Apuntes de Análisis y Diseño de Algoritmos

Conrado Martínez

LSI-UPC

Septiembre 2006

El análisis de algoritmos tiene como objetivo establecer propiedades sobre la eficiencia permitiendo la comparación entre soluciones alternativas y predecir los recursos que usará un algoritmo o ED.

Consideremos el siguiente algoritmo para determinar el mínimo de un vector A de n elementos:

```
i = 0; min = 0;
while (i < n) {
  if (A[i] < A[min])
    min = i;
  ++i;
}
// n = 0 y min = 0 ó
// n > 0 y A[min] = mín{A[i] : 0 ≤ i < n}
```

El número de operaciones elementales (comparaciones, asignaciones, sumas, etc.) que realiza es básicamente proporcional al número n de elementos del vector A y el tiempo de ejecución será, consecuentemente, de la forma $a \cdot n + b$.

- La "forma" del coste $a \cdot n + b$ es independiente del hardware sobre el que se implementa, el sistema operativo, el lenguaje de programación, etc.; sólo las constantes a y b dependen de estos factores
- Para analizar algoritmos sólo contaremos el número de operaciones elementales, asumiendo que cada una de ellas consume una unidad de tiempo
- En general, la eficiencia de un algoritmo dependerá de cada entrada concreta, no sólo del tamaño de la entrada

Dado un algoritmo A cuyo conjunto de entradas es $\mathcal{A}$ su eficiencia o coste (en tiempo, en espacio, en número de operaciones de E/S, etc.) es una función T de $\mathcal{A}$ en $\mathbb{N}$ (o $\mathbb{Q}$ ó $\mathbb{R}$, según el caso):

$$T : \mathcal{A} \to \mathbb{N}$$
$$\alpha \mapsto T(\alpha)$$

Caracterizar la función T puede ser muy complicado y además proporciona información inmanejable, difícilmente utilizable en la práctica.

Sea $\mathcal{A}_n$ el conjunto de entradas de tamaño n y $T_n : \mathcal{A}_n \to \mathbb{N}$ la función T restringida a $\mathcal{A}_n$.

- Coste en caso mejor:

$$T_{\text{mejor}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- Coste en caso peor:

$$T_{\text{peor}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- Coste promedio:

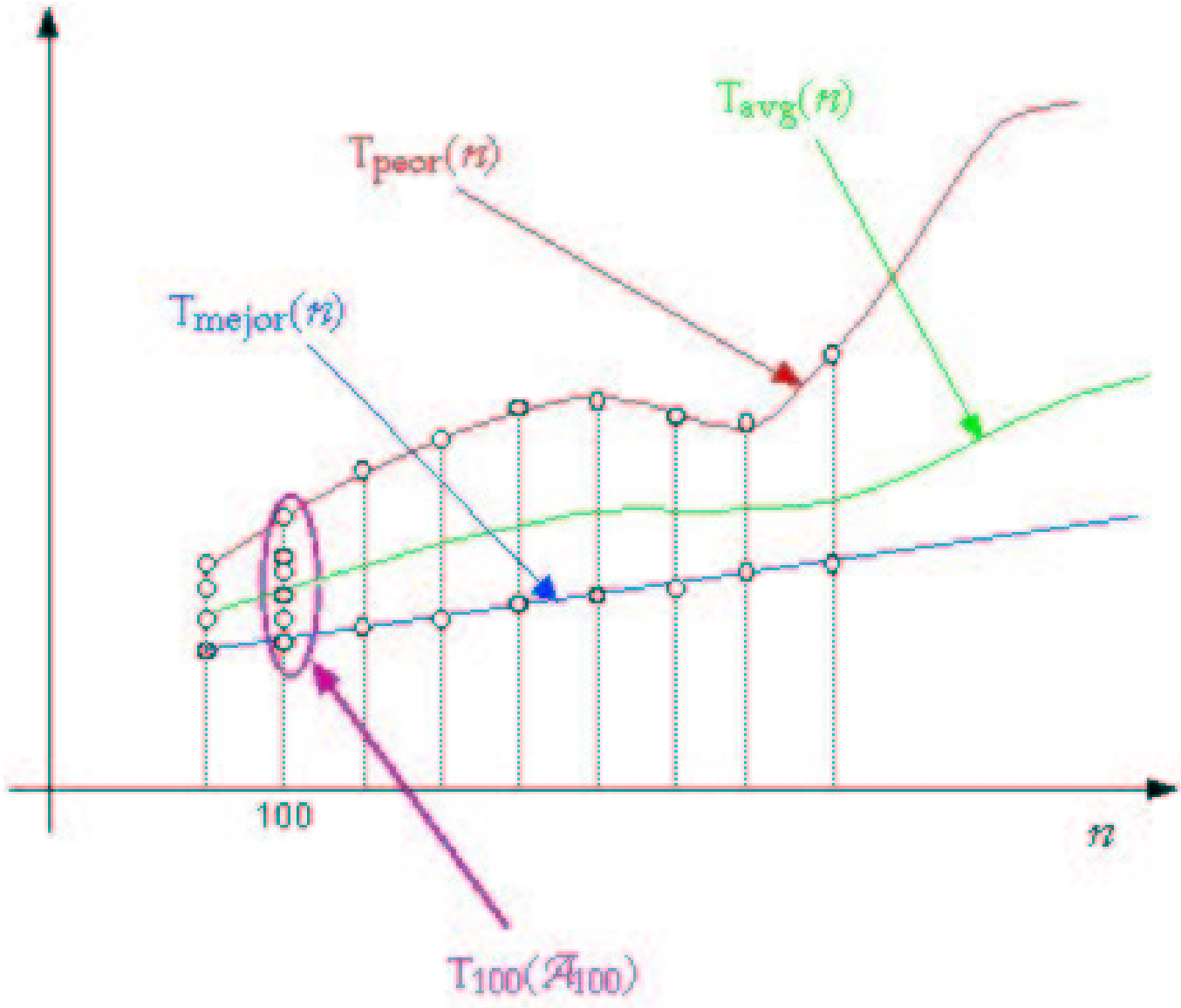
$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha)\, T_n(\alpha) \\ &= \sum_{k \geq 0} k \, \Pr(T_n = k). \end{aligned}$$

