Programación Procedural

Unidad 4

Recursividad

2020

Recursividad

Es una alternativa a la iteración, un proceso mediante el cual se puede definir una función en términos de sí misma

Ej : Función Factorial

Definición Iterativa

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

Definición Recursiva

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

Factorial de n

$$n! = n * (n - 1)!$$

$$(n - 1) * (n - 2)!$$

$$(n - 2) * (n - 3)!$$

$$\vdots$$

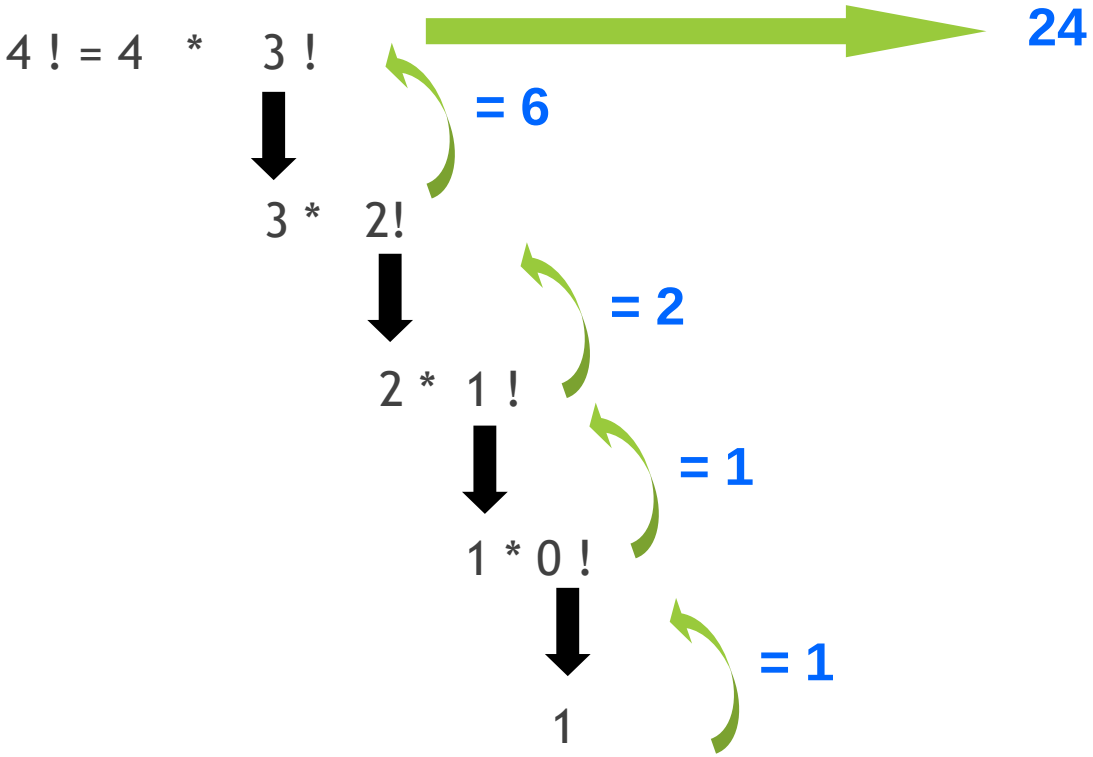
$$3 * 2!$$

$$2 * 1!$$

$$1 * 0!$$

$$1$$

Ejemplo: Factorial de 4



Construcción y ejecución de funciones recursivas

Al construir una función recursiva, se debe asegurar la existencia de:

1. **Un caso base (puede haber más de uno)**, que permite detener la invocación sucesiva de la función; caso contrario se tendrían una serie infinita de invocaciones sucesivas.

2. **Uno o más casos generales**, que permiten que la función se invoque a sí misma con valores de parámetros que cambian en cada llamada; acercándose cada vez más al caso base.

Estructura de función Recursiva

If *CONDICION*

{

Caso General

}

- Proceso Invocación Proceso
- Proceso Invocación
- Invocación Proceso
- Invocación +Proceso

else

{

Caso Base

}

- Explicito: Proceso
- Implícito: Nada

Construcción y ejecución de funciones recursivas

En la ejecución, las sentencias ***que aparecen después de cada invocación no se resuelven inmediatamente, éstas quedan pendientes***.

Cada invocación recursiva, genera un registro de activación y se apila en el stack o pila.

Esos registros de activación se van desapilando en el orden inverso a como fueron apilados.

