UNIVERSIDAD DE EL SALVADOR
FACULTAD DE INGENIERIA Y ARQUITECTURA
ESCUELA DE INGENIERIA DE SISTEMAS INFORMATICOS
PROGRAMACION II

## ***Introducción a la recursividad***.

El concepto de recursividad no es nuevo. En el campo de la matemática podemos encontrar muchos ejemplos relacionados con él. En primer lugar muchas definiciones matemáticas se realizan en términos de sí mismas. Considérese el clásico ejemplo de la función factorial:

$$n! = 1 \text{ si } n = 0$$
$$n\ (n - 1)! \text{ si } n > 0 \quad (1)$$

Donde la definición se realiza por un lado para un caso que denominamos base (n = 0) y por otro para un caso general (n > 0) cuya formulación es claramente recursiva. Consideremos por otro lado la conocida demostración por inducción que generalmente solo necesita realizar una simple demostración para un caso base y una segunda para un caso general de tamaño n (con la ventaja de haberlo demostrado para cualquier caso menor que n) de forma que queda demostrado para todos los casos. De una manera similar, cuando programamos una función recursiva podemos decir que esta resuelta para un tamaño menor (véase mas adelante). Se podría decir que la gran virtud de la programación recursiva radica en que para resolver un problema, el usar esta técnica implica que ya “tenemos resuelto" el problema (de menor tamaño como ya veremos). De esta forma, la recursividad constituye una de las herramientas más potentes en programación.

Básicamente, *un problema podría resolverse de forma recursiva si es posible expresar una solución al problema (algoritmo) en términos de él mismo, es decir, para obtener la solución será necesario resolver este mismo problema sobre un conjunto de datos o entrada de menor tamaño*. En principio podemos decir que los problemas recursivos que generalmente son más fáciles de resolver son aquellos que de forma natural se formulan recursivamente. De esta forma, estos problemas ofrecen directamente una idea de como debe ser el algoritmo recursivo. Considérese por ejemplo el caso del factorial, que por definición nos dice que hay un caso base (n = 0) y un caso recursivo ((n - 1)!) Que junto con una multiplicación por n nos da el resultado.

Por otro lado, hay problemas que no se expresan de forma recursiva y por tanto es necesario formular un algoritmo que, resolviendo el problema, se exprese en términos de sí mismo. Por tanto, este tipo de problemas requiere un esfuerzo adicional que esta especialmente indicado para los que la solución no recursiva es especialmente compleja o problemas sobre los que aplicando algún tipo de técnica de diseño de algoritmos nos llevan a soluciones recursivas que son simples de construir y no requieren un nivel alto de recursos. Un ejemplo es el conocido algoritmo de búsqueda binaria o dicotomica (véase la guía de laboratorio de recursividad) . El problema consiste en localizar un elemento dentro de un vector de elementos ordenados. Es claro que el algoritmo se podría resolver sin ningún problema de forma iterativa pero si aplicamos la técnica de divide y vencerás podemos llegar al citado algoritmo.

La división del vector en dos partes y la selección del subvector (donde debe estar el elemento a buscar según la ordenación) para continuar buscando (el mismo problema sobre el subvector).

***Diseño de algoritmos recursivos.***

Para resolver un problema, el primer paso será la identificación de un algoritmo recursivo, es decir, descomponer el problema de forma que su solución quede definida en función de ella misma pero para un tamaño menor y la tarea a realizar para un caso simple. Es necesario diseñar casos base, casos generales y la solución en términos de ellos.

***Los casos base.*** Son los casos del problema que se resuelve con un segmento de código sin recursividad. Normalmente corresponden a instancias del problema simples y fáciles de implementar para los cuales es innecesario expresar la solución en términos de un subproblema de la misma naturaleza, es decir, de forma recursiva. Obviamente el numero y forma de los casos base son hasta cierto punto arbitrarios, pues depende del programador y su habilidad el hecho de identificar el mayor conjunto de casos en términos de simplicidad y eficiencia. En general no es difícil proponer y resolver un conjunto de casos base ya que por un lado corresponden a instancias simples y por otro en caso de problemas muy complejos siempre es posible proponer varios casos para asegurar que el conjunto es suficientemente amplio como para asegurar el "n" de la recursion. Por supuesto, la solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

Una regla básica a tener en cuenta con respecto al diseño de una función recursiva es básicamente que siempre debe existir al menos un caso base. Es obvio que las llamadas recursivas para la solución de un problema no pueden darse de forma indefinida sino que debe llegar a un caso en el que no es necesario expresar la solución como un problema de menor tamaño, es decir, tienen que existir casos base. Por ejemplo, si consideramos una definición de factorial del tipo n! = n(n 1)!, se necesita añadir el caso 0! = 1 para indicar una condición de parada en la aplicación de la ecuación recursiva. Por supuesto, esta regla es bastante obvia pues básicamente estamos hablando que la recursividad debe parar en algún momento.

