TEMA 4. ANEXO. PROGRAMACIÓN RECURSIVA

Es una técnica de programación que permite a un subprograma llamarse a sí mismo para resolver una versión reducida del programa original. Esta obliga a analizar el problema y a detectar el final de este.

Siempre que queramos encarar un problema de manera recursiva podemos tratar de pensar en ese problema en función de sí mismo, pero con una versión más reducida.

Uno de los problemas que se utilizan para explicar la recursividad es el factorial.

Nos ponemos un ejemplo de factorial. Sabemos que el factorial de 5 se calcula de la siguiente forma:

5! = 5 * 4 * 3 * 2 * 1 dando como resultado 120.

Pero podemos darnos cuenta que:

5! = 5 * 4!

Es decir, que **n! = n * (n-1)!**

Pero cuidado si pensamos así siempre ya que

4! = 4 * 3! y a su vez **3! = 3 * 2!** y

2! = 2 * 1! y **1! = 1 * 0!** y **0! = 0 * -1!....**

y así hasta el infinito.

En una función recursiva debemos tener un **caso base**, que hará que termine la función, y un **caso recursivo**, que será una versión reducida del problema original.

El caso recursivo debe converger hacia el caso base. Si no es así tendremos una recursividad infinita.

En el caso del factorial podemos pensar por ejemplo en el 0 como el caso base, ya que 0! = 1.

Así, podríamos definir recursivamente el factorial de la siguiente forma:

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

n! = 1 si n = 0.

De esta manera podemos traducir directamente a una función recursiva, que finalmente es simplemente una función que se llama a sí misma, teniendo una llamada recursiva y una parte no recursiva (caso base).

Os pongo a continuación la definición no recursiva y la recursiva de la función que calcula el factorial de un número:

Versión no recursiva:

```
int factorialIterativo(int num)
{
    int res, i;

    res = 1;
    for (i = 1; i <= num; i++)
        res = res * i;

    return res;
}
```

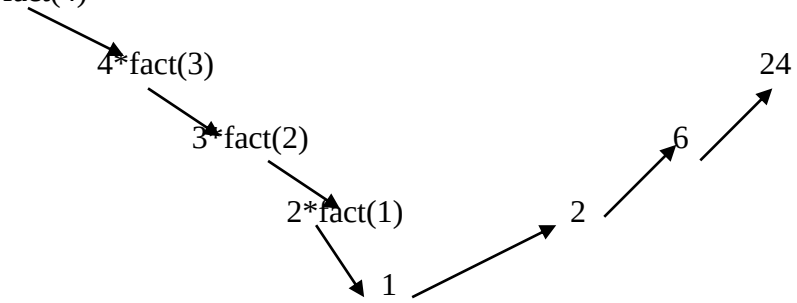
Versión Recursiva:

```
int factorialRecursivo(int num)
{
    int res;

    if (num == 0)
        res = 1;
    else
        res = num * factorialRecursivo(num - 1);

    return res;
}
```

num=4;
fact(4)



La recursividad es una técnica muy potente que permite resolver problemas muy complejos de una forma muy intuitiva. Esto no quiere decir que la recursividad siempre sea la técnica más efectiva para resolver un problema, aunque para ciertos problemas nos presenta una solución muy sencilla.

Por ejemplo vamos a ver cómo resolver la sucesión de Fibonacci, haciendo una función que nos devuelva el número situado en la posición correspondiente:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

F (n) = F (n-1) + F (n-2) si n >= 2

F (n) = 1 si n = 0 ó n = 1

Versión Recursiva:

```
int fibonacciRecursivo(int pos)
{
    int res;

    if (pos == 0 || pos == 1)
        res = 1;
    else
        res = fibonacciRecursivo(pos - 1) + fibonacciRecursivo(pos - 2);

    return res;
}
```

Versión Iterativa:

```
int fibonacciIterativo(int pos)
{
    int n1, n2, res, i;

    n1 = 1;
    n2 = 1;
    res = 1;

    for (i = 1; i < pos; i++)
    {
        res = n1 + n2;
        n1 = n2;
        n2 = res;
    }

    return res;
}
```

Podemos ver un ejemplo de la potencia de la programación recursiva en el problema del juego de las **Torres de Hanoi**…

A finales del siglo XIX un juego llamado las torres de Hanoi, apareció en las tiendas de Europa.

El juego consistía en tres palos, sobre el primero reposaban ocho discos de madera de tamaño diferente.

El objetivo del juego es mover la torre de discos del palo izquierdo al derecho con la condición de que solamente un disco puede ser movido a la vez y en ningún momento un disco puede reposar sobre otro más pequeño.

https://es.wikipedia.org/wiki/Torres_de_Han%C3%B3i

http://www.uterra.com/juegos/torre_hanoi.php

Si habéis intentado jugar os daréis cuenta de que conforme aumenta el número de discos, aumenta la dificultad y el número de movimientos **de manera exponencia**l.