Factorial de 3

$n! = n * (n - 1)!$ si $n > 0$

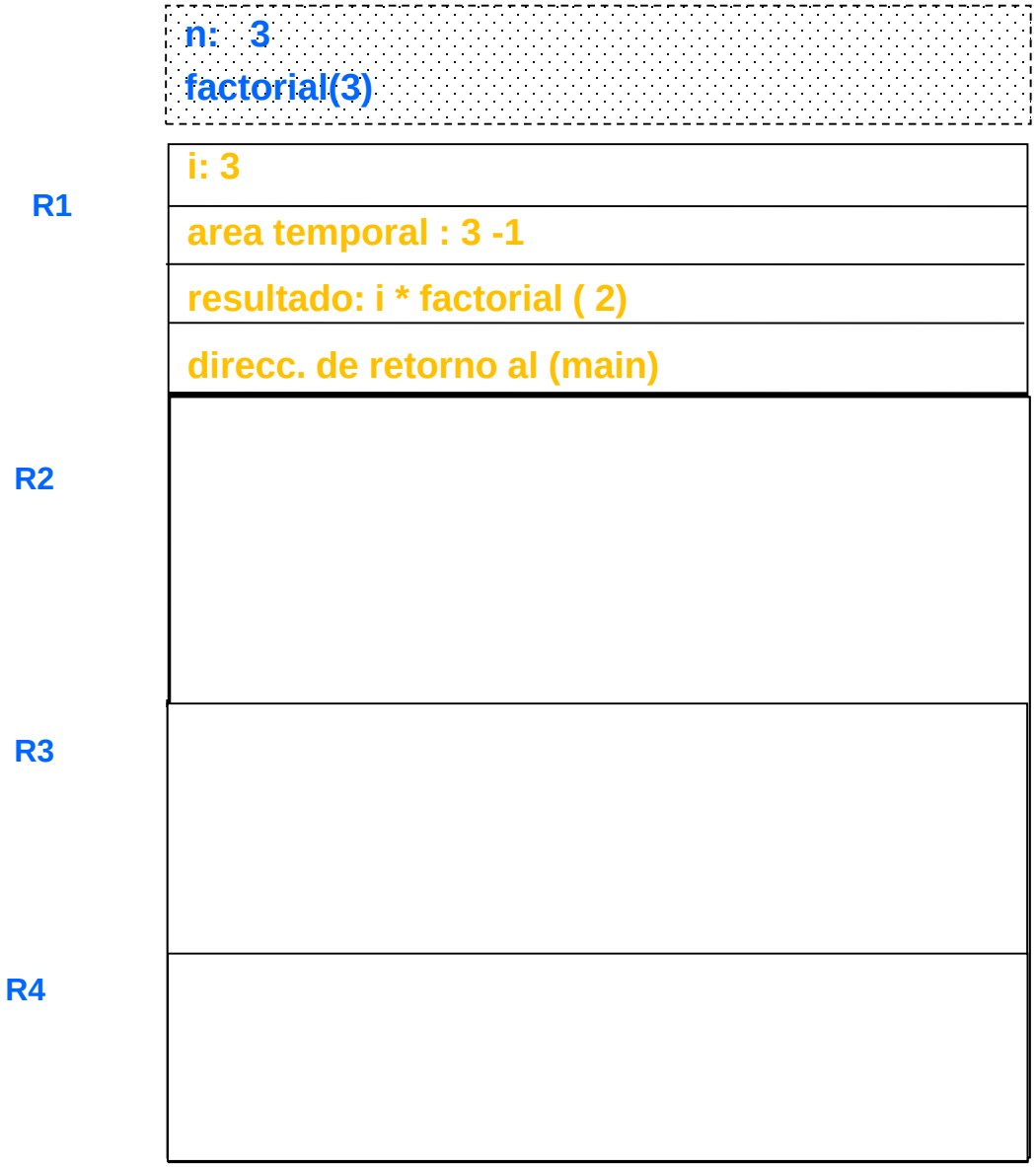
$n! = 1$, si $n = 0$

```
long int factorial (  int i )
 {
  if ( i = = 0 )
            return 1;
 else
            return i * factorial ( i - 1 ) ;
}
main ( )
{
   int n;
   scanf (" %d",& n);
   printf ("\n n! =  %l ", factorial ( n ) );
}
```

Caso Base

Caso General

Mapa de memoria




```
long  factorial ( long int i )
 {
  if ( i = = 0 )
        return  1
 else
    return (i * factorial ( i–1 ));
 }

int main ( )
{
 int n;
 scanf (" %d",& n);
printf (" n! = %d ", factorial(n ));
}
```

eje2_op2.cpp | activ1.CPP | factorial_recurs.CPP

```
1  # include<stdio.h>
2  # include<conio.h>
3  long factorial(long &n); /* prototipo de la función factorial */
4  // long factorial(long n) ;
5  main()
6  {
7     int n;
8     printf("\n n = ");
9     scanf(" %d", &n);
10    printf("\n n! =  %d\n",factorial(n));
11       printf("\n valor del numero despues de la función n! =  %d\n",n);
12    getche();
13    /*getchar();*/
14 }
15 long factorial(long &i)
16   //  long factorial(long i)
17  {   printf("\n firección de la variable en cada invocación  %x\n",&i);
18   if(i==0)
19     return(1);
20   else
21     return(i*factorial(i-1));
22 }
23
```

