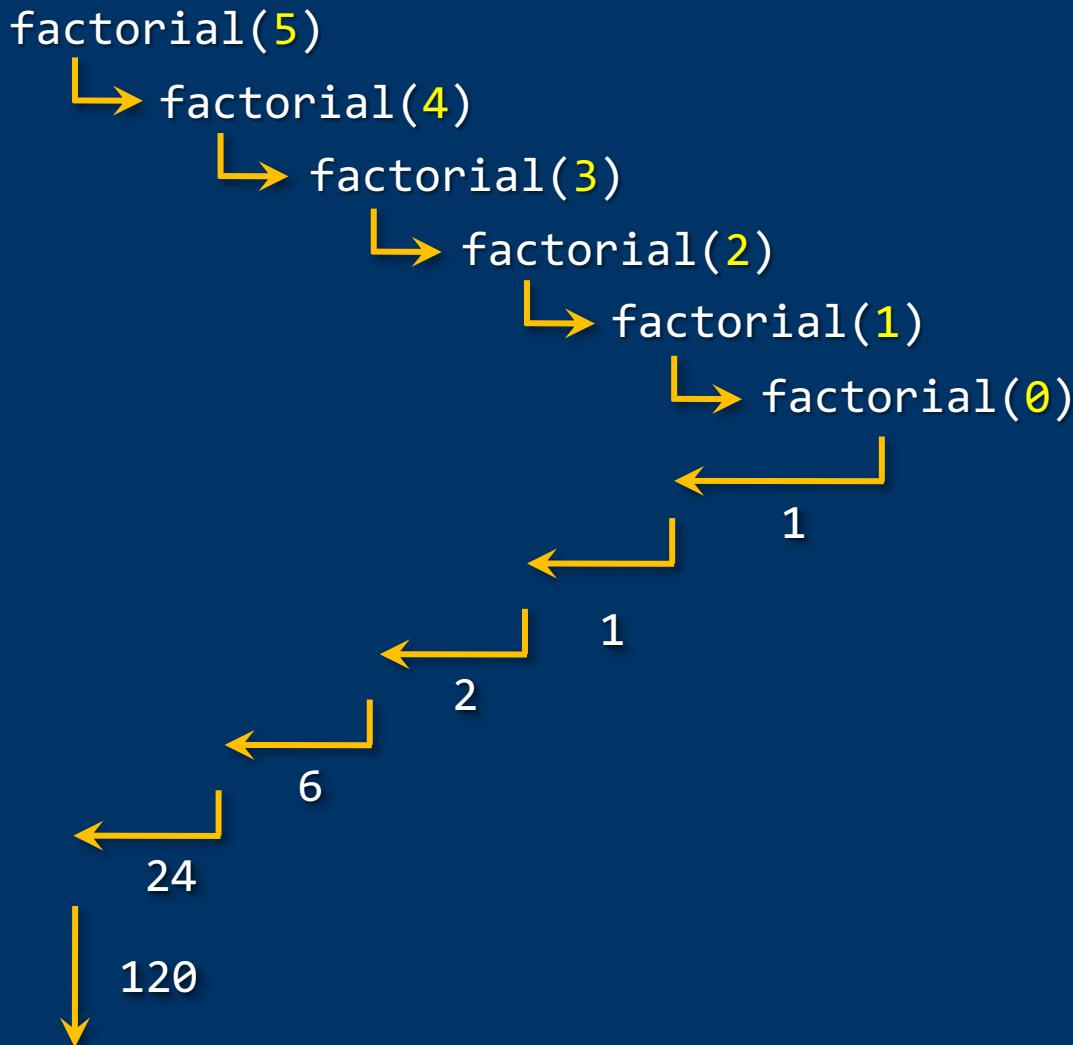
Ejecución de la función factorial()



Pila



Ejecución de la función factorial()



Pila



Tipos de recursión



Tipos de recursión

Recursión simple: Sólo hay una llamada recursiva

Ejemplo: cálculo del 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;  
}
```



Una llamada recursiva



Tipos de recursión

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



Los números de *Fibonacci*

```
...
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;
}
```

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

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



Tipos de recursión

Recursión anidada: En una llamada recursiva hay argumentos que son llamadas recursivas

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



Los 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;
}
```

