Fundamentos de Programación II

Tema 2. Recursividad

Luís Rodríguez Baena (luis.rodriguez@upsam.net)

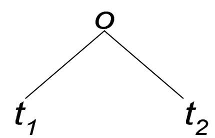
Universidad Pontificia de Salamanca (campus Madrid)
Escuela Superior de Ingeniería y Arquitectura

Naturaleza de la recursividad

- Se dice que un *objeto es recursivo* cuando forma parte de si mismo.
  - Permite definir un número infinito de objetos mediante un enunciado finito.
- En programación…
  - La recursividad es la propiedad que tienen los procedimientos y funciones de llamarse a si mismos para resolver un problema.
  - Permite describir un número infinito de operaciones de cálculo mediante un programa recursivo finito **sin implementar de forma explícita estructuras repetitivas.**

Naturaleza de la recursividad (II)

- Ejemplos de definiciones recursivas:
  - Números naturales.
    - ✓ 0 es un número natural.
    - ✓ El sucesor del número natural *x* (*sucesor(x)*) es también un número natural.
  - Estructuras de árbol.
    - ✓ Si *O* no tiene hijos es un árbol vacío.
    - ✓ Si *O* tiene dos hijos, t_1 y t_2 , éstos también son árboles.



  - Multiplicación.
    - ✓ $x \cdot 0 = 0$
    - ✓ Para $y > 0$, $x \cdot y = x + x \cdot (y-1)$
  - Factorial de un número.
    - ✓ $0! = 1$
    - ✓ Si n es mayor que 0, $n! = n \cdot (n-1)!$
  - Potencia de un número.
    - ✓ $x^0 = 1$
    - ✓ Si $y > 0$, $x^y = x \cdot x^{y-1}$

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
```

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.

```
procedimiento Recursivo()
…
inicio
   …
   Recursivo(…)
   …
fin_función
```

Recursividad directa

```
procedimiento Recursivo1()
…
inicio
   …
   Recursivo2(…)
   …
fin_función
```

```
procedimiento Recursivo2()
…
inicio
   …
   Recursivo1(…)
   …
fin_función
```

Recursividad indirecta

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-2)
   fin_si
   ...
fin_procedimiento
```

