Capítulo 4

Diseño de algoritmos recursivos[1]

Para entender la recursividad
primero hay que entender la recursividad
Anónimo

Resumen: En este tema se presenta el mecanismo de la recursividad como una manera de diseñar algoritmos fiables y fáciles de entender. Se introducen distintas técnicas para diseñar algoritmos recursivos, analizar su complejidad, modificarlos para mejorar esta complejidad y verificar su corrección.

1. Introducción

⋆ La recursión aparece de forma natural en programación, tanto en la definición de tipos de datos (como veremos en temas posteriores) como en el diseño de algoritmos.

⋆ Hay lenguajes (funcionales y lógicos) en los que no hay iteración (no hay bucles), sólo hay recursión. En C++ tenemos la opción de elegir entre iteración y 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.

[1] Gonzalo Méndez es el autor principal de este tema.

⋆ Intuitivamente, el subprograma se invoca a sí mismo con datos cada vez más simples hasta que son tan simples que la solución es inmediata. Posteriormente, la solución se puede ir elaborando hasta obtener la solución para los datos iniciales.

⋆ Tres elementos básicos en la recursión:

- Autoinvocación: el proceso se llama a sí mismo.
- Caso directo: el proceso de autoinvocación eventualmente alcanza una solución directa (sin invocarse a sí mismo) al problema.
- Combinación: los resultados de la llamada precedente son utilizados para obtener una solución, posiblemente combinados con otros datos.

⋆ 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:

```
{n ≥ 0}
fun factorial (int n) return int r
{r = n!}

int factorial ( int n )
{
  int r;

  if  ( n == 0 ) r = 1;
  else   // n > 0
      r = n * factorial(n-1);
  return r;
}
```

¿Cómo se ejecuta la llamada *factorial(3)*?

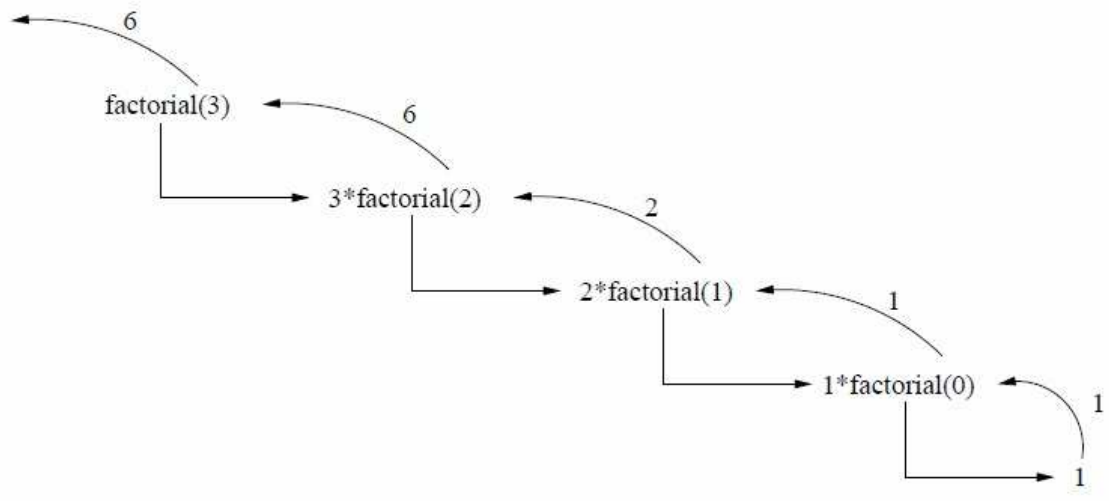

Figura 1: Ejecución de *factorial(3)*

¿Y qué ocurre en memoria?

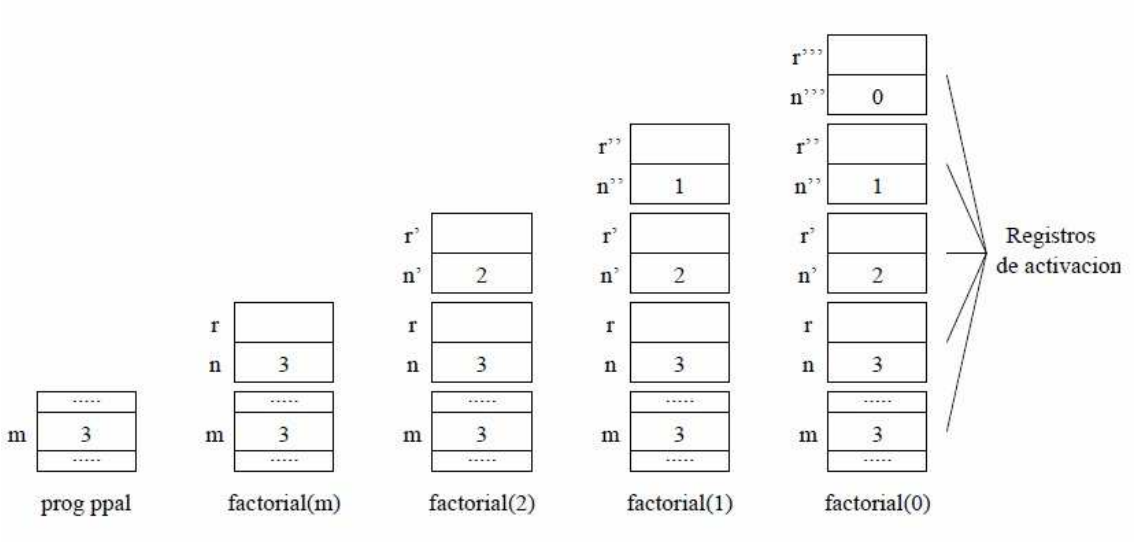

Figura 2: Llamadas recursivas de *factorial(3)*

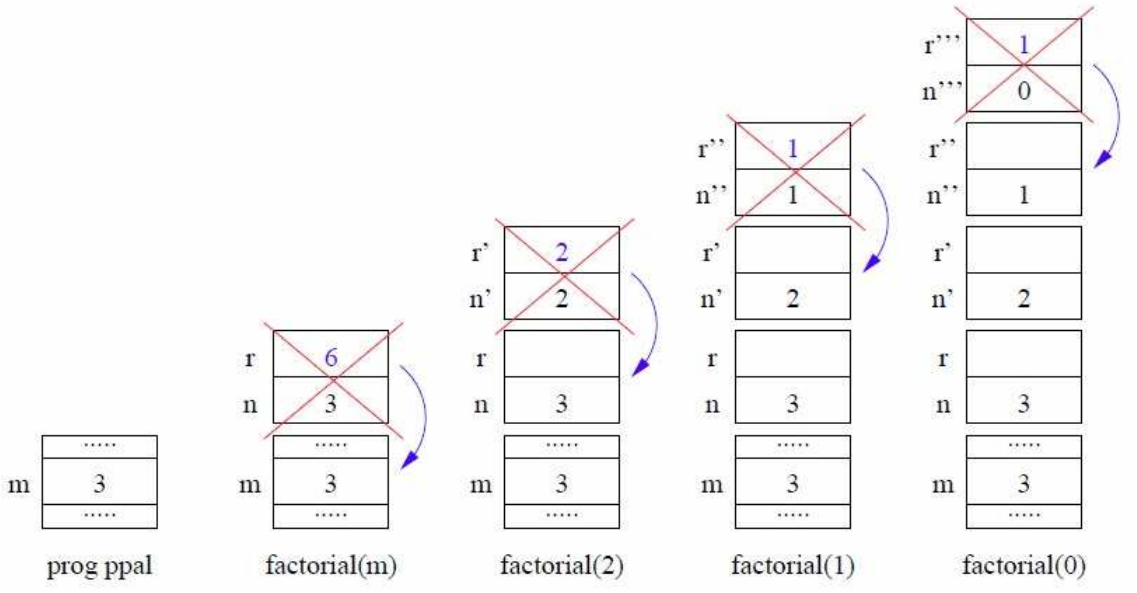

Figura 3: Vuelta de las llamadas recursivas de *factorial(3)*

⋆ ¿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.

