TEMA 2
DISEÑO DE ALGORITMOS RECURSIVOS

1. Introducción a la recursión
2. Diseño de algoritmos recursivos
3. Análisis de algoritmos recursivos
4. Transformación de la recursión final a forma iterativa
5. Técnicas de generalización

Bibliografía:

Fundamentals of Data Structures in C++
E. Horowitz, S. Sahni, D. Mehta
Computer Science Press, 1995

Data Abstraction and Problem Solving with C++, Second Edition
Carrano, Helman y Veroff

2.1 Introducción a la recursión

- Optamos por una solución recursiva cuando sabemos cómo resolver de manera directa un problema para un cierto conjunto de datos, y para el resto de los datos somos capaces de resolverlo utilizando la solución al mismo problema con unos datos "más simples".

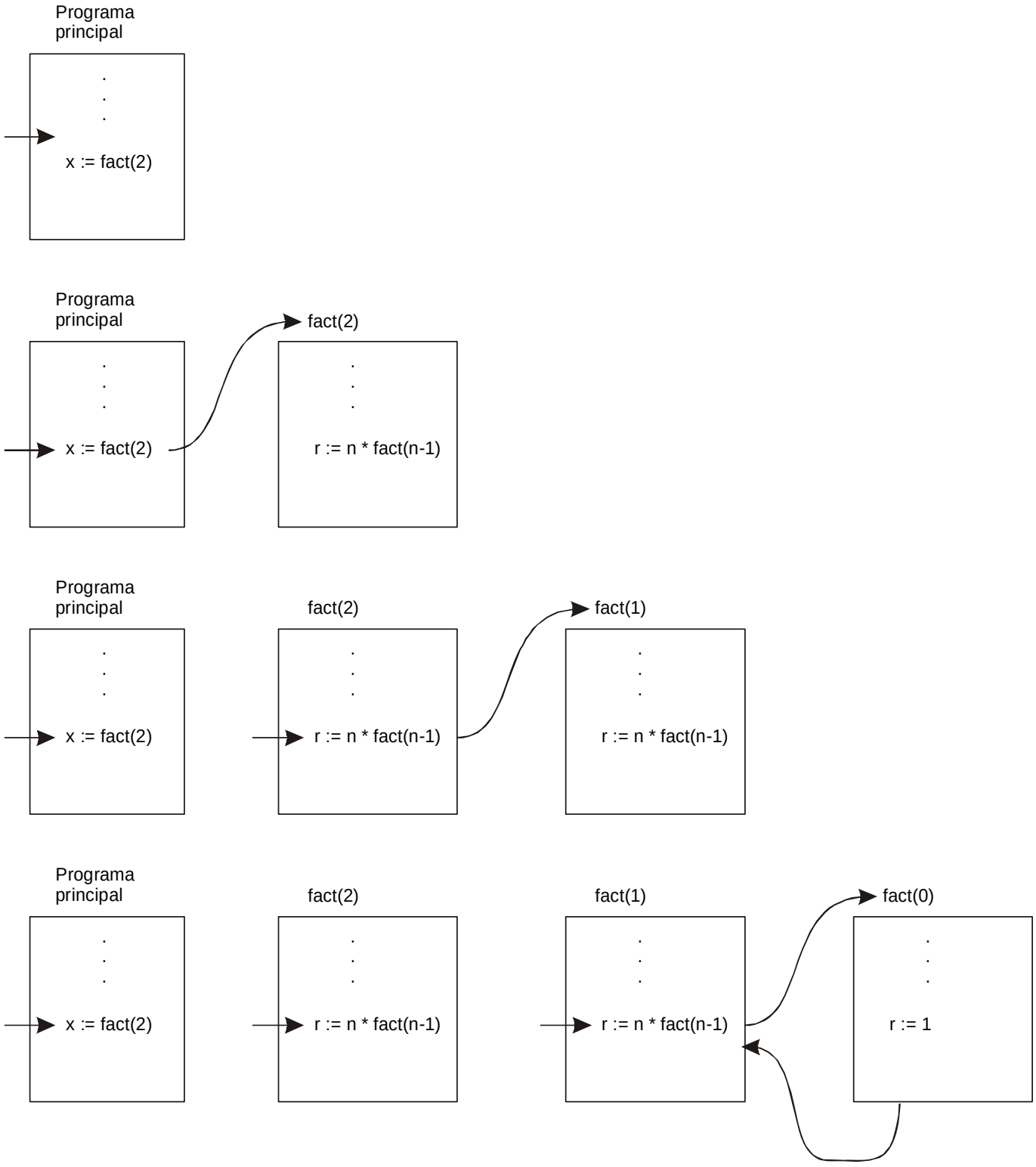
- Cualquier solución recursiva se basa en un análisis (clasificación) de los datos, $\vec{x}$, para distinguir los casos de solución directa y los casos de solución recursiva:
  - caso(s) directo(s): $\vec{x}$ es tal que el resultado $\vec{y}$ puede calcularse directamente de forma sencilla.
  - caso(s) recursivo(s): sabemos cómo calcular a partir de $\vec{x}$ otros datos más pequeños $\vec{x}$', y sabemos además cómo calcular el resultado $\vec{y}$ para $\vec{x}$ suponiendo conocido el resultado $\vec{y}$' para $\vec{x}$'.

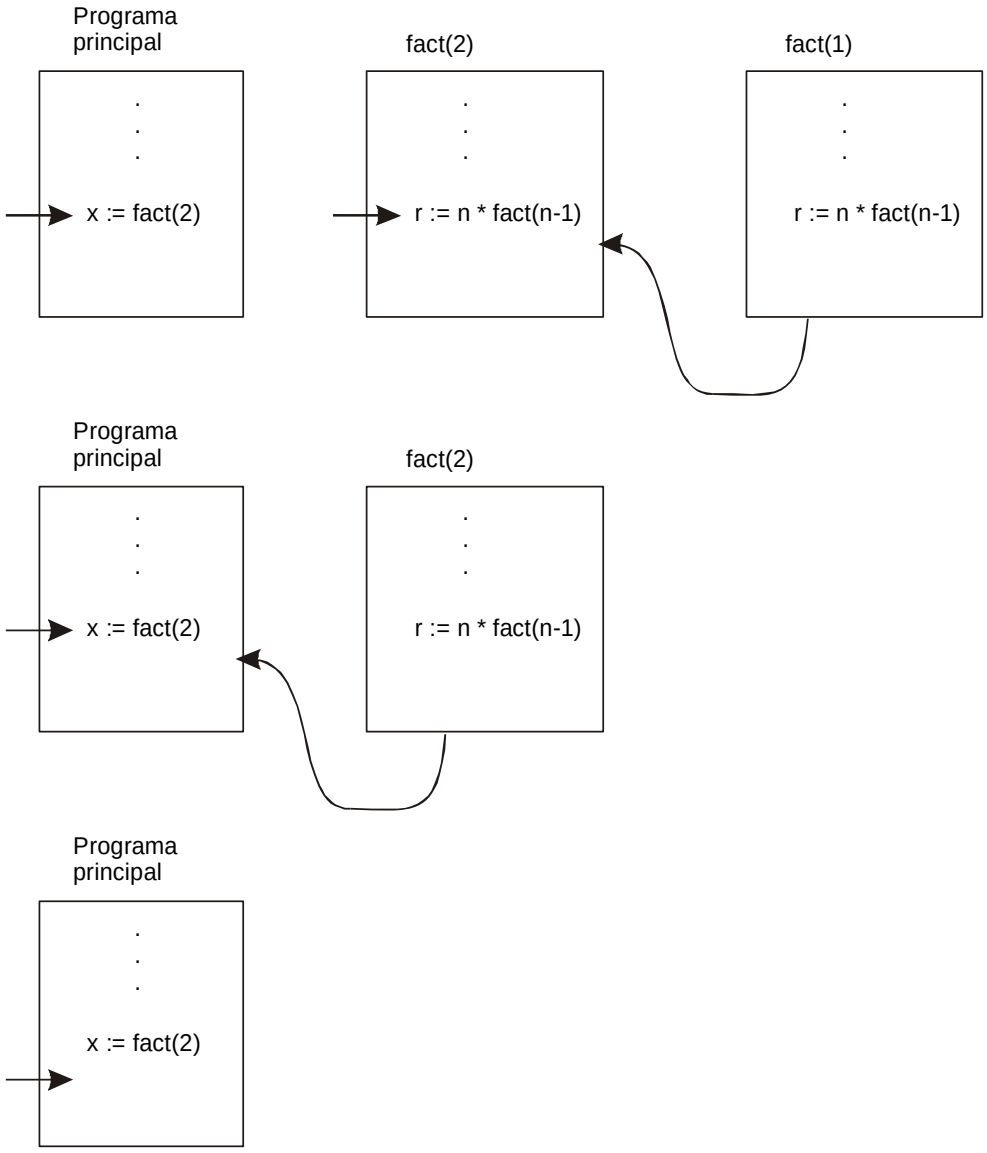
- Para implementar soluciones recursivas en un lenguaje de programación tenemos que utilizar una acción que se invoque a sí misma –con datos cada vez "más simples"–: funciones o procedimientos.

- Para entender la recursividad, a veces resulta útil considerar cómo se ejecutan las acciones recursivas en una computadora, como si se crearan múltiples copias del mismo código, operando sobre datos diferentes (en realidad sólo se copian las variables locales y los parámetros por valor).
  El ejemplo clásico del factorial:

```
 int fact ( int n ) {
// P : n >= 0

   int r;

   if      ( n == 0 ) r = 1;
   else if ( n > 0  ) r = n * fact(n-1);
   return r;

 // Q : devuelve n!
 }
```

¿Cómo se ejecuta la llamada *fact(2)*?


Programa
principal
x := fact(2)
Programa
principal
fact(2)
x := fact(2)
r := n * fact(n-1)
Programa
principal
fact(2)
fact(1)
x := fact(2)
r := n * fact(n-1)
r := n * fact(n-1)
Programa
principal
fact(2)
fact(1)
fact(0)
x := fact(2)
r := n * fact(n-1)
r := n * fact(n-1)
r := 1



- ¿La recursión es importante?
  - Un método muy potente de diseño y razonamiento formal.
  - Tiene una relación natural con la inducción y, por ello, facilita conceptualmente la resolución de problemas y el diseño de algoritmos.
  - Algunos lenguajes no incluyen instrucciones iterativas.

2.1.1 Recursión simple

- Decimos que una acción recursiva tiene recursión simple si cada caso recursivo realiza exactamente una llamada recursiva. Abusando de la notación en las asignaciones, podemos describirlo mediante el esquema general:

```
void nombreProc ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym ) {
// Precondición
// declaración de constantes
  τ1 x1' ; ... ; τn xn' ;  // x⃗'
  δ1 y1' ; ... ; δm ym' ;  // y⃗'

  if ( d(x⃗) )
    y⃗ = g(x⃗);
  else if ( ¬d(x⃗) ) {
    x⃗' = s(x⃗);
    nombreProc(x⃗', y⃗');
    y⃗ = c(x⃗, y⃗');
  }
// Postcondición
}
```

donde:

- $\vec{x}$ representa a los parámetros de entrada x_1, ... , x_n, $\vec{x}$' a los parámetros de la llamada recursiva x_1', ... , x_n', $\vec{y}$' a los resultados de la llamada recursiva y_1', ... , y_m', e $\vec{y}$ a los parámetros de salida y_1, ... , y_m.
- $d(\vec{x})$ es la condición que determina el caso directo
- $\neg d(\vec{x})$ es la condición que determina el caso recursivo
- g calcula el resultado en el caso directo
- s, la *función sucesor*, calcula los argumentos para la siguiente llamada recursiva
- c, la *función de combinación*, obtiene la combinación de los resultados de la llamada recursiva $\vec{y}$' junto con los datos de entrada $\vec{x}$, proporcionando así el resultado $\vec{y}$.

- A la recursión simple también se la conoce como *recursión lineal* porque el número de llamadas recursivas depende linealmente del tamaño de los datos.