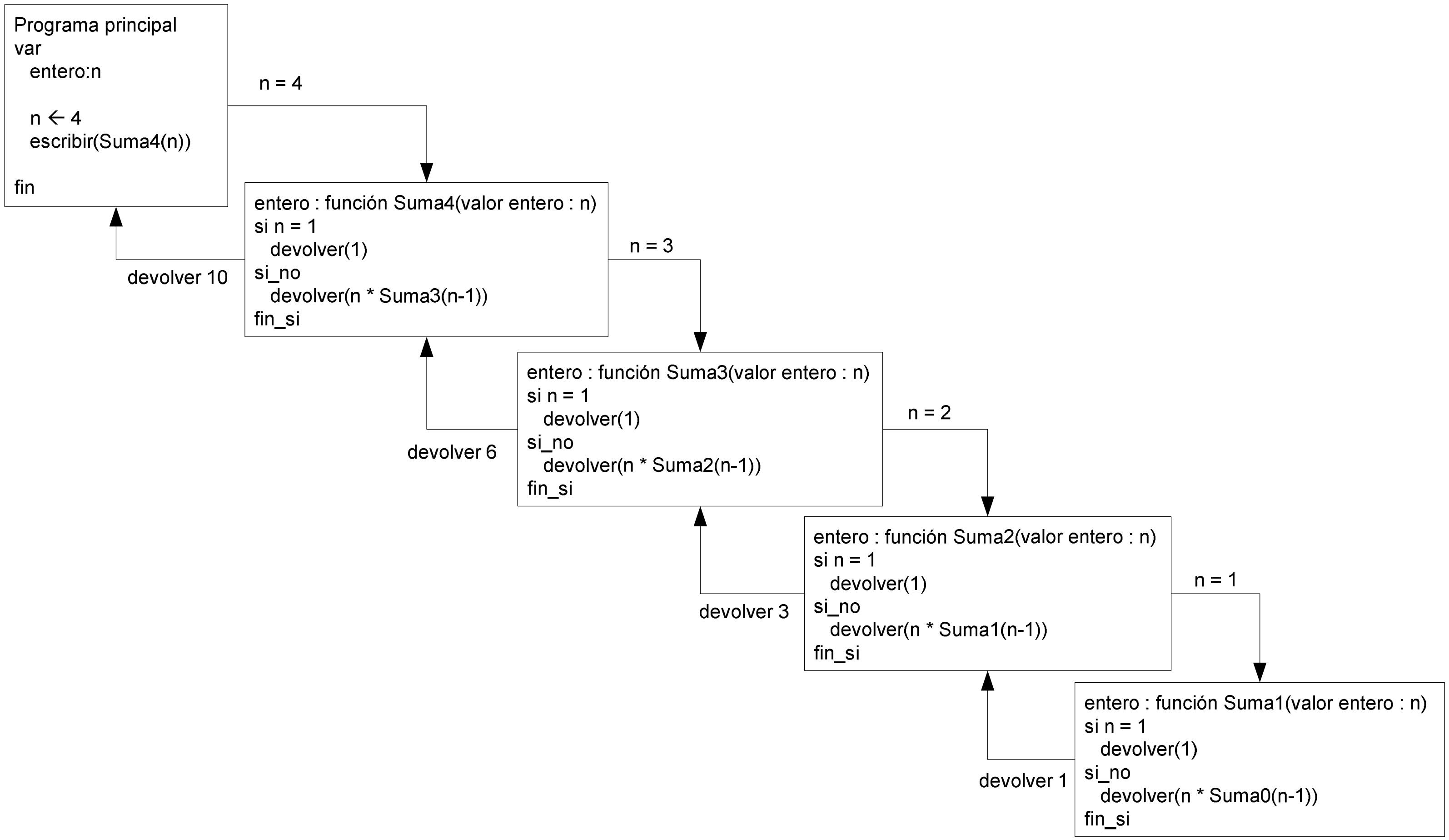
Recursividad múltiple

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

Recursividad anidada

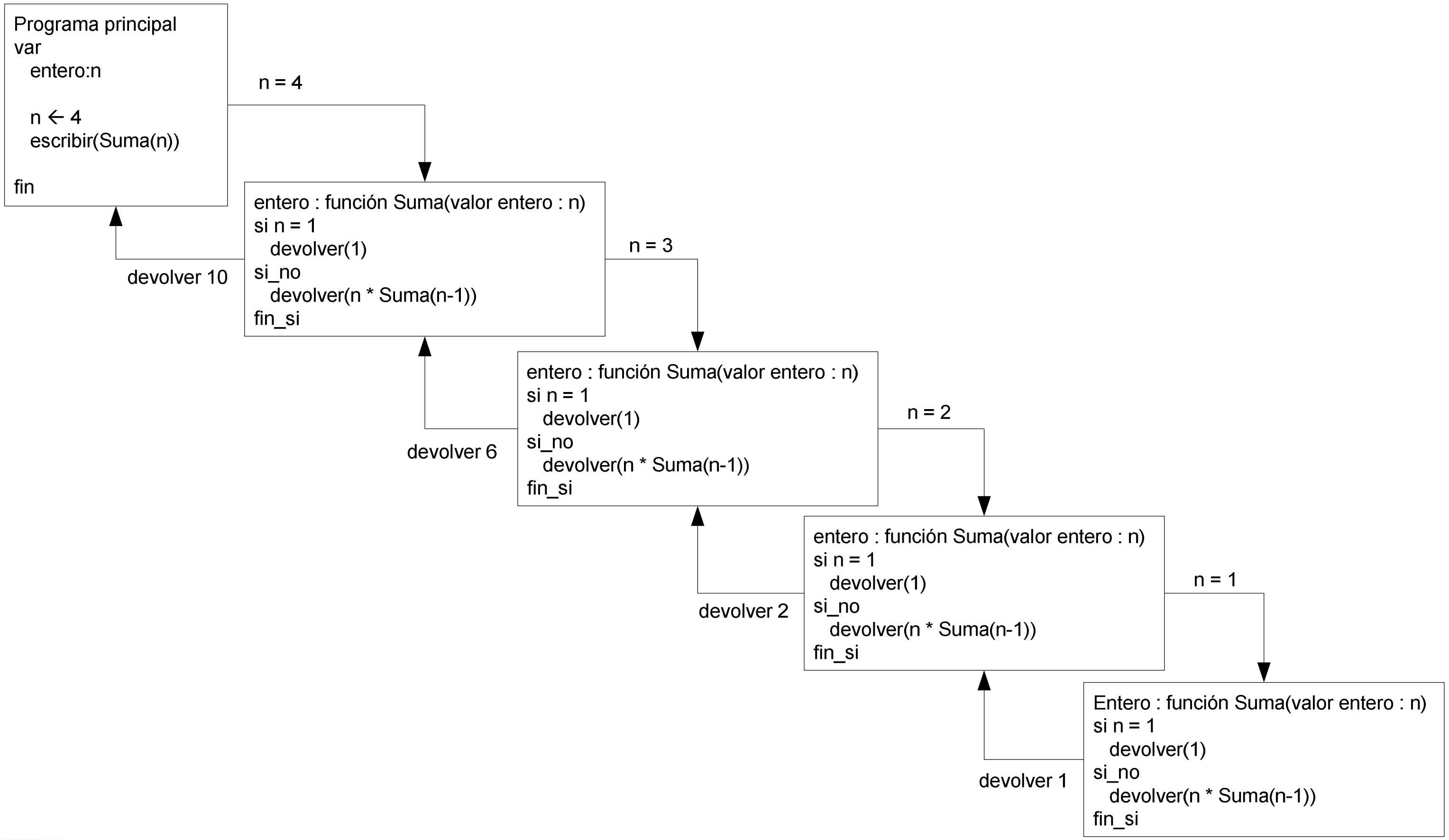
Llamadas a módulos recursivos

- 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, suma(1) = 1). Por inducción se puede suponer que las llamadas más pequeñas (por ejemplo, Suma(n-1)) quedan resueltas.

devuelve 10

Suma(4) = 4 + Suma(3) = 4 + 6

devuelve 6

Suma(3) = 3 + Suma(2) = 3 + 3

devuelve 3

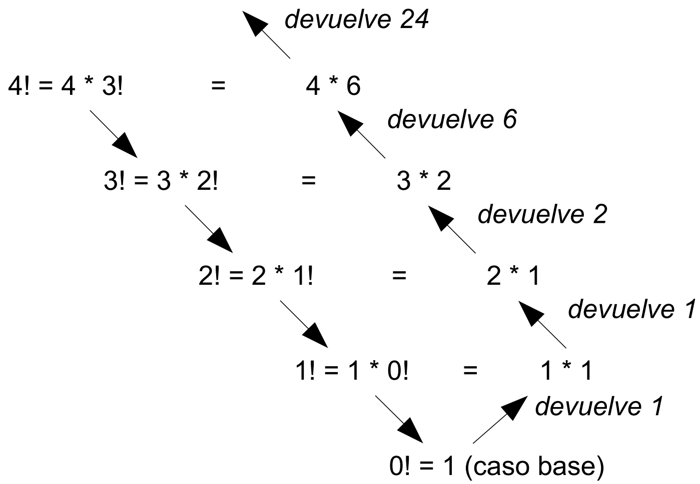
Suma(2) = 2 + Suma(1) = 2 + 1

devuelve 1

Suma(1) = 1 (caso base)

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.