1.1. Recursión simple

⋆ Una acción recursiva tiene recursión simple (o lineal) si cada caso recursivo realiza exactamente una llamada recursiva. Puede describirse mediante el esquema general:

```
void nombreProc ( τ1x1 , ... , τnxn , δ1 & y1 , ... , δm & ym )
{
// P: Precondición
// declaración de constantes

   τ1x'1 ; ... ; τnx'n ; // x', parámetros de la llamada recursiva
   δ1y'1 ; ... ; δmy'm ; // y', resultados de la llamada recursiva

  if ( d(x) )          // caso directo
    y = g(x);          // solución al caso directo
  else if ( r(x) )      // caso no directo
  {
    x' = s(x);         // función sucesor: descomposición de datos
    nombreProc(x', y');// llamada recursiva
    y = c(x, y'); // función de combinación: composición de solución
  }

// Q: 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
- $r(\vec{x})$ 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 $\vec{y}$.

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

```
{n ≥ 0}
fun factorial (int n) return int r
{r = n!}

int factorial ( int n )
{
  int r;

  if ( n == 0 ) r = 1;
  else   // n > 0
      r = n * factorial(n-1);
  return r;
}
```

$d(n) = (n == 0)$
$g(n) = 1$

$r(n) = (n > 0)$
$s(n) = n - 1$
$c(n, fact(s(n))) = n * fact(s(n))$

⋆ El esquema de recursión simple puede ampliarse considerando varios casos directos y también varias descomposiciones para el caso recursivo. d($\vec{x}$) y r($\vec{x}$) pueden desdoblarse en una alternativa con varios casos. Lo importante es que las alternativas sean **exhaustivas** y **excluyentes**, y que en cada caso sólo se ejecute una llamada recursiva.

⋆ Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*.

$\{(a \geq 0) \wedge (b \geq 0)\}$
`fun prod (int a, int b) return int r`
$\{r = a * b\}$

```
int prod ( int a, int b )
{

  int r;

  if       ( b == 0 )
         {r = 0;}
  else if ( b == 1 )
         {r = a;}
  else if (b % 2 == 0)           // b > 1
         {r = prod(2*a, b/2);}
  else                          // b > 1 && (b %2 == 1)
         {r = prod(2*a, b/2) + a;}

  return r;
}
```

$d_1(a,b) = (b == 0)$ | $d_2(a,b) = (b == 1)$
$g_1(a,b) = 0$ | $g_2(a,b) = 1$
$r_1(a,b) = ((b > 1)\ \&\&\ par(b))$ | $r_2(a,b) = ((b > 1)\ \&\&\ impar(b))$
$s_1(a,b) = (2 * a, b/2)$ | $s_2(a,b) = (2 * a, b/2)$
$c_1(a,b,prod(s_1(a,b))) = prod(s_1(a,b))$ | $c_2(a,b,prod(s_2(a,b))) = prod(s_2(a,b)) + a$

1.2. Recursión final

⋆ La recursión final o de cola (*tail recursion*) es un caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada recursiva. 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.

```
void nombreProcItr ( τ1x1 , ... , τnxn , δ1 & y1 , ... , δm & ym ) {
// Precondición
// declaración de constantes
  τ1x'1 ; ... ; τnx'n ; // x⃗'

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

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

```
{(a > 0) ∧ (b > 0)}
fun mcd (int a, int b) return int r
{r = mcd(a, b)}

int mcd( int a, int b )
{
  int m;

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

que se ajusta al esquema de recursión simple:

$$\begin{array}{ll} d(a,b) = (a == b) & g(a,b) = a \\ r_1(a,b) = (a > b) & r_2(a,b) = (a < b) \\ s_1(a,b) = (a-b, a) & s_2(a,b) = (a, b-a) \end{array}$$

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

$c_1(a, b, mcd(s_1(a,b))) = mcd(s_1(a,b))$
$c_2(a, b, mcd(s_2(a,b))) = mcd(s_2(a,b))$

Si traducimos esta versión recursiva a una iterativa del algoritmo obtenemos:

```
int itmcd( int a, int b )
{
  int auxa =a; int auxb=b;
  while (auxa!=auxb)
    {
     if (auxa>auxb)
        {auxa = auxa-auxb;}
     else
```

```
            {auxb = auxb-auxa;}
        };
     return auxa;
}
```

1.3. Recursión múltiple

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

```
void nombreProc ( τ1x1 , ... , τnxn , δ1 & y1 , ... , δm & ym ) {
// Precondición
// declaración de constantes
  τ1x11 ; ... ; τnx1n ; ... ; τ1xk1 ; ... ; τnxkn ;
  δ1y11 ; ... ; δmy1m ; ... ; δ1yk1 ; ... ; δmykm ;

  if ( d(x⃗) )
     y⃗ = g(x⃗);
  else if ( r(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 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
- $r(\vec{x})$ 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:

```
{n ≥ 0}
fun fib (int n) return int r
{r = fib(n)}
```

```
int fib( int n )
{
  int r;

  if      ( n == 0 ) r = 0;
  else if ( n == 1 ) r = 1;
  else    // n >  1
       r = fib(n-1) + fib(n-2);

  return r;
}
```

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

$$
\begin{array}{ll}
d_1(n) = (n == 0) & d_2(n) = (n == 1) \\
g(0) == 0 & g(1) == 1 \\
r(n) = (n > 1) & \\
s_1(n) = n - 1 & s_2(n) = n - 2 \\
c(n, fib(s_1(n)), fib(s_2(n))) & = fib(s_1(n)) + fib(s_2(n))
\end{array}
$$

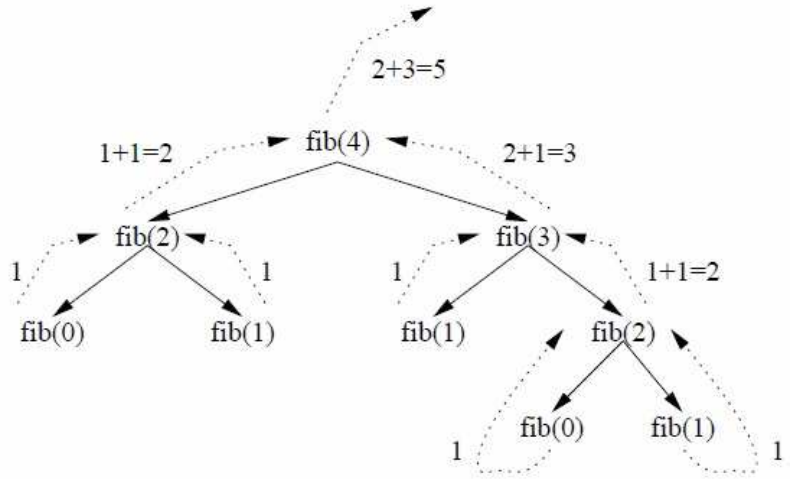

Figura 4: Ejecución de *fib(4)*

⋆ En la recursión múltiple, el número de llamadas puede crecer muy rápidamente, como se puede ver en el cómputo de *fib(4)* que se muestra en la Figura 4. Nótese que algunos valores se computan más de una vez (p.e. *fib(2)* se evalúa 2 veces).

