INTRODUCCIÓN

La recursividad es aquella propiedad que posee una función por la cual puede llamarse a sí misma. Se puede utilizar la recursividad como una alternativa a la iteración. Una solución recursiva es normalmente menos eficiente en términos de tiempo de computadora que una solución iterativa debido a las operaciones auxiliares que llevan consigo las llamadas suplementarias a las funciones; sin embargo, en muchas ocasiones el uso de la misma permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver.

Los programas analizados hasta el momento, se componen de una serie de funciones que se llaman una a otras de modo disciplinado. En ocasiones para algunos problemas es muy útil disponer de funciones que se llamen a sí mismas.

Funciones recursivas

Se dice que un algoritmo es recursivo si dentro del cuerpo del algoritmo y de forma directa o indirecta se realiza una llamada a él mismo. Dicho de otra manera, una función que tiene sentencias entre las cuales se encuentra al menos una que llama a la propia función se dice que es recursiva.

La mayoría de los lenguajes soportan los algoritmos recursivos. En un algoritmo recursivo, los bucles típicos de un algoritmo iterativo (for, do, while...) se sustituyen por llamadas al propio algoritmo.

Supongamos que disponemos de dos funciones: f1 y f2, la organización de un programa no recursivo podría adoptar una forma similar a la que se muestra a continuación:

```
f1(...)
{
    ...
}
f2(...)
{
    ...
    f1();//LLama a la función 1
    ...
}
```

Con una organización recursiva, se podría tener esta otra situación:

```
f1(...)
{
    ...
    f1();//La función se llama así misma
    ...
}
```

Un algoritmo recursivo consta de:

- *Caso base*: es la condición de salida, es esencial dado que detiene una cadena de llamadas recursivas, es un problema puntual cuya solución es inmediata.
- *Parte recursiva*: puede dar como resultado muchas llamadas recursivas a la función con un conjunto menor de elementos hasta llegar a la obtención de un solo elemento que se contempla en el caso base y que es la condición para dejar de llamar a la función.

Para ampliar lo expuesto anteriormente analizamos el comportamiento del algoritmo de cálculo del factorial de un número entero positivo.

La definición matemática de factorial (n!) de un número es:

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

Veamos un ejemplo concreto, para $n = 4$:

```
4!= 4*(4-1)*(4-2)*(4-3)*(4-4)= 4*3*2*1= 24
```

Esto se puede expresar de la siguiente forma:

```
4!=4*(3!)
3!=3*(2!)
2!=2*(1!)
1!=1*(0!)
0!=1
```

Para generalizar lo expresamos de la siguiente forma:

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

De esta manera obtenemos la definición recursiva ya que contiene las dos partes mencionadas anteriormente.

- Caso base o condición de salida: $n = 0$.
- Parte recursiva: $n! = n * (n - 1)!$

Sobre la base de esta definición veamos ahora el código de la función factorial de un número entero positivo (recursiva) en C:

```
int factorial(int a)
{
    if (a==0) {// Caso base
        return 1;
    }
    else
    {
        return a*factorial(a-1);//Parte recursiva
    }
}
```

Diseño de Algoritmos Recursivos

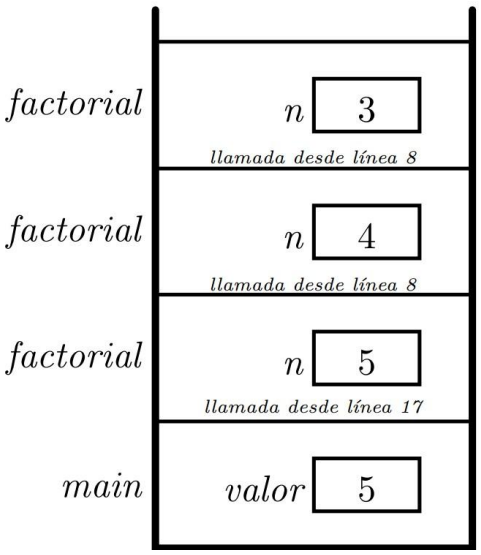
Para poder construir cualquier rutina recursiva teniendo en cuenta lo anterior, podemos usar el siguiente método:

- Primero, obtener una definición exacta del problema.
- A continuación, determinar el tamaño del problema completo a resolver. Así se determinarán los valores de los parámetros en la llamada inicial al procedimiento o función.
- Resolver el caso base en el que problema puede expresarse no recursivamente.
- Por último, resolver correctamente el caso general, en términos de un caso más pequeño del mismo problema (una llamada recursiva).

El seguimiento de la recursividad

Cuando un programa llama a una función que llama a otra, la cual llama a otra y así sucesivamente, las variables y valores de los parámetros de cada llamada a la función se guardan en la pila o stack, junto con la dirección de la siguiente línea de código a ejecutar una vez finalizada la ejecución de la función invocada.
Esta pila va creciendo a medida que se llama a más funciones y decrece cuando cada función termina. Si una función se llama a sí misma recursivamente un número muy grande de veces existe el riesgo de que se agote la memoria de la pila, causando la terminación brusca del programa.

Por ejemplo, cuando llamamos a factorial(5), que ha llamado a factorial(4), que a su vez ha llamado a factorial(3), la pila presentará la siguiente configuración:



Como vemos, no se conoce la cantidad de memoria que va a utilizarse al ejecutar un procedimiento o función recursiva sino que se produce una asignación dinámica de memoria, es decir, a medida que se producen instancias nuevas, se van "apilando" las que quedan pendientes. Cuando la última instancia de la recursión, elige el caso base, es decir, se cumple, se "desapila" esa instancia y el control vuelve a la instancia anterior; así hasta "desapilar" todas las instancias. Esta "pila" que se va generando en memoria es importante tenerla en cuenta por lo que debemos asegurarnos de que el algoritmo recursivo no sea divergente.

