


# ALGORITMOS Y PROGRAMACIÓN

Clase 8  
Recursión

# Temario

- Recursión
  - Función recursiva
  - Pila de ejecución
- 

# Recursión

- Es una técnica que permite resolver un problema en términos de sí mismo
- *La recursión es una técnica de programación muy poderosa, en la cual una función realiza llamadas a si misma en pos de resolver un problema.*
- Razones para su uso:
  - Reemplazo de estructuras repetición.
  - Problema de naturaleza recursiva.
  - Problemas “casi” irresolubles con las estructuras iterativas.

La función *factorial* es muy usada en probabilidad y estadística.

El factorial de  $n$  se expresa como  $n!$  y está definido para los números naturales como:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 \quad 3! = 3 \cdot 2 \cdot 1$$

De manera matemática el factorial se define como:

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

La sucesión de Fibonacci también es una función definida para los números naturales. Es una secuencia de enteros donde sus elementos subsecuentes son la suma de los dos anteriores.

$$\begin{aligned} 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots & \quad \begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \quad n > 1 \end{aligned} \end{aligned}$$

# Recursión

- En un algoritmo recursivo se identifican dos elementos:
- **Caso Recursivo:** aquí el algoritmo realiza algunas operaciones con las que *se reduce la complejidad del problema* y luego realiza un *llamado a sí mismo* (autoinvocación). Puede tener 1 o más casos recursivos.
- **Caso Base:** Se da cuando el cálculo es tan simple que *se puede resolver directamente sin necesidad de hacer una llamada recursiva. Determina el corte del proceso recursivo.* Puede tener uno o más casos base.

# Recursión

- Cada autoinvocación o llamado recursiva se aplica a un problema de igual naturaleza que el original, pero de menor tamaño.
- Esta reducción nos asegura que, en algún momento, la complejidad del problema será tan simple que su solución será trivial y directa: es lo que llamamos caso base, donde se corta el proceso repetitivo y se asegura encontrar la solución al problema de partida.
- **Los algoritmos recursivos funcionan siempre que las llamadas recursivas lleguen en algún momento a uno de los casos bases. En caso contrario, se produce un loop o bucle infinito.**

# Tipos de Recursión

La recursión puede ser:

Simple: un solo llamado recursivo dentro del algoritmo

Múltiple: dos o mas llamados recursivos dentro del algoritmo

Directa: se autoinvoca dentro del cuerpo del algoritmo

Indirecta: A invoca a B, y B invoca a A

# Recursión - Ejemplo

- Definir una función recursiva que reciba un número entero y devuelva la suma desde 1 hasta dicho número.

`int sumar(int n) → return 1+2+3+....+n`

Pensemos primero cómo es la solución iterativa:

```
public int sumar(int n)
{
    int suma=0;
    for ( int j=1;j<=n; j++)
        suma=suma + j;
    return suma;
}
```



# Recursión - Ejemplo

- Podemos pensarla de otra forma equivalente:

`int sumar(int n) → return n + (n-1) + ...3+2+1`

- Con lo cual la función quedaría:

```
public int sumar(int n)
{
    int suma=0;
    for ( int j=n; j>=1; j--)
        suma=suma + j;
    return suma;
}
```

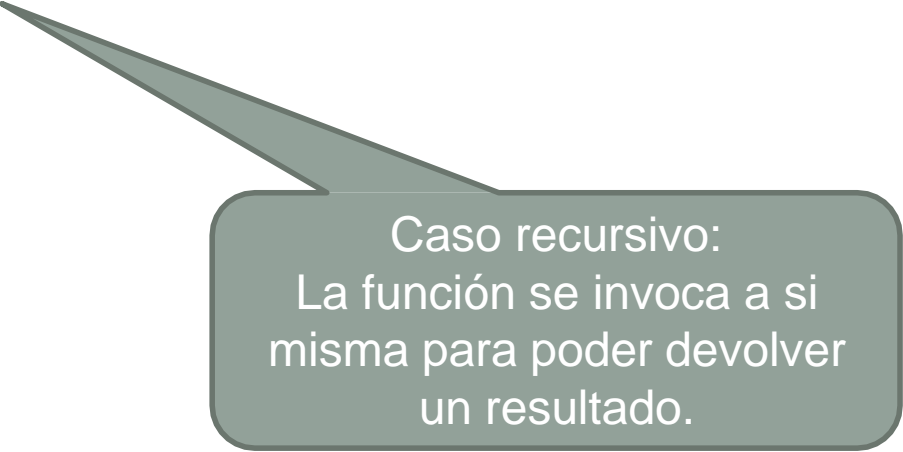
Como planteamos la solución recursiva???

# Recursión - Ejemplo

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

# Recursión - Ejemplo

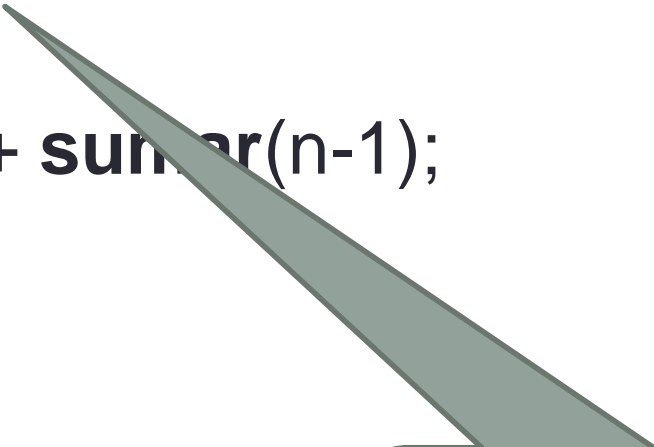
```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



Caso recursivo:  
La función se invoca a si misma para poder devolver un resultado.

# Recursión - Ejemplo

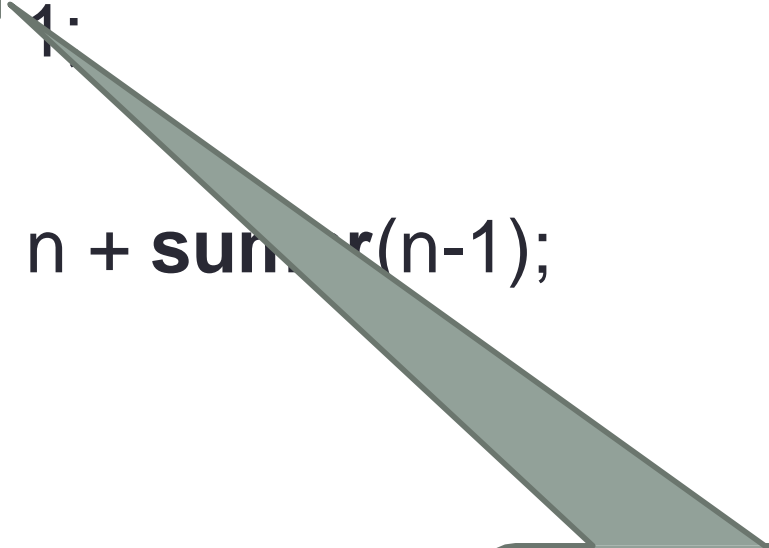
```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



Caso base:  
La función finaliza con las  
llamadas recursivas.

