

## Capítulo 8

1. Problemas: especificaciones iterativas y recursivas
2. Definición de la iteración
3. Definición de la recursividad
4. Método para generar funciones recursivas: caso base y caso recurrente
5. Implementando soluciones recursivas (manejo de la memoria)
6. Ejemplos
7. Métodos de ordenación : Quick Sort y Merge Sort
8. Recursividad Indirecta
9. Errores comunes
10. Comparación Iteración – Recursividad
11. Conclusión

### 1. Procesos Recursivos

Las soluciones que conllevan a la repetición de operaciones, pueden ser implementadas por medio de procesos iterativos o recursivos..

A continuación algunos problemas con especificación iterativa y recursiva.

Problema	Solución	
	Iterativa	Recursiva
Factorial $n!$	$1*2*3 \dots *(n-2)*(n-1)*n$	Si $n = 0$ o $n = 1 \rightarrow 1$ Si $n > 1 \rightarrow n*(n-1)!$
Potencia $X^n$	$X*X*X \dots *X$ (n veces)	Si $n = 0 \rightarrow 1$ Si $n > 0 \rightarrow X * X^{n-1}$ Si $n < 0 \rightarrow 1 / X^{-n}$
Nro. de Fibonacci	1, 1, 2, 3, 5, 8,...	Si $n = 1$ o $n = 2 \rightarrow 1$ Si $n > 2 \rightarrow \text{Fib}(n-1) + \text{Fib}(n-2)$
Sumatoria de 1 a n $\sum_{i=1}^n i$	$1 + 2 + 3 + 4 + 5 \dots + n$	Si $n = 1 \rightarrow 1$ Si $n > 1 \rightarrow n + \sum_{i=1}^{n-1} i$

### 2. Definición de la iteración

Un proceso iterativo es aquel en el que se repite la ejecución de un conjunto de instrucciones una determinada cantidad de veces (que puede ser conocida a priori o no).

### 3. Definición de la recursividad

Un proceso recursivo es aquel que implementa una solución en etapas, cada una de ellas involucra una simplificación del mismo problema en la etapa anterior (caso recurrente), hasta que una condición da por finalizada la recursión, devolviendo un resultado (caso base). Se denomina recursivo pues se vuelve a utilizar las mismas instrucciones para las distintas etapas.

### 4. Método para generar funciones recursivas: caso base y caso recurrente

La recursión puede utilizarse para reemplazar la iteración, aunque generalmente es menos eficiente en tiempo y espacio de memoria, permite una solución natural y simple a ciertos problemas que serían más complejos de resolver utilizando ciclos.

Todo cálculo iterativo puede ser planteado en forma recursiva, de la siguiente forma:

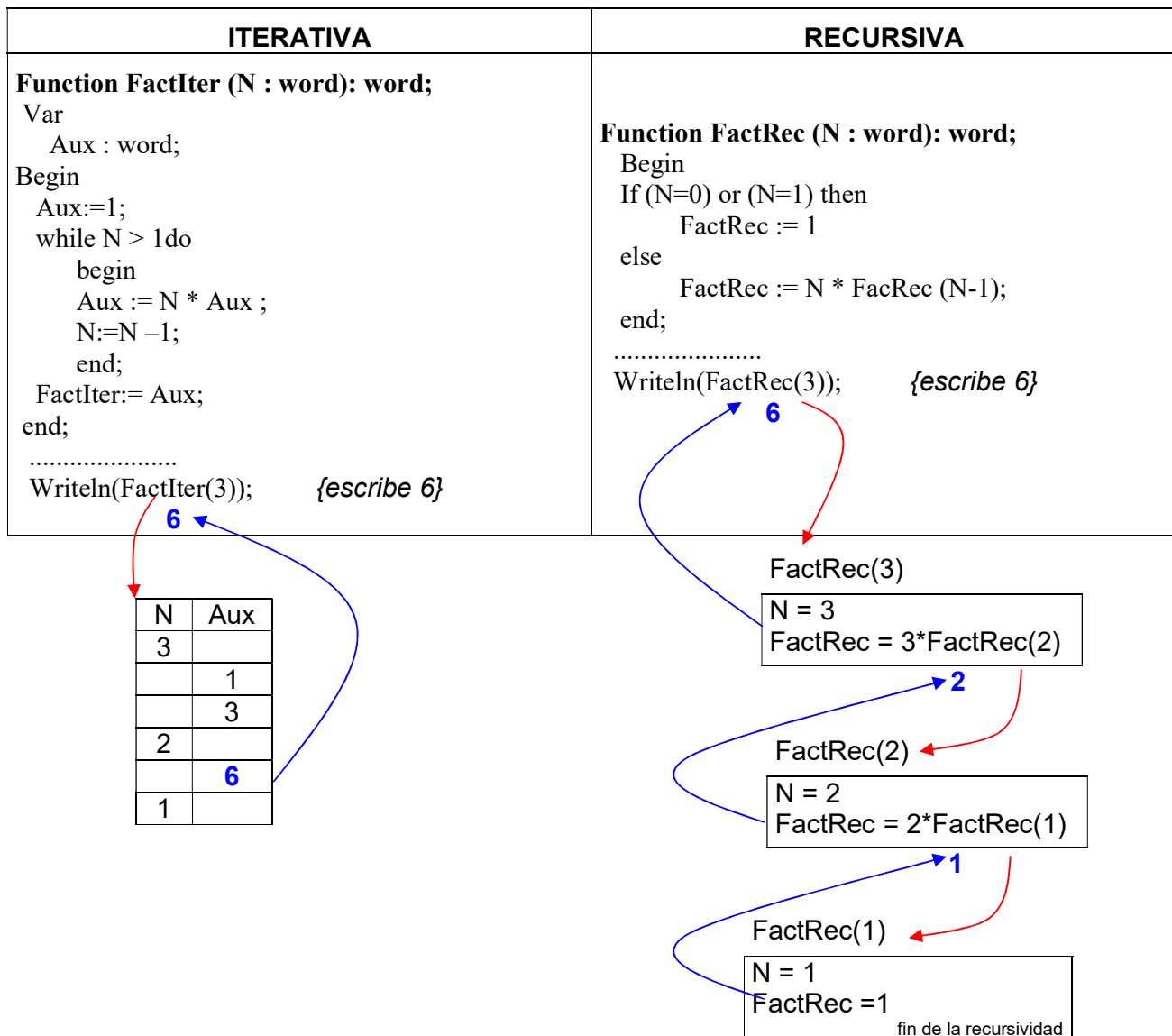
a) identificar el caso base y el caso recurrente

- a1) caso base la solución es simple, no se genera un llamado recursivo, y devuelve la solución. Ejemplo :  $0! = 1$
- a2) caso recurrente la solución no es trivial, se evalúa puntualmente los parámetros recibidos (compara, opera) y se invoca a si misma con diferentes parámetros para resolver un problema de “menor tamaño” o un caso más simple. Ejemplo:  $4! = 4*3!$ . Estos llamados deben tender al caso base

b) suprimir el ciclo e implementar una sentencia selectiva

```
if condición then
    Caso base
else
    Caso recurrente;
```

E1.-Desarrollo de la función Factorial iterativa y recursiva





## 5. Implementando soluciones recursivas (manejo de la memoria)

Una función o procedimiento recursivo es aquel que se invoca a sí mismo y repite la ejecución de su código sucesivas veces, con distintos parámetros.