1.4. Resumen de los distintos tipos de recursión

⋆ 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. Diseño de algoritmos recursivos

⋆ Dada la especificación $\{P_0\}A\{Q_0\}$, 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.

⋆ Se propone 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.

Se trata, por tanto, de describir de manera poco formal la estrategia a seguir para resolver el problema planteado.

(R.2) **Análisis de casos.** Se trata de identificar y 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:

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

(R.3) **Caso directo.** Hemos de encontrar la acción A_1 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), es decir, la función sucesor debe proporcionar unos datos que cumplan la precondición de la función recursiva para estar seguros de que se realiza la llamada recursiva con datos válidos.

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

(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

```
{ P_0 ∧ r(x⃗) }
    x⃗' = s(x⃗);
    nombreProc(x⃗', y⃗');
    y⃗= c(x⃗, y⃗')
{ Q_0 }
```

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

```
{ P_0 ∧ r(x⃗) }
    y⃗ = c(x⃗, nombreFunc(s(x⃗)))
{ Q_0 }
```

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.

Veamos un ejemplo. Decimos que un vector de naturales es *pareado* si la diferencia entre el número de pares de su mitad izquierda y su mitad derecha no excede la unidad, y además ambas mitades son a su vez pareadas. Vamos a diseñar una función que nos diga si un vector de naturales es o no pareado.

Especificación

$\{longitud(v) \geq num \wedge \forall k : 0 \leq k < num : v[k] >= 0\}$
`fun pareado (int v[],int num) return bool b`
$\{b = esPareado(v, 0, num)\}$

donde el predicado *esPareado* se define según el enunciado de la siguiente manera para $0 \leq i \leq j \leq num$:

$esPareado(v, i, j) \equiv$
$(j == i) \vee$

$(j > i \wedge |(\#k : i \leq k < (i + j)/2 : v[k] \ mod \ 2 = 0) - (\#l : (i + j)/2 + 1 \leq l < j : v[l] \ mod \ 2 = 0)| \leq 1$

$\wedge esPareado(v, i, (i + j)/2) \wedge esPareado(v, (i + j)/2) + 1, j))$

(R.1) Planteamiento recursivo

(R.2) y análisis de casos. El planteamiento recursivo parece directo, ya que para saber si un vector es pareado hemos de comprobar, entre otras cosas, que sus dos mitades cumplen a su vez la misma propiedad. La cuestión es cómo hacer referencia en la llamada recursiva a una mitad del vector. No basta con escribir $pareado(v, num/2)$, ya que hay dos mitades, la de la izquierda y la de la derecha. Adicionalmente, cada una de ellas tiene a su vez dos mitades que habrá que comprobar que cumplen la propiedad y asi sucesivamente, lo cual quiere decir que necesitamos saber cuando un segmento válido cualquiera del vector es pareado.

Para ello vamos a utilizar una función auxiliar que nos permita hacer el planteamiento recursivo descrito anteriormente:

$\{P_0 \equiv 0 \leq i \leq j \leq longitud(v) \wedge \forall k : 0 \leq i \leq k < j : v[k] >= 0\}$
`fun pareado (int v[],int i,int j) return bool b`
$\{b = esPareado(v, i, j)\}$

Se dice que esta función es una *generalización* de la función a desarrollar porque tiene más parámetros que la función que se desea definir, y además la función original se puede calcular como un caso particular de la auxiliar:

$$pareado(v, num) = pareado(v, 0, num)$$

Para calcular lo que se desea por tanto basta con una llamada a $pareado(v, 0, num)$. Esta llamada recibe el nombre de *llamada inicial*.

La generalización es una técnica muy utilizada en el diseño de algoritmos recursivos que consiste en añadir parámetros adicionales y/o resultados adicionales que ayudan a diseñar la función o a hacerla más eficiente.

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.

Vamos por tanto a diseñar recursivamente la función auxiliar. El problema se resuelve trivialmente cuando el segmento es vacio, es decir, cuando $i = j$, ya que en tal caso el segmento se considera pareado. Por ello distinguimos los casos:

$$d(v, i, j) : i = j$$
$$r(v, i, j) : i < j$$

Claramente, estos dos casos cubren los admitidos por la precondición.

El planteamiento recursivo consiste en comprobar que las dos mitades son pareadas: $pareado(v, i, (i + j)/2)$ y $pareado(v, (i + j)/2 + 1, j)$, y hacer una comprobación adicional que consiste en contar el número de pares de cada una de esas mitades y comprobar que la diferencia en valor absoluto no supera a 1.

(R.3) Solución en el caso directo.

{ $P_0 \wedge i = j$ }
A_1
{ $esPareado(v, i, j)$ }

Claramente, por definición de $esPareado$: $A_1 \equiv b = true$;

(R.4) Descomposición recursiva.

$$s_1(v, i, j) = (v, i, (i + j)/2)$$
$$s_2(v, i, j) = (v, (i + j)/2 + 1, j)$$

(R.5) Función de acotación y terminación. El valor que disminuye hasta llegar al caso base es

$$t(i, j) = j - i$$

Efectivamente, se decrementa en cada una de las dos llamadas recursivas:

$$t(s_1(i, j)) = (i + j)/2 - i$$
$$t(s_2(i, j)) = j - (i + j)/2 - 1$$

ya que si $i < j$ entonces $i \leq (i + j)/2 < j$ y por tanto $(i + j)/2 - i < j - i$ y también $j - (i + j)/2 - 1 < j - i$.

(R.6) Es posible hacer las dos llamadas recursivas, es decir, los argumentos de las llamadas recursivas cumplen la precondición

$$P_0(v,i,j) \wedge r(i,j) \Rightarrow P_0(v,i,(i+j)/2)$$
$$P_0(v,i,j) \wedge r(i,j) \Rightarrow P_0(v,(i+j)/2+1,j)$$

Puesto que el vector no se modifica, la parte que correponde al hecho de que el vector es de naturales se cumple trivialmente, por lo que nos centramos en los indices:

$$0 \leq i \leq j \leq longitud(v) \Rightarrow 0 \leq i \leq (i+j)/2 \leq longitud(v)$$
$$0 \leq i \leq j \leq longitud(v) \Rightarrow 0 \leq (i+j)/2+1 \leq j \leq longitud(v)$$

(R.7) Función de combinación y escritura del caso recursivo. Como hemos mencionado previamente hemos de comprobar que ambas mitades son pareadas, para lo cual haremos dos llamadas recursivas. Llamemos m a la posición central $(i+j)/2$. Además de las llamadas recursivas hemos de contar el número de pares en cada una de las dos mitades, para lo que vamos a utilizar una función iterativa, que diseñaremos después y cuya especificación es:

$\{0 \leq i \leq j < longitud(v) \wedge \forall k : 0 \leq i \leq k < j : v[k] >= 0\}$
`fun contarPares (int v[],int i,int j) return int r`
$\{r = \#l : i <= l < j : v[l] \ \ mod \ \ 2 = 0\}$

Utilizando dicha función, el caso recursivo será:

```
int m = (i+j)/2;
b = (abs(contarPares(v,i,m)-contarPares(v,m+1,j))<=1)
              && pareado(v,i,m) && pareado(v,m+1,j);
```

Finalmente, el algoritmo queda:

```
bool pareado (int v[],int i,int j)
{
// Pre:  0 ≤ i ≤ j < longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] >= 0
bool b;
if (i==j) {b=true;}
else
 {b=(abs(contarPares(v,i,m)-contarPares(v,m+1,j))<=1)
          && pareado(v,i,m) && pareado(v,m+1,j);}
return b;
//Post: b = esPareado(v,i,j)
}
```

siendo la llamada inicial

```
bool pareado (int v[],int num)
 {// Pre: longitud(v) ≥ num ∧ ∀k : 0 ≤ k < num : v[k] >= 0
  return pareado(v,0,num);
  //  Post: b = esPareado(v,0,num)
 }
```

En secciones posteriores de este tema veremos que el coste de este algoritmo está en $O(num \log(num))$ y cómo modificar el diseño de la función para obtener una versión más eficiente con coste en $O(num)$.