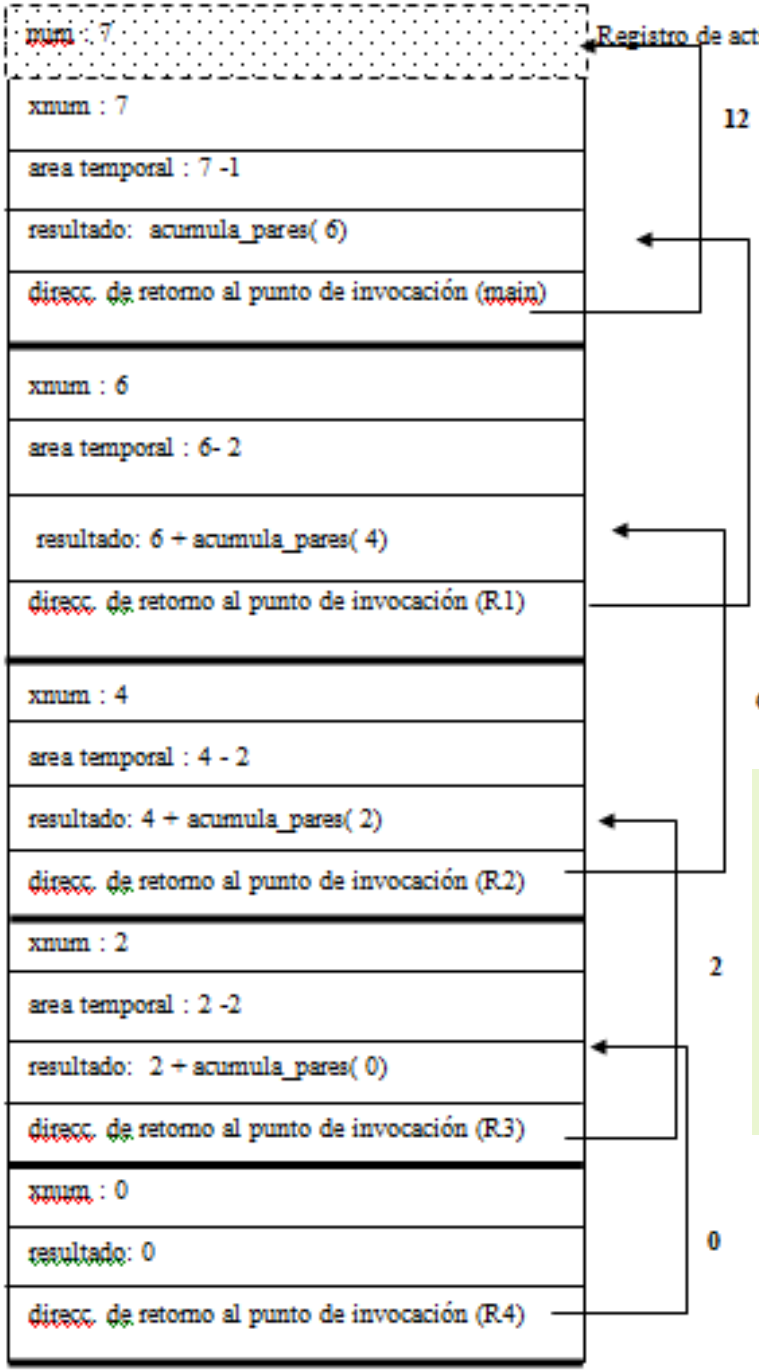
D:\IVO\PROCEDURAL 2017\Ejercicios c\recursividad\factorial_recurs.exe

```
n = 4
firecci¾n de la variable en cada invocaci¾n  12ff4c
firecci¾n de la variable en cada invocaci¾n  12ff3c
firecci¾n de la variable en cada invocaci¾n  12ff28
firecci¾n de la variable en cada invocaci¾n  12ff14
firecci¾n de la variable en cada invocaci¾n  12ff00
n! =  24
valor del numero despues de la funci¾n n! =  4
```

Sumar recursivamente los números pares menores o iguales que un valor ingresado por teclado.
Señalar Caso Base y General
Mostrar el estado de memoria para num=7

```
int  acumula_pares ( int xnum)
 {    if  (xnum)
         if ((xnum % 2)== 0)
              return xnum + acumula_pares (xnum - 2);
         else return  acumula_pares (xnum – 1);
    else return (0);
}
int main()
 {   int  num;
    scanf("%d", &num);
    printf("\n suma de pares menores que:  %d es  %d ", num,
     acumula_pares(num));
    getch();
}
```