***Los casos generales.*** Cuando el tamaño del problema es suficientemente grande o complejo la solución se expresa de forma recursiva, es decir, es necesario resolver el mismo problema, aunque para una entrada de menor tamaño. Es justo aquí donde se explota la potencia de la recursividad ya que un problema que inicialmente puede ser complejo se expresa como: (a) Solución de uno o mas subproblemas (de igual naturaleza pero menor tamaño). (b) Un conjunto de pasos adicionales. Estos pasos junto con las soluciones a los subproblemas componen la solución al problema general que queremos resolver.

Si consideramos los subproblemas como algo resuelto (de hecho generalmente solo requieren de llamadas recursivas) el problema inicial queda simplificado a resolver solo el conjunto de pasos adicionales. Por ejemplo, para el caso del factorial es necesario calcular el factorial de n -1 (llamada recursiva) y adicionalmente realizar una multiplicación (mucho mas simple que el problema inicial). Obviamente esto nos llevara a la solución si sabemos resolver el subproblema, lo cual esta garantizado pues en ultima instancia se reduce hasta uno de los casos base. Por tanto una regla básica a tener en cuenta con respecto al diseño de los casos generales de una función recursiva es que los casos generales siempre deben avanzar hacia un caso base. Por ejemplo, cuando diseñamos la función del calculo de factorial, el caso general hace referencia al subproblema (n - 1)! que como es obvio esta mas cerca de alcanzar el caso base n = 0.

### *Implementación de funciones recursivas.*

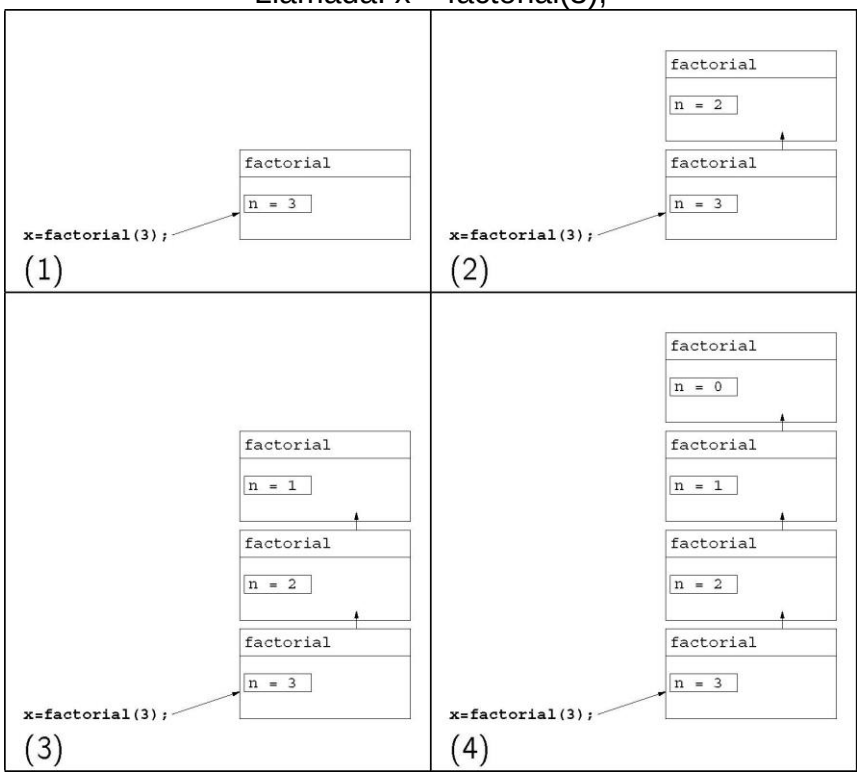
Es este apartado vamos a ver algunos consejos y reglas practicas para implementar funciones recursivas. En primer lugar, se plantean las posibles consideraciones a la hora de especificar la cabecera de la función recursiva (sintaxis y semántica). En este sentido, una regla básica a tener en cuenta es que la cabecera debe ser valida tanto para llamar al problema original como a uno de sus subproblemas.