2.1. 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 donde 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.

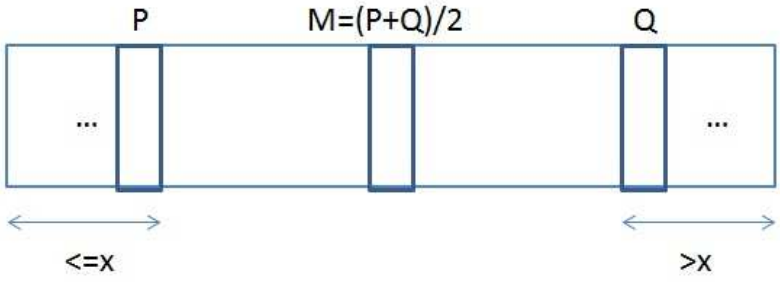

Figura 5: Cálculo del punto medio

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

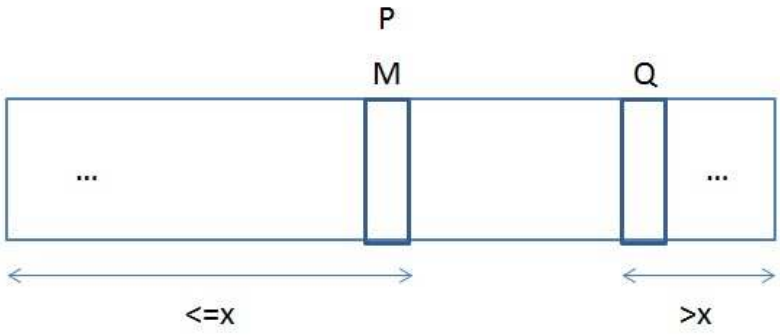

Figura 6: Búsqueda en la mitad derecha

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.

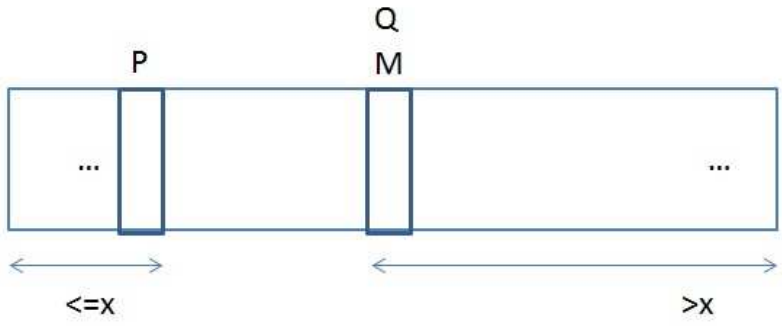

Figura 7: Búsqueda en la mitad izquierda

$\{0 \leq num \leq longitud(v) \wedge ord(v, num)\}$
`fun buscaBin (TElem v[], int num, TElem x) return int pos`
$\{(\exists u : 0 \leq u < num : v[u] \leq x \wedge pos = \text{máx}\, k : (0 \leq k \leq num - 1) \wedge v[k] \leq x : k)$
$\vee(\forall z : 0 \leq z < num : x < v[z] \wedge pos = -1)\}$

```
typedef int TElem;

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

// cuerpo de la función

  return pos;
}
```

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 $\leq$.
- 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.

⋆ De forma similar al ejemplo anterior, para comprobar si un vector es pareado, utilizaremos una función -o un procedimiento- auxiliar que nos permita implementar el planteamiento recursivo.

En el caso de la búsqueda binaria, se trata de 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 (ambos incluidos en este caso).

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

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

$$buscaBin(v, x) = buscaBin(v, x, 0, longitud(v) - 1)$$

⋆ La función recursiva es, por tanto, la función auxiliar:

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

// cuerpo de la función

}
```

y para buscar un elemento en el vector podemos llamar a `buscaBin( v, x, 0, l-1)`, siendo l la longitud del vector v.

⋆ Diseño del algoritmo:

(R.1) Planteamiento recursivo

(R.2) 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  p = a-1;  // x no está en v  y v[a] >  x
```

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

$$x = 8 \quad a = 0 \quad b = 1 \quad v[0] = 10 \quad v[1] = 15$$

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

$$a = 0 \quad b = -1$$

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:

$\{ord(v, longitud(v))$
$\wedge(0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v) - 1) \wedge (a \leq b + 1)$
$\wedge(\forall i : 0 \leq i < a : v[i] \leq x) \wedge (\forall j : b < j < longitud(v) : v[j] > x)\}$

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

$\{(\exists u : 0 \leq u < longitud(v) : v[u] \leq x \wedge pos = \max k : (0 \leq k \leq longitud(v) - 1) \wedge v[k] \leq x : k)$
$\vee(\forall z : 0 \leq z < longitud(v) : x < v[z] \wedge pos = -1)\}$

```
typedef int TElem;

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

// cuerpo de la función

  return p;
}
```

Donde se tiene la siguiente distinción de casos

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

$$r(v, x, a, b) : a \leq b$$

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 $\leq x$,
- 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 $\leq x$ 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 que 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 ordenado entre $0..longitud(v)-1 \Rightarrow v$ ordenado entre $0..longitud(v)-1$
  $0 \leq a \leq longitud(v) \Rightarrow 0 \leq a \leq longitud(v)$
  $-1 \leq b \leq longitud(v)-1 \wedge a \leq m \leq b \wedge 0 \leq a \leq longitud(v) \Rightarrow -1 \leq m-1 \leq longitud(v)-1$
  $a \leq m \Rightarrow a \leq m-1+1$

  todos los elementos a la izquierda de a son $\leq x \Rightarrow$ todos los elementos a la izquierda de a son $\leq x$

  v está ordenado entre $0..longitud(v)-1 \wedge$ todos los elementos a la derecha de b son $> x \wedge m \leq b \wedge x < v[m] \Rightarrow$ todos los elementos a la derecha de $m-1$ son $> x$

- $x \geq v[m]$
  v está ordenado entre $0..longitud(v)-1$
  $(0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v)-1) \wedge (a \leq b+1)$

todos los elementos a la izquierda de a son $<= x$
todos los elementos a la derecha de b son $> x$

$\Rightarrow v$ está ordenado entre $0..longitud(v)-1$
$(0 \leq m+1 \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v)-1) \wedge (m+1 \leq b+1)$

todos los elementos a la izquierda de $m+1$ son $\leq x$
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 $0..longitud(v)-1 \wedge a = 0 \wedge b = longitud(v)-1 \Rightarrow v$ está ordenado entre $0..longitud(v)-1$

$(0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v)-1) \wedge (a \leq b+1)$

todos los elementos a la izquierda de a son $<= x$
todos los elementos a la derecha de b son $> x$

Que es cierto porque $longitud(v) \geq 0$.

(R.7) Función de combinación.