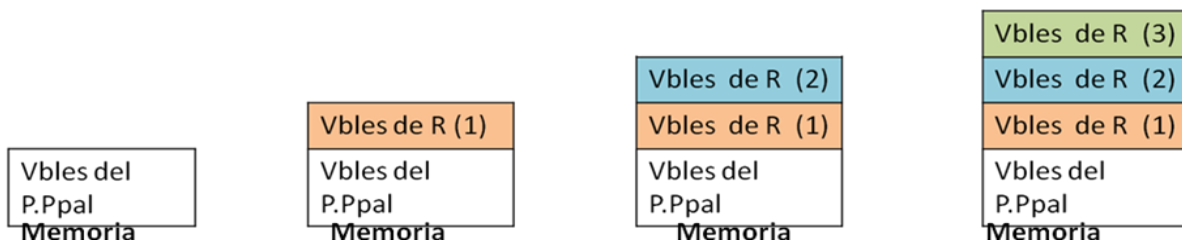
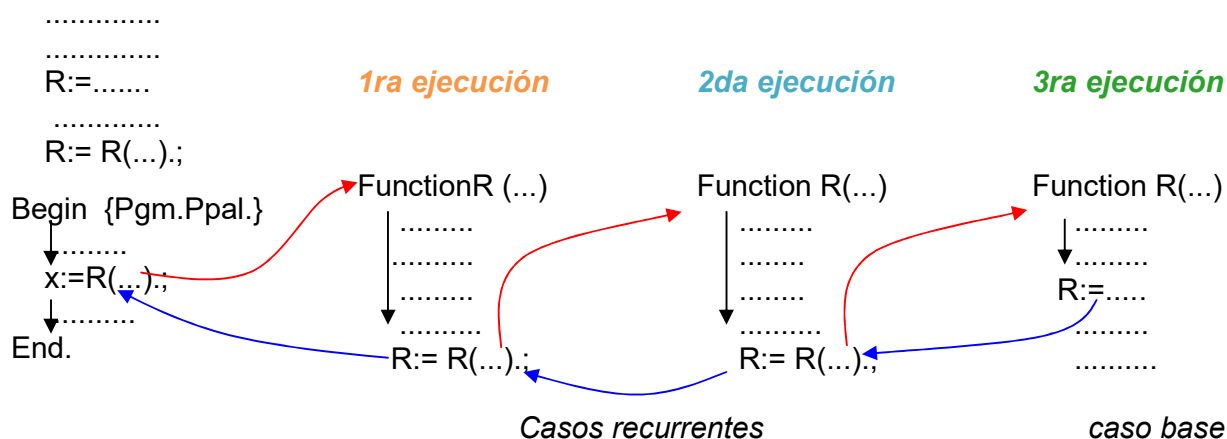
Para comprender el mecanismo de la recursividad, es preciso imaginar que cada llamado a un proceso recursivo desencadena la ejecución del mismo código con diferentes parámetros.

Cada nueva instancia de ejecución se comporta como un proceso independiente con su propio espacio en el segmento de memoria (stack) para almacenar parámetros y variables locales.

Cuando un proceso se invoca a si mismo, su ejecución queda "suspendida" (manteniendo en el stack los valores de sus parámetros y variables locales) para dar lugar (  ) a la ejecución que desencadena la invocación recursiva (con otros parámetros y variables).

Cuando la ejecución del proceso invocado recursivamente termina, éste libera su memoria, y el control vuelve al " punto de llamada" (  ) retomando los valores propios en parámetros y variables.

```
Program xxx;
Function R(...):tipo;
```



## 6. Situaciones a considerar

- ✓ Ausencia del Caso Base: En algunas situaciones el fin de la recursividad es controlada por una expresión condicional y el caso base puede estar vacío, solo finaliza la recursión:

```
if condición then
    Caso recurrente;
```

E2: Dado un numero natural N mostrar la secuencia descendente: N, N-1, N-2,..., 2, 1, 0

Procedure Cuenta (n:integer);

Begin

writeln(n);

if n > 0 then

Cuenta(n-1); { caso Recurrente }  
~~{ caso Base }~~

end;

While n >= 0 do

begin

writeln(n);

n := n - 1;

end;

- ✓ Orden de Ejecución: Según el problema a resolver el orden de las sentencias es indistinto, o producir un resultado diferente en cada caso

Caso 1  
Sentencias;  
if condición then  
Caso recurrente;

Caso 2  
if condición then  
Caso recurrente;  
Sentencias;

En el Caso 1 decimos que la problemática se 'resuelve a la ida', es decir que primero se ejecutan las sentencias y luego se invoca la recursión. En el caso 2 se dice que se 'resuelve a la vuelta', después de invocar la recursión.