- Veamos cómo la función factorial se ajusta a este esquema de declaración:

```
int fact ( int n ) {
// P : n >= 0

  int r;

  if      ( n == 0 ) r = 1;
  else if ( n > 0  ) r = n * fact(n-1);
  return r;

// Q : devuelve n!
}
```

$d(n) \Leftrightarrow n == 0$

$g(n) = 1$

$\neg d(n) \Leftrightarrow n > 0$

$s(n) = n-1$

$c(n, \text{fact}(s(n))) = n * \text{fact}(s(n))$

- $d(\vec{x})$ y $\neg d(\vec{x})$ pueden desdoblarse en una alternativa con varios casos.
  Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*

```
int prod ( int a, int b ) {
// P : ( a >= 0 ) && ( b >= 0 )
  int r;
  if      ( b == 0 )                  r = 0;
  else if ( b == 1 )                  r = a;
  else if ( b > 1 && (b % 2 == 0) ) r = prod(2*a, b/2);
  else if ( b > 1 && (b % 2 == 1) ) r = prod(2*a, b/2) + a;
  return r;
// Q : r = a * b
}
```

$d_1(a, b) \Leftrightarrow b == 0$

$g_1(a, b) = 0$

$\neg d_1(a, b) \Leftrightarrow b > 1$ && par(b)

$s_1(a, b) = (2*a, b/2)$

$c_1(a, b, \text{prod}(s_1(a, b))) = \text{prod}(s_1(a, b))$

$d_2(a, b) \Leftrightarrow b == 1$

$g_2(a, b) = 1$

$\neg d_2(a, b) \Leftrightarrow b > 1$ && impar(b)

$s_2(a, b) = (2*a, b/2)$

$c_2(a, b, \text{prod}(s_2(a, b))) = \text{prod}(s_2(a, b))+a$

Recursión final

- La *recursión final* (*tail recursion*) es un caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada recur-

siva. Se llama *final* porque lo último que se hace en cada pasada es la llamada recursiva.

El resultado será siempre el obtenido en uno de los casos base.

Los algoritmos recursivos finales tienen la interesante propiedad de que pueden ser traducidos de manera directa a soluciones iterativas, más eficientes.

- Como ejemplo de función recursiva final veamos el algoritmo de cálculo del máximo común divisor por el método de Euclides.

```
int mcd( int a, int b ){
// P : ( a > 0 ) && ( b > 0 )

  int m;

  if      ( a == b ) m = a;
  else if ( a >  b ) m = mcd(a-b, b);
  else if ( a <  b ) m = mcd(a, b-a);
  return m;
// Q : devuelve mcd(a, b)
}
```

que se ajusta al esquema de recursión simple:

$d(a, b) \Leftrightarrow a == b$
$g(a, b) = a$
$\neg d_1(a, b) \Leftrightarrow a > b$ $\qquad\qquad$ $\neg d_2(a, b) \Leftrightarrow a < b$
$s_1(a, b) = (a-b, a)$ $\qquad\qquad$ $s_2(a, b) = (a, b-a)$

y donde las funciones de combinación se limitan a devolver el resultado de la llamada recursiva

$c_1(a, b, \text{mcd}(s_1(a, b))) = \text{mcd}(s_1(a, b))$
$c_2(a, b, \text{mcd}(s_2(a, b))) = \text{mcd}(s_2(a, b))$

2.1.2 Recursión múltiple

- Este tipo de recursión se caracteriza por que, al menos en un caso recursivo, se realizan varias llamadas recursivas. El esquema correspondiente:

```
void nombreProc ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym ) {
// Precondición
// declaración de constantes
  τ1 x11 ; ... ; τn x1n ; ... τ1 xk1 ; ... ; τn xkn ;
  δ1 y11 ; ... ; δm y1m ; ... δ1 yk1 ; ... ; δm ykm ;

  if ( d(x⃗) )
    y⃗ = g(x⃗);
  else if ( ¬d(x⃗) ) {
    x⃗1 = s1(x⃗);
    nombreProc(x⃗1, y⃗1);
    ...
    x⃗k = sk(x⃗);
    nombreProc(x⃗k, y⃗k);
    y⃗ = c(x⃗, y⃗1, ... , y⃗k);
  }
// postcondición
}
```

  donde

  - $k > 1$, indica el número de llamadas recursivas
  - $\vec{x}$ representa a los parámetros de entrada $x_1,...,x_n$, $\vec{x}_i$ a los parámetros de la *i-ésima* llamada recursiva $x_{i1},...,x_{in}$, $\vec{y}_i$ a los resultados de la *i-ésima* llamada recursiva $y_{i1},...,y_{im}$, para $i = 1,...,k$, e $\vec{y}$ a los parámetros de salida $y_1,...,y_m$
  - $d(\vec{x})$ es la condición que determina el caso directo
  - $\neg d(\vec{x})$ es la condición que determina el caso recursivo
  - g calcula el resultado en el caso directo
  - s_i, las *funciones sucesor*, calculan la descomposición de los datos de entrada para realizar la *i-ésima* llamada recursiva, para $i = 1, ... , k$
  - c, la *función de combinación*, obtiene la combinación de los resultados $\vec{y}_i$ de las llamadas recursivas, para $i = 1, ... , k$, junto con los datos de entrada $\vec{x}$, proporcionando así el resultado $\vec{y}$.

- Otro ejemplo clásico, los números de Fibonacci:

```
int fibo( int n ) {
// Pre: n >= 0

  int r;

  if      ( n == 0 ) r = 0;
  else if ( n == 1 ) r = 1;
  else if ( n >  1 ) r = fibo(n-1) + fibo(n-2);
  return r;
// Post: devuelve fib(n)
}
```

que se ajusta al esquema de recursión múltiple ($k = 2$)

$d_1(n) \Leftrightarrow n == 0$ $\quad$ $d_2(n) \Leftrightarrow n == 1$

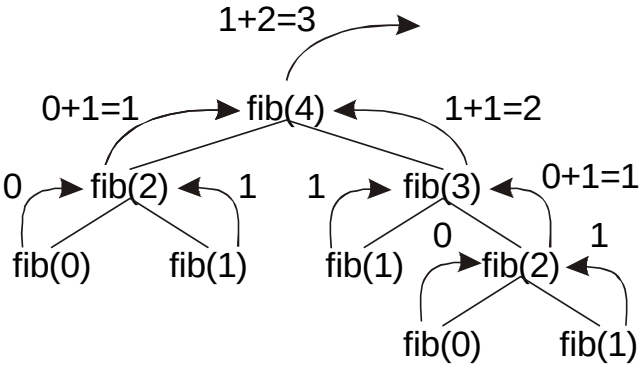
$g(0) == 0$ $\quad$ $g(1) == 1$

$\neg d(n) \Leftrightarrow n > 1$

$s_1(n) = n-1$ $\quad$ $s_2(n) = n-2$

$c(n, \text{fibo}(s_1(n)), \text{fibo}(s_2(n))) = \text{fibo}(s_1(n)) + \text{fibo}(s_2(n))$

- En la recursión múltiple, el número de llamadas no depende linealmente del tamaño de los datos. Por ejemplo, el cómputo de *fib(4)*



Nótese que algunos valores se computan más de una vez.

- Para terminar con la introducción, recapitulamos los distintos tipos de funciones recursivas que hemos presentado:
  - Simple. Una llamada recursiva en cada caso recursivo
    - No final. Requiere combinación de resultados
    - Final. No requiere combinación de resultados
  - Múltiple. Más de una llamada recursiva en algún caso recursivo.

2.2 Diseño de algoritmos recursivos

- Dada la especificación { P } A { Q }, hemos de obtener una acción A que la satisfaga.

  Nos planteamos implementar A como una función o un procedimiento recursivo cuando podemos obtener –fácilmente– una definición recursiva de la postcondición.

- Proponemos un método que descompone la obtención del algoritmo recursivo en varios pasos:

(R.1) **Planteamiento recursivo**. Se ha de encontrar una estrategia recursiva para alcanzar la postcondición, es decir, la solución. A veces, la forma de la postcondición, o de las operaciones que en ella aparecen, nos sugerirá directamente una estrategia recursiva.

(R.2) **Análisis de casos**. Se trata de obtener las condiciones que permiten discriminar los casos directos de los recursivos. Deben tratarse de forma exhaustiva y mutuamente excluyente todos los casos contemplados en la precondición:

$$P_0 \Rightarrow d(\vec{x}) \vee \neg d(\vec{x})$$

(R.3) **Caso directo**. Hemos de encontrar la acción que resuelve el caso directo

$$\{ P_0 \wedge d(\vec{x}) \} \; A_1 \; \{ Q_0 \}$$

Si hubiese más de un caso directo, repetiríamos este paso para cada uno de ellos.

(R.4) **Descomposición recursiva**. Se trata de obtener la función sucesor $s(\vec{x})$ que nos proporciona los datos que empleamos para realizar la llamada recursiva.

Si hay más de un caso recursivo, obtenemos la función sucesor para cada uno de ellos.

(R.5) **Función de acotación y terminación**. Determinamos si la función sucesor escogida garantiza la terminación de las llamadas, obteniendo una función que estime el número de llamadas restantes hasta alcanzar un caso base –la *función de acotación*– y justificando que se decrementa en cada llamada.

Si hay más de un caso recursivo, se ha de garantizar la terminación para cada uno de ellos.

(R.6) **Llamada recursiva**. Pasamos a ocuparnos entonces del caso recursivo. Cada una de las descomposiciones recursivas ha de permitir realizar la(s) llamada(s) recursiva(s).

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0[\vec{x}/s(\vec{x})]$$

(R.7) **Función de combinación**. Lo único que nos resta por obtener del caso recursivo es la función de combinación, que, en el caso de la recursión simple, será de la forma $\vec{y} = c(\vec{x}, \vec{y}')$.

Si hubiese más de un caso recursivo, habría que encontrar una función de combinación para cada uno de ellos.

(R.8) **Escritura del caso recursivo**. Lo último que nos queda por decidir es si necesitamos utilizar en el caso recursivo todas las variables auxiliares que han ido apareciendo. Partiendo del esquema más general posible

```
{ P0 ∧ ¬d(x⃗) }
  x⃗' = s(x⃗);
  nombreProc(x⃗', y⃗');
  y⃗ = c(x⃗, y⃗')
{ Q0 }
```

a aquel donde el caso recursivo se reduce a una única sentencia

```
{ P0 ∧ ¬d(x⃗) }
  y⃗ = c(x⃗, nombreFunc(s(x⃗)))
{ Q0 }
```

Repetimos este proceso para cada caso recursivo, si es que tenemos más de uno, y lo generalizamos de la forma obvia cuando tenemos recursión múltiple.

- Ejemplo: una función que dado un natural *n* calcule la suma de los dobles de los naturales hasta *n*:

Especificación

```
int sumaDoble ( int n ) {
// Pre : n >= 0
  int s;

// cuerpo de la función

  return s;
// Post : devuelve Σ i : 0 ≤ i ≤ n : 2 * i
}
```

(R.1) (R.2) Planteamiento recursivo y análisis de casos

El problema se resuelve trivialmente cuando $n == 0$, donde el resultado es 0. La distinción de casos queda entonces:

```
   d(n) : n == 0
  ¬d(n) : n > 0
```

que cubren exhaustivamente todos los casos posibles

```
n ≥ 0 ⇒ n == 0 ∨ n > 0
```

La estrategia recursiva se puede obtener manipulando la expresión que formaliza el resultado a obtener

```
Σ i : 0 ≤ i ≤ n : 2 * i  = (Σ i : 0 ≤ i ≤ n-1 : 2 * i) + 2
* n
```

que nos lleva directamente a la estrategia recursiva:

```
sumaDoble(n) = sumaDoble(n–1) + 2*n
```

(R.3) Solución en el caso directo.

```
{ n == 0 }
  A1
{ s == Σ i : 0 ≤ i ≤ n : 2 * i }
```

Se resuelve trivialmente con la asignación

```
A1 ≡ s = 0
```

(R.4) Descomposición recursiva.

La descomposición recursiva ya aparecía en el planteamiento recursivo

```
s(n) = n–1
```

(R.5) Función de acotación y terminación.

El tamaño del problema viene dado por el valor de *n*, que ha de disminuir en las sucesivas llamadas recursivas hasta llegar a 0:

```
t(n) = n
```

Efectivamente, se decrementa en cada llamada recursiva:

$t(s(n)) < t(n) \Leftrightarrow n-1 < n$

(R.6) Es posible hacer la llamada recursiva. En el caso recursivo se cumple

$P_0(n) \wedge \neg d(n) \Rightarrow P_0(s(n))$

$n \geq 0 \wedge n > 0 \Rightarrow n > 0 \Rightarrow n - 1 \geq 0$

(R.7) Función de combinación.

Esta función aparecía ya en el planteamiento recursivo. La postcondición se puede alcanzar con una simple asignación:

```
s = s' + 2*n
```

(R.8) Escritura del caso recursivo

Siguiendo el esquema teórico, tenemos que el caso recursivo se escribe:

```
{ P0 ∧ n > 0 }
  n' = n-1;
  s' = sumaDoble(n');
  s  = s' + 2*n;
{ Q0 }
```

Podemos evitar el uso de las dos variables auxiliares, simplemente incluyendo las expresiones correspondientes en la función de combinación:

```
    s = sumaDoble(n-1) + 2*n;
```

Con todo esto, la función queda:

```
int sumaDoble ( int n ) {
// Pre: n >= 0
  int  s;
  if      ( n == 0 ) s = 0;
  else if ( n >  0 ) s = sumaDoble( n-1 ) + 2*n;
  return s;
// Post: devuelve Σ i : 0 ≤ i ≤ n : 2*i
}
```

- Ejemplo: suma de las componentes de un vector de enteros.

