


Lección: recursividad

Tema 1: Introducción

Algorítmica 1

Miguel García Torres

Lenguajes y Sistemas Informáticos, Universidad Pablo de Olavide 



Índice

- 1 Introducción
- 2 Verificación de procedimientos recursivos
- 3 Escritura de programas recursivos
- 4 Funcionamiento de la recursividad
 - Asignación estática de memoria
 - Asignación dinámica de memoria



- La recursividad es una técnica de programación que aparece de forma natural en programación, tanto en la definición de tipos de datos como en el diseño de algoritmos.
- Consiste en que el algoritmo se llama a sí mismo con un problema de menor tamaño y así sucesivamente hasta que la solución es inmediata.
- Algoritmos más simples y compactos que los iterativos pero es más lenta y consumen más recursos.



- **Ámbito de aplicación**
 - General.
 - Problemas que pueden resolverse de forma directa a partir de ciertos tamaños del problema planteado.
- **Razones de uso**
 - Problemas complejos con estructuras iterativas.
 - Soluciones elegantes.
 - Soluciones simples.
- **Condición necesaria: Asignación dinámica de memoria.**



Características básicas

Autoinvocación (caso general) \equiv el proceso se llama a sí mismo.

Caso directo (caso base) \equiv el problema puede resolverse para casos de menor tamaño.

Combinación \equiv los resultados de la llamada precedente son utilizados para obtener una solución, posiblemente combinados con otros datos.



Ejemplo: cálculo del factorial

$$Factorial(n) = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot Factorial(n-1) & \text{si } n > 0. \end{cases}$$

Código iterativo vs recursivo

```
int factorial(int n) {
```



Ejemplo: cálculo del factorial

$$\text{Factorial}(n) = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot \text{Factorial}(n - 1) & \text{si } n > 0. \end{cases}$$

Código iterativo

```
int factorial(int n) {  
    int i, r = 1; {  
        for (i = 0; i < n; i++);  
            r = r · (n - i);  
    }  
    return r;  
}
```



Ejemplo: cálculo del factorial

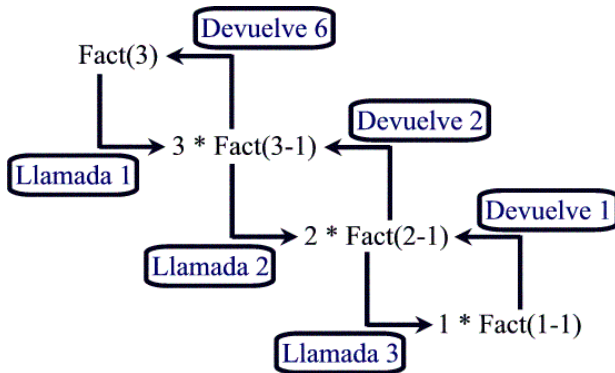
$$\text{Factorial}(n) = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot \text{Factorial}(n - 1) & \text{si } n > 0. \end{cases}$$

Código recursivo

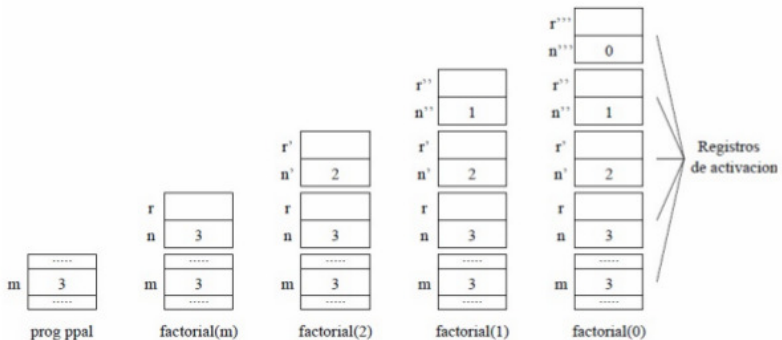
```
int factorial(int n) {  
    int r = 0;  
    if (n == 0) r = 1;  
    else if (n > 0) r = n * factorial(n - 1);  
    return r;  
}
```



