Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Apunte: Recursividad

1- Concepto:

En las actividades cotidianas podemos observar manifestaciones de este tipo, por ejemplo en las muñecas rusas que se contienen una dentro de otra más grande, y en una amplia variedad de planteos matemáticos.

En la actualidad casi todos los lenguajes de programación soportan recursividd, esto es brindan al programador una herramienta para resolver ciertos tipos de problemas reduciendo la complejidad.

La recursividad es una técnica de programación que nos permite que un bloque de instrucciones se ejecute n veces sin utilizar estructuras repetitivas, por lo que en ocasiones reemplaza a dichas estructuras repetitivas.

Existe recursividad cuando en un lenguaje de programación, dentro del código de implementación de un método, existe al menos una llamada a sí mismo directa o indirectamente.

Cuando un método se llama a sí mismo, se asigna espacio en la pila para las nuevas variables locales y parámetros. Al volver de una llamada recursiva, se recuperan de la pila las variables locales y los parámetros antiguos y la **ejecución se reanuda en el punto siguiente al de la llamada al módulo**.

Ejemplo: Mostrar un mensaje por pantalla repetidas veces.

```
MODULO repetir () RETORNA Ø
    repetir()  (* llamada recursiva *)
    ESCRIBIR....
FIN MODULO
```

```
    static void repetir() {
        repetir();
    }

    public static void main(String[] ar) {
         repetir();
    }
```

El método repetir es recursivo porque dentro de se llama a sí mismo. El problema de este método es que tiene sucesivas llamadas a sí mismo y no corta su ejecución nunca. Por lo cual, cuando se ejecuta este programa se bloqueará y generará una excepción: "Exception in thread "main" java.lang.StackOverflowError"

Analicemos como funciona:

- Primero se ejecuta la función main, luego de crear un objeto llamamos a la función repetir.
- Hay que tener en cuenta que cada vez que se llama a una función se reservan 4 bytes de la memoria que se liberarán cuando finalice su ejecución.
- La primera línea de la función llama a la función repetir, es decir que se reservan 4 bytes nuevamente. Se ejecuta nuevamente una instancia de la función repetir y así sucesivamente hasta que la pila estática se

colme (que tiene un tamaño muy grande) y se cuelgue el programa.

En los algoritmos recursivos la invocación directa o indirecta a sí mismo depende en general de alguna condición que actúa como condición de corte que llamamos caso base o trivial, además se debe asegurar de llegar al caso base en sucesivas llamadas.

De esta manera un algoritmo recursivo consta de:

- Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se activa cuando se cumple cierta condición,
- Un caso general que es el que vuelve a invocar al algoritmo con un subcaso del mismo garantizando llegar al caso base o trivial.

2-Diseño de un algoritmo recursivo

La ejecución de un algoritmo recursivo implica que una nueva version del algoritmo comienza a ejecutarse, por lo tanto debe organizar la manera de guardar determinada información de cada llamada. Esta información debe estar organizada de forma que al terminar cada una de las versiones que se estan ejecutando, se devuelva la información correcta a la invocación o instancia del algoritmo inmediatamente anterior.

El compilador gestiona esta tarea a través de una estructura tipo pila: lo último que entra es lo primero que sale. Esta estructura pila que se va generando, supone una mayor ineficiencia de tiempo y de espacio en memoria, que depende sobre todo, del nivel de recursividad alcanzado.

El nivel de recursividad es el número de versiones del programa recursivo que se estan ejecutándo en forma simultánea (no debe confundirse con el número de llamadas recursivas).

Un algoritmo contiene, ademas de variables locales, argumentos o parámetros. Mediante éstos se transmiten datos en las llamadas a los algoritmos invocados. Además de los parámetros, la nueva invocación recursiva del algoritmo debe guardar la dirección de retorno en el algoritmo que invoca.

Una vez ejecutado, se devuelve el control al algoritmo en el punto en que se realizo la llamada recursiva.

Las llamadas recursivas se gestionan con el uso de pilas en las cuales se guarda:

- Cada parámetro o argumento de llamada.
- Cada variable local.
- La dirección o punto de retorno al programa llamador.

