

# 75.41 Algoritmos y Programación II

## Recursividad

Dr. Mariano Méndez <sup>1</sup>

<sup>1</sup>Facultad De Ingeniería. Universidad de Buenos Aires

29 de abril de 2020

### 1. Introducción

Una gran mayoría de algoritmos útiles son recursivos. Esto quiere decir que para resolver un problema determinado, ellos se llaman a sí mismos. Existen varios tipos de recursividad:

- Recursividad Directa
- Recursividad Indirecta
- Recursividad de Cola

La **recursividad** es una característica de ciertos problemas los cuales son resolubles mediante la solución de una nueva instancia del mismo problema. En informática la recursividad está fuertemente asociada a las funciones y a los tipos de datos. En este apunte se estudia su relación con la Implementación de las funciones recursivas. Se dice que *una función es recursiva si en el cuerpo de la función hay una llamada a sí misma*. Para poder ver cual sería este caso, es conveniente mirar algunos problemas matemáticos cuya naturaleza es recursiva, por ejemplo, los números factoriales. Un número factorial se define de la siguiente forma:

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

En este caso, puede verse como en la definición de la función factorial se invoca a sí mismo, este es uno de los mejores ejemplos del concepto de recursivo. Por el momento la versión conocida, para determinar el factorial de un número, debe estar construida en forma iterativa:

```
1 long factorial (int un_numero){
2     long producto;
3     int contador;
4
5     producto=1;
6     contador=un_numero;
7     while ( contador > 0 ){
8         producto *= contador;
9         contador--;
10    }
11    return producto;
12 }
```

### 2. Recursividad Directa

¿Será posible escribir una versión recursiva del mismo algoritmo? En el lenguaje de programación C es posible construir funciones recursivas de la misma forma que en matemática. Para ello lo único que hay que hacer es llamar otra vez a la función dentro de su ámbito. Para lograr realizar esto se utiliza el **stack o pila de ejecución**, este no es más que la parte del programa donde se mantiene la información sobre cual función es la que está siendo ejecutada, sus variables y parámetros. Todos los programas tienen su propio stack, que es creada cuando comienza a ejecutarse, ver Figura 1.

Si se observa detalladamente una función recursiva debe cumplir con ciertas reglas para que funcione correctamente, estas son:

- debe poseer una condición de corte
- debe poseer una llamada recursiva, es decir así misma dentro de la función.

llamada recursiva

```

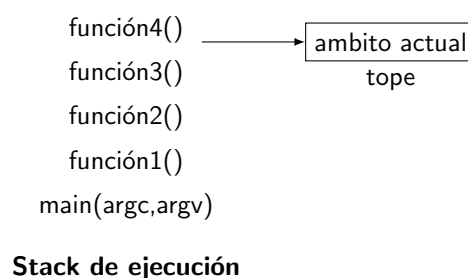
1 long factorial (int un_numero){
2   if (un_numero > 0 )
3   return un_numero * factorial(un_numero-1);
4   else
5   return 1; ←-- condición de corte
6 }
7

```

condición de corte

### 2.0.1. El Ámbito de una Función

Cuando un programa pasa de estar almacenado en disco a ser ejecutado, el mismo es cargado en memoria y estructurado en cuatro partes (.code, .data, .heap y .stack). Una de estas partes, **el stack**, es la encargada de mantener el estado de cómo se van llamando las funciones y cual es la función que se está ejecutando en un determinado instante  $t_k$ . El stack cumple sus funciones de forma automática y es independiente del programador. Esta parte del programa mantiene los datos de las funciones en ejecución justamente como lo hace el **tda** que lleva su nombre Pila o Stack. Por ende, el stack de ejecución de un programa es una estructura de datos de la cual únicamente se puede obtener información de su tope.



El ámbito de una función no es nada más que una porción de memoria. que se crea cuando la función es llamada y se destruye cuando la función termina su ejecución. Dentro del stack se almacenan:

- los argumentos que son pasados a la función.
- todas las variables locales declaradas en la función.

