

Unidad 3 – Eficiencia de Algoritmos

Tema VIII: Recursividad.

Naturaleza. Recursividad directa e indirecta.

Comparación con la iteración. Recursividad infinita. Resolución de problemas complejos con recursividad.

Objetivos

- Definir concepto de recursividad
- Plantear su uso para formular soluciones a problemas que por sus características requieren diseñar algoritmos que deben llamarse a si mismos.
- Presentar una estrategia de implementación de problemas recursivos
- Análisis comparativo entre soluciones iterativas clásicas y las soluciones recursivas

Recursividad

Premisas

- Las definiciones recursivas suelen responder a funciones que se definen en base a un caso menor de sí mismas. Pero la recursividad en programación tiene otras implicaciones.
- Substantial diferencia entre una función matemática y un función programada.
- La recursividad en programación, aunque está permitida en prácticamente todos los lenguajes modernos, no es una herramienta demasiado útil en un entorno productivo.
 - ¿Cuándo debo utilizar entonces recursividad?

Definición

La recursividad es una herramienta que permite expresar la resolución de un problema evolutivos, donde es posible que un modulo de software se invoque a sí mismo en la solución del problema (Garland, 1986) (Helman, 1991) (Aho, 1988).

Recursividad

- Esta técnica puede entenderse como un caso particular de la programación con subprogramas en la que se planteaba la resolución de un problema en términos de otros subproblemas más sencillos.
- El concepto de recursión aparece en varias situaciones de la vida cotidiana.
- Otro tipo de recursión es la referente a los tipos de datos que se definen en función de si mismos.
- La recursión como herramienta de programación permite definir un objeto en términos de si mismo. (*listas circulares*).
- Funciones recursivas en matemática

Aspectos a tener en cuenta en una solución recursiva

Si la guía contiene una sola página

Entonces Buscar secuencialmente la persona dentro de la página

Sino

Abrir la guía a la mitad

Determinar en que mitad está la persona

Si la persona está en la primera mitad

Entonces Buscar en la primera mitad de la guía

Si no

Buscar en la segunda mitad de la guía

Fin

Tres observaciones sobre estos considerandos:

- Una vez dividida la guía, y determinada la parte que contiene a la persona, el método de búsqueda a aplicar sobre esa mitad es el mismo que el empleado para la guía completa.
- La mitad de la guía donde no está el dato se descarta, lo cual significa una reducción del espacio del problema (eficiencia algorítmica).
- Hay un caso especial que se resuelve de una manera diferente que el resto, y sucede cuando la guía queda reducida a una sola página (luego de varias subdivisiones).

Al escribir como procedimiento podemos hacer las siguientes observaciones:

- Una de las acciones del procedimiento es llamarse a si mismo.
- Cada llamada al procedimiento Buscar realizada desde dentro del procedimiento Buscar pasa como parámetro la mitad de la guía, “guía actual”.
- Hay una instancia de Buscar que se resuelve de manera distinta a los demás Este es el caso en que la guía contiene una solo página y se lo denomina caso base.

Recursividad

Resumen de conceptos ya establecidos

- Un programa o subprograma que se llama a si mismo se dice que es recursivo.
- El concepto de recursividad está ligado, en los lenguajes de programación, al concepto de procedimiento o función.
- La recursividad es una de las formas de control más importantes en la programación.
- Los procedimientos recursivos son la forma más natural de representación de muchos algoritmos.
- Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción. La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

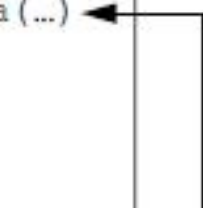
Premisas

1. El procedimiento se llama a si mismo
2. El problema se resuelve, resolviendo el mismo problema pero de tamaño menor
3. La manera en la cual el tamaño del problema disminuye asegura que el caso base se alcanzará.

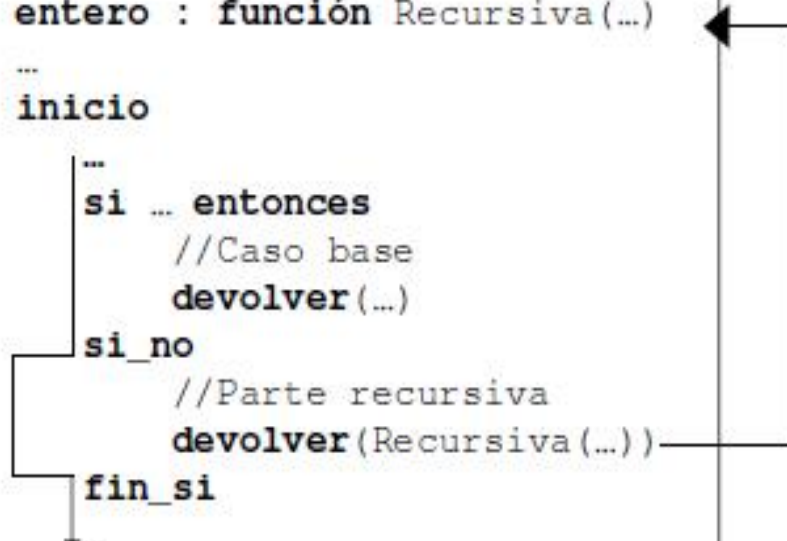
Partes de un algoritmo recursivo

- ❑ Un algoritmo recursivo genera la repetición de una o más instrucciones (como un bucle).
 - Como cualquier bucle puede crear un bucle infinito.
 - Es necesario establecer una condición de salida para terminar la recursividad.
- ❑ Para evitar un bucle infinito, un algoritmo recursivo tendrá:
 - Caso trivial, caso base o fin de recursión.
 - ✓ La función devuelve un valor simple sin utilizar la recursión ($0! = 1$).
 - Parte recursiva o caso general.
 - ✓ Se hacen llamadas recursivas que se van aproximando al caso base.

```
entero : función Recursiva(...)
...
inicio
    ...
    devolver(Recursiva(...))
    ...
fin_función
```



```
entero : función Recursiva(...)
...
inicio
    ...
    si ... entonces
        //Caso base
        devolver(...)
    si_no
        //Parte recursiva
        devolver(Recursiva(...))
    fin_si
    ...
fin_función
```



Recursividad. Caso de estudio

**** Factorial de un número ****

Preguntas

- 1.- Como se puede definir el problema en términos de un problema más simple de la misma naturaleza?
- 2.- Como será disminuido el tamaño del problema e cada llamado recursivo?
- 3.- Que instancia del problema servirá como caso base?

Matemáticamente, el factorial de un número (n) se define de la siguiente manera:

$$\text{Factorial}(n) = n * (n - 1) * (n - 2) * (n - 3) * \dots * 4 * 3 * 2 * 1.$$

$$\text{Factorial}(0) = 1.$$

¿Cómo funciona la recursividad del factorial?



Recursividad. Caso de estudio.

La posibilidad de que la función Fac se llame a si misma existe, porque en Pascal el identificador Fac es valido dentro del bloque de la propia función (acuérdense de los temas de los ámbitos de los bloques en las funciones y los procedimientos).

Al ejecutarlo sobre el argumento 4, se produce la cadena de llamadas sucesivas a Fac (4), Fac (3), Fac (2), Fac (1) y a Fac (0), así:

Fac(4) \leadsto 4 * Fac(3)
 \leadsto 4 * (3 * Fac (2))
 \leadsto 4 * (3 * (2 * Fac(1)))
 \leadsto 4 * (3 * (2 * (1 * Fac(0))))
 \leadsto ...

... \leadsto 4 * (3 * (2 * (1 * 1)))
 \leadsto 4 * (3 * (2 * 1))
 \leadsto 4 * (3 * 2)
 \leadsto 4 * 6
 \leadsto 24