Caso 1: En el ejemplo anterior vemos que se resuelve 'a la ida'

Program CuentaRec;

Procedure Cuenta (n:integer);

Begin

writeln(n);

if n > 0 then

Cuenta(n-1);

end;

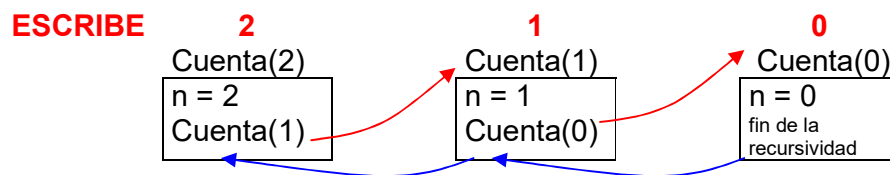
Begin

Cuenta(2);

Se visualiza : 2 1 0

end.

Para hacer un seguimiento de cada invocación recursiva, se representa el espacio de memoria vinculado a dicha ejecución, se utiliza un rectángulo que contiene los valores de parámetros actuales y variables locales (si las hubiera). A diferencia de las tablas con columnas, utilizadas en las pruebas de escritorio de las soluciones iterativas.



Caso 2: Si invertimos el orden de la impresión en el procedimiento Cuenta, la propuesta queda:

Procedure Cuenta (n:integer);

Begin

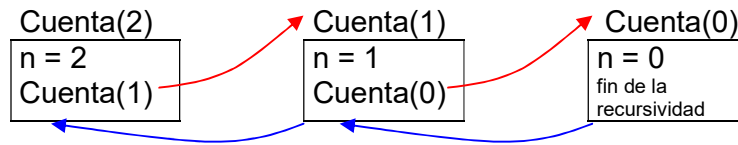
if n > 0 then

Cuenta(n-1);

writeln(n);

end;

Se visualiza : 0 1 2



En este ejemplo se evidencia que el orden de las sentencias, antes o después del llamado recursivo, cambia el resultado del proceso.

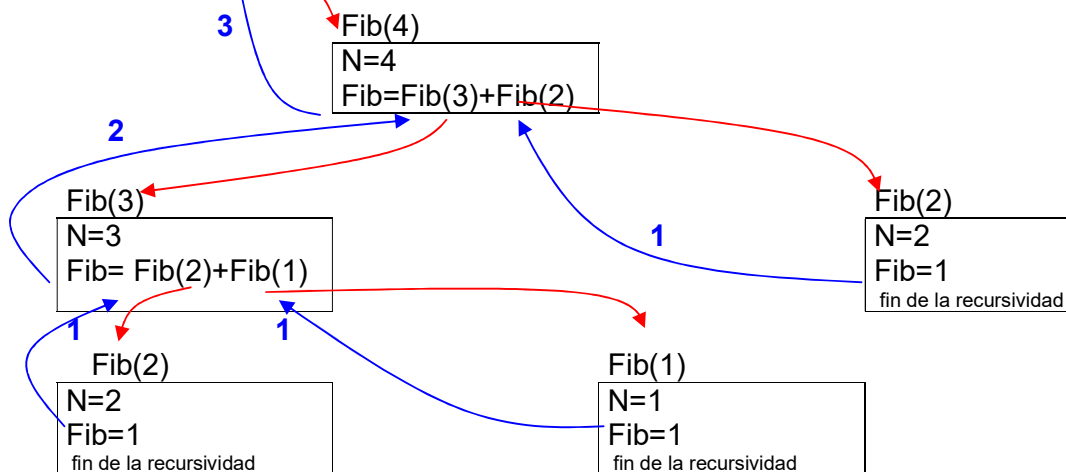
El peligro de la recursividad es que una mala definición pueda llevar a infinitas invocaciones. Siempre debe existir una condición de terminación, en la mayoría de los casos una estructura de decisión. (if-then-else). Un ejemplo de mala definición sería omitir el "if" del procedimiento Cuenta.

E3.-La función de Fibonacci, devuelve el n-ésimo termino de la serie: 1 1 2 3 5 8 13 21... Los dos primeros términos son 1 y los restantes la suma de los dos anteriores.

Fibonacci (1)= 1  
 Fibonacci (2)= 1  
 Fibonacci N)= Fibonacci (N-1) + Fibonacci (N-2)

```
Function Fib (N:byte): word;      {devuelve el N-ésimo termino de la serie Fibonacci}
Begin
  if (N = 1) or (N = 2) then
    Fib := 1
  else
    Fib:= Fib(N-1) + Fib(N-2);
end;
```

.....  
 writeln(Fib (4)); {escribe 3}  
 .....