# Recursión - Ejemplo

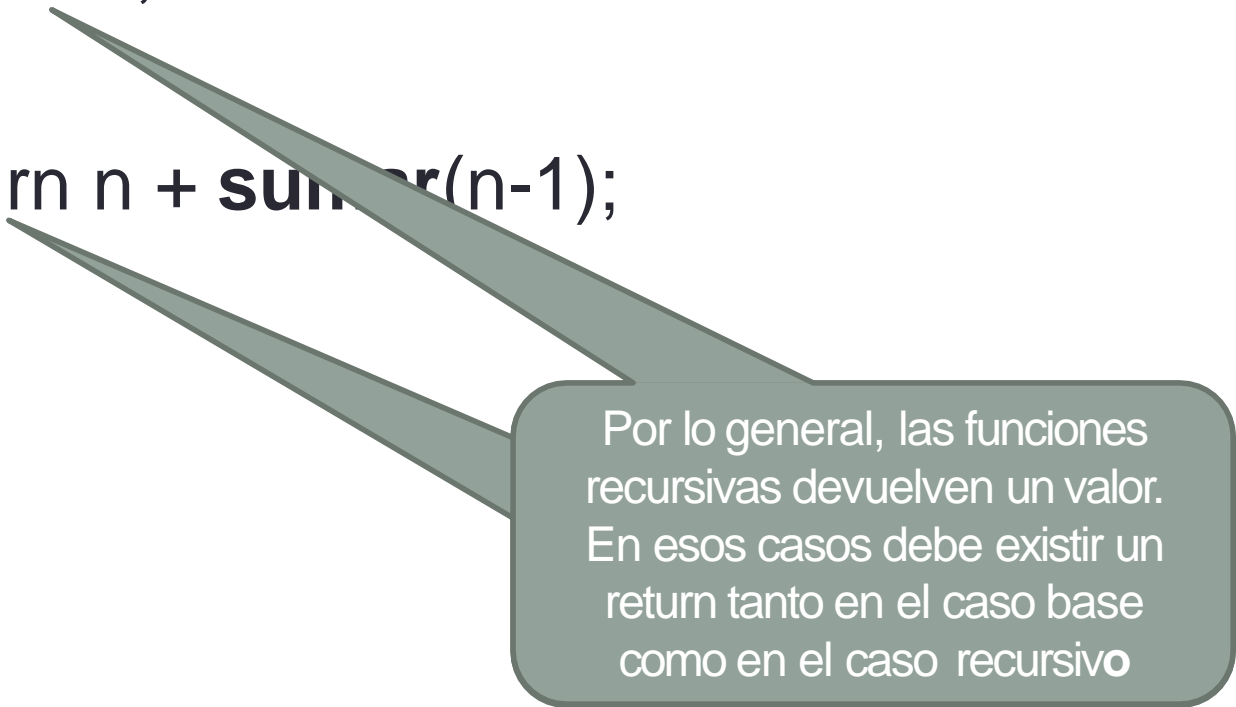
```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



La ejecución del caso base o el caso recursivo depende de una condición.

# Recursión - Ejemplo

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



Por lo general, las funciones recursivas devuelven un valor. En esos casos debe existir un return tanto en el caso base como en el caso recursivo

# Pila de ejecución

- Una función recursiva se invoca desde el Main como a cualquier otra.

