Capítulo 2: Recursividad.

2.1.- Introducción.

La recursividad consiste en realizar una definición de un concepto en términos del propio concepto que se está definiendo.

Ejemplos:

- Los números naturales se pueden definir de la siguiente forma: 0 es un Número natural y el sucesor de un número natural es también un número natural.

- El factorial de un número natural n, es 1 si dicho número es 0, o n multiplicado por el factorial del número n-1, en caso contrario.

- La n-ésima potencia de un número x, es 1 si n es igual a 0, o el producto de x por la potencia (n-1)-ésima de x, cuando n es mayor que 0.

En todos estos ejemplos se utiliza el concepto definido en la propia definición.

Solución de problemas recursivos:

- División sucesiva del problema original en uno o varios más pequeños, del mismo tipo que el inicial.

- Se van resolviendo estos problemas más sencillos.

- Con las soluciones de éstos se construyen las soluciones de los problemas más complejos.

O lo que es lo mismo:

1. Un problema P se puede resolver conociendo la solución de otro problema Q que es del mismo tipo que P, pero más pequeño.

2. Igualmente, supongamos que pudiéramos resolver Q mediante la búsqueda de la solución de otro nuevo problema, R, que sigue siendo del mismo tipo que Q y P, pero de un tamaño menor que ambos.

3. Si el problema R fuera tan simple que su solución es obvia o directa, entonces, dado que sabemos la solución de R, procederíamos a resolver Q y, una vez resuelto, finalmente se obtendría la solución definitiva al primer problema, P.

Ejemplos simples de recursividad.

A) Cálculo del factorial de un número, por ejemplo, 5.

5! = 5 * 4!

4! = 4 * 3!

3!= 3 * 2!

2! = 2 * 1!

DESCOMPOSICIÓN
DEL PROBLEMA

1! = 1 * 0!

SOLUCIÓN CONOCIDA O DIRECTA **0! = 1**

1! = 1 * 0! = 1

2! = 2 * 1! = 2

3! = 3 * 2! = 6

4! = 4 * 3! = 24

RESOLUCIÓN DE PROBLEMAS MÁS COMPLEJOS A PARTIR DE OTROS MÁS SIMPLES

5! = 5 * 4! = 120

B) Búsqueda de una palabra en un diccionario.

Características de los problemas que pueden ser resueltos de manera recursiva:

4. Los problemas pueden ser redefinidos en términos de uno o más subproblemas, idénticos en naturaleza al problema original, pero de alguna forma menores en tamaño.

5. Uno o más subproblemas tienen solución directa o conocida, no recursiva.

6. Aplicando la redefinición del problema en términos de problemas más pequeños, dicho problema se reduce sucesivamente a los subproblemas cuyas soluciones se conocen directamente.

7. La solución a los problemas más simples se utiliza para construir la solución al problema inicial.

Algoritmos recursivos:

diseño de la solución de un problema de manera recursiva

El algoritmo se llamará a sí mismo varias veces

¡Ojo al diseñar algoritmos recursivos!

pueden ser menos eficientes que que su solución iterativa

Algoritmo recursivo ⇒ Implementación en un lenguaje de alto nivel que permita recursividad (en nuestro caso, en Java).

2.2.- Diseño de módulos recursivos.

- Módulo M con una llamada a sí mismo: **módulo recursivo directo**.

- Módulo M con una llamada a otro F, que hace una llamada a M: **Módulo recursivo indirecto**.

Ejemplo: Implementación del factorial de un número.

```
public long factorial (long n)
  {
    if (n == 0) return 1;
    else return n * factorial(n-1);
  }
```

Dos partes principales:

8. La llamada recursiva, que expresa el problema original en términos de otro más pequeño, y

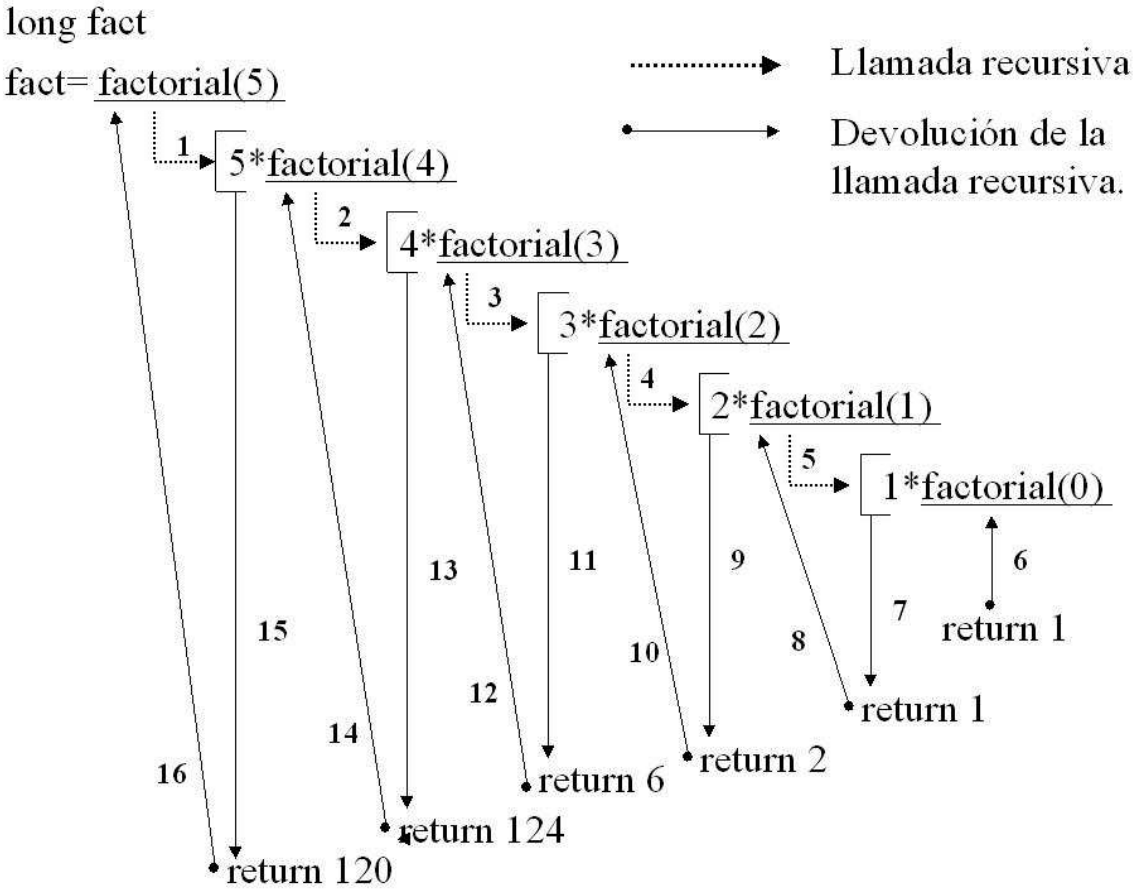
9. el valor para el cual se conoce una solución no recursiva. Esto es lo que se conoce como **caso base**: una instancia del problema cuya solución no requiere de llamadas recursivas.

El caso base

10. Actúa como condición de finalización de la función recursiva. Sin el caso base la rutina recursiva se llamaría indefinidamente y no finalizaría nunca.

11. Es el cimiento sobre el cual se construirá la solución completa al problema.

Ejemplo: Traza del factorial de 5.



Búsqueda de soluciones recursivas: cuatro preguntas básicas.

- [P1] ¿Cómo se puede definir el problema en términos de uno o más problemas más pequeños del mismo tipo que el original?

- [P2] ¿Qué instancias del problema harán de caso base?

- [P3] Conforme el problema se reduce de tamaño ¿se alcanzará el caso base?

- [P4] ¿Cómo se usa la solución del caso base para construir una solución correcta al problema original?

Ejemplo: la sucesión de Fibonacci.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- El tercer término de la sucesión se obtiene sumando el segundo y el primero. El cuarto, a partir de la suma del tercero y el segundo.

- El problema: calcular el valor del n-ésimo término de la solución, que se obtendrá sumando los términos n-1 y n-2.

Las respuestas a la preguntas anteriores serían:

- [P1] fibonacci(n) = fibonacci(n-1) + fibonacci(n-2).

- [P2] Casos bases: fibonacci(1) = 1 y fibonacci(2)=1.

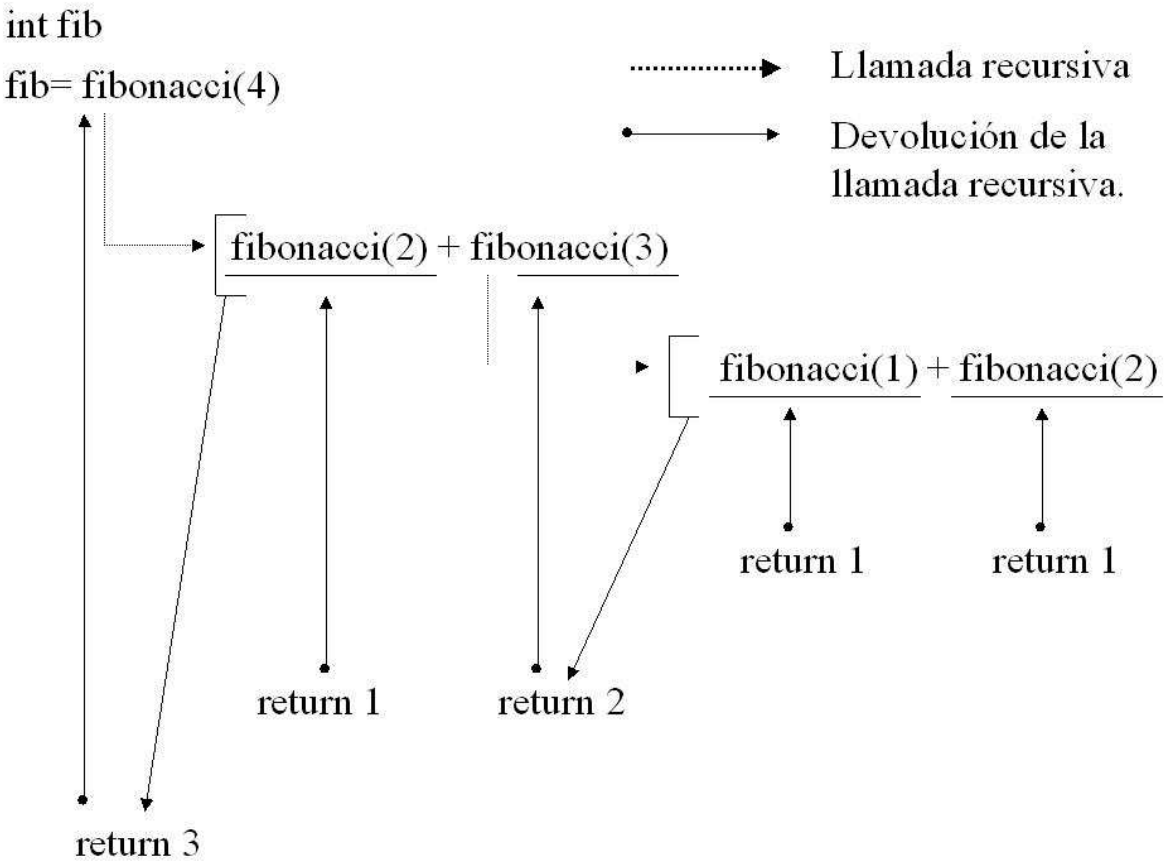
- [P3] En cada llamada a la rutina fibonacci se reduce el tamaño del problema en uno o en dos, por lo que siempre se alcanzará uno de los dos casos bases.

