Metodología de la Programación II

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.

3.1 Concepto de Recursividad

La recursividad constituye una de las herramientas más potentes en programación. Es un concepto matemático conocido. Por ejemplo,

- Definición recursiva

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

- Demostración por inducción: demostrar para un caso base y después para un tamaño `n`, considerando que está demostrado para valores menores que `n`.

> Una función que se llama a sí misma se denomina **recursiva**

Podemos usar recursividad si la solución de un problema está expresada en función de si misma, aunque de menor tamaño y conocemos la solución no-recursiva para un determinado caso.

- Ventajas: No es necesario definir la secuencia de pasos exacta para resolver el problema.

- Desventajas: Podría ser menos eficiente.

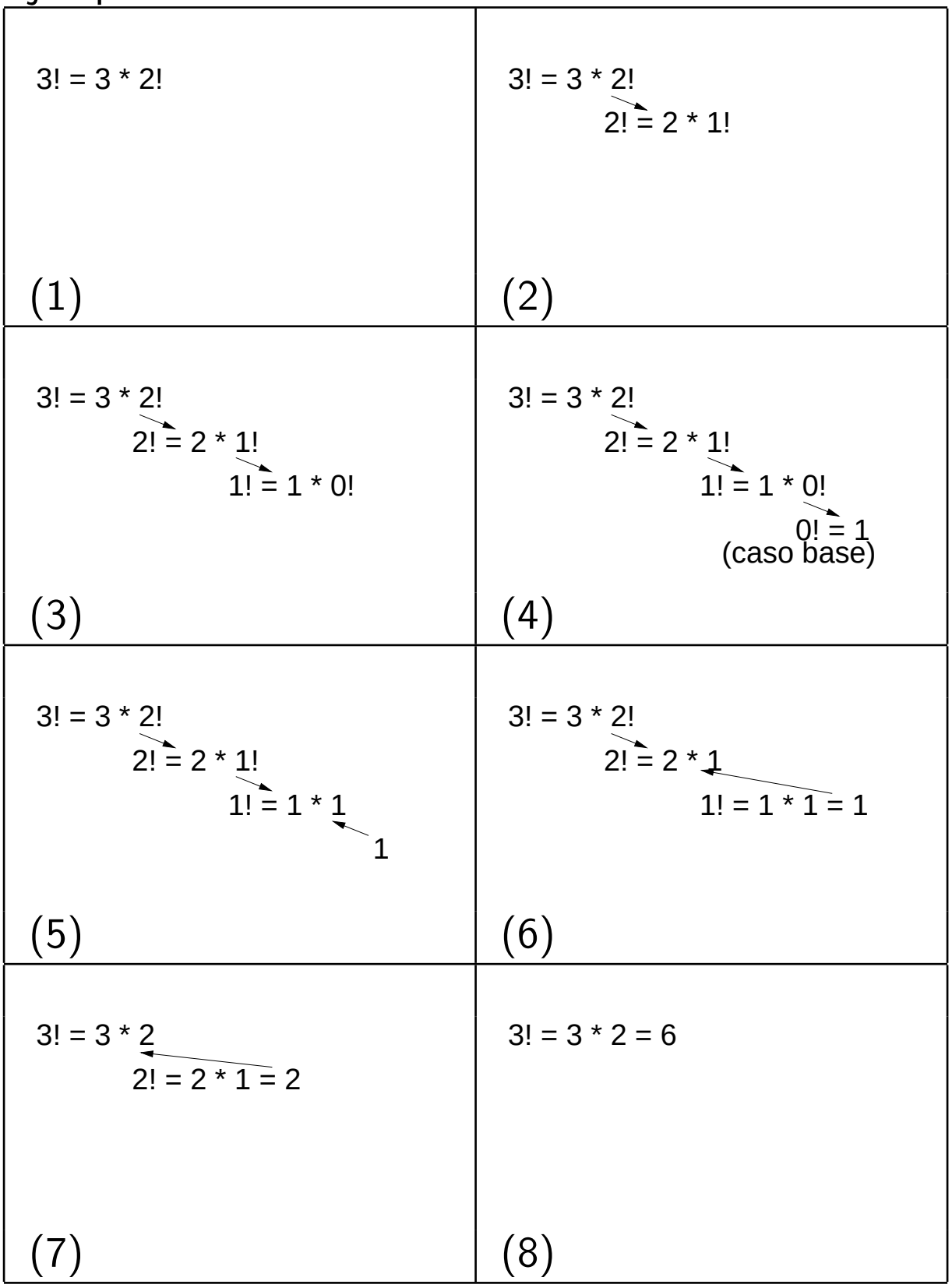
Para que una definición recursiva esté completamente identificada es necesario tener un $caso\ base$ que no se calcule utilizando casos anteriores y que la división del problema converja a ese caso base.

$$0! = 1$$

Ejemplo:

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

Ejemplo: Cálculo del factorial con n=3.



3.2 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. .

Tendremos que diseñar: casos base, casos generales y la solución en términos de ellos.

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

  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.

- Casos generales: Si el problema es suficientemente complejo, la solución se expresa, de forma recursiva, como la unión de

1. La solución de uno o más subproblemas (de igual naturaleza pero menor tamaño).
2. Un conjunto de pasos adicionales. Estos pasos junto con las soluciones a los subproblemas componen la solución al problema general que queremos resolver.

> Los casos generales siempre deben avanzar hacia un caso base. Es decir, la llamada recursiva se hace a un subproblema más pequeño y, en última instancia, los casos generales alcanzarán un caso base.

Ejemplo:

```
// Solucion no estructurada
int factorial (int n) {
  if (n==0)          //Caso base
    return 1;
  else               //Caso general
    return n*factorial(n-1);
}

// Solucion estructurada
int factorial (int n) {
  int resultado;
  if (n==0)          //Caso base
    resultado = 1;
  else               //Caso general
    resultado = n*factorial(n-1);
  return resultado;
}
```

3.3 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 módulo 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.

1. Dentro de factorial, cada llamada

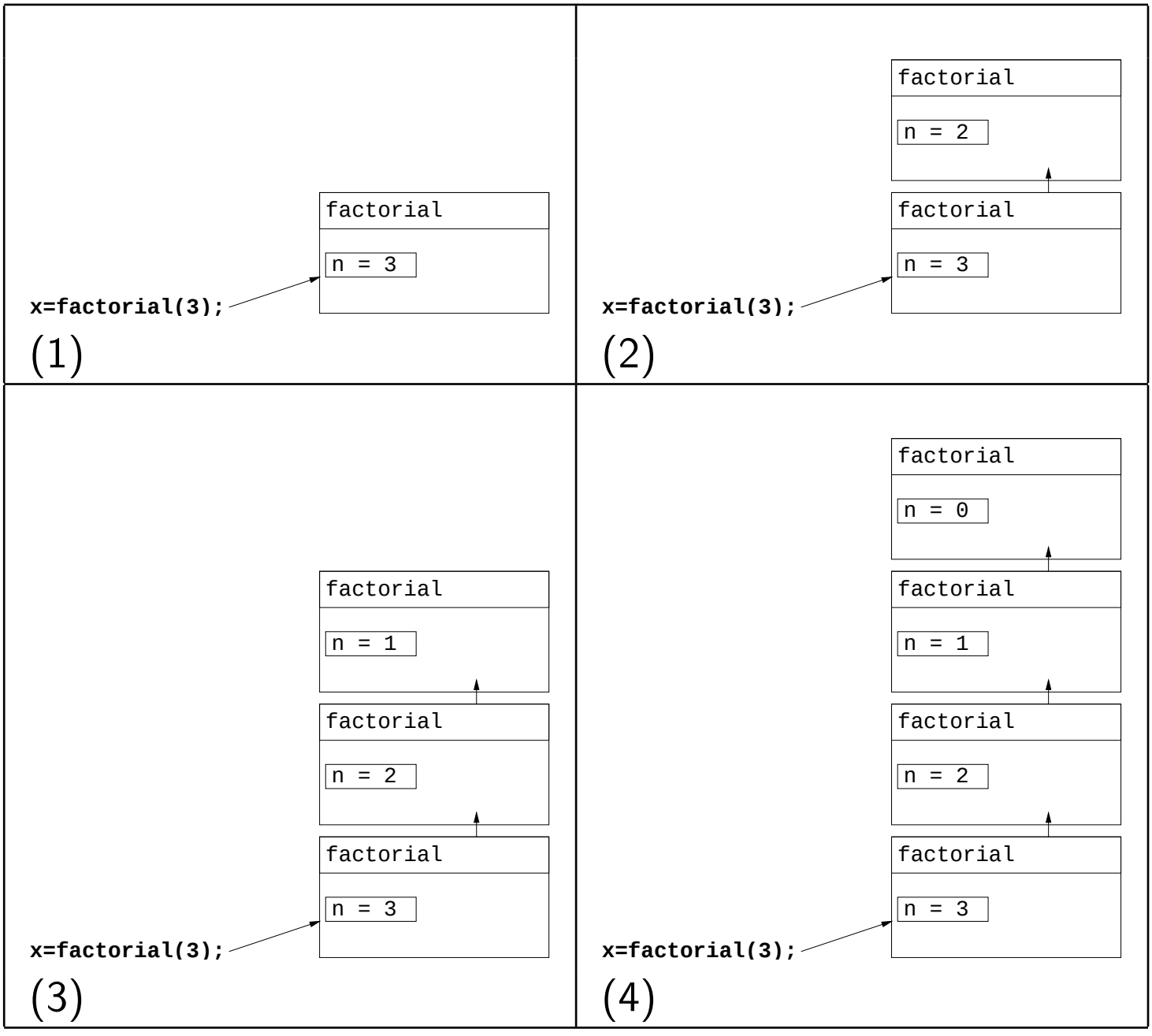
   ```
   return n*factorial(n-1);
   ```