  Especificación

```
int sumaVec ( int v[], int num ) {
// Pre : v es un array de al menos num elementos
  int  s;

// cuerpo de la función

  return s;
// devuelve Σ i : 0 ≤ i < num : v[i]
}
```

(R.1) (R.2) Planteamiento recursivo y análisis de casos

El problema se resuelve trivialmente cuando *num* == 0, porque la suma de las 0 primeras componentes de un vector es 0. Por lo tanto, distinguimos los casos

$$d(v, num) \equiv num == 0$$
$$\neg d(v, num) \equiv num > 0$$

¿qué ocurre si *num* < 0?

Podemos optar por:

– exigir en la precondición que sólo son válidas invocaciones donde *num* sea ≥ 0, o

– tratar también el caso en el que *num* < 0 devolviendo también 0

Optando por la segunda solución: $d(v, num) \equiv num \leq 0$

$$P_0 \Rightarrow num < 0 \vee num == 0 \vee num > 0 \Leftrightarrow cierto$$

El planteamiento recursivo puede ser: para obtener la suma de las *num* componentes de un vector, obtenemos la suma de las *num*–1 primeras y le sumamos la última.

```
sumaVec(v, num) = sumaVec(v, num-1) + v[num-1]
```

(R.3) Solución en el caso directo.

$$\{ P_0 \wedge num \leq 0 \}$$
$$A_1$$
$$\{ s = \Sigma\ i : 0 \leq i < num : v[i] \}$$

$$A_1 \equiv s = 0;$$

(R.4) Descomposición recursiva.

```
s(v, num) = (v, num-1)
```

(R.5) Función de acotación y terminación.

Al avanzar la recursión, *num* se va acercando a 0

```
t(v, num) = num
```

Se cumple

```
num - 1 < num
```

(R.6) Llamada recursiva.

```
        v es un array de al menos num elementos ∧ num > 0
    ⇒   v es un array de al menos num-1 elementos
```

(R.7) Función de combinación.

Se resuelve con la asignación

```
s = s' + v[num-1];
```

(R.8) Escritura del caso recursivo.

Bastará con la asignación

```
s = sumaVec(v, num-1) + v[num-1];
```

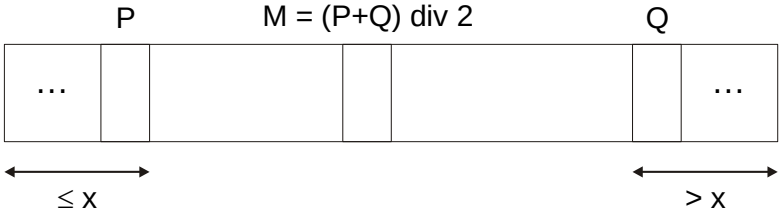
De forma que la función resultante queda:

```
int sumaVec ( int v[], int num ) {
// Pre : v es un array de al menos num elementos
  int  s;

 if      ( num <= 0 ) s = 0;
 else if ( num >  0 ) s = sumaVec(v, num-1) + v[num-1];
  return s;
// devuelve Σ i : 0 ≤ i < num : v[i]
}
```

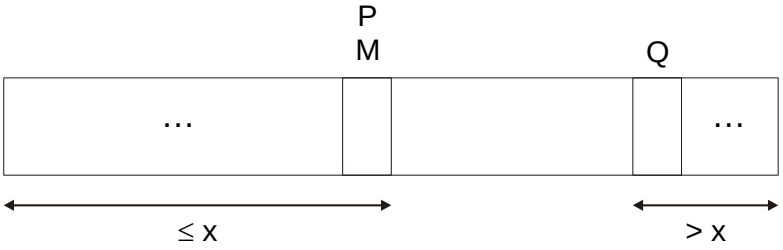
Implementación recursiva de la búsqueda binaria

- Partimos de un vector ordenado, donde puede haber elementos repetidos, y un valor x que pretendemos encontrar en el vector. Buscamos la aparición más a la derecha del valor x, o, si no se encuentra en el vector, buscamos la posición anterior a dónde se debería encontrar –por si queremos insertarlo–. Es decir, estamos buscando el punto del vector donde las componentes pasan de ser $\leq x$ a ser $> x$.

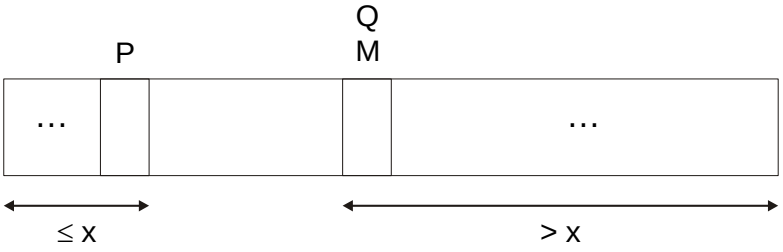
  La idea es que en cada pasada por el bucle se reduce a la mitad el tamaño del subvector donde puede estar el elemento buscado.



  si $v[m] \leq x$ entonces debemos buscar a la derecha de m



  y si $v[m] > x$ entonces debemos buscar a la izquierda de m



- Como el tamaño de los datos se reduce a la mitad en cada pasada tenemos claramente una complejidad logarítmica en el caso peor (en realidad en todos los casos, pues aunque encontremos x en el vector hemos de seguir buscando ya que puede que no sea la aparición más a la derecha).

- Hay que ser cuidadoso con los índices, sobre todo si:

– x no está en el vector, o si, en particular,
– x es mayor o menor que todos los elementos del vector;

además, es necesario pensar con cuidado cuál es el caso base.

- La especificación del algoritmo

```
typedef int TElem;
int buscaBin( TElem v[], int num, TElem x ) {
// Pre: v está ordenado entre 0 .. num-1
  int pos;

// cuerpo de la función

  return pos;
// Post: devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//       v[i] <= x
//       si x es menor que todos los elementos de v, devuelve
-1
}
```

Comentarios:

– Utilizamos el tipo *TElem* para resaltar la idea de que la búsqueda binaria es aplicable sobre cualquier tipo que tenga definido un orden, es decir, los operadores == y <=.
– Si x no está en v devolvemos la posición anterior al lugar donde debería estar. En particular, si x es menor que todos los elementos de v el lugar a insertarlo será la posición 0 y, por lo tanto, devolvemos –1.

- El planteamiento recursivo parece claro: para buscar x en un vector de n elementos tenemos que comparar x con el elemento central y
  – si x es mayor o igual que el elemento central, seguimos buscando recursivamente en la mitad derecha,
  – si x es menor que el elemento central, seguimos buscando recursivamente en la mitad izquierda.

  El problema es ¿cómo indicamos en la llamada recursiva que se debe seguir buscando "en la mitad izquierda" o "en la mitad derecha"?

  Evidentemente la llamada

```
buscaBin( v, num / 2, x );
```

  no funciona.
- Al diseñar algoritmos recursivos, en muchas ocasiones es necesario utilizar una función –o un procedimiento– auxiliar que nos permita implementar el plan-

teamiento recursivo. Estas funciones auxiliares son una *generalización* de la función –o el procedimiento– a desarrollar porque

- tienen más parámetros y/o más resultados, y
- la función original se puede calcular como un caso particular de la función auxiliar.

- En el caso de la búsqueda binaria, una posible generalización consiste en desarrollar una función que en lugar de recibir el número de elementos del vector, reciba dos índices, *a* y *b*, que señalen dónde empieza y dónde acaba el fragmento de vector a considerar.

```
int buscaBin( TElem v[], TElem x, int a, int b )
```

de esta forma, la función que realmente nos interesa se obtiene como

```
buscaBin( v, num, x ) = buscaBin( v, x, 0, num-1)
```

Nótese que no es necesario inventarse otro nombre para la función auxiliar porque C++ permite la sobrecarga de funciones: definir varias funciones con el mismo nombre que se distinguen por los parámetros.

- La función recursiva es por tanto la función auxiliar, mientras que en la implementación de la función original nos limitaremos a realizar la llamada inicial a la función auxiliar.

```
int buscaBin( TElem v[], TElem x, int a, int b ) {

// cuerpo de la función

}

int buscaBin( TElem v[], int num, TElem x ) {

  return buscaBin( v, x, 0, num-1);
}
```

Nótese que es necesario escribir primero la función auxiliar para que sea visible desde la otra función.

- Diseño del algoritmo

(R.1) (R.2) Planteamiento recursivo y análisis de casos.

Aunque el planteamiento recursivo está claro: dados *a* y *b*, obtenemos el punto medio *m* y

– si $v[m] \leq x$ seguimos buscando en $m+1$ .. b
– si $v[m] > x$ seguimos buscando en a .. $m-1$,

es necesario ser cuidadoso con los índices. La idea consiste en garantizar que en todo momento se cumple que:

– todos los elementos a la izquierda de *a* –sin incluir *v[a]*– son menores o iguales que *x*, y
– todos los elementos a la derecha de *b* –sin incluir *v[b]*– son estrictamente mayores que *x*.

Una primera idea puede ser considerar como caso base *a* == *b*. Si lo hiciésemos así, la solución en el caso base quedaría:

```
if ( a == b )
  if      ( v[a] == x ) p = a;
  else if ( v[a] <  x ) p = a;    // x no está en v
  else if ( v[a] >  x ) p = a-1;  // x no está en v
```

Sin embargo, también es necesario considerar el caso base *a* == *b+1* pues puede ocurrir que en ninguna llamada recursiva se cumpla *a* == *b*. Por ejemplo, en un situación como esta

```
x == 8    a == 0    b == 1    v[0] == 10    v[1] == 15
```

el punto medio $m=(a+b)/2$ es 0, para el cual se cumple $v[m] > x$ y por lo tanto la siguiente llamada recursiva se hace con

```
a == 0    b == -1
```

que es un caso base donde debemos devolver –1 y donde para alcanzarlo no hemos pasado por *a* == *b*.

Como veremos a continuación, el caso *a*==*b* se puede incluir dentro del caso recursivo si consideramos como caso base el que cumple *a* == *b+1*, que además tiene una solución más sencilla y que siempre se alcanza.

Con todo lo anterior, la especificación de la función recursiva auxiliar queda:

```
int buscaBin( TElem v[], TElem x, int a, int b ) {
// Pre: v está ordenado entre 0 .. num-1
//      ( 0 <= a <= num ) && ( -1 <= b <= num-1 ) && ( a <=
b+1 )
//      todos los elementos a la izquierda de 'a' son <= x
//      todos los elementos a la derecha de 'b' son > x

  int p;
// cuerpo de la función
  return p;
// Post: devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//       v[i] <= x
//       si x es menor que todos los elementos de v, devuelve
-1
 }
```

Donde se tiene la siguiente distinción de casos

$d(v,x,a,b) : a == b+1$

$\neg d(v,x,a,b) : a <= b$

para la que efectivamente se cumple

$P_0 \Rightarrow a \leq b+1 \Rightarrow a == b+1 \vee a \leq b$

(R.3) Solución en el caso directo.

$\{ P_0 \wedge a == b+1 \}$
A_1
$\{ Q_0 \}$

Si

– todos los elementos a la izquierda de *a* son ≤ *x*,
– todos los elementos a la derecha de *b* son > *x*, y
– *a* == *b+1*, es decir, *a* y *b* se han cruzado,

entonces el último elemento que cumple que es ≤ *x* es *v[a–1]*, y por lo tanto,

$A_1 \equiv p = a-1;$

(R.4) Descomposición recursiva.

Los parámetros de la llamada recursiva dependerán del resultado de comparar x con la componente central del fragmento de vector que va desde a hasta b. Por lo tanto, obtenemos el punto medio

```
m = (a+b) / 2;
```

de forma que

- si $x < v[m]$ la descomposición es

  s_1(v, x, a, b) = (v, x, a, m–1)

- si $x \geq v[m]$ la descomposición es

  s_2(v, x, a, b) = (v, x, m+1, b)

(R.5) Función de acotación y terminación.

Lo que va a ir disminuyendo, según avanza la recursión, es la longitud del subvector a considerar, por lo tanto tomamos como función de acotación:

t(v, x, a, b) = b–a+1

y comprobamos

$a \leq b \wedge a \leq m \leq b \Rightarrow t(s_1(\vec{x})) < t(\vec{x}) \wedge t(s_2(\vec{x})) < t(\vec{x})$

que efectivamente se cumple, ya que

$a \leq b \Rightarrow a \leq (a+b)/2 \leq b$

$(m-1)-a+1 < b-a+1 \Leftarrow m-1 < b \Leftarrow m \leq b$

$b-(m+1)+1 < b-a+1 \Leftrightarrow b-m-1 < b-a \Leftarrow b-m \leq b-a \Leftrightarrow m \geq a$

(R.6) Llamada recursiva.

La solución que hemos obtenido, sólo funciona si en cada llamada se cumple la precondición, por lo tanto, debemos demostrar que de una llamada a la siguiente efectivamente se sigue cumpliendo la precondición.