Analicemos en profundidad la secuencia de llamadas del factorial para n=4:

$(1) factorial(4) = 4 * factorial(3)$ $return\ (4 * 6)$


$(2) factorial(3) = 3 * factorial(2)$ $return\ (3 * 2)$


$(3) factorial(2) = 2 * factorial(1)$ $return\ (2 * 1)$


$(4)\ factorial(1) = 1$  $return\ 1;$

Cada fila del anterior gráfico supone una instancia distinta de ejecución de la función factorial. Cada instancia tiene un conjunto diferente de variables locales.

Tipos de recursividad

Se pueden establecer diferentes tipos de recursividad en virtud de la característica del algoritmo analizada:

- Según el punto desde el cual se hace la llamada recursiva: *recursividad directa o indirecta*.
- Según el número de llamadas recursivas efectuadas en tiempo de ejecución: *recursividad lineal o no lineal*.
- Según el punto del algoritmo desde donde se efectúa la llamada recursiva: *recursividad final o no final*.

Recursividad directa y recursividad indirecta

- Recursividad directa: Se da cuando la función efectúa una llamada a sí misma.

```
funcion A(...)
{
    ...
    A();//La función se llama así misma
    ...
}
```

- Recursividad indirecta: Se da cuando una función A llama a otra función B la cual a su vez, y de forma directa o indirecta, llama nuevamente a A.

```
funcion A(...)
{
    ...
    B();
    ...
}
```

```
funcion B(...)
{
    ...
    A();
    ...
}
```

Recursividad lineal y recursividad no lineal

- Recursividad lineal o simple: Se da cuando la recursividad es directa y además cada llamada a la función recursiva sólo hace una nueva llamada recursiva.

```
funcion A(...)
{
    ...
    A();
    ...
}
```

- Recursividad no lineal o múltiple: La ejecución de una llamada recursiva da lugar a más de una llamada a la función recursiva.

```
funcion A(...)
{
    if (condicion)
       {
               A();
       }
    A();
}
```

En esta última gráfica si la condición se cumple la recursividad será no lineal, ya que se producen dos llamadas recursivas.

Recursividad final y recursividad no final

- Recursividad final: Se da cuando la llamada recursiva es la última operación efectuada en el cuerpo de la función.

```
funcion A(...)
{
    ...
    return A();
}
```

Note en esta gráfica que no existe ningún proceso posterior a la llamada a la función A().

- Recursividad no final: Se da cuando la llamada recursiva no es la última operación realizada dentro de la función.

```
funcion A(...)
{
  ...
  return A(...)+5;
}
```

En esta gráfica la llamada a la función recursiva no es la última operación efectuada sino que es una suma.

Veamos otro ejemplo de no final:

```
funcion A(...)
{
  int n= 10;
  A(...);
  return n;
}
```

¿Por qué escribir programas recursivos?

Tenemos argumentos por los cuales elegir la recursividad:

• Son más cercanos a la descripción matemática.
• Generalmente más fáciles de analizar.
• Se adaptan mejor a las estructuras de datos recursivas.
• Los algoritmos recursivos ofrecen soluciones estructuradas, modulares, elegantes y simples.

Pero entonces, resta saber cuándo es que conviene la recursividad.

¿Cuándo usar recursividad?

• Para simplificar el código.
• Cuando la estructura de datos es recursiva ejemplo: listas, árboles.

¿Cuándo NO usar recursividad?

• Cuando los métodos usen vectores grandes.
• Cuando las iteraciones sean la mejor opción.

Recursividad vs Iteración

La mayoría de los algoritmos pueden expresarse tanto de forma iterativa como de forma recursiva.
Existen distintos consideraciones; a continuación enumeramos los más importantes:

- Conveniencia

  ➔ por el lenguaje de programación utilizado

Algunos lenguajes se benefician del uso de funciones recursivas respecto a las funciones iterativas. Por ejemplo, los llamados lenguajes funcionales, como Lisp, Scheme o Haskell, están orientados y optimizados para el trabajo con funciones recursivas.

➔ por la definición del problema

Hay problemas cuya definición y/o solución son inherentemente recursivas y una solución iterativa resultaría demasiado complicada. o viceversa.

- Eficiencia

En general la versión iterativa de una función siempre es más eficiente que la versión recursiva, tanto en términos de velocidad de ejecución como de memoria utilizada:

➔ Velocidad de ejecución:

En la versión recursiva, el hecho de que la función se llame a sí misma implica o supone un costo de tiempo extra de proceso en comparación con el que supone reiniciar una nueva iteración de un bucle. Si el compilador soporta las funciones recursivas finales, la eficiencia se pone a la par de la versión iterativa.

➔ Uso de memoria:

En la versión recursiva, la llamada a la función hace uso de la pila para almacenar los parámetros pasados (si los hay), dirección de retorno, resultado, etc. Si el nivel de recursión es muy elevado, el uso de memoria puede ser considerable o incluso prohibitivo.

- Sencillez

En ocasiones es mucho más sencillo escribir la versión recursiva de un algoritmo que la versión iterativa, y viceversa. En general, la versión recursiva de una función es mucho más elegante y sencilla a nivel de cantidad de código requerido que la versión iterativa.