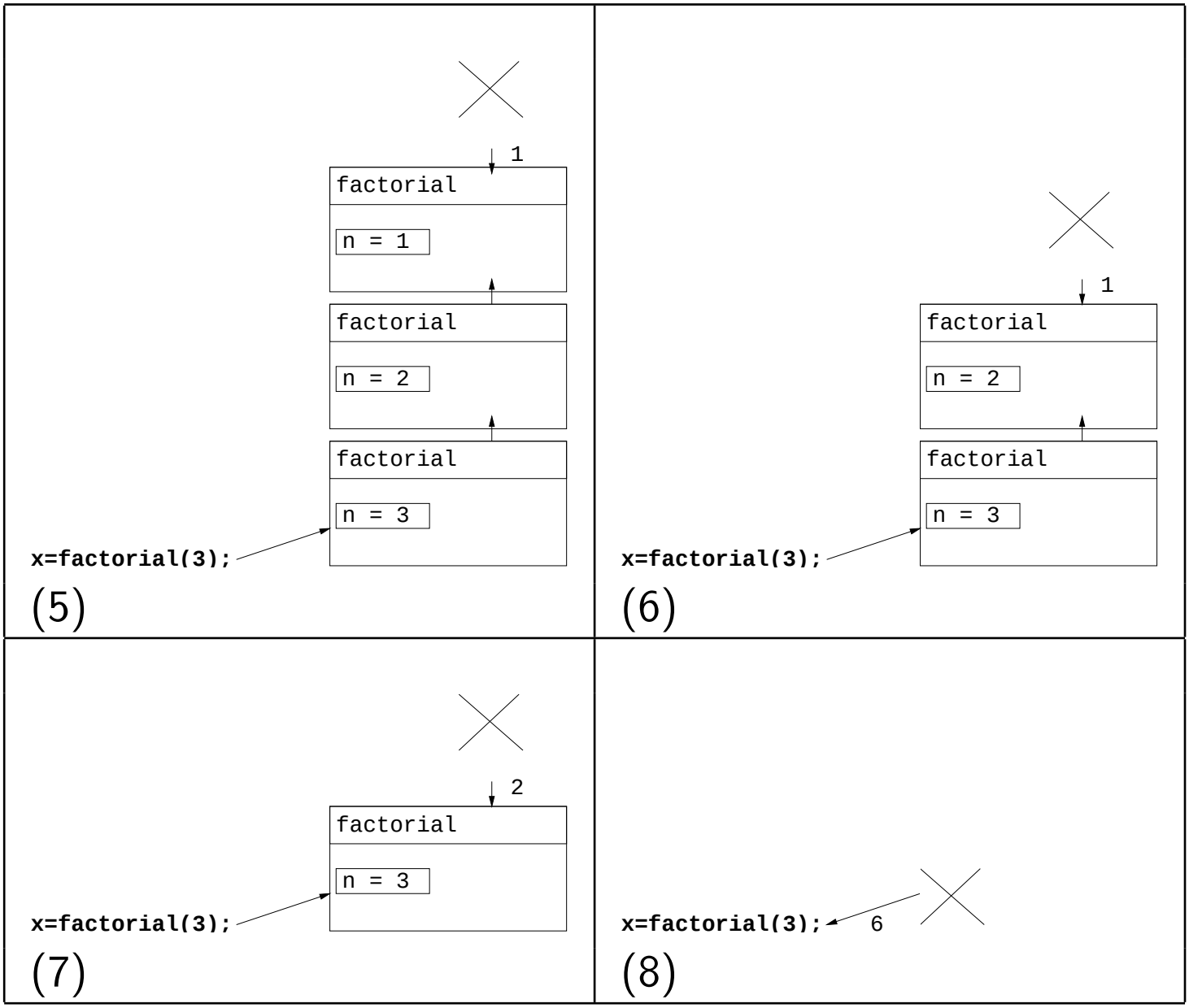
   genera una nueva zona de memoria en la pila, siendo `n-1` el correspondiente parámetro actual para esta zona de memoria y queda pendiente la evaluación de la expresión y la ejecución del `return`.

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.

Llamada: `x = factorial(3);`

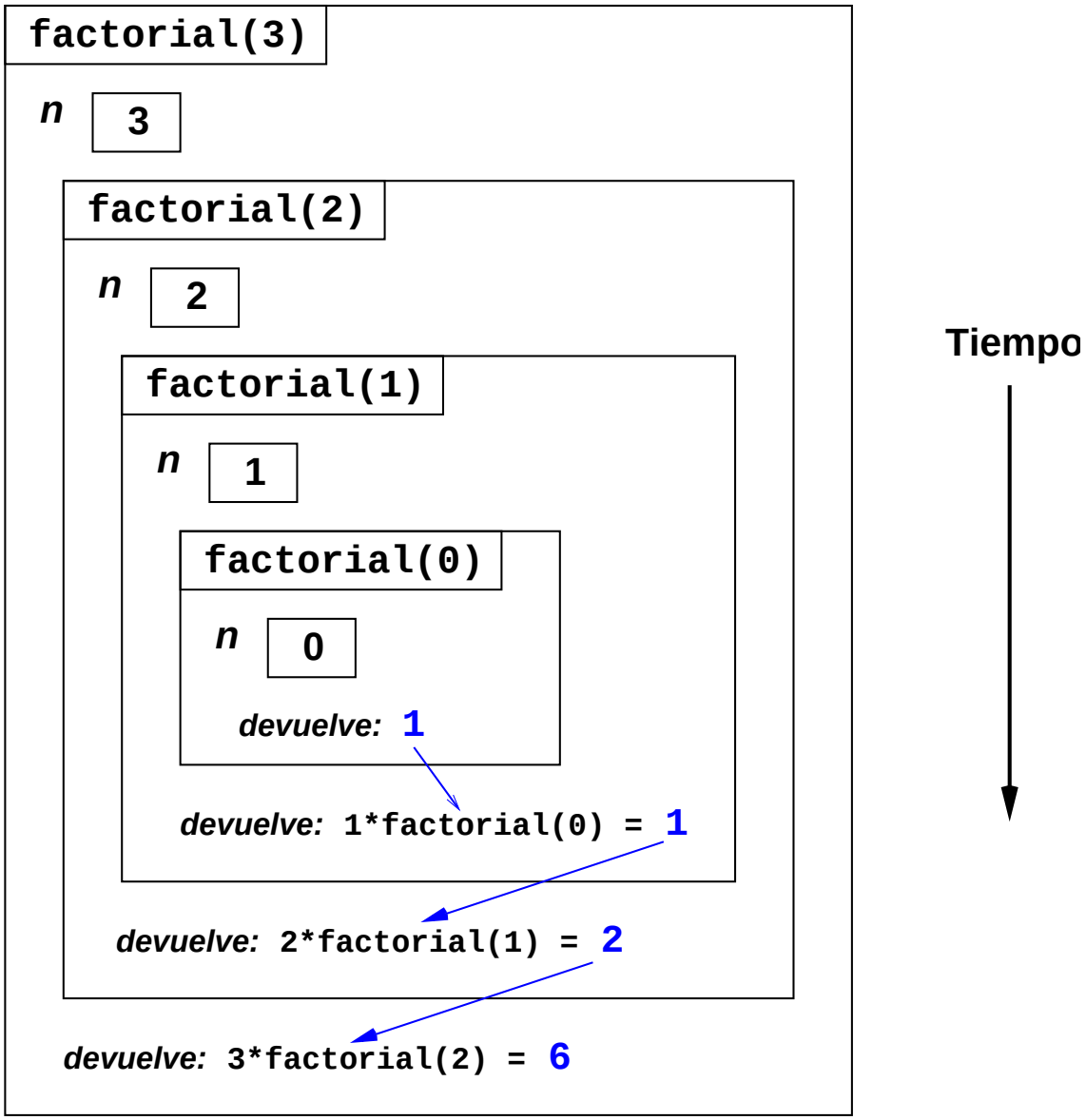



1
factorial
n = 1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(5)
1
factorial
n = 2
factorial
n = 3
x=factorial(3);
(6)
2
factorial
n = 3
x=factorial(3);
(7)
x=factorial(3);
6
(8)