Tratamos por separado cada caso recursivo

— $x < v[m]$

$P_0 \wedge a \leq b \wedge a \leq m \leq b \wedge x < v[m] \Rightarrow P_0[b/m-1)]$

Que es cierto porque:

```
   v está ordenado entre 0 .. num-1
⇒ v está ordenado entre 0 .. num-1

   0 ≤ a ≤ num
⇒ 0 ≤ a ≤ num

   -1 ≤ b ≤ num-1 ∧ a ≤ m ≤ b ∧ 0 ≤ a ≤ num
⇒ -1 ≤ m-1 ≤ num-1

   a ≤ m
⇒ a ≤ m-1+1

   todos los elementos a la izquierda de 'a' son <= x
⇒ todos los elementos a la izquierda de 'a' son <= x

   v está ordenado entre 0 .. num-1 ∧
   todos los elementos a la derecha de 'b' son > x ∧
   m ≤ b ∧ x < v[m]
⇒ todos los elementos a la derecha de 'm-1' son > x
```

— $x \geq v[m]$

```
   v está ordenado entre 0 .. num-1
   ( 0 <= a <= num ) && ( -1 <= b <= num-1 ) && ( a <= b+1
)
   todos los elementos a la izquierda de 'a' son <= x
   todos los elementos a la derecha de 'b' son > x
⇒ v está ordenado entre 0 .. num-1
   ( 0 <= m+1 <= num ) && ( -1 <= b <= num-1 ) && ( m+1 <=
b+1 )
   todos los elementos a la izquierda de 'm+1' son <= x
   todos los elementos a la derecha de 'b' son > x
```

Se razona de forma similar al anterior.

Debemos razonar también que la llamada inicial a la función auxiliar cumple la precondición

```
    v está ordenado entre 0 .. num-1 ∧
    a == 0 ∧ b == num-1
⇒ v está ordenado entre 0 .. num-1
    ( 0 <= a <= num ) && ( -1 <= b <= num-1 ) && ( a <= b+1
)
    todos los elementos a la izquierda de 'a' son <= x
    todos los elementos a la derecha de 'b' son > x
```

Que no es cierto si *num* < *0* ya que en ese caso no se cumple *a* ≤ *b+1*. De hecho si *num* < *0* no está claro qué resultado se debe devolver, por lo tanto lo mejor es añadir esta restricción a la precondición de la función original.

Y con esta nueva condición (*num* ≥ *0*) sí es sencillo demostrar la corrección de la llamada inicial:

```
   a == 0 ∧ b == num-1 ∧ num ≥ 0
⇒ 0 ≤ a ≤ num ∧ -1 ≤ b ≤ num-1 ∧ a ≤ b+1

   a == 0
⇒ todos los elementos a la izquierda de 'a' son <= x

   a == num-1
⇒ todos los elementos a la derecha de 'b' son > x
```

(R.7) Función de combinación.

En los dos casos recursivos nos limitamos a propagar el resultado de la llamada recursiva:

```
p = p'
```

(R.9) Escritura de la llamada recursiva.

Cada una de las dos llamadas recursivas se puede escribir como una sola asignación:

```
p = buscaBin( v, x, m+1, b )
```

y

```
p = buscaBin( v, x, a, m–1 )
```

- Con lo que finalmente la función queda:

```
int buscaBin( TElem v[], TElem x, int a, int b ) {
// Pre: v está ordenado entre 0 .. num-1
//      ( 0 <= a <= num ) && ( -1 <= b <= num-1 ) && ( a <=
b+1 )
//      todos los elementos a la izquierda de 'a' son <= x
//      todos los elementos a la derecha de 'b' son > x

  int p, m;

  if ( a == b+1 )
    p = a - 1;
  else if ( a <= b ) {
    m = (a+b) / 2;
    if ( v[m] <= x )
      p = buscaBin( v, x, m+1, b );
    else
      p = buscaBin( v, x, a, m-1 );
  }
  return p;
// Post: devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//       v[i] <= x
//       si x es menor que todos los elementos de v, devuelve
-1
 }

int buscaBin( TElem v[], int num, TElem x ) {
// Pre: los num primeros elementos de v están ordenados y
//      num >= 0

  return buscaBin(v, x, 0, num-1);

// Post : devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//        v[i] <= x
//        si x es menor que todos los elementos de v,
devuelve -1
}
```

Nótese que la generalización está pensada como una función auxiliar y no para ser utilizada por sí sola, en cuyo caso deberíamos repensar la precondición y la postcondición.

2.2.1 Algoritmos avanzados de ordenación

- La ordenación rápida (*quicksort*) y la ordenación por mezcla (*mergesort*) son dos algoritmos de ordenación de complejidad cuasi-lineal, $O(n \log n)$.

  Las idea recursiva es similar en los dos algoritmos: para un ordenar un vector se procesan por separado la mitad izquierda y la mitad derecha.

  – En la ordenación rápida, se colocan los elementos pequeños a la izquierda y los grandes a la derecha, y luego se sigue con cada mitad por separado.
  – En la ordenación por mezcla, se ordena la mitad de la izquierda y la mitad de la derecha por separado y luego se mezclan los resultados.

Ordenación rápida

- Especificación:

```
void quickSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0

  quickSort(v, 0, num-1);

// Post: se han ordenado las num primeras posiciones de v
}

void quickSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

// Post: v está ordenado entre a y b
}
```

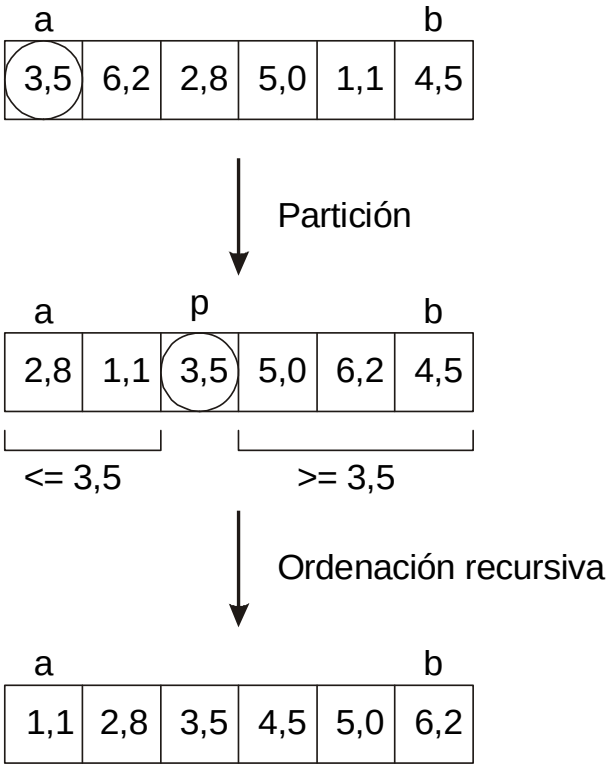
Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros *a* y *b*, para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

- El planteamiento recursivo consiste en:

  1. Elegir un *pivote*: un elemento cualquiera del subvector v[a..b]. Normalmente se elige v[a] como pivote.

  2. *Particionar* el subvector v[a..b], colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha. Al final del proceso de partición, el pivote debe quedar en el *centro*, separando los menores de los mayores.

  3. Ordenar recursivamente los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.



- Análisis de casos
  - Caso directo: $a == b+1$

    El subvector está vacío y, por lo tanto, ordenado.
  - Caso recursivo: $a \leq b$

    Se trata de un segmento no vacío y aplicamos el planteamiento recursivo:
    - considerar $x == v[a]$ como elemento pivote
    - reordenar parcialmente el subvector $v[a..b]$ para conseguir que x quede en la posición p que ocupará cuando $v[a..b]$ esté ordenado.
    - ordenar recursivamente $v[a\ ..\ (p-1)]$ y $v[(p+1)\ ..\ b]$.

- Función de acotación:

  `t(v, a, b) = b – a + 1`

- Suponiendo que tenemos una implementación correcta de *partición*, el algoritmo nos queda:

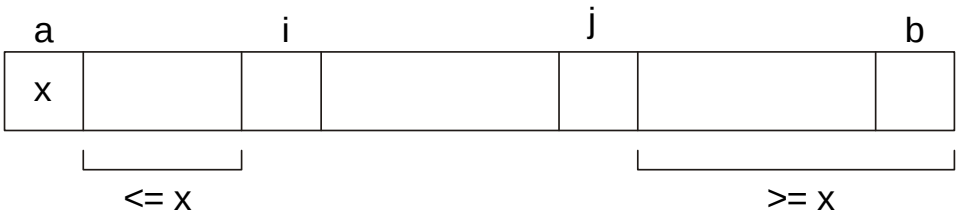
```
void quickSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

   int  p;

   if ( a <= b ) {
     particion(v, a, b, p);
     quickSort(v, a, p-1);
     quickSort(v, p+1, b);
   }

// Post: v está ordenado entre a y b
}
```

- Diseño de partición.

  Partimos de la especificación:

```
{ P: 0 ≤ a ≤ b ≤ num-1 }
  particion
{ Q : 0 ≤ a ≤ p ≤ b ≤ num-1 ∧
      todos los elementos desde 'a' hasta 'p-1' son ≤ v[p] ∧
      todos los elementos desde 'p+1' hasta 'b' son ≥ v[p]
}
```

  La idea es obtener un bucle que mantenga invariante la siguiente situación



  de forma que *i* y *j* se vayan acercando hasta cruzarse, y finalmente intercambiemos *v[a]* con *v[j]*

  – El invariante se obtiene generalizando la postcondición con la introducción de dos variables nuevas, *i*, *j*, que indican el avance por los dos extremos del subvector

```
  a+1 ≤ i ≤ j+1 ≤ b+1 ∧
todos los elementos desde 'a+1' hasta 'i-1' son ≤ v[a] ∧
todos los elementos desde 'j+1' hasta 'b' son ≥ v[a]
```

  – Condición de repetición

    El bucle termina cuando se cruzan los índices *i* y *j*, es decir cuando se cumple *i* == *j+1*, y, por lo tanto, la condición de repetición es

```
i <= j
```

    A la salida del bucle, el vector estará particionado salvo por el pivote *v[a]*. Para terminar el proceso basta con intercambiar los elementos de las posiciones *a* y *j*, quedando la partición en la posición *j*.

```
p = j;
aux = v[a];
v[a] = v[p];
v[p] = aux;
```

  – Expresión de acotación

```
C : j - i + 1
```

– Acción de inicialización

```
i = a+1;
j = b;
```

Esta acción hace trivialmente cierto el invariante porque *v[(a+1)..(i–1)]* y *v[(j+1) .. b]* se convierten en subvectores vacíos.

– Acción de avance

El objetivo del bucle es conseguir que *i* y *j* se vayan acercando, y además se mantenga el invariante en cada iteración. Para ello, se hace un análisis de casos comparando las componentes *v[i]* y *v[j]* con *v[a]*

– $v[i] \leq v[a]$

→ incrementamos *i*

– $v[j] \geq v[a]$

→ decrementamos *j*

– $v[i] > v[a]$ && $v[j] < v[a]$

→ intercambiamos *v[i]* con *v[j]*, incrementamos *i* y decrementamos *j*

De esta forma el avance del bucle queda

```
if      ( v[i] <= v[a] ) i = i + 1;
else if ( v[j] >= v[a] ) j = j - 1;
else if ( (v[i] > v[a]) && (v[j] < v[a]) ) {
  aux = v[i];
  v[i] = v[j];
  v[j] = aux;
   i = i + 1;
   j = j - 1;
}
```

Nótese que las dos primeras condiciones no son excluyentes entre sí pero sí con la tercera, y, por lo tanto, la distinción de casos se puede optimizar teniendo en cuenta esta circunstancia.