En cada llamada al programa recursivo, los valores actuales de los argumentos, variables locales y dirección de retorno se introducen en las pilas correspondientes. Cada vez que se produce un retorno del algoritmo de una llamada recursiva se restauran, desde las cimas de las pilas respectivas, los valores de los argumentos y variables locales. La dirección de retorno se puede suponer que se guarda almacenando la línea de la siguiente sentencia a la de llamada y el nivel de profundidad en la recursión.

Para construir una traza que nos permita verificar el correcto funcionamiento de un algoritmo recursivo seguimos el siguiente proceso:

Se crea una pila para cada argumento, una pila para cada variable local, y una pila para la dirección de retorno en la que se coloca la proxima invocación al algoritmo.

En cada llamada recursiva tenemos que:

Apilar el valor actual de cada argumento o parámetro.

Apilar el valor actual de cada variable local.

Apilar la direccion de la siguiente sentencia a la llamada actual.

Al analizar cada ejecución recursiva del algoritmo:

Si la pila de direcciones esta vacía, debe devolver el control al programa de llamada.

Desapilar los valores de las variables locales y los argumentos de las cimas.

Devolver la ejecución a la sentencia entrada de la pila de direcciones.

Establecemos la dirección de retorno no como una dirección propiamente dicha, sino como el código del algoritmo que queda pendiente y que deberá ir ejecutándose en la composición del resultado.

La estructura de pila que utiliza la memoria es la misma que utilizamos al construir la traza, es decir que al componer se van devolviendo los resultados a la invocación inmediatamente superior en la traza.

Para poder construir cualquier rutina recursiva se deben seguir los siguientes pasos:

- Definición exacta del problema.
- Determinar el tamaño del problema completo a resolver. Así se determinarán los valores de los parámetros en la invocación inicial.
- Resolver el caso base .
- Resolver el caso general, en términos de un caso más pequeño del mismo problema (una llamada recursiva).

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

Los pasos claves para el diseño de una rutina recursiva son:

- DESCOMPOSICION: >Cómo sería un subproblema idéntico al problema planteado?
- COMPOSICION: >Cómo se combinan las soluciones de los subproblemas para formar la solución del problema original?
- CASO BASE O CONDICION DE CORTE: >Cuál es el subproblema que puede resolverse fácilmente sin descomposición?

Ejemplo: Cálculo de factorial de un numero n = producto de los n primeros numeros naturales.

Solucion iterativa

n! = 1 * 2 * ... * (n-1) * n

Sin embargo, hay un patrón recursivo

n! = 1 * 2 * ... * (n-1) * n

n! =(n-1)! * n

```
MODULO factorial (ENTERO n) RETORNA ENTERO
   ENTERO f    (* Entrada: nùmero entero positivo*)
    Si (n = 0)ENTONCES
         f = 1
    SINO
          f = n * factorial (n-1)
     FIN SI
     RETORNA f
FIN MODULO
```

Ejemplo: Un número natural n que se encuentra en el sistema de numeracion decimal puede ser expresado utilizando ceros y unos solamente mediante el denominado sistema binario.

Para obtener la representación binaria de n, deben calcularse los restos de dividir sucesivamente n por 2 hasta

que el dividendo sea 0. Los restos obtenidos (listados en orden inverso) son la representacion binaria de n.

Por ejemplo para calcular la representacion binaria de n = 28, se deben calcular los siguientes restos:

28 dividido 2 da 14 y resto 0

14 dividido 2 da 7 y resto 0

7 dividido 2 da 3 y resto 1

3 dividido 2 da 1 y resto 1

1 dividido 2 da 0 y resto 1

Resultado de 28 en binario es 1110

Las operaciones de division (div) y modulo (mod) corresponden a la division entera y al resto de la division entera respectivamente.

```
MODULO pasarBinario(ENTERO n) RETORNA ENTERO
  ENTERO f  (* Entrada: nùmero entero decimal positivo*)
   Si (n >= 1)ENTONCES
      f = pasarBinario (n/2)*10 + (n %2)
   SINO
      f = n
    FIN SI
   RETORNAR f
FIN MODULO
```

Traza:

Descomposicion

	n		binario	
pasarABinario(28)		28	(pasarABinario(14)*10)+ (28 mod 2)	
pasarABinario(14)		14	(pasarABinario(7)*10)+ (14 mod 2)	
pasarABinario(7)	7	(pasarABinario(3)*10)+ (7 mod 2)		
pasarABinario(3)	3	(pasarABinario(1)*10)+ (3 mod 2)		
pasarABinario(1)	1		1	Caso base