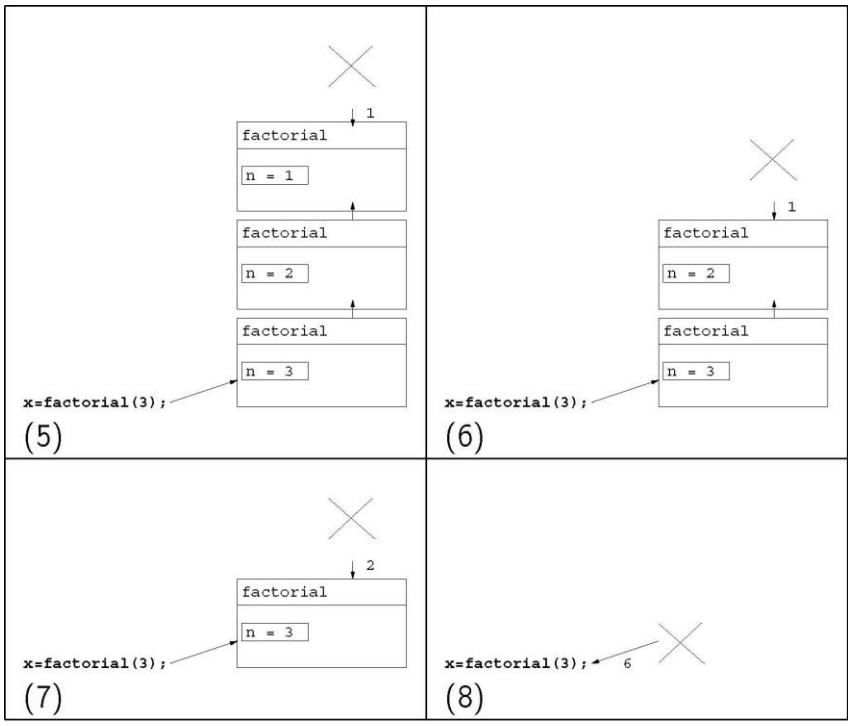
### *Ejecución de un módulo recursivo*

En general, en la pila se almacena el entorno asociado a las distintas funciones que se van activando. En particular, en un modulo recursivo, cada llamada recursiva genera una nueva zona de memoria en la pila independiente del resto de llamadas.
Ejemplo: Ejecución del factorial con n=3.
Dentro de factorial, cada llamada return n*factorial(n-1); genera una nueva zona de memoria en la pila, siendo n-1 el correspondiente parámetro actual para esta zona de memoria y que dependiente la evaluación de la expresión y la ejecución del return.

El proceso anterior se repite hasta que la condición del caso base se hace cierta. Se ejecuta la sentencia return1; Empieza la vuelta atrás de la recursion, se evalúan las expresiones y se ejecutan los return que estaban pendientes.

Llamada: x = factorial(3);

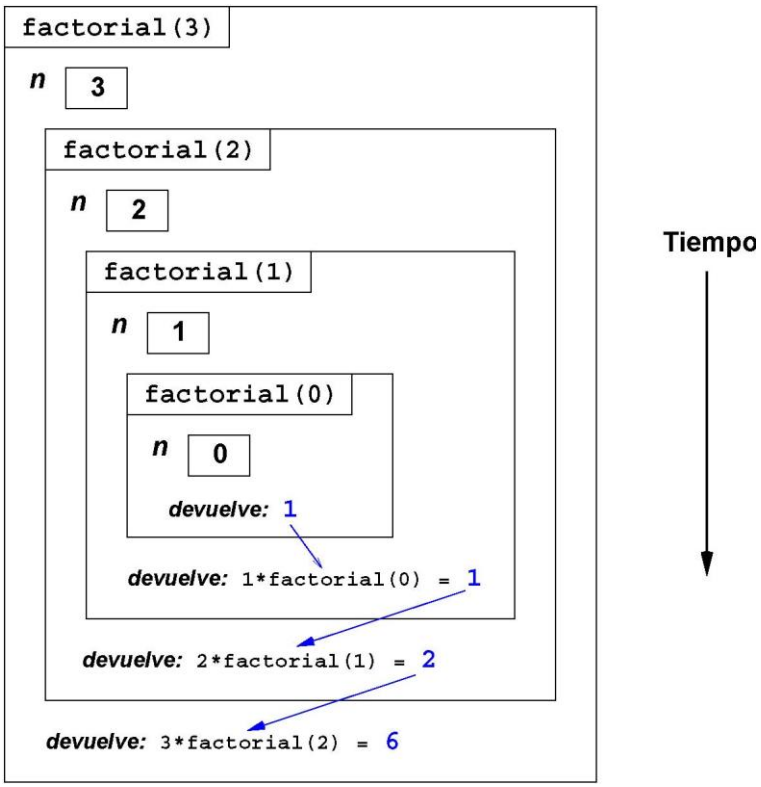





Traza de algoritmos recursivos

Se representan en cascada cada una de las llamadas al modulo recursivo, así como sus respectivas zonas de memoria y los valores que devuelven.

