Tema VII

Recursividad

Objetivos:

- ▷ **Entender el concepto de recursividad.**
- ▷ **Conocer los fundamentos del diseño de algoritmos recursivos.**
- ▷ **Comprender la ejecución de algoritmos recursivos.**
- ▷ **Aprender a realizar trazas de algoritmos recursivos.**
- ▷ **Comprender las ventajas e inconvenientes de la recursividad.**

Este tema ha sido elaborado por Francisco Cortijo y Juan Carlos Cubero

VII.1. El concepto matemático de Recursividad

VII.1.1. Soluciones recursivas

"La iteración es humana; la recursividad, divina".

"Para entender la recursividad es necesario antes entender la recursividad".

Recursividad (recursion) **es la forma en la cual se especifica un proceso basado en su propia definición.**

Podemos usar recursividad si la solución de un problema está expresada en términos de la resolución del mismo tipo de problema, aunque de *menor tamaño* y conocemos la solución no-recursiva para un determinado *caso base (base case)* (o varios).

En Matemáticas se usa este concepto en varios ámbitos:

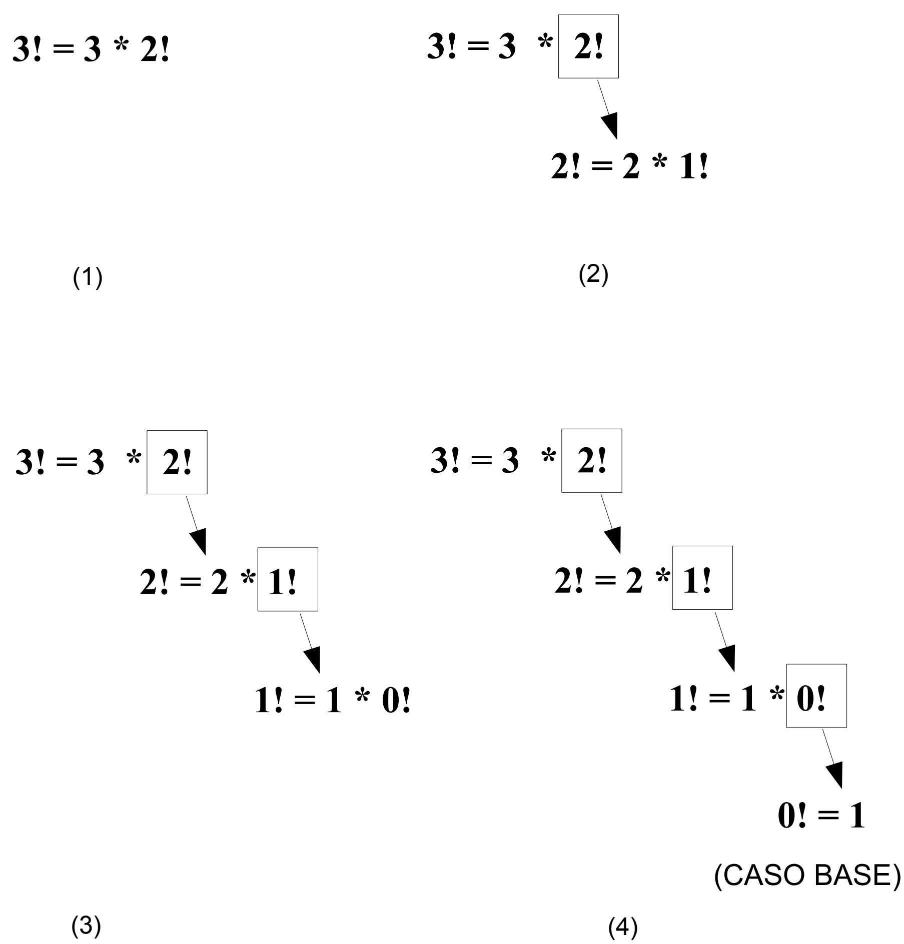
- ▷ **Demostración por *inducción (induction)* :**
  - **Demostrar para un caso base.**
  - **Demostrar para un tamaño n, considerando que está demostrado para un tamaño menor que n.**
- ▷ **Definición recursiva**

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

▷ **No es necesario definir la secuencia de pasos exacta para resolver el problema. Podemos considerar que *“tenemos resuelto”* el problema (de menor tamaño).**

▷ **Usaremos recursividad cuando surja *<u>de forma natural</u>* al plantear la solución a un problema. Los problemas más interesantes se verán en el segundo cuatrimestre.**

Ejemplo. **Cálculo del factorial con n=3.**



$$3! = 3 * \boxed{2!}$$
$$2! = 2 * \boxed{1!}$$
$$1! = 1 * 1 \quad (0!)$$
$$1$$

(5)

$$3! = 3 * \boxed{2!}$$
$$2! = 2 * 1 \quad (1!)$$
$$1! = 1 * 1 = 1$$

(6)

$$3! = 3 * 2 \quad (2!)$$
$$2! = 2 * 1 = 2$$

(7)

$$3! = 3 * 2 = 6$$

(8)

En resumen:

$$0! = 1$$
$$1! = 1 * \boxed{0!}$$
$$2! = 2 * \boxed{1!}$$
$$3! = 3 * \boxed{2!}$$

$$1! = 1*1 = 1$$
$$2! = 2 * \boxed{1!}$$
$$3! = 3 * \boxed{2!}$$

etc.

VII.1.2. Diseño de algoritmos recursivos

Debemos diseñar:

▷ ***Casos base*:**

Son los casos del problema que se resuelve con un segmento de código sin recursividad.

Para que una definición recursiva esté completamente identificada es necesario tener un caso base que no se calcule utilizando casos anteriores y que la división del problema converja a ese caso base.

Siempre debe existir al menos un caso base

El número y forma de los casos base son hasta cierto punto arbitrarios. La solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

```
0! = 1
```

▷ ***Casos generales*:**

Debemos encontrar la solución a un problema (o varios) del mismo tipo, pero de menor tamaño.

```
(n-1)!
```

▷ ***Composición*:**

Una vez encontradas las soluciones a los problemas menores, habrá que ejecutar un conjunto de pasos adicionales. Estos pasos, junto con las soluciones a los subproblemas, componen la solución al problema general que queremos resolver.

```
n! = n * (n-1)!
```

Ejercicio. **Plantear la solución recursiva al problema de calcular la potencia de un real x elevado a un entero positivo n, es decir: x^n**

▷ **Expresado con palabras:**

- **Caso base: Solución de Potencia$(x, 0) = 1$**
- **Caso general:**
  Para hallar la solución de Potencia(x, n),
  hallar primero la solución de Potencia$(x, n-1)$
- **Composición:**
  Solución de Potencia$(x, n) =$
  $x *$ Solución de Potencia$(x, n-1)$

▷ **Expresado en lenguaje matemático:**

- **Caso base: $x^0 = 1$**
- **Caso general: x^{n-1}**
- **Composición: $x^n = x * x^{n-1}$**

Ejercicio. **Plantear la solución recursiva al problema de sumar los enteros positivos menores que un entero n, es decir: $\sum_{i=1}^{i=n} i$**

▷ **Expresado con palabras:**

– **Caso base: Solución de SumaHasta$(1) = 1$**

– **Caso general:**

Para hallar la solución de SumaHasta(n),

hallar primero la solución de SumaHasta$(n-1)$,

– **Composición:**

Solución de SumaHasta$(n) =$

$n +$ Solución de SumaHasta$(n-1)$

▷ **Expresado en lenguaje matemático:**

– **Caso base: $\sum_{i=1}^{i=1} i = 1$**

– **Caso general: $\sum_{i=1}^{i=n-1} i$**

– **Composición: $\sum_{i=1}^{i=n} i = n + \sum_{i=1}^{i=n-1} i$**

Ejercicio. **Plantear la solución recursiva al problema de multiplicar dos enteros a y b, es decir: $a * b$**

- ▷ **Expresado con palabras:**
  - **Caso base: Solución de Multiplica cualquier número con 0** $= 0$
  - **Caso general:**

    Para hallar la solución de Multiplica(a, b)**,**

    hallar primero la solución de Multiplica$(a, b-1)$
  - **Composición:**

    Solución de Multiplica$(a, b) =$

    $a +$ **Solución de Multiplica**$(a, b-1)$
- ▷ **Expresado en lenguaje matemático:**
  - **Casos base:** $0 * b = 0, \;\; a * 0 = 0$
  - **Caso general:** $a * (b-1)$
  - **Composición:** $a * b = a + a * (b-1)$

VII.2. Funciones recursivas

VII.2.1. Definición de funciones recursivas

¿Cómo se expresa en un lenguaje de programación la solución a un problema? Una solución será el valor devuelto por la llamada a una función.

$n! \rightarrow$ `Factorial(n)`

¿Cómo se traduce a un lenguaje de programación el planteamiento recursivo de una solución? A través de una función que, dentro del código que la define, realiza una o más llamadas a la propia función. Una función que se llama a sí misma se denomina *función recursiva (recursive function)* .

```
\\ Prec:  0 <= n <= 20
long long Factorial (int n) {
   long long resultado;

   if (n==0)
      resultado = 1;                //Caso base
   else
      resultado = n*Factorial(n-1); //Caso general

   return resultado;
}

\\ Prec:  0 <= n <= 20
long long Factorial (int n) {
   if (n==0)
      return 1;                     //Caso base
   else
      return n*Factorial(n-1);      //Caso general
}
```

VII.2.2. Ejecución de funciones recursivas

En general, en la pila se almacena el marco asociado a las distintas funciones que se van activando. En particular, cada llamada recursiva genera una nueva zona de memoria en la pila independiente del resto de llamadas.

Ejemplo. **Ejecución del factorial con n=3.**

1. **Dentro de factorial, cada llamada**

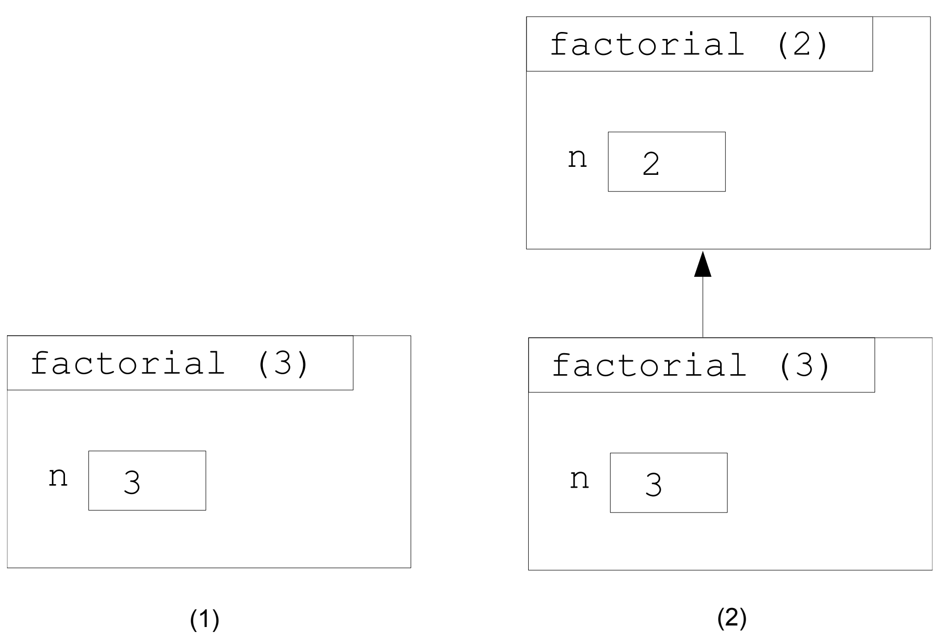
   ```
   return (n * factorial(n-1));
   ```

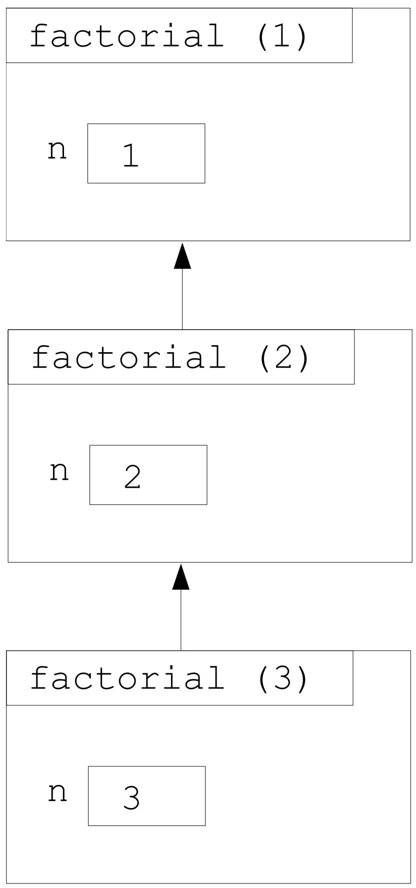
   genera una nueva zona de memoria en la pila, siendo `n-1` el correspondiente parámetro actual para esta zona de memoria y queda pendiente la evaluación de la expresión y la ejecución del `return`.