```
int resultado = sumar(5);
```

- ¿Qué sucede en la memoria de la computadora cuando se ejecuta una función recursiva?

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

En nuestro ejemplo n vale 5.  
Al ejecutar la función se reserva memoria para la ejecución de la función y poder almacenar esa variable. Se crea un registro de activación (R.A.).

n = 5



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

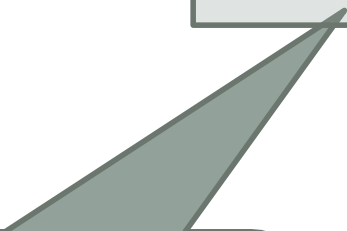
Con  $n == 5$  se ejecuta el caso recursivo.

¿Qué tiene que hacer la función para obtener el resultado a devolver?

$n = 5$

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



La función se invoca a si misma con el valor 4.

n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Al ejecutar la función por segunda vez se reserva memoria para almacenar la variable  $n$ .  
Notar que NO es la misma posición de memoria, hay dos posiciones de memoria distintas para la variable  $n$ .  
Cada llamada de la función tiene su propio ámbito (R.A.) donde almacena sus variables.

$n = 4$

$n = 5$

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Nuevamente la función se  
invoca a si misma con  
 $n=3$

$n = 4$

$n = 5$

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Una nueva reserva de Memoria. Se apila un nuevo R.A.

n = 3

n = 4

n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Nuevamente la función se  
invoca a si misma con  
 $n=2$

$n = 3$

$n = 4$

$n = 5$

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Una nueva reserva de  
Memoria. Se apila un  
nuevo R.A.

n = 2

n = 3

n = 4

n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Nuevamente la función se invoca a si misma con  $n=1$ .

$n = 2$

$n = 3$

$n = 4$

$n = 5$



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Una nueva reserva de  
memoria

n = 1

n = 2

n = 3

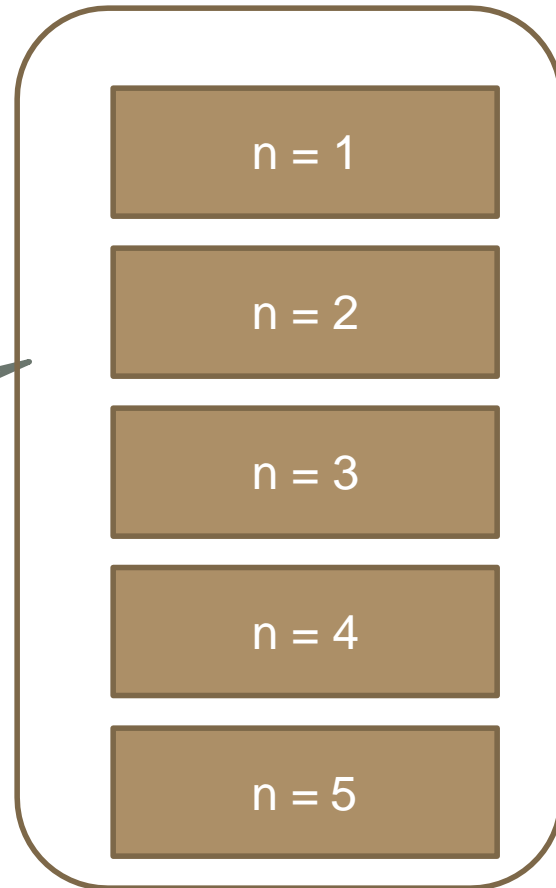
n = 4

n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Esto es lo que se conoce como pila de ejecución.  