REGISTROS APILADOS acumula_pares




```
int acumula_pares ( int xnum)
 {    if  (xnum)
          if ((xnum % 2)== 0)
           return xnum + acumula_pares( xnum - 2);
          else return acumula_pares( xnum – 1);
     else return (0);
}
```

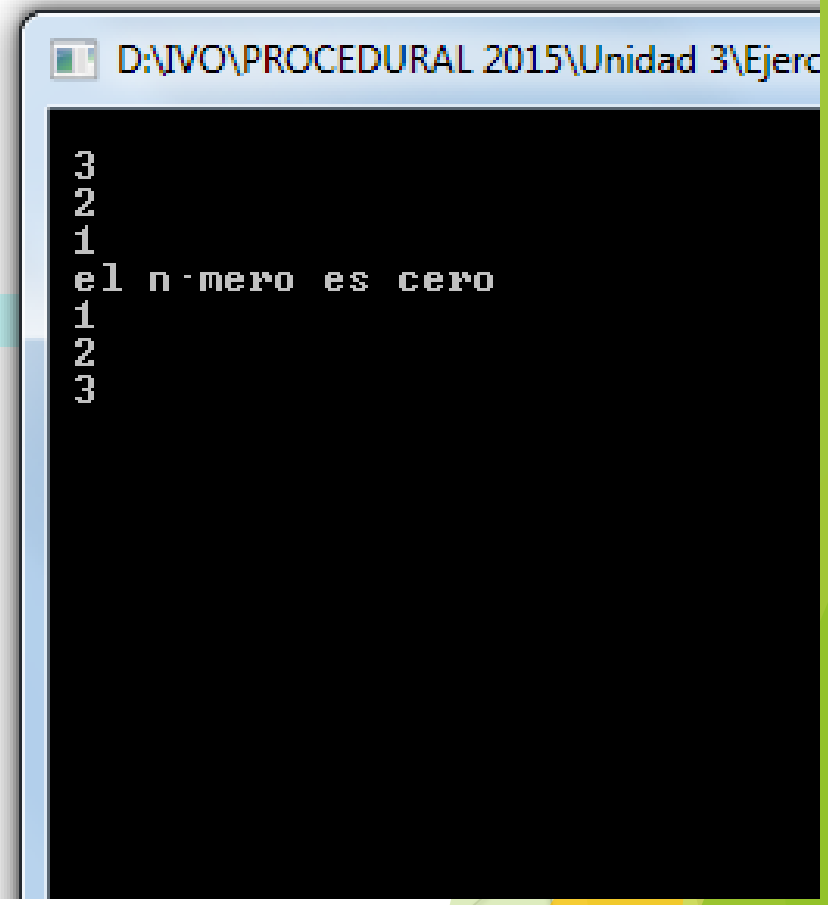
```
int main()
 {
 int  num;
  scanf("%d", &num);
  printf("\n suma de pares menores que:  %d es
                    %d ", num,  acumula_pares(num));
  getch();
}
```

Dada la siguiente definición de una función, se pide:

a) Construir la función main con la invocación.
b) Señalar caso base y general
c) Graficar el mapa de memoria al invocar a listar (3) y realizar el seguimiento.

```
void listar ( int n)
  { if ( n ! =  0 )
       {  printf (" %d", n);
          listar ( n – 1 );
          printf (" %d", n);  }
    else printf (" el número es cero");
  return;
}
```

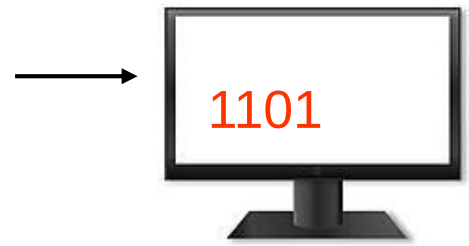
```c
#include <stdio.h>
#include <conio.h>

void listar( int n)
   {       if (n!=0)
            {  printf (" \n %d", n);
             listar(n-1);
              printf (" \n %d", n);  }
     else printf ("\n el número es cero");
  return;
}
int main(void)
{
 int a=3;
 listar(a);
 getche();
}
```



Transformar un número a Binario

```
13 | 2
1  6 | 2
   0  3 | 2
      1  1 | 2
         1    0
```