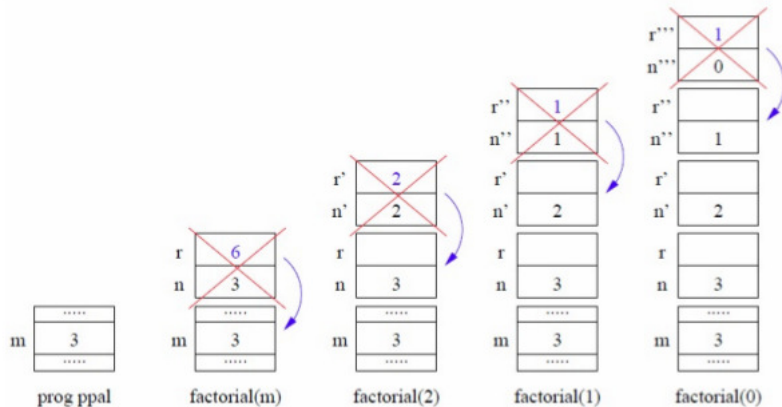
Ejecución del cálculo del factorial



Memoria en el cálculo del factorial



Memoria en el cálculo del factorial



Ejercicio

Diseñar un algoritmo recursivo y otro iterativo que encuentre el término n -ésimo de la siguiente secuencia:

$$a_0 = 1, a_1 = 2$$

$$a_n = a_{n-1} * a_{n-2}$$



Solución: Algoritmo recursivo

```
int a(int n) {  
    int r = 0;  
    if (n >= 0) {  
        r = (n <= 1) ? n+1 : a(n-1) * a(n-2);  
    }  
    return r;  
}
```



Solución: Algoritmo iterativo

```
int a(int n) {  
    int r = 0;  
    if (n >= 0) {  
        if (n <= 1) r = n+1;  
        else {  
            int an = 2, an1 = 2, an2;  
            for (int i = 3; i <= n; i++) {  
                an2 = an1;  
                an1 = an;  
                an = an1 * an2;  
            }  
            r = an;  
        }  
    }  
    return r;  
}
```



Método de las tres preguntas

1 Pregunta Caso-base

- ¿Existe una salida no recursiva o caso base del subalgoritmo?
- ¿El subalgoritmo funciona correctamente?

2 Pregunta Más-pequeño

- ¿Cada llamada recursiva se refiere a un caso más pequeño del problema original?

3 Pregunta Caso-general

- ¿Es correcta la solución en los casos no base?



- 1 Obtención de una definición exacta del problema.
- 2 Determinar el tamaño del problema original a resolver \Rightarrow Parámetros en la llamada inicial.
- 3 Resolver los casos bases o triviales sin recursión.
- 4 Resolver un caso general en términos de un caso más pequeño (llamada recursiva).



Ejemplo

Función de Fibonacci (f)

Calcular el valor de la función de Fibonacci para un número n dado.

- $n \equiv$ Posición de la sucesión a calcular.
- Caso base \equiv Si $n \leq 1 \Rightarrow f(n) = n$
- Caso general \equiv Si $n > 1 \Rightarrow f(n) = f(n - 1) + f(n - 2)$

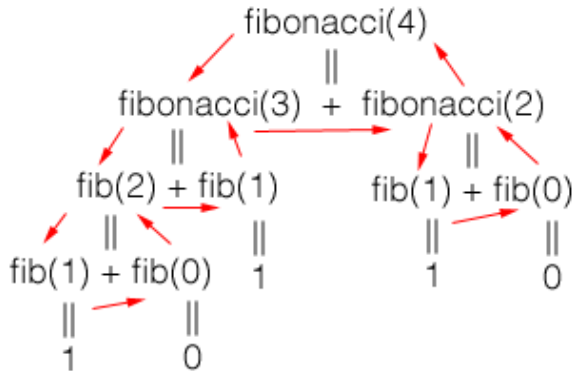


Solución: Función de Fibonacci

código

```
int f(int n) {  
    int res;  
    if (n <= 1) {  
        res = n;  
    } else {  
        res = f(n-1) + f(n-2);  
    }  
    return res;  
}
```