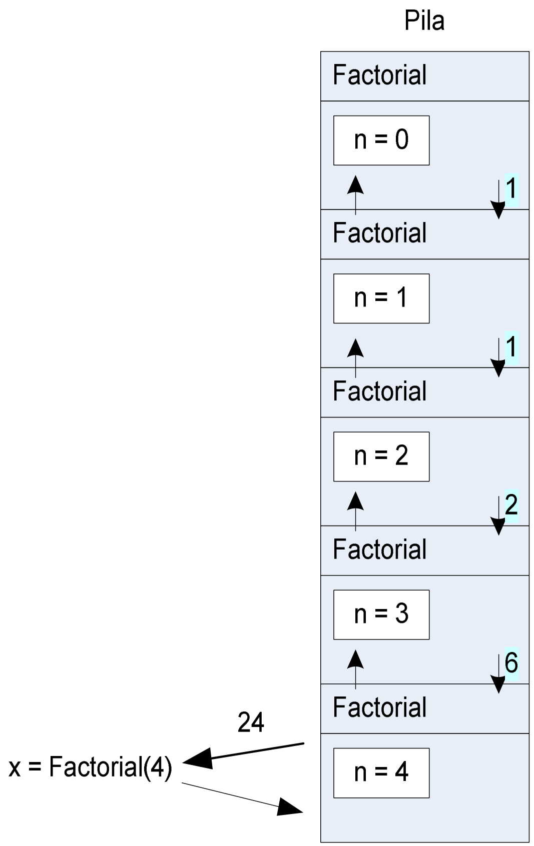
```
entero función Factorial(valor entero : n)
inicio
   //Para cualquier n entero positivo <= 1, n! = 1
   //n <= 1 sería el caso base, caso trivial o fin de la recursión
   si n < 1 entonces
      devolver(1)
   si_no
      //En caso contario n! = n * (n-1)!
      //En cada llamada n se acerca al caso 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 llamadas recursivas a la función (1, 1, 2, 6, 24).

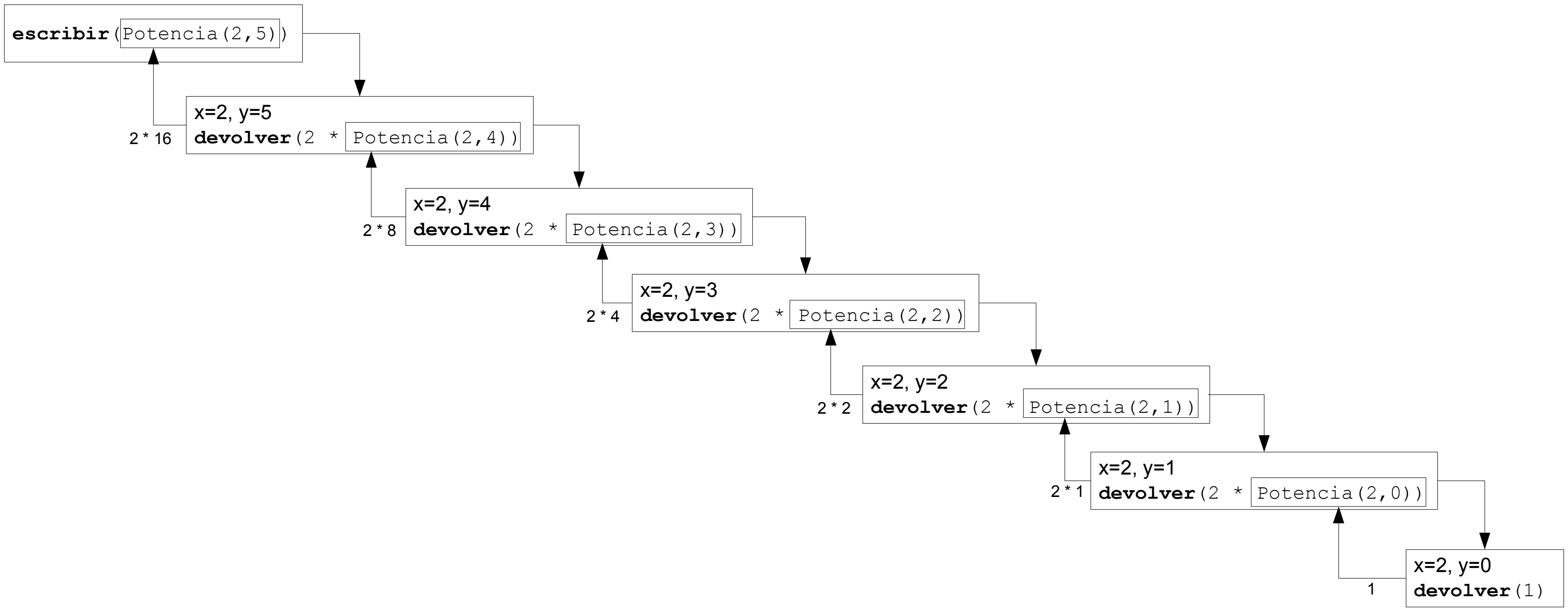


Algoritmos recursivos

- 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
```