Reservas de memoria consecutivas donde se almacena el estado de la función en cada una de sus ejecuciones.



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

¿Qué sucede cuando n vale 1?

n = 1

n = 2

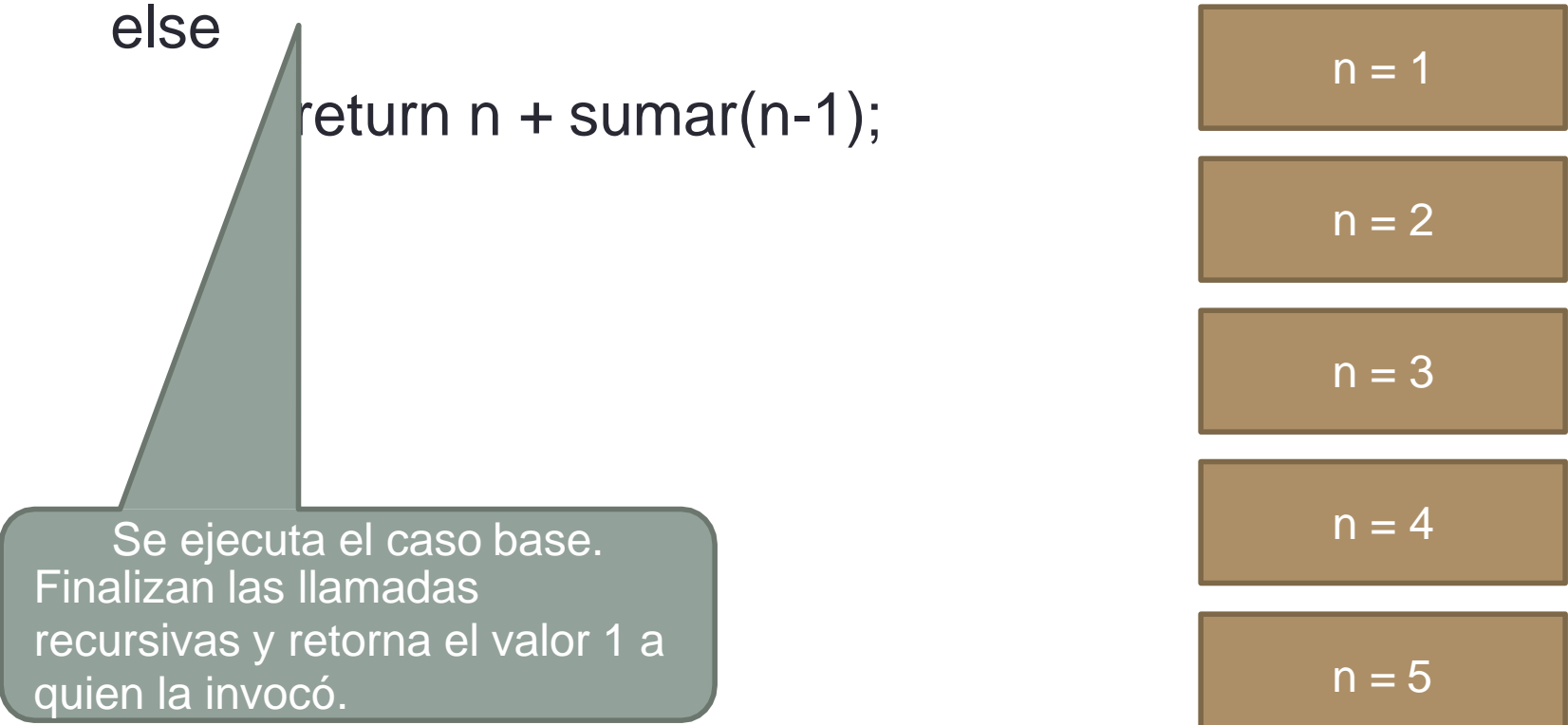
n = 3

n = 4

n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```



Se ejecuta el caso base.  
Finalizan las llamadas  
recursivas y retorna el valor 1 a  
quien la invocó.

n = 1

n = 2

n = 3

n = 4

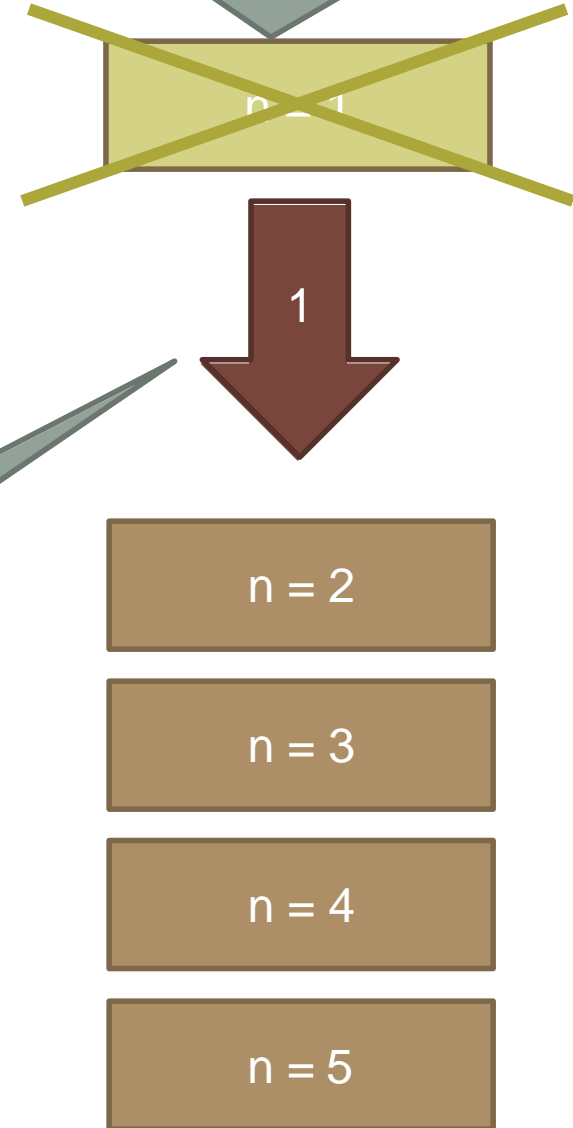
n = 5

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

En este caso la función devuelve el valor 1. Este valor lo recibe el estado de la función sumar que hizo la llamada.

Al finalizar se libera la memoria reservada para su ejecución.

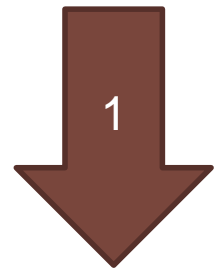


# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

En este punto se recibe el valor 1. Por lo tanto, sabiendo el valor devuelto por la función sumar, ya puede realizar la operación y devolver el resultado de la suma.

Acá  $n = 2 \rightarrow 2 + 1 = 3$



$n = 2$

$n = 3$

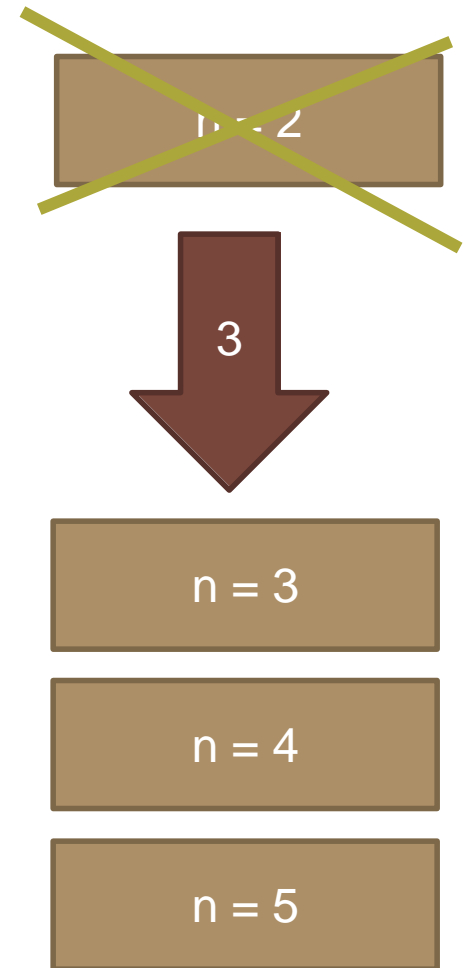
$n = 4$

$n = 5$

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

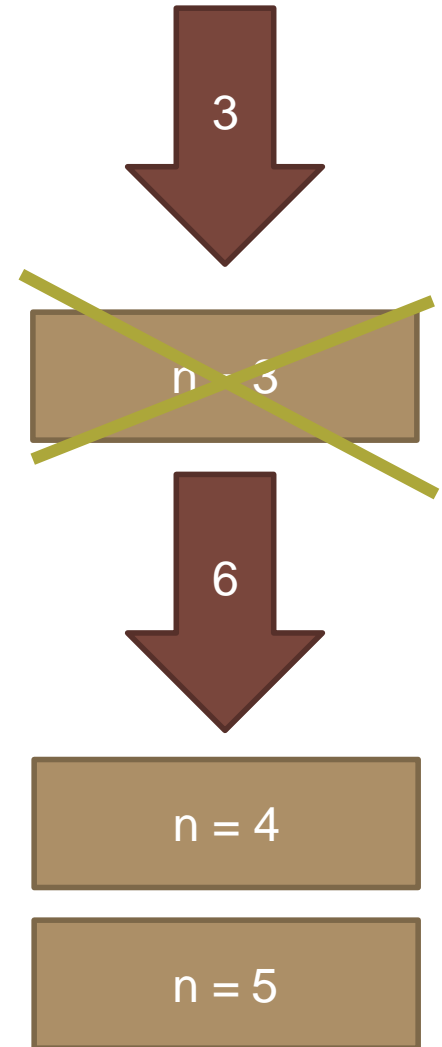
El resultado de esta suma lo recibe el estado anterior de la función.



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

En este estado n vale 3, se le suma 3 (el valor recibido como resultado) y se devuelve la suma  
 $3 + 3 = 6$

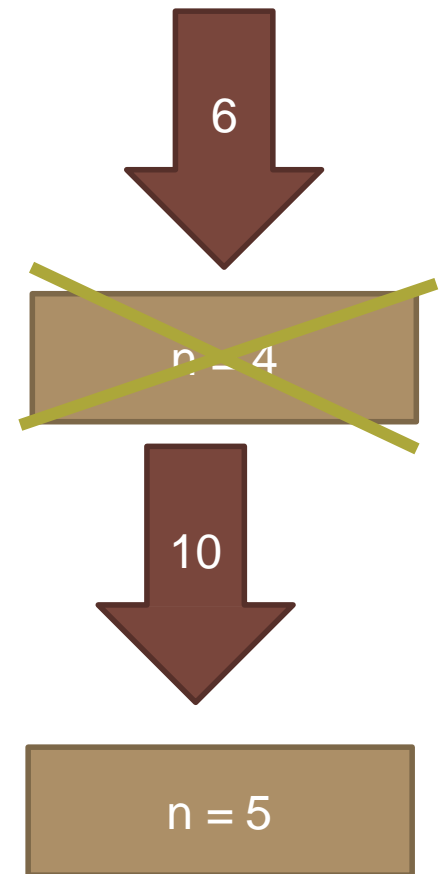




# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

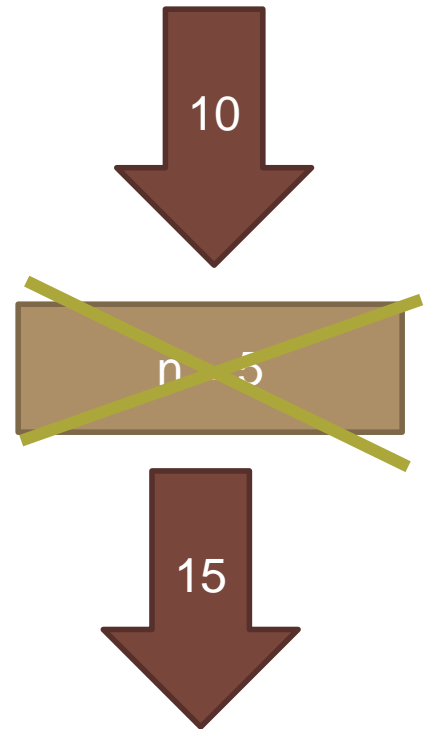
En este estado  $n$  vale 4, se le suma 6 (el valor recibido como resultado) y se devuelve la suma  $4 + 6 = 10$



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

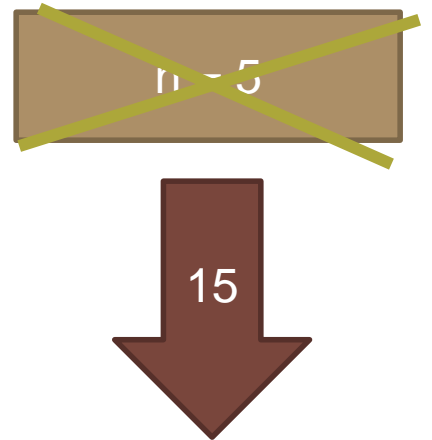
En este estado n vale 5, se le suma 10 (el valor recibido como resultado) y se devuelve la suma  $5 + 10 = 15$



# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return 1;  
    else  
        return n + sumar(n-1);  
}
```