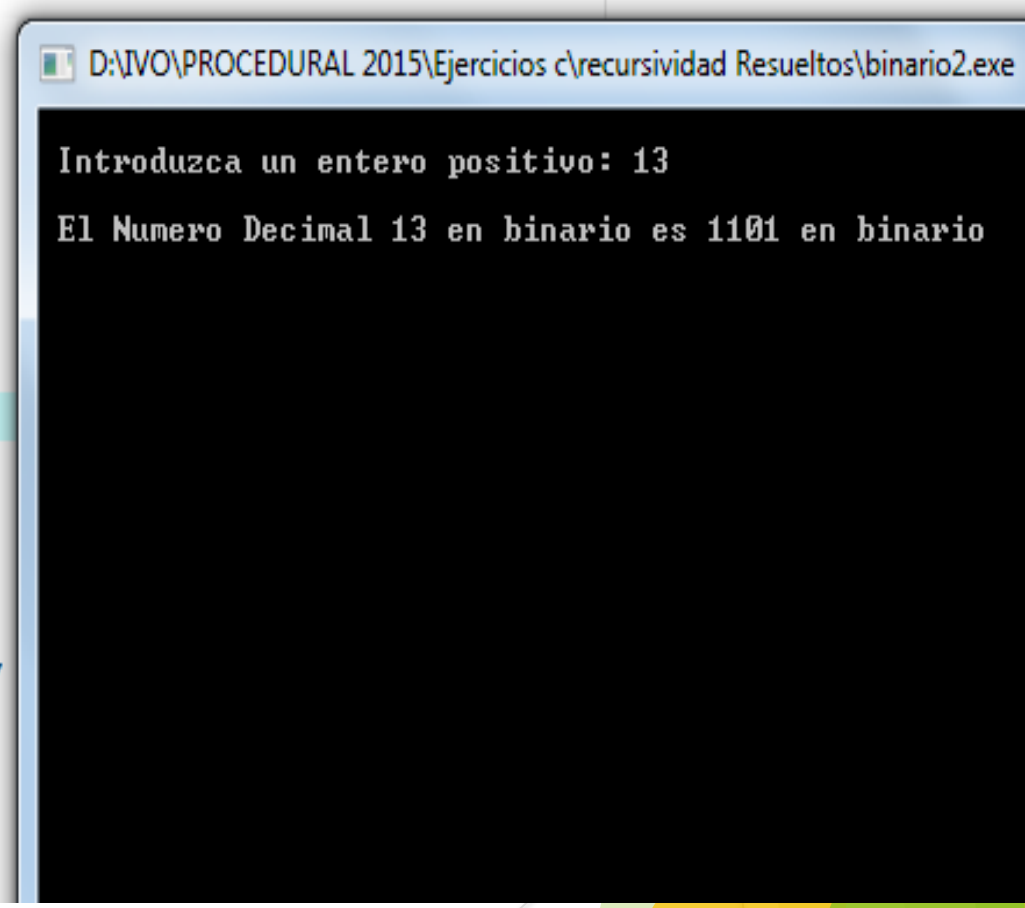
```
void Binario (int n)
{
  if (n > 0)
  {
    Binario(n/2);
    printf("%d", n % 2);
   }
}
void  main()
{
int n;
printf("\n Introduzca un entero positivo: ");
scanf("%d", &n) ;
printf("\n El Numero Decimal %d en binario es ", n);
Binario(n);
printf(" en binario");
}
```



a) Señalar caso base y general

b) Construir el Mapa de Memoria

```c
#include <conio.h>
#include <stdio.h>

void Binario (int n)
{
  if (n > 0)
   {
    Binario(n/2);
    printf("%d", n % 2);
}
}
int  main()
{
int n;
printf("\n Introduzca un entero positivo: ");
scanf("%d", &n) ;
printf("\n El Numero Decimal %d en binario es ",
Binario(n);
printf(" en binario");
getche();
}
```



Funciones recursivas que devuelven más de un resultado

Sumar recursivamente los números pares e impares menores o iguales que un valor ingresado por teclado. Realizar seguimiento y mapa de memoria para num=4

¿Qué modificaciones deberían realizarse al algoritmo utilizado anteriormente ?

```
int acumula_pares ( int xnum)
{  if  (xnum)
        if ((xnum % 2)== 0)
              return xnum + acumula_pares( xnum - 2);
        else  return(acumula_pares( xnum - 1));
   else  return (0);
}
int main()
 {   int  num;
   scanf("%d", &num);
   printf("\n suma de pares menores que:  %d es  %d ", num, acumula_pares(num));
}
```