- [P4] fibonacci(3) = fibonacci(2) + fibonacci(1) = 1 + 1.

Se construye la solución del problema n==2 a partir de los dos casos bases.

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

Peculiaridades de esta función recursiva: más de un caso base y más de una llamada recursiva (recursión no lineal).



Ejemplo: máximo común divisor de dos números m y n.

Ejemplo de problema en el que la solución al encontrar el caso base es la propia solución del problema.

Algoritmo de Euclides:

Si n divide a m,
entonces MCD(m, n)=n,
Si no, MCD(m, n) = MCD(n, m % n),

Soluciones a las cuatro preguntas:

- [P1] Es evidente que MCD(m, n) ha sido definida en términos de un problema del mismo tipo, pero justifiquemos que MCD(m, n % m) es de tamaño menor que MCD(m, n):

12. El rango de m % n es 0, ..., n-1, por tanto el resto es siempre menor que n.

13. Si m > n, entonces MCD(n, m % n) es menor que MCD(m, n).

14. Si n > m, el resto de dividir m entre n es m, por lo que la primera llamada recursiva es MCD(n, m mod n) es equivalente a MCD(n, m), lo que tiene el efecto de intercambiar los argumentos y producir el caso anterior.

- [P2] Si n divide a m, si y sólo si, m mod n = 0, en cuyo caso MCD(m, n) = n, por tanto, se conseguirá el caso base cuando el resto sea cero.

- [P3] Se alcanzará el caso base ya que n se hace más pequeño en cada llamada y además se cumple que 0 * m mod n * n-1.

- [P4] En este caso, cuando se llega al caso base es cuando se obtiene la solución final al problema, por lo que no se construye la solución general en base a soluciones de problemas más simples.

```
public int mcd (int m, int n)
{
  if (m % n == 0) return n;
  else return (mcd(n, m % n));
}
```

Ejemplo: conteo de ocurrencias de un valor en un vector.

Dado un vector de n enteros, el problema a resolver recursivamente consiste en contar el número de veces que un valor dado aparece en dicho vector.

Respuestas:

- [P1] Si el primer elemento = valor buscado, entonces la solución será 1 más el número de veces que aparece dicho valor en el resto del vector.

  Si no, la solución al problema original será el número de veces que el valor se encuentra en las posiciones restantes

- [P2] El vector a está vacío =>ocurrencias en el vector= 0.

- [P3] Cada llamada recursiva se reduce en uno el tamaño del vector, por lo que nos aseguramos que en N llamadas habremos alcanzado el caso base.

- [P4] Cuando se encuentra una coincidencia se suma uno al valor devuelto por otra llamada recursiva. El retorno del control de las sucesivas llamadas comenzará inicialmente sumando 0 cuando nos hayamos salido de las dimensiones del vector.

Función en Java que implementa la solución recursiva:

Parámetros:

- el tamaño del vector (n),
- el propio vector (vector),
- el valor a buscar (objetivo) y
- el índice del primer elemento del vector a procesar (primero).

```
public int contarOcurrencias (int n, int[] vector, int objetivo,
int primero)
{
 if (primero > n-1) return 0;
else
   {
    if (vector[primero] == objetivo)

       return(1+contarOcurrencias(n,vector,objetivo,primero+
1));
    else return(contarOcurrencias(n,vector, objetivo,
primero+1));
   }
}
```

Ejemplo: combinaciones sin repetición de n objetos tomados de k en k.

Cuántas combinaciones de k elementos se pueden hacer de entre un total de n (combinaciones sin repetición de n elementos tomados de k en k).

Respuestas:

- [P1] El cálculo del número de combinaciones de n elementos tomados de k en k se puede descomponer en dos problemas:

  a) calcular el número de combinaciones de n-1 elementos tomados de k en k y

  b) el número de combinaciones de n-1 elementos tomados de k-1 en k-1.