Factorial de un Número en “C”

```
#include<stdio.h>

int factorial(int n)
{
    int r;
    if (n==1)
    {
        return 1;
    }
    r=n*factorial(n-1 ) ;
    return (r ) ;
}

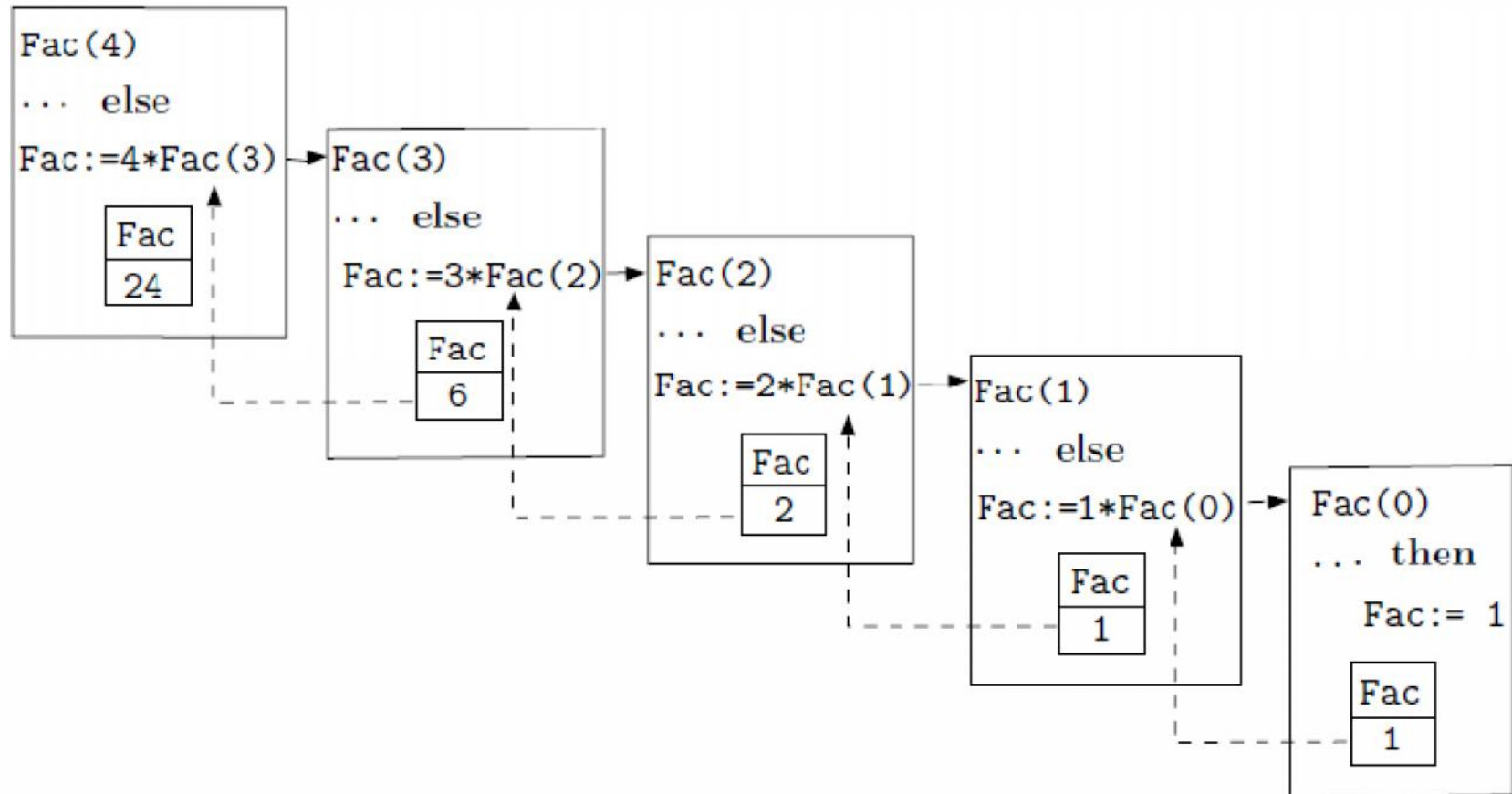
int main()
{
    int n,res;
    printf("*****n" ) ;
    printf("tFactorial Recursivon" ) ;
    printf("*****n" ) ;
    printf("Dame un numero: " ) ;
    scanf("%d",&n ) ;
    res=factorial(n ) ;
    printf("El factorial de %d es: %d",n,res ) ;
    return 0;
}
```

Ejecución de un programa y la pila de activación

Se define “pila de activación” como una estructura de datos que se comporta de la siguiente manera:

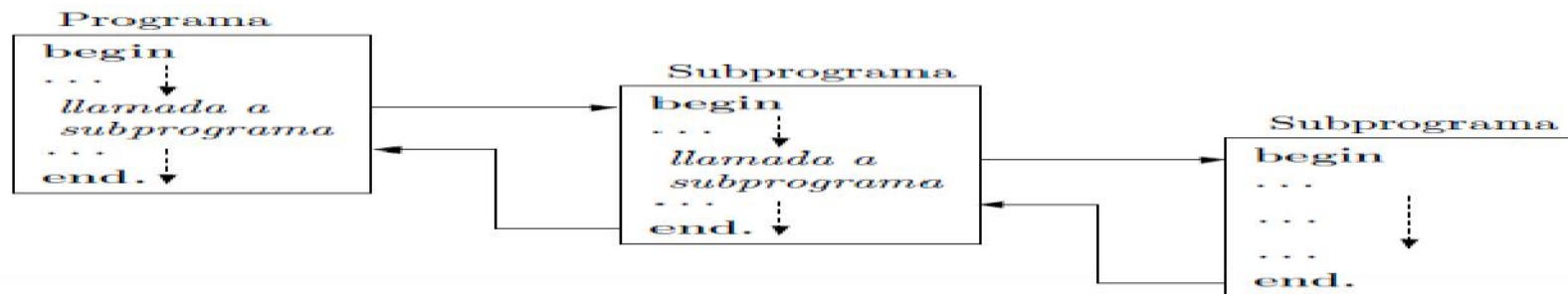
- Tiene un tope (top) y un fondo (bottom)
- Cada elemento es código más dato
- Los elementos pueden sacarse solo por el tope. El único elemento posible de sacar es el que está en el tope

Ejecución de un programa y la pila de activación



Ejecución de un programa y la pila de activación

Concluimos sobre la ejecución de un programa recursivo diciendo que básicamente consiste en una cadena de generación de llamadas (suspendiéndose los restantes cálculos) y reanudación de los mismos al término de la ejecución de las llamadas

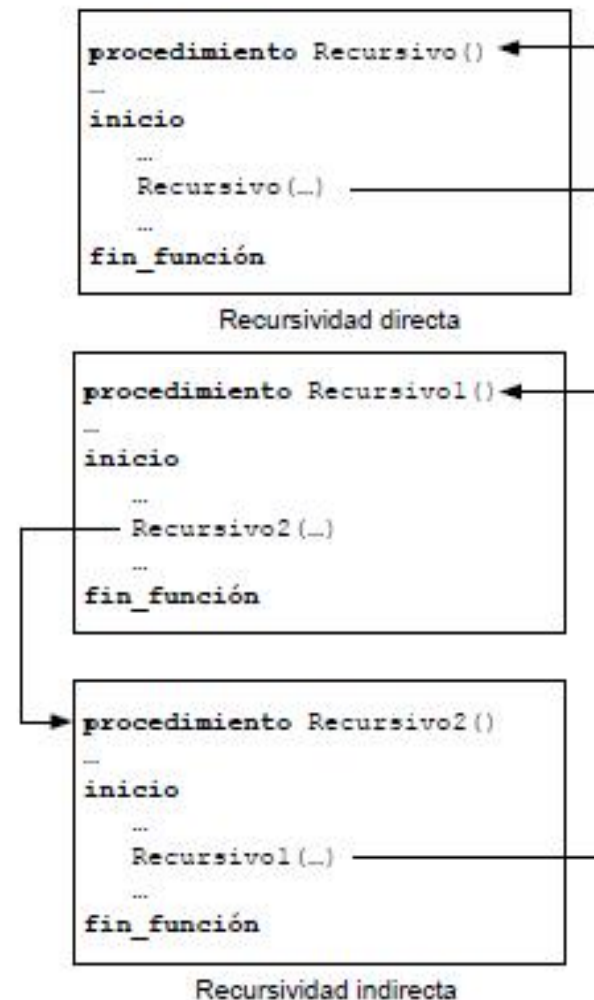


En este proceso se destacan los siguientes detalles:

- El subprograma comienza a ejecutarse normalmente y, al llegar a la llamada, se reserva el espacio para una nueva copia de sus objetos locales y parámetros. Estos datos particulares de cada ejemplar generado se agrupan en la llamada pila de activación del subprograma.
- El nuevo ejemplar del subprograma pasa a ejecutarse sobre su pila de activación, que se apila sobre las llamadas recursivas anteriores formando la llama pila recursiva.
- Este proceso termina cuando un ejemplar no genera mas llamadas recursivas por consistir sus argumentos en casos básicos. Entonces se libera el espacio reservado para la pila de activación de ese ejemplar, reanudándose las instrucciones del subprograma anterior sobre la tabla penúltima.
- Este proceso de retorno finaliza con la llamada inicial.

Tipos de recursividad

- ❑ Según el subprograma al que se llama, existen dos tipos de recursión:
 - Recursividad simple o directa.
 - ✓ La función incluye una referencia explícita a si misma.
`devolver (recursiva (...))`
 - Recursividad mutua o indirecta.
 - ✓ El módulo llama a otros módulos de forma anidada y en la última llamada se llama al primero.



Tipos de recursividad (II)

- ❑ Según el modo en que se hace la llamada recursiva la recursividad puede ser:
 - De cabeza.
 - ✓ La llamada se hace al principio del subprograma, de forma que el resto de instrucciones se realizan después de todas las llamadas recursivas.
 - Las instrucciones se hacen en orden inverso a las llamadas.
 - De cola.
 - ✓ La llamada se hace al final del subprograma, de forma que el resto de instrucciones se realizan antes de hacer la llamada.
 - Las instrucciones se hacen en el mismo orden que las llamadas.
 - Intermedia.
 - ✓ Las instrucciones aparecen tanto antes como después de las llamadas.
 - Múltiple.
 - ✓ Se producen varias llamadas recursivas en distintos puntos del subprograma.
 - Anidada.
 - ✓ La recursión se produce en un parámetro de la propia llamada recursiva.
 - ✓ La llamada recursiva utiliza un parámetro que es resultado de una llamada recursiva.