Con todo esto el algoritmo queda:

```
void particion ( TElem v[], int a, int b, int & p ) {
// Pre: 0 <= a <= b <= num-1

  int i, j;
  TElem aux;

  i = a+1;
  j = b;
  while ( i <= j ) {
    if ( (v[i] > v[a]) && (v[j] < v[a]) ) {
      aux = v[i]; v[i] = v[j]; v[j] = aux;
      i = i + 1; j = j - 1;
    }
    else {
      if ( v[i] <= v[a] ) i = i + 1;
      if ( v[j] >= v[a] ) j = j – 1;
    }
  }
  p = j;
 aux = v[a]; v[a] = v[p]; v[p] = aux;

// Post: 0 <= a <= p <= b <= num-1 y
//       todos los elementos desde ‘a’ hasta ‘p-1’ son ≤
v[p] y
//       todos los elementos desde ‘p+1’ hasta ‘b’ son ≥
v[p]
}
void quickSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1
   int  p;
   if ( a <= b ) {
     particion(v, a, b, p);
     quickSort(v, a, p–1);
     quickSort(v, p+1, b);
   }
// Post: v está ordenado entre a y b
}

void quickSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0
  quickSort(v, 0, num-1);
// Post: se han ordenado las num primeras posiciones de v
}
```

Ordenación por mezcla

- Partimos de una especificación similar a la del quickSort

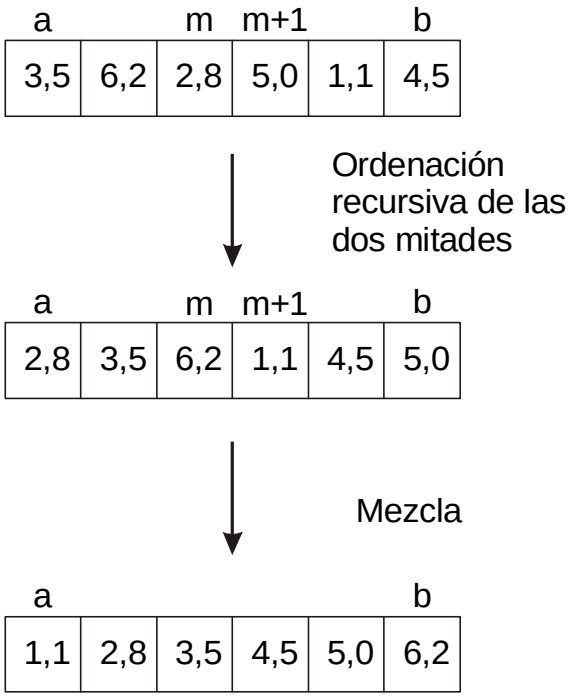
```
void mergeSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0

  mergeSort(v, 0, num-1);

// Post: se han ordenado las num primeras posiciones de v
}

void mergeSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

// Post: v está ordenado entre a y b
}
```

- Planteamiento recursivo.

  Para ordenar el subvector *v[a..b]*

  – Obtenemos el punto medio *m* entre *a* y *b*, y ordenamos recursivamente los subvectores *v[a..m]* y *v[(m+1)..b]*.

  – Mezclamos ordenadamente los subvectores *v[a..m]* y *v[(m+1)..b]* ya ordenados.



- Análisis de casos

  – Caso directo: a ≥ b

  El subvector está vacío o tiene longitud 1 y, por lo tanto, está ordenado.

  $P_0 \wedge a \geq b \Rightarrow a = b \vee a = b+1$

  $Q_0 \wedge (a = b \vee a = b+1) \Rightarrow v = V$

– Caso recursivo: a < b

Tenemos un subvector de longitud mayor o igual que 2, y aplicamos el planteamiento recursivo:

– Dividir *v[a..b]* en dos mitades. Al ser la longitud ≥ 2 es posible hacer la división de forma que cada una de las mitades tendrá una longitud estrictamente menor que el segmento original (por eso hemos considerado como caso directo el subvector de longitud 1).
– Tomando *m = (a+b)/2* ordenamos recursivamente *v[a..m]* y *v[(m+1)..b]*.
– Usamos un procedimiento auxiliar para mezclar las dos mitades, quedando ordenado todo *v[a..b]*.

- Función de acotación.

```
t(v, a, b) = b – a + 1
```

- Suponiendo que tenemos una implementación correcta para *mezcla*, el procedimiento de ordenación queda:

```
void mergeSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

   int  m;

   if ( a < b ) {
     m = (a+b) / 2;
     mergeSort( v, a, m );
     mergeSort( v, m+1, b );
     mezcla( v, a, m, b );
   }

// Post: v está ordenado entre a y b
}
```

- Diseño de mezcla.

El problema es que para conseguir una solución eficiente, $O(n)$, necesitamos utilizar un vector auxiliar donde iremos realizando la mezcla, para luego copiar el resultado al vector original.

El problema es que el tamaño del array auxiliar dependerá del valor de *a* y *b* y en C++ –y en general en cualquier lenguaje de programación– no es posible ubicar un array utilizando variables para indicar el tamaño. Es decir, no es correcta la declaración:

```
void mezcla( TElem v[], int a, int m, int b ) {
  TElem u[b-a+1];
...
}
```

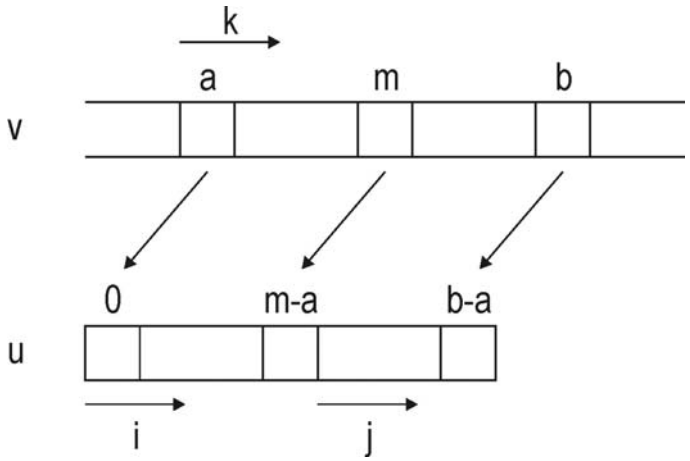
En C++ podemos resolver este problema aprovechándonos de la "dualidad puntero-array": el identificador de un array es un puntero al primer elemento del array[1], lo cual nos permite crear el array auxiliar con la declaración

```
TElem *u = new TElem[b-a+1];
```

De esta forma, *u* se puede tratar como un array de *b–a+1* elementos, aunque al final del procedimiento, deberemos liberar el espacio que ocupa

```
delete[] u;
```

La idea del algoritmo es colocarse al principio de cada subvector e ir tomando, de uno u otro, el menor elemento, y así ir avanzando. Uno de los subvectores se acabará primero y habrá entonces que copiar el resto del otro subvector. En el array auxiliar tendremos los índices desplazados pues mientras el subvector a mezclar es *v[a..b]*, en el array auxiliar tendremos los elementos almacenados en *v[0..b–a]*, y habrá que ser cuidadoso con los índices que recorren ambos arrays.



Con todo esto, el procedimiento de mezcla queda:

```
void mezcla( TElem v[], int a, int m, int b ) {
// Pre: a <= m < b y
//      v está ordenado entre a y m y v está ordenado entre
m+1 y b

  TElem *u = new TElem[b-a+1];
  int i, j, k;

  for ( k = a; k <= b; k++ )
    u[k-a] = v[k];
```

[1] Más sobre esto en el tema 3

```
    i = 0;
    j = m-a+1;
    k = a;
    while ( (i <= m-a) && (j <= b-a) ) {
      if ( u[i] <= u[j] ){
        v[k] = u[i];
        i = i + 1;
      } else {
        v[k] = u[j];
        j = j + 1;
      }
      k = k + 1;
    }
    while ( i <= m-a ) {
      v[k] = u[i];
      i = i+1;
      k = k+1;
    }
    while ( j <= b-a ) {
      v[k] = u[j];
      j = j+1;
      k = k+1;
    }
    delete[] u;
  // Post: v está ordenado entre a y b
  }
```

Cota inferior de la complejidad para los algoritmos de ordenación basados en intercambios

- Dado un algoritmo A de ordenación basado en intercambios, se tienen los siguientes resultados sobre la cota inferior de su complejidad en el caso peor, $T_A(n)$, tomando como tamaño de los datos la longitud del vector:
  (a) $T_A(n) \in \Omega(n \cdot \log n)$
  (b) $T_A(n) \in \Omega(n^2)$, en el caso de que A sólo efectúe intercambios de componentes vecinas.

- Para demostrar (*a*) razonamos sobre el número de comparaciones que es necesario realizar en el caso peor. Según el resultado de cada comparación, realizaremos o no un intercambio.

  Con una comparación podemos generar 2 permutaciones distintas, según si realizamos o no el intercambio; con dos comparaciones podemos generar 2^2 permutaciones, y así sucesivamente, de forma que con t comparaciones podemos obtener 2^t permutaciones distintas.

  En el caso peor, deberemos considerar un número de comparaciones que nos permita alcanzar cualquiera de las $n!$ permutaciones posibles:

  $$2^t \geq n! \Rightarrow t \geq \log(n!) \Rightarrow t \geq c \cdot n \cdot \log n$$

  realizando t comparaciones, la complejidad del algoritmo debe ser

  $$T_A(n) \geq t \geq c \cdot n \cdot \log n \Rightarrow T_A(n) \in \Omega(n \cdot \log n)$$

- Para demostrar (*b*) definimos una medida del *grado de desorden* de un vector, como el número de *inversiones* que contiene

  $$\text{inv}(v, i, j) \Leftrightarrow_{\text{def}} i < j \wedge v[i] > v[j]$$

  un vector estará ordenado si contiene 0 inversiones.

  El caso peor se tendrá cuando el vector esté ordenado en orden inverso, donde el número de inversiones será

  $$(\Sigma\, i : 1 \leq i < n : n - i) \geq c \cdot n^2$$

  Si usamos un algoritmo que sólo realiza intercambios entre componentes vecinas, a lo sumo podrá deshacer una inversión con cada intercambio, y, por lo tanto, en el caso peor realizará un número de intercambios del orden de n^2

  $$T_A(n) \in \Omega(n^2)$$

2.3 Análisis de algoritmos recursivos

2.3.1 Ecuaciones de recurrencias

- La recursión no introduce nuevas instrucciones en el lenguaje, sin embargo, cuando intentamos analizar la complejidad de una función o un procedimiento recursivo nos encontramos con que debemos conocer la complejidad de las llamadas recursivas ...

  La *definición natural* de la función de complejidad de un algoritmo recursivo también es recursiva, y viene dada por una o más *ecuaciones de recurrencia.*

- Cálculo del factorial.

  Tamaño de los datos: n

  Caso directo, $n = 1$: $T(n) = 2$

  Caso recursivo:

  - 1 de evaluar la condición +
  - 1 de evaluar la descomposición *n–1* +
  - 1 de la asignación de *n * fact(n–1)* +
  - $T(n–1)$ de la llamada recursiva.

  Ecuaciones de recurrencia: $$T(n) = \begin{cases} 2 & \text{si } n = 1 \\ 3 + T(n–1) & \text{si } n > 1 \end{cases}$$

- Multiplicación por el método del campesino egipcio.

  Tamaño de los datos: $n = b$

  Caso directo, $n = 0, 1$: $T(n) = 3$

  En ambos casos recursivos:

  - 4 de evaluar todas las condiciones en el caso peor +
  - 1 de la asignación +
  - 2 de evaluar la descomposición *2*a* y *b div 2* +
  - $T(n/2)$ de la llamada recursiva.

  Ecuaciones de recurrencia: $$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 7 + T(n/2) & \text{si } n > 1 \end{cases}$$

- Para calcular el orden de complejidad no nos interesa el valor exacto de las constantes, ni nos preocupa que sean distintas (en los casos directos, o cuando se suma algo constante en los casos recursivos): ¡estudio asintótico!

- Números de fibonacci.
  Tamaño de los datos: n
  Ecuaciones de recurrencia:
  $$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c & \text{si } n > 1 \end{cases}$$

- Ordenación rápida (*quicksort*)
  Tamaño de los datos: $n = num$
  En el caso directo tenemos complejidad constante c_0.
  En el caso recursivo:
  - El coste de la partición: $c \cdot n +$
  - El coste de las dos llamadas recursivas. El problema es que la disminución en el tamaño de los datos depende de los datos y de la elección del pivote.
    - El caso peor se da cuando el pivote no separa nada (es el máximo o el mínimo del subvector): $c_0 + T(n-1)$
    - El caso mejor se da cuando el pivote divide por la mitad: $2 \cdot T(n/2)$

  Ecuaciones de recurrencia en el caso peor:
  $$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n-1) + c \cdot n + c_0 & \text{si } n \geq 1 \end{cases}$$

  Ecuaciones de recurrencia en el caso mejor:
  $$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 \cdot T(n/2) + c \cdot n & \text{si } n \geq 1 \end{cases}$$

  Se puede demostrar que en promedio se comporta como en el caso mejor.

  Cambiando la política de elección del pivote se puede evitar que el caso peor sea un vector ordenado.

2.3.2 Despliegue de recurrencias

- Hasta ahora, lo único que hemos logrado es expresar la función de complejidad mediante ecuaciones recursivas. Pero es necesario encontrar una fórmula explícita que nos permita obtener el orden de complejidad buscado.
- El objetivo de este método es conseguir una *fórmula explícita* de la función de complejidad, a partir de las ecuaciones de recurrencias. El proceso se compone de tres pasos:
  1. Despliegue. Sustituimos las apariciones de T en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas k.
  2. Postulado. A partir de la fórmula paramétrica resultado del paso anterior obtenemos una fórmula explícita. Para ello, se obtiene el valor de k que nos permite alcanzar un caso directo y, en la fórmula paramétrica, se sustituye k por ese valor y la referencia recursiva T por la complejidad del caso directo.
  3. Demostración. La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Podemos comprobarlo demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

Ejemplos

- Factorial.