El número de movimientos viene dado por la fórmula $2^{\text{núm discos}} - 1$

Es decir, con 3 discos tenemos que hacer 7 movimientos.

Con 4 discos 15 movimientos, con 5 debemos hacer 31 movimientos…

Con 8 discos 255 movimientos.

La leyenda dice que el juego original consistía en 64 discos y que al completarlo acabaría el Universo.

64 discos supondrían 18.446.744.073.709.551.616 movimientos. Tengamos en cuenta que si moviéramos un disco al segundo y sin equivocarnos nunca tardaríamos unos 500.000 millones de años en completar el juego con 64 discos.

Sin embargo, es un problema que **se resuelve muy fácilmente** utilizando programación recursiva.

La idea es la siguiente.

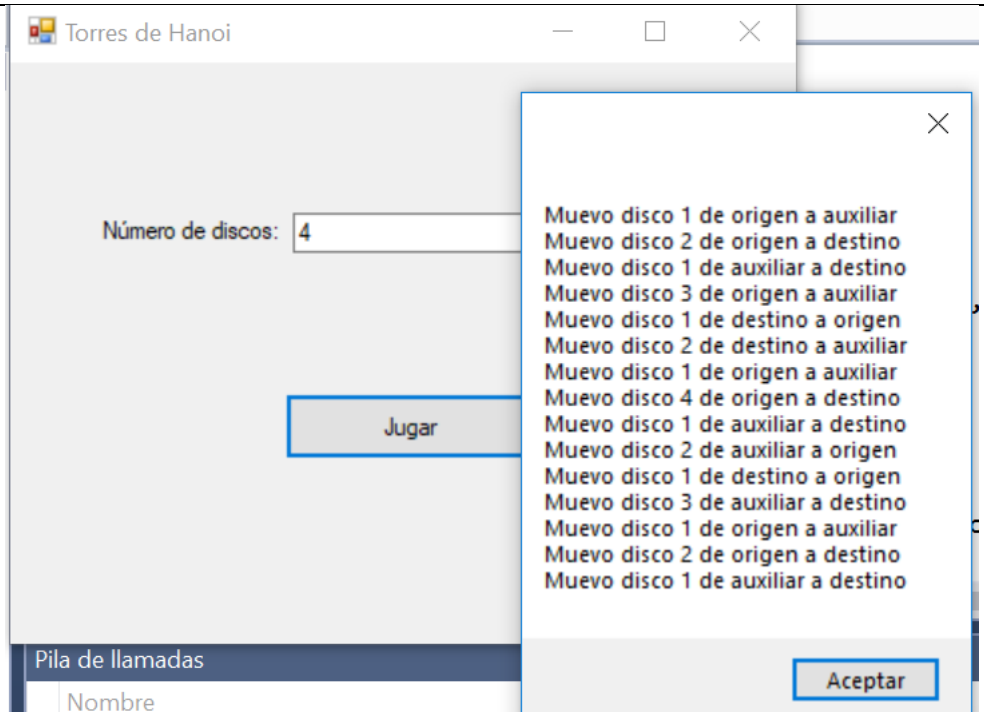
Jugar a hanoi con n discos del origen al destino utilizando el auxiliar se puede hacer en tres pasos (dos de ellos recursivos):

- (Paso recursivo) Jugar a hanoi con **n-1 discos** del origen al auxiliar (utilizando el destino).
- Mover el disco n del origen al destino (ya que está libre)
- (Paso recursivo) Jugar a hanoi con **n-1 discos** del auxiliar al destino (utilizando el origen).

Os pongo a continuación el programa que resuelve este problema, tanto la ejecución así como el código de la función, y la llamada en el botón.

No es un programa gráfico. Simplemente indica un texto con los pasos que tendríamos que hacer moviendo cada disco de un palo a otro para resolver el problema.

Podéis utilizar el enlace que os he puesto antes para comprobar que los pasos son correctos.




```
string hanoi(int ndiscos, string origen, string destino, string aux)
{
    string res = "";

    if (ndiscos> 0)
    {
        res = res + hanoi(ndiscos - 1, origen, aux, destino);

        res = res + "Muevo disco " + ndiscos + " de " + origen + " a " +
                    destino + "\n";

        res = res + hanoi(ndiscos - 1, aux, destino, origen);
    }

    return res;
}

private void button1_Click(object sender, EventArgs e)
{
    int ndiscos;
    string res;

    ndiscos = int.Parse(tNumDiscos.Text);

    res = hanoi(ndiscos, "origen", "destino", "auxiliar");

    MessageBox.Show(res);
}
```