3.4 Traza de algoritmos recursivos

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

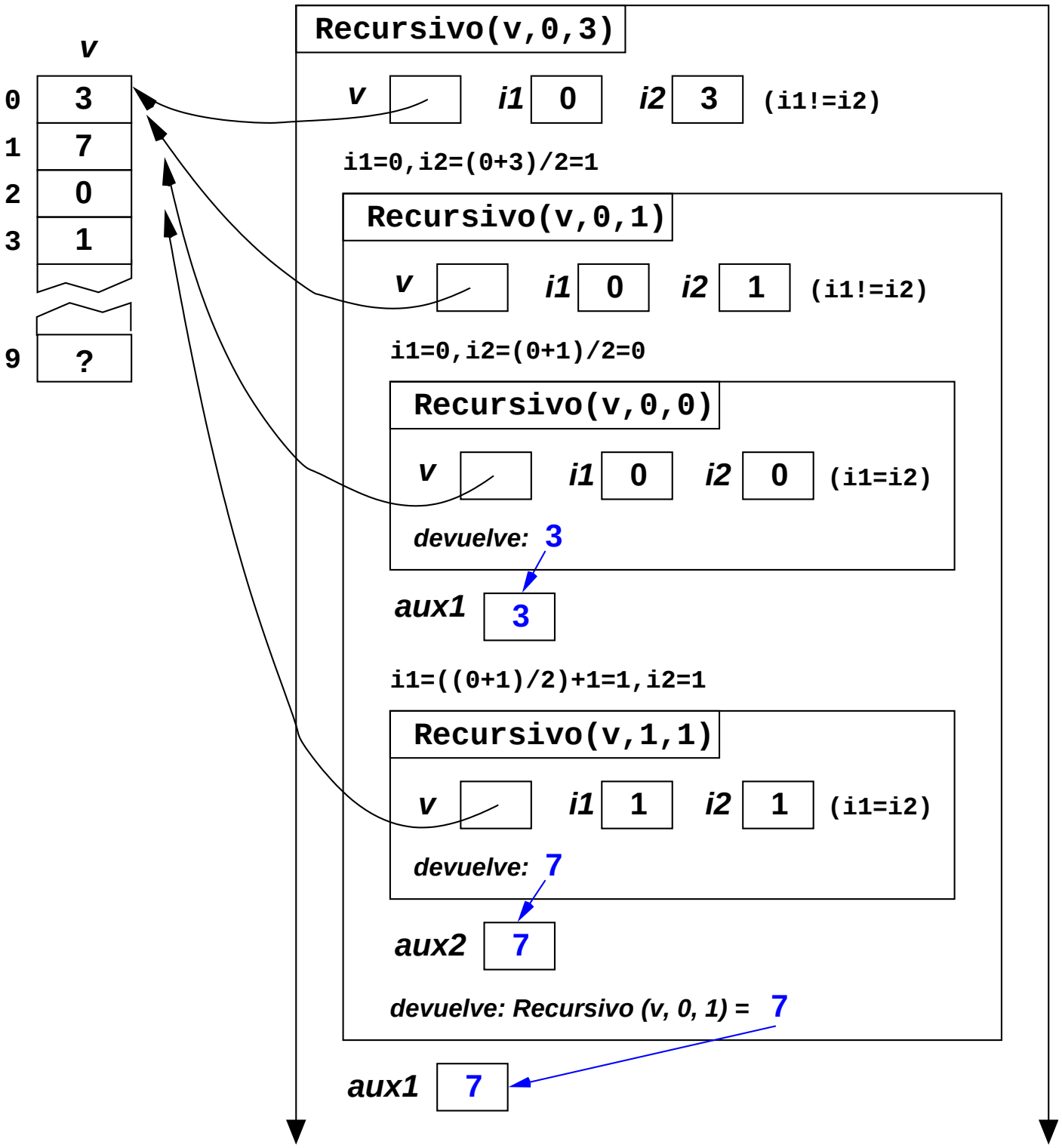
Llamada: `factorial(3)`

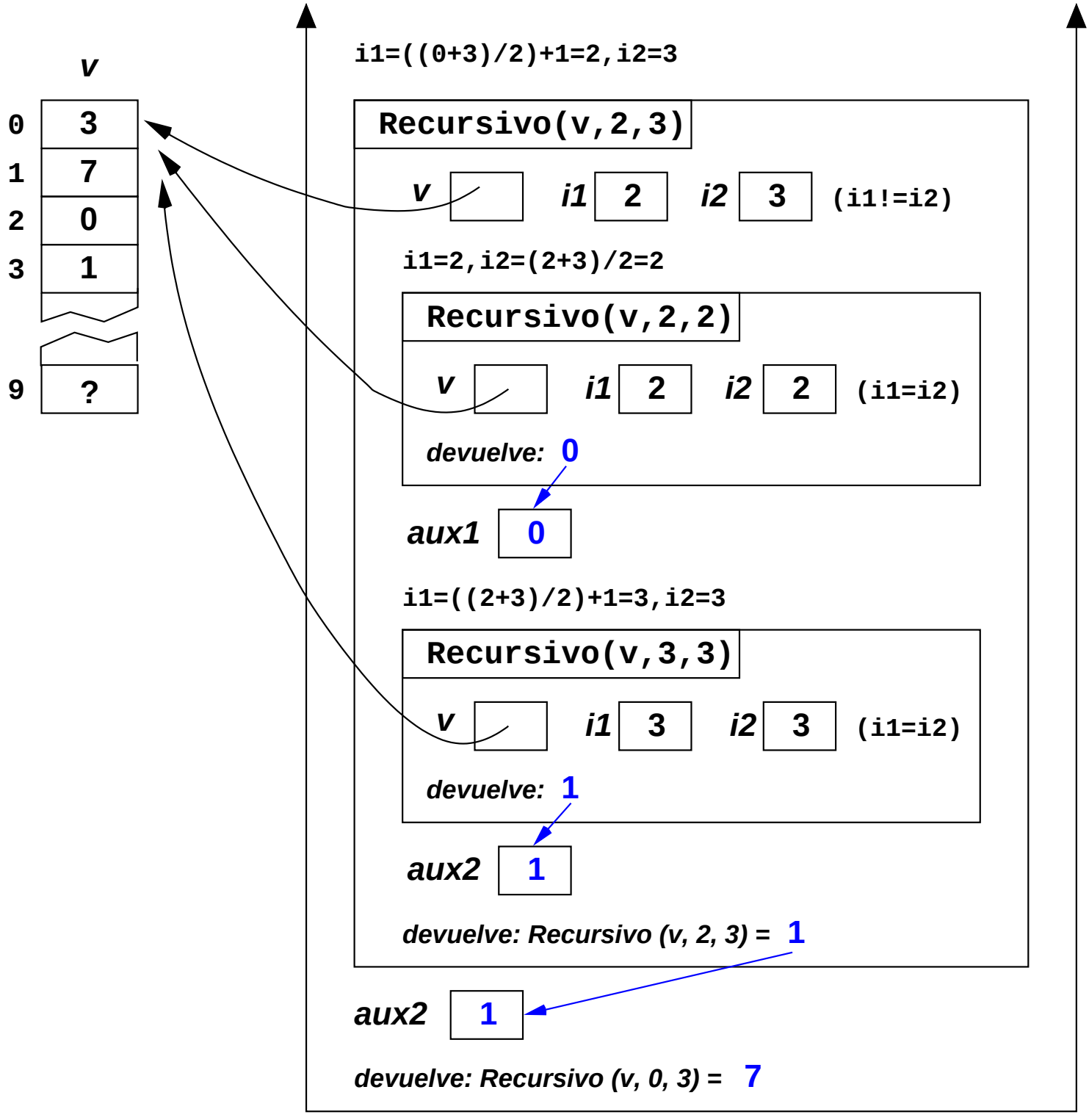


Ejemplo: Averigua qué hace este algoritmo

```
int Recursivo (int *V, int i1, int i2) {
   int aux1, aux2;

   if (i1==i2)  //Caso base
      return V[i1];
   else {       //Caso general
      aux1 = Recursivo(V, i1, (i1+i2)/2);
      aux2 = Recursivo(V, ((i1+i2)/2)+1, i2);
      if (aux1>aux2)
         return aux1;
      else
         return aux2;
}
```

llamada: `Recursivo(V,0,3)`, con `V es [3,7,0,1]`




v
0
3
1
7
2
0
3
1
9
?
i1=((0+3)/2)+1=2,i2=3
Recursivo(v,2,3)
v
i1
2
i2
3
(i1!=i2)
i1=2,i2=(2+3)/2=2
Recursivo(v,2,2)
v
i1
2
i2
2
(i1=i2)
devuelve: 0
aux1
0
i1=((2+3)/2)+1=3,i2=3
Recursivo(v,3,3)
v
i1
3
i2
3
(i1=i2)
devuelve: 1
aux2
1
devuelve: Recursivo (v, 2, 3) = 1
aux2
1
devuelve: Recursivo (v, 0, 3) = 7

3.5 Ejemplos de funciones recursivas

1. Cálculo de la potencia

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \cdot x^{n-1} & \text{si } n > 0 \end{cases}$$

```
int potencia(int base, int expo){
  if (expo==0)
    return 1;
  else
    return base * potencia(base,expo-1);
}
```

2. La suma de forma recursiva

$$suma(a,b) = \begin{cases} a & \text{si } b = 0 \\ 1 + suma(a, b-1) & \text{si } b > 0 \end{cases}$$

```
int suma(int a, int b){
  if (b==0)
    return a;
  else
    return 1+suma(a,b-1);
}
```

3. El producto de forma recursiva

$$producto(a,b) = \begin{cases} 0 & \text{si } b = 0 \\ a + producto(a, b-1) & \text{si } b > 0 \end{cases}$$

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

4. Suma recursiva de los elementos de un vector

$$sumaV(V,n) = \begin{cases} V[0] & \text{si } n = 0 \\ V[n] + sumaV(V, n-1) & \text{si } n > 0 \end{cases}$$

```
int SumaV (int *V, int n){
  if (n==0)
    return V[0];
  else
    return V[n]+sumaV(V,n-1);
}
```

5. Buscar el máximo de un vector (I)

$$Mayor1(V,n) = \begin{cases} V[0] & \text{si } n = 0 \\ V[n] \text{ ó } Mayor1(V, n-1) & \text{si } n > 0 \end{cases}$$

```
int Mayor1 (int *V, int n){
  int aux;

  if (n==0)
    return V[0];
  else {
    aux = Mayor1 (V, n-1);
    if (V[n]> aux)
      return V[n];
    else
      return aux;
}
```

6. Buscar el máximo entre dos posiciones de un vector

$$Mayor2(V, i, d) =$$

$$= \begin{cases} V[i] & \text{si } i = d \\ Mayor2(V, i, (i+d)/2 \text{ ó} & \\ Mayor2(V, ((i+d)/2)+1, d) & \text{si } i < d \end{cases}$$

```
int Mayor2 (int *V, int izq, int der)
{
  int m_izq, m_der;
  if (izq==der)
    return V[izq];
  else {
    m_izq = Mayor2(V, izq, (izq+der)/2);
    m_der = Mayor2(V, ((izq+der)/2)+1, der);
    if (m_izq> m_der)
      return m_izq;
    else
      return m_der;
  }
}
```

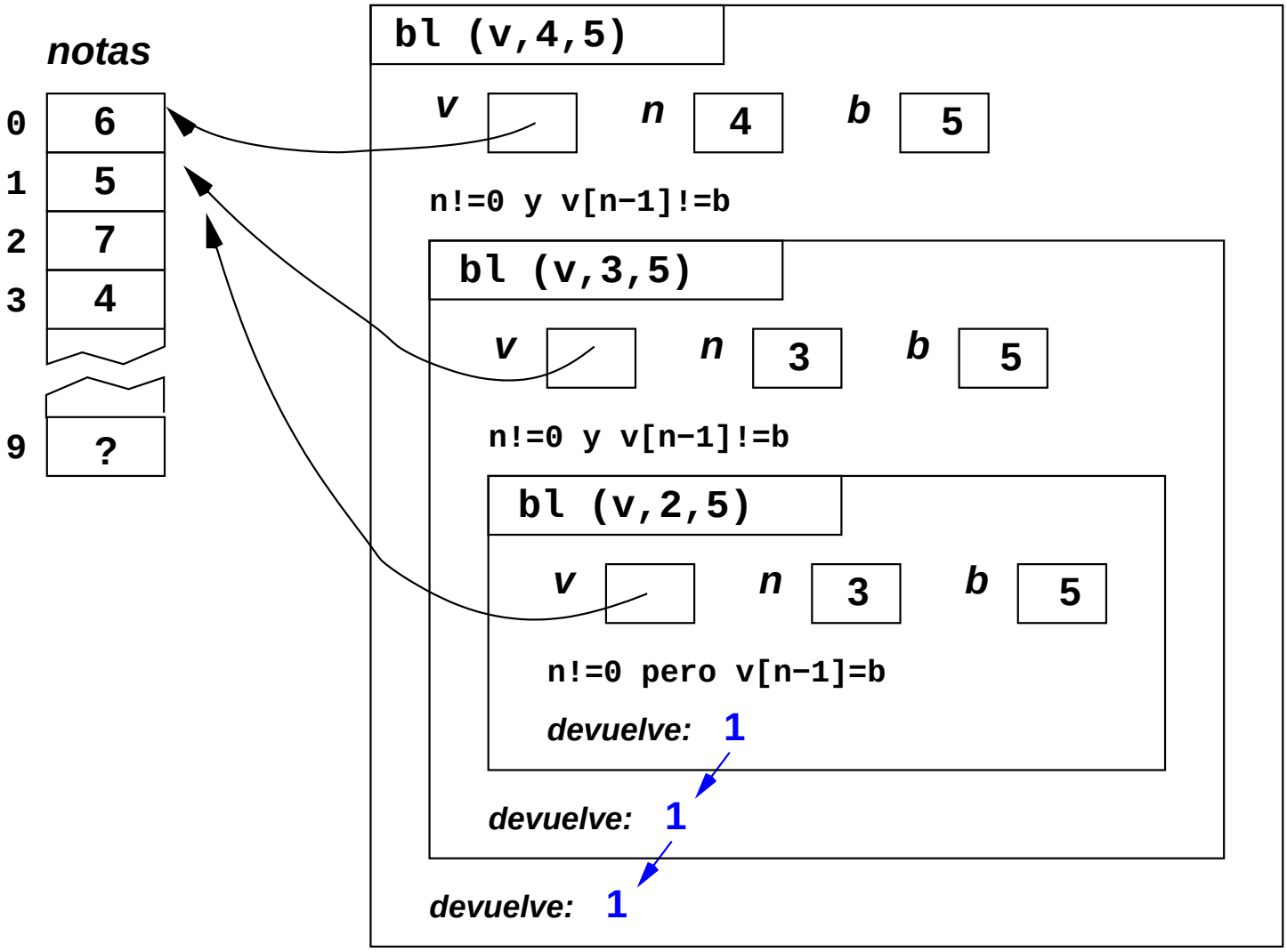
7. Búsqueda lineal recursiva (con dos casos base)