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 3 + T(n-1) & \text{si } n > 0 \end{cases}$$

  – Despliegue

```
T(n) = 3 + T(n–1) = 3 + 3 + T(n–2) = 3 + 3 + 3 + T(n–3)
     ...
     = 3 · k + T(n–k)
```

  – Postulado

    El caso directo se tiene para $n = 0$

```
n – k = 0 ⇔ k = n
T(n) = 3 n + T(n–n) = 3 n + T(0) = 3 n + 2 = 3 · n – 1
```

    por lo tanto $T(n) \in O(n)$

- Multiplicación por el método del campesino egipcio.

$$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 7 + T(n/2) & \text{si } n > 1 \end{cases}$$

  - Despliegue

```
T(n) = 7 + T(n/2) = 7 + (7 + T(n/2/2)) = 7 + 7 + 7 +
T(n/2/2/2)
     ...
     = 7 · k + T(n/2^k)
```

  - Postulado

    Las llamadas recursivas terminan cuando se alcanza 1

```
n/2^k = 1 ⇔  k = log n

T(n) = 7 · log n + T(n/2^(log n)) = 7 · log n + T(1)
     = 7 · log n + 3
```

    por lo tanto $T(n) \in O(\log n)$

    Si k representa el número de llamadas recursivas ¿qué ocurre cuando $k = \log n$ no tiene solución entera?

    La complejidad $T(n)$ del algoritmo es una función monótona no decreciente, y, por lo tanto, nos basta con estudiar su comportamiento sólo en algunos puntos: los valores de n que son una potencia de 2. Esta simplificación no causa problemas en el cálculo asintótico.

- Números de Fibonacci.

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c_1 & \text{si } n > 1 \end{cases}$$

  Podemos simplificar la resolución de la recurrencia, considerando que lo que nos interesa es una cota superior:

$$T(n) \leq 2 \cdot T(n-1) + c_1 \quad \text{si } n > 1$$

– Despliegue:

$$
\begin{aligned}
T(n) &\leq c_1 + 2 * T(n-1) \\
&\leq c_1 + 2 * (c_1 + 2 * T(n-2)) \\
&\leq c_1 + 2 * (c_1 + 2 * (c_1 + 2 * T(n-3))) \\
&\leq c_1 + 2 * c_1 + 2^2 * c_1 + 2^3 * T(n-3) \\
&\ldots \\
&\leq c_1 \cdot \sum_{i=0}^{k-1} 2^i + 2^k\, T(n-k)
\end{aligned}
$$

– Postulado

Las llamadas recursivas terminan cuando se alcanzan 0 y 1. Debido a la simplificación anterior, consideramos 1.

$$n-k = 1 \Leftrightarrow k = n-1$$

$$
\begin{aligned}
T(n) &\leq c_1 \cdot \sum_{i=0}^{n-1} 2^i + 2^k\, T(n-k) \\
&= c_1 \cdot \sum_{i=0}^{n-1} 2^i + 2^n\, T(n-n+1) \\
&= c_1 \cdot \sum_{i=0}^{n-1} 2^i + 2^n\, T(1) \\
(*) &= c_1 \cdot (2^n - 1) + c_0 \cdot 2^n \\
&= (c_0 + c_1) \cdot 2^n - c_1
\end{aligned}
$$

donde en (*) hemos utilizado la fórmula para la suma de progresiones geométricas:

$$\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1} \qquad r \neq 1$$

Por lo tanto $T(n) \in O(2^n)$

Las funciones recursivas múltiples donde el tamaño del problema se disminuye por sustracción tienen costes prohibitivos, como en este caso donde el coste es exponencial.

2.3.3 Resolución general de recurrencias

- Utilizando la técnica de despliegue de recurrencias y algunos resultados sobre convergencia de series, se pueden obtener unos resultados teóricos para la obtención de fórmulas explícitas, aplicables a un gran número de ecuaciones de recurrencias.

Disminución del tamaño del problema por sustracción

- Cuando: (1) la descomposición recursiva se obtiene restando una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \le n < n_0 \\ a \cdot T(n-b) + c \cdot n^k & \text{si } n \ge n_0 \end{cases}$$

donde:

- c_0 es el coste en el caso directo,
- $a \ge 1$ es el número de llamadas recursivas,
- $b \ge 1$ es la disminución del tamaño de los datos, y
- $c \cdot n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

$$T(n) \in \begin{cases} O(n^{k+1}) & \text{si } a = 1 \\ O(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Vemos que, cuando el tamaño del problema disminuye por sustracción,

- En recursión simple (a=1) el coste es polinómico y viene dado por el producto del coste de cada llamada ($c \cdot n^k$) y el coste lineal de la recursión (n).
- En recursión múltiple (a>1), por muy grande que sea b, el coste siempre es exponencial.

Disminución del tamaño del problema por división

- Cuando: (1) la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \le n < n_0 \\ a \cdot T(n/b) + c \cdot n^k & \text{si } n \ge n_0 \end{cases}$$

donde:

- c_1 es el coste en el caso directo,
- $a \ge 1$ es el número de llamadas recursivas,
- $b \ge 2$ es el factor de disminución del tamaño de los datos, y
- $c \cdot n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

$$T(n) \in \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k \cdot \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Si $a \le b^k$ la complejidad depende de n^k que es el término que proviene de $c \cdot n^k$ en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.

Si $a > b^k$ las mejoras en la eficiencia se pueden conseguir

- disminuyendo el número de llamadas recursivas a o aumentando el factor de disminución del tamaño de los datos b, o bien
- optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir k suficientemente, podemos pasar a uno de los otros casos: $a = b^k$ o incluso $a < b^k$.

Ejemplos

- Suma recursiva de un vector de enteros.

  Tamaño de los datos: $n = num$

  Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n-1) + c & \text{si } n > 0 \end{cases}$$

  Se ajusta al esquema teórico de disminución del tamaño del problema por sustracción, con los parámetros:

  $a = 1, b = 1, k = 0$

  Estamos en el caso $a = 1$, por lo que la complejidad resulta ser:

  $O(n^{k+1}) = O(n)$

- Búsqueda binaria.

  Tamaño de los datos: $n = num$

  Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n/2) + c & \text{si } n > 0 \end{cases}$$

  Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

  $a = 1, b = 2, k = 0$

  Estamos en el caso $a = b^k$ y la complejidad resulta ser:

  $O(n^k \cdot \log n) = O(n^0 \cdot \log n) = O(\log n)$

- Ordenación por mezcla (*mergesort*).

Tamaño de los datos: $n = num$

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1 \\ 2 \cdot T(n/2) + c \cdot n & \text{si } n \geq 2 \end{cases}$$

donde $c \cdot n$ es el coste del procedimiento *mezcla*.

Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

$a = 2, b = 2, k = 1$

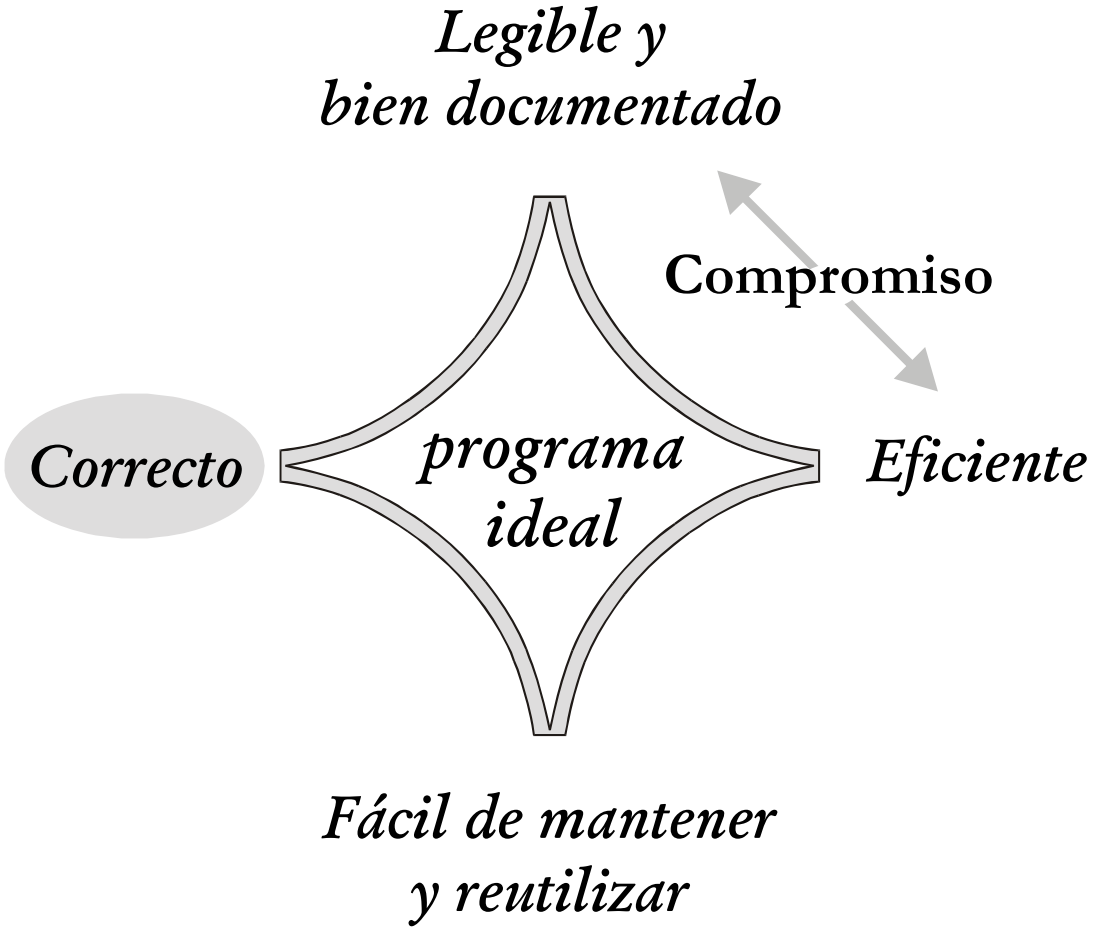
Estamos en el caso $a = b^k$ y la complejidad resulta ser:

$O(n^k \cdot \log n) = O(n \cdot \log n)$

2.4 Transformación de recursivo a iterativo

- En general un algoritmo iterativo es más eficiente que uno recursivo porque la invocación a procedimientos o funciones tiene un cierto coste.

- El inconveniente de transformar los algoritmos recursivos en iterativos radica en que puede ocurrir que el algoritmo iterativo sea menos claro, con lo cual se mejora la eficiencia a costa de perjudicar a la facilidad de mantenimiento.



- En los casos más sencillos (recursión final), ciertos compiladores nos liberan de este compromiso porque son capaces de transformar automáticamente las versiones recursivas que nosotros programamos en versiones iterativas equivalentes (que son las que en realidad se ejecutan).

2.4.1 Transformación de la recursión final

- El esquema general de un procedimiento recursivo final es:

```
void nombreProc ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym ) {
// Precondición
// declaración de constantes
  τ1 x1' ; ... ; τn xn' ;  // x'

  if ( d(x) )
    y = g(x);
  else if ( ¬d(x) ) {
    x' = s(x);
    nombreProc(x', y);
  }
// Postcondición
}
```

- Podemos imaginar la ejecución de una llamada recursiva *nombreProc*$(\vec{x}, \vec{y})$ como un "bucle descendente"

$$\vec{x} \rightarrow \text{nombreProc}(\vec{x}, \vec{y})$$
$$\downarrow$$
$$\text{nombreProc}(s(\vec{x}), \vec{y})$$
$$\downarrow$$
$$\text{nombreProc}(s^2(\vec{x}), \vec{y})$$
$$\downarrow$$
$$\ldots$$
$$\downarrow$$
$$\text{nombreFunc}(s^n(\vec{x}), \vec{y}) \rightarrow \vec{y} = g(s^n(\vec{x}))$$

  En realidad, hay otro "bucle ascendente" que va devolviendo el valor de $\vec{y}$; sin embargo, cuando la recursión es final no se modifica $\vec{y}$ y podemos ignorarlo.

- Existe una traducción directa a la versión iterativa del esquema anterior:

```
void nombreProcItr ( τ1 x1 , … , τn xn , δ1 & y1 , … , δm & ym )
{
// Precondición
// declaración de constantes
  τ1 x1' ; ... ; τn xn' ;  // x⃗'

  x⃗' = x⃗;
  while ( ¬d(x⃗') )
    x⃗' = s(x⃗');
  y⃗ = g(x⃗');
  }
// Postcondición
}
```

– Este paso se puede realizar de forma mecánica.

– Se puede demostrar que la corrección del algoritmo recursivo garantiza la del algoritmo iterativo obtenido de esta manera.

– Si en la versión recursiva se han tenido que añadir parámetros adicionales para permitir la obtención de un planteamiento recursivo, en la versión iterativa se pueden sustituir dichos parámetros por variables locales, inicializadas correctamente.