El resultado del problema inicial:

$$ncsr(n, k) = ncsr(n- 1, k-1) + ncsr(n-1, k).$$

n personas y queremos determinar cuántos grupos de k personas se pueden formar de entre éstas.

4 personas (A, B, C y D), grupos de 2

$\Downarrow$

¿ncsr(4,2)=ncsr(3,1)+ ncsr(3,2)?

Suma de:

- los grupos que se pueden hacer de un tamaño k sin contar a la persona A: ncsr(n-1,k) :número de comités que se pueden hacer, de tamaño 2, pero sin A ncsr(3,2): {B,C}, {B,D} y {D,C}.

- los grupos que se puedan realizar incluyendo a esa misma persona, ncsr(n-1, k-1): ncsr(3,1): {B}, {C} y {D}.(contamos todos los que se pueden hacer con una persona menos y luego le añadimos esa persona).

Solución final: 3 + 3 = 6

•[P2] Casos bases:

Si k > n, entonces ncsr(n,k) =0 (no hay elementos suficientes).

Si k = 0, entonces este caso sólo ocurre una vez (ncsr(n,0)=1).

Si k = n, entonces ncsr(n,n)=1.

•[P3] 1ª llamada recursiva: restamos uno al número de elementos, y en la 2ª, al número de elementos y al tamaño de la combinación.

•[P4] Apenas se haya encontrado un caso base, uno de los sumandos tendrá un valor. Cuando el otro sumando alcance a un caso base, se podrá hacer la suma y se devolverá a un valor que irá construyendo la solución final conforme se produzcan las devoluciones de control.

```
public int ncsr(int n, int k)
{
  if (k > n) return 0;
  else if (n == k || k == 0) return 1;
       else return ncsr(n-1, k) + ncsr(n-1, k-1);
}
```

2.3.- La pila del ordenador y la recursividad.

La memoria de un ordenador a la hora de ejecutar un programa queda dividida en dos partes:

- la zona donde se almacena el código del programa y
- la zona donde se guardan los datos: pila (utilizada para llamadas recursivas).

Programa principal llama a una rutina M,

⇓

se crea en la pila de un registro de activación o entorno, E

⇓

almacena constantes, variables locales
y parámetros formales.

⇓

Estos registros se van apilando conforme se llaman sucesivamente desde una función a otras

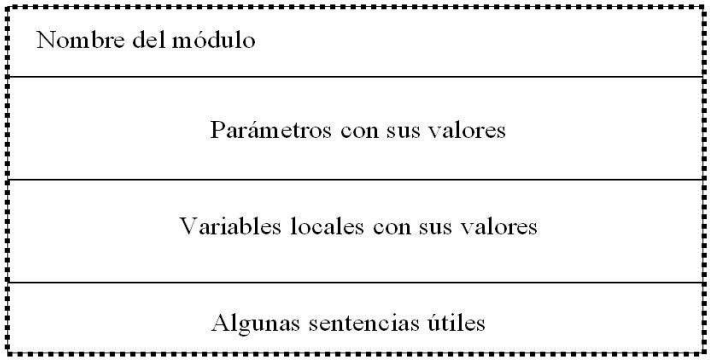
⇓

Cuando finaliza la ejecución, se va liberando el espacio

Profundidad de la recursión: número de registros de activación en la pila en un momento dado.

Problema:
Si profundidad es muy grande => desbordamiento pila.

Representación gráfica del registro de activación:



Ejemplo de evolución de la pila: impresión de palabras en sentido contrario al leído.

Función recursiva: imp_OrdenInverso.
Parámetros: número de palabras que quedan por leer (n).

Respuestas:

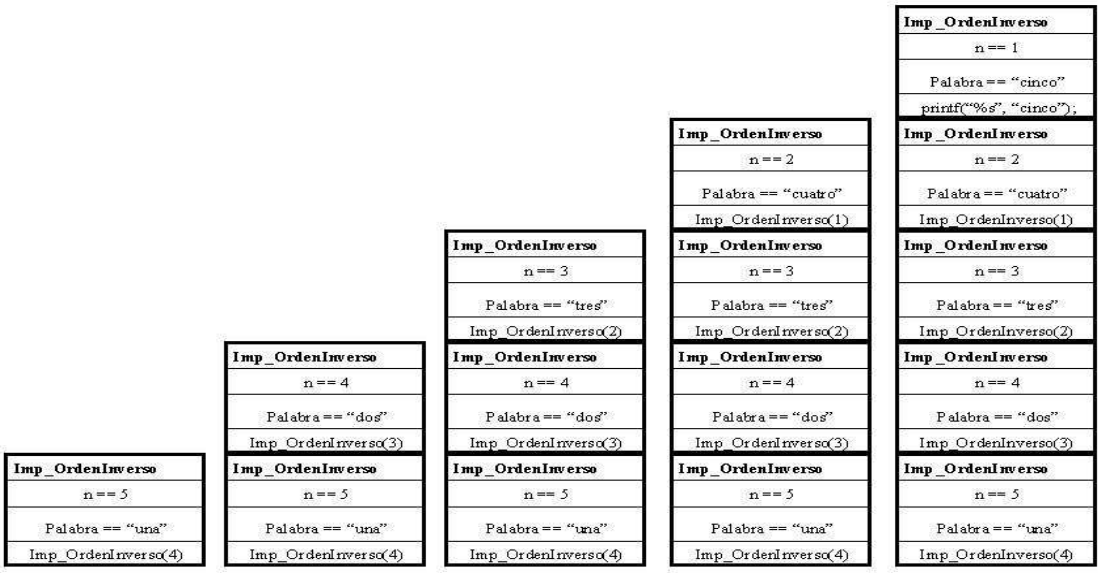
1. [P1] Se lee la primera palabra, y recursivamente se llama a la función para que lea e imprima el resto de palabras, para que finalmente se imprima la primera leída.