Llamada: factorial(3)



***Recursivo vs iterativo.***

En muchos casos, la recursividad requiere mas recursos (tiempo y memoria) para resolver el problema que una versión iterativa, por lo que, independientemente de la naturaleza del problema, la recursividad tiene su gran aplicación para problemas complejos cuya solución recursiva es más fácil de obtener, mas estructurada y sencilla de mantener. Así por ejemplo, los problemas de calculo de factorial o de búsqueda binaria son fáciles de programar iterativamente por lo generalmente no se utilizan las versiones recursivas:

1. En el caso del factorial, no necesitamos mas que un bucle para calcular el producto mientras que en la versión recursiva el numero de llamadas que se anidaran es igual que el valor de entrada de manera que usar la versión recursiva es mas un error que algo poco recomendable.

2. Para la búsqueda binaria el máximo numero de llamadas anidadas (profundidad de recursion) es del orden del logaritmo del tamaño del vector por lo que este factor es menos importante. A pesar de ello, y considerando que la versión iterativa es muy simple, también para este caso podemos rechazar el uso de la recursividad.

Incluso para casos más complejos pero correspondientes a funciones que se llaman muchas veces en un programa puede ser interesante eliminar la recursividad, ya sea de una manera sencilla como es el caso de la recursion de cola o de forma mas compleja mediante la introducción de estructuras adicionales como pilas definidas por el programador.

Además, cuando se resuelven problemas de manera recursiva hay que tener especial cuidado ya que el que la función tenga varias llamadas recursivas puede hacer que el numero de subproblemas a resolver crezca de forma exponencial haciendo incluso que el resultado obtenido sea mucho mas ineficiente. Por tanto, podemos plantear una ultima regla en la programación recursiva: *No resolver varias veces el mismo subproblema en llamadas recursivas distintas***.** Para ilustrar esta idea véase el ejemplo de la sucesión de Fibonacci. A pesar de estas consideraciones, existen multitud de problemas que se ven radicalmente simplificados mediante el uso de la recursividad y que cuya diferencia en recursos necesarios con respecto a la solución iterativa puede ser obviada a efectos prácticos.

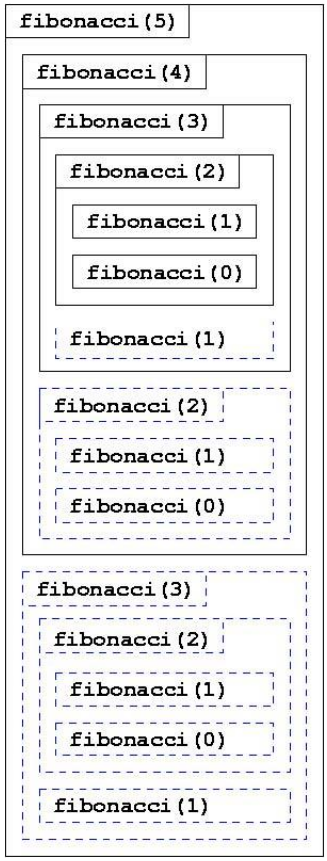
La sucesión de Fibonacci viene dada por:

**F(n) = 1 si n = 0 o n = 1**
**F(n 1) + F(n 2) si n > 1 (2)**

```
int fibonacci (int n)
{
        if (n == 0 || n == 1)
                return 1;
        else
                return fibonacci(n-1) + fibonacci(n-2);
}
```

```
int fibonacci_nr (int n)
{
        int ant1 = 1, ant2 = 1; // anterior y anteanterior int actual;
        // valor actual
         if (n == 0 || n == 1)
                actual = 1;
                else
                for (i=2; i<=n; i++) {
                        actual = ant1 + ant2; // suma los anteriores ant2 = ant1;
                        // actualiza "ant2" ant1 = actual; // y "ant1"
                }
return actual;
}
```

Ejemplo:Calculo de fibonacci(5)




El interés en esta función no es tanto la dificultad de implementarla sino la discusión sobre la conveniencia de una implementación recursiva. Es importante tener cuidado de no resolver varias veces el mismo subproblema. En este caso podemos observar que para cada n hay que calcular F(n-1) y en segundo lugar F(n-2) que ya esta incluido en el primero ya que si n - 1 > 1; F(n - 1) = F(n - 2) + F(n - 3). Por tanto el resultado será mucho más ineficiente al estar duplicando trabajo.