Composicion

	n		binario		
pasarABinario(28)		28	(1110*10) + (28 mod 2)		=11100
pasarABinario(14)		14	(111*10)+ (14 mod 2)		=1110 *
pasarABinario(7)	7	(11*10)+ (7 mod 2)		=111 *	
pasarABinario(3)	3	(1*10)+ (3 mod 2)			=11 *
pasarABinario(1)	1	1			

3-Recursividad con arreglos

En los arreglos el caso base se da generalmente al llegar al final del arreglo, o en la posición inicial. Pongamos como ejemplo

Programar un algoritmo recursivo que permita sumar los elementos de un vector.

```
MODULO sumaArreglo(ARREGLO ENTERO v, ENTERO n) RETORNA ENTERO
   ENTERO aux    (* Entrada: nùmero entero positivo*)
    Si (n = 0)ENTONCES
         aux = v[n]
```

```
    SINO
          aux =  sumaArreglo (v, n-1) + v[n]
     FIN SI
     RETORNA aux
FIN MODULO
```

Código en Java

```
int sumaVec1(int v [], int n)
{int aux
if (n == 0)
     aux =  v [n];
else
     aux = sumaVec1(v, n - 1) + v [n];
return aux
}
```

```
int sumaVec2(int v [], int n)
{
int aux =0;
   if (n < v.length)
     aux = sumaVec2(v, n + 1) + v [n];
    return aux{i
}
```

La diferencia de estos dos códigos es cómo es el llamado inicial

```
  private int[] arreglo = new int[6]; //Creación de la matriz

   public static void main(String[] args) {
        llenarArreglo(arr);
        System.out.println ("la suma de sus valores es : ",
           sumaVec1(arr,arr.length ) , " ", sumaVec2(arr,0);
     }
```

3-Tipos de Recursividad

Existen distintos tipos de recursividad: Simple, múltiple, anidada, indirecta.

1. Simple: Aquella en cuya definicion solo aparece una llamada recursiva. En esta categoría se encuentran los algoritmos factorial, pasarABinario y sumaVec presentados anteriormente.
2. Múltiple: Se da cuando hay mas de una llamada a sí misma dentro del cuerpo de la función, resultando mas difícil de hacer de forma iterativa. En esta categoría se encuentran el algoritmo de Fibonacci. la sucesion de Fibonacci.

   La sucesion de Fibonacci es la siguiente sucesion infinita de numeros naturales: 0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89; 144; : : : donde el primer elemento es 0, el segundo es 1 y cada elemento restante es la suma de los dos anteriores. El modelo matemático es:

fibonacci (n)	0	si n = 0
	1	si n =1
	fibonacci (n-1) + fibonacci(n-2)	si n > 1

```
MODULO fibonacci(ENTERO n) RETORNA ENTERO
   ENTERO aux    (* Entrada: número entero positivo*)
    Si (n = 1  o  n = 2)ENTONCES
         aux = 1
    SINO
          aux =  fibonacci(n-1) + fibonacci (n-2);
    FIN SI
    RETORNA aux
FIN MODULO
```

```
int fibonacci(int n){
   int aux;
   if(n==1 || n==2) aux = 1;
   else aux = fibonacci(n-1)+fibonacci(n-2);
   return aux;
}
```

3. Anidada: En algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma. En esta categoría colocamos a la función de Ackerman. La función de Ackerman es una función recursiva que toma dos números naturales como argumentos y devuelve un único número natural. Como norma general se define como sigue:

   El modelo matemático es:

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

Código

Esta función es aparentemente simple y se utiliza para examinar el comportamiento del compilador . La Función de Ackermann es una función altamente recursiva que no sirve para nada útil, ya que no calcula nada de importancia, la función de Ackermann a menudo se utiliza para probar los niveles de optimización de un compilador.

```
MODULO ack(ENTERO m, ENTERO n) RETORNA ENTERO
   ENTERO aux    (* Entrada: dos números enteros positivos*)
    Si (m = 0)ENTONCES
         aux = n-1
    SINO Si (m > 0 Y n == 0)
              aux =  ack(m-1,1)
         SINO
               aux =  ack(m-1, ack(m, n-1))
    FIN SI
    RETORNA aux
FIN MODULO
```

```
long ack(int m, int  n){
   int  aux;
   if(m==0 )
       aux = n-1;
   else
        if(m> 0 &&  n==0)
            aux =  ack(m-1,1);
       else
            aux =  ack(m-1, ack(m, n-1));
  return aux;
}
```