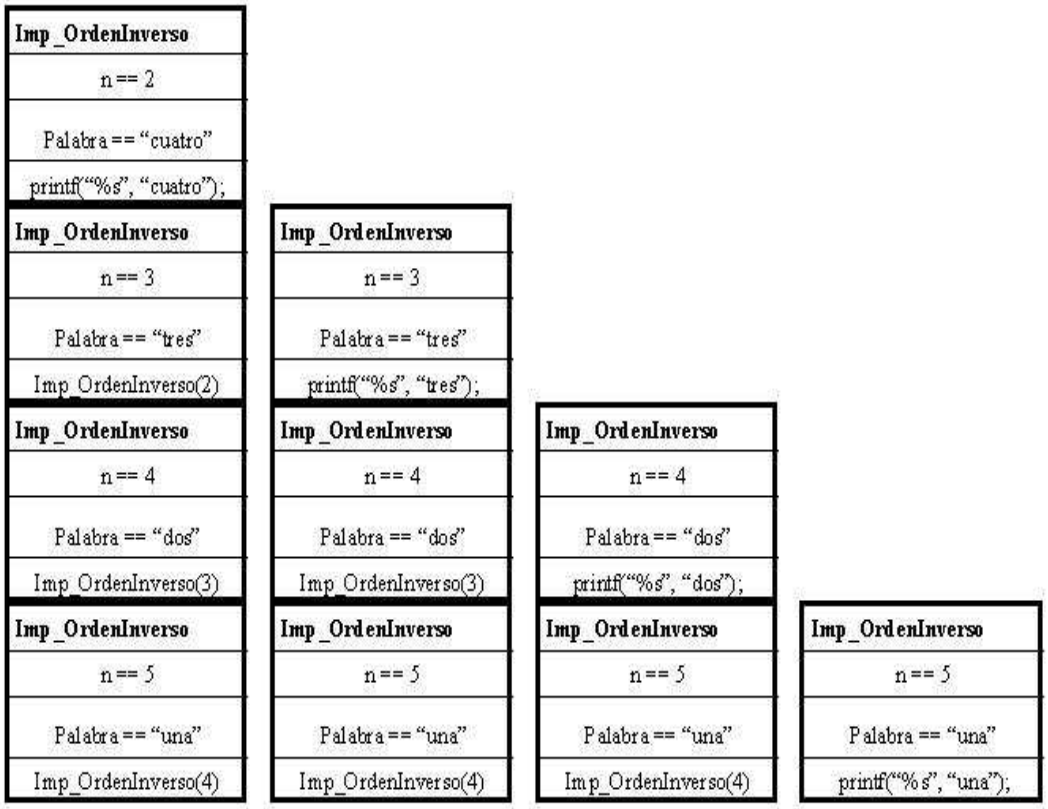
2. [P2] El caso base: cuando sólo quede una palabra (n == 1), en cuyo caso se imprimirá directamente.

3. [P3] Como en cada llamada recursiva hay que leer una palabra menos del total, en n-1 llamadas se habrá alcanzado el caso base.

4. [P4] Una vez se haya alcanzado el caso base al devolver continuamente el control a las funciones invocantes se irán imprimiendo de atrás a delante las palabras obtenidas desde la entrada estándar.

```
public void imp_OrdenInverso(int n)
      {
        String palabra;
        if (n == 1)
          {
            palabra =Console.in.readln();
            System.out.println(palabra);
          }
        else
          {
            palabra =Console.in.readln();
            imp_OrdenInverso(n-1);
            System.out.println(palabra);
          }
      }
```




Imp_OrdenInverso
n == 2
Palabra == "cuatro"
printf("%s", "cuatro");
Imp_OrdenInverso
n == 3
Palabra == "tres"
Imp_OrdenInverso(2)
Imp_OrdenInverso
n == 3
Palabra == "tres"
printf("%s", "tres");
Imp_OrdenInverso
n == 4
Palabra == "dos"
Imp_OrdenInverso(3)
Imp_OrdenInverso
n == 4
Palabra == "dos"
Imp_OrdenInverso(3)
Imp_OrdenInverso
n == 4
Palabra == "dos"
printf("%s", "dos");
Imp_OrdenInverso
n == 5
Palabra == "una"
Imp_OrdenInverso(4)
Imp_OrdenInverso
n == 5
Palabra == "una"
Imp_OrdenInverso(4)
Imp_OrdenInverso
n == 5
Palabra == "una"
Imp_OrdenInverso(4)
Imp_OrdenInverso
n == 5
Palabra == "una"
printf("%s", "una");

!OJO! Corrección de los casos base como forma de evitar una recursión infinita, y por tanto, que la pila se agote:

1. Cada rutina recursiva requiere como mínimo un caso base.

2. Se debe identificar todos los casos base ya que en caso contrario se puede producir recursión infinita.

3. Los casos bases deben ser correctos: las sentencias ejecutadas cuando se dé un caso base deben originar una solución correcta para una instancia particular del problema. En caso contrario, se generará una solución incorrecta.