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

$$p = p'$$

(R.8) 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 )
```

$\star$ Con lo que finalmente la función queda:

```
int buscaBin( TElem v[], TElem x, int a, int b) {
// Pre:  v está ordenado entre 0 .. longitud(v)-1
// ( 0 ≤ a ≤ longitud(v) ) && ( -1 ≤ b ≤ longitud(v)-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 {    // 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 i (0 ≤ i ≤ longitud(v) − 1) que cumple v[i] ≤ x
//   si x es menor que todos los elementos de v, devuelve −1
}
```

También nos puede interesar definir una versión de la función en la que se recibe el vector junto con su tamaño n:

```
int buscaBin( TElem v[], int n, TElem x ) {
//0 ≤ n ≤ longitud(v) y todos los elementos a la derecha de n−1 son > x
  return buscaBin( v, x, 0, n-1);
}
```

En este caso debemos llamar a `buscaBin(v,x,l)`, siendo l la longitud del vector v.

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

2.2. Algoritmos avanzados de ordenación

⋆ La ordenación rápida (quicksort) y la ordenación por mezcla (mergesort) son dos algoritmos de ordenación de complejidad cuasilineal, $O(nlogn)$.

⋆ Las idea recursiva es similar en los dos algoritmos: para 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.

2.2.1. Ordenación rápida (*quicksort*)

⋆ Especificación:

$\{0 \leq n \leq longitud(v)\}$
`proc quickSort ( TElem v[], int n)`
$\{ord(v, 0, n-1)\}$

```
void quickSort ( TElem v[], int n) {

  quickSort(v, 0, n-1);

}
```

$\{0 \leq a \leq longitud(v) \wedge -1 \leq b \leq longitud(v) - 1 \wedge a \leq b + 1\}$
`proc quickSort( TElem v[], int a, int b)`
$\{ord(v, a, b)\}$

```
void quickSort( TElem v[], int a, int 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.

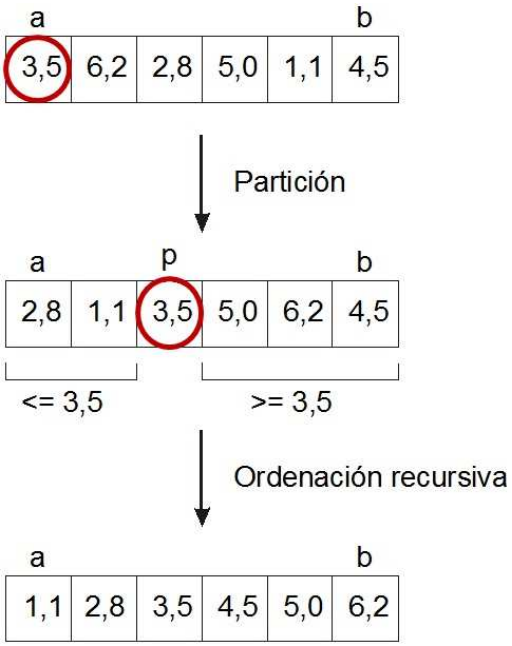

Figura 8: Planteamiento del algoritmo *quicksort*

⋆ El planteamiento recursivo consiste en:

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

- 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 *particion*, el algoritmo nos queda:

```
void quickSort( TElem v[], int a, int b) {
// Pre: 0 ≤ a ≤ longitud(v) && −1 ≤ b ≤ longitud(v)−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 ≤ longitud(v)-1 }
  particion
{ Q: 0 ≤ a ≤ p ≤ b ≤ longitud(v) − 1 ∧ (∀x : a ≤ x ≤ p − 1 : v[x] ≤ v[p])
       ∧(∀x : p + 1 ≤ x ≤ b : v[x] ≥ 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
  $a + 1 \leq i \leq j + 1 \leq b + 1 \wedge$
  todos los elementos desde $a + 1$ hasta $i - 1$ son $\leq v[a] \wedge$
  todos los elementos desde $j + 1$ hasta b son $\geq v[a]$

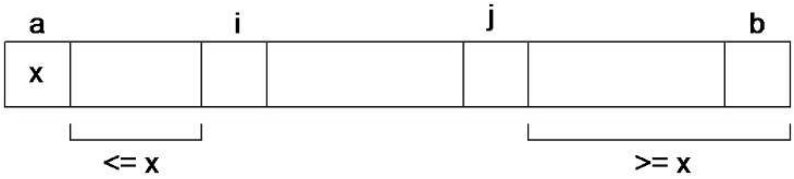

Figura 9: Diseño de *particion*

- 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

  $$i \leq j$$

  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 $v[a]$
  - $v[i] \leq v[a] \rightarrow$ incrementamos i
  - $v[j] \geq v[a] \rightarrow$ decrementamos j
  - $v[i] > v[a] \wedge v[j] < v[a] \rightarrow$ 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 {   //  (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 ≤ longitud(v)−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 ≤ longitud(v)−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 ≤ longitud(v) && −1 ≤ b ≤ longitud(v)−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
}
```

de forma que para ordenar todo el vector la llamada inicial es `quickSort(v,0,l-1)` siendo l la longitud del vector v.

Otra versión que nos puede interesar es:

```
void quickSort ( TElem v[], int n) {
// Pre: 0 ≤ n ≤ longitud(v)

  quickSort(v, 0, n-1);

// Post: se ha ordenado v entre 0 y n−1
}
```

En este caso debemos llamar a `quickSort(v,l)`, siendo l la longitud del vector v.

2.2.2. Ordenación por mezcla (*mergesort*)

⋆ Partimos de una especificación similar a la del *quickSort*.

$\{0 \leq n \leq longitud(v)\}$
`proc mergeSort ( TElem v[], int n)`
$\{ord(v, 0, n-1)\}$

```
void mergeSort ( TElem v[], int n) {
  mergeSort(v, 0, n-1);
}
```

$\{0 \leq a \leq longitud(v) \wedge -1 \leq b \leq longitud(v) - 1 \wedge a \leq b + 1\}$
`proc mergeSort( TElem v[], int a, int b)`
$\{ord(v, a, b)\}$

```
void mergeSort( TElem v[], int a, int b) { }
```

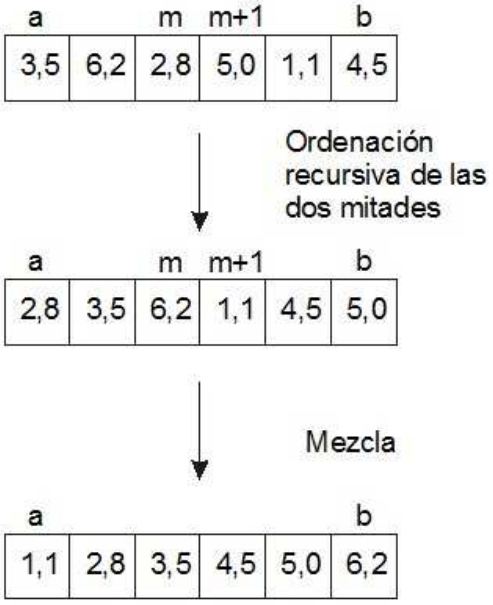

Figura 10: Planteamiento del algoritmo *mergesort*

⋆ 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 $\geq$ 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 ≤ longitud(v) && −1 ≤ b ≤ longitud(v)−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
}
```

de forma que para ordenar todo el vector la llamada inicial es `mergeSort(v,0,l-1)` siendo l la longitud del vector v.

Otra versión que nos puede interesar es:

```
void mergeSort ( TElem v[], int n) {
// Pre: 0 ≤ n ≤ longitud(v)

  mergeSort(v, 0, n-1);

// Post: se ha ordenado v entre 0 y n−1
}
```

En este caso debemos llamar a `mergeSort(v,l)`, siendo l la longitud del vector v.

⋆ 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.

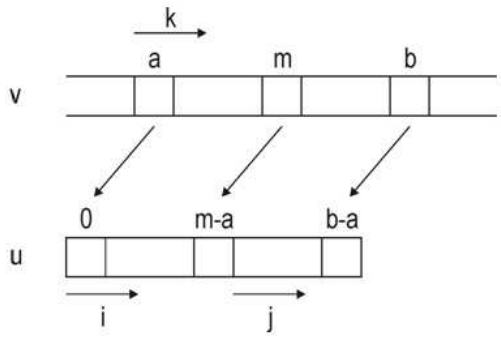

Figura 11: Diseño de *mezcla*

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: 0 ≤ a ≤ m < b ≤ longitud(v)−1 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];
  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
}
```

3. Análisis de la complejidad de algoritmos recursivos

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.*

3.1.1. Ejemplos

⋆ Cálculo del factorial.
Tamaño de los datos: n
Caso directo, n = 0 : T(n) = 2
Caso recursivo:

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

Ecuaciones de recurrencia: $T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n-1) & \text{si } n > 0 \end{cases}$

⋆ 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 $b/2$ +
- 1 de la suma $prod(2 * a, b/2) + a$ en una de las ramas +
- T(n/2) de la llamada recursiva.