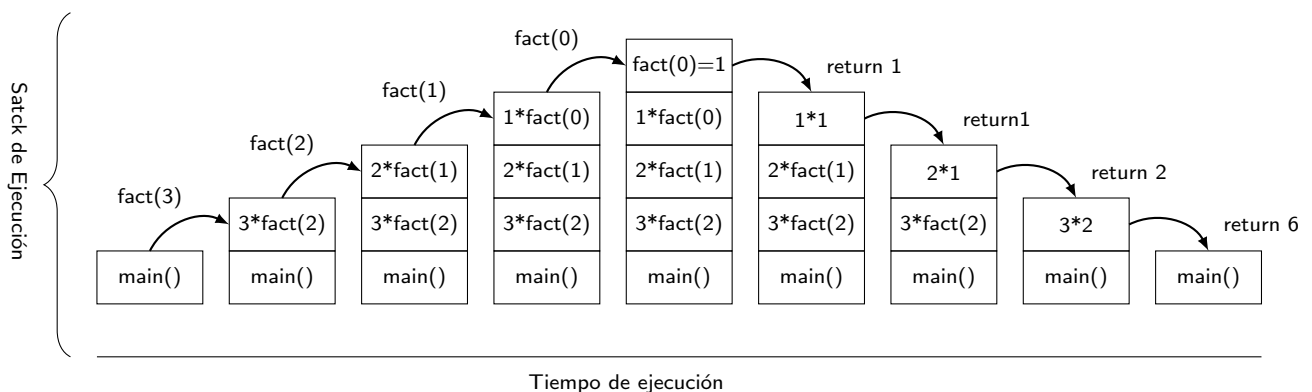


Figura 1: stack de ejecución

**Ejemplo:** Se le solicita al usuario que ingrese un número por teclado, luego se calcula e imprime por pantalla el resultado de la Serie de Fibonacci para dicho número.

```

1 long int fibonacci(numero){
2     if (numero < 2) // Condicion de corte
3         return numero;
4
5     return fibonacci(numero-1)+fibonacci(numero-2); //Llamado recursivo para numero-1 y
    luego para numero-2
6 }
7
8 int main (){
9
10     int numero;
11     scanf ("%i",&numero);
12
13     long int resultado = fibonacci(numero);
14
15     printf("El Fibonacci de %i es %lu",numero,resultado);
16     return 0;
17 }

```

Este tipo de recursividad se denomina recursividad directa pues la función se llama a sí misma tantas veces como sea necesario o lo permita la condición de corte.

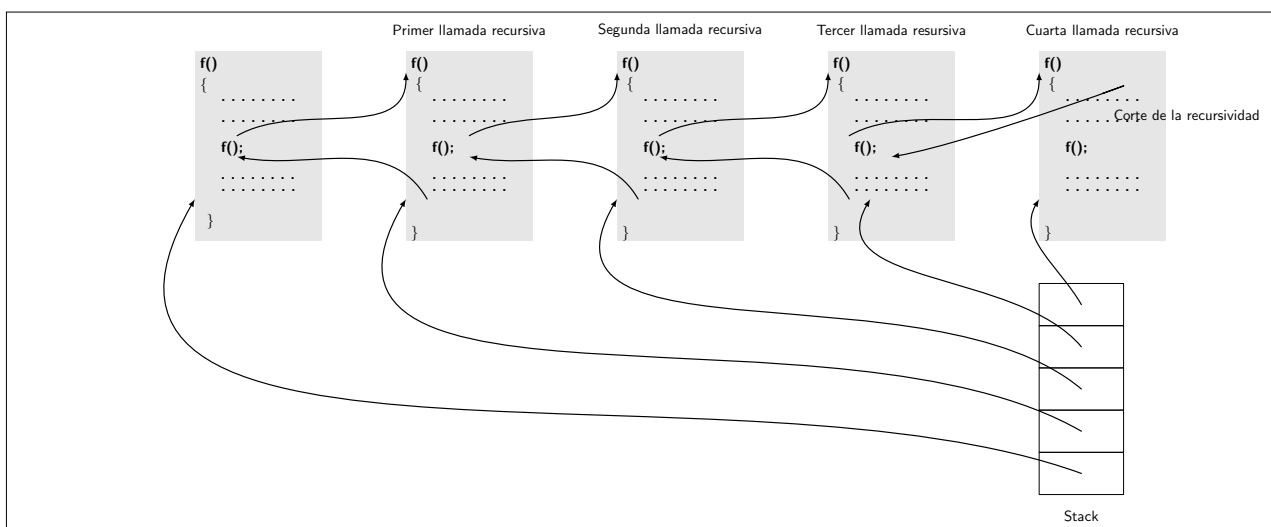


Figura 2: Simulación de recursividad mutua o indirecta

### 3. Recursividad Indirecta

Se dice que una función usa recursividad directa, si la misma se llama a sí misma. Por otro lado existe otra forma de recursividad, la **recursividad indirecta**, que se da cuando una función `f` llama a una función `g`, la función `g` llama a la función `f` y así sucesivamente... En la Figura 5 se puede ver el esquema de funcionamiento de la recursividad indirecta:

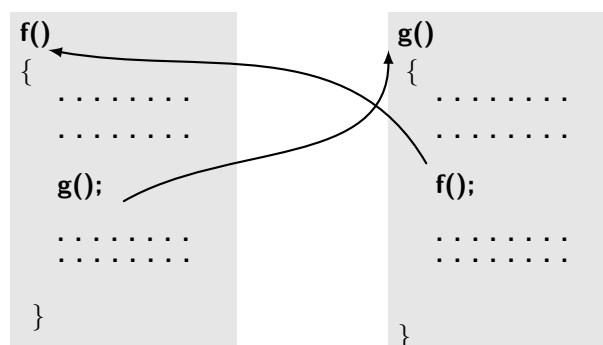


Figura 3: Simulación de recursividad mutua o indirecta

Al igual que en la recursividad directa ambas funciones deben tener un punto de corte.

## Ejercicio 1

La Granja de los Conejos Inmortales Tomás el granjero le compró al brujo del pueblo un par de conejos inmortales recién nacidos con la intención que se reproduzcan y tener su propio ejercito de conejos inmortales super adorables y poder así conquistar el mundo. Tomás le preguntó al brujo que es lo que tenía que tener en cuenta a la hora de tener su propia granja de conejos inmortales y el brujo le comentó lo siguiente:

1. Un par de conejos bebe tardan un mes en crecer.
  2. Un par de conejos adulto tardan un mes en producir otro par de conejos bebe.
  3. A partir de que un par de conejos adulto tienen un par de bebés, producen uno nuevo cada mes indefinidamente.
1. Tomás inmediatamente se da cuenta que va a ser difícil llevar la cuenta de cuántos conejos tendrá en el futuro así que nos pidió que lo ayudemos y hagamos una serie de algoritmos en C que determinen cuántos conejos bebe, adultos y totales tendrá el n-ésimo mes.

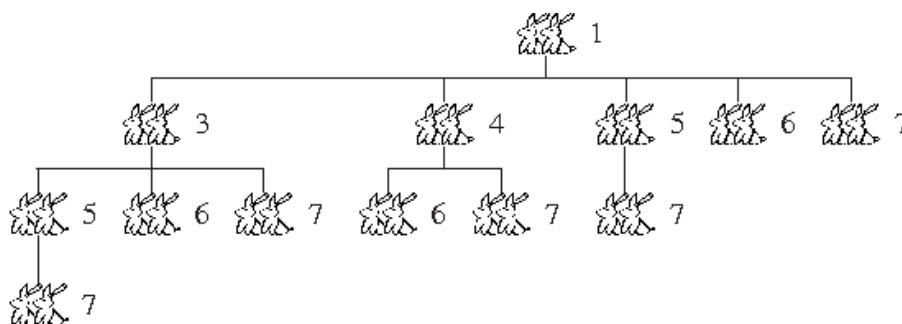


Figura 4

### Solución del ejercicio 1

**Análisis:** en primer lugar se debería poder escribir un ejemplo a mano del problema planteado.

Mes	Bebes	Adultos
1	1	0
2	0	1
3	1	1
4	1	2
5	2	3
6	3	5
7	5	8

Figura 5: Simulación de recursividad mutua o indirecta

Para saber la cantidad de conejos adultos basta con mirar la tabla, y se puede observar que:

$$Adultos(n) = \begin{cases} 0 & \text{si } n = 0 \\ Bebes(n-1) + Adultos(n-1) & \text{si } n > 0 \end{cases}$$

para saber la cantidad conejos bebes en un determinado tiempo n, basta ver en la tabla que :

$$Bebes(n) = \begin{cases} 1 & \text{si } n = 0 \\ Adultos(n-1) & \text{si } n > 0 \end{cases}$$