- **Asignación estática de memoria** \equiv cada variable tiene asignada una zona de memoria fija en tiempo de compilación.
- **Asignación dinámica de memoria** \equiv la zona de memoria se determina en tiempo de ejecución.

```
void funcion(int X, int Y) {  
    int Z;  
    ...  
}
```



Índice

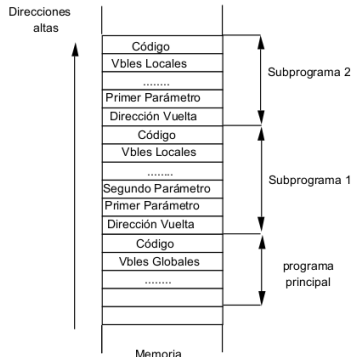
- 1 Introducción
- 2 Verificación de procedimientos recursivos
- 3 Escritura de programas recursivos
- 4 Funcionamiento de la recursividad**
 - **Asignación estática de memoria**
 - Asignación dinámica de memoria



- Se reserva espacio en memoria a partir de una posición **FIJA**, tanto para el código como para los parámetros formales y variables locales de cada subprograma.
- La zona reservada para variables locales y parámetros formales usualmente preceden al código del subprograma
- Asignación memoria
 - $X \rightarrow 0100$
 - $Y \rightarrow 0101$
 - $Z \rightarrow 0111$



Subprogramas con asignación estática de memoria



- ¿Qué conlleva que las variables tengan asignadas posiciones fijas de memoria antes de la ejecución?
- ¿Dónde se almacenan las distintas versiones de estas variables generadas recursivamente?



- Recursividad \Rightarrow los valores intermedios de los parámetros y variables locales deben retenerse.
- La llamada recursiva no puede almacenar sus argumentos en las posiciones fijas que se tenían en tiempo de compilación \Rightarrow Esto haría que los valores de las llamadas recursivas previas se sobrescribieran y se perdieran.



Índice

- 1 Introducción
- 2 Verificación de procedimientos recursivos
- 3 Escritura de programas recursivos
- 4 Funcionamiento de la recursividad**
 - Asignación estática de memoria
 - Asignación dinámica de memoria**



Subprogramas con asignación dinámica de memoria

Las variables se traducen a una posición relativa de memoria respecto a alguna dirección de referencia que llamaremos **Cab**.

Asignación estática

A <---> 0100

B <---> 0101

C <---> 0111

Asignación dinámica

A <---> 0

B <---> 1

C <---> 2

respecto a Cab



- A cada variable y parámetro se le asigna un espacio relativo al Cab.
- Los parámetros son accedidos hacia atrás.
- La posición de Cab variará a medida que se creen variables y parámetros.

```
ALGORITMO uno (X,Y:R)
VARIABLES
    Z : N
INICIO
    .....
FIN
```

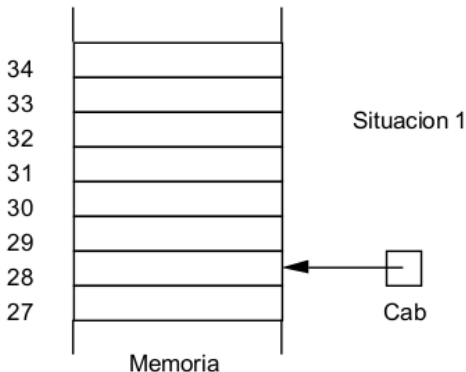
Cuando se traduce (en tiempo de compilación):

dirección de vuelta	<----> 0
X	<----> 1
Y	<----> 2
Z	<----> 3
	Respecto a Cab



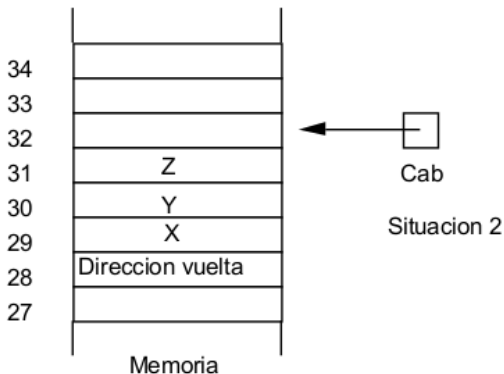
Ejemplo

Tiempo de ejecución: Se reserva espacio para las variables y parámetros a partir de la situación actual de CAB.