Ecuaciones de recurrencia: $T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$

⋆ 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!

3.1.2. Ejemplos

⋆ 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: $c * n$ +
- 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: $2 * T(n/2)$

Ecuaciones de recurrencia en el caso peor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n-1) + c * n + c_0 & \text{si } n \geq 1 \end{cases}$$

Ecuaciones de recurrencia en el caso mejor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

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.

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.

3.2.1. Ejemplos

⋆ Factorial

- Ecuaciones
$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n-1) & \text{si } n > 0 \end{cases}$$
- Despliegue
$$\begin{aligned} T(n) &= 4 + T(n-1) \\ &= 4 + 4 + T(n-2) \\ &= 4 + 4 + 4 + T(n-3) \\ &... \\ &= 4 * k + T(n-k) \end{aligned}$$
- Postulado
El caso directo se tiene para $n = 0$
$$n - k = 0 \Leftrightarrow k = n$$
$$T(n) = 4n + T(n-n) = 4n + T(0) = 4n + 2$$
Por lo tanto $T(n) \in O(n)$

⋆ Multiplicación por el método del campesino egipcio

- Ecuaciones
$$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$$
- Despliegue
$$\begin{aligned} T(n) &= 8 + T(n/2) \\ &= 8 + (8 + T(n/2/2)) \\ &= 8 + 8 + 8 + T(n/2/2/2) \\ &... \\ &= 8 * k + T(n/2^k) \end{aligned}$$

- Postulado
  Las llamadas recursivas terminan cuando se alcanza 1

$$n/2^k = 1 \Leftrightarrow k = \log n$$

$$T(n) = 8\log n + T(n/2^{\log n}) = 8\log n + T(1) = 8\log n + 3$$

  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

- Ecuaciones

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

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

$$T(n) \leq 2 * T(n-1) + c_1 \quad si\ n > 1$$

- Despliegue

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

- Postulado
  Las llamadas recursivas terminan cuando se alcanzan 0 y 1. Como buscamos una cota superior, consideramos 0.
  $n - k = 0 \Leftrightarrow k = n$

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

  donde hemos utilizado la fórmula para la suma de progresiones geométricas:

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

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

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

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.

3.3.1. 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:

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

donde:

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

Se puede demostrar:

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

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 ($c * n^k$) y el coste lineal de la recursión (n).
- En recursión múltiple ($a > 1$), por muy grande que sea b, el coste siempre es exponencial.

3.3.2. 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:

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

donde:

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

Se puede demostrar:

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

Si $a \leq b^k$ la complejidad depende de n^k que es el término que proviene de $c * n^k$ 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 $a < b^k$.

3.3.3. Ejemplos

⋆ 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:

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

⋆ Ordenación por mezcla (mergesort).
Tamaño de los datos: $n = num$
Recurrencias:

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

donde $c * n$ 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:

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

Este es también el coste del algoritmo *pareado* que vimos en la sección anterior, ya que la recurrencia es la misma. En la siguiente sección vamos a ver cómo generalizar aun más esta función para hacerla más eficiente.

4. Técnicas de generalización de algoritmos recursivos

⋆ 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≤ longitud(v) && −1≤b≤longitud(v)−1 && a≤b+1

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

void quickSort ( TElem v[], int n) {
// Pre: 0 ≤ n ≤ longitud(v)

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

  v está ordenado entre a y b $\wedge a = 0 \wedge b = longitud(v) - 1 \Rightarrow$ se ha ordenado 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 $a * n! \wedge a = 1 \Rightarrow$ devuelve $n!$

4.1. 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: cálculo recursivo de una potencia. Tenemos la siguiente función recursiva no final para calcular la potencia natural de un entero a:

```
// Pre: n ≥ 0
int potencia (int a, int n)
{
    int p;

    if (n == 0)
```

```
            p = 1;
        else    // n>0
            p = potencia(a, n-1) * a;
        return p;
    }
    //Post: p = a^n
```

Una forma de obtener una versión recursiva final consiste en añadir un parámetro ac, que lleve calculado parte del producto. Para poder diseñar la función nos hace falta saber qué potencia lleva acumulada, así que introducimos otro parámetro i y aseguramos en la precondición que $ac = a^i$. Por tanto, el caso base, cuando $i == n$, ac ya lleva acumulado lo que queremos y ese es el resultado de la función. En caso contrario, acumulamos un factor a más a ac e incrementamos la i en 1 para que se cumpla la precondición en la llamada recursiva. El algoritmo es el siguiente:

De esa forma

```
int potencia(int a, int n, int ac, int i)
{//Pre : ac = a^i ∧ 0 ≤ i ≤ n
 int p;
 if (i==n)
   {r=ac;}
 else
    {r=potencia(a,n,ac*a,i+1)
  return p;
  //Post: p = a^n
}
```

donde la llamada inicial se hace con $i = 0$, ya que aun no hemos acumulado nada y consecuentemente $ac = 1$:

```
int potencia(int a, int n)
  // Pre: n ≥ 0
{
return potencia(a,n,1,0);
}
 //Post: p = a^n
```

Recuérdese que esto no hace que la función sea asintóticamente más eficiente.

4.2. 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.

4.2.1. 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 y 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 $\vec{b} = e(\vec{x}, \vec{y})$ 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: recordemos el algoritmo *pareado* visto en la Sección 2. Podemos obtener una versión más eficiente del algoritmo añadiendo un resultado adicional que determina el número de pares que hay en el segmento considerado: de esa manera, las propias llamadas recursivas nos devolverán el número de pares de cada mitad y la comprobación adicional que necesitamos se hará en tiempo constante. Como ahora tenemos dos resultados, utilizamos un procedimiento con dos parámetros de salida: el booleano b y el entero p que representa el número de números pares:

$\{0 \leq i \leq j < longitud(v) \wedge \forall k : 0 \leq i \leq k < j : v[k] >= 0\}$
`proc pareado (int v[],int i,int j, out bool b, out int p)`
$\{b = esPareado(v, i, j) \wedge p = \#l : i <= l < j : v[l] \ mod \ 2 = 0\}$

El caso base sigue siendo el mismo, cuando $i == j$ pero en ese caso no solamente tenemos que asignar *true* a b sino que p ha de tomar el valor 0, ya que no hay ningún par en un segmento vacio.