E4.-Dado un número entero positivo N calcular la suma  $1+2+3+4+ \dots + N$

Procedure Suma1 (N: word; Var Res: word);

Begin

If N > 0 then

Begin

Res := Res + N; { \* acumula antes de invocar recursivamente }

Suma (N-1, Res);

end;

end;

.....  
Res:=0; { \* es necesario asignar 0 antes de invocar al procedimiento Suma }

Suma(4,Res); {devuelve en Res 10}  
.....

*Se puede evitar la asignación de cero a Res antes de invocar a Suma y hacer que el mismo procedimiento la inicie, en ese caso, se acumula después de invocar recursivamente.*

Procedure Suma2 (N: word; Var Res: word);

Begin

If N > 0 then

Begin

Suma (N-1, Res);

Res := Res + N; { \* acumula después de invocar recursivamente }

end

else

Res := 0

end;

Notar que si se invierte el orden de la invocación recursiva a Suma, cuando N=0 asigna 0 a Res y pierde lo que tiene acumulado.

Procedure Suma3 (N: word; Var Res: word);

Begin

If N > 0 then

Begin

Res := Res + N; { acumula antes de invocar recursivamente }

Suma (N-1, Res);

end

else

Res := 0 {pierde lo acumulado}

end;

*otra forma de resolverlo es mediante una función*

Function Suma4 (N:word):word;

Begin

If (N = 0) or (N = 1) then

Suma := N

else

Suma := N + Suma(N-1)

end;

E5.-Verificar si un valor X se encuentra en un arreglo V de N elementos

En cada ejecución se analiza un elemento V[N], comenzando por el último y “moviéndose” al inmediato de la izquierda con una invocación recursiva. La búsqueda finaliza cuando se encuentra X o cuando se han examinado todos los elementos y no se encontró.

El parámetro N es el índice de la componente que se analiza en cada ejecución, la primera vez la posición del último (se evita usar dos parámetros para controlar el índice i y N).

Function Esta(V:TV; N:byte; X:real): boolean;

Begin

if N>0 then

if V[N] = X then

Esta := True

else

Esta := Esta(V, N-1, X)

else

Esta := False

End;

} Esta := (V[N] = X) or Esta(V, N-1, X)

Como se observa a la derecha, la estructura alternativa interna, puede reemplazarse por una expresión lógica.

Incluso la alternativa externa puede formar parte de una única expresión lógica que controla el caso recurrente y el caso base.

Function Esta (V:TV; N:byte; X:real): boolean;

Begin

Esta := (N> 0) and ( (V[N] = X) or Esta(V, N-1, X) )

End;

E6.-Procesos recursivos que “trabajan en dos direcciones”, horizontal y vertical.

Desarrollar un procedimiento que imprima la siguiente secuencia para cualquier límite f

Program Ascendente;

{c comienzo, a actual, f límite}

Procedure Recursivo (c,a,f:word);

Begin

write(a:4);

if c < f then

if a = f then

Begin

writeln;

recursivo(c+1,c+1,f);

end

else

recursivo(c,a+1,f);

end;

begin

recursivo(1,1,5); {f= 5}

readln

end.

Ejemplo f = 5

1 2 3 4 5

2 3 4 5

3 4 5

4 5

5

E7.-Recorrer una matriz numérica de NxM y devolver la cantidad de negativos que almacena. El recorrido de la matriz se efectúa desde el elemento Mat[N,M] hasta el Mat[1,1], desde la última fila hasta la primera y para cada fila desde la última columna hasta la primera.

Function CantNeg (Mat :TM; i,j,M: byte): Byte;

Var

incr: byte;

Begin

if i = 0 then

CantNeg:= 0

else

begin

if Mat[ i, j ] < 0 then { determina el incremento, 1 si es negativo }

incr:= 1

else

incr:= 0;

if j > 1 then {sigue analizando la misma fila}

CantNeg:= incr + CantNeg (Mat, i, j -1, M); {misma fila, columna de la izquierda}

else

CantNeg:= incr + CantNeg (Mat, i -1, M, M) {fila anterior, última columna}

end;

end;

.....

write (CantNeg(Mat, N,M,M)); {invocación}

Es importante aclarar la forma de moverse en la matriz para interpretar el algoritmo, como así también mostrar la primera invocación, que da a los parámetros los valores iniciales para que este movimiento sea correcto.