2. **El proceso anterior se repite hasta que la condición del caso base se hace cierta.**

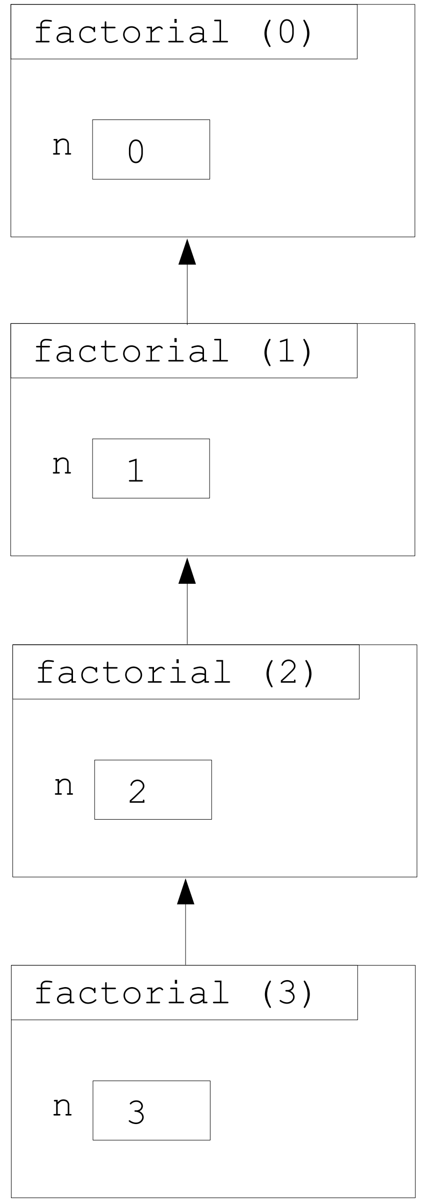
   ▷ **Se ejecuta la sentencia `return 1;`**

   ▷ **Empieza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los `return` que estaban pendientes.**

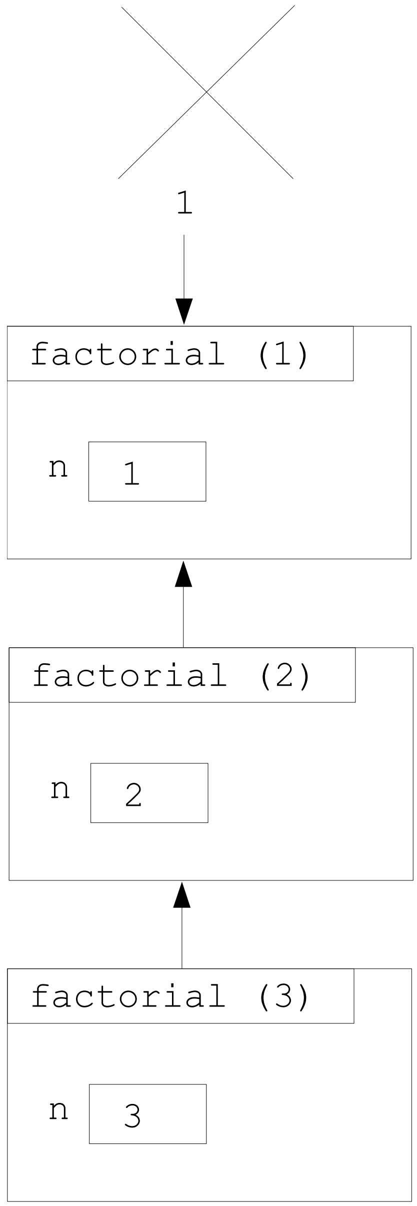



factorial (1)
n
1
factorial (2)
n
2
factorial (3)
n
3

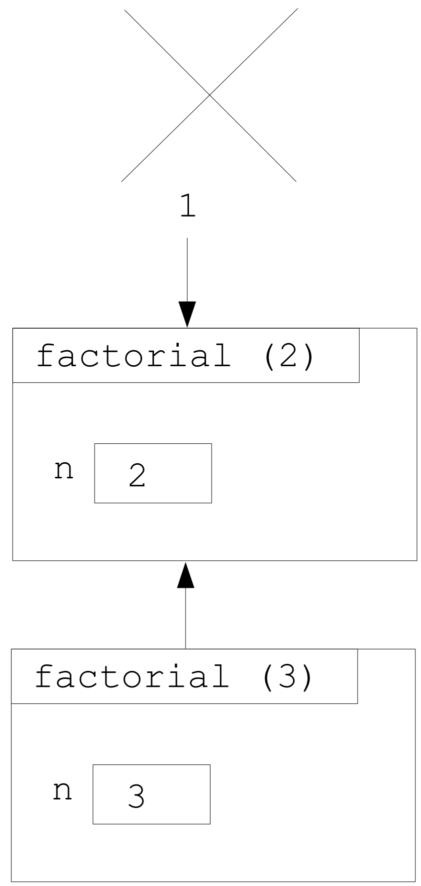
(3)


factorial (0)
n
0
factorial (1)
n
1
factorial (2)
n
2
factorial (3)
n
3

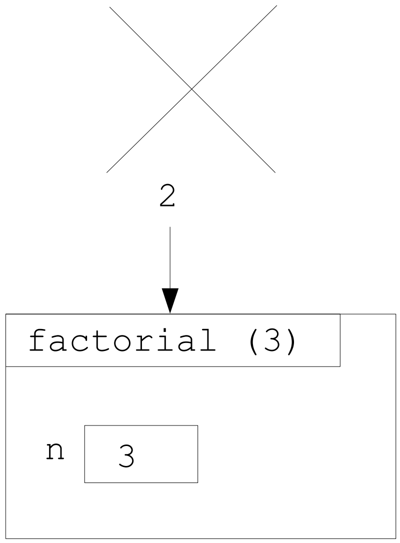
(4)


1
factorial (1)
n
1
factorial (2)
n
2
factorial (3)
n
3

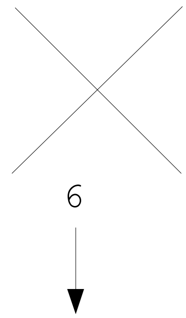
(5)


1
factorial (2)
n
2
factorial (3)
n
3

(6)



(7)



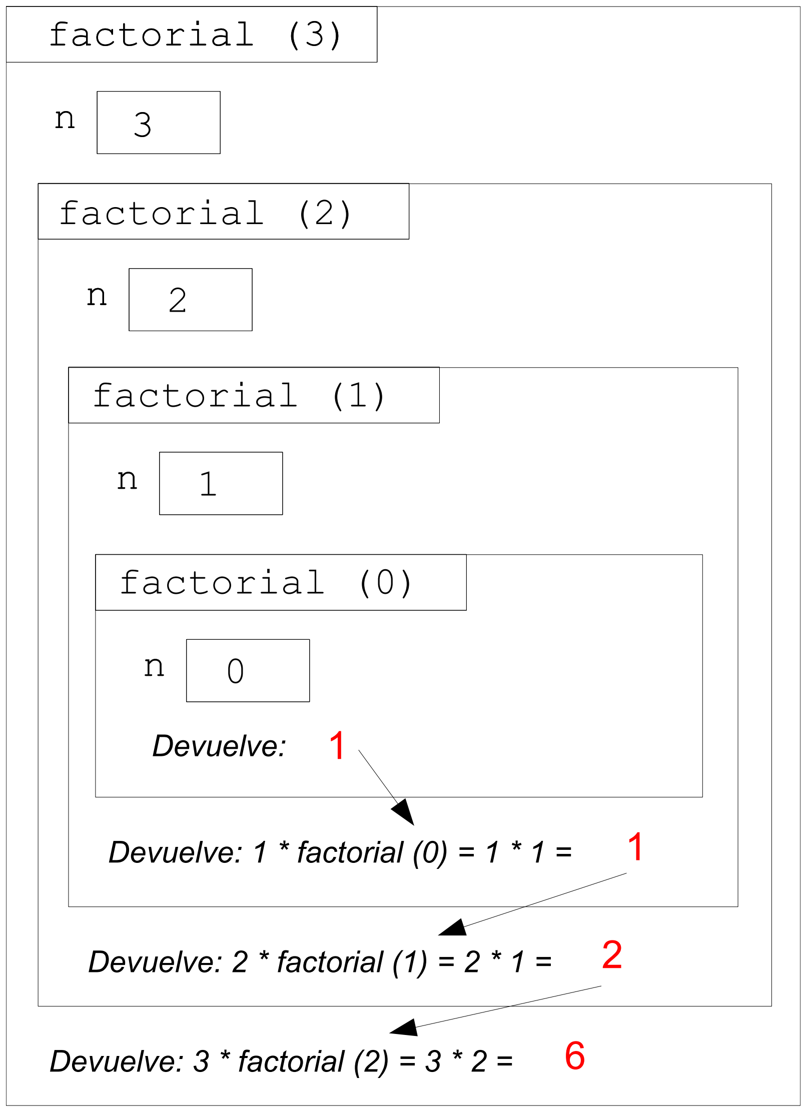
(8)

Cada llamada a una función recursiva genera un marco de trabajo en la pila → Cada marco tiene su propia copia del parámetro formal.

Cada llamada a una función recursiva genera un marco de trabajo en la pila → Esto supone una recarga computacional importante

Una forma alternativa de representar gráficamente las llamadas recursivas es hacerlo *en cascada*:

Llamada: `factorial(3)`



Ejercicio. **Calcular la potencia de un real elevado a un entero positivo n**

- ▷ **Caso base ($n = 0$):** $x^0 = 1$
- ▷ **Caso general:** x^{n-1}
- ▷ **Composición:** $x^n = x * x^{n-1}$

```
// Prec: n >= 0
double Potencia(double x, int n){
   if (n == 0)
      return 1.0;
   else
      return x * Potencia(x, n - 1);
}
```

`Potencia(3.0, 300)` **devuelve** `3.63603 e+238`

`Potencia(3.0, 700)` **devuelve** `1.#INF`

`Potencia(3.0, 800)` **devuelve** `1.#INF`

`Potencia(3.0, 2000)` **produce un error en tiempo de ejecución.**

Como cada llamada recursiva implica la creación de un marco de trabajo en la pila, la memoria asociada al proceso del programa puede saturarse, produciéndose entonces un *desbordamiento de la pila (stack overflow)* y por tanto, un error en tiempo de ejecución.