$$BusquedaLineal(V, n, b) =$$

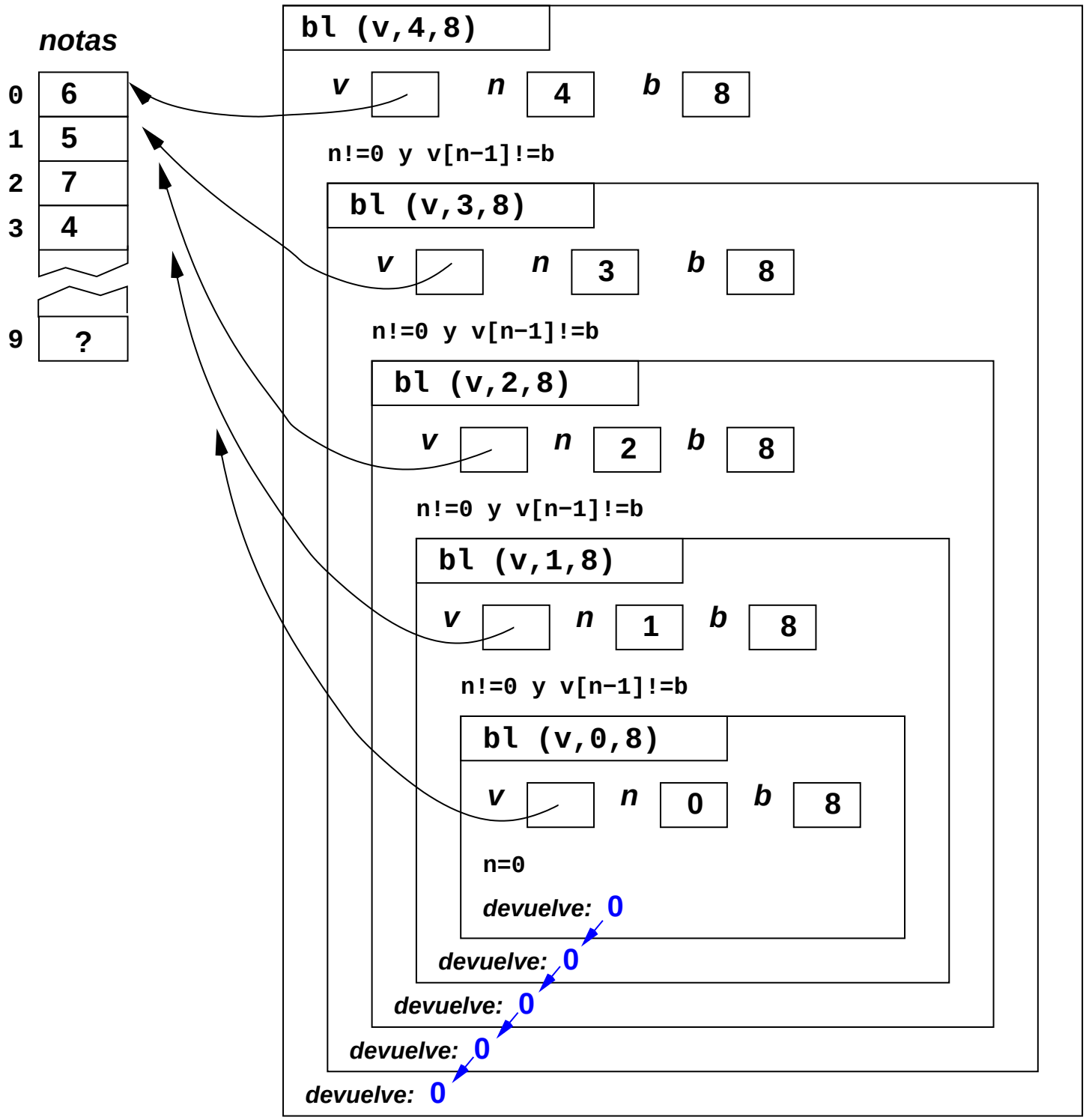
$$= \begin{cases} \text{Verdad} & \text{si } V[n] = b \\ \text{Falso} & \text{si } V[0] \neq b \\ (V[n] == b) \text{ ó} & \\ (b \in \{V[0], \ldots, V[n-1]\}) & \text{en otro caso} \end{cases}$$

```
int BusquedaLineal(int *V, int n, int b)
{
  if (n<0)
    return 0;
  else
    if (V[n]==b)
      return 1;
    else
      return BusquedaLineal(V,n-1,b);
 }
```

Ejemplo: Llamada `BusquedaLineal(notas, 4, 5);`



Ejemplo: Llamada `BusquedaLineal(notas, 4, 8);`



3.6 Ejemplos más complejos

3.6.1 Busqueda binaria recursiva

- **Motivación**: La búsqueda entre dos posiciones de un *vector ordenado* se puede realizar comparando el valor buscado con el elemento central:
  - Si es igual, la búsqueda termina con éxito.
  - Si es menor, la búsqueda debe continuar en el subvector izquierdo.
  - Si es mayor, la búsqueda debe continuar en el subvector derecho.

- Cabecera de una función de búsqueda:

```
int BUSCA (int v[], int i, int d, int x);
```

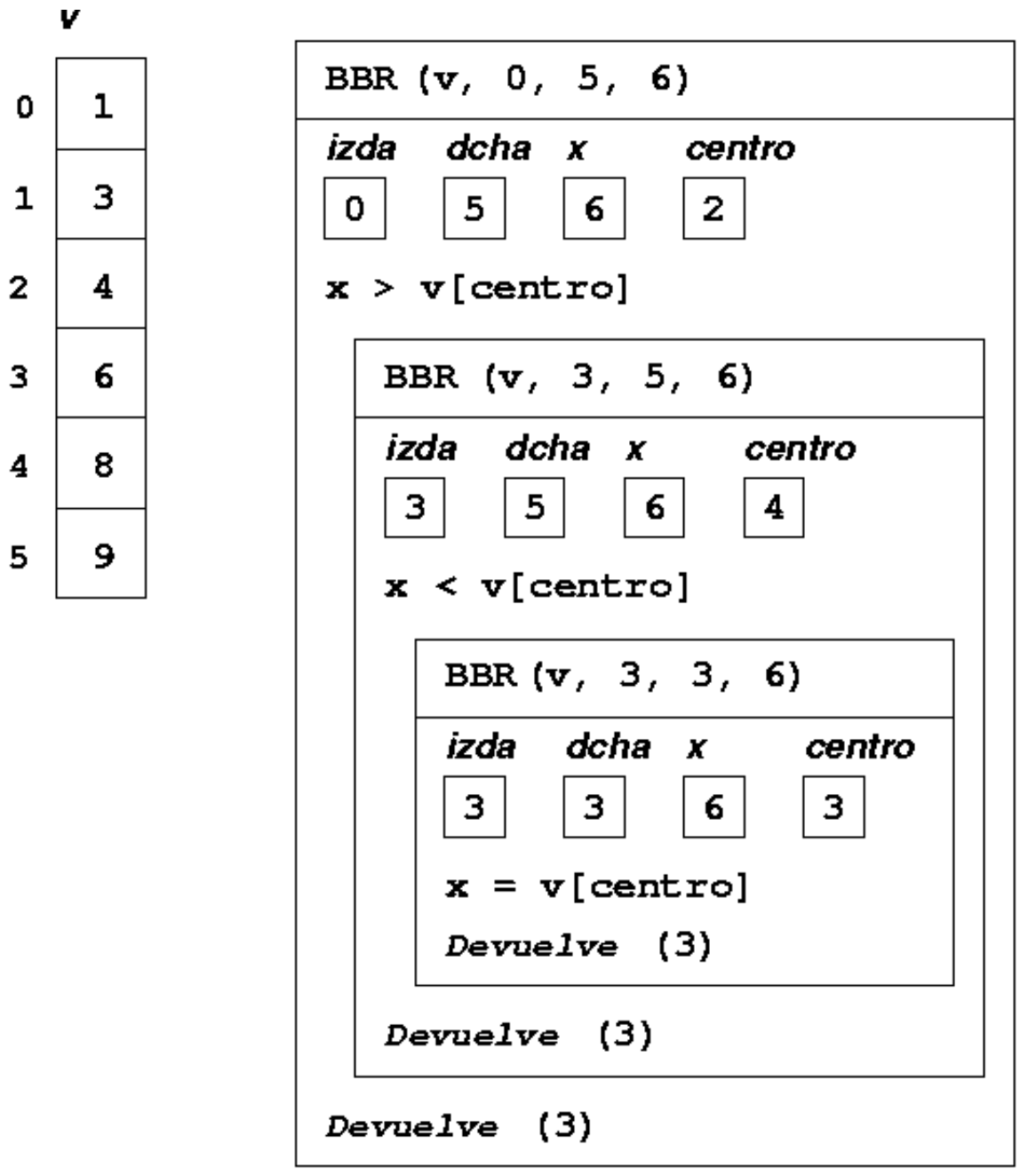
  Devuelve la posición en `v` donde se encuentra `x`. La búsqueda se realiza entre las posiciones `i` y `d`. Si `x` no está en el vector, la función devuelve `-1`.