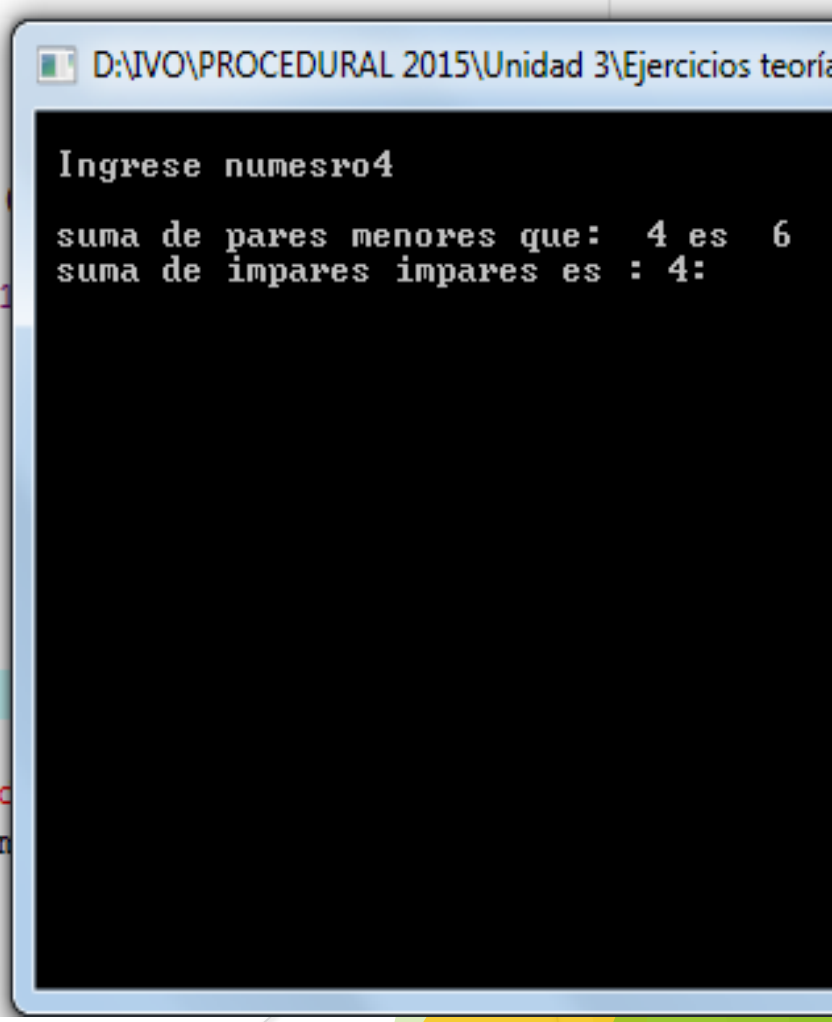
num : 4
impares: 0 imp

	xnum : 4
	área temporal : 4 -1
R1	resultado: xnum+ acu.. (3,imp)
	direcc. de retorno al main
	xnum : 3
R2	área temporal : 3 -1
	resultado:
	direcc. de retorno al punto de invocación (R1)
	xnum : 2
R3	area temporal : 2 -1
	resultado:
	direcc. de retorno al punto de invocación (R2)
	xnum : 1
R4	area temporal : 1 -1
	resultado:
	direcc. de retorno al punto de invocación (R3)
	xnum : 0
R5	resultado: 0
	Resultdo
	direcc. de retorno al punto de invocación (R4)

```
int acumula_pares_impares ( int xnum, int &imp)
{
 if  (xnum)
    if ((xnum % 2)== 0)
    return (xnum + acumula_pares_impares ( xnum - 1,imp));
    else { imp+= xnum;
            return(acumula_pares_impares ( xnum - 1,imp));
           }
 else return (0);
 }
```

```
  int main(void)
   {
    int  num;
    int impares = 0;
    printf("\n ingrese un numero");
    scanf("%d", &num);
    printf("\n  la suma de los números
                                    pares <=  %d  ", num);
    printf("es : %d:",
                acumula_pares_impares (num,impares));
     printf("\n impares es : %d:", impares);
     getch();
    }
```

```c
#include <stdio.h>
#include <conio.h>
int acumula_pares_impares ( int xnum, int &imp)
 {
    if  (xnum)
        if ((xnum % 2)== 0)
               return xnum + acumula_pares_impares (
        else { imp+= xnum;
             return(acumula_pares_impares ( xnum - 1
               }
    else return (0);
}


int main()
 {   int  num,impares=0;
    printf("\n Ingrese numesro");
    scanf("%d", &num);
    printf("\n suma de pares menores que:  %d es  %d
    printf("\n suma de impares impares es : %d:", im

    getch();
}
```