Este estado es el último en ejecutarse. Notar que fue el primero en ejecutarse. El valor de retorno se devuelve a donde se hizo la primer llamada.



# Pila de ejecución

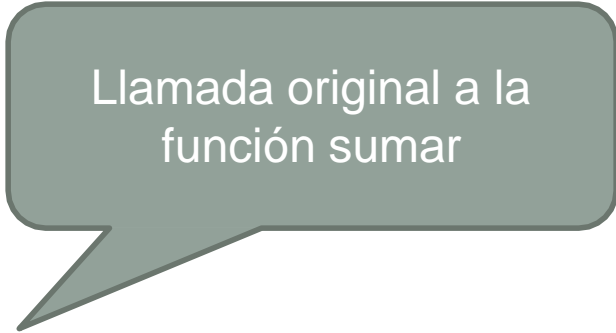
```
int resultado = sumar(5),
```

Llamada original a la  
función sumar

El valor recibido (15) se  
almacena en la variable  
"resultado".

# Pila de ejecución

```
int resultado = 15;
```



Llamada original a la  
función sumar

# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

Veamos el paso a paso de nuevo sin las pilas:

resultado=sumar(5)=?

sumar → n=5

→ 5+ sumar(4)

Sumar n=4

→ 4+ sumar(3)

Sumar n=3

→ 3 + sumar(2)

Sumar n=2

→ 2+ sumar(1)

Sumar n=1

→ Sumar=1

# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar → n=5

→ 5+ sumar(4)

Sumar n=4

→ 4+ sumar(3)

Sumar n=3

→ 3 + sumar(2)

Sumar n=2

→ 2+ sumar(1)

Sumar n=1

→ Sumar=1



# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar → n=5

→ 5+ sumar(4)

Sumar n=4

→ 4+ sumar(3)

Sumar n=3

→ 3 + sumar(2)

Sumar n=2

→ 2+ sumar(1)=2+**1**

Sumar n=1

→ Sumar=**1**





# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar → n=5

→ 5+ sumar(4)

Sumar n=4

→ 4+ sumar(3)

Sumar n=3

→ 3 + sumar(2)

Sumar n=2

→ 2+ sumar(1)=2+1=3

Sumar n=1

→ Sumar=1



# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar → n=5

→ 5+ sumar(4)

Sumar n=4

→ 4+ sumar(3)

Sumar n=3

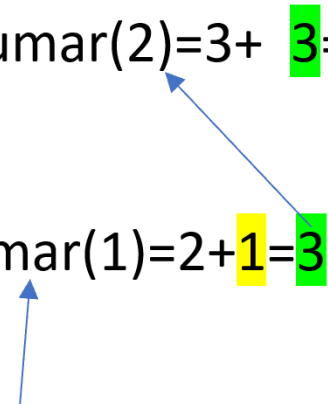
→ 3 + sumar(2)=3+ 3=

Sumar n=2

→ 2+ sumar(1)=2+1=3

Sumar n=1

→ Sumar=1



# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar  $\rightarrow$  n=5

$\rightarrow$  5+ sumar(4)

Sumar n=4

$\rightarrow$  4+ sumar(3)

Sumar n=3

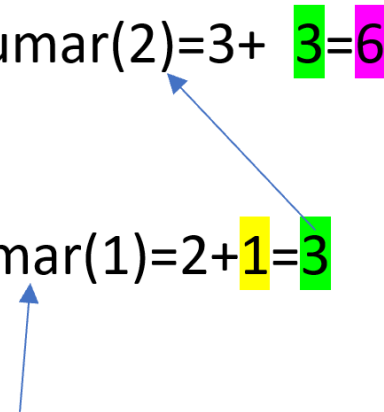
$\rightarrow$  3 + sumar(2)=3+ 3=6

Sumar n=2

$\rightarrow$  2+ sumar(1)=2+1=3

Sumar n=1

$\rightarrow$  Sumar=1



# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar  $\rightarrow$  n=5

$\rightarrow$  5+ sumar(4)

Sumar n=4

$\rightarrow$  4+ sumar(3)=4+ 6= 10

Sumar n=3

$\rightarrow$  3 + sumar(2)=3+ 3=6

Sumar n=2

$\rightarrow$  2+ sumar(1)=2+1=3

Sumar n=1

$\rightarrow$  Sumar=1



# Pila de ejecución

```
int sumar (int n)
{
    if (n == 1)
        return 1;
    else
        return n + sumar(n-1);
}
```

resultado=sumar(5)=

sumar  $\rightarrow$  n=5

$\rightarrow 5 + \text{sumar}(4) = 5 + 10 = 15$

Sumar n=4

$\rightarrow 4 + \text{sumar}(3) = 4 + 6 = 10$

Sumar n=3

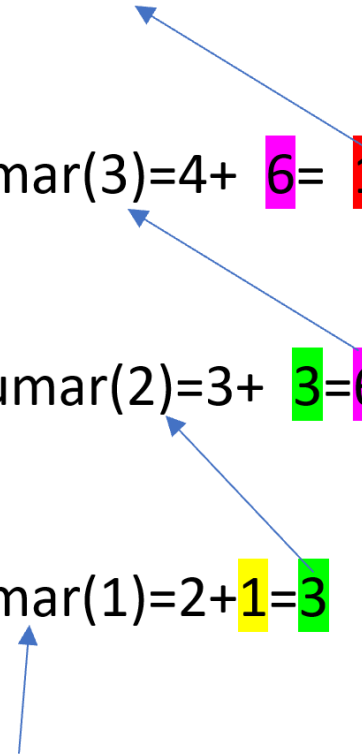
$\rightarrow 3 + \text{sumar}(2) = 3 + 3 = 6$