Con la definición de estas funciones ya se puede implementar el problema en forma computacional. **Implementación:**

```
1 #include <stdio.h>
2
3 //Linea de compilacion: gcc resuelto.c -o resuelto -Wall -Werror -std=c99 -g
4
5 /*
6  * Todas estas funciones devuelven la cantidad de parejas
7  * de conejos totales, bebes y adultas respectivamente.
```

```

8  * Pre: El numero del mes es mayor o igual a 0
9  */
10 unsigned int cantidad_total(unsigned int mes);
11 unsigned int cantidad_bebes(unsigned int mes);
12 unsigned int cantidad_adultos(unsigned int mes);
13
14 int main(){
15     unsigned int mes;
16     printf("El codigo funciona con cualquier número, pero es recomendable no
17 poner numeros mayores a 40 porque puede llegar a tardar mucho\n", );
18     printf("Mes número:");
19     scanf("%d", &mes);
20     printf("La cantidad total de parejas es:%d\n", cantidad_total(mes));
21     printf("La cantidad de parejas bebes es:%d\n", cantidad_bebes(mes));
22     printf("La cantidad de parejas adultas es:%d\n", cantidad_adultos(mes));
23     return 0;
24 }
25
26 unsigned int cantidad_total(unsigned int mes){
27     return cantidad_bebes(mes)+cantidad_adultos(mes);
28 }
29
30 unsigned int cantidad_bebes(unsigned int mes){
31     /*
32      * La cantidad de bebes en un mes es igual a la cantidad
33      * de adultos del mes anterior. Si el mes es el primero,
34      * la cantidad de bebes es 1.
35      */
36
37     if(mes == 1)
38         return 1;
39
40     return cantidad_adultos(mes-1);
41 }
42
43 unsigned int cantidad_adultos(unsigned int mes){
44     /*
45      * La cantidad de adultos en un mes es igual a la cantidad
46      * de adultos y bebes del mes anterior. Si el mes es el primero,
47      * la cantidad de adultos es 0.
48      */
49
50     if(mes == 1)
51         return 0;
52
53     return cantidad_adultos(mes-1)+cantidad_bebes(mes-1);
54 }

```

## 4. Recursividad De Cola

Es una técnica para optimizar la recursividad eliminando las constantes llamadas recursivas. Cuando se realiza una llamada recursiva se genera un ámbito nuevo por cada llamada, este ambito se traduce en el stack de ejecución como un **stack frame**, es decir, un ámbito es equivalente a un stack frame, en la pila de ejecución.

**Tail recursion** se da cuando la llamada recursiva es la última instrucción de la función. Sin embargo, las funciones tail recursive deben cumplir la condición que en la parte que realiza la llamada a la función, no debe existir ninguna otra instrucción.

Una ventaja de la recursividad por cola es que se puede evitar la sobrecarga de cada llamada a la función y se evita el gasto de memoria de pila. Con una función tail recursive se puede evitar lo que se conoce como stack overflow, que ocurre cuando la pila de llamadas (call stack) consume mucha memoria.

### 4.0.1. Función Recursiva Clásica

```

1 #include <stdio.h>
2
3 unsigned factorial(unsigned n){
4     if(n==0)
5         return 1;
6     return n*factorial(n-1);
7 }
8
9 int main(){

```

```

10     unsigned resultado=0;
11     unsigned n=15;
12     resultado = factorial(n);
13     printf("%u! = %u\n", n, resultado);
14     return 0;
15 }

```

En este pequeño caso de calcular el  $n!$ , con  $n=15$  el stack de llamadas debe apilar 15 entradas, una por cada llamada recursiva. El resultado sería 1307474368000. Y el stack se comportaría de la siguiente forma

```

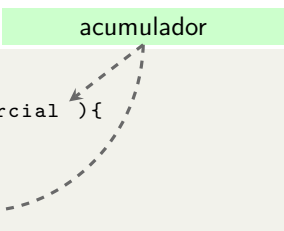
factorial(15)
15 * factorial(14)
15 * (14 * factorial(13))
15 * (14 * (13 * factorial(12)))
15 * (14 * (13 * (12 * factorial(11))))
15 * (14 * (13 * (12 * (11 * factorial(10)))))
15 * (14 * (13 * (12 * (11 * (10 * factorial(9))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * factorial(8)))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * factorial(7))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * factorial(6)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * factorial(5)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * (5 * factorial(4)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * (5 * (4 * factorial(3)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * (5 * (4 * (3 * factorial(2)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * (5 * (4 * (3 * (2 * factorial(1)))))))))
15 * (14 * (13 * (12 * (11 * (10 * (9 * (8 * (7 * (6 * (5 * (4 * (3 * (2 * 1)))))))))
1307474368000

```

Puede verse como en cada llamada recursiva se crea un nuevo ámbito hasta llegar a la condición de corte. Si fueran necesarias 10000 llamadas, se tendrían 10001 ámbitos creados, uno por cada llamada y el de la función que llamó originalmente a la función recursiva.