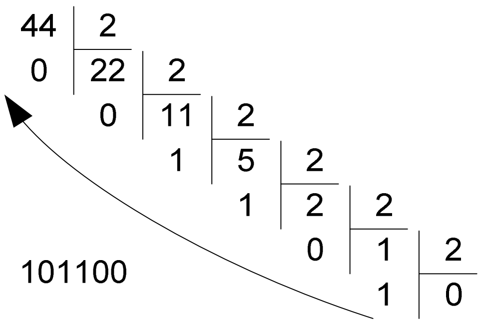
Algoritmos recursivos (II)



Algoritmos recursivos (III)

- Escribir un número decimal en binario.
  - Caso base: si n < 2 n en binario es n.
  - Si n >= 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) → `10`.
    - ✓ 3 en binario es `3 div 2` en binario (1), seguido de `3 mod 2` (1) → `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
```

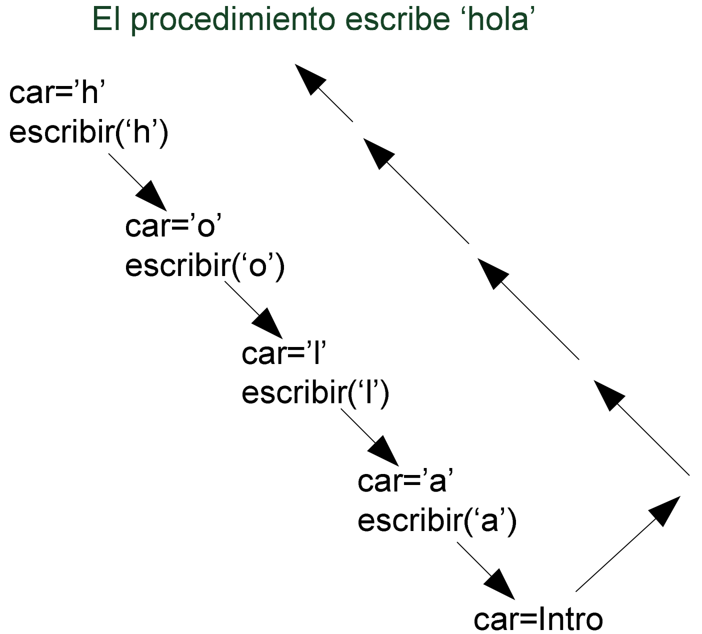


Algoritmos recursivos (IV)

- 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.

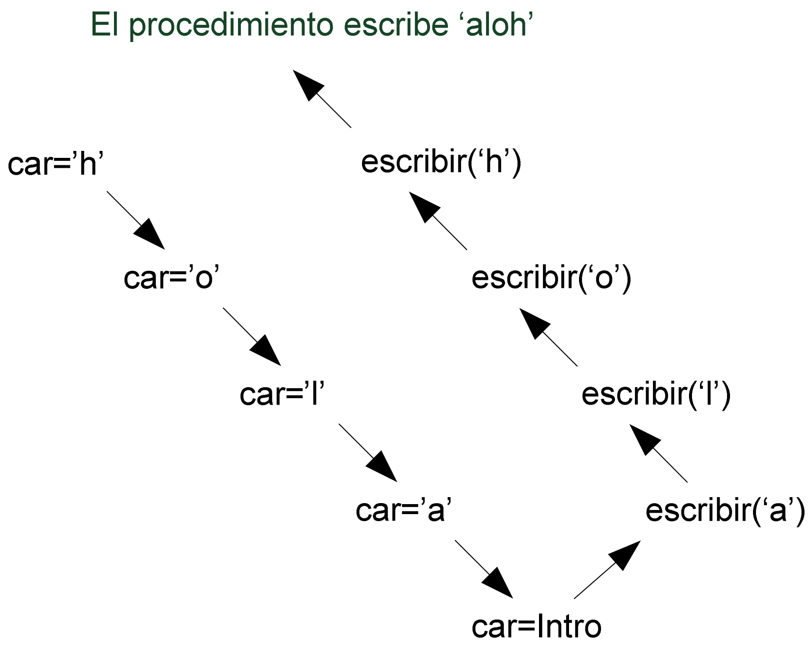


Algoritmos recursivos (IV)

- 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
```