- Líneas básicas (`BUSCA (v, t+1, d, x)`):

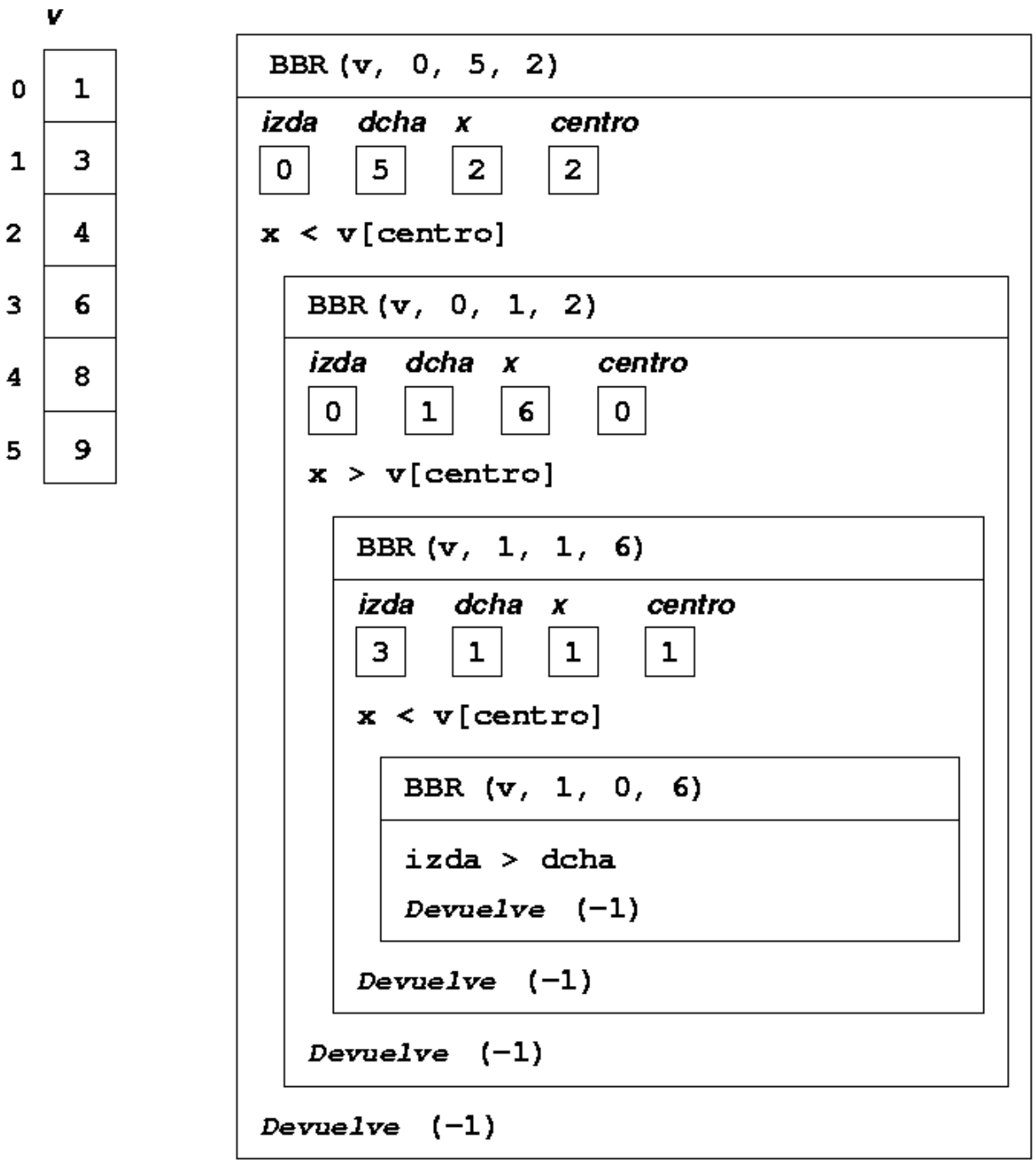
1. Seleccionar una casilla cualquiera, t, entre las casillas i y j ($i \leq t \leq j$). Sea $c = v[t]$. P.e. $t = (i + j)/2$
2. Comparar c con x.
   a) Si $c = x$, el elemento buscado está en la posición t (Éxito).
   b) Si $c < x$, el elemento buscado debe estar en una posición mayor que t: `BUSCA (v, t+1, d, x)`
   c) Si $c > x$, el elemento buscado debe estar en una posición menor que t: `BUSCA (v, i, t-1, x)`
   d) Al modificar los extremos puede darse el caso de que $i > d \Longrightarrow$ terminar (Fracaso).

```
int BBR (int v[], int i, int d, int x)
{
   int centro;

   if (i<=d) {

      centro = (i+d)/2;

      if (v[centro]==x) // Caso base 1
         return centro;

      else
         if (v[centro]>x) // Buscar izda.
            return BBR (v,i,centro-1,x);
         else // Buscar dcha.
            return BBR (v,centro+1,d,x);
   }

   else // i > d
      return -1; // Caso base 2
}
```

- Ejemplo: Búsqueda con éxito



- Ejemplo: Búsqueda con fracaso



Si queremos utilizar la solución recursiva anterior para buscar sobre un vector completo sólo tendremos que hacer lo siguiente:

```
int busqueda_binaria (int v[], int n, int x)
{
  return BBR(v, 0, n-1, x);
}
```

3.6.2 Paso de parámetros

- Paso por valor:cada ejecución trabaja con una copia.

- Paso por referencia: existe una misma variable para ese parámetro formal, eso quiere decir que se si cambia en una llamada queda cambiado para todas.

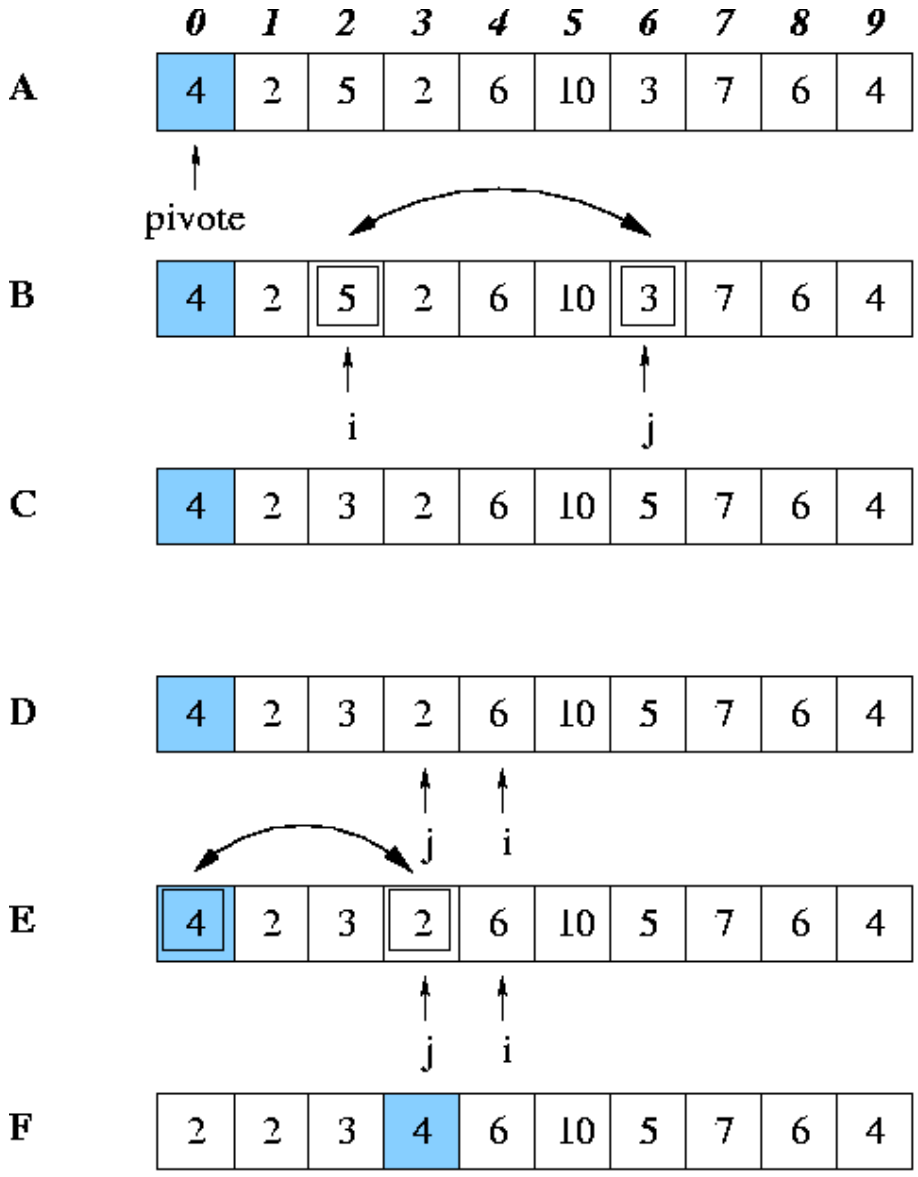
```
string invertir(string cad)
{
  if (cad.length()<=1)
    return cad;
  else
    return cad[cad.length()-1] + invertir(cad.substr(1,cad
}

void invertir2(string cad, string &res)
{
  if (cad.length()<=1)
    res = cad;
  else {
    string aux, auxcad;
    //Todos menos primer y último car
    auxcad = cad.substr(1,cad.length()-2);.
    invertir2(auxcad, aux);

    res = cad[cad.length()-1] + aux +cad[0];
  }
}
```

3.6.3 Ordenación rápida

- **Líneas básicas**:

1. Se toma un elemento arbitrario del vector, al que denominaremos **pivote**. Sea p su valor.
2. Se recorre el vector, de izquierda a derecha, hasta encontrar un elemento situado en una posición i tal que $v[i] > p$.
3. Se recorre el vector, de derecha a izquierda, hasta encontrar otro elemento situado en una posición j tal que $v[j] < p$.
4. Una vez localizados, se intercambian los elementos situados en las casillas i y j (ahora, $v[i] < p < v[j]$).
5. Repetir hasta que los dos procesos de recorrido se encuentren.
6. Si ahora colocamos el pivote en el sitio que le corresponde, el vector está particionado en dos zonas delimitadas por el pivote.

