

Recursividad

Definición

Un procedimiento o función se dice recursivo si durante su ejecución se invoca directa o indirectamente a sí mismo. Esta invocación depende al menos de una condición que actúa como condición de corte que provoca la finalización de la recursión.

Un algoritmo recursivo consta de:

1- Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se ejecuta cuando se cumple cierta condición, y

2- el caso general que es el que vuelve a invocar al algoritmo con un caso más pequeño del mismo.

Los lenguajes como Pascal, que soportan recursividad, dan al programador una herramienta poderosa para resolver ciertos tipos de problemas reduciendo la complejidad u ocultando los detalles del problema.

La recursión es un medio particularmente poderoso en las definiciones matemáticas. Para apreciar mejor cómo es una llamada recursiva, estudiemos la descripción matemática de factorial de un número entero no negativo:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n - 1) * (n - 2) * \dots * 1 = n * (n - 1)! & \text{si } n > 0 \end{cases}$$

Esta definición es recursiva ya que expresamos la función factorial en términos de sí misma.

Ejemplo: $4! = 4 * 3!$

Por supuesto, no podemos hacer la multiplicación aún, puesto que no sabemos el valor de $3!$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

Podemos ahora completar los cálculos

$$1! = 1 * 1 = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 = 6$$

$$4! = 4 * 6 = 24$$

Diseño de Algoritmos Recursivos

Para que una función o procedimiento recursivo funcione se debe cumplir que:

- Existe una salida no recursiva del procedimiento o función y funciona correctamente en ese caso.
- Cada llamada al procedimiento o función se refiere a un caso más pequeño del mismo.
- Funciona correctamente todo el procedimiento o función.

Para poder construir cualquier rutina recursiva teniendo en cuenta lo anterior, podemos usar el siguiente método:

- Primero, obtener una definición exacta del problema.
- A continuación, determinar el tamaño del problema completo a resolver. Así se determinarán los valores de los parámetros en la llamada inicial al procedimiento o función.
- Tercero, resolver el caso base en el que problema puede expresarse no recursivamente. Esto asegurará que se cumple el punto 1 del test anterior.
- Por último, resolver correctamente el caso general, en términos de un caso más pequeño del mismo problema (una llamada recursiva). Esto asegurará cumplir con los puntos 2 y 3 del test.

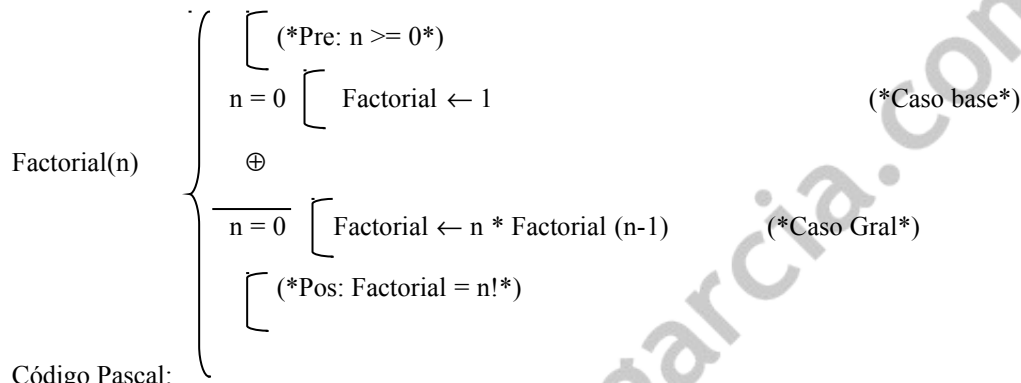
Cuando el problema tiene una definición formal, posiblemente matemática, como el ejemplo del cálculo del factorial, el algoritmo deriva directamente y es fácilmente implementable en otros casos debemos encontrar la solución más conveniente.