Ejercicio. Sumar los enteros positivos menores que un entero n

- **Caso base ($n = 1$):** $\sum_{i=1}^{i=1} i = 1$
- **Caso general:** $\sum_{i=1}^{i=n-1} i$
- **Composición:** $\sum_{i=1}^{i=n} i = n + \sum_{i=1}^{i=n-1} i$

```
\\ Prec: n > 0
long long SumaPositivosHasta(int n){
   if (n <= 1)
      return 1;
   else
      return n + SumaPositivosHasta(n - 1);
}
```

Si se viola la precondición

```
total = SumaPositivosHasta(-5);
```

el resultado sería un valor erróneo. Si optamos por suprimir la precondición, entonces debemos añadir más casos base:

```
long long SumaPositivosHasta(int n){
   if (n <= 0)
      return -1;   // o mejor, lanzar una excepción
   else if (n == 1)
      return 1;
   else
      return n + SumaPositivosHasta(n - 1);
}
```

Supongamos la llamada siguiente:

```
int main(){
   long long suma_total;

   suma_total = SumaPositivosHasta(10);
```

¿Cuántas veces se comprueban las condiciones?:

- `if (n <= 0)`: **11 veces.**
- `if (n == 1)`: **11 veces**

Los condicionales incluidos en el código de una función recursiva se evalúan cada vez que hay una llamada recursiva.

Si suponemos que las llamadas se realizarán usualmente con valores positivos (10, 4, 8, etc), podríamos mejorar la eficiencia cambiando el orden de comprobación:

```
long long SumaPositivosHasta(int n){
   if (n > 1)
      return n + SumaPositivosHasta(n - 1);
   else if (n == 1)
      return 1;
   else
      return -1;  // o mejor, lanzar excepción
}
```

De todas formas, como ya hemos señalado anteriormente, la recarga computacional importante es debida a la creación de un marco en la pila por cada llamada recursiva.

Ejercicio. **Multiplicar dos enteros positivos $a * b$**

- ▷ **Casos base:** $0 * b = 0, \;\; a * 0 = 0$
- ▷ **Caso general:** $a * (b - 1)$
- ▷ **Composición:** $a * b = a + a * (b - 1)$

```
long long Producto(unsigned int a, unsigned int b){
   if (b == 0 || a == 0)
      return 0;
   else
      return a + producto(a, b-1);
}
```

El caso base correspondiente al condicional `a == 0` no sería estrictamente necesario: en las llamadas recursivas iría sumando 0 y también funcionaría correctamente. En definitiva:

```
long long Producto(unsigned int a, unsigned int b){
   if (b == 0 || a == 0)          // Realiza siempre más comprobaciones
      return 0;
   else
      return a + producto(a, b-1);
}
```

```
long long Producto(unsigned int a, unsigned int b){
   if (b == 0)                    // Realiza más llamadas cuando a es 0
      return 0;
   else
      return a + producto(a, b-1);
}
```

VII.3. Clases con métodos recursivos

VII.3.1. Métodos recursivos

El planteamiento y definición de un método recursivo es análogo a las funciones. La diferencia es que todas las llamadas recursivas del método pueden acceder a los datos miembro del objeto. Si una llamada recursiva modifica algún dato miembro, el resto de llamadas se verán afectadas, ya que todas ellas acceden a los mismos datos miembro.

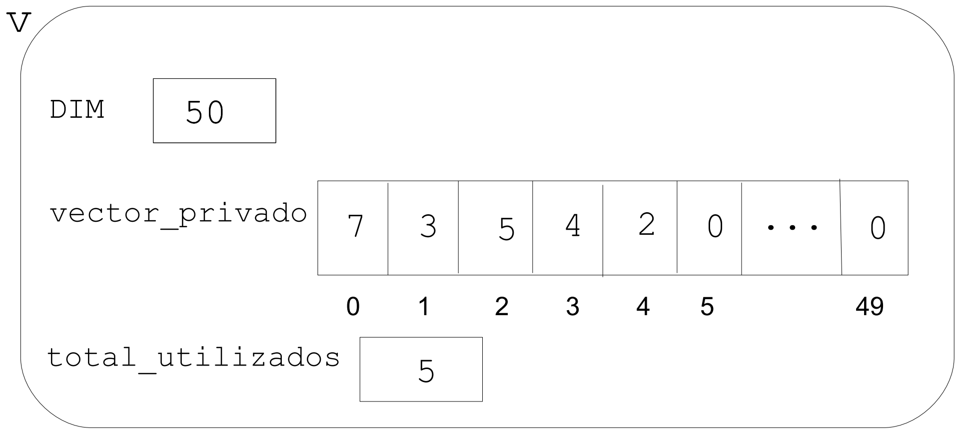
> ***Cada llamada a un método recursivo genera un marco de trabajo en la pila. Cada marco tiene su propia copia del parámetro formal. Sin embargo, todos los marcos acceden a los mismos datos miembro.***

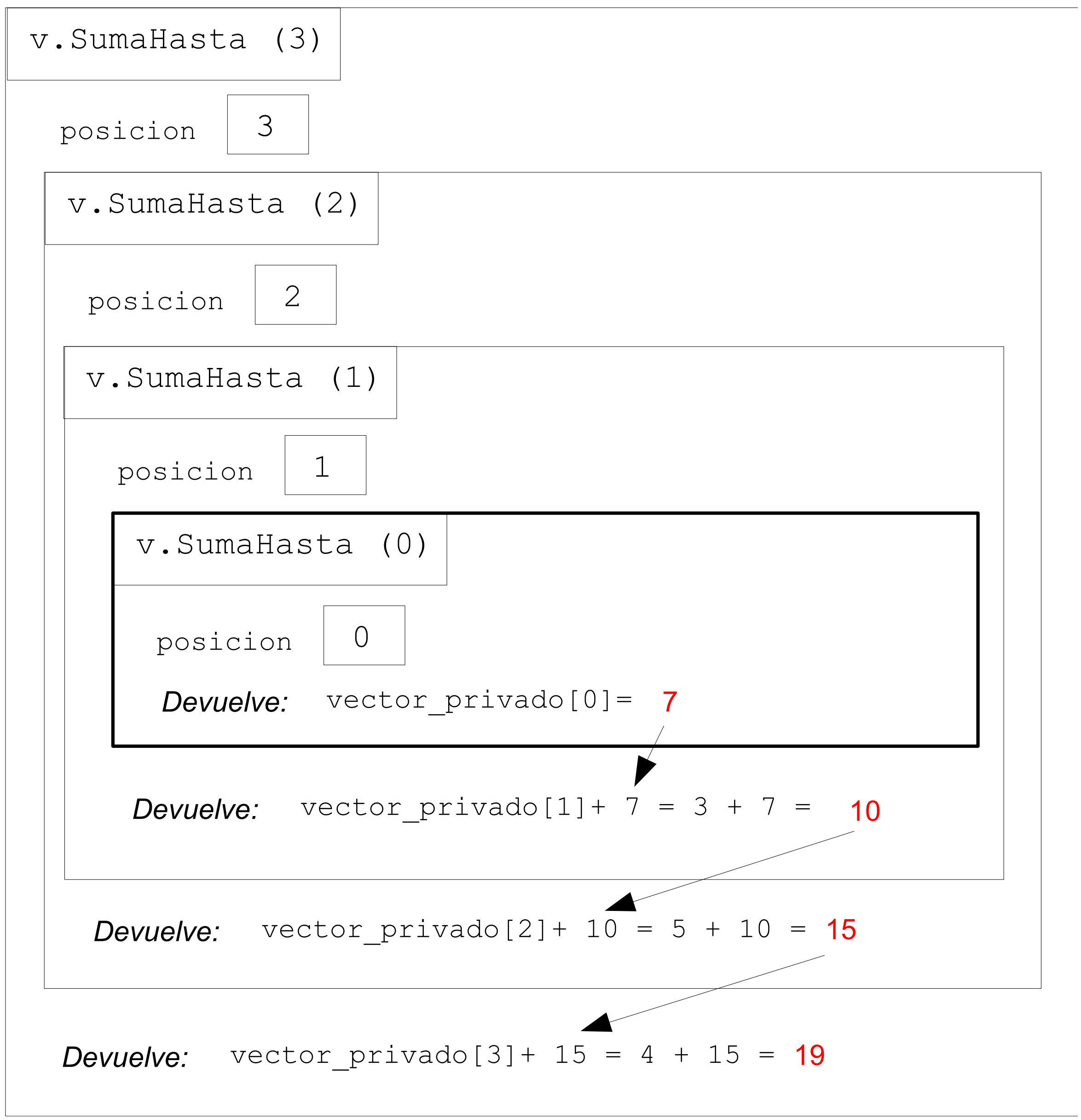
Ejercicio. **Plantear una solución recursiva al problema de sumar los elementos de un vector y definir un método recursivo que la implemente.**

```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   ......
   long long SumaHasta(int posicion){
      ......
   }
```

Cada llamada recursiva tendrá su propio valor de `posicion`**, pero todas las llamadas accederán al mismo dato miembro** `vector_privado`

```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   MiVectorEnteros()
      :total_utilizados(0)
   {
   }
   int TotalUtilizados(){
      return total_utilizados;
   }
   void Aniade(int nuevo){
      if (total_utilizados < DIM){
         vector_privado[total_utilizados] = nuevo;
         total_utilizados++;
      }
      // else
      //    lanzar excepción
   }
   int Elemento(int indice){
      return vector_privado[indice];
   }
   long long SumaHasta(int posicion){
      if (posicion > 0 && posicion < total_utilizados)
         return vector_privado[posicion] + SumaHasta(posicion-1);
      else if (posicion == 0)
         return vector_privado[0];
      // else
      //    lanzar excepción
   }
};
```