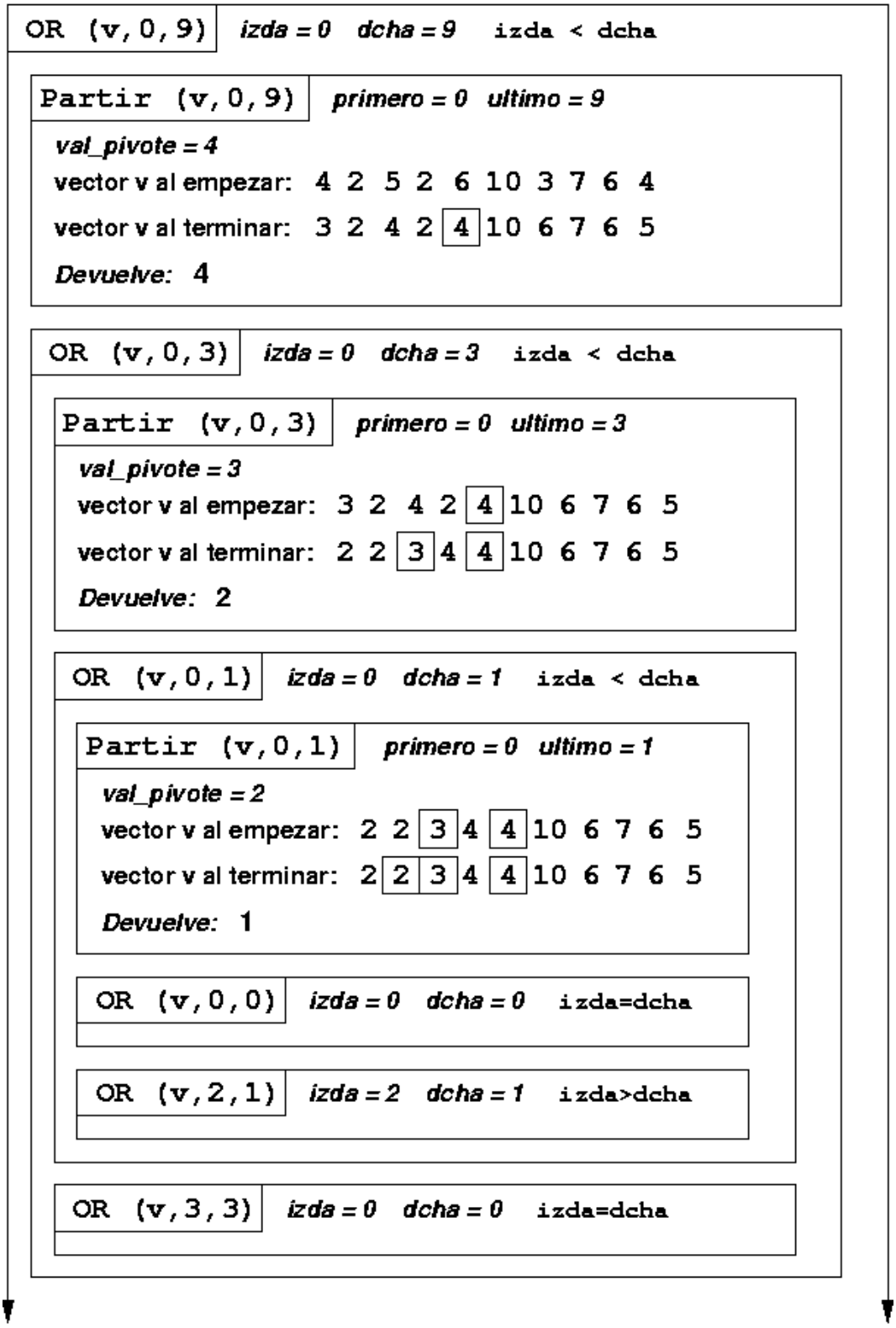
- **Ejemplo**:

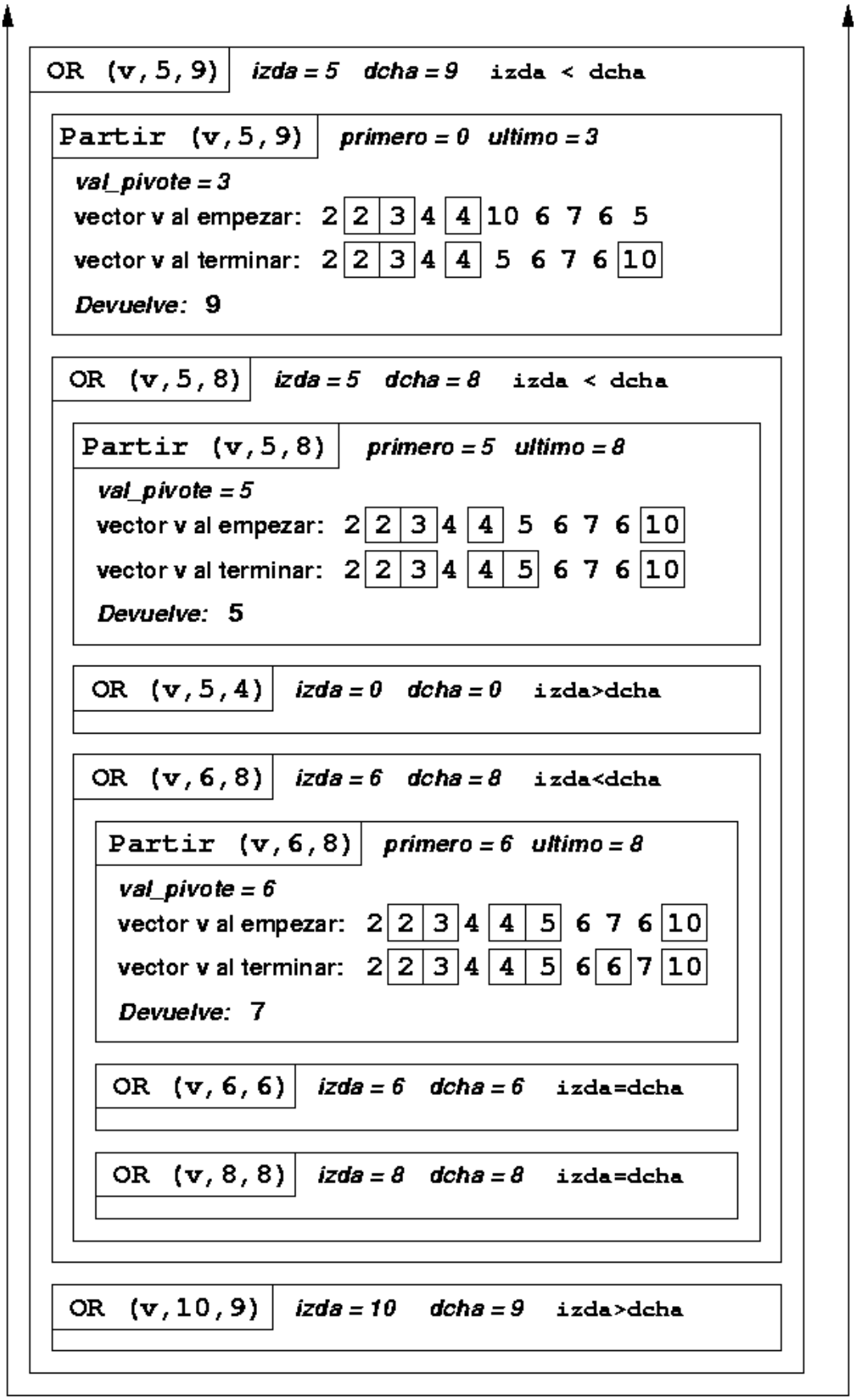

0 1 2 3 4 5 6 7 8 9
A
4 2 5 2 6 10 3 7 6 4
pivote
B
4 2 5 2 6 10 3 7 6 4
i
j
C
4 2 3 2 6 10 5 7 6 4
D
4 2 3 2 6 10 5 7 6 4
j
i
E
4 2 3 2 6 10 5 7 6 4
j
i
F
2 2 3 4 6 10 5 7 6 4

```
// Funcion de particion
int partir (int *v, int primero, int ultimo);

void OR (int *v, int izda, int dcha)
{
   int pos_pivote; // Pos. del pivote tras particion

   if (izda < dcha) {

      // Particionar "v"
      pos_pivote = partir (v, izda, dcha);

      // Ordenar la primera mitad
      OR (v, izda, pos_pivote-1);

      // Ordenar la segunda mitad
      OR (v, pos_pivote+1, dcha);
   }
}

int partir (int *v, int primero, int ultimo)
{
   void intercambia_int (int &a, int &b);

   int izda, dcha;  // Indices para recorrer v
   int val_pivote;  // Valor del pivote.

   // El pivote es el primer elemento.
   val_pivote = v[primero];
```

```
   izda = primero + 1;  // "izda" va a la dcha.
   dcha = ultimo;       // "dcha" va a la izda.

   do { // Buscar e intercambiar elementos
      // Buscar un elemento mayor que el pivote
      // avanzando desde la izquierda
      while ((izda<=dcha) && (v[izda]<=val_pivote))
         izda++;

      // Buscar un elemento menor o igual que el pivote
      // avanzando desde la derecha
      while ((izda<=dcha) && (v[dcha]>val_pivote))
         dcha--;

      if (izda < dcha) { // Intercambiar
         intercambia_int (v[izda], v[dcha]);
         dcha--;
         izda++;
      }
   } while (izda <= dcha); // Terminar cuando se
                           //cruzan "izda" y "dcha"

   // Colocar el pivote en su sitio correcto
   intercambia_int (v[primero], v[dcha]);
   return dcha; // Devolver la pos. del pivote
}
```

Ejemplo:


OR (v,0,9) izda = 0 dcha = 9 izda < dcha
Partir (v,0,9) primero = 0 ultimo = 9
val_pivote = 4
vector v al empezar: 4 2 5 2 6 10 3 7 6 4
vector v al terminar: 3 2 4 2 4 10 6 7 6 5
Devuelve: 4
OR (v,0,3) izda = 0 dcha = 3 izda < dcha
Partir (v,0,3) primero = 0 ultimo = 3
val_pivote = 3
vector v al empezar: 3 2 4 2 4 10 6 7 6 5
vector v al terminar: 2 2 3 4 4 10 6 7 6 5
Devuelve: 2
OR (v,0,1) izda = 0 dcha = 1 izda < dcha
Partir (v,0,1) primero = 0 ultimo = 1
val_pivote = 2
vector v al empezar: 2 2 3 4 4 10 6 7 6 5
vector v al terminar: 2 2 3 4 4 10 6 7 6 5
Devuelve: 1
OR (v,0,0) izda = 0 dcha = 0 izda=dcha
OR (v,2,1) izda = 2 dcha = 1 izda>dcha
OR (v,3,3) izda = 0 dcha = 0 izda=dcha


OR (v,5,9) izda = 5 dcha = 9 izda < dcha
Partir (v,5,9) primero = 0 ultimo = 3
val_pivote = 3
vector v al empezar: 2 2 3 4 4 10 6 7 6 5
vector v al terminar: 2 2 3 4 4 5 6 7 6 10
Devuelve: 9
OR (v,5,8) izda = 5 dcha = 8 izda < dcha
Partir (v,5,8) primero = 5 ultimo = 8
val_pivote = 5
vector v al empezar: 2 2 3 4 4 5 6 7 6 10
vector v al terminar: 2 2 3 4 4 5 6 7 6 10
Devuelve: 5
OR (v,5,4) izda = 0 dcha = 0 izda>dcha
OR (v,6,8) izda = 6 dcha = 8 izda<dcha
Partir (v,6,8) primero = 6 ultimo = 8
val_pivote = 6
vector v al empezar: 2 2 3 4 4 5 6 7 6 10
vector v al terminar: 2 2 3 4 4 5 6 6 7 10
Devuelve: 7
OR (v,6,6) izda = 6 dcha = 6 izda=dcha
OR (v,8,8) izda = 8 dcha = 8 izda=dcha
OR (v,10,9) izda = 10 dcha = 9 izda>dcha

3.7 ¿Recursividad o iteración?