En el caso recursivo, cada una de las dos llamadas, devuelve un booleano ($b1$ y $b2$ respectivamente) y un entero ($p1$ y $p2$ respectivamente):$pareado(v, i, m, b1, p1)$ y $pareado(v, i, m, b2, p2)$. Con lo cual el caso recursivo sería ser de la siguiente forma:

```
int m = (i+j)/2;
bool b1,b2;
int p1,p2;
pareado(v,i,m,b1,p1);
if (b1) //para no hacer llamadas innecesariamente
  {
   pareado(v,i,b2,p2);
   b=b2 && (abs(p1-p2)<=1); //comprobacion de coste constante
  }
else
```

```
  {b=false;}
p=p1+p2;
if (v[m]%2==0) {p++};
```

Puesto que ahora, el algoritmo devuelve también el número de pares, debemos asignarle un valor a p, que no es mas que la suma de los números pares en la mitad izquierda y derecha, mas 1 en el caso de que el punto medio lo sea. El algoritmo en C++ queda por tanto:

```
// Pre: 0 ≤ i ≤ j < longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] >= 0
void pareado (int v[],int i,int j, bool &b, int &p)
{
if (i==j) {b=true; p=0;}
else
 {
   int m = (i+j)/2;
   bool b1,b2;
   int p1,p2;
   pareado(v,i,m,b1,p1);
   if (b1) //para no hacer llamadas innecesariamente
      {
       pareado(v,i,b2,p2);
       b=b2 && (abs(p1-p2)<=1); //comprobacion de coste constante
      }
   else
      {b=false;}
   p=p1+p2;
   if (v[m] %2==0) {p++};
  }
}
//Post: b = esPareado(v,i,j) ∧ p = #l : i <= l < j : v[l] mod 2 = 0}
```

siendo la llamada inicial

```
// Pre: longitud(v) ≥ num ∧ ∀k : 0 ≤ k < num : v[k] >= 0
bool pareado (int v[],int num)
 {
  bool b; int p;
  pareado(v,0,num,b,p);
  return b;
 }
 //   Post: b = esPareado(v,0,num)
```

En este caso tenemos la siguiente recurrencia, siendo n el tamaño del segmento $(j-i)$

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

Estamos en el caso en que $a = 2$, $b = 2$ y $k = 0$, por lo que $T(n) \in O(n^{\log_2 2})$, es decir, $T(n) \in O(n)$.

4.2.2. 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:
  $\{\vec{a} = e(\vec{x}) \wedge r(\vec{x})\}$
  $(\vec{a}',\vec{x}') = s'(\vec{a},\vec{x})$
  $\{\vec{x}' = s(\vec{x}) \wedge \vec{a}' = e(\vec{x}')\}$

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: supongamos que deseamos rellenar un vector de tamaño $2^i - 1$ para un cierto i con naturales de la siguiente manera:

- El elemento del centro es 1.

			1			

- El punto medio de la mitad izquierda se obtiene multiplicando por 2 el elemento del centro y el de la mitad derecha multiplicando por 3 el elemento del centro.

	2		1		3	

- Para rellenar el resto de posiciones se procede de la misma manera: multiplicando por 2 cuando vamos a la izquierda y por 3 si vamos a la derecha.

4	2	6	1	6	3	9

Cada posición del vector contendrá por tanto $2^i * 3^j$ para ciertos valores de i y j, dependiendo de la posición. Para no tener que calcular dichas potencias en el punto de asignación al vector, llevaremos un parámetro adicional que acumula el valor de dichas potencias:

```
// Pre: 0<=i<=j+1<=n
 void rellenar(int v[],int i,int j, int ac)
 { if (i<=j)
    { int m= (i+j)/2; v[m]=ac;
      rellenar(v,i,m-1,ac*2);
      rellenar(v,m+1,j,ac*3);
    }
 }
// Post: rellenado(v,i,j,ac)
```

siendo la propiedad *rellenado* fácilmente definible de manera recursiva:

$$\begin{aligned} rellenado(v,i,j,ac) \equiv (i \leq j) \Rightarrow \quad & v[(i+j)/2] == ac \\ & \wedge rellenado(v,i,(i+j)/2-1,ac*2) \\ & \wedge rellenado(v,(i+j)/2+1,j,ac*3) \end{aligned}$$

La llamada inicial sería $rellenar(v,0,n-1,1)$ siendo n la longitud del vector; y por tanto se cumplirá $rellenado(v,0,n-1,1)$.

5. Verificación de algoritmos recursivos

⋆ Supondremos que la llamada recursiva devuelve los valores deseados para demostrar que la etapa de combinación cumple con la especificación.

⋆ Debemos probar que:

1. Se cubren todos los casos:

$$P(\vec{x}) \Rightarrow d(\vec{x}) \vee r(\vec{x})$$

2. El caso base es correcto:

$$P(\vec{x}) \wedge d(\vec{x}) \Rightarrow Q(\vec{x}, \vec{y})$$

3. Los argumentos de la llamada recursiva deben satisfacer su precondición:

$$P(\vec{x}) \wedge r(\vec{x}) \Rightarrow P(s(\vec{x}))$$

4. El paso de inducción es correcto ($\Leftrightarrow$ la etapa de combinación es correcta):

$$P(\vec{x}) \wedge r(\vec{x}) \wedge Q(s(\vec{x}), \vec{y}') \Rightarrow Q(\vec{x}, c(\vec{x}, \vec{y}'))$$

Para garantizar que termine la secuencia de llamadas a la función debemos exigir además:

1. Existe una función de cota $t(\vec{x})$ tal que:

$$P(\vec{x}) \Rightarrow t(\vec{x}) \geq 0$$

2. La función de cota debe decrecer en cada iteración:

$$P(\vec{x}) \wedge r(\vec{x}) \Rightarrow t(s(\vec{x})) < t(\vec{x})$$

⋆ **Ejemplo**. Verifica el siguiente algoritmo:

```
// fun potencia (int a, int n): return (int p)
// P: n ≥ 0
int potencia (int a, int n)
{
    int p;

    if (n == 0)
        p = 1;
    else    // n>0
        p = potencia(a, n-1) * a;
    return p;
}
// Q: p = a^n
```

Solución:

- Como $n \geq 0$: $n = 0 \vee n > 0$ es cierto.

- El caso base verifica la postcondición:

$$\{n = 0\}p = 1\{p = a^n\}$$

  ya que $1 = a^n \Leftarrow n = 0$.
- En cada iteración la llamada recursiva verifica la precondición. El único problema consistiría en que n dejase de ser ≥ 0. Como en el caso recursivo $n > 0$, $n - 1$ sigue siendo ≥ 0.
- El paso de inducción es correcto:

$$n \geq 0 \wedge n > 0 \wedge p = a^{n-1} \Rightarrow (p = a^n)_p^{p*a}$$

- Consideremos la siguiente cota:

$$t = n(\geq 0)$$

- Decrece:

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

Notas bibliográficas

El contenido de este capítulo se basa en su mayor parte en el capítulo correspondiente de (**?**). El apartado final de verificación de algoritmos recursivos se puede encontrar en (**?**). Finalmente, pueden encontrarse ejercicios resueltos relacionados con este tema en (**?**).

Ejercicios

Diseño de algoritmos recursivos

1. Dado un vector de enteros de longitud n, diseñar un algoritmo de complejidad $O(n \log n)$ que encuentre el par de enteros más cercanos (en valor). ¿Se puede hacer con complejidad $O(n)$?

2. Dado un vector de enteros de longitud n, diseñar un algoritmo de complejidad $O(n \log n)$ que encuentre el par de enteros más lejanos (en valor). ¿Se puede hacer con complejidad $O(n)$?