E8.-Verificar si una matriz Mat de NxM, cumple que un elemento X se encuentra al menos una vez en cada columna.

Se recorre la matriz por columnas desde la última columna hasta la primera y para cada una de ellas, desde la última fila hasta la primera. Se compara cada elemento con X, si coincide se pasa a la siguiente columna, si no coincide se pasa al elemento de la fila superior en la misma columna. En el caso de no encontrar X en un columna se interrumpe el proceso devolviendo False.

Function Cumple (Mat :TM; X : real; i,j,N: byte): boolean;

Begin

if j = 0 then

Cumple := True

else

if Mat[ i, j ] = X then

Cumple:= Cumple(Mat, X, N, j -1, N) {columna anterior , última fila}

else

if i>1 then {sigue buscando en la misma columna}

Cumple:= Cumple(Mat, X, i -1, j, N) {fila anterior , misma columna}

else

Cumple:= False { X no está en la columna j }

end;

.....



```
write(' Tiene al menos una vez el valor ', X, ' en cada columna ', Cumple(Mat, X, N, M, N));
```

E9.- Verificar que en una matriz cuadrada de NxN es triangular inferior (sólo ceros debajo de la diagonal).

Se recorre solo el triángulo inferior, comenzando por Mat[ N,N - 1] y moviéndose hacia la columna de la izquierda en la misma fila hasta alcanzar la columna 1, luego a la fila superior columna a la izquierda de la diagonal.

```
Function Diagonal (Mat:TM; i, j :byte): boolean;
Begin
  if i = 1 then
    Diagonal := True
  else
    if Mat [ i , j ] <> 0 then
      Diagonal := False
    else
      if j = 1 then
        Diagonal := Diagonal (Mat, i - 1, i - 2) {fila anterior, izquierda de la diagonal}
      else
        Diagonal := Diagonal (Mat, i , j - 1) {misma fila anterior, columna izquierda }
      end;
    .....
  write(' Es diagonal', Diagonal ( Mat, N, N - 1));
```

### 7.1.Método de Ordenación Rápida – Quick Sort

En la actualidad está considerado como el método de ordenación interna más eficiente y veloz. Diseñado por Hoare en 1962, es una mejora del método de intercambio directo.

Descripción:

Dado una lista A de N elementos, el método consiste en ordenar sus elementos particionándolo de la siguiente manera:

Se elige un elemento del arreglo, al que llamaremos pivote.

Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Program QS;

Type

TV= array[1..12] of integer;

Const

a:TV=(5,1,12,8,-1,3,10,9,14,0,15,7); {para probarlo evitando la lectura}

```

Procedure QuickSort ( Var a:TV; pri, ult: byte);
Var
  i, j : byte;
  aux, p: integer;
Begin
  i:= pri; j:= ult; p:= a[(pri + ult) div 2] ; { p pivote entre dos subarreglos}
  repeat
    while (i <= j) and (a[i]<p) do {busca un a[i] >= p a la izquierda}
      i :=i+1;
    while (i <= j) and (a[j]>p) do {busca un a[j] <= p a la derecha}
      j := j-1;
    if i<j then {si encuentra de ambos lados intercambia}
      begin
        aux:=a[i]; a[i]:=a[j]; a[j]:=aux
      end;
    i:=i+1; j:= j-1; {continua avanzando hacia el pivote}
  until i>=j; {hasta que a la izquierda queden los menores y a la derecha los mayores}
  if pri<j then { si existe subarreglo izquierdo lo ordena}
    quicksort(a, pri, j);

  if i<ult then { si existe subarreglo derecho lo ordena}
    quicksort(a, i, ult);

end;
Var
  i: byte;
Begin
  QuickSort(a,1,12);
  For i:= 1 to 12 do
    write (a[i]:4);
    readln;
  end.