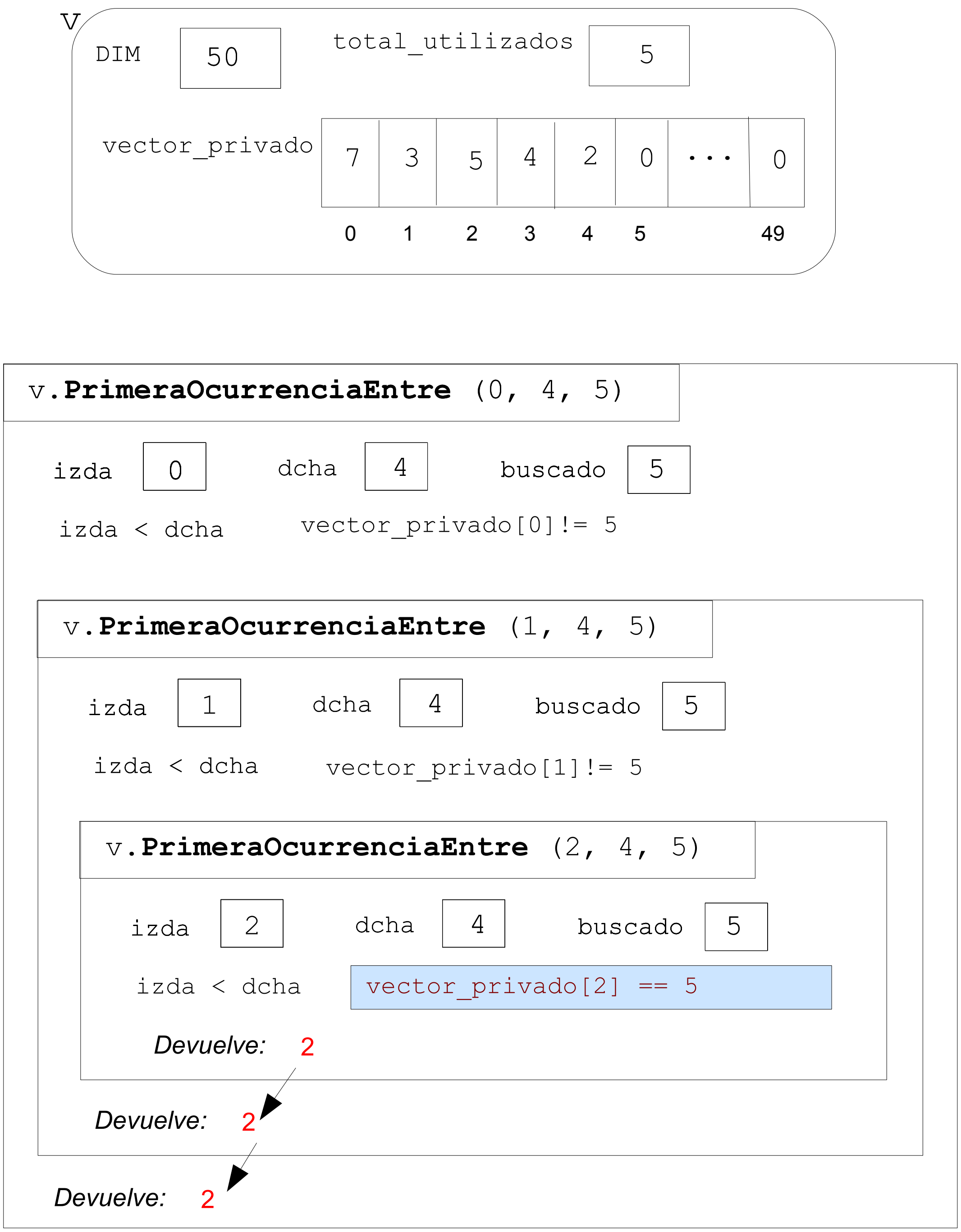

v
DIM
50
vector_privado
7
3
5
4
2
0
. . .
0
0
1
2
3
4
5
49
total_utilizados
5

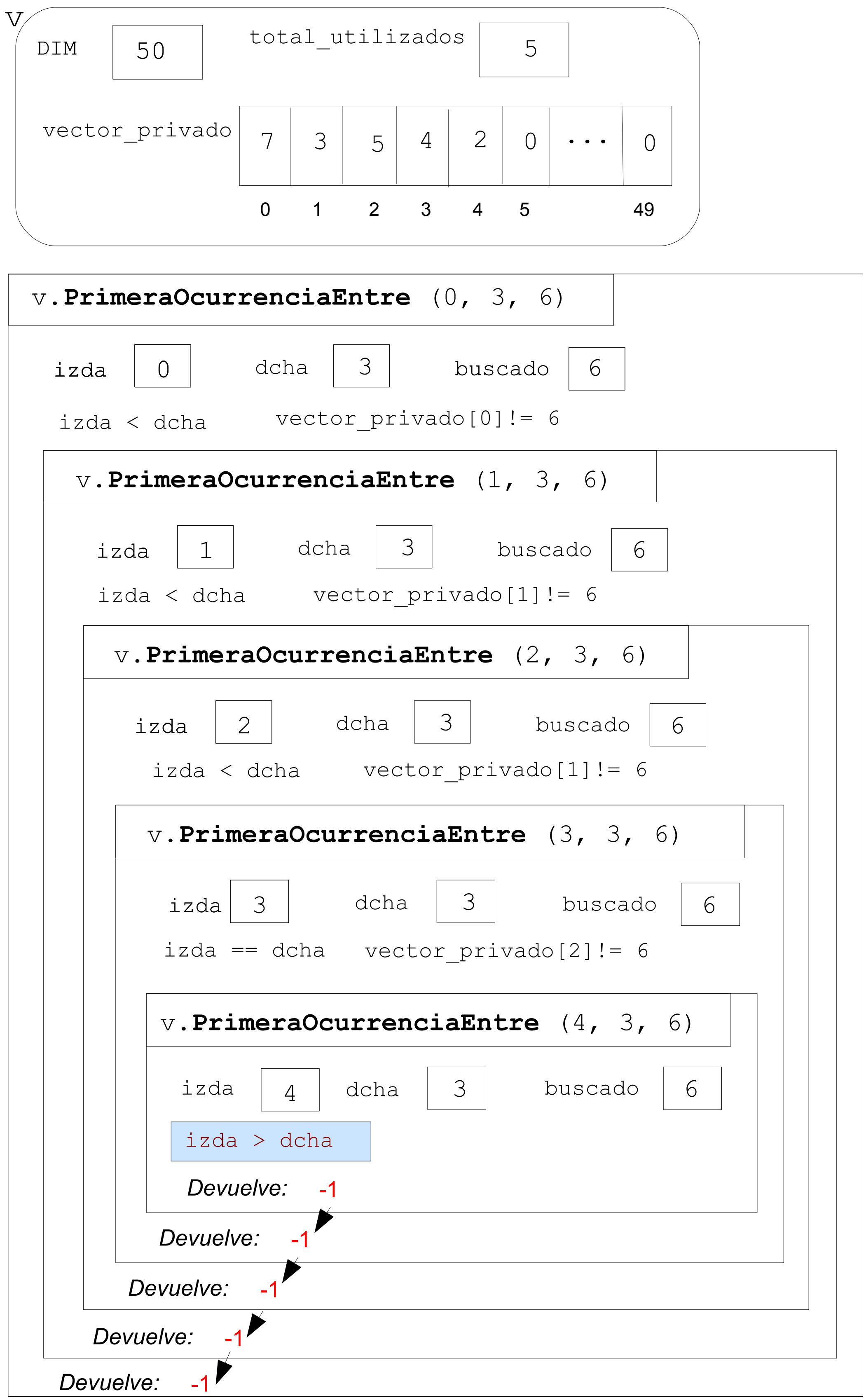

v.SumaHasta (3)
posicion
3
v.SumaHasta (2)
posicion
2
v.SumaHasta (1)
posicion
1
v.SumaHasta (0)
posicion
0
Devuelve: vector_privado[0]= 7
Devuelve: vector_privado[1]+ 7 = 3 + 7 = 10
Devuelve: vector_privado[2]+ 10 = 5 + 10 = 15
Devuelve: vector_privado[3]+ 15 = 4 + 15 = 19

Ejercicio. **Plantear la solución recursiva al problema de buscar la primera ocurrencia de una componente de un vector y definir un método que la implemente.**

Observad que, en este ejemplo, se necesitan dos casos base.

```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   .......
   // Prec: 0 < izda , dcha < TotalUtilizados()
   int PrimeraOcurrenciaEntre(int izda, int dcha, int buscado){
      if (izda > dcha)
         return -1;
      else if (vector_privado[izda] == buscado)
         return izda;
      else
         return PrimeraOcurrenciaEntre(izda + 1, dcha, buscado);
   }
};
```


v
DIM 50
total_utilizados 5
vector_privado
7 3 5 4 2 0 ··· 0
0 1 2 3 4 5 49
v.PrimeraOcurrenciaEntre (0, 4, 5)
izda 0
dcha 4
buscado 5
izda < dcha
vector_privado[0]!= 5
v.PrimeraOcurrenciaEntre (1, 4, 5)
izda 1
dcha 4
buscado 5
izda < dcha
vector_privado[1]!= 5
v.PrimeraOcurrenciaEntre (2, 4, 5)
izda 2
dcha 4
buscado 5
izda < dcha
vector_privado[2] == 5
Devuelve: 2
Devuelve: 2
Devuelve: 2

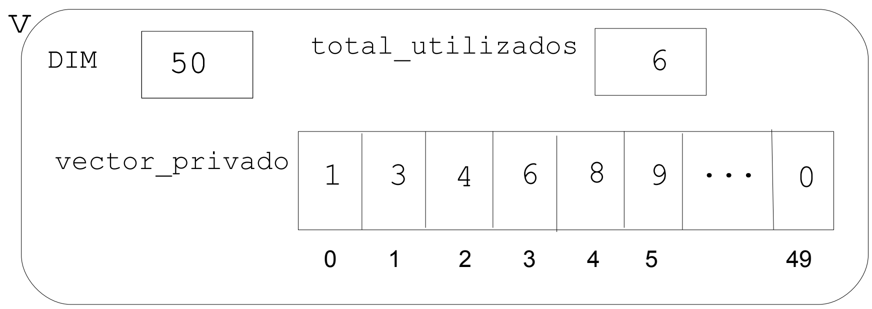

v
DIM
50
total_utilizados
5
vector_privado
7
3
5
4
2
0
· · ·
0
0
1
2
3
4
5
49
v.PrimeraOcurrenciaEntre (0, 3, 6)
izda
0
dcha
3
buscado
6
izda < dcha
vector_privado[0]!= 6
v.PrimeraOcurrenciaEntre (1, 3, 6)
izda
1
dcha
3
buscado
6
izda < dcha
vector_privado[1]!= 6
v.PrimeraOcurrenciaEntre (2, 3, 6)
izda
2
dcha
3
buscado
6
izda < dcha
vector_privado[1]!= 6
v.PrimeraOcurrenciaEntre (3, 3, 6)
izda
3
dcha
3
buscado
6
izda == dcha
vector_privado[2]!= 6
v.PrimeraOcurrenciaEntre (4, 3, 6)
izda
4
dcha
3
buscado
6
izda > dcha
Devuelve: -1
Devuelve: -1
Devuelve: -1
Devuelve: -1
Devuelve: -1

Ejercicio. **Plantear la solución recursiva al problema de buscar con el método de *búsqueda binaria* la primera ocurrencia de una componente de un vector ordenado y definir un método que la implemente.**

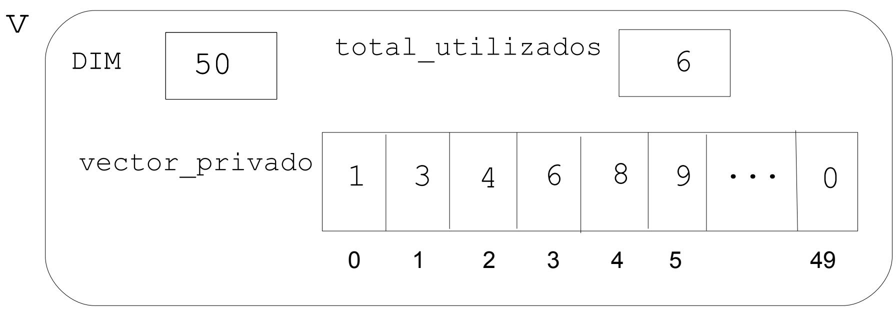
Recordemos el método de búsqueda binaria:

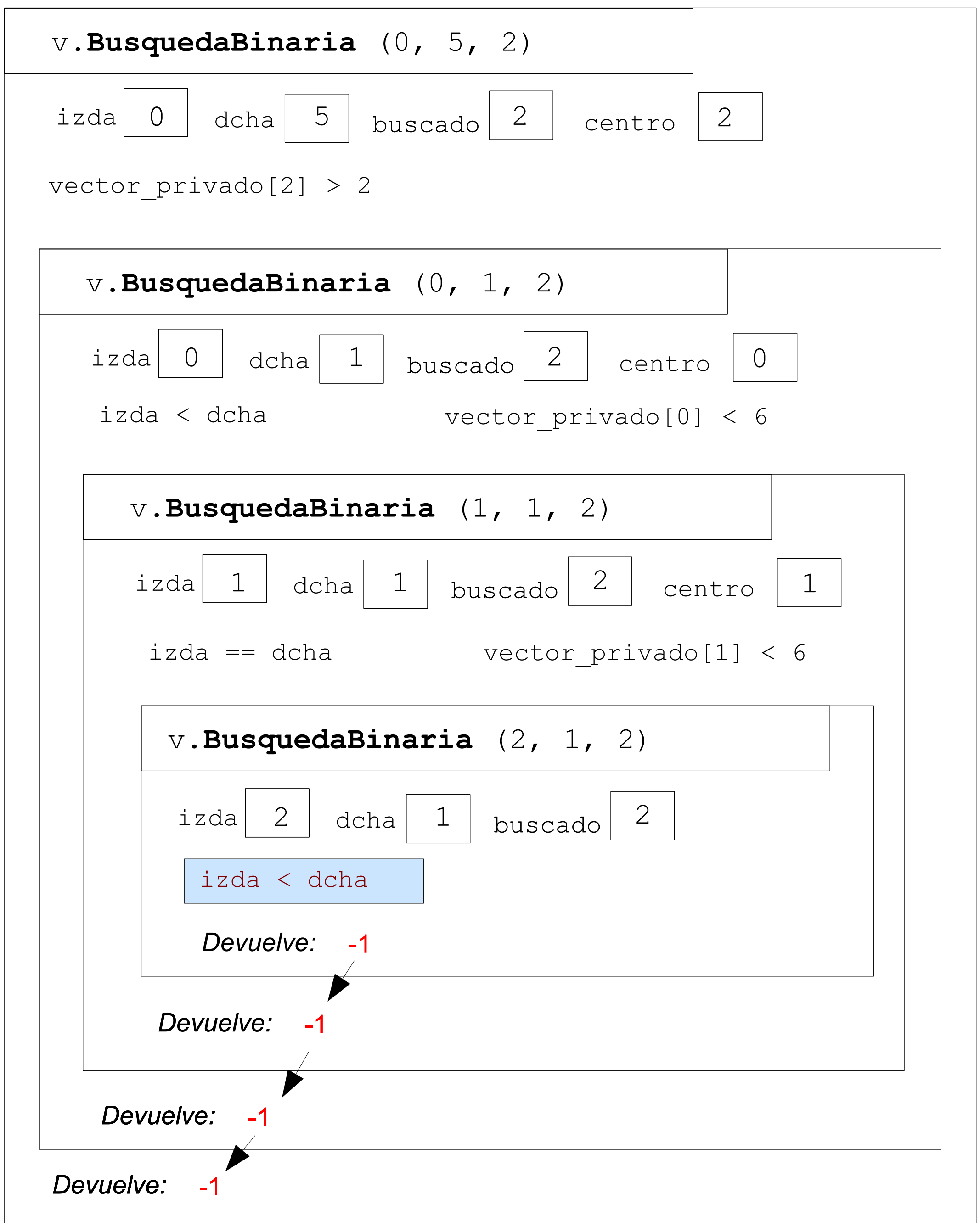
```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   .......
   int BusquedaBinaria (int buscado){
      int izda, dcha, centro;

      izda = 0;
      dcha = total_utilizados-1;
      centro = (izda+dcha)/2;

      while (izda <= dcha  &&  vector_privado[centro] != buscado) {
         if (buscado < vector_privado[centro])
            dcha = centro-1;
         else
            izda = centro+1;

         centro = (izda+dcha)/2;
      }

      if (izda > dcha)
         return -1;
      else
         return centro;
   }
};
```

```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   .......
   // Prec: 0 < izda , dcha < TotalUtilizados()
   int BusquedaBinaria(int izda, int dcha, int buscado){
      int centro;

      if (izda > dcha)
         return -1;
      else{
         centro = (izda + dcha) / 2;

         if (vector_privado[centro] == buscado)
            return centro;
         else
            if (vector_privado[centro] > buscado)
               return BusquedaBinaria(izda, centro-1, buscado);
            else
               return BusquedaBinaria(centro + 1, dcha, buscado);
      }
   }
};
```


v
DIM
50
total_utilizados
6
vector_privado
1 3 4 6 8 9 ... 0
0 1 2 3 4 5 49


v.BusquedaBinaria (0, 5, 6)
izda 0 dcha 5 buscado 6 centro 2
vector_privado[2] < 6
v.BusquedaBinaria (3, 5, 6)
izda 3 dcha 5 buscado 6 centro 4
vector_privado[4] > 6
v.BusquedaBinaria (3, 3, 6)
izda 3 dcha 3 buscado 6 centro 3
vector_privado[3] == 6
Devuelve: 3
Devuelve: 3
Devuelve: 3


v
DIM
50
total_utilizados
6
vector_privado
1
3
4
6
8
9
...
0
0
1
2
3
4
5
49


v.BusquedaBinaria (0, 5, 2)
izda
0
dcha
5
buscado
2
centro
2
vector_privado[2] > 2
v.BusquedaBinaria (0, 1, 2)
izda
0
dcha
1
buscado
2
centro
0
izda < dcha
vector_privado[0] < 6
v.BusquedaBinaria (1, 1, 2)
izda
1
dcha
1
buscado
2
centro
1
izda == dcha
vector_privado[1] < 6
v.BusquedaBinaria (2, 1, 2)
izda
2
dcha
1
buscado
2
izda < dcha
Devuelve: -1
Devuelve: -1
Devuelve: -1
Devuelve: -1

Ejercicio. **Plantear la solución recursiva al problema de encontrar el mayor elemento de un vector (entre dos posiciones dadas) y definir un método que la implemente.**

```
class MiVectorEnteros{
private:
   static const int DIM = 50;
   int vector_privado[DIM];
   int total_utilizados;
public:
   .......
   // Prec: 0 < izda , dcha < TotalUtilizados()
   int PosMayor(int izda, int dcha){
      int pos_mayor_anterior;

      if (izda < dcha){
         pos_mayor_anterior = PosMayor(izda + 1, dcha);

         if (vector_privado[pos_mayor_anterior] > vector_privado[izda])
            return pos_mayor_anterior;
         else
            return izda;
      }
      else if (izda == dcha)
         return izda;
      else
         return -1;
   }
};
```

Ejemplo. Invertir un vector (el subvector entre las posiciones `izda` y `dcha`)

```
   Si izda == dcha
      Solución: Construir un vector con esa única componente
   si no
      Calcular la solución al problema de invertir el vector
      entre las posiciones izda + 1 y dcha
      Solución: Añadir la componente izda al final de la solución anterior
```

```
class MiVectorCaracteres{
private:
   static const int DIM = 50;
   char vector_privado[DIM];
   int total_utilizados;
public:
   ......
   // Prec: 0 < izda , dcha < TotalUtilizados()

   MiVectorCaracteres InvierteRecursivo(int izda, int dcha){
      MiVectorCaracteres inverso;

      if (izda < dcha){
         inverso = InvierteRecursivo(izda+1, dcha);
         inverso.Aniade(vector_privado[izda]);
      }
      else if (izda == dcha)
         inverso.Aniade(vector_privado[izda]);

      return inverso;    // Si izda > dcha, devuelve vector vacío.
   }
};
```

VII.3.2. Ordenación con Quicksort (Ampliación)

Ideado por Charles Antony Richard Hoare en 1960.

Este es un método de ordenación que utiliza la recursividad para ir fijando los rangos del vector sobre los que otro método (`partir`) realizará los intercambios necesarios para ir ordenándolo. *Idea*:

- ▷ **Para ordenar alfabéticamente un montón de exámenes, se hacen dos pilas: en una se van echando los que tienen apellidos menores que un *pivote*, por ejemplo, `'M'` y en la otra los mayores que dicho pivote.**
- ▷ **Se vuelve a repetir el proceso con cada montón, pero cambiando el elemento *pivote*. En la segunda iteración recursiva, en el primer montón podría escogerse como pivote `'J'` y en el segundo `'R'`.**
- ▷ **Si no conocemos a priori el rango de valores, el pivote lo elegimos de forma arbitraria (por ejemplo, el primero que cojamos de la pila)**

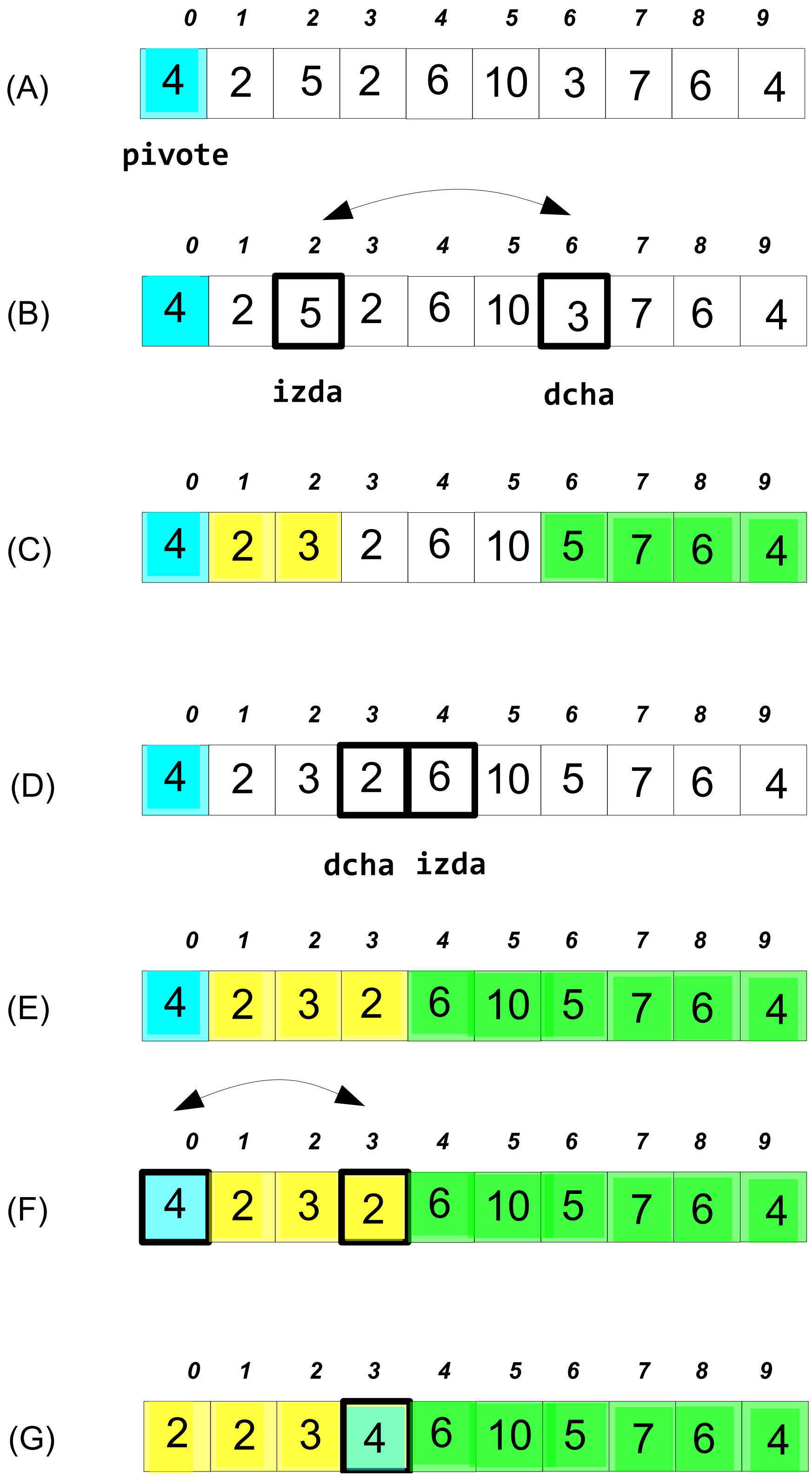
```
class MiVectorEnteros{
private:
   int partir(int primero, int ultimo)           // <- No recursivo
      .......
public:
   void QuickSort(int inicio, int final){        // <- Recursivo
      int pos_pivote;

      if (inicio < final) {
         pos_pivote = partir (inicio, final);
         QuickSort (inicio, pos_pivote - 1);     // Ordena primera mitad
         QuickSort (pos_pivote + 1, final);      // Ordena segunda mitad
      }
   }
```