Ejemplo

Cálculo del factorial para enteros no negativos

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n - 1) ! & \text{si } n > 0 \end{cases}$$

Diagrama de Llaves:



```
function factorial(n: integer):integer;
begin
if n=0 then factorial = 1 (* caso base *)
else factorial = n * factorial(n-1) (* caso general *)
end;
```

Cómo funcionan los Algoritmos Recursivos

Para entender cómo funciona la recursividad es necesario que tengamos presente las reglas y los tipos de pasaje de parámetros provistos por Pascal.

Si un procedimiento o función p invoca a otro q , durante la ejecución de q se reservarán locaciones de memoria para todas las variables locales de q y para los parámetros pasados por valor. Al terminar la ejecución de q este espacio es desocupado. Ahora bien, si durante la ejecución de q se produce una llamada a sí mismo, tendremos una segunda "instancia" de q en ejecución, la primera instancia se suspende hasta que la instancia recursiva termine. Antes de iniciarse la ejecución recursiva de q , se ocupan nuevas locaciones de memoria para las variables locales y parámetros por valor de q . Cualquier referencia a estas variables accederá a estas locaciones. Las locaciones reservadas durante la ejecución inicial de q son inaccesibles para la 2da. instancia de q .

Cuando la 2da. instancia de q termine, el control vuelve a la primera instancia de q , exactamente al lugar siguiente a la llamada recursiva. Cualquier referencia a las variables locales o parámetros por valor de q accederá a las locaciones reservadas inicialmente, inaccesibles para la segunda instancia de q .

Como vemos, no se conoce la cantidad de memoria que va a utilizarse al ejecutar un procedimiento o función recursivo sino que se produce una asignación dinámica de memoria, es decir, a medida que se producen instancias nuevas, se van "apilando" las que quedan pendientes: se ponen al menos tres elementos en la pila:

- una para la dirección de vuelta,
- otra para el/los parámetro/s formal/es y
- otra para el identificador de la función que en esencia es un parámetro pasado por variable.

Cuando la última instancia de la recursión - elige el caso base- se cumple, se "desapila" esa instancia y el control vuelve a la instancia anterior; así hasta "desapilar" todas las instancias.

Esta "pila" que se va generando en memoria es importante tenerla en cuenta por lo que debemos asegurarnos que el algoritmo recursivo no sea divergente.

Vamos a ver cómo funciona la recursión en algunos problemas vistos a través de una traza dónde iremos guardando los valores de los parámetros formales.

Algoritmos & Diagramación

www.profmatiasgarcia.com.ar

Hagamos un recorrido del código de la función factorial:

```
function factorial(n: integer):integer;
begin
if n=0 then factorial = 1           (* caso base *)
else factorial = n * factorial(n-1) (* caso general *)
end;
```

La llamada inicial es: write(factorial(5)).

La primera instancia se coloca en la pila, luego a medida que se va invocando la función, vamos apilando el valor del parámetro formal que se envía:

Como n=0 el algoritmo va por la rama del then llegando al caso base obteniendo factorial(0) = 1

0	Instancia 6
1	Instancia 5
2	Instancia 4
3	Instancia 3
4	Instancia 2
5	Instancia 1
N	

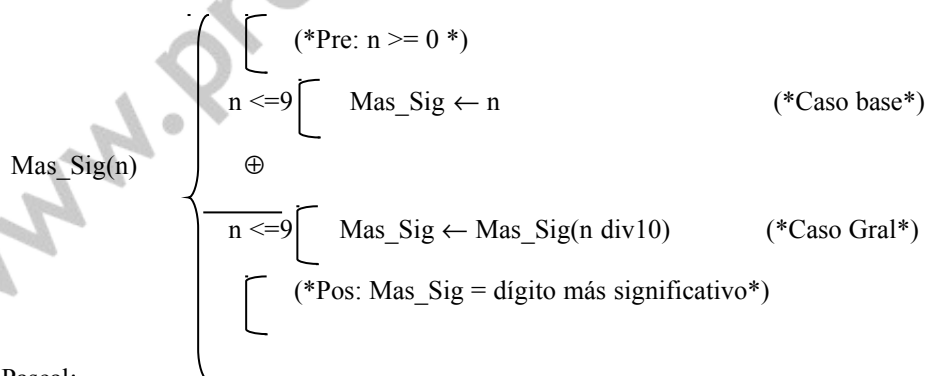
Ahora desapilamos y vamos reemplazando el resultado obtenido en la instancia posterior:

1- Desapilo Instancia 6	0	factorial(0)	1
2- Desapilo Instancia 5	1	1 * factorial(0)	1 * 1 = 1
3- Desapilo Instancia 4	2	2 * factorial(1)	2 * 1 = 2
4- Desapilo Instancia 3	3	3 * factorial(2)	3 * 2 = 6
5- Desapilo Instancia 2	4	4 * factorial(3)	4 * 6 = 24
6- Desapilo Instancia 1	5	5 * factorial(4)	5 * 24 = 120
	N	n * factorial(n-1)	factorial

Ejemplo

Dado un número natural N, encontrar el dígito más significativo (el dígito más a la derecha).

Diagrama de Llaves:



Código Pascal:

```
function Mas_sig (N: integer):integer;
begin
if N <= 9 then Mas_sig := N
else Mas_sig := Mas_sig (N div 10)
end;
```

Para $N = 25743$ es:
 Como $N \leq 9$ el algoritmo va por la rama del then llegando al caso base obteniendo $Mas_sig = 2$

2	Instancia 5
25	Instancia 4
257	Instancia 3
2574	Instancia 2
25743	Instancia 1
N	

Ahora desapilamos:

- 1- Desapilo Instancia 5
- 2- Desapilo Instancia 4
- 3- Desapilo Instancia 3
- 4- Desapilo Instancia 2
- 5- Desapilo Instancia 1

2	2
25	Mas_Sig(2)
257	Mas_Sig(25)
2574	Mas_Sig(257)
25743	Mas_Sig(2574)
N	Mas_Sig(N div 10) Mas_Sig

Cuándo usar recursividad y cuándo iteración

La potencia de la recursión reside en la posibilidad de definir un número infinito de objetos mediante un enunciado finito. De igual forma, un número infinito de operaciones de cálculo puede describirse mediante un programa recursivo finito, incluso si este programa no contiene repeticiones explícitas. Los algoritmos recursivos son apropiados principalmente cuando el problema a resolver, o la función a calcular, o la estructura de datos a procesar, están ya definidos recursivamente.

Hay varios factores a considerar en la decisión sobre si usar o no una solución recursiva en un problema. La solución recursiva puede necesitar un considerable gasto de memoria para las múltiples llamadas al procedimiento o función, puesto que deben almacenarse las direcciones de vueltas y copias de las variables locales y temporales. Si un programa debe ejecutarse frecuentemente (como un compilador) y/o debe ser muy eficiente, el uso de programación recursiva puede no ser una buena elección.

Sin embargo, para algunos problemas una solución recursiva es más natural y sencilla de escribir para el programador. Además la recursividad es una herramienta que puede ayudar a reducir la complejidad de un programa ocultando algunos detalles de la implementación. Conforme el costo del tiempo y el espacio de memoria de las computadoras disminuye y aumenta el costo del tiempo del programador, puede ser útil usar soluciones recursivas en estos casos.

En general, si la solución no recursiva no es mucho más larga que la versión recursiva, usar la no recursiva. De acuerdo a esta regla, los ejemplos que hemos visto no son buenos ejemplos de programación recursiva aunque sirven para ilustrar cómo comprender y escribir procedimientos y funciones recursivas, sería más eficiente y sencillo escribirlas iterativamente.