Sumar n=2

$\rightarrow 2 + \text{sumar}(1) = 2 + 1 = 3$

Sumar n=1

$\rightarrow \text{Sumar} = 1$



# Pila de ejecución

resultado=sumar(5)=15

sumar  $\rightarrow$  n=5

$\rightarrow 5 + \text{sumar}(4) = 5 + 10 = 15$

Sumar n=4

$\rightarrow 4 + \text{sumar}(3) = 4 + 6 = 10$

Sumar n=3

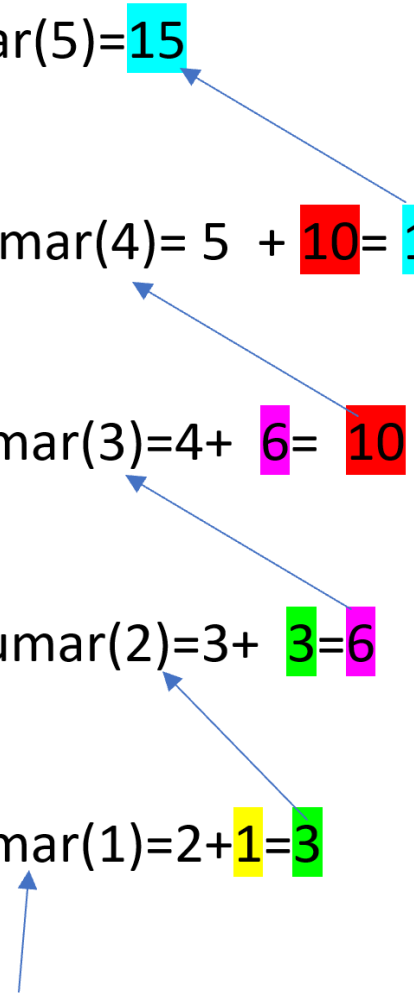
$\rightarrow 3 + \text{sumar}(2) = 3 + 3 = 6$

Sumar n=2

$\rightarrow 2 + \text{sumar}(1) = 2 + 1 = 3$

Sumar n=1

$\rightarrow \text{Sumar} = 1$



# Pila de ejecución

¿ Qué sucede con esta llamada ?

```
int resultado = sumar (-1);
```

# Pila de ejecución

```
int sumar (int n){  
    if (n == 1)  
        return n;  
    else  
        return n + sumar(n-1);  
}
```



A diagram of a call stack, represented as a vertical container with a pointed bottom. It contains a list of values: -1, -2, -3, -4, -5, and an ellipsis (...). A line points from the recursive call in the code to the bottom of the stack.

-1  
-2  
-3  
-4  
-5  
...

- Si una función recursiva no alcanza nunca el caso base, seguirá haciendo llamadas recursivas para siempre y nunca terminaría.
- Esta circunstancia se conoce como *recursión infinita*, y produce error de ejecución.