***Líneas básicas del algoritmo de la función* `partir`:**

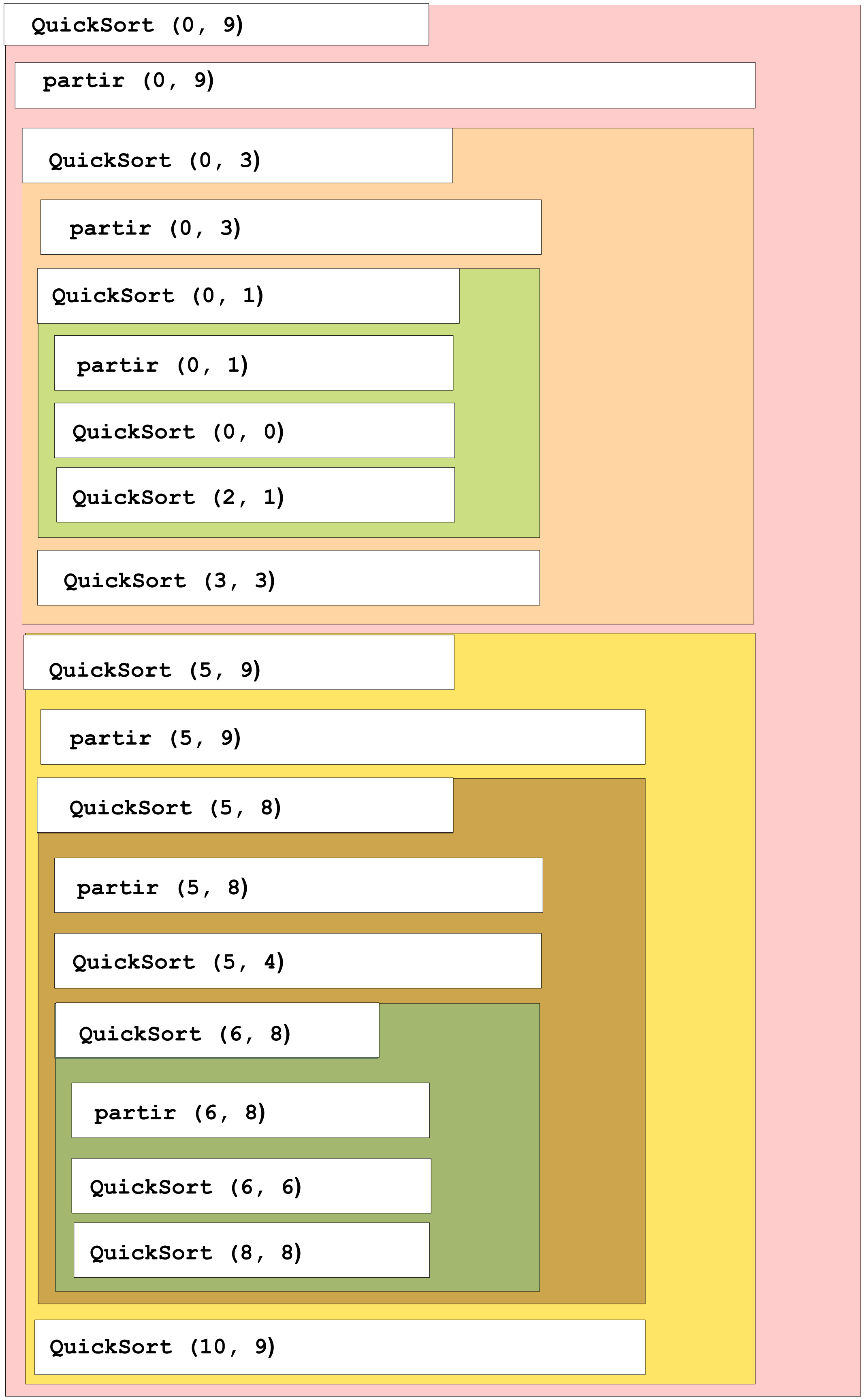
1. **Tomar un elemento arbitrario del vector:** `pivote`
2. **Recorrer el vector de izquierda a derecha hasta encontrar un elemento situado en una posición** `izda` **tal que** `v[izda] > pivote`
3. **Recorrer el vector de derecha a izquierda hasta encontrar otro elemento situado en una posición** `dcha` **tal que** `v[dcha] < pivote`
4. **Intercambiar los elementos de las casillas** `izda` **y** `dcha`

   Una vez hecho el intercambio tendremos que:

   `v[izda] < pivote < v[dcha]`
5. **Repetir hasta que los dos procesos de recorrido se encuentren (**`izda > dcha`**).**
6. **Colocar el pivote en el sitio que le corresponde.**
7. **Una vez hecho lo anterior, el vector está particionado en dos zonas delimitadas por el pivote. El pivote está ya colocado correctamente en su sitio.**


0 1 2 3 4 5 6 7 8 9
(A) 4 2 5 2 6 10 3 7 6 4
pivote
0 1 2 3 4 5 6 7 8 9
(B) 4 2 5 2 6 10 3 7 6 4
izda
dcha
0 1 2 3 4 5 6 7 8 9
(C) 4 2 3 2 6 10 5 7 6 4
0 1 2 3 4 5 6 7 8 9
(D) 4 2 3 2 6 10 5 7 6 4
dcha
izda
0 1 2 3 4 5 6 7 8 9
(E) 4 2 3 2 6 10 5 7 6 4
0 1 2 3 4 5 6 7 8 9
(F) 4 2 3 2 6 10 5 7 6 4
0 1 2 3 4 5 6 7 8 9
(G) 2 2 3 4 6 10 5 7 6 4

```
int partir (int primero, int ultimo){
   int intercambia;
   int izda, dcha;
   int valor_pivote = vector_privado[primero];

   izda = primero + 1;      // izda avanza hacia delante
   dcha = ultimo;           // dcha retrocede hacia atrás

   while (izda <= dcha){
      while (izda <= dcha  &&  vector_privado[izda] <= valor_pivote)
         izda++;

      while (izda <= dcha  &&  vector_privado[dcha] >= valor_pivote)
         dcha--;

      if (izda < dcha) {
         intercambia = vector_privado[izda];
         vector_privado[izda] = vector_privado[dcha];
         vector_privado[dcha] = intercambia;
         dcha--;
         izda++;
      }
   }
   intercambia = vector_privado[primero];
   vector_privado[primero] = vector_privado[dcha];
   vector_privado[dcha] = intercambia;

   return dcha;
}
```


QuickSort (0, 9)
partir (0, 9)
QuickSort (0, 3)
partir (0, 3)
QuickSort (0, 1)
partir (0, 1)
QuickSort (0, 0)
QuickSort (2, 1)
QuickSort (3, 3)
QuickSort (5, 9)
partir (5, 9)
QuickSort (5, 8)
partir (5, 8)
QuickSort (5, 4)
QuickSort (6, 8)
partir (6, 8)
QuickSort (6, 6)
QuickSort (8, 8)
QuickSort (10, 9)

Ampliación:

▷ Quicksort es uno de los mejores algoritmos de ordenación. Se analizará su eficiencia en el segundo cuatrimestre.

▷ Quicksort puede paralelizarse fácilmente. Cada CPU se encargaría de la resolución de una llamada recursiva. Al final basta juntar los subvectores ordenados. Esto es posible porque el pivote calculado en cada momento, es colocado en el lugar que le corresponde.

▷ La elección del pivote es muy importante. Las mejoras de Quicksort pasan por desarrollar mecanismos rápidos de elección de *buenos* pivotes (elementos que dejen más o menos el mismo número de elementos a la izquierda y derecha)

▷ Algunos applets de demostración.

Para tener una idea *global*:

`http://maven.smith.edu/~thiebaut/java/sort/`

Ejecución paso a paso:

`http://pages.stern.nyu.edu/~panos/java/Quicksort/`

`http://math.hws.edu/TMCM/java/xSortLab/` (en este applet, `partir` recorre primero la derecha y luego la izquierda)

VII.4. Recursividad versus iteración

Desventajas de la recursividad:

▷ **La carga computacional (tiempo-espacio) asociada a una llamada a una función y el retorno a la función que hace la llamada.**

▷ **Algunas soluciones recursivas pueden hacer que la solución para un determinado tamaño del problema se calcule varias veces.**

Ventajas de la recursividad:

▷ **La solución recursiva suele ser concisa, legible y elegante.**

Tal es el caso, por ejemplo, del recorrido de estructuras *complejas* como árboles, listas, grafos, etc. Se verá en el segundo cuatrimestre.

Habrá que analizar en cada caso la conveniencia o no de usar recursividad.

Recursividad de cola

Se dice que una función tiene *recursividad de cola (tail recursion)* si sólo se produce una llamada recursiva y no se ejecuta posteriormente ningún código. Estas funciones pueden traducirse fácilmente a un algoritmo no recursivo, más eficiente. De hecho, muchos compiladores realizan automáticamente esta conversión.

Los métodos `BusquedaBinaria` **y** `PrimeraOcurrenciaEntre` **tienen recursividad de cola. En el resto de ejemplos vistos, siempre hay que hacer algo más después de realizar la llamada recursiva. Por ejemplo:**

```
// Prec: exponente >= 0
double Potencia(double base, int exponente){
   if (exponente == 0)
      return 1.0;
   else
      return base * Potencia(base, exponente-1);
}
```

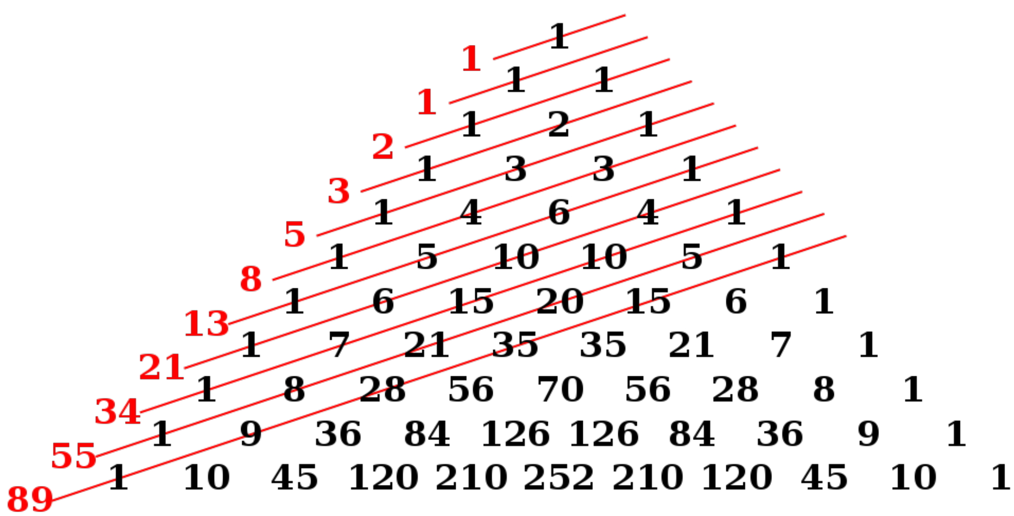
En cualquier caso, muchos métodos/funciones sin recursividad de cola también pueden traducirse de forma fácil como un algoritmo iterativo.