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

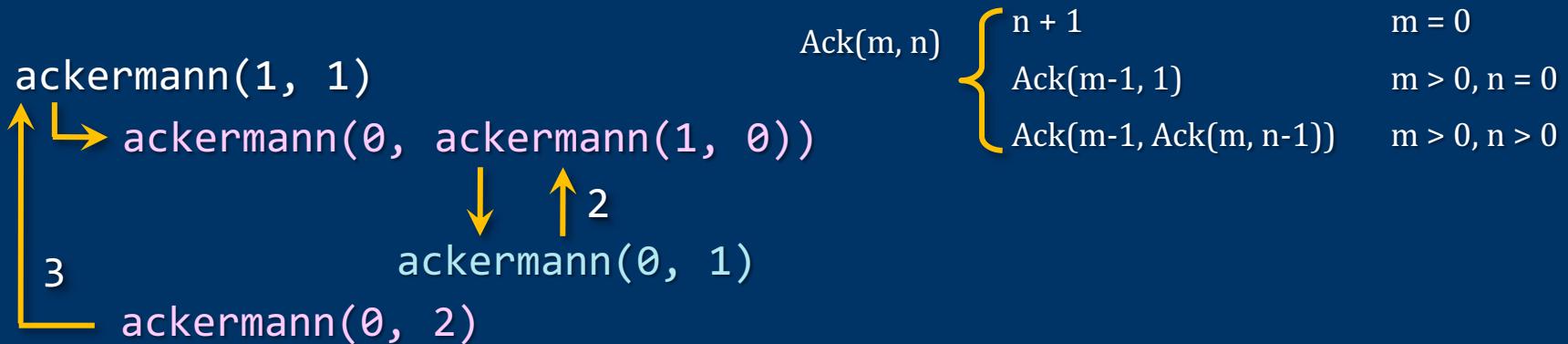
Pruébalo con números muy bajos

Se generan MUCHAS llamadas recursivas

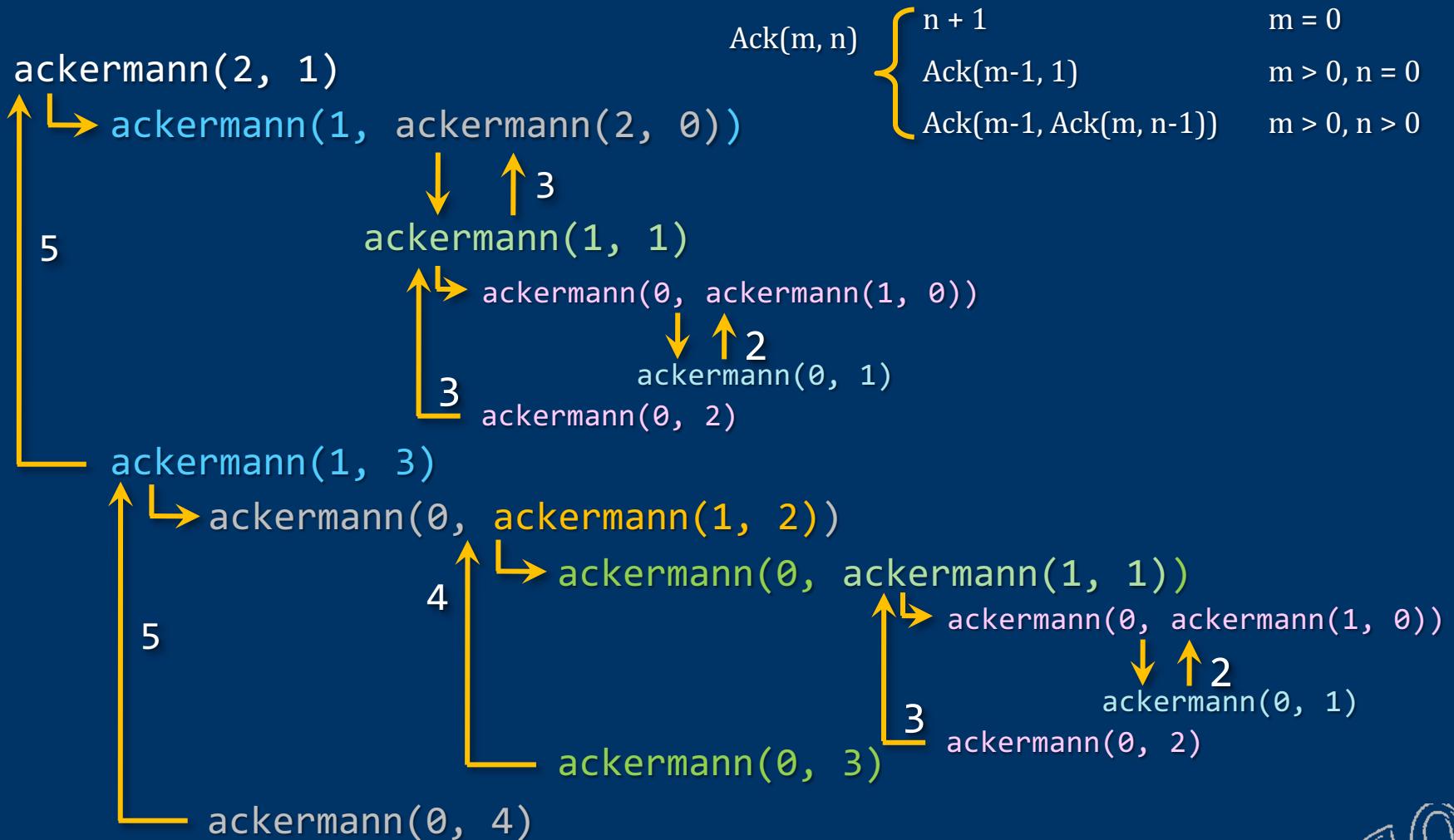
```
int main() {
    int m, n;
    cout << "M = "; cin >> m;
    cout << "N = "; cin >> n;
    cout << ackermann(m, n) << endl;
    return 0;
}
```