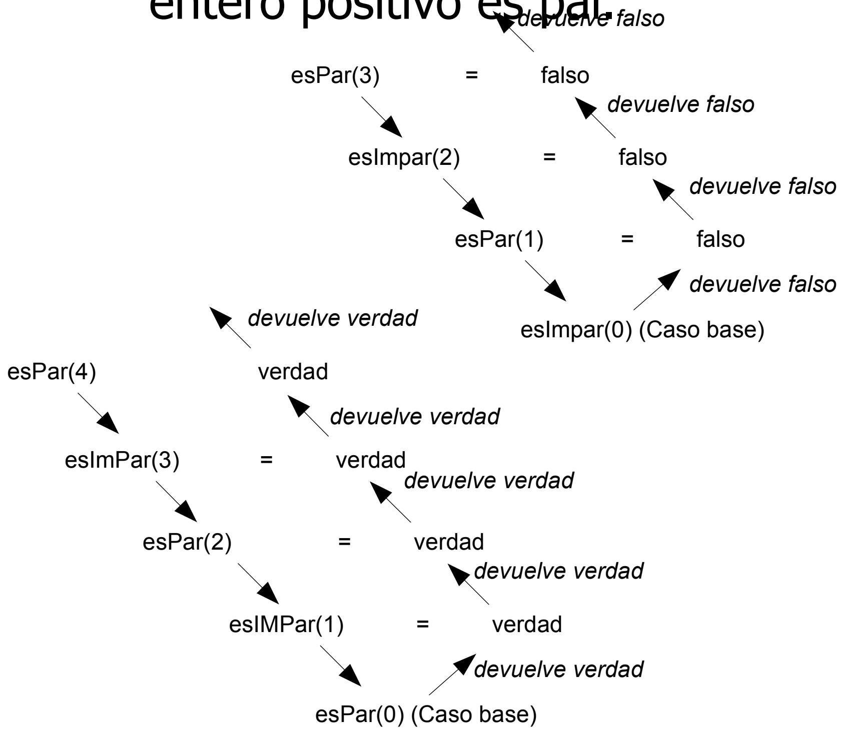
- La secuencia leída de caracteres es h,o,l,a,Intro.



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.



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.

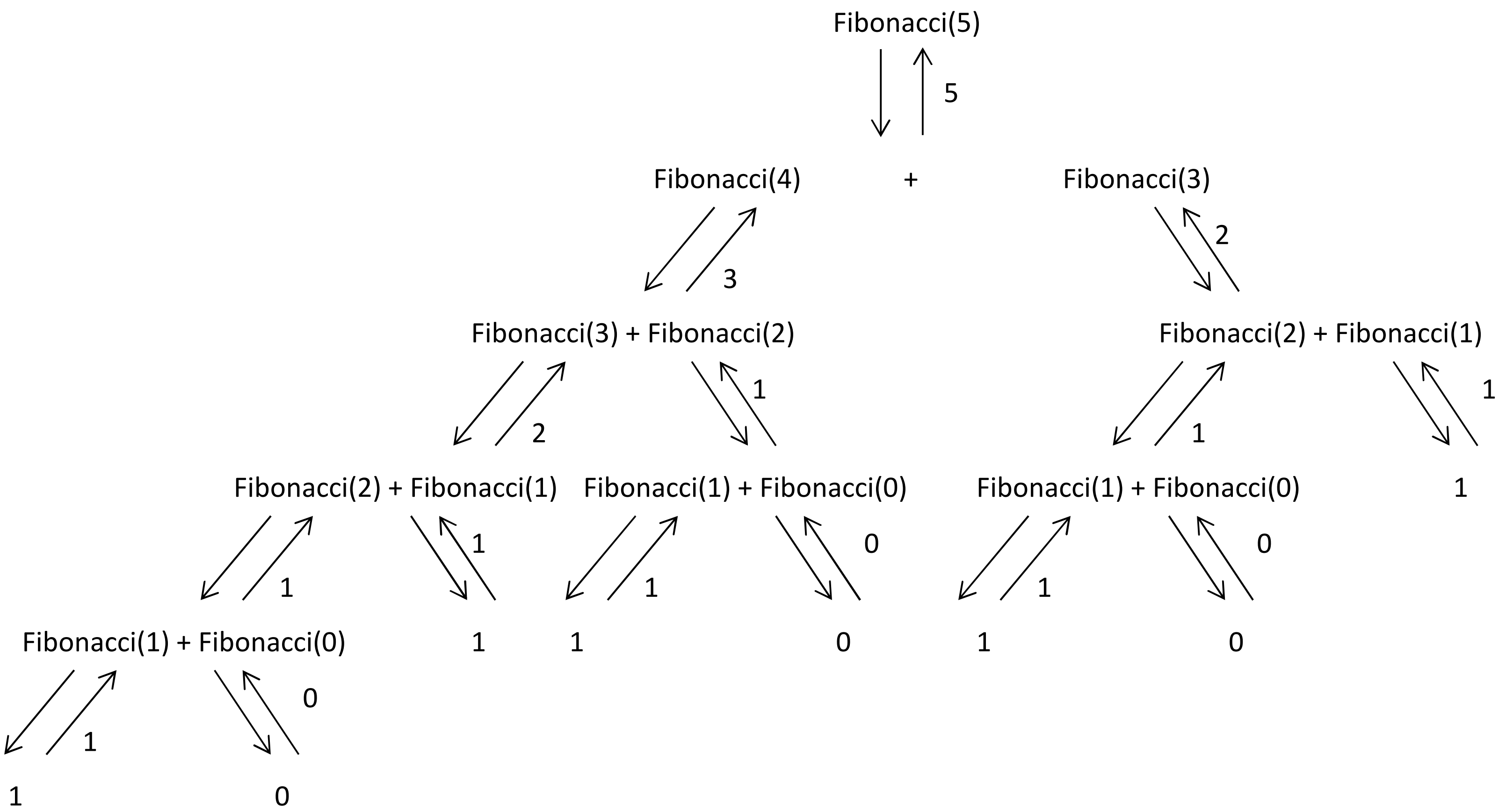
```
entero función Fibonacci(valor entero: n)
var
   entero: término, último, penúltimo
inicio
   último ← 1
   penúltimo ← 0
   desde i ← 2 hasta n hacer
      término ← penúltimo + último
      penúltimo ← último
      último ←  término
   fin_desde
   devolver(último)
fin_función
```

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 $n > 1$, $Fib_n = Fib_{n-1} + Fib_{n-2}$.