Repetición de cómputos

Los algoritmos *interesantes* serán los que realicen más de una llamada recursiva (como Quicksort), pero debemos evitar la repetición de cómputos.

Ejemplo. **Sucesión de Fibonacci.**

$$Fib(0) = Fib(1) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

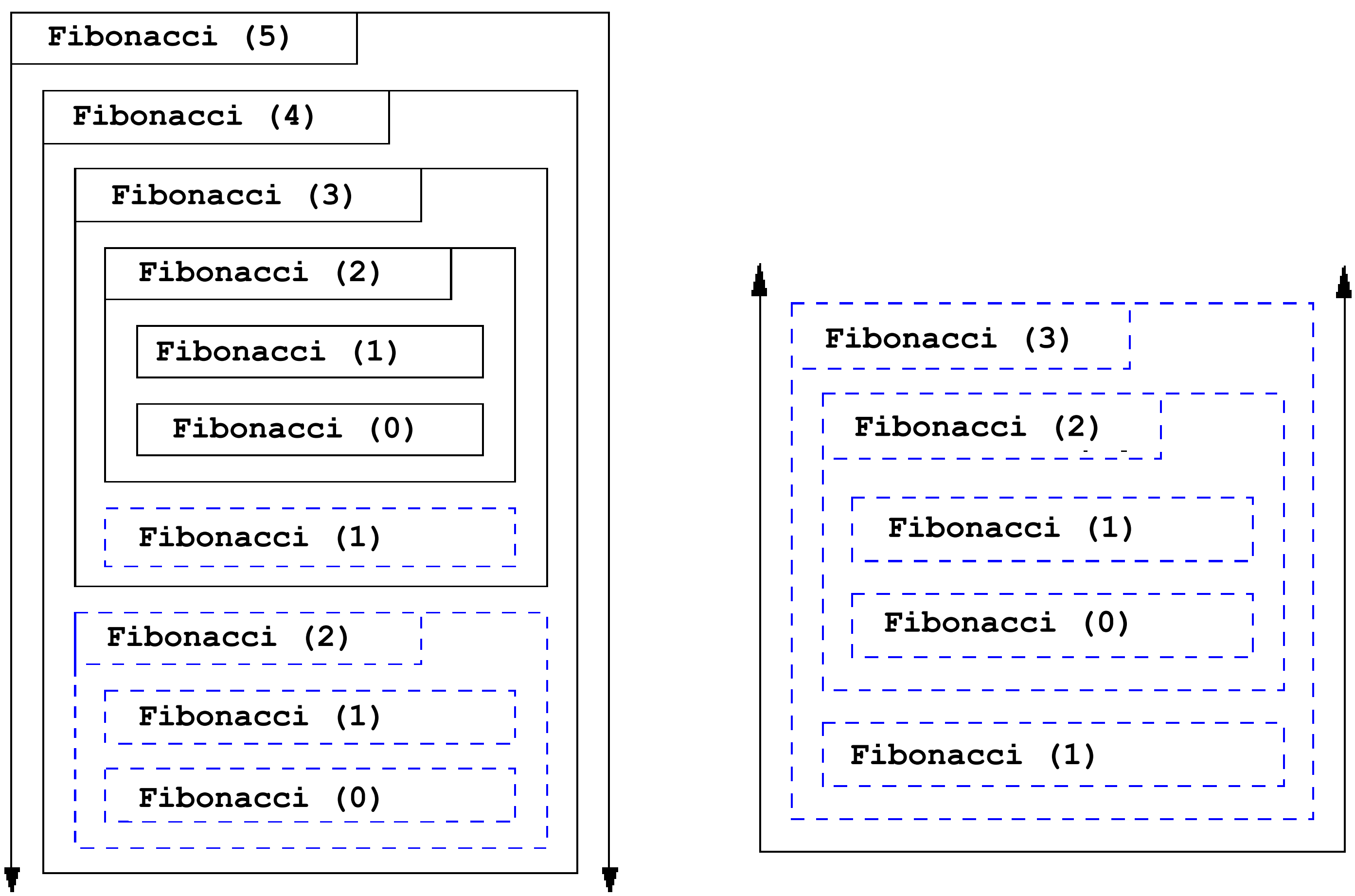


Ampliación:

La sucesión de Fibonacci tiene muchas aplicaciones. Por ejemplo Fibonacci(n+2) da como resultado el número de subconjuntos de $\{1, \cdots, n\}$ que no contienen enteros consecutivos. En la naturaleza, están presentes en la fórmula que determina cómo se disponen las hojas en los tallos. Hay una revista matemática dedicada exclusivamente a resultados sobre esta sucesión. Para más información, consultad wikipedia (en inglés) http://en.wikipedia.org/wiki/Fibonacci_number y la web de la Enciclopedia de sucesiones enteras http://oeis.org/A000045.

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

Cálculo de `Fibonacci(5)`:



La solución recursiva de Fibonacci repite muchos cómputos, por lo que es muy ineficiente.

La solución iterativa es sencilla de implementar y mucho más rápida:

```
long long Fibonacci (int n){
   long long anterior = 1, pre_anterior = 1;
   long long actual;

   if (n == 0 || n == 1)
      actual = 1;
   else
      for (int i = 2; i <= n; i++) {
         actual = anterior + pre_anterior;
         pre_anterior = anterior;
         anterior = actual;
      }

   return actual;
}
```

Ampliación:

> **Una forma alternativa de evitar el problema de los cómputos repetidos en recursividad, lo proporcionan las técnicas de la *programación dinámica (dynamic programming)* que, a grosso modo, van guardando adecuadamente los resultados obtenidos previamente.**

Caso base demasiado pequeño

Muchos problemas recursivos tienen como caso base un problema de un tamaño reducido. En ocasiones es *excesivamente* pequeño.

Para resolverlo, se suele aplicar un algoritmo iterativo cuando se ha llegado a un caso base suficientemente pequeño.

Ejemplo. **Búsqueda binaria.**

Aplicamos el algoritmo recursivo en general y el iterativo cuando el tamaño del subvector en el que estoy buscando es pequeño.

```
   Recorrer las componentes con dos apuntadores izda y dcha
      Si el número de elementos entre izda y dcha es pequeño
         Buscar el valor con un algoritmo no recursivo
      en otro caso
         Colocarse en el centro y comprobar si es el valor buscado
         Si no lo es
            Buscar a la izda o a la derecha de centro
            (realizando la llamada recursiva)
            según sea el elemento buscado menor o mayor que v[centro]
```

```
class MiVectorEnteros{
   .......
   int PrimeraOcurrenciaEntre (int pos_izda, int pos_dcha, char buscado){
      int i = pos_izda;
      bool encontrado = false;

      while (i <= pos_dcha  &&  !encontrado)
         if (vector_privado[i] == buscado)
            encontrado = true;
         else
            i++;
```

```
      if (encontrado)
         return i;
      else
         return -1;
   }
   int BusquedaBinaria(int izda, int dcha, int buscado){
      const int umbral = 5;
      int centro;

      if (dcha - izda < umbral)
         return PrimeraOcurrenciaEntre(izda, dcha, buscado);
      else{
         centro = (izda + dcha) / 2;

         if (vector_privado[centro] == buscado)
            return centro;
         else
            if (vector_privado[centro] > buscado)
               return BusquedaBinaria(izda, centro-1, buscado);
            else
               return BusquedaBinaria(centro + 1, dcha, buscado);
      }
   }
};
```

***Nota*. El caso base `(izda > dcha)` de la función ya no es necesario.**

Ejemplo. **(Ampliación) Mejoramos QuickSort de forma análoga:**

```
class MiVectorEnteros{
private:
   int partir(int primero, int ultimo)
      .......
public:
   .......
   void Ordena_por_Seleccion_entre(int primero, int ultimo){
      int pos_min;

      for (int izda=primero ; izda<ultimo ; izda++){
         pos_min = PosicionMinimoEntre (izda, ultimo);
         IntercambiaComponentes_en_Posiciones (izda, pos_min);
      }
   }
   void QuickSort(int inicio, int final){
      const int umbral = 5;
      int pos_pivote;

      if (final - inicio < umbral)
         Ordena_por_Seleccion_entre(inicio, final);
      else{
         if (inicio < final) {
            pos_pivote = partir (inicio, final);
            QuickSort (inicio, pos_pivote - 1);    // Ordena primera mitad
            QuickSort (pos_pivote + 1, final);     // Ordena segunda mitad
         }
      }
   }
};
```

***Nota*. Para Quicksort, lo usual es usar umbrales entre 4 y 15.**

Problemas complejos con una solución recursiva sencilla

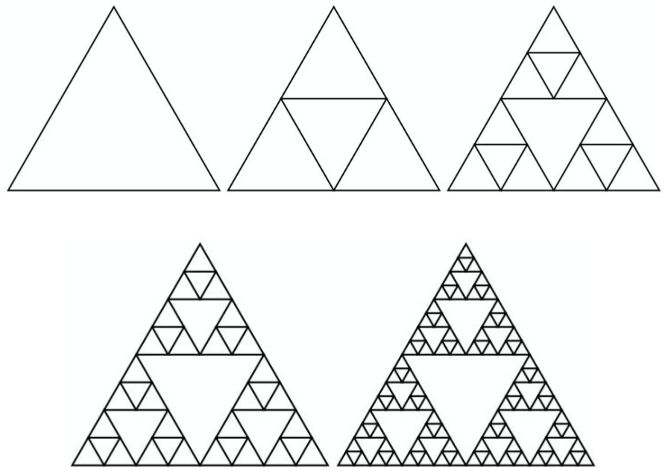
Algunos problemas se resuelven directamente con recursividad. Normalmente, también es posible resolverlos iterativamente, pero puede resultar más complejo.

Ejemplo. **Definición de *fractal (fractal)* de la Wikipedia:**

Un fractal es un objeto semigeométrico cuya estructura básica, fragmentada o irregular, se repite a diferentes escalas. A un objeto geométrico fractal se le atribuyen las siguientes características:

- ***Es demasiado irregular para ser descrito en términos geométricos tradicionales.***
- ***Es autosimilar.- Su forma es hecha de copias más pequeñas de la misma figura. Las copias son similares al todo: misma forma pero diferente tamaño.***
- ...
- ***Se define mediante un simple algoritmo recursivo.***