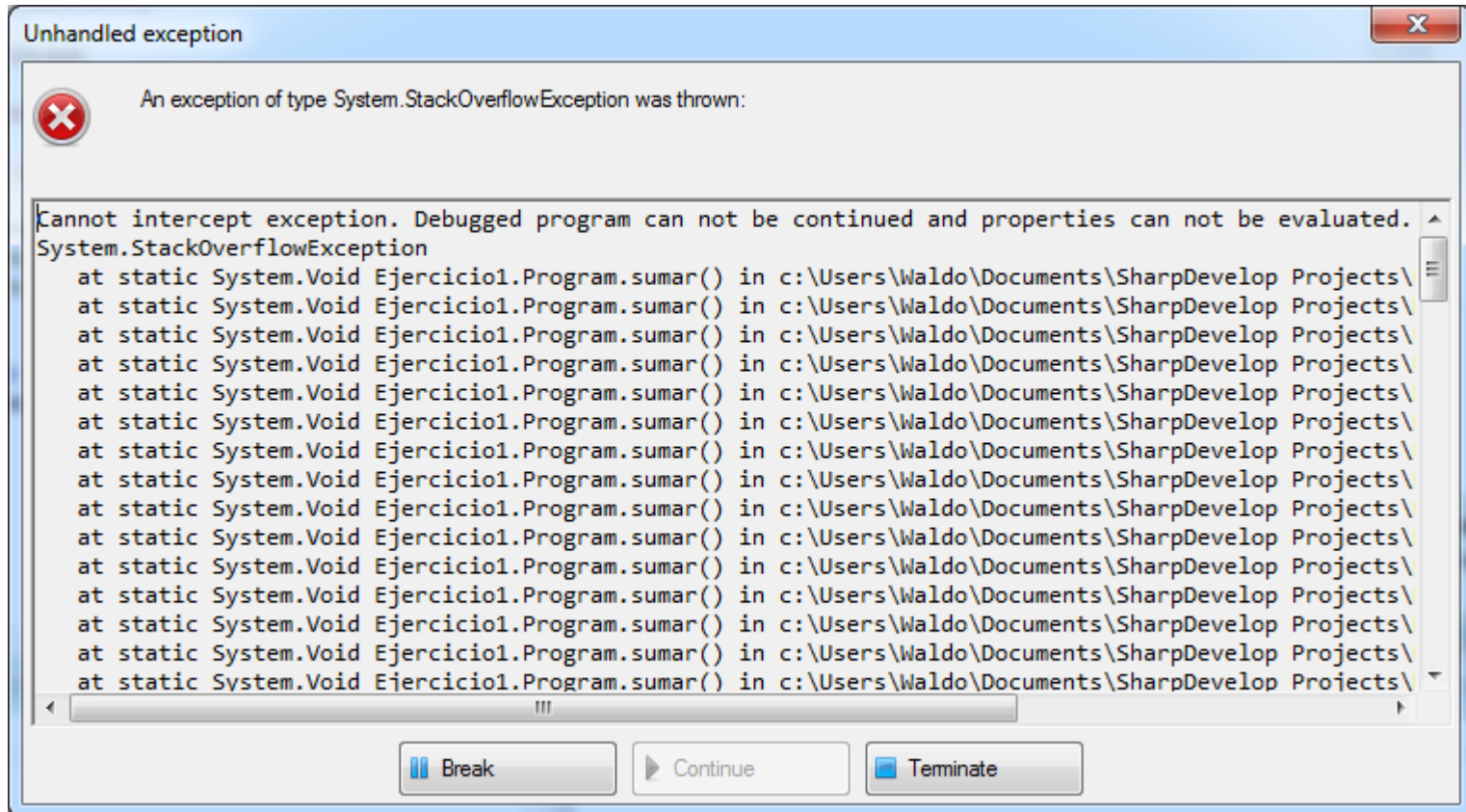


# Pila de ejecución

- En realidad el verdadero problema de la recursión es la cantidad máxima de memoria con la cual cuenta un programa para su ejecución.
- Si una función recursiva hace muchas llamadas, como cada llamada reserva memoria, ésta podría acabarse.
- Además, si cada estado o R.A. de una llamada recursiva debe almacenar muchos datos, la memoria también podría acabarse con pocas llamadas.

# Pila de ejecución

- Cuando la memoria se llena .NET levanta una excepción llamada StackOverflowException



# Ventajas y desventajas del uso de recursión

## Ventajas

Permite desarrollar :

- Soluciones elegantes.
- Soluciones más simples

## Desventajas

En cuanto a la eficiencia la recursion:

- ocupa más memoria que la misma solución iterativa,
- demanda mayor tiempo de ejecución debido al tiempo de apilado y desapilado de cada R.A.

# Aplicación de la recursión

- Es muy común implementar funciones recursivas que trabajen sobre colecciones de datos: strings, arreglos, ArrayList, Pilas, Colas.
- En estos casos se trabaja siempre sobre la misma idea, resolver solo una parte del problema con un elemento y el resto del problema (los elementos restantes) se resuelven de manera recursiva:
  - En el caso recursivo se debe sacar un elemento de la colección, para tratar el resto de la colección de manera recursiva
  - Por lo general el caso base se da cuando la colección queda vacía o con un único elemento.

# Recursión – Ejemplo 1

- Implementar un algoritmo recursivo que permita imprimir un arreglo de 10 elementos.

```
public static void Main(string[] args)
{
    int [] vec=new int [10];
    int tam=10;
    int pos=0;

    imprime(vec,pos,tam); /*le paso el vector, la posición inicial y el tamaño del vector*/
    .....
}
```

# Recursión – Ejemplo 1

- ```
public static void imprime ( int [] v, int k, int dim)
{
    if (k<dim)                /*condición de repetición */
    {
        Console.WriteLine( v[k]);
        imprime ( v, k+1, dim);    /*llamado recursivo*/
    }
}
```
- En este ejemplo el caso base está implícito: cuando se llega a que  $k==dim$ , significa que el vector se terminó y no hay nada más que imprimir. No hace nada, por eso no está codificado. Solo retorna la ejecución al estado anterior.
- Observación: es un procedimiento (o una función void que no lleva return)

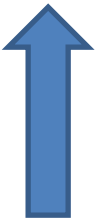
# Recursión – Ejemplo 1

- ```
public static void imprime ( int [] v, int k, int dim)
{
    if (k<dim -1)                /*condición de repetición */
    {
        Console.WriteLine( v[k]);
        imprime ( v, k+1, dim);    /*llamado recursivo*/
    }
    else
        Console.WriteLine( v[dim -1]); /*caso base*/
}
```

- En este ejemplo el caso base está explícito

## Seguimiento de Pila de ejecución (de abajo hacia arriba se lee)

```
public static void imprime ( int [] v, int k, int  
dim)  
{  
    if (k<dim)  
    {  
        Console.WriteLine( v[k]);  
        imprime ( v, k+1, dim);  
    }  
}
```



v=2,3,6

k=0

dim=3

WriteLine v[0]

En pantalla se ve:

2



## Seguimiento de Pila de ejecución (de abajo hacia arriba se lee)

```
public static void imprime ( int [] v, int k, int  
dim)  
{  
    if (k<dim)  
    {  
        Console.WriteLine( v[k]);  
        imprime ( v, k+1, dim);  
    }  
}
```

V=2,3,6

K=1

Dim=3

WriteLine v[1]

V=2,3,6

K=0

Dim=3

WriteLine v[0]

En pantalla se ve:

2

3

## Seguimiento de Pila de ejecución (de abajo hacia arriba se lee)

```
public static void imprime ( int [] v, int k, int  
dim)  
{  
    if (k<dim)  
    {  
        Console.WriteLine( v[k]);  
        imprime ( v, k+1, dim);  
    }  
}
```

V=2,3,6

K=2

Dim=3

WriteLine v[2]

V=2,3,6

K=1

Dim=3

WriteLine v[1]

V=2,3,6

K=0

Dim=3

WriteLine v[0]

En pantalla se ve:

2

3

6

V=2,3,6

K=3

Dim=3

V=2,3,6

K=2

Dim=3

WriteLine v[2]

V=2,3,6

K=1

Dim=3

WriteLine v[1]

V=2,3,6

K=0

Dim=3

WriteLine v[0]

Seguimiento de Pila de ejecución  
(de abajo hacia arriba se lee)

Llegó al caso base (k=3) , no hace nada. Empieza a desapilar los R.A.

```
public static void imprime ( int [] v, int k, int dim)
{
    if (k<dim)
    {
        Console.WriteLine( v[k]);
        imprime ( v, k+1, dim);
    }
}
```

En pantalla se ve:

2

3

6

V=2,3,6

K=2

Dim=3

WriteLine v[2]

En pantalla se ve:

2

3

6

Retorna el control a la caja previa,  
en el camino de regreso al Main.

