IES CHAN DO MONTE
C.S. de Desarrollo de Aplicaciones Multiplataforma
Modulo Acceso a datos

ANEXO:

Java: RECURSIVIDAD

Un método es recursivo si contiene invocaciones a sí mismo.

Un gran número de problemas se plantean de forma recursiva. Esto significa que su solución se apoya en la solución del mismo problema pero para un caso más fácil.

- El concepto de recursividad no es nuevo. En matemáticas, es frecuente definir conceptos en términos de sí mismos. Por ejemplo la siguiente definición de factorial de un número positivo n es recursiva:

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

Donde la definición se realiza por un lado para un caso que denominamos base (n = 0) y por otro para un caso general (n > 0) cuya definición es claramente recursiva, por que estamos definiendo el factorial de *n* como el producto de *n* por el factorial de *n-1*, si *n* es distinto de cero. El concepto que definimos (factorial) aparece en la propia definición. Si queremos calcular, por ejemplo, el factorial de 4 con la definición anterior procederíamos del siguiente modo:

4!= 4X3!= 4×3× 2!= 4× 3× 2×1!= 4× 3× 2×1× 0!= 4× 3× 2×1×1 = 24

Básicamente, un problema **podrá resolverse de forma recursiva** si es posible expresar una solución al problema (algoritmo) en **términos de él mismo**, es decir, para **obtener la solución será necesario resolver este mismo problema sobre un conjunto de datos o entrada de menor tamaño**.

Condiciones que debe cumplir un método recursivo:

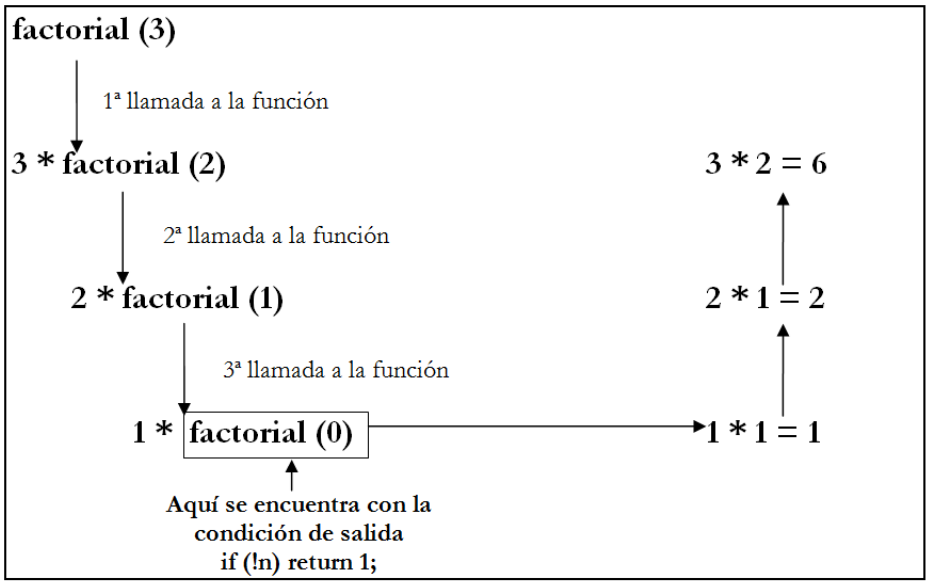
- Asegurar que **existe una condición de salida**, en la que **no se producen llamadas recursivas** (caso base)
- Asegurar que se **cubren todos los posibles casos entre el base y los no base.**
- Cada llamada, en el **caso no base (general)**, conduce a **problemas cada vez más pequeño**s que terminarán en el caso base.

Ejemplo: función recursiva factorial:

```
static long factorial (int num){
       long f=1;
        if(num>0)   f=num*factorial(num-1);
           return f;
    }
```

Funcionamento:

Representaremos ahora en un esquema el proceso de recursión al llamar, por ejemplo, a factorial(3).



Se puede comparar este proceso con el que genera la solución iterativa:

```
static long factorial (int num){
       long f=1;
        for (int i;i<=num;i++)
              f=f*num;
           return f;
    }
```

En el primer caso se hacen varias llamadas a la función, situando **en la pila los valores correspondientes** y "dejando pendientes" los cálculos intermedios, hasta que se llega a la condición de salida. En ese momento comienza el proceso inverso, sacando de la pila los datos previamente almacenados y solucionando los cálculos que "quedaron sin resolver". Obviamente esta solución es más lenta que la iterativa en la que el resultado se obtiene con una sola llamada a la función y con menos requerimientos de memoria.

- La ejecución de una función recursiva hace, en general, que éste sea llamada varias veces. Para asegurar que este proceso termina, es necesario la función recursiva definida cumpla las siguientes condiciones:

1) Hay, al menos, **una posible llamada a la función que produce un resultado sin provocar una nueva llamada recursiva.** A esto se le llama caso base de la recursión. En el caso de la función factorial, el caso base se produce cuando n es cero.