Tipos de recursividad (III)

```
procedimiento f(valor entero: n)
...
inicio
  si n>0 entonces
    f(n-1)
  fin_si
  instrucción A
  instrucción B
fin_procedimiento
```

Recursividad de cabeza

```
procedimiento f(valor entero: n)
...
inicio
  instrucción A
  instrucción B
  si n>0 entonces
    f(n-1)
  fin_si
fin_procedimiento
```

Recursividad de cola

```
procedimiento f(valor entero: n)
...
inicio
  instrucción A
  si n>0 entonces
    f(n-1)
  fin_si
  instrucción B
fin_procedimiento
```

Recursividad de intermedia

```
procedimiento f(valor entero: n)
...
inicio
  ...
  si n>0 entonces
    f(n-1)
  fin_si
  si n<5 entonces
    f(n-3)
  fin_si
  ...
fin_procedimiento
```

Recursividad múltiple

```
entero función f(valor entero: n)
...
inicio
  ...
  si n>0 entonces
    devolver(f(n-1)+f(n-2))
  fin_si
fin_función
```

Recursividad anidada

```
#include<stdio.h>
```

```
void rechazar(char varres);
```

```
void leer(char varres);
```

```
int main( void ){
```

```
    leer(resp);
```

```
    return 0;
```

```
}
```

```
void rechazar(char varres){
```

```
    printf ("no es una respuesta aceptable %d: ", res);
```

```
    Leer (resp);
```

```
}
```

```
void leer(char varres){
```

```
    printf("S para si o N para No");
```

```
    scanf ("%c", &resp);
```

```
    If ( resp != 'S') && (resp != 'N') {
```

```
        rechazar (resp);
```

```
    }
```

```
}
```

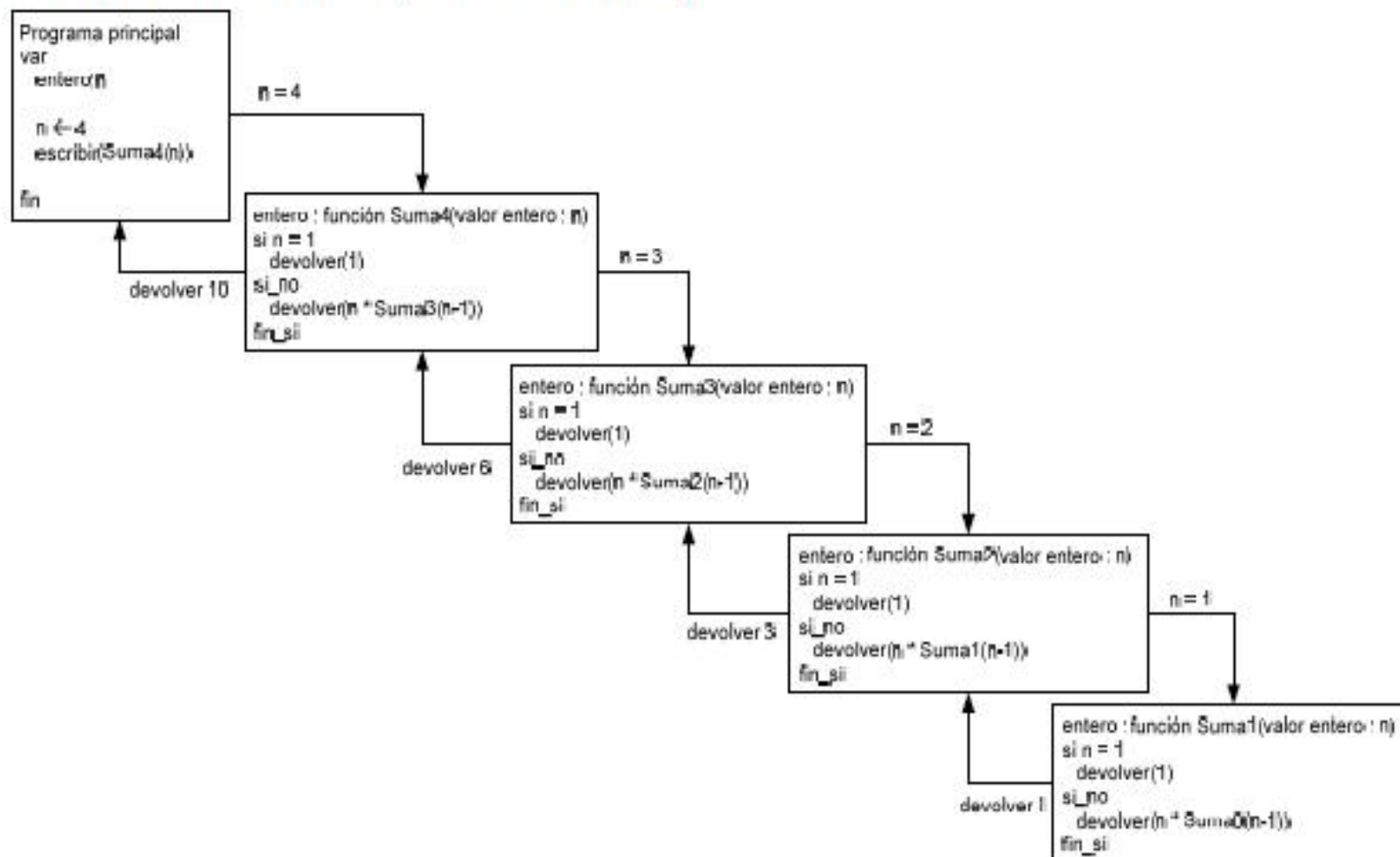
Recursividad Infinita

Recursividad Infinita

- Es muy importante que toda función recursiva tenga un caso en el que no se llame a sí misma, o las llamadas serían infinitas y el programa no tendría fin.
- Por eso, siempre una función recursiva tiene una condición inicial en la que no debe llamarse a sí misma.

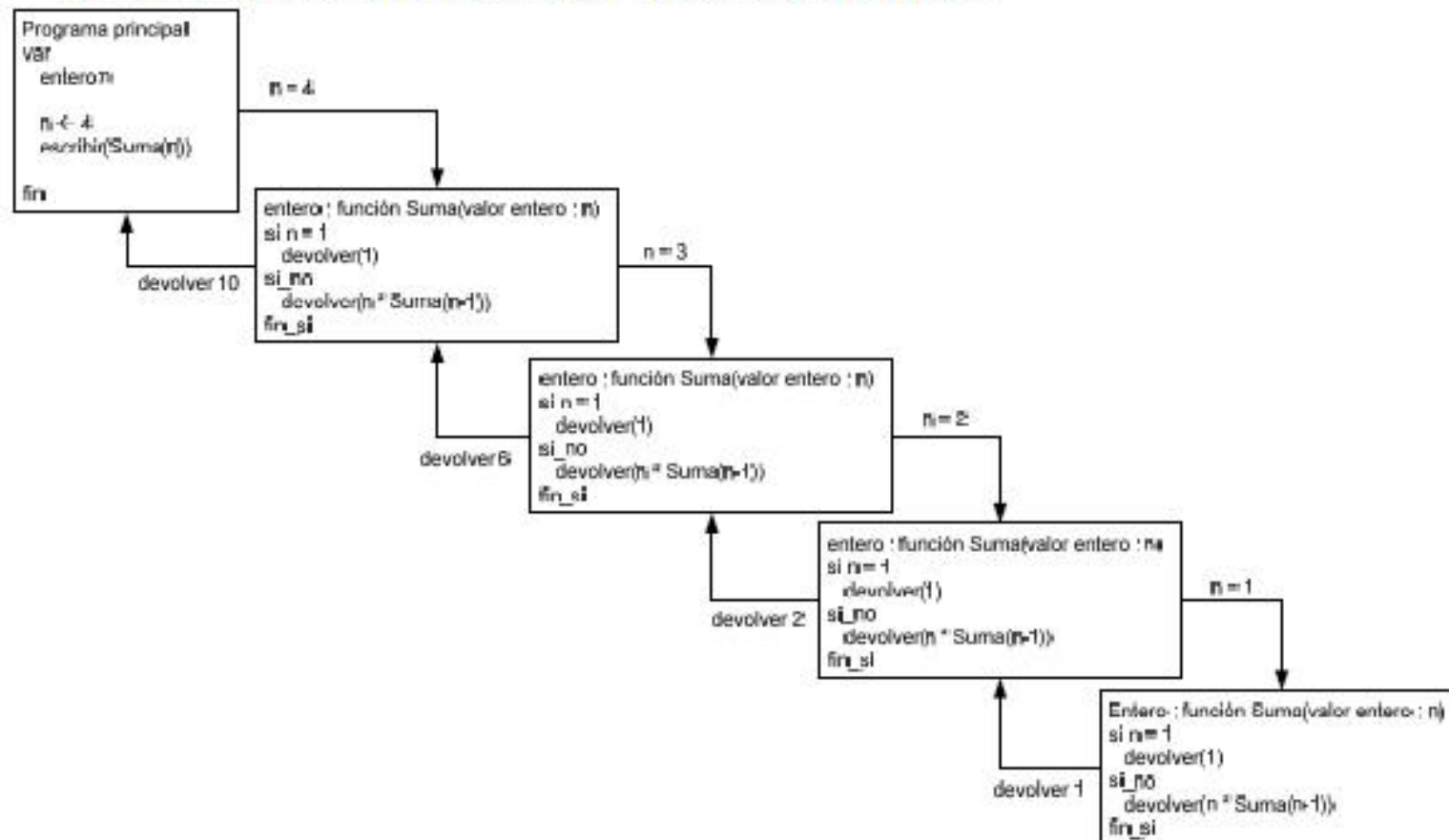
Llamadas a módulos recursivos

□ Llamadas anidadas (no recursivas).



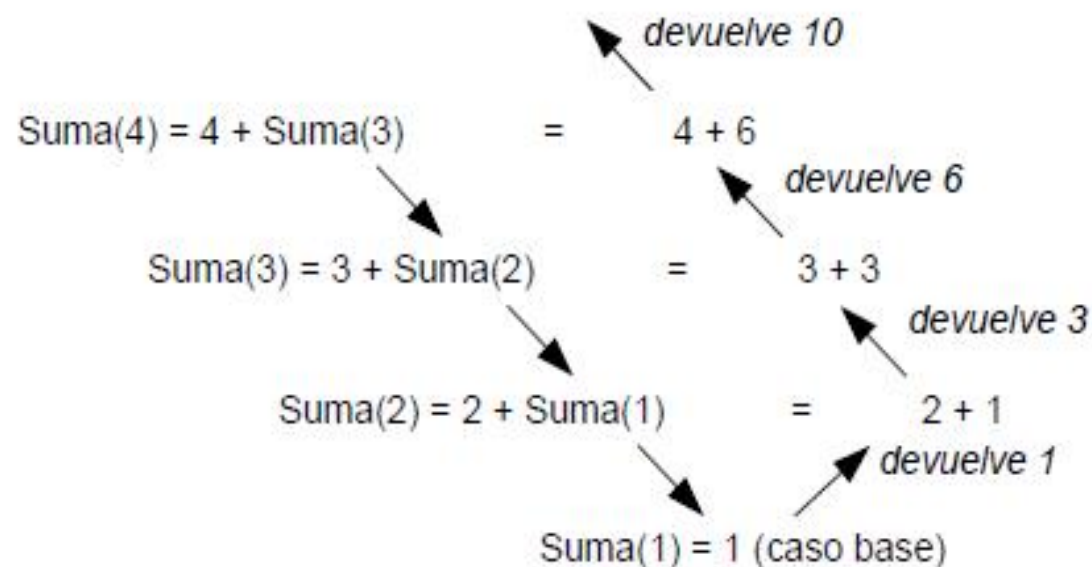
Llamadas a módulos recursivos (II)

- El funcionamiento sería el mismo si se tratara de una única función que se llamara a sí misma de forma recursiva.