```

## 7.2.Método de Ordenación por mezcla – Merge Sort

Se basa en partir el arreglo en 2 y ordenar cada parte, luego intercalarla (mezclarla). Este proceso se aplica recursivamente a cada parte. Fue desarrollado en 1945 por John Von Neumann . <https://www.youtube.com/watch?v=EeQ8pwjQxTM>

```

Program MS;
Type
  TV= array[1..12] of integer;
Const
  a:TV=(5,1,12,8,-1,3,10,9,14,0,15,7); {para probarlo evitando la lectura}

Procedure Mezclar ( Var a:TV; pri, medio, ult: byte);
Var
  i, j, k, n, t : byte;
  b:TV;

```

```

Begin
k:=0; i:= pri; j:= medio +1; n:= ult - pri + 1;
while (i <= medio) and (j<= ult) do
  begin
    k:=k+1;
    if a[i] < a[j] then
      begin
        b[k] := a[i]; i:= i + 1
      end
    else
      begin
        b[k] := a[j]; j:= j + 1
      end
    end;
  For t:= i to medio do
    begin
      k:=k+1; b[k] := a[t];
    end;
  For t:= j to ult do
    begin
      k:=k+1; b[k] := a[t];
    end;
  For k:= 1 to n do
    begin
      a[pri]:=b[k]; pri:=pri + 1;
    end;
  end;

Procedure OrdenarMezcla(Var a:TV; pri,ult:byte);
Var
  medio: byte;
begin
  If pri < ult then
    begin
      medio:= (pri + ult) div 2;
      OrdenarMezcla(a, pri, medio);
      OrdenarMezcla(a, medio + 1, ult);
      Mezclar(a, pri, medio, ult);
    end
  end;
Var
  k: byte;
Begin
  OrdenarMezcla(a,1,12);
  For k:= 1 to 12 do
    writeln (a[k]:4);
    readln;
  end.

```

## 8. Recursividad Indirecta

Existen dos tipos o formas de recursividad en Pascal:

a. Directa o simple: Un subprograma se llama a sí mismo directamente. Es el tipo de recursividad empleado en los ejemplos vistos hasta ahora.

b. Indirecta: Un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A.

Una subrutina no puede ser utilizada antes de ser declarada. Este problema, en Pascal, se resuelve mediante la palabra reservada "forward".

Ejemplo: Programa que determina la paridad de un entero positivo implementando dos funciones mutuamente recurrentes: function Par (n:word) : boolean; y function Impar (n:word) : boolean;

```
Program Paridad;
```

```
var
```

```
  n:integer;
```

```
Function Par(n:word ): boolean; forward;
```

```
Function Impar(n:word): boolean;
```

```
  begin
```

```
    if n = 0 then
```

```
      Impar := false
```

```
    else
```

```
      Impar := Par(n-1)
```

```
  end; {Impar}
```

```
Function Par (n:word): boolean;
```

```
  begin
```

```
    if n=0 then
```

```
      Par := true
```

```
    else
```

```
      Par := Impar(n-1)
```

```
  end; {Par}
```

```
.....
```

```
If Par (n) then
```

```
  writeln(n, ' es par ')
```

```
else
```

```
  writeln(n, ' es impar ');
```

```
.....
```

### 9. Errores frecuentes

- a) recursión infinita por no utilizar en la invocación parámetros diferentes a los de entrada.
- b) no asignar el resultado de la función recursiva a su nombre.
- c) el caso *base* no controla la condición de fin de recursividad o no devuelve la solución.
- d) el caso *recurrente* no resuelve la etapa recursiva

### 10. Iteración versus Recursividad

Aspectos	Iteración	Recursividad
Estructura de Control	Repetitiva	Selectiva
Finalización	Evalúa la condición y concluye el ciclo	Evalúa la condición y reconoce el caso base
Desventajas	No presenta una solución sencilla y clara para los problemas de naturaleza recursiva	Requiere mayor tiempo de ejecución y más memoria (pasaje y almacenamiento de parámetros)
Ventajas	Requiere menor tiempo de ejecución y menos memoria.	Permite una solución simple a problemas de naturaleza recursiva
Condición crítica de tiempo y memoria	Se utiliza esta técnica	No se la puede utilizar

### 11.-Conclusión

La recursividad debe utilizarse cuando no existe una solución iterativa simple, sencilla y clara.