La siguiente definición de la función factorial no es correcta ya que no posee caso base:

```
static long factorial (int num){
       long f=1;
        f=num*factorial(num-1);
        return f;
    }
```

2) **Todas las llamadas recursivas se refieren a un caso que es más cercano al caso base**. En el caso de la función factorial, cada llamada decrementa en uno el valor del argumento, por lo que éste se va acercando a cero, que es el caso base.

Esta función presenta un caso base, sin embargo no es correcto ya que no cumple la condición 2). Cada llamada recursiva incrementa en uno el valor del argumento por lo que éste se va alejando del caso base y la recursión nunca termina.

```
static long factorial (int num){
       long f=1;
        if(num>0)   f=num*factorial(num+1);
           return f;
    }
```

3) **El <u>caso base debe acabar alcanzándose</u>**. La siguiente función cumple las condiciones 1) y 2), pero no es correcta. Si la llamamos con un argumento cuyo valor sea impar, el caso base nunca se alcanza.

```
static long factorial (int num){
       long f=1;
        if(num>0)   f=num**num-1*factorial(num-2);
           return f;
    }
```

<u>Recursividad Frente a Iteración.</u>

Cualquiera de los problemas resueltos de modo recursivo puede ser resuelto de modo iterativo. En algunos casos, como en el de la función factorial, la solución iterativa es bastante simple.

Podemos observar que, en la versión iterativa, la recursión ha sido sustituida por una iteración (un bucle for, en este caso).

- En general, en **<u>los algoritmos recursivos se suele utilizar una sentencia de selección (if),</u>** mientras que en los iterativos equivalentes se usa una sentencia de iteración (for, while, o do-while) y es necesario introducir variables locales que almacenen los resultados intermedios.

Los algoritmos **recursivos suelen ser menos eficientes** (en tiempo) que los iterativos correspondientes. Una **<u>llamada a una función recursiva produce varias llamadas más a la propia función</u>**. Cada una de estas llamadas requiere cierto tiempo para:

- Hacer la llamada a la función.
- Crear las variables locales y copiar los parámetros por valor.
- Ejecutar las instrucciones en parte ejecutivas de la función.
- Destruir las variables locales y parámetros por valor.
- Salir de la función.

Si se realizan unas cuantas llamadas recursivas a una función, puede ocurrir que el tiempo necesario para hacer las llamadas y crear y destruir variables prevalezca sobre el tiempo necesario para realizar el cómputo propiamente dicho.

Este tipo de sobrecarga aparece en todos los algoritmos recursivos. Si se utiliza una función iterativa equivalente, esta sobrecarga no aparece, ya que la función iterativa no realiza varias llamadas a sí misma.

• El hecho de que la recursividad presente cierta sobrecarga (en tiempo de ejecución del algoritmo) frente a la iteración **<u>no significa que no se deba utilizar, sino que hay que utilizarla cuando realmente sea apropiada.</u>**

- Cuando el problema a resolver es relativamente simple, suele ser igualmente fácil escribir la versión iterativa y la recursiva a éste. En este caso, es preferible utilizar la solución iterativa ya que es más eficiente.

- Sin embargo, las soluciones a ciertos problemas más complicados pueden ser mucho más fáciles si utilizamos para ello la recursividad. En este caso, la claridad y simplicidad del algoritmo recursivo debe prevalecer frente al tiempo extra que consume la recursión. Este enfrentamiento entre simplicidad y eficiencia aparece a menudo en la programación de ordenadores.

El verdadero valor de la recursividad es como herramienta para resolver problemas para los que no hay soluciones iterativas simples.

PROBLEMA DE LAS TORRES DE HANOI

Es clásica la referencia al juego comúnmente conocido como "Las Torres de Hanoi" cuando se estudia la recursividad.

"Se cuenta que una leyenda india narra la historia en la cual se dice que cuando el ser supremo creó el mundo situó en un gran templo en la sagrada ciudad de Benarés, junto al río Ganges, tres astas de diamante. En una de ellas ensartó 64 discos de oro, todos de distinto diámetro y apilados de forma que el de menor diámetro estaba arriba del todo y el mayor, como base, abajo. Los monjes del templo tenían encomendada la tarea de trasladar los discos de oro a otra de las astas pero cumpliendo obligadamente las siguientes condiciones:

- *Sólo se puede trasladar un disco cada vez.*
- *No está permitido situar un disco sobre otro de menor diámetro.*
- *La segunda asta puede utilizarse para realizar traslados intermedios.*

La leyenda termina señalando que el fin del mundo tendría lugar cuando se hubiese completado el traslado de todos los discos de un asta a otra. "

Este juego, a lo largo del tiempo, ha sido conocido con otros nombres como: "Las torres de Brama", "El juego del Prof. Lucas", etc.