4. Cruzada o indirecta: Son algoritmos donde una funcion ejecuta una llamada a sì misma de forma

indirecta, esto es a través de otras funciones. Las operaciones par e impar que describimos a continuacion, utilizan la recursion cruzada para comprobar la paridad de un numero entero positivo.

```
public boolean par(int n){
     if(n==0) return true;
     else return impar(n-1);
}

 public boolean impar(int n){
      if(n==0) return false;
      else return par(n-1);
   }
```

4-Cuando usar iteración y cuando recursividad?

Hay varios factores a considerar en la decisión sobre si usar una solución recursiva o una iterativa en un problema. La solución recursiva puede necesitar un considerable gasto de memoria para las multiples llamadas al procedimiento o función, puesto que deben almacenarse las direcciones de vueltas y copias de las variables locales y temporales.

Sin embargo, para algunos problemas una solución recursiva es más natural y sencilla de escribir para el programador. 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.

Se recomienda en general, utilizar la solución no recursiva cuando en el planteo no se reconocen ventajas en lo que se refiere al uso de los resultados parciales. Aunque cuando la complejidad de la solución no recursiva se torna inmanejable, es preferible utilizar un algoritmo recursivo como en los algoritmos de recorrido de árboles y grafos.

A veces la definición del problema no induce a ninguna estructura de control en particular, recién al plantear un método de resolución es posible elegir entre iteración y recursividad.

Hay problemas cuyas características los hacen naturales para la iteración y 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 el cálculo del término de bonacci lo natural es la recursión. Existe otro factor muy importante a tener en cuenta: la eficiencia. Veamos el algoritmo factorial sin recursión:

```
static int factorialIterativo(int n){
   int prod=1;
   int cod=1;

   while(cont<= n) {
      prod = prod *cont;
      cont++;
     }
   return prod;
}
```

La traza de este algoritmo solamente corresponde a dos variables locales prod y cont y sobre esos espacios de memoria se va acumulando la productoria e incrementando el contador. Podemos ver que el uso de memoria en el algoritmo recursivo es notoriamente importante y que es más eficiente el algoritmo iterativo porque no almacena resultados parciales.

Aquí presentamos un juego que sólo puede resolverse recursivamente:

Ejemplo: El juego de ingenio Las torres de Hanoi.

Este juego consta de tres columnas y una serie de discos de distintos tamaños. Los discos estan acomodados de mayor a menor en una de las columnas. El juego consiste en pasar todos los discos de una columna a otra, respetando las siguientes reglas:

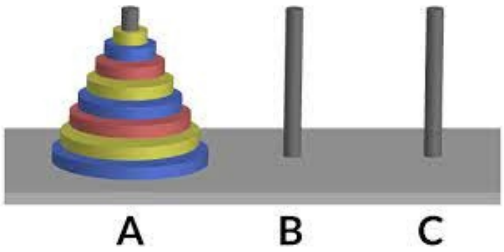
Sóolo se puede mover un disco cada vez.

Para cambiar los discos de lugar se pueden usar las tres columnas del juego; es decir que los distintos discos se pueden ir acomodando en las columnas segun convenga.

Nunca debera quedar un disco grande sobre un disco mas pequeño.

En un juego de Torres de Hanoi tenemos tres torres A, B y C donde los discos estan inicialmente en la torre A y queremos pasarlos a la torre C con las restricciones precendentes.

Vamos a intentar resolverlo probando con 1 o 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 que 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 as: 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

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.

```
MODULO mover(ENTERO disco,LETRA fuente,LETRA dest,LETRA aux) RETORNA Ø
   Si (disco = 1)ENTONCES
     moverDisco (disco, fuente, dest) //imprime disco desde fuente a dest
    SINO
         mover (disco-1,fuente, aux, destino)
         moverDisco (disco,fuente, destino)
         mover (disco-1,aux, destino, fuente)
    FIN SI
FIN MODULO
```