4. Hay que asegurarse que cada subproblema esté más cercano a algún caso base.

2.4.- Paso de parámetros a los módulos recursivos.

Utilización de parámetros formales: cuidado al decidir si los parámetros serán de entrada, salida o entrada/salida.

Veamos un ejemplo:

```
public long factorial (long n)
  {
    if (n == 0) return 1;
    else
     {
       n = n -1;
       return (n+1) * factorial(n);
  }
```

¿Qué ocurre cuando se empiezan a devolver las llamadas recursivas?

n siempre será igual a 0, y se devolverá siempre 1

⇓

El factorial de cualquier número será 0 (INCORRECTO)

long *fact, tamanio= 3;*

fact= factorial(tamanio);

fact= factorial(n == 3)

→(n == 2) * factorial (n == 2)

→ (n == 1) *factorial(n == 1)

→(n==0)*factorial(n==0)

→return 1.

return (n+1==1) * 1 = 1

return (n+1==1) * 1= 1

return (n+1==1) * 1= 1

fact= 1

Regla general para decidir si un parámetro será pasado por copia o por referencia es la siguiente:

si se necesita recordar el valor que tenía un parámetro una vez que se ha producido la vuelta atrás de la recursividad, entonces ha de pasarse por valor. En caso contrario, se podrá pasar por referencia.

```
public void  factorial (Long n, Long fact)
  {
    if (n.longValue() == 0) fact=new Long(1);
    else  {
      factorial(new      Long(n.longValue()-1),
fact);
      fact=new
Long(fact.longValue()*n.longValue());
    }
  }
```

¡OJO! Cuidado con el número y tipo de parámetros formales de una función recursiva.

Incorrecto:
public int funcion (**int[] vector**, int objetivo, int primero)

Correcto:
public int funcion (int n, **int[] vector**, int objetivo, int primero)

Soluciones:

1. Declarar algunas variables como globales.

2. Hacer pasos por referencia.

3. Crear una función y dentro de ella implementar la función recursiva (la primera recubrirá a la segunda).

PARA EVITAR EFECTOS SECUNDARIOS SÓLO SE PERMITIRÁ LA **SEGUNDA** DE ESTAS OPCIONES.

Algunas reglas generales para el paso de parámetros:

- Generalmente pasar por copia los parámetros necesarios para determinar si se ha alcanzado el caso base.

- Si la rutina acumula valores en alguna variable o vector, ésta debe ser pasada por referencia. Cualquier cambio en dicho vector se repercutirá en el resto de llamadas recursivas posteriores.

- Los vectores y las estructuras de datos grandes se pasarán por referencia y para evitar que se puedan modificar de manera errónea, se utilizará la palabra reservada const.

2.5.- ¿Recursividad o iteración?

La propia definición inherente recursiva de un problema no significa que sea más eficiente que una solución iterativa.

Dos aspectos que influyen en la ineficiencia de algunas soluciones recursivas:

- El tiempo asociado con la llamada a las rutinas es una cuestión a tener en cuenta:

¿Merece la pena la facilidad de elaboración del algoritmo con respecto al costo en tiempo de ejecución que incrementará la gestión de la pila originado por el gran número de posibles llamadas recursivas?

- La ineficiencia inherente de algunos algoritmos recursivos. Por ejemplo, en la solución fibonacci, la mayoría de los cálculos se repiten varias veces.

Solución: Evitar procesamientos duplicado mediante la utilización de un vector o una lista, que mantuviera toda la información calculada hasta el momento.

Otro problema: el tamaño del problema a ser resuelto recursivamente. Si este implica una profundidad de recursión muy grande, olvidarla.

2.6.- Eliminación de la recursividad.

- Cuando los lenguajes no lo permitan.
- La solución recursiva sea muy costosa.

a) Eliminación de la recursión de cola.

Recursión de cola: cuando existe una llamada recursiva en un módulo que es la última instrucción de una rutina

Eliminar la recursión de cola:

```
if (condición)
   {
     Ejecutar una tarea.
     Actualizar condiciones.
     Llamada recursiva.
   }
```

El paso a estructura iterativa es inmediato, ya que es equivalente a un ciclo while:

```
while (condición)
  {
    Ejecutar una tarea.
    Actualizar condiciones.
  }
```

Ejemplo: solución al problema de las Torres de Hanoi.

```
public void torresHanoi(int n; char de; char hacia; char usando)
{
 if (n==1)
   System.out.println("Moviendo disco 1 en"+de+"a"+hacia);
 else
    if (n>1)
      {
       torresHanoi(n-1, de, usando, hacia);
       System.out.println("Moviendo"+n+"en"+de+"al"+ hacia);
       torresHanoi(n-1, usando,hacia, de);
      }
}
```