Ejemplos

- Versión recursiva final del factorial.

```
int acuFact( int a, int n ) {
// Pre: a >= 0 && n >= 0
  int r;
  if      ( n == 0 ) r = a;
  else if ( n >  0 ) r = acuFact( a*n, n-1 );
  return r;
// Post: devuelve a * n!
}

int fact( int n ) {
// Pre: n >= 0

  return acuFact( 1, n );

// Post: devuelve n!
}
```

Para aplicar mecánicamente el esquema, tenemos que ajustarla al esquema general de recursión final, lo que nos obliga a declarar las variables locales *a'* y *n'*. La variable extra *a* se convierte en una variable local inicializada a 1.

```
int fact( int n ) {
// Pre: n >= 0

  int a, r, a', n';

  a = 1;
  a' = a;
  n' = n;
 while ( n' > 0 ) {
    a' = a' * n';
    n' = n' - 1;
  }
  r = a';
  return r;
// Post: devuelve n!
}
```

Si eliminamos las variables innecesarias y sustituimos el bucle *while* por un bucle *for* , obtendremos la versión iterativa habitual de la función factorial.

- Búsqueda binaria.

```
int buscaBin( TElem v[], TElem x, int a, int b ) {

  int p, m;

  if ( a == b+1 )
    p = a - 1;
  else if ( a <= b ) {
    m = (a+b) / 2;
    if ( v[m] <= x )
      p = buscaBin( v, x, m+1, b );
    else
      p = buscaBin( v, x, a, m-1 );
  }
  return p;
}

int buscaBin( TElem v[], int num, TElem x ) {

  return buscaBin(v, x, 0, num-1);
}
```

Aplicando el esquema, incluyendo las variables extra *a* y *b* como variables locales y haciendo algunas simplificaciones obvias obtenemos:

```
int buscaBin( TElem v[], int num, TElem x ) {

   int a, b, p, m;

   a = 0;
   b = num-1;
   while ( a <= b ) {
     m = (a+b) / 2;
     if ( v[m] <= x )
       a = m+1;
     else
       b = m-1;
   }
     p = a - 1;
   return p;
}
```

2.4.2 Transformación de la recursión lineal

- En su formulación más general necesita del uso de una pila, un tipo abstracto de datos que estudiaremos en un tema posterior. En ese punto estudiaremos este tipo de transformación como una aplicación de las pilas.

2.4.3 Trasformación de la recursión múltiple

- En su formulación más general necesita usar árboles generales. Por ello, no trataremos este tipo de transformaciones. Sin embargo, sí nos ocuparemos de un caso especialmente interesante, la implementación iterativa de la ordenación por mezcla, ya que la versión iterativa puede hacerla aún más eficiente.

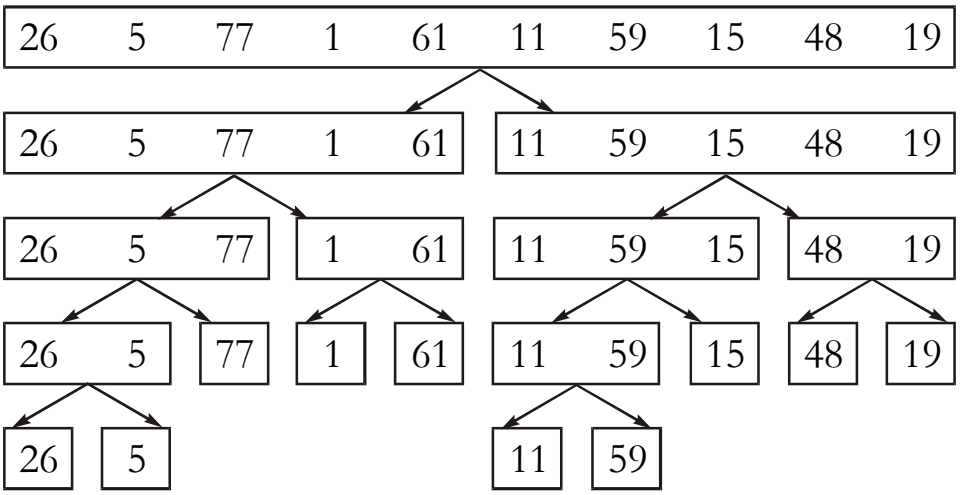
- Versión recursiva.

```
void mergeSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

   int  m;

   if ( a < b ) {
     m = (a+b) / 2;
     mergeSort( v, a, m );
     mergeSort( v, m+1, b );
     mezcla( v, a, m, b );
   }

// Post: v está ordenado entre a y b
}

void mergeSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0

  mergeSort(v, 0, num-1);

// Post: se han ordenado las num primeras posiciones de v
}
```

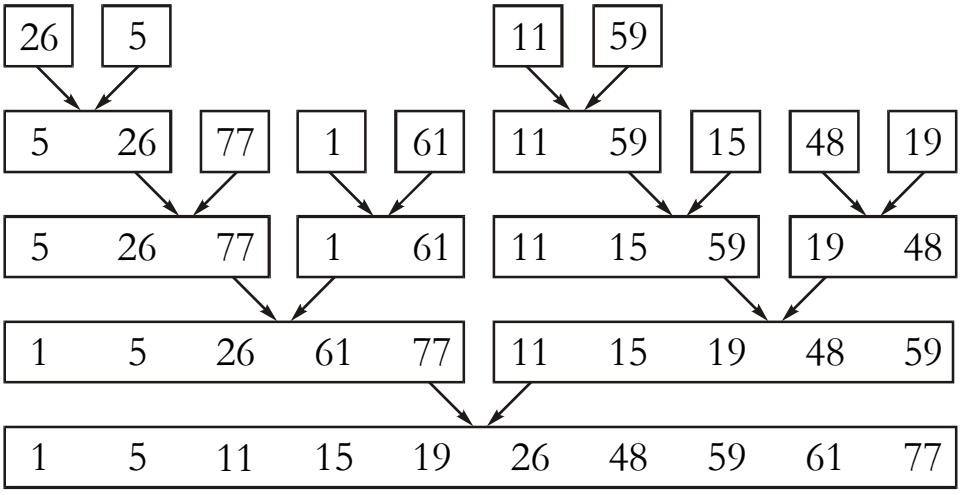
¿Cómo funcionaría una versión iterativa?

- Veamos gráficamente cómo funciona la versión recursiva.

Avance de las llamadas recursivas.

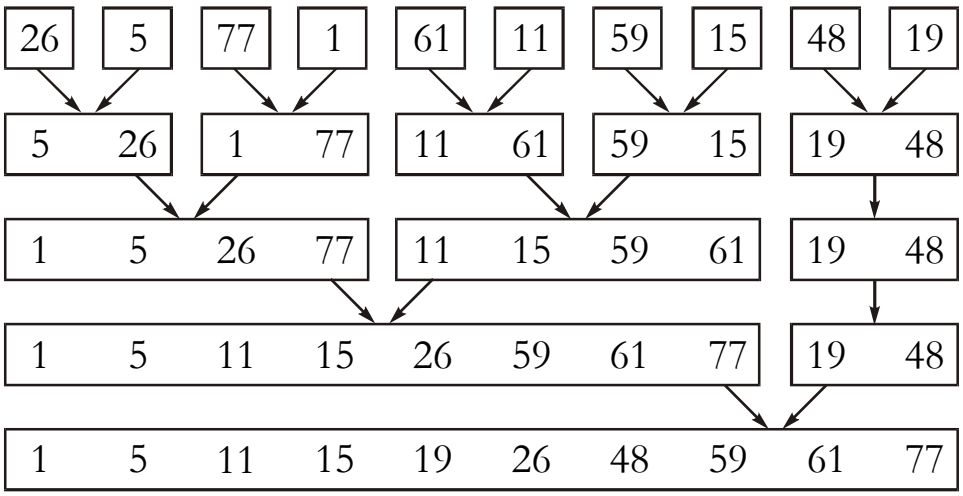


Retroceso de las llamadas recursivas y combinación de los resultados.



¿Cómo funcionaría una versión iterativa?

- La idea de la transformación consiste en obviar la fase de avance de la recursión, y comenzar directamente por los casos base, realizando la combinación de los subvectores mediante la operación de mezcla.



- Versión iterativa.

```
void mergeSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0

   int a, b, l;

   l = 1;
   while ( l < num ) { // no se ha ordenado el subvector entero
     a = 1;
     while ( a+l-1 < num-1 ) { // queda más de un subvector por mez
       b = a + 2*l - 1;
       if ( b > num-1 )
         b = num-1;
       mezcla( v, a, a+l-1, b );
       a = a + 2*l;
     }
     l = 2 * l;
   }

// Post: se han ordenado las num primeras posiciones de v
}
```

2.5 Técnicas de generalización

- En este tema, ya hemos utilizado varias veces este tipo de técnicas (también conocidas como *técnicas de inmersión*), con el objetivo de conseguir planteamientos recursivos.

  La ordenación rápida

```
void quickSort( TElem v[], int a, int b ) {
// Pre: 0 <= a <= num  && -1 <= b <= num-1 && a <= b+1

// Post: v está ordenado entre a y b
}

void quickSort ( TElem v[], int num ) {
// Pre: v tiene al menos num elementos y
//      num >= 0

// Post: se han ordenado las num primeras posiciones de v
}
```

- Además de para conseguir realizar un planteamiento recursivo, las generalizaciones también se utilizan para
  - transformar algoritmos recursivos ya implementados en algoritmos recursivos finales, que se pueden transformar fácilmente en algoritmos iterativos.
  - mejorar la eficiencia de los algoritmos recursivos añadiendo parámetros y/o resultados acumuladores.

  La versión recursiva final del factorial

```
int acuFact( int a, int n ) {
// Pre: a >= 0 && n >= 0

// Post: devuelve a * n!
}

int fact( int n ) {
// Pre: n >= 0

// Post: devuelve n!
}
```

- Decimos que una acción parametrizada (procedimiento o función) *F* es una generalización de otra acción *f* cuando:
  - *F* tiene más parámetros de entrada y/o devuelve más resultados que *f*.

– Particularizando los parámetros de entrada adicionales de F a valores adecuados y/o ignorando los resultados adicionales de F se obtiene el comportamiento de *f*.

En el ejemplo de la ordenación rápida:

– *f* : ***void*** *quickSort (TElem v[ ],* ***int*** *num)*
– *F* : ***void*** *quickSort(TElem v[ ],* ***int*** *a,* ***int*** *b)*
– En *F* se sustituye el parámetro *num* por los parámetros *a* y *b*. Mientras *f* siempre se aplica al intervalo 0 .. *num-1*, *F* se puede aplicar a cualquier subintervalo del array determinado por los índices *a* .. *b*.
– Particularizando los parámetros adicionales de *F* como *a* = 0, *b* = *num-1*, se obtiene el comportamiento de *f*. Razonando sobre las postcondiciones:

```
    v está ordenado entre a y b ∧ a = 0 ∧ b = num-1
⇒ se han ordenado las num primeras posiciones de v
```

En el ejemplo del factorial:

– *f* : ***int*** *fact(* ***int*** *n)*
– *F* : ***int*** *acuFact(* ***int*** *a,* ***int*** *n)*
– En *F* se ha añadido el nuevo parámetro *a* donde se va acumulando el resultado a medida que se construye.
– Particularizando el parámetro adicional de *F* como *a* = *1*, se obtiene el comportamiento de *f*. Razonando sobre las postcondiciones:

```
    devuelve a * n! ∧ a = 1
⇒ devuelve n!
```

Planteamientos recursivos finales

- Dada una especificación E_f pretendemos encontrar una especificación E_F más general que admita una solución recursiva final.

  El resultado se ha de obtener en un caso directo y, para conseguirlo, lo que podemos hacer es añadir nuevos parámetros que vayan acumulando el resultado obtenido hasta el momento, de forma que al llegar al caso directo de *F,* el valor de los *parámetros acumuladores* sea el resultado de *f.*

- Para obtener la especificación E_F a partir de E_f
  - Fortalecemos la precondición de E_f para exigir que alguno de los parámetros de entrada ya traiga calculado una parte del resultado.
  - Mantenemos la misma postcondición.

- Ejemplo: el producto escalar de dos vectores

```
int prodEsc( int u[], int v[], int num ) {
// Pre: 'u' y 'v' contienen al menos 'num' elementos

  int r;

  if      ( num == 0 ) r = 0;
  else if ( num >  0 ) r = v[num-1]*u[num-1] + prodEsc( u,
v, num-1 );
  return r;

// Post: devuelve el sumatorio para i desde 0 hasta 'num-1'
//       de u[i] * v[i], es decir, el producto escalar de u
y v
}
```

Para obtener la precondición de la generalización recursiva final añadimos un nuevo parámetro que acumula la parte del producto escalar calculado hasta el momento por el algoritmo recursivo.

```
int prodEscGen( int u[], int v[], int num, int a ) {
// Pre: 'u' y 'v' contienen al menos 'num' elementos y
//      a es igual al sumatorio para i desde el valor
inicial
//      de 'num-1' hasta 'num' de v[i]*u[i]


// Post: devuelve el sumatorio para i desde 0 hasta 'num-1'
//       de u[i] * v[i], es decir, el producto escalar de u
y v
}
```

Para esta especificación resulta sencillo encontrar un planteamiento recursivo final:

```
int prodEscGen( int u[], int v[], int num, int a ) {
// Pre: 'u' y 'v' contienen al menos 'num' elementos y
//      a es igual al sumatorio para i desde el valor
inicial
//      de 'num-1' hasta 'num' de v[i]*u[i]

  int r;

  if      ( num == 0 )
    r = a;
  else if ( num >  0 )
    r = prodEscGen( u, v, num-1, a + v[num-1]*u[num-1] );
  return r;

// Post: devuelve Σ i : 1 ≤ i ≤ N : u[i] * v[i]
}

int prodEsc( int u[], int v[], int num ) {
// Pre: 'u' y 'v' contienen al menos 'num' elementos

  return prodEscGen(u, v, num, 0);

// Post: devuelve el sumatorio para i desde 0 hasta 'num-1'
//       de u[i] * v[i], es decir, el producto escalar de u
y v
}
```

Donde hemos fijado a 0 el valor inicial del parámetro acumulador.