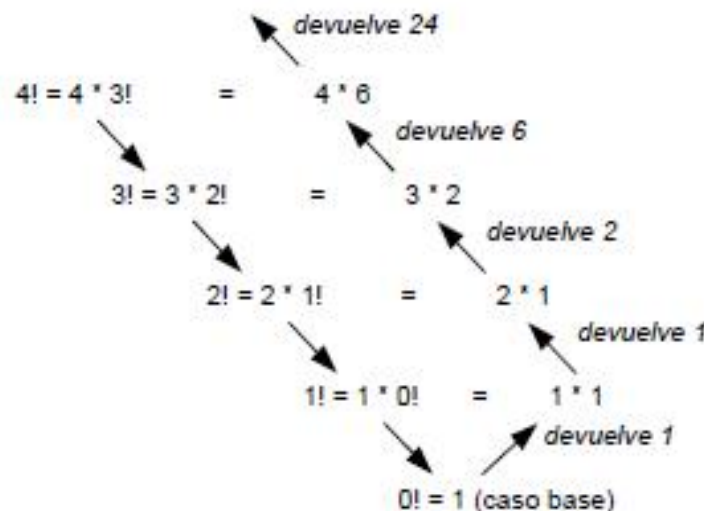
Llamadas a módulos recursivos (III)

- El problema de sumar números entre 1 y n se puede definir en función de su tamaño (n), el problema se puede dividir en partes más pequeñas del mismo problema y se conoce la solución del caso más simple (caso base, $\text{suma}(1) = 1$). Por inducción se puede suponer que las llamadas más pequeñas (por ejemplo, $\text{Suma}(n-1)$) quedan resueltas.



Llamadas a módulos recursivos (IV)

- ❑ En el caso del factorial...
 - Por definición, $0! = 1$,
 - Para cualquier número entero mayor que 0, $n! = n * (n-1)!$
- ❑ En este problema:
 - La solución de $n!$ puede ser definida en función de su tamaño (n).
 - Se puede dividir en instancias más pequeñas ($<n$) del mismo problema.
 - Se conoce la solución de las instancias más simples ($n=0$).
 - Por inducción, las llamadas más pequeñas ($<n$) pueden quedar resueltas.
 - ✓ Sabemos que $4! = 4 * 3!$
- ❑ Conclusión: Se puede resolver por recursividad.



Llamadas a módulos recursivos (IV)

- ❑ Cada llamada a la función factorial devolverá el valor del factorial que se pasa como argumento.

```
entero función Factorial(valor entero : n)
inicio
    //Para cualquier n entero positivo  $\leq 1$ ,  $n! = 1$ 
    //n  $\leq 1$  sería el caso base, caso trivial o fin de la recursión
    si n < 1 entonces
        devolver(1)
    si_no
        //En caso contrario  $n! = n * (n-1)!$ 
        //En cada llamada n se acerca al caso base
        devolver(n * Factorial(n-1))
    fin_si
fin_función
```

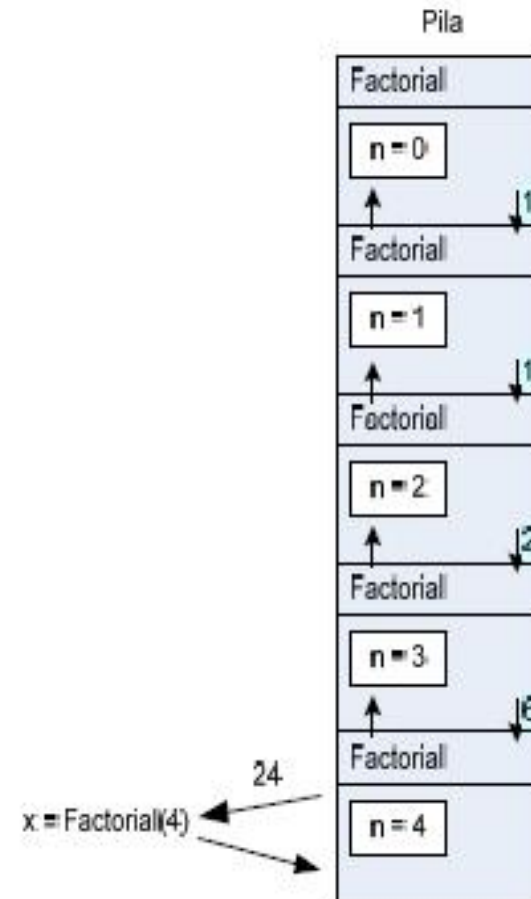
Llamadas a módulos recursivos (V)

□ La pila de llamadas a subrutinas.

- Una estructura de tipo *pila* es una estructura de datos en la que la información se añade y se elimina por un extremo llamado cima.
 - ✓ El último elemento que entra en la pila es el primero que sale.
- El *registro de activación de procedimientos* es un bloque de memoria que contiene información sobre las constantes, variables locales y parámetros que se pasan al procedimiento, junto con la información de la dirección de retorno de esa llamada.
- Cada vez que se llama a un procedimiento (sea o no una llamada anidada, sea o no una llamada recursiva) se almacena el registro de activación del procedimiento en la pila de llamadas a subrutinas.
 - ✓ Cuando el procedimiento termina, su registro de activación se desapila, volviendo a la dirección de retorno que se ha almacenado y recuperando el estado de las constantes, variables locales y parámetros.

Llamadas a módulos recursivos (VI)

- ❑ En cada llamada a la función factorial carga en la pila de llamadas su registro de activación (cómo en cualquier llamada a una subrutina).
 - El último registro de activación que entra es el primero que sale.
 - Aunque los identificadores sean los mismos no existe ambigüedad:
 - ✓ Siempre se refieren al ámbito en el que han sido declarados.
- ❑ Cuando se llega al caso base ($n < 1$), el registro de activación se desapila y el flujo de control del programa regresa a la última llamada que se ha hecho (cuando $n=1$), devolviendo además el valor de retorno (1).
- ❑ Los registros de activación se van desapilando, restaurando los valores anteriores de n y devolviendo los valores de retorno de cada una de las llamadas recursivas a la función (1, 1, 2, 6, 24).

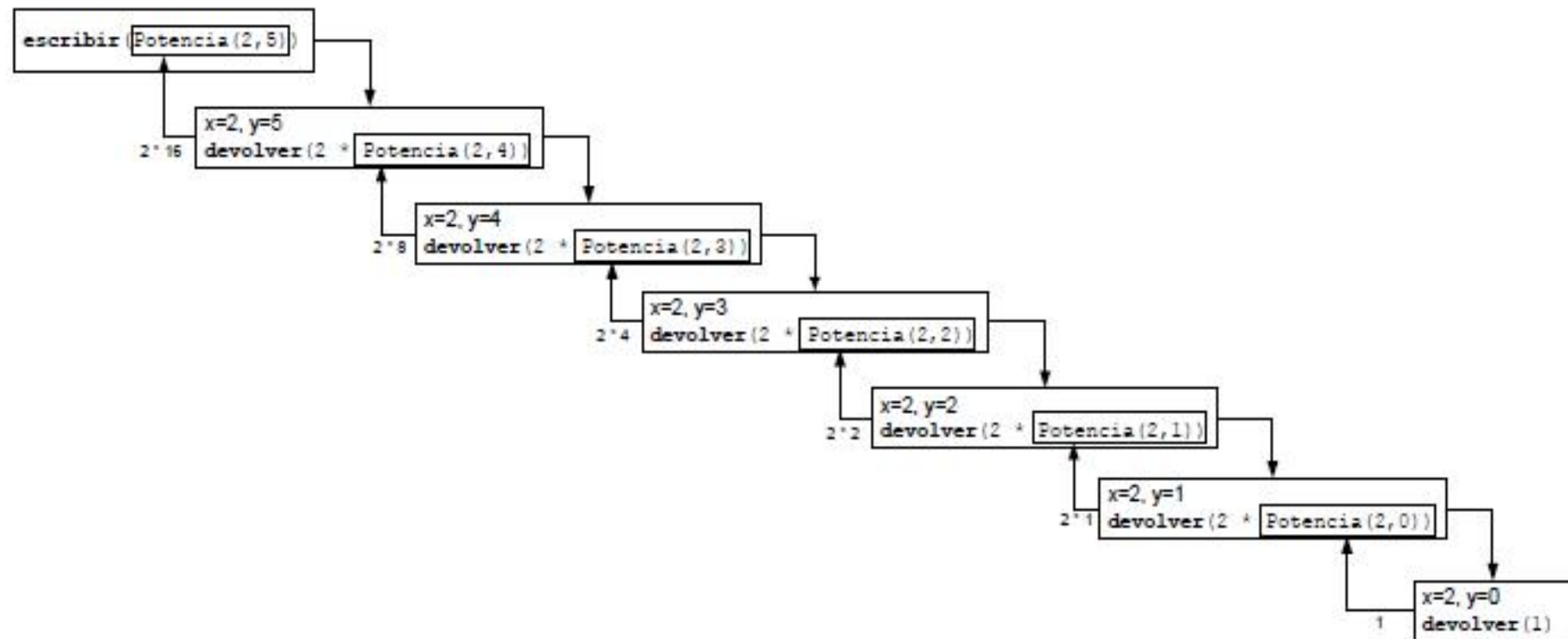


Algoritmos recursivos

- ❑ Cualquier algoritmo iterativo puede resolverse recursivamente.
 - Una llamada recursiva, genera un bucle con una condición de salida cuando se llega al caso base: se ejecuta la llamada hasta que se cumple la condición de salida, como un bucle.
- ❑ También, cualquier algoritmo recursivo puede resolverse de forma iterativa.
- ❑ Potencia.

```
entero función Potencia(valor entero : x,y)
inicio
  si y = 0 entonces
    devolver(1)
  si_no
    devolver(x * Potencia(x,y-1))
  fin_si
fin_función
```