Sustituir la última llamada torresHanoi() por las asignaciones correspondientes para que los parámetros actuales tengan los valores con los que se hacía esta llamada.

```
public void torresHanoi(int n, char de, char hacia, char usando)
{
 char temp;
 while (n>1)
   {
    torresHanoi(n-1, de, usando, hacia);
    System.out.println ( " Moviendo " + n + " desde " + de + "al " +hacia);
    n=n-1; temp= de; d= usando; usando= temp;
  }
```

```
 if (n==1)
   System.out.println("Moviendo                    1
en"+de+"al"+hacia);
}
```

Ejemplo impresión en orden inverso de un vector.

```
public void imp_OrdenInversoVector( int vector [], int tamanio)
{
        if (tamanio >0)
        {
                System.out.println (" "+vector[tamanio-1]);
                imp_OrdenInversoVector(vector, tamanio-1);
        }
}
```

Eliminando la recursión de cola:

```
public void imp_OrdenInversoVector (int vector[], int tamanio)
{
        while (tamanio>0)
        {
                System.out.println (" "+ vector[tamanio-1]);
                -- tamanio;
        }
}
```

b) Eliminación de la recursión mediante la utilización de pilas.

Llamada a una rutina:

- se introducen en la pila los valores de las variables locales,

- la dirección de la siguiente instrucción a ejecutar una vez que finalice la llamada,

- se asignan los parámetros actuales a los formales y

- se comienza la ejecución por la primera instrucción de la rutina llamada.

Finalización de la rutina:

- se saca de la pila la dirección de retorno

- y los valores de las variables locales,

- se ejecuta la siguiente instrucción.

En esta técnica de eliminación de recursividad:

- el programador implementa una estructura de datos con la que simulará la pila del ordenador,
- introducirá los valores de las variables locales y parámetros en la pila, en lugar de hacer las llamadas recursivas,
- los sacará para comprobar si se cumplen los casos bases.
- Se añade un bucle, generalmente while, que iterará mientras la pila no esté vacía, hecho que significa que se han finalizado las llamadas recursivas.

Ejemplo: eliminación de la recursividad de la función ncsr:

```
        public int ncsr(int n, int k)
        {
          if (k > n) return 0;
          else if ((n == k) || (k == 0)) return 1;
                else return ncsr(n-1, k) + ncsr(n-1, k-1);
        }

    public int ncsr_nr(int n, int k)
    {
     tPila pila;
     int suma=0;
     pila=crear();
     push(n, pila);
     push(k, pila);
     while (!vacia(pila))  {
         k= tope(pila);
         pop(pila);
         n= tope(pila);
         pop(pila);
         if (k >n ) suma += 0;
         else if (k == n || k== 0) suma +=1;
         else
           {
             push(n-1, pila); push(k-1, pila); push(n-1,
    pila);
             push(k, pila);
           }
       }
     destruir(pila);   return suma;
    }
```

De manera general, dada una estructura recursiva básica de la forma:

```
rutinaRecursiva(parámetros)
Si se han alcanzado los casos basesEntonces
    Acciones que correspondan a dichos casos bases.
Si no
Llamada recursiva con los parámetros correspondientes.
```

La estructura básica de la eliminación de la recursividad es la siguiente:

```
rutinaNoRecursiva(parámetros)
Meter los parámetros y variables locales en la pila.
Repetir mientras no esté vacía la pila
    Sacar los elementos que están en el tope de la pila
    Si éstos cumplen los casos bases
    Entonces
        Acciones que correspondan a dichos casos bases.
    Si no
        Meter en la pila los valores con los que se
        produciría la(s) llamada(s) recursiva(s).
```

¡OJO! Tantas pilas como tipos de parámetros y variables locales.

2.7.- Algunas técnicas de resolución de problemas basadas en la recursividad.

2.7.1.- Divide y vencerás.

Si el tamaño n del problema < umbral dado
    entonces aplicar un algoritmo sencillo para resolverlo
        o encontrar la solución directa.
si no
    dividir el problema de en m subproblemas y encontrar
    la solución a cada uno de ellos.

Combinar las soluciones de los m problemas para obtener la solución final.

Ejemplo: búsqueda binaria de un valor en un vector.

Si el elemento buscado está en la posición que ocupa la mitad del vector, concluir la búsqueda.

Si no es así, determinar en qué mitad debería estar el elemento:
    si fuera <= que el valor de la mitad buscar en la mitad izquierda, y si fuera mayor, en la parte derecha.

¿Cuándo acabaríamos? el valor buscado es encontrado, o cuando el tamaño del vector sea cero, lo que indica que no está el valor buscado.