- El uso de acumuladores es una técnica que tiene especial relevancia en lenguajes en los que sólo se dispone de recursión. Esta es la forma de conseguir funciones recursivas eficientes.

  En programación imperativa tiene menos sentido pues, en muchos casos, resulta más natural obtener directamente la solución iterativa.

2.5.1 Generalización por razones de eficiencia

- Suponemos ahora que partimos de un algoritmo recursivo ya implementado y que nos proponemos mejorar su eficiencia introduciendo parámetros y/o resultados adicionales.

  Se trata de simplificar algún cálculo auxiliar, sacando provecho del resultado obtenido para ese cálculo en otra llamada recursiva. Introducimos parámetros adicionales, o resultados adicionales, según si queremos aprovechar los cálculos realizados en llamadas anteriores, o posteriores, respectivamente. En ocasiones, puede interesar añadir tanto parámetros como resultados adicionales.

Generalización con parámetros acumuladores

- Se aplica esta técnica cuando en un algoritmo recursivo *f* se detecta una expresión $e(\vec{x})$, que sólo depende de los parámetros de entrada, cuyo cálculo se puede simplificar utilizando el valor de esa expresión en *anteriores* llamadas recursivas.

- Se construye una generalización *F* que posee parámetros de entrada adicionales $\vec{a}$, cuya función es transmitir el valor de $e(\vec{x})$. La precondición de *F* se plantea como un fortalecimiento de la precondición de *f*

  $$P'(\vec{a}, \vec{x}) \Leftrightarrow P(\vec{x}) \wedge \vec{a} = e(\vec{x})$$

  mientras que la postcondición se mantiene constante

  $$Q'(\vec{a}, \vec{x}, \vec{y}) \Leftrightarrow Q(\vec{x}, \vec{y})$$

- El algoritmo *F* se obtiene a partir de *f* del siguiente modo:
  - Se reutiliza el texto de *f*, reemplazando $e(\vec{x})$ por $\vec{a}$
  - Se diseña una nueva función sucesor $s'(\vec{a}, \vec{x})$, a partir de la original $s(\vec{x})$, de modo que se cumpla:

    $\{ \vec{a} = e(\vec{x}) \wedge \neg d(\vec{x}) \}$
    $(\vec{a}', \vec{x}') = s'(\vec{a}, \vec{x})$
    $\{ \vec{x}' = s(\vec{x}) \wedge \vec{a}' = e(\vec{x}') \}$

  La técnica resultará rentable cuando en el cálculo de $\vec{a}'$ nos podamos aprovechar de los valores de $\vec{a}$ y $\vec{x}$ para realizar un cálculo más eficiente.

- Ejemplo: función que calcula cuántas componentes de un vector son iguales a la suma de las componentes situadas a su derecha:

```
int numCortesDer( int v[], int num ) {
// Pre: v contiene al menos num elementos

// Post: devuelve el número de posiciones entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}
```

Para obtener un planteamiento recursivo, es necesario generalizar esta especificación, añadiendo un parámetro que nos indique en qué posición del array nos encontramos.

Hasta ahora no había sido necesario realizar este tipo de generalizaciones en algoritmos recursivos sobre vectores porque utilizábamos el parámetro *num* para indicar el avance por el array. En este caso, sin embargo, debemos mantener *num* pues en cada llamada recursiva se hace necesario obtener la suma desde la posición actual hasta *num–1*.

```
int numCortesDerGen( int v[], int num, int i ) {
// Pre: v contiene al menos num elementos
//      0 <= i <= num-1

// Post: devuelve el número de posiciones entre 0 e i
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}
```

La implementación de esta generalización

```
int numCortesDerGen( int v[], int num, int i ) {
// Pre: v contiene al menos num elementos
//      0 <= i <= num-1

  int r, s, j;

  if ( i < 0 )
    r = 0;
  else if ( i >= 0 ) {
    s = 0;
    for ( j = i+1; j < num; j++ )
      s = s + v[j]
    if ( s == v[i] )
      r = numCortesDerGen( v, num, i-1 ) + 1;
    else
      r = numCortesDerGen( v, num, i-1 );
  }
  return r;
// Post: devuelve el número de posiciones entre 0 e i
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}

int numCortesDer( int v[], int num ) {
// Pre: v contiene al menos num elementos

  return numCortesDerGen( v, num, num-1);

// Post: devuelve el número de posiciones entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}
```

Observamos que este algoritmo de complejidad $O(n^2)$ ($n = num$) se podría optimizar si en el cálculo de la suma de las componentes s utilizásemos el resultado de la llamada anterior. Es decir, en este caso queremos optimizar el cálculo de la expresión $s(i, v) = \Sigma j : i < j < num : v[j]$.

Para obtener la nueva generalización:

– Eliminamos el parámetro *i* porque ahora podemos utilizar *num* para indicar la posición actual dentro del array.
– Añadimos un nuevo parámetro *s* donde se recibirá la suma de las componentes hasta *num*, y fortalecemos la precondición para incluir las condiciones sobre el nuevo parámetro

```
P'(v, num, s) : s es igual a la suma desde num hasta el
   valor inicial
                de num-1
```

La implementación de esta nueva generalización:

```
int numCortesDerGen( int v[], int num, int s ) {
// Pre: v contiene al menos num elementos
//      s es igual a la suma desde num hasta el valor
   inicial
//      de num-1
  int r;
  if ( num == 0 )
    r = 0;
  else if ( num > 0 ) {
    if ( s == v[num-1] )
      r = numCortesDerGen( v, num-1, s+v[num-1] ) + 1;
    else
      r = numCortesDerGen( v, num-1, s+v[num-1] );
  }
  return r;
// Post: devuelve el número de posiciones entre 0 y num
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha, hasta el valor inicial de
num-1.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}

int numCortesDer( int v[], int num ) {
// Pre: v contiene al menos num elementos
  return numCortesDerGen( v, num, 0);
// Post: devuelve el número de posiciones entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       situadas a su derecha.
//       Se considera que la suma a la derecha de v[num-1]
es 0
}
```

Una vez que hemos conseguido mejorar la complejidad –ahora es de *O(n)*–, es inmediato obtener otra generalización, añadiendo un parámetro más, que convierta esta función en recursiva final.

Generalización con resultados acumuladores

- Se aplica esta técnica cuando en un algoritmo recursivo *f* se detecta una expresión $e(\vec{x}', \vec{y}')$, que puede depender de los parámetros de entrada y los resultados de la llamada recursiva, cuyo cálculo se puede simplificar utilizando el valor de esa expresión en *posteriores* llamadas recursivas. Obviamente, si la expresión depende de los resultados de la llamada recursiva, debe aparecer después de dicha llamada.

  Se construye una generalización *F* que posee resultados adicionales $\vec{b}$, cuya función es transmitir el valor de $e(\vec{x}, \vec{y})$. La precondición de *F* se mantiene constante

  $P'(\vec{x}) \Leftrightarrow P(\vec{x})$

  mientras que la postcondición de *F* se plantea como un fortalecimiento de la postcondición de *f*

  $Q'(\vec{x}, \vec{b}, \vec{y}) \Leftrightarrow Q(\vec{x}, \vec{y}) \wedge \vec{b} = e(\vec{x}, \vec{y})$

- El algoritmo *F* se obtiene a partir de *f* del siguiente modo:
  - Se reutiliza el texto de *f*, reemplazando $e(\vec{x}', \vec{y}')$ por $\vec{b}'$
  - Se añade el cálculo de $\vec{b}$, de manera que la parte $\vec{b} = e(\vec{y}, \vec{x})$ de la postcondición $Q'(\vec{x}, \vec{b}, \vec{y})$ quede garantizada, tanto en los casos directos como en los recursivos.

  La técnica resultará rentable siempre que *F* sea más eficiente que *f*.

- Ejemplo: cuántas componentes de un vector son iguales a la suma de las componentes que la preceden.

```
int numCortesIzq( int v[], int num ) {
// Pre: v contiene al menos num elementos

// Post: devuelve el número de posiciones i entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden.
//       Se considera que la suma que precede a v[0] es 0
}
```

Cuya implementación:

```
int numCortesIzq( int v[], int num ) {
// Pre: v contiene al menos num elementos

  int r, s, j;

  if ( num == 0 )
    r = 0;
  else if ( num > 0 ) {
    s = 0;
    for ( j = 0; j < num-1; j++ )
      s = s + v[j];
    if ( s == v[num-1] )
      r = numCortesIzq( v, num-1 ) + 1;
    else
      r = numCortesIzq( v, num-1 );
  }
  return r;

// Post: devuelve el número de posiciones i entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden.
//       Se considera que la suma que precede a v[0] es 0
}
```

Observamos que este algoritmo de complejidad $O(n^2)$ ($n = num$) se podría optimizar si en el cálculo de la suma de las componentes *s* utilizásemos el resultado obtenido en la llamada recursiva. Es decir, en este caso queremos optimizar el cálculo de la expresión $s(e, v) = \Sigma j : 0 \leq i \leq N : v[i]$. Para ello:

– Añadimos un nuevo resultado *s* donde se devolverá la suma de las componentes situadas a izquierda de *num*. Como se devuelven dos resultados, debemos convertir la función en un procedimiento.
– Fortalecemos la postcondición para incluir las condiciones sobre el nuevo resultado

```
Q'(v,num, r, s): r es igual el número de posiciones i entre
0 y num-1
       que cumplen que su valor es igual a la suma de las
componentes
       que le preceden y
       s es igual a la suma de las componentes de v desde 0
hasta num-1
```

Y la implementación de la generalización:

```
void numCortesIzqGen( int v[], int num, int& r, int& s ) {
// Pre: v contiene al menos num elementos
```

```
  if ( num == 0 ) {
    r = 0;
    s = 0;
  }
  else if ( num > 0 ) {
    numCortesIzqGen( v, num-1, r, s );
    if ( s == v[num-1] )
      r = r + 1;
    s = s + v[num-1];
  }

// Post: r es igual el número de posiciones i entre 0 y
num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden y
//     s es igual a la suma de las componentes de v desde 0
hasta num-1
//       Se considera que la suma que precede a v[0] es 0
}

int numCortesIzq( int v[], int num ) {
// Pre: v contiene al menos num elementos

  int r, s;

  numCortesIzqGen( v, num, r, s );

  return r;

// Post: devuelve el número de posiciones i entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden.
//       Se considera que la suma que precede a v[0] es 0
}
```

Hemos conseguido pasar de complejidad cuadrática a complejidad lineal.

¿Se te ocurre cómo se podría conseguir una versión recursiva final del algoritmo?

- Para conseguir que sea recursivo final es necesario:
  - añadir un parámetro acumulador donde se almacene el número de posiciones que cumplen la condición, y

– recorrer el array de izquierda a derecha en lugar de hacerlo de derecha a izquierda, y, para ello, necesitamos añadir un nuevo parámetro que controle la posición del array por la que vamos.

De esta forma:

```
int numCortesIzqGen( int v[], int num, int a, int s, int i
) {
// Pre: v contiene al menos num elementos
//      0 <= i <= num y s es igual a la suma desde 0 hasta
i-1 y
//      a acumula el número de componentes desde 0 hasta i-
1 que
//      cumplen la condición

  int r;

  if ( i == num )
    r = a;
  else if ( i < num ) {
    if ( s == v[i] )
      r = numCortesIzqGen( v, num, a+1, s+v[i], i+1 );
    else
      r = numCortesIzqGen( v, num, a, s+v[i], i+1 );
  }
  return r;

// Post: r es igual el número de posiciones i entre 0 y
num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden
}

int numCortesIzq( int v[], int num ) {
// Pre: v contiene al menos num elementos

  return numCortesIzqGen( v, num, 0, 0, 0 );

// Post: devuelve el número de posiciones i entre 0 y num-1
//       que cumplen que su valor es igual a la suma de las
componentes
//       que le preceden.
//       Se considera que la suma que precede a v[0] es 0
}
```