"Las Torres de Hanoi" es un juego cuya solución se **simplifica mucho si se piensa como un problema recursivo**.

Se tienen 3 palos de madera, que llamaremos palo ORIGEN, AUXILIAR y DESTINO. El palo origen tiene ensartados un montón de discos concéntricos de tamaño decreciente, de manera que el disco mayor está abajo y el menor arriba.

El problema consiste en mover los discos del palo origen al destino respetando las siguientes reglas:

– Sólo se pueden mover los discos de un palo a otro.

– Sólo se puede mover un disco cada vez.

– No se puede poner un disco encima de otro más pequeño.

Se quiere diseñar un programa que recibiendo un valor entero N y escriba la secuencia de pasos necesarios para resolver el problema de las torres de Hanoi para N discos.

Solución

Para ver el problema de forma recursiva tenemos que pensar que la solución para N = 1 es trivial, mientras que para un valor mas alto de N se puede reducir en cada ocasión en un valor N − 1, descomponiendo la solución en 3 fases:

1. Mover los N−1 discos superiores del palo origen al palo auxiliar, utilizando el palo destino como palo intermedio.

2. Mover el disco que queda del palo origen al destino.

3. Mover los N − 1 discos del palo auxiliar al palo destino, utilizando el palo origen como palo intermediario.

Si N = 1, el problema se resuelve inmediatamente sin mas que mover el disco del palo origen al destino.

Para representar estos pasos en el ordenador, planteamos un procedimiento recursivo que recibe cuatro parámetros:

El número de discos a mover.
El número del palo a tomar como origen desde donde moverlos.
El numero del palo a tomar como destino hacia el que moverlos.
El numero del palo a usar como auxiliar.

(1 para el palo origen, 2 para el palo auxiliar y 3 para el destino)

```
Procedimiento  Mueve (N:entero, origen:entero, auxiliar:entero, destino:entero)
 Inicio
   SI N = 1 ENTONCES
      Visualizar "Mueve un disco del palo", origen, "al", destino
        SINO
      Mueve(N-1, origen, destino, auxiliar)
      Visualizar ("Mueve un disco del palo ", origen, " al ", destino)
      Mueve(N-1, auxiliar, origen, destino)
   FINSI
 Fin
```

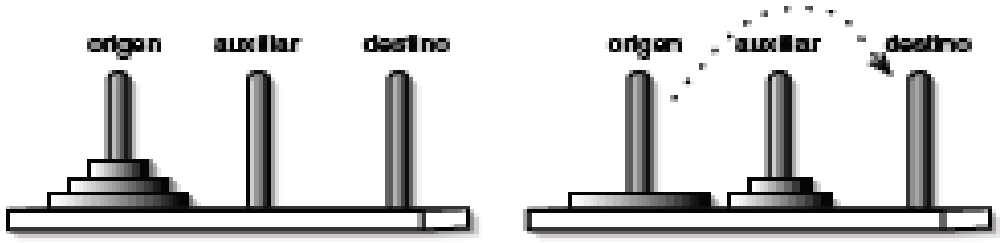
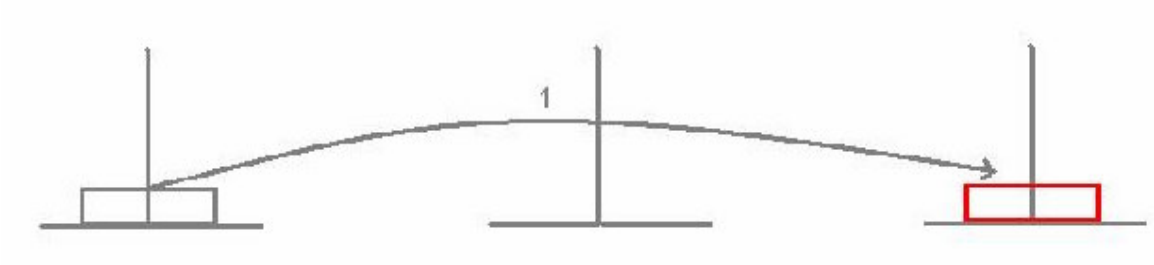


Figura 6: Mover n discos se reduce a mover $n-1$ al palo libre, mover después el que queda al destino, y volver a plantear recursivamente el mover, pero ahora, $n-1$ discos.

Vamos a hacer un seguimiento de este algoritmo para los casos en los que el número de discos sea 1 y 2.

- Para n=1 nos encontramos en el caso base de la función Mueve, la llamada sería:

  mueve(1,1,2,3) : 1→3 , esto quiere decir que moveríamos el disco del palo 1 al 3. Luego realizamos un solo movimiento.



- Para n=2 realizamos 3 movimientos que se corresponde con el número de llamadas que realizamos a la función mueve, la secuencia de llamadas que realiza lo mostramos en el siguiente árbol:

  Mueve (nº de discos, origen, auxiliar, destino)