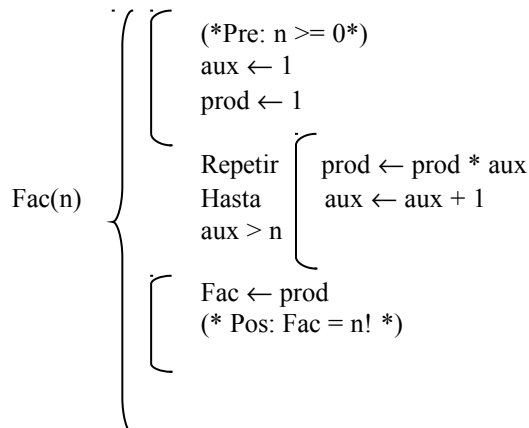
La iteración y la recursividad cumplen con el mismo objetivo: ejecutar un bloque de sentencias n veces o que con una condición de fin adecuada lo termine. Es importante distinguir las diferentes instancias de la ejecución, los valores de las variables determinan la instancia particular que está siendo resuelta en ese momento. Es necesario que la ejecución del bloque termine, es decir, el mismo bloque debe contener la condición de corte que permita que cualquier instancia menor del problema pueda ser resuelta directamente. Entonces, cómo decir cuál alternativa es más adecuada?. A veces la definición del problema no induce ninguna estructura de control en particular, recién al plantear un método de resolución es posible elegir entre iteración y recursividad. Esta decisión tiene que ver con la costumbre que tenga el programador al plantear sus algoritmos y, una vez planteado un método de resolución, se busca cuál herramienta conviene más.

Hay problemas que aparecen más naturales para la iteración y para otros para la recursividad, por ejemplo, en problemas con contadores, sumatorias y productorias lo natural es la iteración, pero en problemas en los que distinguimos un caso base y uno general, como factorial, Fibonacci, MCD lo natural es la recursión. Pero existe otro factor muy importante a tener en cuenta: la eficiencia. Veamos el factorial sin recursión:

Algoritmos & Diagramación

www.profmatiasgarcia.com.ar

Diagrama de Llaves:

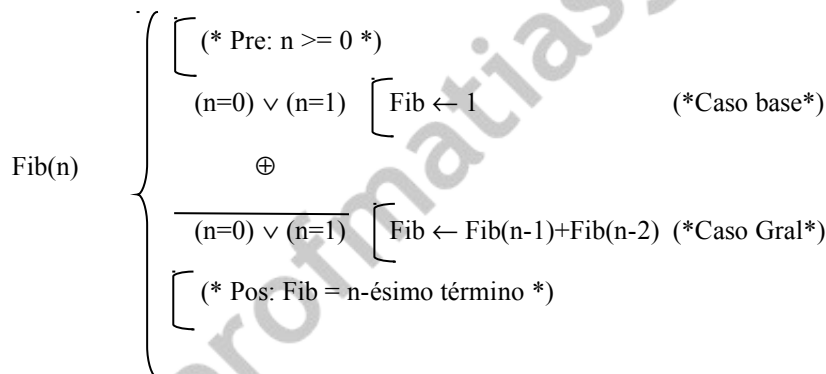


La cantidad de iteraciones o de llamadas recursivas respectivamente depende del tamaño de n , el tiempo de ejecución de ambas no varía sustancialmente, pero la versión iterativa reserva tres lugares en memoria: para aux, prod y Fac, en cambio la recursión guarda memoria para cada instancia y, si n es grande, el espacio ocupado será mucho más grande.

Hay problemas altamente complejos en lo que es casi imposible aplicar iteración.