1. Para todo $n \geq 0$ y para cualquier $\alpha \in \mathcal{A}_n$

$$T_{\text{mejor}}(n) \leq T_n(\alpha) \leq T_{\text{peor}}(n).$$

2. Para todo $n \geq 0$

$$T_{\text{mejor}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{peor}}(n).$$


$T_{peor}(n)$
$T_{avg}(n)$
$T_{mejor}(n)$
100
n
$T_{100}(A_{100})$

Por lo general, estudiaremos el coste en caso peor de los algoritmos por dos razones:

- proporciona garantías sobre la eficiencia del algoritmo ya que el coste nunca excederá el coste en caso peor
- es más fácil de calcular que el coste promedio

Una característica esencial del coste (en caso peor, en caso mejor, promedio) es su tasa de crecimiento u orden de magnitud.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6{,}87 \cdot 10^{10}$
6	64	384	4096	262144	$4{,}72 \cdot 10^{21}$
			$\ldots$		
ℓ	N	L	C	Q	E
$\ell + 1$	$2N$	$2(L + N)$	$4C$	$8Q$	E^2

Definición

Dada una función $f : \mathbb{R}^+ \to \mathbb{R}^+$ la clase $\mathcal{O}(f)$ (O-grande de f) es

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \, \exists c \, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}.$$

Aunque $\mathcal{O}(f)$ es un conjunto de funciones por tradición se escribe $g = \mathcal{O}(f)$ en vez de $g \in \mathcal{O}(f)$. Sin embargo, $\mathcal{O}(f) = g$ no tiene sentido. Algunas propiedades básicas de la notación $\mathcal{O}$:

1. Si $\lim_{n\to\infty} g(n)/f(n) < +\infty$ entonces $g = \mathcal{O}(f)$.
2. Es reflexiva: para toda función $f : \mathbb{R}^+ \to \mathbb{R}^+$, $f = \mathcal{O}(f)$.
3. Es transitiva: si $f = \mathcal{O}(g)$ y $g = \mathcal{O}(h)$ entonces $f = \mathcal{O}(h)$.
4. Para toda constante $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$.

Además de la notación O-grande se utilizan las notaciones Ω (omega) y Θ (zita). La primera define un conjunto de funciones acotada inferiormente por una dada:

$$\Omega(f) = \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \, \exists c > 0 \, \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}.$$

La notación Ω es reflexiva y transitiva; si $\lim_{n\to\infty} g(n)/f(n) > 0$ entonces $g = \Omega(f)$. Por otra parte, si $f = \mathcal{O}(g)$ entonces $g = \Omega(f)$ y viceversa.
Se dice que $\mathcal{O}(f)$ es la clase de las funciones que crecen no más rápido que f. Análogamente, $\Omega(f)$ es la clase de las funciones que crecen no más despacio que f.
Finalmente,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

es la clase de la funciones con la misma tasa de crecimiento que f.

La notación Θ es reflexiva y transitiva, como las otras. Es además simétrica: $f = \Theta(g)$ si y sólo si $g = \Theta(f)$. Si $\lim_{n\to\infty} g(n)/f(n) = c$ donde $0 < c < \infty$ entonces $g = \Theta(f)$. Otras propiedades adicionales de las notaciones asintóticas son (las inclusiones son estrictas):

1. Para cualesquiera constantes $\alpha < \beta$, si f es una función creciente entonces $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$.
2. Para cualesquiera constantes a y $b > 0$, si f es creciente, $\mathcal{O}((\log f)^a) \subset \mathcal{O}(f^b)$.
3. Para cualquier constante $c > 0$, si f es creciente, $\mathcal{O}(f) \subset \mathcal{O}(c^f)$.