Algoritmos recursivos (II)



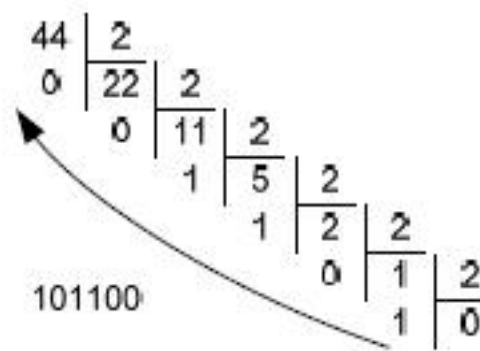
Algoritmos recursivos (III)

- ❑ Escribir un número decimal en binario.
 - Caso base: si $n < 2$ n en binario es n .
 - Si $n \geq 2$, n en binario es la división entera de n entre 2 en binario seguido del resto de dividir n entre 2.
 - ✓ 2 en binario es 2 **div** 2 en binario (1), seguido de 2 **mod** 2 (0) \rightarrow 10.
 - ✓ 3 en binario es 3 **div** 2 en binario (1), seguido de 3 **mod** 2 (1) \rightarrow 11.

```

procedimiento EscribirEnBinario(valor entero : n)
inicio
    si n < 2 entonces
        //Escribe n sin hacer un salto de línea
        escribir(n)
    si_no
        EscribirEnBinario(n div 2)
        //Escribe n mod 2 sin hacer salto de línea
        escribir(n mod 2)
    fin_si
fin procedimiento

```

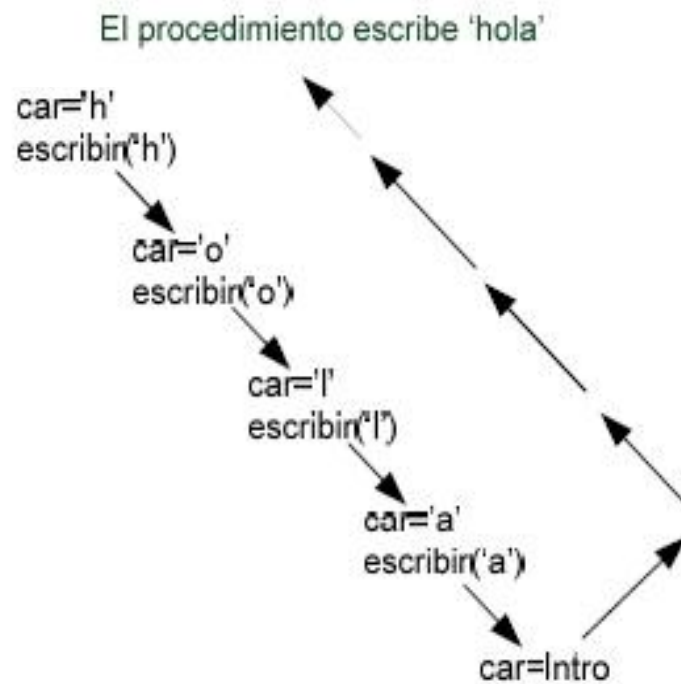


Algoritmos recursivos (IV)

- ❑ Las acciones que se realicen antes de la llamada recursiva se ejecutarán en el mismo orden que la llamada.
 - Lo que ocurre en la recursión "de cabeza".
- ❑ Leer y escribir caracteres hasta que se pulsa Intro.

```
procedimiento LeerCaracteres()  
var  
    carácter : car  
inicio  
    leer(car)  
    si código(car) <> 13  
        escribir(car)  
        LeerCaracteres()  
    fin_si  
fin_procedimiento
```

- ❑ La secuencia leída de caracteres es h,o,l,a,Intro.

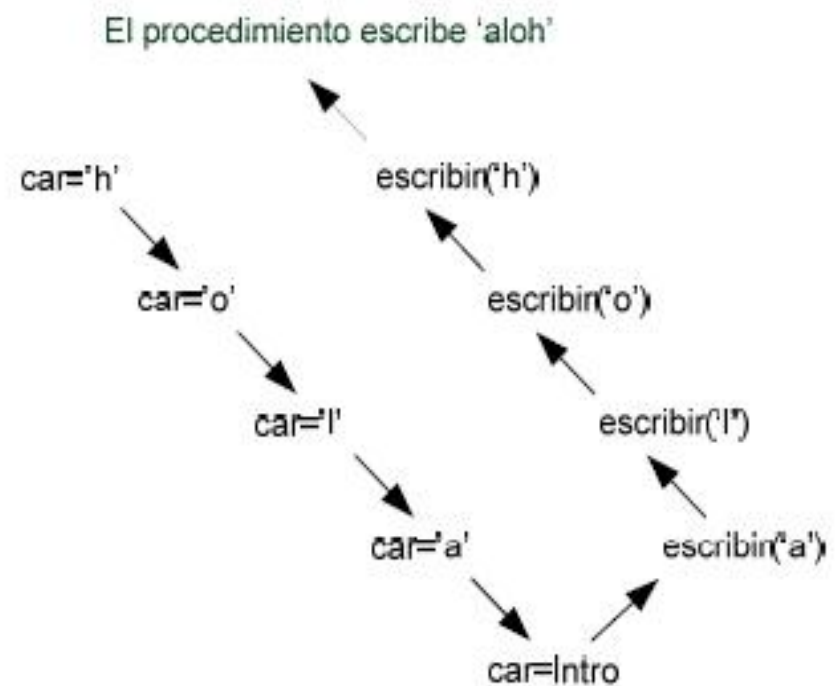


Algoritmos recursivos (IV)

- ❑ Las acciones que se realicen después de la llamada recursiva se ejecutarán en orden inverso a las llamadas.
 - Lo que ocurre en la recursión "de cola".
- ❑ Leer y escribir caracteres hasta que se pulsa Intro.

```
procedimiento LeerCaracteres()  
var  
    carácter : car  
inicio  
    leer(car)  
    si código(car) <> 13  
        LeerCaracteres()  
        escribir(car)  
    fin_si  
fin_procedimiento
```

- ❑ La secuencia leída de caracteres es h,o,l,a,Intro.

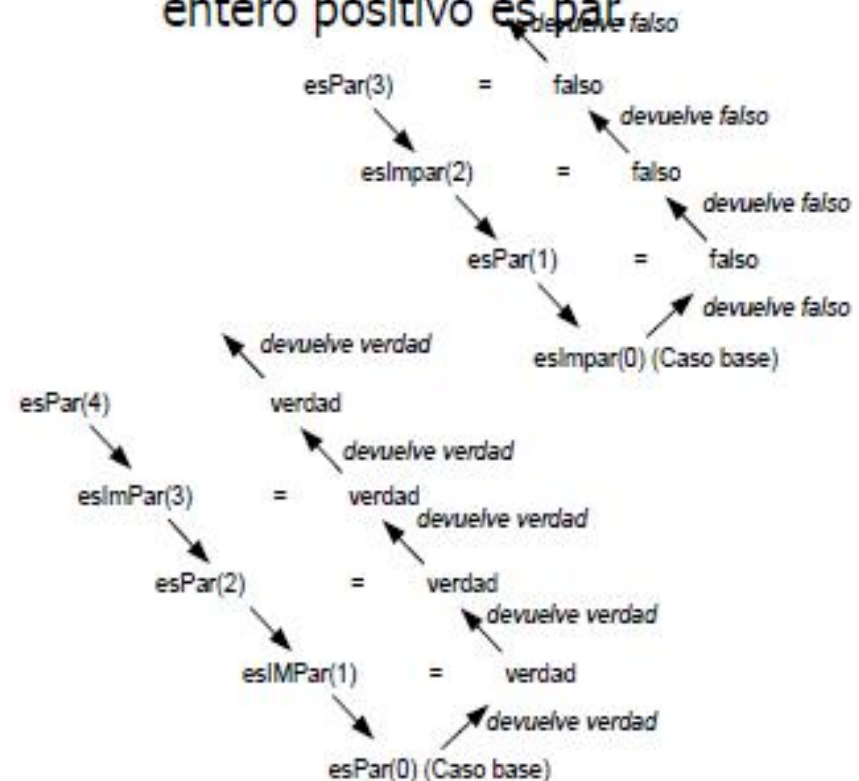


Algoritmos recursivos (V)

```
lógico función esPar(valor entero:n)
inicio
  si n = 0 entonces
    devolver(verdad)
  si_no
    devolver(esImpar(n-1))
  fin_si
fin_función
lógico función esImPar(valor entero:n)
inicio
  si n = 0 entonces
    devolver(falso)
  si_no
    devolver(esPar(n-1))
  fin_si
fin_función
//Ejemplo de llamada
si esPar(5) entonces
  ...
```

- Ejemplo de recursividad mutua o indirecta.

- Determinar si un número entero positivo es par.



Corrección de subprogramas recursivos

Un subprograma recursivo no es más que un caso particular de subprograma en el que aparecen llamadas a sí mismo. Esta peculiaridad hace que tengamos que recurrir a alguna herramienta matemática, de aplicación no demasiado complicada en la mayoría de los casos.