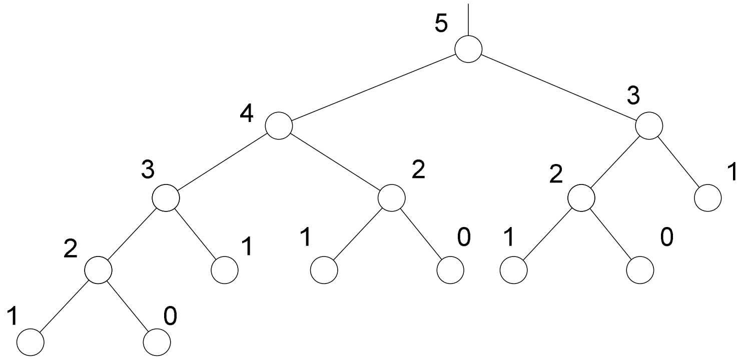
```
entero función Fibonacci(valor entero : n)
inicio
   si (n <= 1) entonces
      devolver(n)
   si_no
      devolver(Fibonacci(n-1) + Fibonacci(n-2)
   fin_si
fin_función
```

Ventajas e inconvenientes (V)



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. Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2).
  2. Fibonacci(0) = 0 y 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. Fibonacci(2) = Fibonacci(1) + Fibonacci(0) = 1 + 0, se construye la solución del problema 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 binara 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)

	1	2	3	4	5	6	7	8	9	10	11	12	
1	4	63	12	40	60	23	67	8	50	7	5	44	23
	izq											der	pivote
2	4	5	12	7	3	23	67	60	50	40	63	44	12
	izq				der								pivote
3	4	5	3	7	12	23	67	60	50	40	63	44	5
	izq			der									pivote
4	4	3	5	7	12	23	67	60	50	40	63	44	4
	izq	der											pivote
5	3	4	5	7	12	23	67	60	50	40	63	44	5
			izq	der									pivote
6	3	4	5	7	12	23	44	40	50	60	63	67	50
							izq					der	pivote
7	3	4	5	7	12	23	44	40	50	60	63	67	44
							izq	der					pivote
8	3	4	5	7	12	23	40	44	50	60	63	67	63
										izq		der	pivote
9	3	4	5	7	12	23	40	44	50	60	63	67	

Elementos menores

Elementos mayores

QuickSort (III)

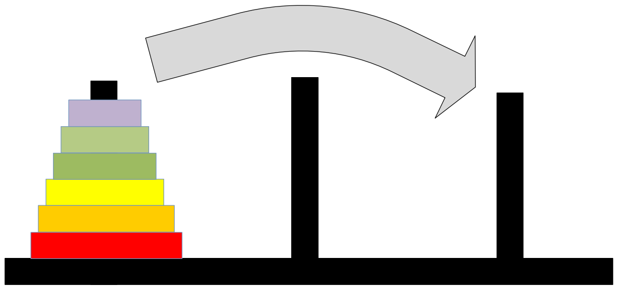
```
procedimiento QuickSort(ref vector : v; valor entero : izq,der)
var
   entero : i,j
   TipoElemento : pivote
inicio
   i ← izq
   j ← der
   pivote ← ObtenerPivote(v,izq,der) //Podría ser v[(izq+der) div 2]
   //Partición de los elementos entre las posiciones izq y der
   repetir
      //Se incrementa la i hasta encontrar un elemento v[i]
      //mal colocado (no menor que el pivote)
      mientras v[i] < pivote hacer
         i ← i + 1
      fin_mientras
      //Se decrementa la j hasta encontrar un elemento v[j]
      //mal colocado (no mayor que el pivote)
      mientras v[j] > pivote hacer
         j ← j - 1
      fin_mientras
      //Si se han encontrado dos elementos mal
      //colocados, se intercambian
```

QuickSort (IV)

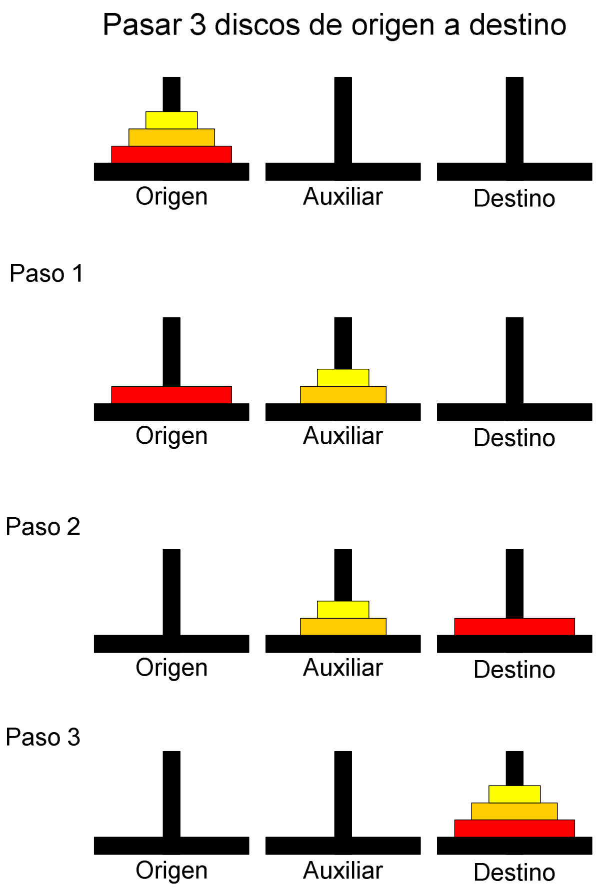
```
      //Si se han encontrado dos elementos mal
      //colocados (i y j no se han cruzado) , se intercambian
      si i <= j entonces
          intercambia(v[i],v[j])
          i ← i + 1
          j ← j - 1
      fin_si
   hasta_que i > j
   //El proceso acaba cuando se han cruzado los índices
   //Los elementos menores se han quedado entre las
   //posiciones izq y j,
   //Los mayores entre las posiciones i y der
   //Si esas sublistas tienen más de un elemento,
   //se vuelven a ordenar mediante llamadas recursivas
   //Si hay sublista izquierda se ordena
   si izq < j entonces
      QuickSort(v,izq,j)
   fin_si
   //Si hay sublista derecha, se ordena
   si i < der entonces
      QuickSort(v,i,der)
   fin_si
fin_procedimiento
```