Los operadores convencionales (sumas, restas, divisiones, etc.) sobre clases de funciones definidas mediante una notación asintótica se extienden de la siguiente manera:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

donde A y B son conjuntos de funciones. Expresiones de la forma $f \otimes A$ donde f es una función se entenderá como $\{f\} \otimes A$.
Este convenio nos permite escribir de manera cómoda expresiones como $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, ó $\Theta(1) + \mathcal{O}(1/n)$.

Regla de las sumas:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Regla de los productos:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Reglas similares se cumplen para las notaciones $\mathcal{O}$ y Ω.

Las dos últimas reglas facilitan el análisis del coste en caso peor de algoritmos iterativos.

1. El coste de una operación elemental es $\Theta(1)$.
2. Si el coste de un fragmento S_1 es f y el de S_2 es g entonces el coste de $S_1; S_2$ es $f + g$.
3. Si el coste de S_1 es f, el de S_2 es g y el coste de evaluar B es h entonces el coste en caso peor de

```
if (B) {
    S1
} else {
    S2
}
```

es $\text{máx}\{f + h, g + h\}$.

4. Si el coste de S durante la i-ésima iteración es f_i, el coste de evaluar B es h_i y el número de iteraciones es g entonces el coste T de

```
while (B) {
    S
}
```

es

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

Si $f = \max\{f_i + h_i\}$ entonces $T = \mathcal{O}(f \cdot g)$.

Análisis de algoritmos recursivos

El coste (en caso peor, medio, ...) de un algoritmo recursivo $T(n)$ satisface, dada la naturaleza del algoritmo, una ecuación recurrente: esto es, $T(n)$ dependerá del valor de T para tamaños menores. Frecuentemente, la recurrencia adopta una de las dos siguientes formas:

$$T(n) = a \cdot T(n-c) + g(n),$$
$$T(n) = a \cdot T(n/b) + g(n).$$

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $c \geq 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.

Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1. \end{cases}$$

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $b > 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.
Sea $\alpha = \log_b a$. Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k. \end{cases}$$

Demostración.

Supondremos que $n = n_0 \cdot b^r$. Aplicando la recurrencia repetidamente,

$$\begin{aligned}
T(n) &= g(n) + a \cdot T(n/b) \\
&= g(n) + a \cdot f(n/b) + a^2 \cdot T(n/b^2) \\
&= \cdots \\
&= g(n) + a \cdot g(n/b) + \cdots + a^{r-1} \cdot g(n/b^{r-1}) \\
&\quad + a^r \cdot T(n/b^r) \\
&= \sum_{0 \le j < r} a^j \cdot g(n/b^j) + a^r \cdot T(n_0) \\
&= \Theta\left(\sum_{0 \le j < r} a^j \left(\frac{n}{b^j} \right)^k \right) + a^r \cdot f(n_0) \\
&= \Theta\left(n^k \cdot \sum_{0 \le j < r} \left(\frac{a}{b^k} \right)^j \right) + a^r \cdot f(n_0).
\end{aligned}$$

Puesto que $f(n_0)$ es una constante tenemos que el segundo término es $\Theta(n^\alpha)$ ya que

$$\begin{aligned} a^r &= a^{\log_b(n/n_0)} = a^{\log_b n} \cdot a^{-\log_b n_0} \\ &= \Theta(b^{\log_b a \cdot \log_b n}) \\ &= \Theta(n^{\log_b a}) = \Theta(n^\alpha). \end{aligned}$$

Ahora tenemos tres casos diferentes a considerar: según que a/b^k sea menor, igual o mayor que 1. Alternativamente, ya que $\alpha = \log_b a$, según que α sea mayor, igual o menor que k. Si $k > \alpha$ (equivalentemente, $a/b^k < 1$) entonces la suma que aparece en el primer término es una serie geométrica acotada por una constante, es decir, el primer término es $\Theta(n^k)$. Como asumimos que $k > \alpha$, es el primer término el que domina y $T(n) = \Theta(n^k) = \Theta(g(n))$.

Si $k = \alpha$, tendremos que $a/b^k = 1$ y la suma vale k. Ya que $k = \log_b(n/n_0) = \Theta(\log n)$, concluímos que $T(n) = \Theta(n^k \cdot \log n) = \Theta(g(n) \cdot \log n)$. Finalmente, si $k < \alpha$ entonces $a/b^k > 1$ y el valor de la suma es

$$\sum_{0 \le j < r} \left(\frac{a}{b^k}\right)^j = \frac{(a/b^k)^r - 1}{a/b^k - 1} = \Theta\left(\left(\frac{a}{b^k}\right)^r\right)$$

$$= \Theta(\frac{n^\alpha}{n^k}) = \Theta(n^{\alpha-k}).$$

Por tanto, es el segundo término el que en este caso domina y $T(n) = \Theta(n^\alpha)$.