Los números de Ackermann



Los números de Ackermann



Tipos de recursión

Recursión cruzada o indirecta: Una función llama a otra y ésta a su vez llama a la primera.

Ejemplo: determinar 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";  
    ... }
```



Tipos de recursión

Código anterior y posterior a la llamada

Recursión simple:

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

- El *código anterior* se ejecuta repetidas veces hasta el caso base.
Se ejecuta en orden directo para las distintas entradas.
- El *código posterior* se ejecuta repetidas veces tras el caso base.
Se ejecuta en orden inverso para las distintas entradas.

Recursión primero: si no hay código anterior

Recursión final: si no hay código posterior



Código anterior y posterior a la llamada

```
void func(int n) {  
    if (n > 0) {  
        cout << "Entrando (" << n << ")" << endl; // Código anterior  
        func(n - 1); // Llamada recursiva  
        cout << "Saliendo (" << n << ")" << endl; // Código posterior  
    }  
}  
  
func(5);
```

El código anterior a la llamada se ejecuta para los sucesivos valores de n

El código posterior a la llamada, al revés

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



Recursión final

Recorrido de una lista (directo)

Procesamos el elemento antes de la llamada recursiva:

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

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

    return 0;
}

void mostrar(const 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
```



Recursión primero

Recorrido de una lista (inverso)

Procesamos el elemento después de la llamada recursiva:

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

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

    return 0;
}

void mostrar(const 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
```



Parámetro acumulador

La recursión final es equiparable a la versión iterativa (bucle): uso de parámetros (denominados acumuladores) para construir el resultado a lo largo de las distintas llamadas

Por valor

```
int factorial(int n, int fact) { // devuelve fact*n!
    if (n == 0) return fact;
    else return factorial(n - 1, n * fac)
}
```



Se realiza la operación que se hacía
posterior a la llamada recursiva

La llamada inicial debe ser: factorial(m, 1);

factorial(2,1) -> factorial(1,2) -> factorial(0,2) -> 2



Parámetro acumulador

Por referencia

```
void factorial(int n, int & fact) { // calcula fact*n!
    if (n > 0) { // parámetro de entrada/salida
        fact = n * fact; ←
        factorial(n - 1, fact);
    }
}
```

Se realiza la operación que se hacía posterior a la llamada recursiva

La llamada inicial debe ser: $r = 1$; `factorial(m, r);`

$\text{factorial}(2, r)$ $\rightarrow \text{factorial}(1, r)$ $\rightarrow \text{factorial}(0, r)$

$r = 2 * r$ $r = 1 * r$



Recursión frente a iteración

¿Qué es preferible?

Siempre hay una alternativa iterativa para un algoritmo recursivo y viceversa.

Algoritmo recursivo: menos eficiente que su alternativa iterativa.
Una recursión final es equiparable a una versión iterativa.

En ocasiones no resulta sencillo obtener una versión iterativa.
Desarrolla una versión iterativa para los números de Fibonacci.
¿Y qué tal una para los números de Ackermann?

