RECURSIVIDAD

Sabemos ya, que un programa puede dividirse en subprogramas y que un subprograma puede invocar a otro subprograma. En algunos lenguajes de programación, los *subprogramas pueden invocarse a sí mismos*. Una función o procedimiento que puede invocarse a sí mismo, se denomina *recursivo.*

- En el cuerpo de sentencias del subprograma se invoca al propio subprograma para resolver "una versión más pequeña" del problema original.
- Cuando el subprograma se invoca a sí mismo, debe hacerse modificando el valor de uno o más argumentos.
- Habrá al menos un caso que pueda resolverse directamente sin necesidad de hacer una llamada recursiva. Si no existiera al menos un caso no recursivo, la secuencia de invocaciones ocurrirían de modo infinito (en teoría) y la función no terminaría de modo controlado, no alcanzando la solución prevista.

Las razones de uso de la recursividad son:

- Problemas "casi" irresolubles con estructuras iterativas
- Solución natural, aunque no necesariamente preferida en términos de eficiencia, a problemas que son inherentemente recursivos
- Soluciones elegantes y simples

EJEMPLOS de tipo MATEMATICOS

Para introducir el tema, es habitual tomar definiciones inherentemente recursivas como las siguientes:

Factorial (precondición : n es positivo)

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

Potencia (precondición : a es distinto de 0)

$$a^b = \begin{cases} 1 & \text{si } b = 0 \\ a & \text{si } b = 1 \\ a * a^{b-1} & \text{si } b > 1 \end{cases}$$

Analizar: b negativo

Fibonacci (precondición : n es positivo)

$$Fib(n) = \begin{cases} 1 & \text{si } n = 1 \vee n = 2 \\ Fib(n-1) + Fib(n-2) & \text{si } n > 2 \end{cases}$$

SOLUCION RECURSIVA para el CALCULO del FACTORIAL

Caso Base (no recursivo) 0! = 1! = 1

Caso Recursivo n! = n * (n-1)!

Por ejemplo:

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1

Entonces: 5! = 5 * 4 * 3 * 2 * 1 =120

Esto implica claramente una definición recursiva, pues el cálculo del factorial de un número, necesita del factorial del número anterior.

En C el código de la función Factorial sería:

```
long facto(int n) {
 if (n==0 || n==1)
   return 1;
 else
   return n * facto(n-1);
};
```

Caso NO RECURSIVO

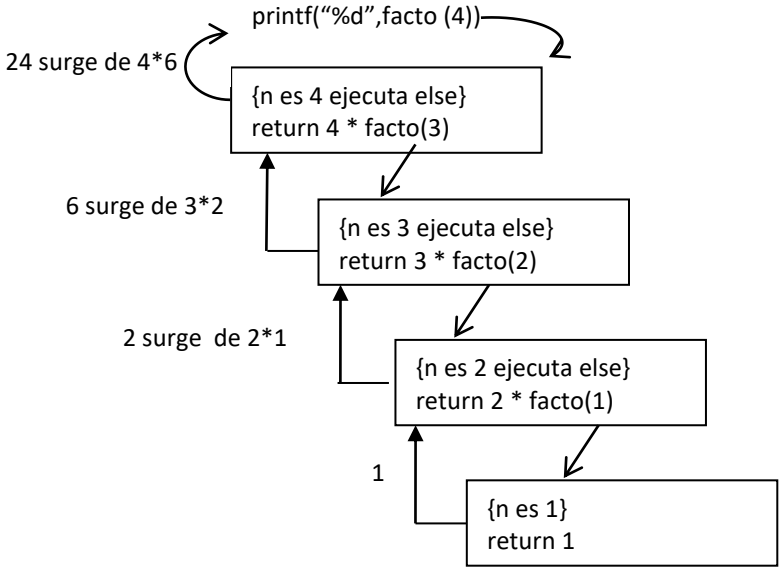
Caso RECURSIVO. Invocación con argumento modificado

La invocación de la función para calcular 4! sería:

```
printf("%l", facto(4));
```

SEGUIMIENTO de la SOLUCION del CALCULO del FACTORIAL

Es importante notar que, como en cualquier subprograma, cada invocación genera una nueva instancia del subprograma invocado que en el caso de la recursividad, son instancias del mismo subprograma. Esto se puede ilustrar de la siguiente forma para el cálculo del factorial de 4:



El resultado se obtiene como una expresión que se va armando "a la vuelta" de la recursividad. Es importante entonces que el caso base se ejecute retornado el valor 1 (neutro del producto).