El proceso de análisis de la corrección de subprogramas recursivos puede ser dividido, en dos partes:

- una primera, en la que consideraremos los pasos de la verificación comunes con los subprogramas no recursivos;
- y una segunda con los pasos en los que se aplican técnicas específicas de verificación de la recursión.

Corrección de subprogramas recursivos

En resumen, para demostrar la corrección de un subprograma recursivo hemos de comprobar:

- La corrección del caso base.
- La corrección de los casos recurrentes. Para ello, se supone la de las llamadas subsidiarias, como ocurre en el paso inductivo con la hipótesis de inducción.
- Que las llamadas recursivas se hacen de manera que los parámetros se acercan al caso base; por ejemplo, en el cálculo del factorial, en las sucesivas llamadas los parámetros son n ; $n - 1$; ..., que desembocan en el caso base 0, siempre que $n > 0$, lo cual se exige en la condición previa de la función.

Ventajas e inconvenientes

❑ Inconvenientes.

- Mayor uso de la pila de memoria.
 - ✓ Cada llamada recursiva implica una nueva entrada en la pila de llamadas dónde se cargará tanto la dirección de retorno como todos los datos locales y argumentos pasados por valor.
 - ✓ El tamaño que reserva el compilador a la pila de llamadas es limitado y puede agotarse, generándose un error en tiempo de compilación.
 - Mayor tiempo en las llamadas.
 - ✓ Cada llamada a un subprograma implica:
 - Cargar en memoria el código del procedimiento.
 - Meter en la pila la dirección de retorno y una copia de los parámetros pasados por valor.
 - Reservar espacio para los datos locales.
 - Desviar el flujo del programa al subprograma, ejecutarlo y retornar al programa llamador.
 - ✓ Esto implica una mayor tiempo de ejecución, sobre todo si hay muchas llamadas anidadas, algo normal en programas recursivos.
 - Estos dos inconvenientes se pueden agravar si se pasan estructuras de datos muy grandes por valor, ya que implica copiar estos datos en la pila de procedimientos.
 - Conclusión: un programa recursivo puede ser menos eficiente que uno iterativo en cuanto a uso de memoria y velocidad de ejecución.
-

Ventajas e inconvenientes (II)

❑ Ventajas.

- Mayor simplicidad del código en problemas recursivos.
 - ✓ Si un problema se puede definir fácilmente de forma recursiva (por ejemplo, el factorial o la potencia) es código resultante puede ser más simple que el equivalente iterativo.
 - También es muy útil para trabajar con estructuras de datos que se pueden definir de forma recursiva, como los árboles.
- Posibilidad de "marcha atrás": *backtracking*.
 - ✓ Las características de la pila de llamadas hacen posible recuperar los datos en orden inverso a como salen, posibilitando cualquier tipo de algoritmo que precise volver hacia atrás.

❑ En resumen:

- Es apropiada cuando el problema a resolver o los datos que maneja se pueden definir de forma recursiva o cuando se debe utilizar *backtracking*.
 - No es apropiada si la definición recursiva requiere llamadas múltiples.
 - Ejemplos:
 - ✓ Puede ser apropiada en el ejemplo de transformar un número en binario (requiere coger los restos en orden inverso).
 - ✓ En los casos del factorial y la potencia, aunque el problema se puede definir fácilmente de forma recursiva, la solución iterativa no es demasiado compleja, por lo que sería prácticamente indiferente utilizar la solución iterativa o recursiva.
 - ✓ En algunos casos es claramente perjudicial, como en el problema de la serie de Fibonacci que se aparece a continuación.
-

Ventajas e inconvenientes (III)

❑ Serie de Fibonacci.

- Sucesión de números naturales en la que a partir del término 2 de la serie, cada número (número de Fibonacci) es la suma de los dos anteriores:
✓ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
- El problema de calcular el número n de la serie se puede resolver de forma iterativa.

```
entero función Fibonacci(valor entero: n)
var
    entero: término, último, penúltimo
inicio
    último ← 1
    penúltimo ← 0
    desde i ← 2 hasta n hacer
        término ← penúltimo + último
        penúltimo ← último
        último ← término
    fin_desde
    devolver(último)
fin_función
```

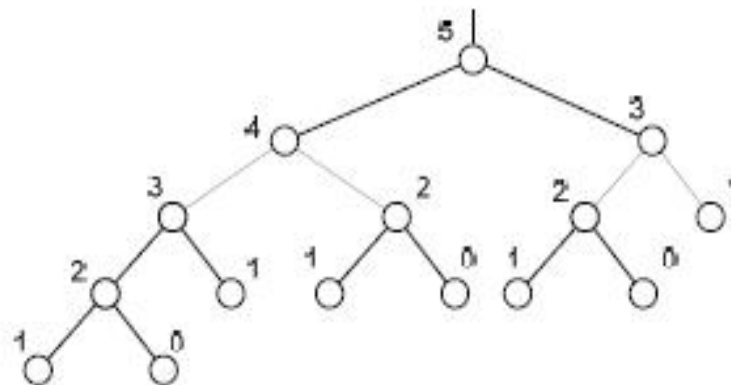
Ventajas e inconvenientes (IV)

- ❑ El problema del cálculo del número n de la serie de Fibonacci también admite una definición recursiva.
 - Si $n = 0$, $Fib_n = 0$
 - Si $n = 1$, $Fib_n = 1$
 - Si $n > 1$, $Fib_n = Fib_{n-1} + Fib_{n-2}$.

```
entero función Fibonacci(valor entero : n)
inicio
  si (n <= 1) entonces
    devolver(n)
  si_no
    devolver(Fibonacci(n-1) + Fibonacci(n-2))
  fin_si
fin_función
```


Ventajas e inconvenientes (VI)

- ❑ Para la llamada Fibonacci(5) el número de llamadas a realizar sería de 15.



- ❑ Un algoritmo iterativo con variables locales que almacenaran los cálculos parciales sería más eficiente.
 - Sólo necesitaría n iteraciones para calcular Fibonacci(n)

Resolución de problemas recursivos

- ❑ Para hallar la solución recursiva a un problema podemos hacernos tres preguntas:
 1. ¿Cómo se puede definir el problema en términos de uno o más problemas más pequeños del mismo tipo que el original?
 2. ¿Qué instancias del problema harán de caso base?
 3. Conforme el problema se reduce de tamaño ¿se alcanzará el caso base?
 4. ¿Cómo se usa la solución del caso base para construir una solución correcta al problema original?
- ❑ Para el problema de Fibonacci, las respuestas serían:
 1. $\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$.
 2. $\text{Fibonacci}(0) = 0$ y $\text{Fibonacci}(1) = 1$.
 3. En cada llamada se reduce el tamaño del problema en 1 o 2, por lo que siempre se llegará a algunos de los casos base.
 4. $\text{Fibonacci}(2) = \text{Fibonacci}(1) + \text{Fibonacci}(0) = 1 + 0$, se construye la solución del problema $\text{Fibonacci}(3)$ a partir de los dos casos base.

Resolución de problemas recursivos (II)

□ Dos estrategias de resolución de problemas recursivos:

- *Divide y vencerás.*
 - ✓ Divide el problema de tamaño n en problemas más pequeños cada uno de los cuales es similar al original pero de menor tamaño.
 - Si se llega a una solución de los subproblemas, se podrá construir de forma sencilla una solución al problema general.
 - Cada uno de esos subproblemas se podrá resolver de forma directa (caso base) o dividiéndolos en problemas más pequeños mediante la recursividad.
 - Los ejemplos que se han hecho utilizan la estrategia de divide y vencerás.
- *Backtracking.*
 - ✓ Divide la solución en pasos, en cada uno de los cuales hay una serie de opciones que ha que probar de forma sistemática.
 - ✓ En cada paso se busca una posibilidad o solución o solución aceptable.
 - Si se encuentra se pasa a decidir el paso siguiente.
 - Si no se encuentra una solución aceptable, se retrocede hasta la última solución aceptable encontrada y se elige una opción distinta a la anterior.
 - ✓ La recursividad se utiliza para poder retroceder hasta encontrar una solución aceptable.
 - ✓ Ejemplos:
 - Juegos de tablero, laberintos,...

Resolución de problemas recursivos (III)

□ Búsqueda binaria recursiva.

- Se trata de un algoritmo que sigue la estrategia de divide y vencerás.
- 1. ¿Cómo se puede definir el problema en términos de uno o más problemas más pequeños del mismo tipo que el original?
 - ✓ La búsqueda binaria en una lista de n elementos, se realiza de la misma forma que la búsqueda binaria entre los elementos de la izquierda, si el elemento es menor que el central, o de la derecha, si el elemento es mayor que el central.
- 2. ¿Qué instancias del problema harán de caso base?
 - ✓ Cuando se encuentra el elemento (el elemento es igual que el elemento central de la lista).
 - ✓ Cuando no se encuentra el elemento (el número de elementos de la lista es 0).
- 3. Conforme el problema se reduce de tamaño ¿se alcanzará el caso base?
 - ✓ Cada paso se va reduciendo la lista, al elegir la parte izquierda o la parte derecha. Llegará un momento en que se pueda llegar a buscar el elemento en una lista de un único componente.
- 4. ¿Cómo se usa la solución del caso base para construir una solución correcta al problema original?
 - ✓ En una lista de un único componente, si el elemento central es el que buscamos, ya se ha encontrado la lista, en caso contrario no está.

Resolución de problemas recursivos (VI)

❑ Casos base:

- Si el elemento central es el buscado, el elemento está en la posición central.
- Si el elemento central no es el buscado y la lista tiene sólo un elemento, el elemento no está.