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.

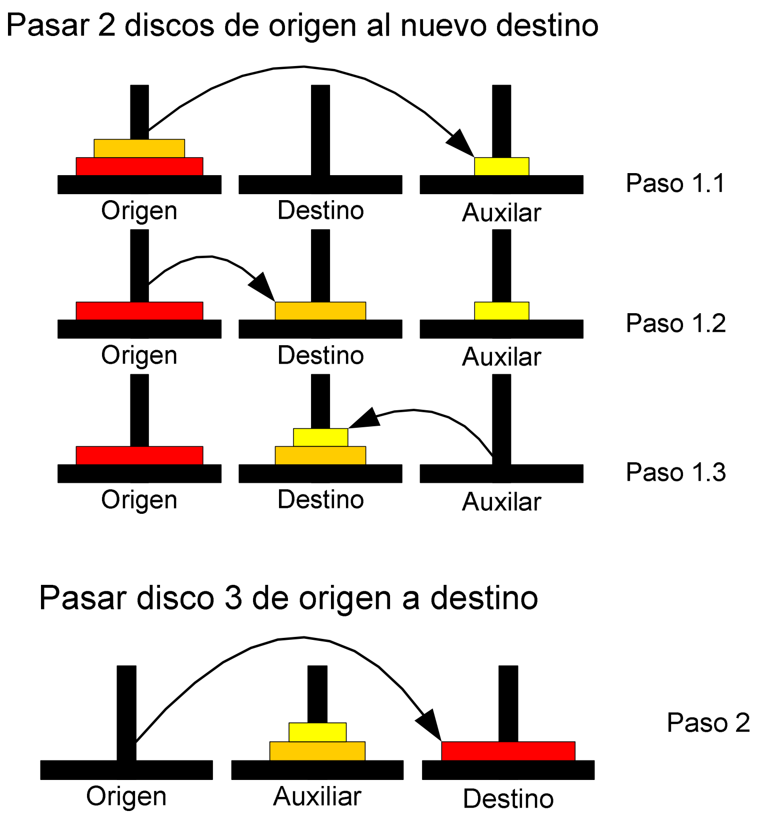


Torres de Hanoi (II)



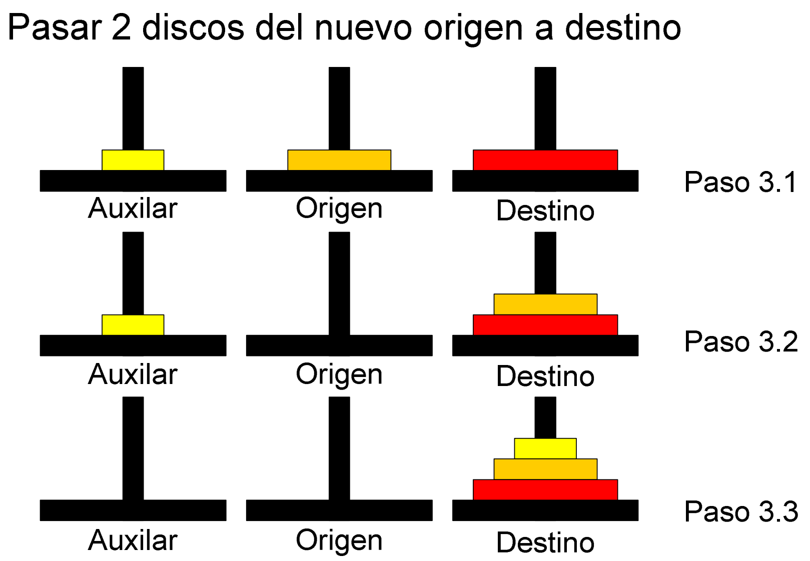
1. Pasar n-1 discos de la varilla de origen a la auxiliar.
2. Pasar el disco n de la varilla de origen a la de destino.
3. Pasar n-1 discos de la varilla auxiliar a la de destino.