Ejemplo

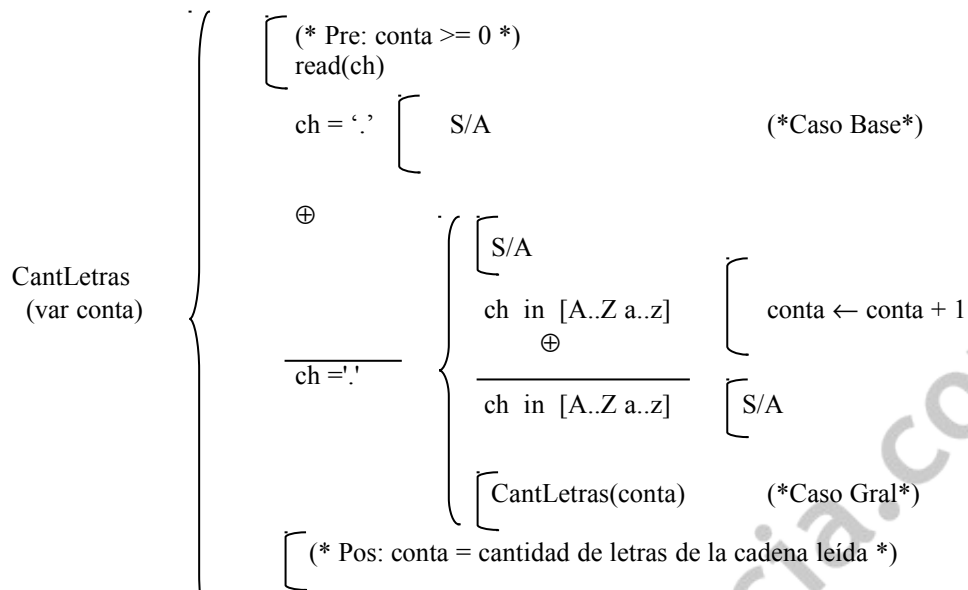
La serie de Fibonacci es: 1,1,2,3,5,8,13,21,34,... donde $t_0=1$, $t_1=1$, $t_2=2$,... $t_i = t_{i-1}+t_{i-2}$...
Veamos ahora el cálculo del n -ésimo término de Fibonacci:



Ejemplo

El Máximo Común Divisor entre dos enteros A y B se define formalmente así:

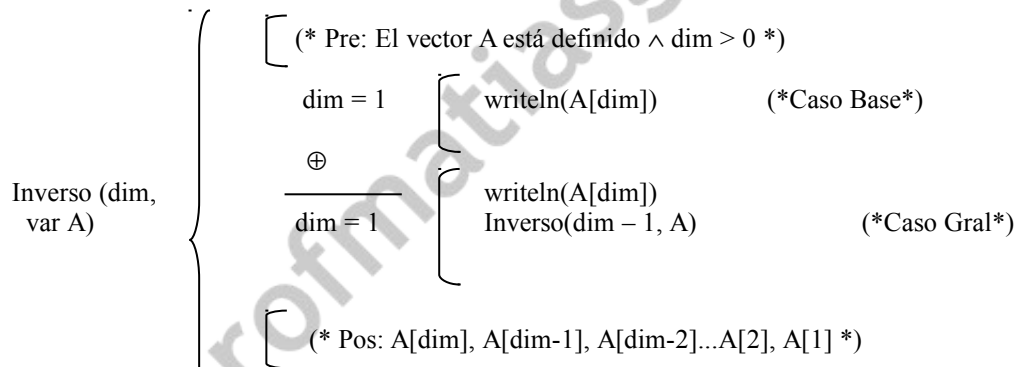
$$\text{MCD}(M,N) = \begin{cases} \text{MCD}(N,M) & \text{si } M < N \\ \text{MCD}(N, M \bmod N) & \text{si } M \text{ es divisible por } N: M \bmod N = 0 \\ & \text{en cualquier otro caso} \end{cases}$$



Notemos que en este procedimiento se pasa conta como parámetro variable. Cómo queda la pila de recursión?

Ejemplo

Imprimir los elementos de un vector en orden inverso con un procedimiento recursivo.