Recursividad en Arreglos

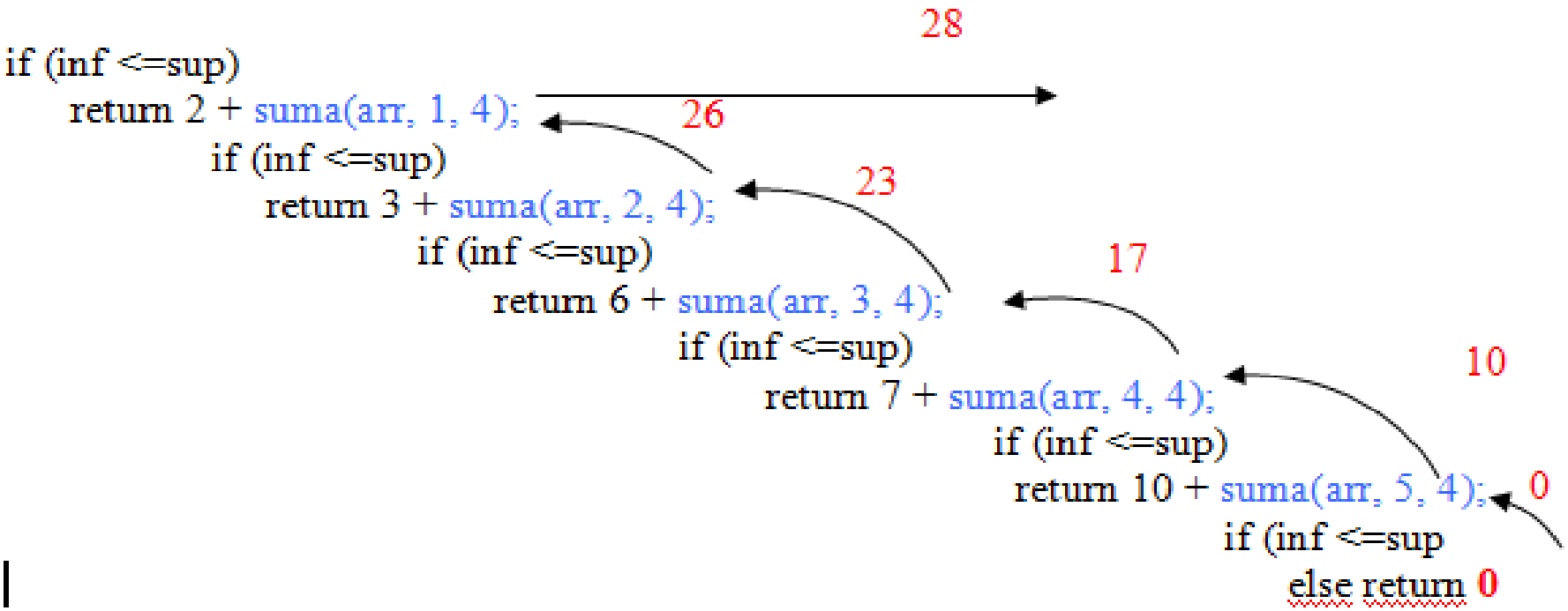
Suma recursiva de las componentes de un arreglo