En general podemos afirmar que en la salida no recursiva, las funciones deben retornar un valor que sea el caso base del cálculo y siempre debe existir una condición que permita ejecutar un caso no recursivo.

A modo de ejemplo, se presentan las siguientes funciones erróneas:

```
long facto(int n) {
 if (n>1)
     return n * facto(n-1);
 }
```

Termina, pero falla en el resultado

```
long facto(int n) {
    return n * facto(n-1);
 }
```

Recursividad "infinita"

Actividad A Realizar el seguimiento indicando qué muestran las siguientes funciones

1)
```
void quehace1(int n) {
  if (n>0) {
    printf("%d ", n);
    quehace1(n-2);
  }
 };
```
Invocación: quehace1(6);

2)
```
void quehace2(int n) {
  if (n>0) {
    quehace2(n-2);
    printf("%d ", n);
   }
};
```
Invocación: quehace2(6);

3)
```
void quehace3(int n) {
    printf("%d ", n);
    quehace3(n-2);
};
```
Invocación: quehace3(7);

4)
```
void quehace4(int n) {
  if (n!=0) {
      printf("%d ", n);
      quehace4(n-2);
   }
  };
```
Invocación: quehace4(7);

5)
```
void quehace5(int n) {
  if (n>0) {
    printf("%d ", n);
    quehace5(n-2);
    printf("%d", n);
  }
}
```
Invocación: quehace5(6);

6)
```
void quehace6(int n) {
  if (n>0) {
    quehace6(n--);
    printf("%d ", n);
  }
}
```
Invocación: quehace6(6);

7)
```
void quehace7(int n) {
  if (n>0) {
    quehace7(--n);
    printf("%d ", n);
  }
}
```
Invocación: quehace7(6);

Actividad B Desarrollar soluciones recursivas para los siguientes problemas:

1) Resolver la potencia a^b, con b entero.

2) Obtener el término i-ésimo de la Sucesión de Fibonacci

3) Escribir los elementos de una cadena de forma invertida

RECURSIVIDAD vs ITERACION

Es importante notar que las soluciones anteriores están forzando la resolución de forma recursiva, pues pueden (y deberían) resolverse de forma iterativa. Como se ha puesto de manifiesto, hay una sobrecarga asociada con las llamadas a los subprogramas (Una simple llamada puede generar un gran número de llamadas recursivas. Por ejemplo, facto(n) genera n llamadas recursivas). La claridad del código no necesariamente compensa la mencionada sobrecarga.

El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas cuya solución iterativa no es sencilla. Dichos problemas, operan en general sobre estructuras no lineales que serán abordadas más adelante.

RECURSIVIDAD sobre ESTRUCTURAS de DATOS

Movimientos sobre un vector

Ejemplo 1 Obtener la suma de los elementos de un arreglo de n elementos.

```
int sumaV(int A[],int i, int n) {
 if (i==n)
   return A[n];
 else
   return A[i]+sumaV(A, i+1, n);
}

{   .......
    ingresa (A,&n);
    printf ("%d", sumaV(A,0,n-1));
}
```

Actividad C

1) ¿Cómo cambiaría **sumaV()** si la condición fuera *if (i>n)*?
2) Modificar **sumaV()** de modo de no incluir el parámetro formal **i**. O sea, la cabecera será:

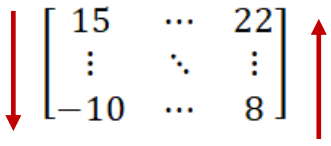
```
int sumaV(int A[], int n);
```

Actividad D Rehacer la función **sumaV()**, como función void. O sea, la cabecera será:

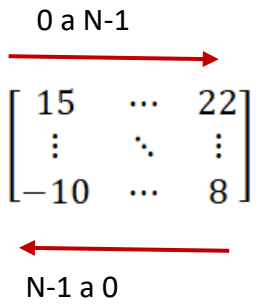
```
void sumaV(int A[], int n, int *res);
```

Movimientos sobre una matriz

Por columnas

0 a N-1  N-1 a 0

Por filas



Ejemplo 2 Obtener la suma de los elementos de una matriz de nxm elementos.

```
int sumaM(int A[][10], int i, int j, int n, int m){
if (i==n && j==m)
   return A[i][j];
else
   if (j==m)
     return A[i][j]+sumaM(A,i+1,0,n,m);
   else
     return A[i][j]+sumaM(A,i,j+1,n,m);
}
{
   .......
    ingresa (A,&n,&m);
    printf ("suma de los elementos %d", sumaM(A,0,0,n-1, m-1));
}
```

Actividad E

1) Modificar **sumaM()** de modo que la invocación sea printf ("%d", **sumaM**(A,n-1,m-1,n-1, m-1));

2) Dada una matriz de nxn de caracteres, desarrollar una función recursiva void que obtenga y muestre la cantidad de vocales por columna.

3) Detectar y corregir los errores en el código para resolver: *Dada una matriz de nxn de enteros, desarrollar una función recursiva int que cuente la cantidad de filas que tienen alguna vocal.* Además, desarrollar **esvocal().**

```
int cantvocc(int A[][10], int i, int j, int n){
if (i==n && j==n)
   return esvocal(A[i][j]);
else
   if (j>n)
     return cantvocc(A,i+1,0,n);
   else
     return vocal(A[i][j])+ cantvocc(A,i,j+1,n);
}
{
   .......
    ingresa (A,&n,&m);
    printf ("filas con al menos una vocal %d", cantvocc (A,0,0,n-1));
}
```

4) Dada una matriz de nxn de enteros, desarrollar una función recursiva int para obtener la cantidad de columnas que tienen todos sus elementos pares.

RECURSIVIDAD INDIRECTA

La recursividad también puede ser *indirecta*, si tenemos un procedimiento P que llama a otro Q y éste a su vez llama a P.

Un ejemplo de programa que utiliza recursión indirecta, y nos dice si un número es par o impar (por supuesto que no es el método más eficiente de determinar si un número es par o impar)

```
int impar(int n);
int par(int n);

int par(int n) {
  return (n == 0) ? 1 : impar(n-1);
}

int impar(int n) {
  return (n == 1) ? 1 : par(n-1);
}
```

Si la invocación fuera, por ejemplo, *if (impar(3))* se realizarían las siguientes invocaciones:

return par(2) luego return impar(1) luego return par(0) que devuelve 1

Por lo tanto 3 es un número impar.

PROBLEMAS TIPICAMENTE RECURSIVOS

Hay situaciones y/o algoritmos que son frecuentemente implementados de forma recursiva, pues la solución iterativa es más compleja. Entre otros es posible mencionar:

- Recorridos, inserción y eliminación sobre árboles (compiladores, índices de BD)
- Recorridos y algoritmos diversos sobre grafos y digrafos (GPS, Camino mínimo)
- Método de ordenación rápida (QuickSort): puede resolverse de modo recursivo o en base a Pilas
- Juego: Torres de Hanoi

CONCLUSIONES

- ☞ Recursión: es la invocación de un subprograma a sí mismo.
- ☞ Cada invocación se realiza con un valor/es de parámetro/s que hace el problema cada vez “de menor tamaño”
- ☞ La invocación recursiva se realiza siempre subordinada a una condición de una sentencia de decisión en la que, al menos debe haber un caso en el que se actúa de forma no recursiva.
- ☞ De forma oculta, se gestiona memoria en las llamadas recursivas (pila interna).

La recursividad puede aplicarse sobre cualquier estructura de datos que implique recorridos iterativos, por lo tanto será posible utilizarla sobre vectores y matrices, así como sobre otras estructuras que se abordarán más adelante.