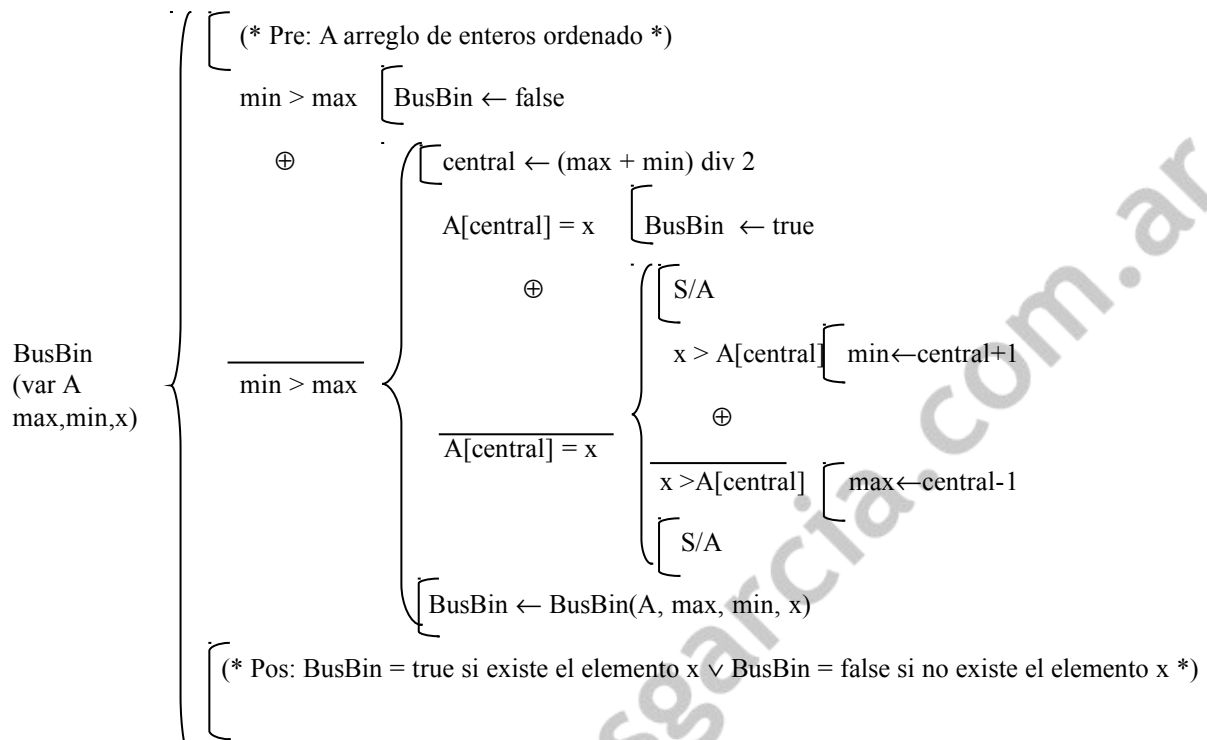


Recomendamos realizar la traza de este procedimiento y analizar qué sucedería si A fuera un parámetro valor.

Ejemplo

Dado un arreglo A de enteros ordenado de forma creciente, vamos a desarrollar el diagrama de llaves para la función de Búsqueda Binaria que realiza la búsqueda de un elemento clave en A de forma recursiva.

Diagrama de Llaves



Problemas Típicamente Recursivos

Se utilizan algoritmos recursivos generalmente en:

- recorrido de árboles
- análisis de gramáticas y lenguajes
- exploración y evaluación de expresiones aritméticas
- juegos en los que un movimiento se puede dar en función de otro de menor nivel o complicación.
- métodos de ordenamiento más eficientes
- Computación gráfica