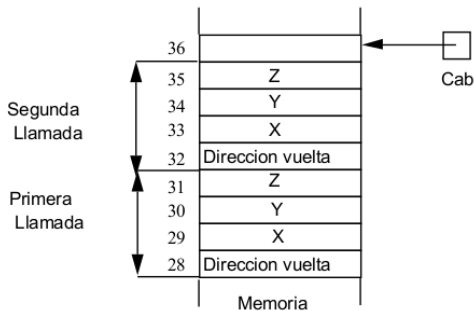
Ejemplo

Llamada a la subrutina.



Ejemplo

Llamada recursiva a la subrutina.



Acceso a la variables por el procedimiento

- $\text{Cab} - 1 \equiv Z$
 - $\text{Cab} - 2 \equiv Y$
 - $\text{Cab} - 3 \equiv X$
 - $\text{Cab} - 4 \equiv \text{Dirección de vuelta}$
-
- Al terminar el procedimiento, se liberan esas posiciones de memoria y se actualiza la variable Cab.
 - Los valores almacenados en cada llamada no se sobrescriben ni se pierden.
 - Se almacena en una estructura de datos llamada Pila (LIFO).
 - El tratamiento interno de la memoria está oculto a la resolución del problema.



Ejemplo: función factorial

- R1 \equiv invocación inicial.
- R2 \equiv invocación recursiva.

ALGORITMO N Factorial (E n:N)

VAR

N fact

INICIO

SI $n == 0$ **ENTONCES** $fact = 1$

SINO $fact = n * \text{Factorial}(n-1)$

FINSI

DEVOLVER $fact$

FIN

La invocación inicial es: **Resultado := Factorial(3)**

R1

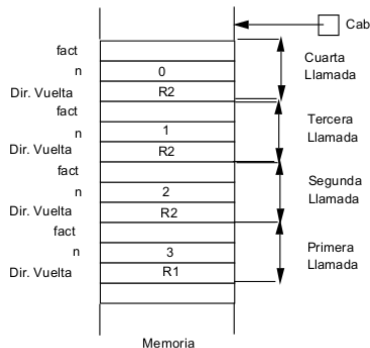
R2



Ejemplo: función factorial

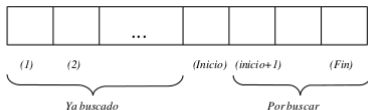
- R1 \equiv invocación inicial.
- R2 \equiv invocación recursiva.

a:= FactorialR(3)



Ejercicio: Búsqueda en un *array*

Buscar un valor dado en un array devolviendo su posición en caso de que esté y -1 en caso contrario.

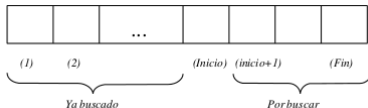


- ¿Parámetros? ¿Llamada a la función?
- ¿Caso base?
- ¿Caso general?



Búsqueda en un *array*

Buscar un valor dado en un array devolviendo verdadero o falso en función de si está o no.



- Invocación: `boolean buscar(vector, x, ini, final)`
- Caso base:
 - `vector[ini] == x \Rightarrow true`
 - `ini == fin && vector[ini] != x \Rightarrow false`
- ¿Caso general?



Solución: Búsqueda recursiva en un *array*

código

```
boolean buscar(int[] v, int x, int ini, int final) {  
    boolean enc;  
    if (v[ini] == x) {  
        enc = true;  
    } else if (ini == fin) {  
        enc = false;  
    } else {  
        enc = buscar(v, x, ini + 1, fin);  
    }  
    return enc;  
}
```