V=2,3,6

K=1

Dim=3

WriteLine v[1]

V=2,3,6

K=0

Dim=3

WriteLine v[0]

En pantalla se ve:

2

3

6

V=2,3,6

K=1

Dim=3

WriteLine v[1]

V=2,3,6

K=0

Dim=3

WriteLine v[0]

En pantalla se ve:

2

3

6

V=2,3,6

K=0

Dim=3

WriteLine v[0]

En pantalla se ve:

2

3

6

# Recursión – Ejemplo 1 b)

- Definir una función recursiva que imprima los elementos pares del vector

```
public static void impriPares ( int [] v, int k, int dim)
{
    if (k<dim)                                /*condición de repetición */
    {
        if ( v[k] % 2 == 0)
            Console.WriteLine( v[k]);

        impriPares ( v, k+1, dim);            /*llamado recursivo*/
    }
}
```

V=[12,3,56,7,19,22,33]



# Recursión – Ejemplo 2

- Implementar una función recursiva que permita cargar un arreglo de 10 elementos.

```
public static void Main(string[] args)
{
    int [] vec=new int [5];
    int tam=5;
    int pos=0;

    cargar(ref vec,pos,tam);  /* el vector se pasa por referencia porque se cambia su contenido*/
    ....
}
```

# Recursión – Ejemplo 2

- ```
public static void cargar( ref int [] v, int k, int dim)
{
    if (k<dim)
    {
        Console.WriteLine("ingrese un valor entero: ");
        v[k]= int.Parse(Console.ReadLine());
        cargar(ref v, k+1, dim);
    }
}
```

- En este ejemplo otra vez el caso base está implícito.

# Recursión – Ejemplo 3

- Implementar una función recursiva que permita contar los elementos de un ArrayList.

```
public static void Main(string[] args)
{
    ArrayList lis = new ArrayList();

    int pos, cont;
    string resp;

    Console.WriteLine("ingresa datos (s/n)?");
    resp = Console.ReadLine();
    while (resp == "s")
    {
        Console.WriteLine("ingrese un valor");
        lis.Add( int.Parse(Console.ReadLine()));
        Console.WriteLine("ingresa datos (s/n)?");
        resp = Console.ReadLine();
    }
    pos = 0;
    cont = cuenta(lis, pos);
    Console.WriteLine("la cantidad de elementos es: {0} ", cont);

    Console.Write("Press any key to continue . . . ");
    Console.ReadKey(true);
}
```

```
public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1 + cuenta(lista, k+1);
    }
}
```

```

public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1 + cuenta(lista, k+1);
    }
}

```

cont=cuenta( [2,14,5,18], 0 )

cuenta( [2,14,5,18], 0)  $\rightarrow$  1 + cuenta([2,14,5,18], 1)

cuenta( [2,14,5,18], 1)  $\rightarrow$  1 + cuenta( [2,14,5,18], 2)

cuenta( [2,14,5,18], 2)  $\rightarrow$  1 + cuenta ([2,14,5,18],3)

cuenta( [2,14,5,18], 3)  $\rightarrow$  1 + cuenta ([2,14,5,18],4)

cuenta( [2,14,5,18], 4)  $\rightarrow$  0

```
public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1+ cuenta(lista, k+1);
    }
}
```

cont=cuenta( [2,14,5,18], 0 ) =??


cuenta( [2,14,5,18], 0)  $\rightarrow$  1 + cuenta ([2,14,5,18], 1)

cuenta( [2,14,5,18], 1)  $\rightarrow$  1 + cuenta([2,14,5,18], 2)

cuenta([2,14,5,18], 2)  $\rightarrow$  1 + cuenta ([2,14,5,18], 3)

cuenta([2,14,5,18], 3)  $\rightarrow$  1 + cuenta ([2,14,5,18],4) = 1 + 0 = 1

cuenta([2,14,5,18], 4)  $\rightarrow$  0



```
public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1+ cuenta(lista, k+1);
    }
}
```

cont=cuenta( [2,14,5,18], 0 ) =???

cuenta( [2,14,5,18], 0)  $\rightarrow$  1 + cuenta([2,14,5,18], 1)

cuenta( [2,14,5,18], 1)  $\rightarrow$  1 + cuenta( [2,14,5,18], 2)

cuenta( [2,14,5,18], 2)  $\rightarrow$  1 + cuenta ([2,14,5,18],3) = 1 + 1 = 2

cuenta( [2,14,5,18], 3)  $\rightarrow$  1 + cuenta ([2,14,5,18],4) = 1 + 0 = 1

cuenta( [2,14,5,18], 4)  $\rightarrow$  0



```
public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1+ cuenta(lista, k+1);
    }
}
```

cont=cuenta( [2,14,5,18], 0 ) =???

cuenta( [2,14,5,18], 0)  $\rightarrow$  1 + cuenta([2,14,5,18], 1)

cuenta( [2,14,5,18], 1)  $\rightarrow$  1 + cuenta( [2,14,5,18], 2) = 1 + 2 = 3

cuenta( [2,14,5,18], 2)  $\rightarrow$  1 + cuenta ([2,14,5,18],3) = 1 + 1 = 2

cuenta( [2,14,5,18], 3)  $\rightarrow$  1 + cuenta ([2,14,5,18],4) = 1 + 0 = 1

cuenta( [2,14,5,18], 4)  $\rightarrow$  0

```

public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1+ cuenta(lista, k+1);
    }
}

```

cont=cuenta( [2,14,5,18], 0 ) = ???

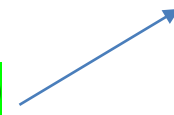
cuenta( [2,14,5,18], 0 )  $\rightarrow$  1 + cuenta([2,14,5,18], 1) = 1 + 3 = 4

cuenta( [2,14,5,18], 1 )  $\rightarrow$  1 + cuenta( [2,14,5,18], 2 ) = 1 + 2 = 3

cuenta( [2,14,5,18], 2 )  $\rightarrow$  1 + cuenta ([2,14,5,18],3) = 1 + 1 = 2

cuenta( [2,14,5,18], 3 )  $\rightarrow$  1 + cuenta ([2,14,5,18],4) = 1 + 0 = 1

cuenta( [2,14,5,18], 4 )  $\rightarrow$  0



```

public static int cuenta( ArrayList lista, int k)
{
    if (k== lista.Count)
        return 0;
    else
    {
        return 1+ cuenta(lista, k+1);
    }
}

```

cont=cuenta( [2,14,5,18], 0 ) = 4

cuenta( [2,14,5,18], 0 )  $\rightarrow$  1 + cuenta([2,14,5,18], 1) = 1 + 3 = 4

cuenta( [2,14,5,18], 1 )  $\rightarrow$  1 + cuenta( [2,14,5,18], 2) = 1 + 2 = 3

cuenta( [2,14,5,18], 2 )  $\rightarrow$  1 + cuenta ([2,14,5,18],3) = 1 + 1 = 2

cuenta( [2,14,5,18], 3 )  $\rightarrow$  1 + cuenta ([2,14,5,18],4) = 1 + 0 = 1

cuenta( [2,14,5,18], 4 )  $\rightarrow$  0