- Cuestiones a tener en cuenta:

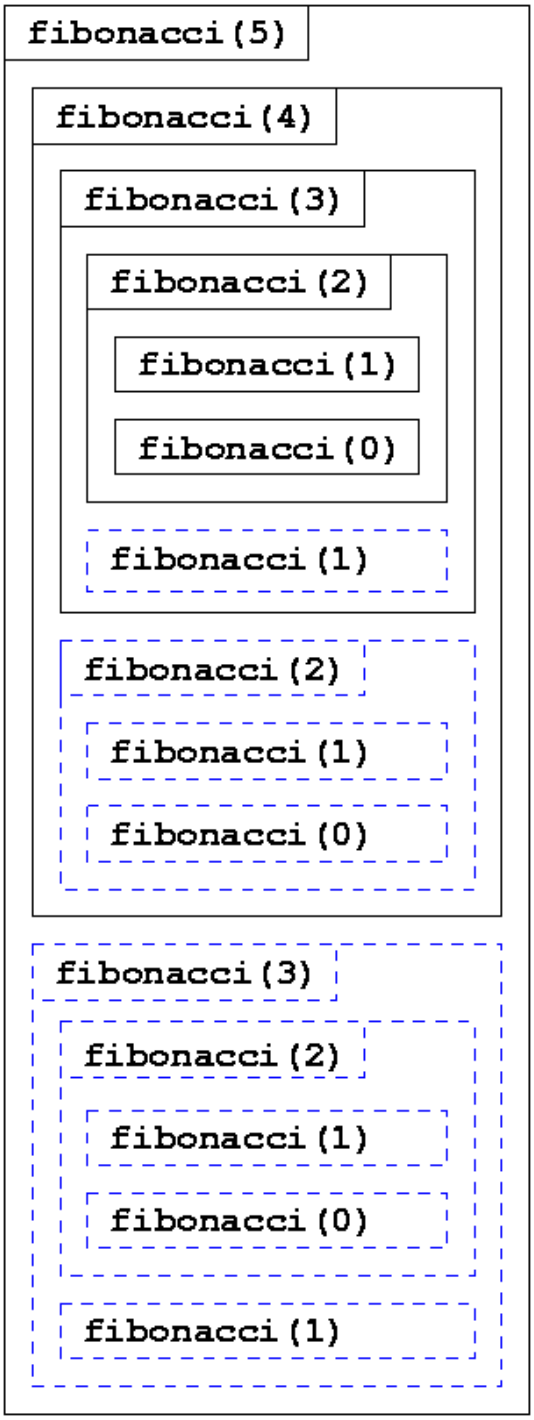
1. La carga computacional (tiempo-espacio) asociada a una llamada a una función y el retorno a la función que hace la llamada.
2. Algunas soluciones recursivas pueden hacer que la solución para un determinado tamaño del problema se calcule varias veces.
3. Muchos problemas recursivos tienen como caso base la resolución del problema para un tamaño muy reducido. En ocasiones resulta $excesiva\text{-}mente$ pequeño.
4. La solución iterativa (igual de eficiente) puede ser muy compleja de encontrar.
5. La solución recursiva es muy concisa, legible y elegante.

3.7.1 Sucesión de Fibonacci

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

```
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: Cálculo de `fibonacci (5)`



3.7.2 Búsqueda binaria recursiva (2)

```
int BLineal (int v[], int i, int d, int x);
bool pequenio_BBR (int);

int BBR2 (int v[], int i, int d, int x)
{
   if (pequenio_BBR (d-i+1))      // usar un
      return BLineal (v,i,d,x); // algoritmo simple

   else {
      int centro = (i+d)/2;

      if (v[centro]==x) // Exito
         return centro;
      else { // Seguir buscando
         if (v[centro]>x) // Buscar izda.
            return BBR (v,i,centro-1,x);
         else // Buscar dcha.
            return BBR (v,centro+1,d,x);
      }
   }
}
```

```
bool pequenio_BBR (int)
{
   return n <= BBR_UMBRAL;
}

int BLineal (int v[], int i, int d, int x)
{
   bool encontrado=false;

   for (int p=i; (i<d) && !encontrado; i++)
       if (v[i] == x) encontrado = true;

   if (encontrado)
     return 1;
   return -1;
}
```

- Notas:

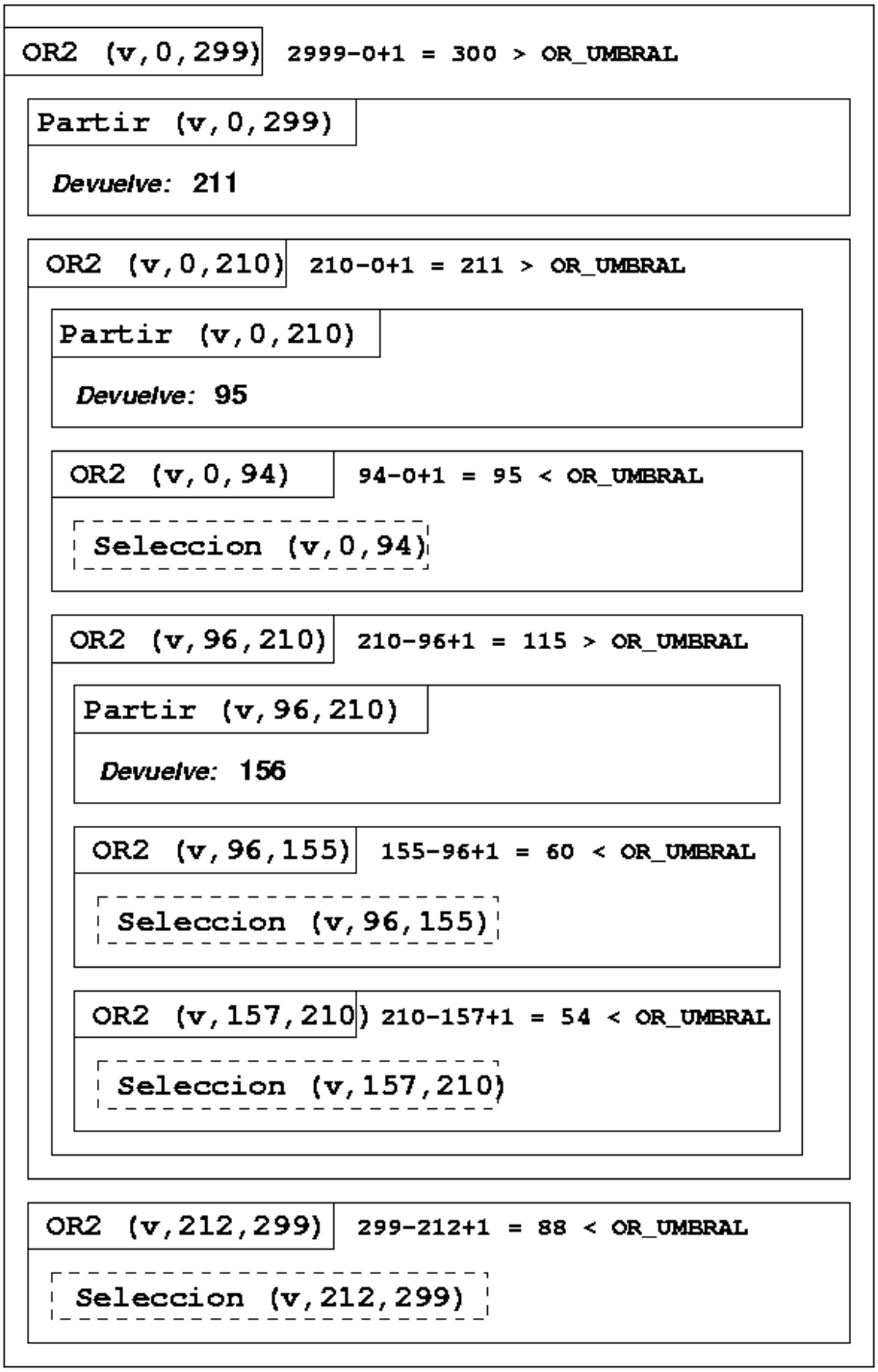
1. El caso base 2 (Fracaso) de la función `BBR()` ya no es necesario porque no se debe dar el caso de que $i > d$.
2. Es obligatorio que la función que resuelve el problema para un tamaño pequeño (`BLineal()`) devuelva un valor coherente con el que devuelve `BBR2()`.

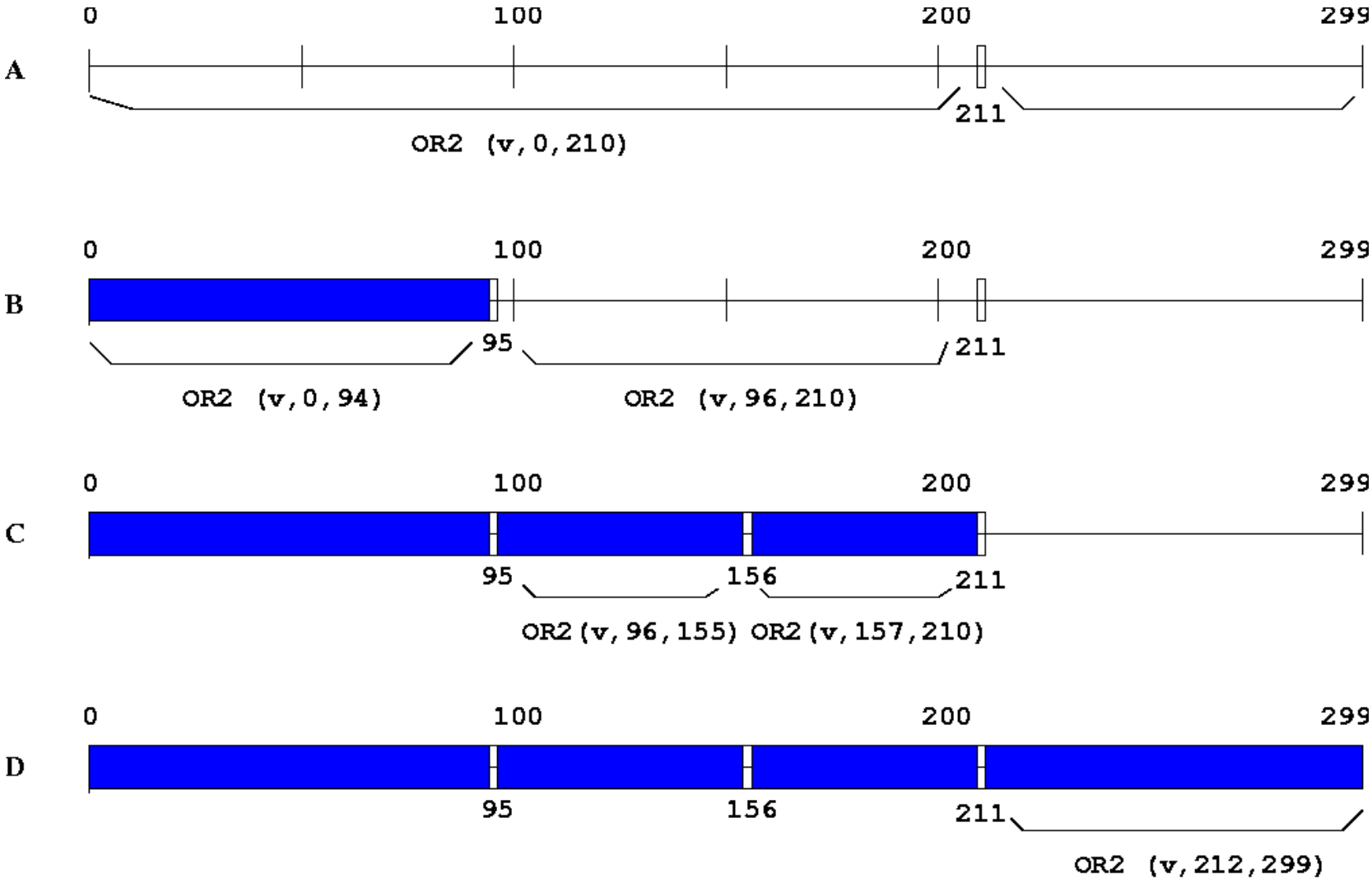
3.7.3 Ordenación rápida (2)