#### 4.0.2. Recursividad de Cola o Tail Recursión

En la versión tail recursive, se puede ver como en cada llamada se pasa por parámetro el subtotal del cálculo que se está resolviendo, con lo cual esto hace posible que no sea necesario crear un ámbito nuevo por cada llamada. Justamente este hecho es conocido por los compiladores, que realizan esta optimización. Lo que permite que si para realizar el cálculo es necesario 10000 llamadas solamente se creen 3 ámbitos o stack frames.



```

1 #include <stdio.h>
2
3 unsigned factorial_rec(unsigned n, unsigned parcial){
4     if(n==0)
5         return parcial;
6
7     return factorial_rec(n-1, parcial*n);
8 }
9
10 unsigned factorial(unsigned n){
11     return factorial_rec(n,1);
12 }
13
14 int main(){
15     unsigned resultado=0;
16     unsigned n=15;
17     resultado = factorial(n);
18     printf("%u = return 0;

```

En el caso de recursividad por cola las llamadas serían:

```

factorial_rec(1, 1307474368000)
factorial_rec(2, 87178291200)
factorial_rec(3, 6227020800)
factorial_rec(4, 479011600)

```

```
factorial_rec(5, 39916800)
factorial_rec(6, 3628800)
factorial_rec(7, 362880)
factorial_rec(8, 40320)
factorial_rec(9, 5040)
factorial_rec(10, 720)
factorial_rec(11, 120)
factorial_rec(12, 24)
factorial_rec(13, 6)
factorial_rec(14, 2)
factorial_rec(15, 1)
1307474368000
```

El resultado igualmente sería 1307474368000, pero el costo computacional baja drásticamente debido a que cuando se hace la llamada `return fact_tail_sum(n - 1, sum*n)` ambos parámetros son inmediatamente resueltos. Con esta llamada podemos calcular sin esperar una llamada a una función recursiva para que nos devuelva un valor.

En este caso el compilador reduce drásticamente el uso de la pila y la cantidad de información que debe ser almacenada, el valor de `n` y `sum` es independiente del número de llamadas recursivas.

Una función recursiva normal se puede convertir a tail recursive usando en la función original un parámetro adicional, usado para ir guardando un resultado de tal manera que la llamada recursiva ya no tiene una operación pendiente.

También se usa una función adicional para mantener la sintaxis de como llamamos normalmente a la función. En el caso del factorial, para seguir llamando a la función de la forma `fact(n)`. En conclusión:

- Una llamada es tail recursive (recursiva por cola) si no tiene que hacer nada más después de la llamada de retorno.
- Tail recursion es cuando la llamada recursiva es la última instrucción en la función.
- Usar tail recursion es muy ventajoso, porque la cantidad de información que debe ser almacenada durante el cálculo es independiente del número de llamadas recursivas.
- El compilador trata una función tail recursive como si fuera una función iterativa.

## 4.1. Divide y Conquista

Si bien en programación divide y conquista es una forma general de solucionar problemas, en el caso de algoritmos, es una metodología de diseñarlos.

Muchos algoritmos recursivos usan la siguiente estrategia para resolver un determinado problema: se llaman a sí mismos una o más veces para resolver sub-problemas de la misma naturaleza pero de menor tamaño. Si los sub-problemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar sub-problemas lo suficientemente pequeños para ser solucionados directamente. Y posteriormente combinar estas soluciones (sub-soluciones) en la solución al problema original [1, 2].

El método de divide y conquista está compuesto de tres fases:

1. **Dividir:** el problema en un número de sub-problemas que sean instancias menores del mismo problema. Es decir, si el tamaño de la entrada es  $n$ , hay que conseguir dividir el problema en  $k$  subproblemas (donde  $1 < k < n$ ), cada uno con una entrada de tamaño  $nk$  y donde  $0 < nk < n$ .
2. **Conquistar:** los subproblemas de modo tal que sean resueltos recursivamente. si el tamaño del subproblema es lo suficientemente pequeño la solución es simple. El hecho de que el tamaño de los subproblemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3. **Combinar:** las soluciones obtenidas de cada subproblema en la solución del problema original

La recursividad es una ecuación o inecuación que describe una función en términos de sus valores en función de entradas más pequeños, el esquema básico de la estrategia “**divide y vencerás**”, se ve en la Figura 6.