3. Diseñar un algoritmo recursivo que realice el cambio de base de un número binario dado en su correspondiente decimal.

4. Implementa una función recursiva simple *cuadrado* que calcule el cuadrado de un número natural n, basándote en el siguiente análisis de casos:

   - Caso directo: Si $n = 0$, entonces $n^2 = 0$
   - Caso recursivo: Si $n > 0$, entonces $n^2 = (n-1)^2 + 2 * (n-1) + 1$

5. Implementa una función recursiva *log* que calcule la parte entera de $log_b n$, siendo los datos b y n enteros tales que $b \geq 2 \wedge n \geq 1$. El algoritmo obtenido deberá usar solamente las operaciones de suma y división entera.

6. Implementa una función recursiva *bin* tal que, dado un número natural n, *bin(n)* sea otro número natural cuya representación decimal tenga los mismos dígitos que la representación binaria de n. Es decir, debe tenerse: $bin(0) = 0; bin(1) = 1; bin(2) = 10; bin(3) = 11; bin(4) = 100$; etc.

7. Implementa un procedimiento recursivo simple *dosFib* que satisfaga la siguiente especificación:

```
void dosFib( int n, int& r, int& s ) {
// Pre:  n ≥ 0
// Post: r = fib(n) ∧ s = fib(n + 1)
}
```

   En la postcondición, *fib(n)* y *fib(n+1)* representan los números que ocupan los lugares n y *n+1* en la sucesión de Fibonacci, para la cual suponemos la definición recursiva habitual.

8. Implementa una función recursiva que calcule el número combinatorio $\binom{n}{m}$ a partir de los datos m, n enteros tales que $n \geq 0 \wedge m \geq 0$. Usa la recurrencia siguiente:

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$$

   siendo $0 < m < n$

9. El problema de las torres de Hanoi consiste en trasladar una torre de n discos de diferentes tamaños desde la varilla *ini* a la varilla *fin*, con ayuda de la varilla *aux*. Inicialmente los n discos están apilados de mayor a menor, con el más grande en la base. En ningún momento se permite que un disco repose sobre otro menor que él. Los movimientos permitidos consisten en desplazar el disco situado en la cima de una de las varillas a la cima de otra, respetando la condición anterior. Construye un procedimiento recursivo *hanoi* tal que la llamada *hanoi(n, ini, fin, aux)* produzca el efecto de escribir una serie de movimientos que represente una solución del problema de Hanoi. Supón disponible un procedimiento *movimiento(i, j)*, cuyo efecto es escribir "Movimiento de la varilla i a la varilla j".

Análisis de algoritmos recursivos

10. En cada uno de los casos que siguen, plantea una ley de recurrencia para la función T(n) que mide el tiempo de ejecución del algoritmo en el caso peor, y usa el método de desplegado para resolver la recurrencia.

    a) La función *cuadrado* (ejercicio 6).

    b) Función *log* (ejercicio 7).

    c) Función *bin* (ejercicio 8).

    d) Procedimiento *dosFib* (ejercicio 10).

    e) Función del ejercicio 11

    f) Procedimiento *hanoi* (ejercicio 13).

11. Aplica las reglas de análisis para dos tipos comunes de recurrencia a los algoritmos recursivos del ejercicio anterior. En cada caso, deberás determinar si el tamaño de los datos del problema decrece por sustracción o por división, así como los parámetros relevantes para el análisis.

12. En cada caso, calcula a partir de las recurrencias el orden de magnitud de T(n). Hazlo aplicando las reglas de análisis para dos tipos comunes de recurrencia.

    a) $T(1) = c_1; T(n) = 4 * T(n/2) + n,$ si $n > 1$

    b) $T(1) = c_1; T(n) = 4 * T(n/2) + n^2,$ si $n > 1$

    c) $T(1) = c_1; T(n) = 4 * T(n/2) + n^3,$ si $n > 1$

13. Usa el método de desplegado para estimar el orden de magnitud de $T(n)$, suponiendo que T obedezca la siguiente recurrencia:

$$T(1) = 1; T(n) = 2 * T(n/2) + n \log n, sin > 1$$

    ¿Pueden aplicarse en este caso las reglas de análisis para dos tipos comunes de recurrencia? ¿Por qué?

Eliminación de la recursión final

14. A continuación se presentan dos implementaciones iterativas del algoritmo de la búsqueda binaria. La primera versión del algoritmo es la misma que aparecía en el tema anterior, mientras que la segunda versión es el resultado de transformar a iterativo la versión recursiva de este algoritmo que se ha visto en este tema.
    ¿En qué se diferencian estos dos algoritmos iterativos?
    Escribe un algoritmo recursivo final con la misma idea de la primera versión iterativa.

```
int buscaBin( TElem v[], int num, TElem x )
{
  int izq, der, centro;

  izq = -1;
  der = num;
  while ( der != izq+1 ) {
    centro = (izq+der) / 2;
    if ( v[centro] <= x )
      izq = centro;
    else
      der = centro;
  }
  return izq;
}
```

```
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;
}
```

Técnicas de generalización

15. Comprueba que el procedimiento *combiGen* especificado como sigue es una generalización de la función del ejercicio 11, y que *combiGen* admite un algoritmo recursivo simple, más eficiente que la recursión doble anterior, implementándolo.

```
void combiGen ( int a, int b, int v[]) {
// Pre:  0 <= b <= a && b < longitud(v)
// Post: para cada i entre 0 y b se tiene v[i] = (a i) }
```

16. Especifica e implementa un algoritmo recursivo que dado un vector v de enteros, que viene dado en orden estrictamente creciente, determine si el vector contiene alguna posición i que cumpla $v[i] = i$. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.

17. (**?**) Un hostal ofrece alojamiento a turistas todos los días comprendidos en un intervalo $[0..N)$ cuya longitud $N \geq 0$ se sabe que es par. El precio de una estancia diaria varía cada día, siendo:

    - 1 euro para los días 0 y $N-1$ (comienzo y fin de temporada),
    - 2 euros para los días 1 y $N-2$,
    - 2^2 euros para los días 2 y $N-3$,

    y así sucesivamente; es decir, el precio se va multiplicando por dos a medida que nos acercamos al centro del intervalo (temporada alta). Especificar e implementar una función recursiva que calcula los ingresos obtenidos por el hostal durante una temporada a partir de un vector que almacena en cada posición k el número de huéspedes del día k.

18. (**?**) Dos cifras decimales (comprendidas entre 0 y 9) son *pareja* si suman 9. Dado un número natural n, llamaremos *complementario* de n al número obtenido a partir de la representación decimal de n, cambiando cada cifra por su pareja. Por ejemplo, el complementario de 146720 es 853279. Diseñar un algoritmo que, dado un número natural n, calcule su complementario.

Verificación de algoritmos recursivos

19. Verifica los algoritmos recursivos de los ejercicios anteriores

20. Verifica el siguiente algoritmo:

$\{n \geq 0\}$

```
int potenciaLog(int a, int n)
{
    int p;
    if (n == 0)
        p = 1;
    else
    {
        p = potenciaLog(a,n/2);
        p = p * p;
        if (n%2 == 1)
            p = p*a
    }
    return p;
}
```

$\{p = a^n\}$

21. Verifica el siguiente algoritmo:

$\{n \geq 0\}$

```
int productoEscalar(int v[], int w[], int n)
{
    int r;

    if (n == 0)
        r = 0;
    else
        r = v[n-1] * w[n-1] + productoEscalar(v,w,n-1);
    return r;
}
```

$\{r = (\sum i : 0 \leq i < n : v[i] * w[i])\}$

22. Verifica el siguiente algoritmo:

$\{n \geq 0 \wedge a \geq 0 \wedge a^2 \leq n\}$

```
int raiz (int n, int a)
{
    int r;

    if ((a+1)*(a+1) > n)
        r = a;
    else
        r = raiz (n, a+1);
    return r;
}
```

$\{r^2 \leq n < (r+1)^2\}$