Ejemplo

Torres de Hanoi

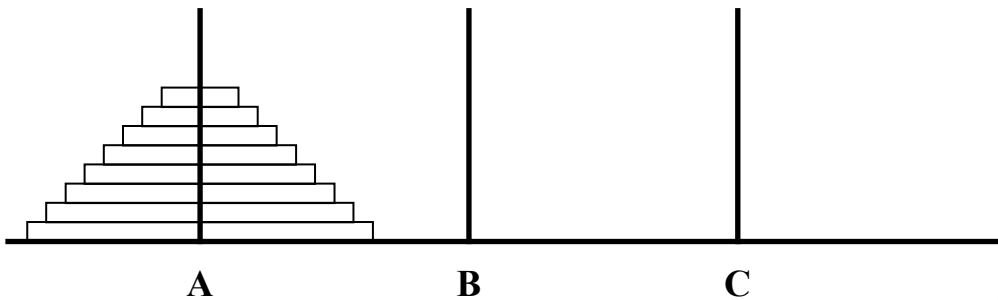
Se dan tres barras verticales y n discos de diferentes tamaños. Los discos pueden apilarse en las barras formando "torres". Los discos aparecen inicialmente en la primera barra en orden decreciente y la tarea es mover los n discos de la primera barra a la tercera de manera que queden ordenados de la misma forma inicial. Las reglas a cumplir son:

En cada paso se mueve exactamente un disco desde una barra a otra.

En ningún momento pueden situarse un disco encima de otro más pequeño.

Algoritmos & Diagramación

www.profmatiasgarcia.com.ar



Vamos a intentar resolverlo probando con 1 ó 2 discos:

1 disco

Mover el disco 1 de A a C

2 discos

Mover el disco 1 de A a B

Mover el disco 2 de A a C

Mover el disco 1 de B a C

Veamos qué pasa con 3 y 4 discos:

3 discos

Mover el disco 1 de A a C

Mover el disco 2 de A a B

Mover el disco 1 de C a B

Mover el disco 3 de A a C

Mover el disco 1 de B a A

Mover el disco 2 de B a C

Mover el disco 1 de A a C

4 discos

Podemos, partir de los movimientos para 3 discos, resolver así: pasamos todos los discos de A a B menos el disco base usando la barra C como auxiliar. Luego movemos la base a C y volvemos a repetir los movimientos pasando los discos de la barra B a C usando A como auxiliar

Mover el disco 1 de A a B

Mover el disco 2 de A a C

Mover el disco 1 de B a C

Mover el disco 3 de A a B

Mover el disco 1 de C a A

Mover el disco 2 de C a B

Mover el disco 1 de A a B

Mover el disco a de A a C (Movemos el disco base)

Mover el disco 1 de B a C

Mover el disco 2 de B a A

Mover el disco 1 de C a A

Mover el disco 3 de B a C

Mover el disco 1 de A a B

Mover el disco 2 de A a C

Mover el disco 1 de B a C

Algoritmos & Diagramación

www.profmatiasgarcia.com.ar

Intentemos aplicar el mismo razonamiento para 8 discos. Podríamos mover 7 discos de A a B, pasar el disco 8 a C y luego mover los 7 discos de B a C. Para mover los 7 discos aplicamos lo mismo: mover 6 discos de B a A, pasar el disco 7 de C a B y luego mover los 6 discos de A a B. Podemos seguir reduciendo el problema hasta llegar a 1 que es trivial, esto es, el caso base es $n=1$ y el procedimiento Pascal recursivo sería:

```
Procedure Hanoi ( n: tipodiscos; Torre_A, Torre_C, Torre_B: tipotorre);  
begin  
  if n = 1 then MoverDiscoBase(Torre_A, Torre_C)  
    else begin  
      Hanoi(n-1, Torre_A, Torre_B, Torre_C);  
      MoverDiscoBase(n, Torre_A, Torre_C);  
      Hanoi(n-1, Torre_B, Torre_C, Torre_A)  
    end  
end;
```

El procedimiento resultante es sencillo y pensar el problema de manera iterativa no hubiese sido natural de acuerdo al razonamiento efectuado además de la complejidad de programación que originaría.