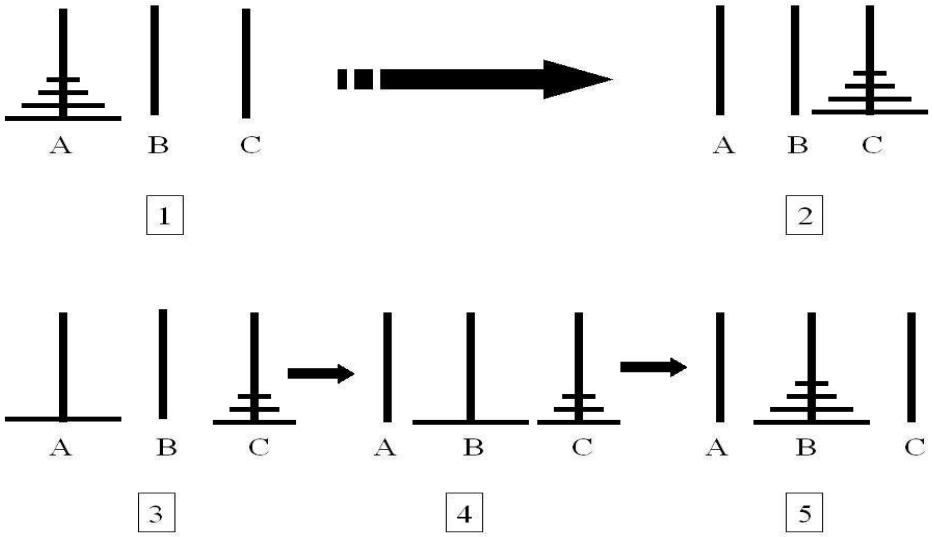
```
public int busqBinaria(int vector [], int limInf, int limSup,
int valor)
{
  int mitad;

  if (limInf > limSup) return -1;
  else
    {
     mitad= (limInf + limSup)/2;
     if (valor == vector[mitad]) return mitad;
     else

        if (valor <= vector[mitad])
        return busqBinaria(vector, limInf, mitad-1, valor);
        else
       return busqBinaria(vector, mitad+1, limSup,valor);
    }
}
```

Ejemplo: Las torres de Hanoi.

Utilizando tres postes (A, B y C) y n discos de diferentes tamaños y con un agujero en sus mitades para que pudieran meterse en los postes, y considerando que un disco sólo puede situarse en un poste encima de uno de tamaño mayor, el problema consistía en pasarlos, de uno en uno, del poste A, donde estaban inicialmente colocados, al poste B, utilizando como poste auxiliar el restante (C), cumpliendo siempre la restricción de que un disco mayor no se pueda situar encima de otro menor.



Solución:

Para un disco: moverlo de A a B.
Para más de un disco:

- Solventar el problema para n-1 discos, ignorando el último de ellos, pero ahora teniendo en cuenta que el poste destino será C y B será el auxiliar

- Una vez hecho esto anterior, los n - 1 discos estarán en C y el más grande permanecerá en A. Por tanto, tendríamos el problema cuando n = 1, en cuyo caso se movería ese disco de A a B

- Por último, nos queda mover los n-1 discos de C a B, utilizando A como poste auxiliar.

```
public void torresHanoi(int numDiscos, char origen,
                        char destino, char auxiliar)
{
 if (numDiscos == 1)
     Sytem.out.println("Mover el disco de arriba del
     poste"+ origen+"al "+ destino);
 else
   {
    torresHanoi(numDiscos-1, origen, auxiliar,
destino);
    torresHanoi(1, origen, destino, auxiliar);
    torresHanoi(numDiscos-1, auxiliar, destino,
origen);
   }
}
```

2.7.2.- Backtracking.

Ejemplo: el problema de las ocho reinas

Un tablero de ajedrez, el cual tiene un total de 64 casillas (8 filas x 8 columnas). El problema consiste en situar ocho reinas en el tablero de tal forma que no se den jaque entre ellas. Una reina puede dar jaque a aquellas reinas que se sitúen en la misma fila, columna o diagonal en la que se encuentra dicha reina.

R							
						R	
				R			
							R
	R						
			R				
					R		
		R					

Solución:

- Meter la primera reina.

- Meter la segunda en una casilla que no esté atacada por la primera reina.

- Meter la tercera en una casilla no atacada por la primera o la segunda.

- Meter sucesivamente el resto de reinas. Si en algún momento no puede colocarse una reina, ir deshaciendo movimientos de las reinas ya colocadas y a partir de esa reorganización continuar la colocación del resto de reinas.

Funciones auxiliares:

- void asignarReinaA(int fila, int columa) => Sitúa un 1 en la casilla (Fila, Columna) indicando que está ocupada por una reina.

- void eliminarReinaDe(int fila, int columna) => Sitúa un 0 en la casilla (Fila, Columna) indicando que esa casilla está libre (antes estaba ocupada por una reina y ahora deja de estarlo).

- int recibeJaqueEn(int fila, int columna) => Devuelve 1 si la casilla (Fila, Columna) recibe jaque de alguna reina y 0 en caso contrario.

public void situarReina(int Columna)

/* Col indica en qué columna se quiere situar la reina, y Situada es una variable que tomará el valor 1 cuando se haya logrado ubicar correctamente a la reina correspondiente, y 0 cuando el intento haya sido infructuoso. */

```
{
 int fila, situada;
 if (columna > 7) {
     situada=1;
     return Situada;
 else
    {
      situada=0;
      fila=1;
      while (!(situada) && (fila <= 7))
          if (fecibeJaqueEn(fila, columna)) ++fila;
          else
       {
          asignarReinaA(fila, columna);
          situada=situarReina(columna+1);
          if ( !(Situada))
             {
                eliminaReinaDe(fila, columna);
                 ++fila;
             }
       }
    }
 return situada;
```

```
}
```