Ejemplo. Triángulos de Sierpinski

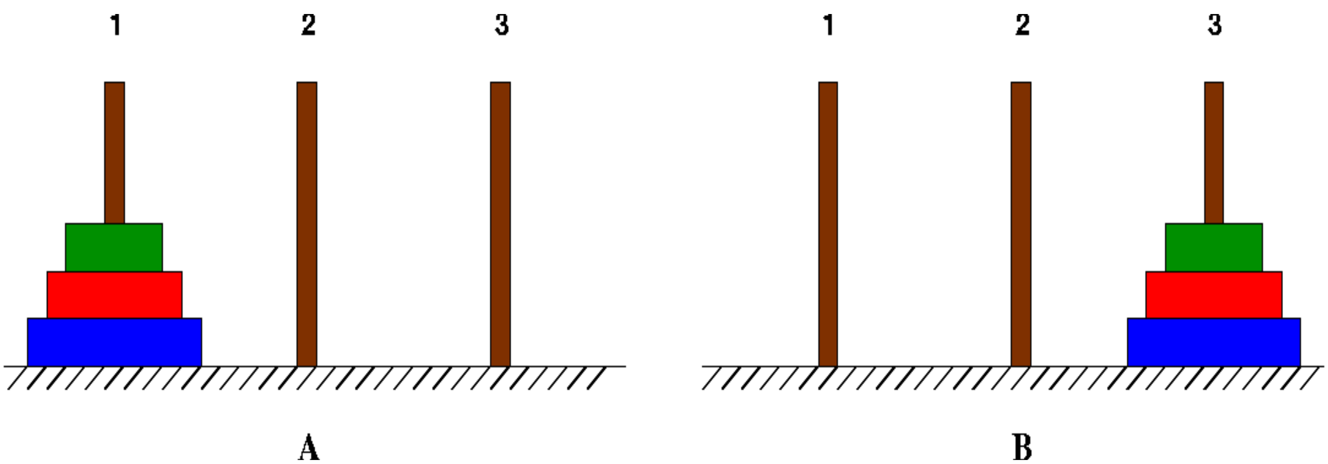


```
class Sierpinski{
public:
   void Triangulo(Punto2D A, Punto2D B, Punto2D C, int grado){
      PintorFiguras pintor;

      if (grado == 0)
         pintor.DibujaTriangulo (A, B, C);
      else{
         SegmentoDirigido segmentoAB(A, B);
         SegmentoDirigido segmentoBC(B, C);
         SegmentoDirigido segmentoCA(C, A);

         Punto2D MedioAB (segmentoAB.Medio());
         Punto2D MedioBC (segmentoBC.Medio());
         Punto2D MedioCA (segmentoCA.Medio());

         Triangulo (A, MedioAB, MedioCA, grado - 1);
         Triangulo (MedioAB, B, MedioBC, grado - 1);
         Triangulo (MedioCA, MedioBC, C, grado - 1);
      }
   }
};
```

Ejemplo. **Torres de Hanoi. Edouard Lucas, 1883.**

Se trata de mover un conjunto de discos apilados en un poste a otro poste, con las condiciones siguientes:

- ▷ **Se tienen 3 postes (hay versiones para trabajar con más postes)**
- ▷ **Sólo se puede mover un disco en cada paso. Además, debe ser el disco que está en la parte superior de la pila.**
- ▷ **Los discos en cualquier poste siempre deben formar una pirámide, es decir, colocados de mayor a menor.**

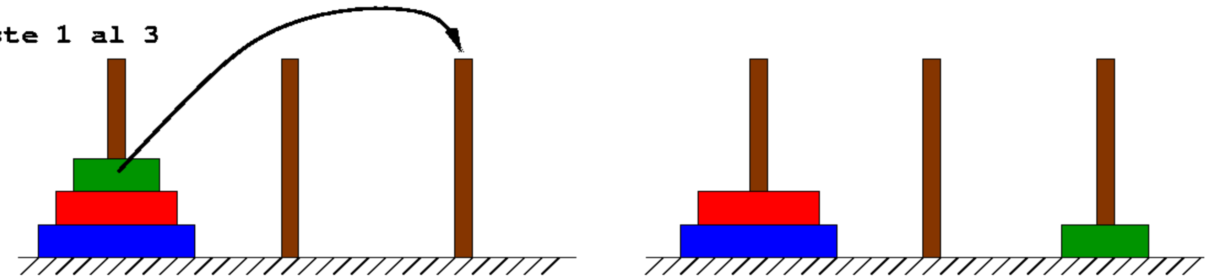


Etiquetando los 3 postes como `inicial`, `intermedio`, `final`, **el algoritmo recursivo que lo resuelve sería:**

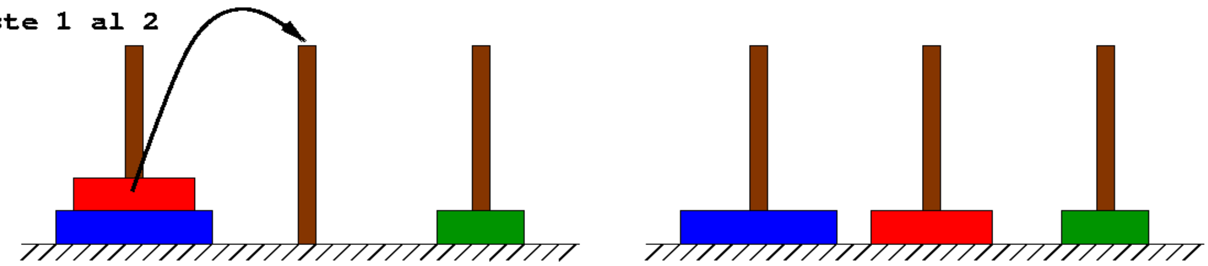
```
Mover n-1 discos desde inicial hasta temporal usando final como intermedio
Mover el disco que queda en inicial hasta final
Mover n-1 discos desde intermedio hasta final usando inicial como intermedio
```

```
#include <iostream>
using namespace std;

void Hanoi (int num_discos, int inicial, int intermedio, int final){
   if (num_discos > 0){
      Hanoi (num_discos-1, inicial, final, intermedio);
      cout <<"\nDel poste " << inicial << " al " << final;
      Hanoi (num_discos-1, intermedio, inicial, final);
   }
}
int main(){
   int n; // Número de discos a mover

   cout << "Número de discos: ";
   cin >> n;

   Hanoi (n, 1, 2, 3); // mover "n" discos del 1 al 3,
                       // usando el 2 como temporal.
}
```

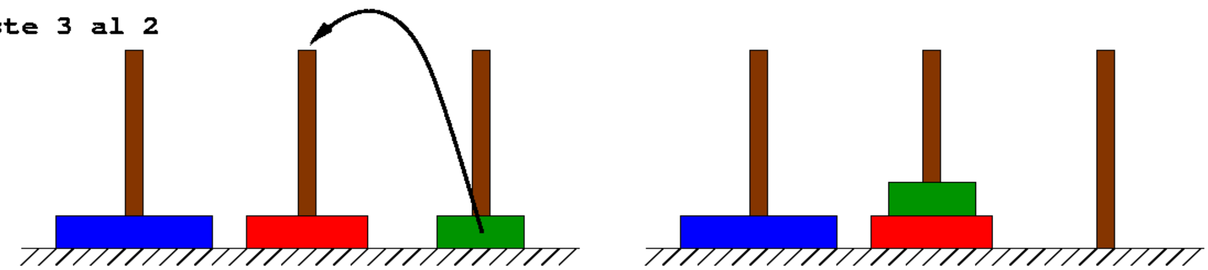
Del poste 1 al 3



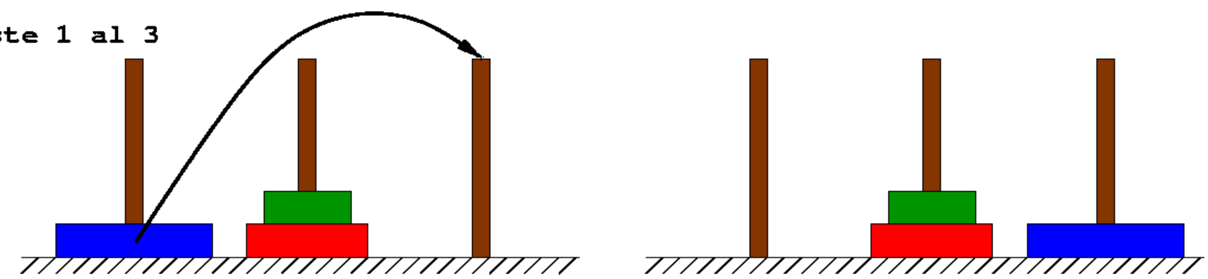
Del poste 1 al 2

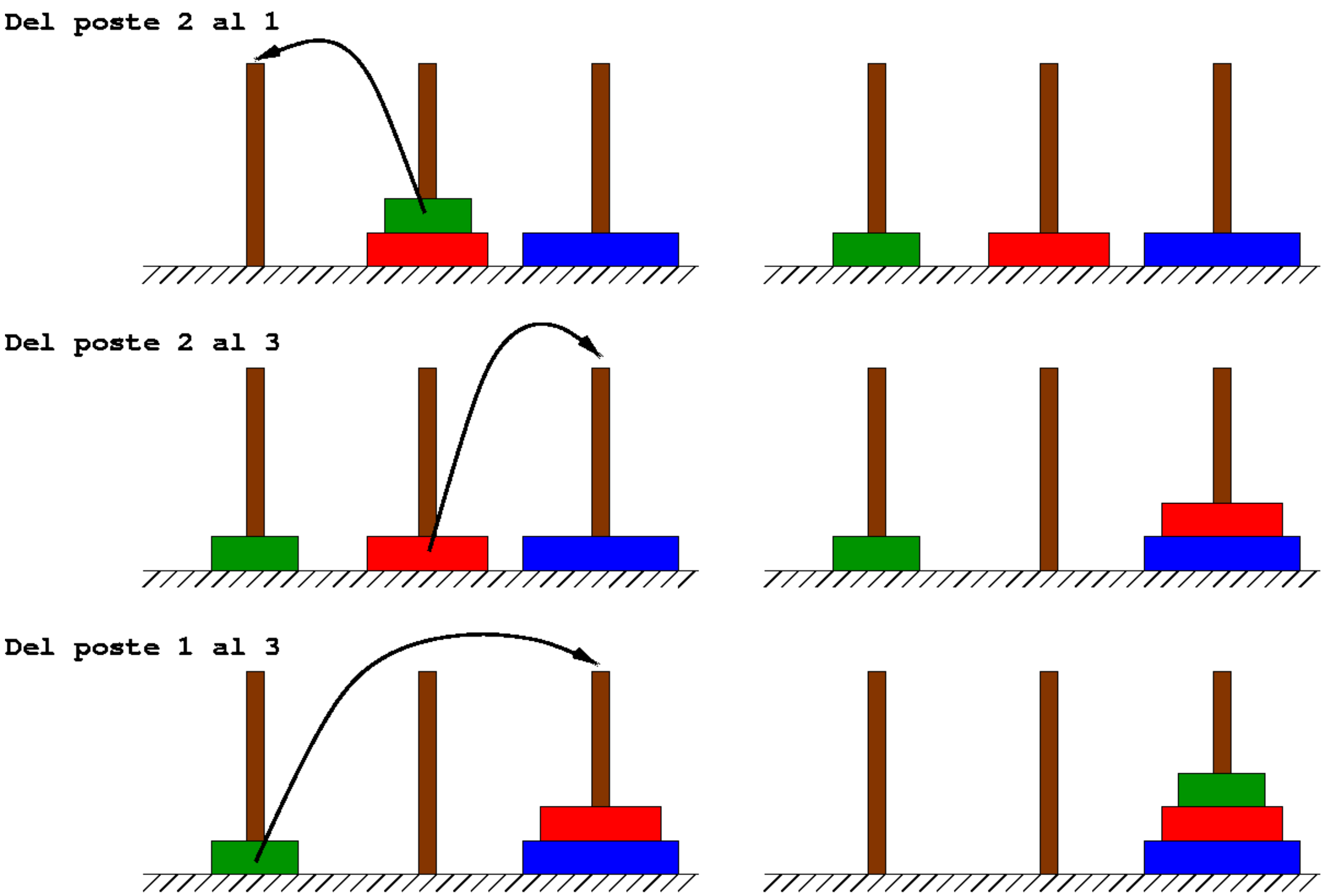


Del poste 3 al 2



Del poste 1 al 3





Algunas curiosidades:

- ▷ **Applet:** http://www.mazeworks.com/hanoi/index.htm
- ▷ **Número de movimientos: 2^k, siendo $k =$ número de discos. Para 64 discos, moviendo 1 disco por segundo, se tardaría algo menos de 600 mil millones de años (el Universo tiene unos 20 mil millones)**
- ▷ **También existe una versión iterativa. No son tres líneas, pero no es demasiado complicada. En cualquier caso, el número de movimientos es el mismo.**