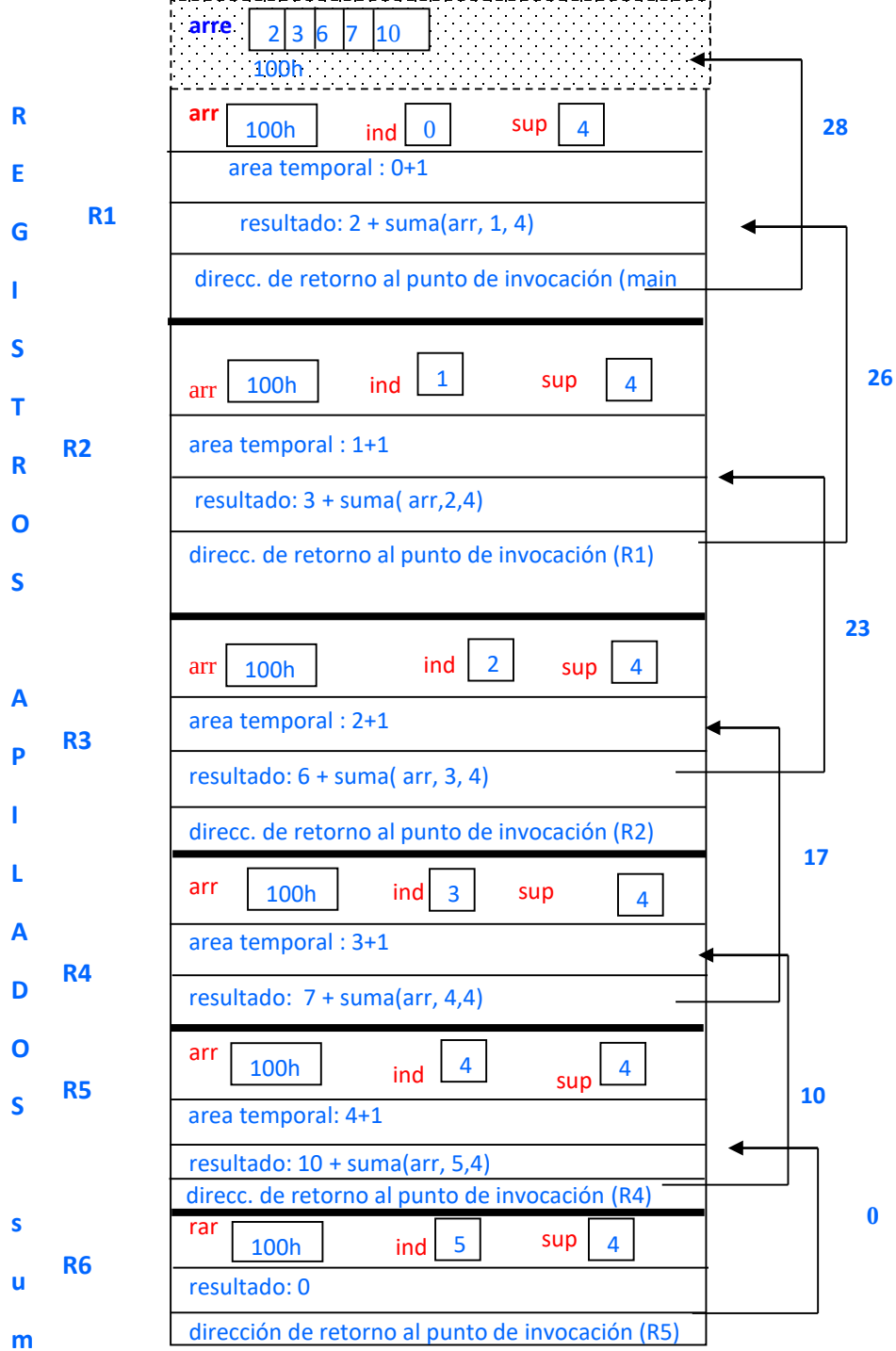
int arre[5]={2, 3, 6, 7, 10}

arre

2	3	6	7	10

Esquema de la solución:






```
#include<stdio.h>
#include<conio.h>

int suma(int arr[], int ind, int sup)
{
    if (ind <=sup)
            return arr[ind] + suma(arr, ind+1, sup);
     else return 0;
}
int main()
{
  int arre[5]={2, 3, 6, 7, 10} ;
  printf("\n Suma de  componentes %d",
                                  suma(arre, 0,4));
  getch();
}
```

Recursividad en Arreglos

```
int suma(int arr[], int ind, int sup)
{
    if (ind <=sup)
          return arr[ind] + suma(arr, ind+1, sup);
   else return 0;
}


int main()
{
  int arre[5]={2, 3, 6, 7, 10} ;
  printf("\n Suma de las componentes %d", suma(arre, 0,4));
  getch();
}
```

Modificar el algoritmo anterior de modo que la función devuelva, la suma de las componentes y el total de componentes mayores a 5. Señalar caso base y caso general

Recursividad en Arreglos

```
void main(void)
 {
  int a[n],b[n],valor;
  printf ("\n CARGA PRIMER VECTOR \n");
  carga(a,0);
  printf ("\n CARGA SEGUNDO VECTOR ");
  carga(b,0);
  printf ("\n Ingrese valor a buscar en los vectores: ");
  scanf("%d",&valor);
  if (busq_sec_rec(a,0,valor)==1)
              printf("\n valor está en  primer vector");
   else printf(" NO está en  primer vector);
  if (busq_sec_rec(b,0,valor)==1)
              printf("\n está en  segundo vector");
  else printf("\n  No esta en  segundo vector ");
  printf ("\n Producto escalar de los vectores = %d", escalar(a,b,n -1));
}
```

Calcular el producto escalar de los dos vectores y dado un valor, indicar en cual de los arreglos se encuentra.

EJEMPLO:

*La función recursiva carga es reusable, permiten cargar **arreglos de n componentes enteras** positivas.*

La función búsqueda también es reusable.

Recursividad en Arreglos

```
#define n 5
void carga (int arr[n], int i)
{
   if (i!=n)
     {  printf ("\n ingrese  valor  la posición %d:", i);
        scanf("%d",arr+i);
        carga(arr, i+1);
     }
}
```

```
int escalar (int x[n],int y[n],int j)
 {
   if (j >= 0)
          return x[j]*y[j]+ escalar(x,y,j-1);
   else  return 0;
 }
```

```
int busq_sec_rec (int arr[n], int xi, int elem)
{
   if( xi==n )
            return -1;
   else    if (arr[xi]==elem)
                  return 1;
           else  return busq_sec_rec ( arr, xi+1,elem);
 }
```

Realizar el seguimiento y el mapa de memoria cuando se ejecuta la función **muestra**.
Señalar caso base y general

```
void muestra(int arr[], int ind, int sup)
{
    if (ind <=sup)
     {
       muestra(arr, ind+1, sup);
       printf("\n %d", arr[ind]);
     }
}


int main(void)
{
   int arre[5]={2, 3, 6, 7, 10};
   muestra(arre, 0,4);
   getch();
}
```

Recursión versus Iteración

Ambas se basan en una estructura de control:

- la iteración utiliza una estructura de repetición
- la recursión utiliza una estructura de selección.

Ambas implican repetición:

- la iteración utiliza la repetición de manera explícita
- a recursión consigue la repetición mediante repetidas llamadas a una misma función.

Ambas involucran una prueba de terminación:

- la iteración termina cuando falla la condición de continuación del ciclo
- la recursión termina cuando se reconoce un caso base.