Torres de Hanoi (III)



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)



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 (V)

```
procedimiento TorresDeHanoi()
var
   entero : NumDiscos
inicio
   leer(NumDiscos)
   MoverDiscos(NumDiscos,1,2,3)
fin_procedimiento
procedimiento MoverDiscos(valor entero : origen,auxiliar,destino,n)
inicio
   //Si hay discos
   si n > 0 entonces
      //Se mueven n-1 discos de origen a auxiliar utilizando destino
      MoverDiscos(n-1,origen,destino,auxiliar)
      //Se mueve el disco n de origen a destino
      escribir('Mover disco', n , 'de' , origen, 'a' ,destino)
      //Se mueven n-1 discos de auxiliar a destino utilizando origen
      MoverDiscos(n-1,auxiliar,origen,destino)
   fin_si
fin_procedimiento
```

Ejercicios

1. Obtener el máximo común divisor de dos números enteros positivos.
2. Escribir un procedimiento recursivo que escriba los números entre 1 y n.
3. Escribir un procedimiento recursivo que escriba los números entre n y 1.
4. Dada la siguiente serie $a_n=a_{n-1}+2^n$, para $n > 0$ (para $n=0$, $a_n=0$), diseñar una función recursiva que calcule el término n de la serie.
5. Escribir una función recursiva que devuelva el producto de dos números enteros mayores que 0 mediante sumas sucesivas.
6. Leer una secuencia de números enteros hasta introducir un 0. Devolver la misma secuencia al revés y eliminando los números repetidos.
   - Si la secuencia, por ejemplo, es 2 3 3 5 7 7 9 4 0, debería devolver 4 9 7 5 3 2.

Ejercicios (II)

7. Escribir una cadena al revés.
   - Notas:
     - ✓ Si la cadena c contiene el valor “hola”, el elemento c[2] contendrá el carácter “h” (el segundo carácter).
     - ✓ La función `longitud(c)` devolvería el número de caracteres de la cadena (4)
8. Escribir los elementos de un vector.
9. El número de combinaciones de m elementos tomados de n en n para n >= 1 y 0 <= m <=n se puede definir de forma recursiva:
   - Si m= 0 o n = n o n= 1, (n,m) = 1
   - En caso contrario (n,m) = (n-1,m) + (n-1,m-1)
   - Diseñar una función recursiva que calcule el número de combinaciones (n,m).

- El producto escalar de dos vectores es un número real obtenido mediante el sumatorio del producto de cada uno de sus elementos.

  Por ejemplo si el vector a tiene los elementos 3, 4 y 8, y el vector b los elementos 2, 1 y 3, el producto escalar a·b será 3x2 + 4x1 + 3x8 = 6+4+24 = 34.

  Diseñe una función recursiva que permita calcular el producto escalar de dos vectores de n elementos.

- La función de Ackerman de dos números m y n ($A_{m,n}$) se puede definir de forma recursiva de la siguiente forma:

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

  Diseñe una función recursiva que obtenga el valor de la función de Ackerman

- Escribir una función recursiva que devuelva el cociente y el resto de la división entera mediante resta sucesivas.
- Los números de Catalan forman una secuencia de números naturales. El n-ésimo número de Catalan (C_n) se puede calcular de forma recursiva:

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

- Diseñe una función recursiva que obtenga el valor del n-ésimo número de Catalan.
- Hacerla un poco más complicada y meter esta en los apuntes

Ejemplos

- Obtener el máximo común divisor de dos números enteros a y b.
  - Caso base: si a es divisible entre b entonces, mcd(a,b) = b
  - En caso contrario, mcd(a,b) = mcd(b, a div b).

```
entero función mcd(valor entero : a,b)
inicio
   si a mod b = 0 entonces
      devolver(b)
   si_no
      devolver(mcd(b, a div b)
   fin_si
fin_función
```

Ejemplos (II)

- Escribir una cadena al revés.
  - Caso base: si la cadena tiene uno o menos caracteres, se escribe la propia cadena.
  - Si tiene más de dos caracteres, se escribe al revés, todos los caracteres de la cadena menos el primero y a continuación el primer carácter.

```
procedimiento reves(E cadena: c)
inicio
   si longitud(c) > 1 entonces
      reves(subcadena(c,2))
   fin_si
   escribir(c[1])
fin_función
```

Ejemplos (III)

- Búsqueda binaria recursiva.
  - Caso base: Si el elemento central es el buscado, el elemento está en la posición central.
  - Caso base: si el elemento central no es el buscado y la lista tiene sólo un elemento, el elemento no está.
  - Caso general: si el elemento central el mayor que el buscado, se realiza una búsqueda binaria en la parte izquierda de la lista.
  - Caso general: si el elemento central es menor que el buscado se realiza una búsqueda binaria en la parte derecha de la lista.

Ejemplos (IV)

```
entero : función Buscar(valor vector: v; valor entero : elem, iz, de)
var
   entero : ce
inicio
   ce ← (iz + de) div 2
   si v[ce] = elem entonces
      devolver(ce)
   si_no
      si iz >= de entonces
          devolver(0)
      si_no
          si elem > v[ce] entonces
              devolver(Buscar(v,elem,iz,ce-1)
          si_no
              devolver(Buscar(v,elem,ce+1,de)
          fin_si
      fin_si
   fin_si
fin_función
```

Ejemplos (V)

- Leer una secuencia de números enteros hasta introducir un 0. Devolver la misma secuencia al revés eliminando los repetidos.

```
procedimiento Repetidos(valor entero: ant)
var
   entero : aux
inicio
   leer(aux)
   si aux <> 0 entonces
      si aux <> ant entonces
          Repetidos(aux)
          escribir(aux)
      si_no
          Repetidos(aux)
      fin_si
   fin_si
fin_función
```

Ejemplos (VI)

- Realizar la suma de los elementos de un vector.
  - Caso base: si el número de elementos del vector es 1, entonces la suma es el elemento 1.
  - Si el número de elementos es mayor que uno, la suma es el elemento n + la suma de todos los elementos menos el último.

```
entero función SumaVector(valor vector : v; valor entero : n)
inicio
   //Se supone que el vector está en base 1
   si n = 1 entonces
      devolver(v[1])
   si_no
      devolver(v[n] + SumaVector(v, n-1))
   fin_si
fin_función
```