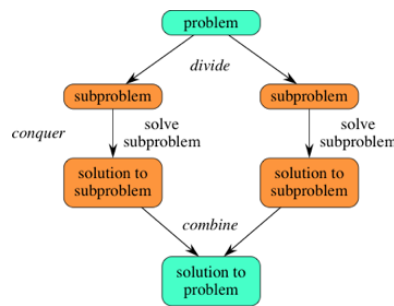


Figura 6

La idea es combinar esta estrategia para diversos valores, teniendo en cuenta que los subproblemas son siempre de la “**misma naturaleza**”, como se vé en la Figura 7:

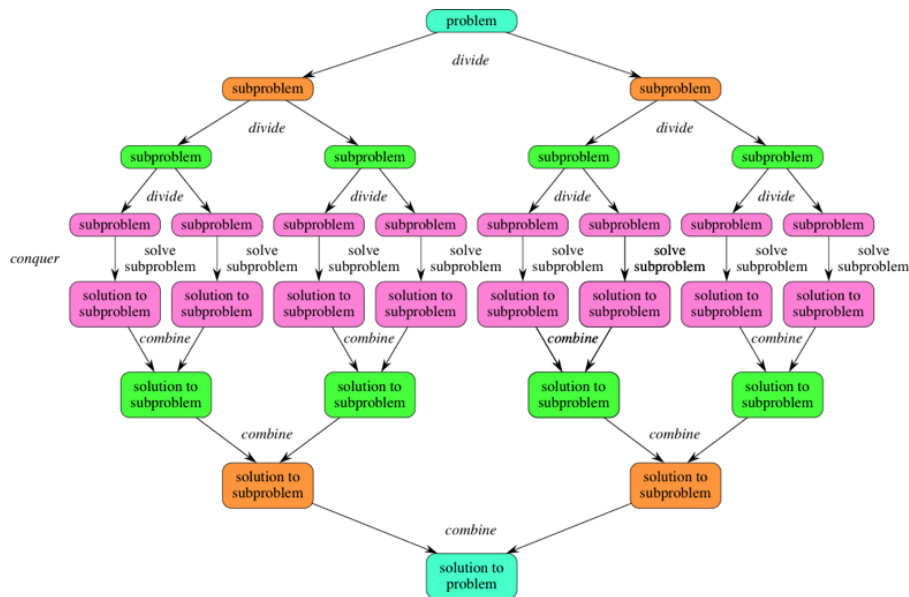


Figura 7

A continuación se ve un pseudo código de la estrategia de un algoritmo recursivo hecho con esta estrategia:

```

1 TipoSolucion DyV(TipoProblema x){
2     size_t i,k;
3     TipoSolucion s;
4     TipoProblema subproblemas[];
5     TipoSolucion subsoluciones[];
6
7     if ( EsCasoBase(x) )
8         s = ResuelveCasoBase(x)
9     else{
10        k = Divide(x,subproblemas);
11        FOR ( int i; i<=k ; i++)
12            subsoluciones[i] = DyV(subproblemas[i]);
13        s = Combina(subsoluciones);
14    }
15    return s;
16 }
  
```

Un punto muy importante de este método de diseño de algoritmos es que la solución de cada subproblema debe ser independiente, ya que sino el tiempo de ejecución del algoritmo tendería a exponencial.

Una característica del método de división y conquista, es que el tiempo de ejecución de los algoritmos que utilizan este método es una ecuación recurrente del tipo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ T(n) = aT(n/b) + cn^k & \text{si } n \geq b \end{cases}$$



### 4.1.1. Ejemplo

Un ejemplo clásico de la utilización de la técnica de divide y conquista es el algoritmo de búsqueda binaria.

```
1 int busqueda_binaria(vector_t vector, int inicio, int fin,
2     elemento_t elemento_buscado) {
3     int centro;
4
5     if(inicio<=fin){
6         centro=(fin+inicio)/2;
7         if(vector[centro]==elemento_buscado)
8             return centro;
9         else if(elementoBuscado < vector[centro])
10            return busqueda_binaria(vector, inicio, centro-1, elemento_buscado);
11
12        else
13            return busqueda_binaria(vector, centro+1, fin, elemento_buscado);
14    } else
15        return -1;
```

## Referencias

- [1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- [2] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 199.