❑ Casos generales:

- Si el elemento central es mayor que el buscado, se realiza una búsqueda binaria en la parte izquierda de la lista.
 - Si el elemento central es menor que el buscado se realiza una búsqueda binaria en la parte derecha de la lista.
-

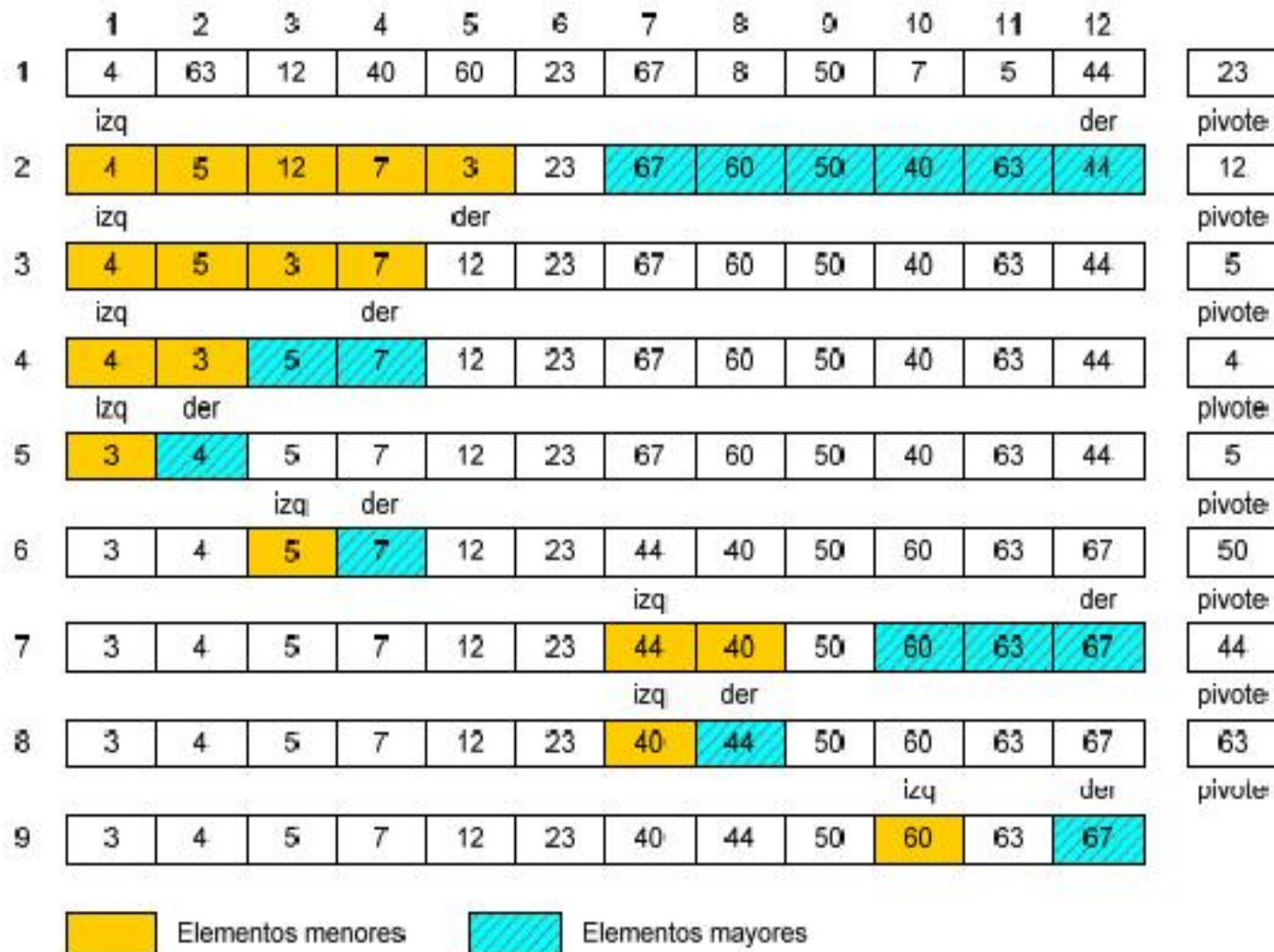
Resolución de problemas recursivos (V)

```
entero función Buscar(valor vector: v; valor entero: izq, der; valor tipoDato: clave)
var
    entero : cen
inicio
    si izq > der entonces
        //No hay lista, el elemento no está
        devolver(0)
    si_no
        cen ← (izq + der) div 2
        si v[cen] = clave entonces
            //El elemento está en la posición cen
            devolver(cen)
        si_no
            si v[cen] > clave entonces
                //El elemento se buscará en la parte izquierda de la lista
                //Es decir entre izq y el elemento central
                devolver(Buscar(v, izq, cen-1, clave))
            si_no
                //El elemento se buscará en la parte derecha de la lista
                //Es decir entre la parte central y derecha
                devolver(Buscar(v, cen+1, der, clave))
        fin_si
    fin_si
fin_si
fin_función
```

Quicksort

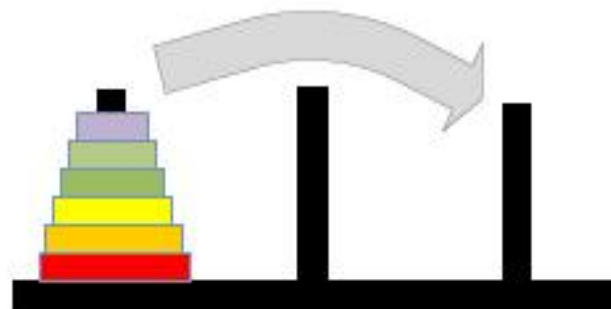
- ❑ También sigue una estrategia de divide y vencerás.
- ❑ Se realiza una partición de una lista en dos partes, dejando los elementos más pequeños a la izquierda y los mayores a la derecha.
 - Si la lista tiene dos o menos elementos ya está ordenada,
 - En caso contrario se realiza la misma operación con cada uno de los trozos de la lista original hasta que cada una tenga dos o menos elementos.
- ❑ La partición de los elementos se realiza a partir de un elemento arbitrario de la lista (*pivote*).
 - Una elección adecuada del pivote puede mejorar la eficiencia.

QuickSort (II)



Torres de Hanoi

- ❑ Se trata de un soporte con tres varillas y en una de ellas se encuentran discos concéntricos de distinto tamaño y ordenados de mayor a menor tamaño.
- ❑ El juego consiste en pasar todos los discos de otra varilla teniendo en cuenta que:
 - Los discos tienen que estar siempre situados en alguno de los soportes.
 - En cada movimiento sólo se puede pasar un disco.
 - Los discos siempre tienen que estar ordenados de mayor a menor tamaño.



Torres de Hanoi (II)

Pasar 3 discos de origen a destino



Paso 1



Paso 2



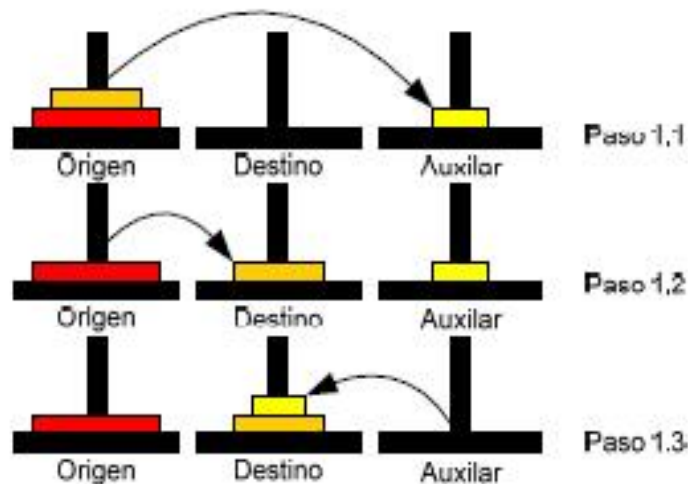
Paso 3



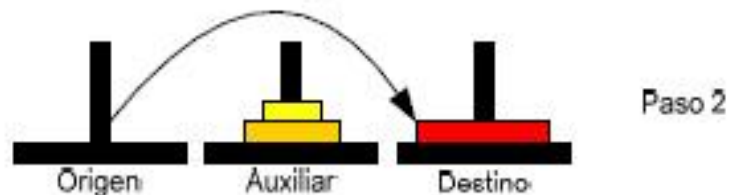
1. Pasar $n-1$ discos de la varilla de origen a la auxiliar.
2. Pasar el disco n de la varilla de origen a la de destino.
3. Pasar $n-1$ discos de la varilla auxiliar a la de destino.

Torres de Hanoi (III)

Pasar 2 discos de origen al nuevo destino



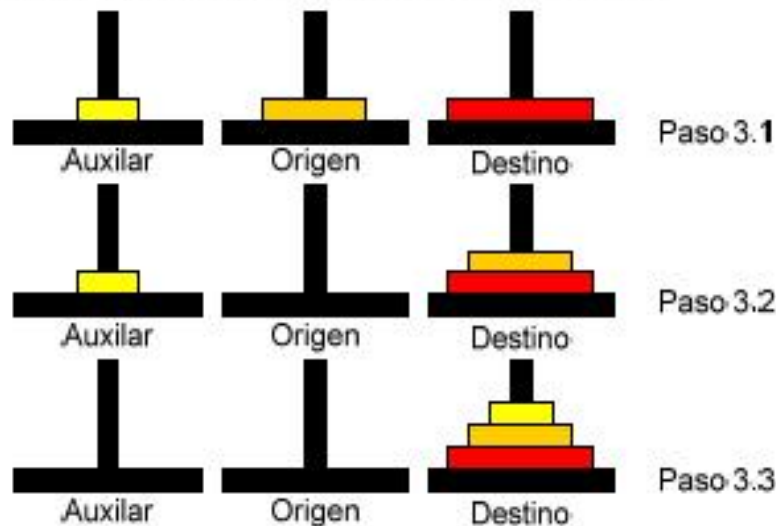
Pasar disco 3 de origen a destino



- 1.1. Pasar $n-2$ discos de la varilla de origen a la nueva varilla auxiliar.
- 1.2. Pasar el disco $n-1$ de la varilla de origen a la nueva varilla destino.
- 1.3. Pasar $n-2$ discos de la nueva varilla auxiliar a la nueva varilla destino
2. Pasar el disco n de la varilla de origen a la de destino.