```
void seleccion (int *v, int izda, int dcha);
int  partir (int *v, int primero, int ultimo);
int  pequenio_OR (int n);


void OR2 (int *v, int izda, int dcha)
{
   if (pequenio_OR (dcha-izda+1)) // usar un
      seleccion (v, izda, dcha);  // algoritmo simple
   else {
      if (izda < dcha) {
          int pos_pivote; // Pos. pivote tras partir

         // Particionar "v"
         pos_pivote = partir (v, izda, dcha);

         // Ordenar la primera mitad
         OR2 (v, izda, pos_pivote-1);

         // Ordenar la segunda mitad
         OR2 (v, pos_pivote+1, dcha);
      }
   }
}
```

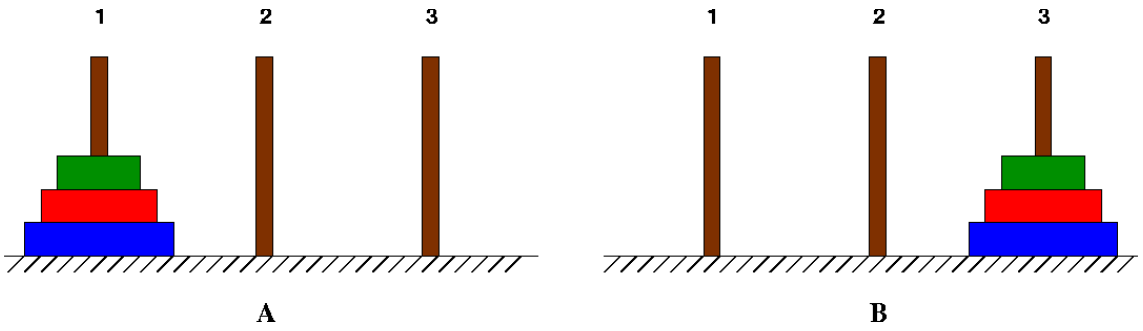
```
bool pequenio_OR (int)
{
   return n <= OR_UMBRAL;
}

void seleccion (int *v, int izda, int dcha)
{
   int i, j, pos_menor;
   int menor;

   for (i = izda; i < dcha; i++) {

      pos_menor = i;
      menor = v[i];

      for (j = i+1; j <= dcha; j++)
         if (v[j] < menor) {
            pos_menor = j;
            menor = v[j];
         }

      v[pos_menor] = v[i];
      v[i] = menor;
   }
}
```

Ejemplo: OR2 (v, 0, 299) y OR_UMBRAL=100




A
0
100
200
299
211
OR2 (v,0,210)
B
0
100
200
299
95
211
OR2 (v,0,94)
OR2 (v,96,210)
C
0
100
200
299
95
156
211
OR2(v,96,155) OR2(v,157,210)
D
0
100
200
299
95
156
211
OR2 (v,212,299)

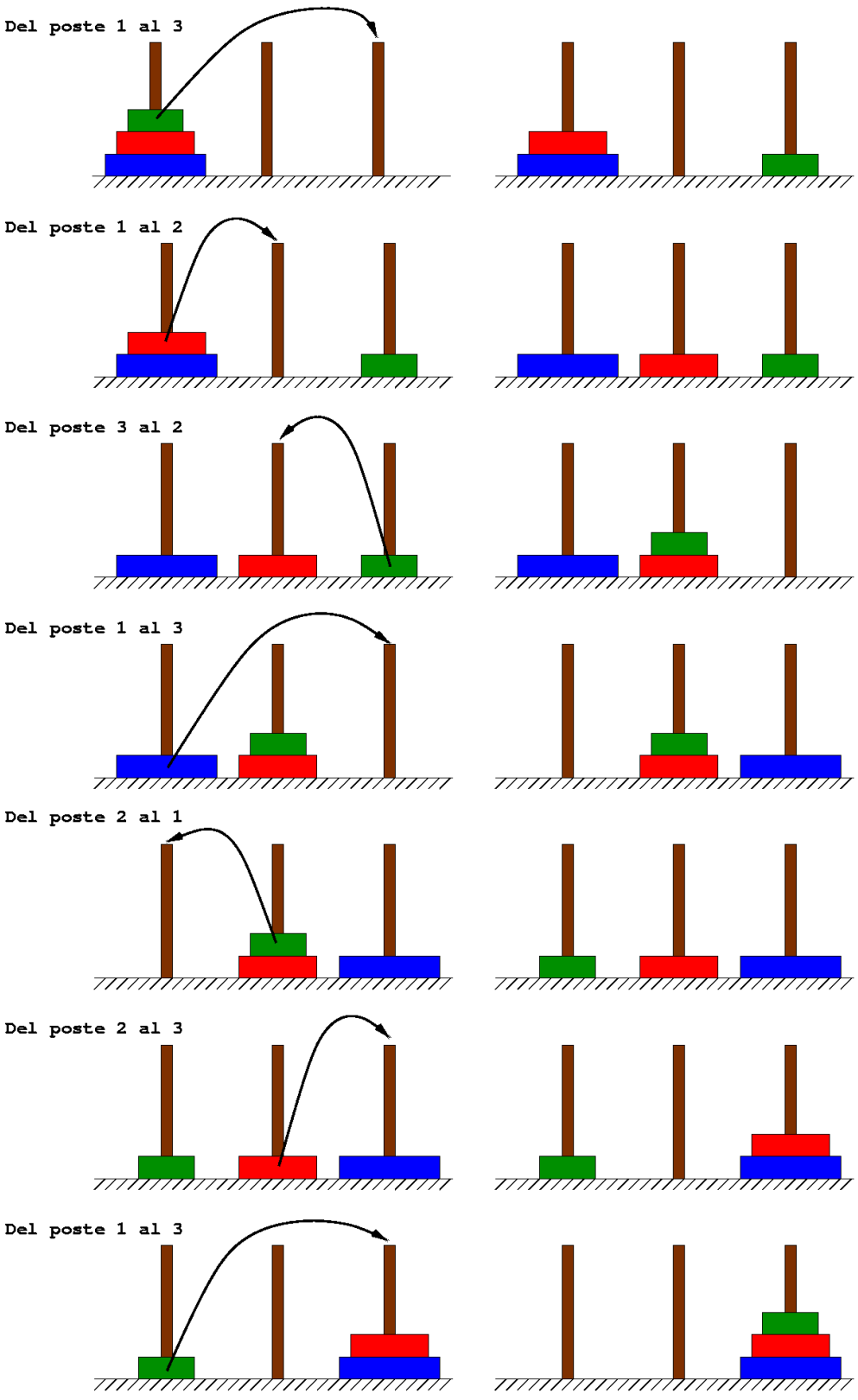
3.7.4 Torres de Hanoi




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

int main (void)
{
   void hanoi (int n, int inic, int tmp, int final);
   int n; // Numero de discos a mover

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

   hanoi (n, 1, 2, 3); // mover "n" discos del 1 al 3,
                       // usando el 2 como temporal.
   return 0;
}
```

```
void hanoi (int n, int inic, int tmp, int final)
{
   if (n > 0) {
      // Mover n-1 discos de "inic" a "tmp".
      // El temporal es "final".
      hanoi (n-1, inic, final, tmp);

      // Mover el que queda en "inic"a "final"
      cout <<"Del poste "<<inic<<" al "<<final<<"\n";

      // Mover n-1 discos de "tmp" a "final".
      // El temporal es "inic".
      hanoi (n-1, tmp, inic, final);
    }
}
```

```
% hanoi
Numero de discos: 3
Del poste  1 al  3
Del poste  1 al  2
Del poste  3 al  2
Del poste  1 al  3
Del poste  2 al  1
Del poste  2 al  3
Del poste  1 al  3
```


Del poste 1 al 3
Del poste 1 al 2
Del poste 3 al 2
Del poste 1 al 3
Del poste 2 al 1
Del poste 2 al 3
Del poste 1 al 3