Torres de Hanoi (IV)

Pasar 2 discos del nuevo origen a destino



3.1. Pasar $n-2$ discos de la nueva varilla de origen a la nueva varilla auxiliar.

3.2. Pasar el disco $n-1$ de la nueva varilla de origen a la de destino.

3.3. Pasar el $n-1$ disco de la nueva varilla auxiliar a la de destino.

Torres de Hanoi en “C”

```
#include <iostream>
using namespace std;

void hanoi(int num,char A,char C,char B)
{
    if(num==1)
    {
        cout<<"Mueva el bloque "<<num<<" desde "<<A<<" hasta  "<<C<<endl;

    }
    else
    {
        hanoi(num-1,A,B,C);
        cout<<"Mueva el bloque "<<num<<" desde "<<A<<" hasta  "<<C<<endl;
        hanoi(num-1,B,C,A);
    }
}

int main()
{
    int n;
    char A,B,C;

    cout<<"Los clavijas son A B C\n";
    cout<<"Numero de discos: ";
    cin>>n;
    hanoi(n,'A','C','B');

}
```

Recursividad versus iteración

- ☐ Si un subprograma se llama a si mismo se repite su ejecución un cierto numero de veces.
- ☐ Los procesos recursivos suelen ocupar más memoria y tardar más que los iterativos.

¿Cuáles son las razones para elegir la recursión?

Recursividad versus iteración

Tanto la iteración como la recursión se basan en una estructura de control:

- la iteración utiliza una estructura repetitiva
- la recursión utiliza una estructura de selección.

La iteración y la recursión implican ambas repetición:

- la iteración utiliza explícitamente una estructura repetitiva
- la recursión consume la repetición mediante llamadas repetidas.

La iteración y la recursión implican cada una un test de finalización

- La iteración termina después de realizar n - veces
- la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

```

#include<stdio.h>
#include<conio.h>
int main()
{
int fac=0;
printf("Ingresa numero para calcular el factorial
");
scanf("%d",&fac);
int temp=fac-1;
int r=fac;
while (temp>=1)
{
r=r*temp;
temp--;
}

printf("El factorial de %d es: %d ", fac,r);
getch();
}

```

Iterativo (while)

```

#include<stdio.h>

int factorial(int n)
{
int r;
if (n==1)
{
return 1;
}
r=n*factorial(n-1 ) ;
return (r ) ;
}

int main()
{
int n,res;
printf("*****n" ) ;
printf("tFactorial Recursivon" ) ;
printf("*****n" ) ;
printf("Dame un numero: " ) ;
scanf("%d",&n ) ;
res=factorial(n ) ;
printf("El factorial de %d es: %d",n,res ) ;
return 0;
}

```

Recursivo

Conclusión

- Se puede decir que la recursividad es una técnica de programación bastante útil y muy interesante de estudiar.
- Recuerden que muchos problemas pueden resolverse con la misma facilidad usando recursividad como iteración.
- En general para un mismo algoritmo, la recursividad permite una expresión más clara y sintética lo que simplifica la comprensión y el mantenimiento del mismo.
- Las soluciones recursivas son normalmente menos eficientes, ya que consumen más tiempo (las invocaciones) y memoria (pila de activación).
- La recursividad es apropiada, también, cuando los datos involucrados en el problema están organizados en estructuras que pueden definirse recursivamente, como listas, árboles, etc.
- Las estrategias de resolución de problemas denominada “divide y vencerás” o “backtracking”, son dos técnicas muy utilizadas para la resolución de algoritmos complejos. Ambas utilizan recursividad.

BIBLIOGRAFIA. Especifica:

- Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2003; Editorial: MCGRAW-HILL. ISBN: 8448136642. Capítulo 16.
- ALGORITMOS, DATOS Y PROGRAMAS con aplicaciones en Pascal, Delphi y Visual Da Vinci. De Guisti. Armando. 2001. editorial: Prentice Hall. ISBN: 987-9460-64-2. Capitulo 7.
- PROGRAMACIÓN; Castor F. Herrmann, María E. Valesani.; 2001; Editorial: MOGLIA S.R.L..ISBN: 9874338326. Capitulo 4.
- ESTRUCTURA DE DATOS; Cairó y Guardati; 2002; Editorial: MCGRAW-HILL. ISBN: 9701035348. Capitulo 11.
- Fundamentos de programación. Algoritmos, estructuras de datos y objetos; Luis Joyanes Aguilar; 2008; Editorial: MCGRAW-HILL. ISBN: 8448136642. Capítulo 14.
- ALGORITMOS Y PROGRAMACION EN PASCAL. Cristobal Parejas Flores y otros.

En internet ...

- http://exa.unne.edu.ar/informatica/programacion1/public_html/
- <http://web.austral.edu.ar/austral-admisionesGrado-ingenieriaInformatica.asp>
- http://www.cepec.edu.ar/informatica.html?gclid=CMj29-q_i6QCFdhA2godpQ0oHA
- <http://www.frgp.utn.edu.ar/carreras/tssi/index.php>
- <http://www.mitecnologico.com/Main/RecursividadProgramacion>

Ejecución de Soluciones a problemas recursivos implementados en Pascal

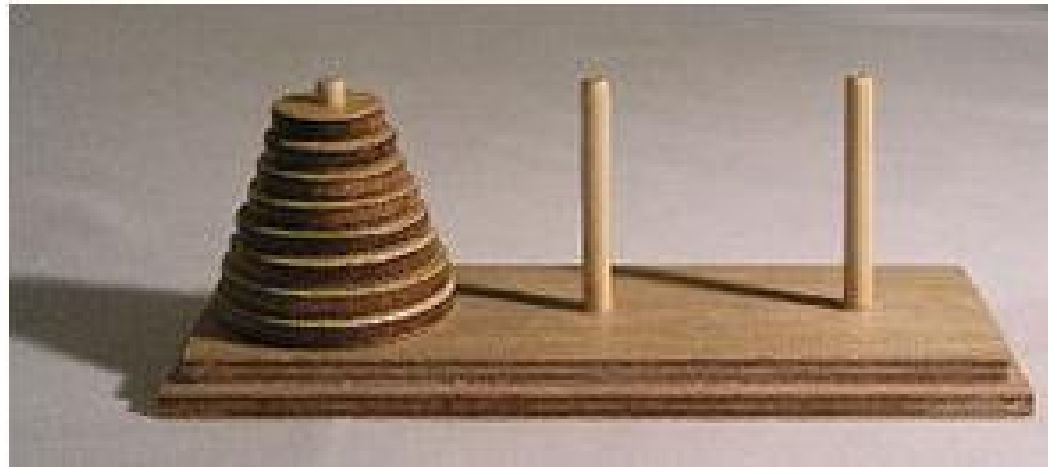
- Función Factorial
- Serie de Fibonacci
- Torres de Hanoi
- Ej. Recursividad Mutua



Gracias ...

Ejemplo 3: Torres de Hanoi

Un problema típico a resolver con recursión es el de las Torres de Hanoi, ya que al aplicar esta herramienta el problema se simplifica enormemente. Las Torres de Hanói es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édouard Lucas



Consiste en tres varillas verticales y un número indeterminado de discos que determinarán la complejidad de la solución. No hay dos discos iguales, están colocados de mayor a menor en la primera varilla ascendentemente, y no se puede colocar ningún disco mayor sobre uno menor a él en ningún momento..

Llamaremos A, B y C a cada una de las agujas sin importar el orden siempre que se mantengan los nombres.

Consideremos inicialmente dos discos en A que queremos pasar a B utilizando C como auxiliar. Las operaciones por realizar son sencillas:

$$\text{Pasar dos discos de A a B} = \begin{cases} \text{Mover un disco de A a C} \\ \text{Mover un disco de A a B} \\ \text{Mover un disco de C a B} \end{cases}$$

- Ahora supongamos que tenemos tres discos en A y queremos pasarlos a B.
- Haciendo algunos tanteos descubrimos que hay que pasar los dos discos superiores de A a C, mover el último disco de A a B y por último pasar los dos discos de C a B. Ya conocemos cómo pasar dos discos de A a B usando C como auxiliar, para pasarlos de A a C usaremos B como varilla auxiliar y para pasarlos de C a B usaremos A como auxiliar:

$$\text{Pasar 3 discos de A a B} = \left\{ \begin{array}{l} \text{Pasar dos de A a C} = \left\{ \begin{array}{l} \text{Mover 1 disco de A a B} \\ \text{Mover 1 disco de A a C} \\ \text{Mover 1 disco de B a C} \end{array} \right. \\ \text{Mover un disco de A a B} \\ \text{Pasar dos de C a B} = \left\{ \begin{array}{l} \text{Mover 1 disco de C a A} \\ \text{Mover 1 disco de C a B} \\ \text{Mover 1 disco de A a B} \end{array} \right. \end{array} \right.$$

En general, Pasar n discos de A a B, consiste en efectuar las siguientes operaciones (siendo $n \geq 1$).

$$\text{Pasar } n \text{ discos de A a B} = \begin{cases} \text{Pasar } n-1 \text{ discos de A a C} \\ \text{Mover 1 disco de A a B} \\ \text{Pasar } n-1 \text{ discos de C a B} \end{cases}$$

siendo 1 el caso base, que consiste en mover simplemente un disco sin generar llamada recursiva. Ahora apreciamos claramente la naturaleza recursiva de proceso, pues para pasar n discos es preciso pasar $n-1$ discos (dos veces), para $n-1$ habrá que pasar $n-2$ (también dos veces) y así sucesivamente.