Introducción

El principio básico de divide y vencerás (en inglés, divide and conquer; en catalán, dividir per conquerir) es muy simple:

1. Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.

Introducción

El principio básico de divide y vencerás (en inglés, divide and conquer; en catalán, dividir per conquerir) es muy simple:

1. Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.
2. En caso contrario, se divide o fragmenta el ejemplar dado en subejemplares $x_1, \ldots, x_k$ y se resuelve, independiente y recursivamente, el problema para cada uno de ellos.

Introducción

El principio básico de divide y vencerás (en inglés, divide and conquer; en catalán, dividir per conquerir) es muy simple:

1. Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.
2. En caso contrario, se divide o fragmenta el ejemplar dado en subejemplares $x_1, \ldots, x_k$ y se resuelve, independiente y recursivamente, el problema para cada uno de ellos.
3. Las soluciones obtenidas $y_1, \ldots, y_k$ se combinan para obtener la solución al ejemplar original.

```
function divide_y_venceras(x : t_dato) return t_resultado
    var x_1, x_2, ..., x_k : t_dato
        y_1, y_2, ..., y_k : t_resultado
        ...
    end
    if x es simple then
      return solucion_directa(x)
    else
      ⟨x_1, x_2, ..., x_k⟩ := divide(x)
      for i := 1 to k do
          y_i := divide_y_venceras(x_i)
      end
      return combina(y_1, y_2, ..., y_k)
    fi
end
```

Si $k = 1$, se habla del esquema de reducción.

El esquema de divide y vencerás se caracteriza adicionalmente por las siguientes propiedades:

- No se resuelve más de una vez un mismo subproblema

El esquema de divide y vencerás se caracteriza adicionalmente por las siguientes propiedades:

- No se resuelve más de una vez un mismo subproblema
- El tamaño de los subejemplares es, en promedio, una fracción del tamaño del ejemplar original, es decir, si x es de tamaño n, el tamaño esperado de un subejemplar cualquiera x_i es n/c_i donde $c_i > 1$. Con frecuencia, esta condición se cumplirá siempre, no sólo en promedio.

El análisis de un algoritmo divide y vencerás requiere, al igual que ocurre con otros algoritmos recursivos, la resolución de recurrencias. La recurrencia típica es

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{si } n > n_0, \\ b(n) & \text{si } n \leq n_0. \end{cases}$$

- n_0 = tamaño máximo de las instancias con solución directa
- n/b = tamaño de los subejemplares resueltos recursivamente
- a = número de llamadas recursivas
- $f(n)$ = coste de dividir y combinar

Búsqueda dicotómica

Problema: Dado un vector $A[0..n-1]$ de n elementos, en orden creciente, y un elemento x, determinar el valor i, $-1 \leq i < n$, tal que $A[i] \leq x < A[i+1]$ (convendremos que $A[-1] = -\infty$ y $A[n] = +\infty$).

Si el vector A fuese vacío la respuesta es sencilla, pues x no está en el vector. Pero para poder obtener una solución recursiva efectiva, debemos realizar la siguiente generalización: dado un segmento o subvector $A[\ell+1..u-1]$, $-1 \leq \ell \leq u \leq n$, ordenado crecientemente, y tal que $A[\ell] \leq x < A[u]$, determinar el valor i, $\ell \leq i \leq u-1$, tal que $A[i] \leq x < A[i+1]$. Este es un ejemplo clásico de inmersión de parámetros.

Si $\ell + 1 = u$, el segmento en cuestión no contiene elementos y $i = u - 1 = \ell$.

```
int bsearch(const vector<T>& A, T x,
            int ℓ, int u) {
    int m = (ℓ + u) / 2;
    if (ℓ == u - 1)
        return ℓ;
    if (x < A[m])
       return bsearch(A, x, ℓ, m);
    else
       return bsearch(A, x, m, u);
}
```

La llamada inicial es $\texttt{bsearch}(A, x, -1, A.\texttt{size}())$, donde $A.\texttt{size}()$ nos da n, el número de elementos del vector A. Prescindiendo de la pequeña diferencia entre el tamaño real de los subejemplares y el valor $n/2$, el coste de la Búsqueda Binaria o dicotómica viene descrito por la recurrencia

$$B(n) = \Theta(1) + B(n/2), \quad n > 0,$$

y $B(0) = b_0$. Aplicamos el teorema I con $\alpha = \log_2 1 = 0$ y por lo tanto $B(n) = \Theta(\log n)$.

Algoritmo de Karatsuba

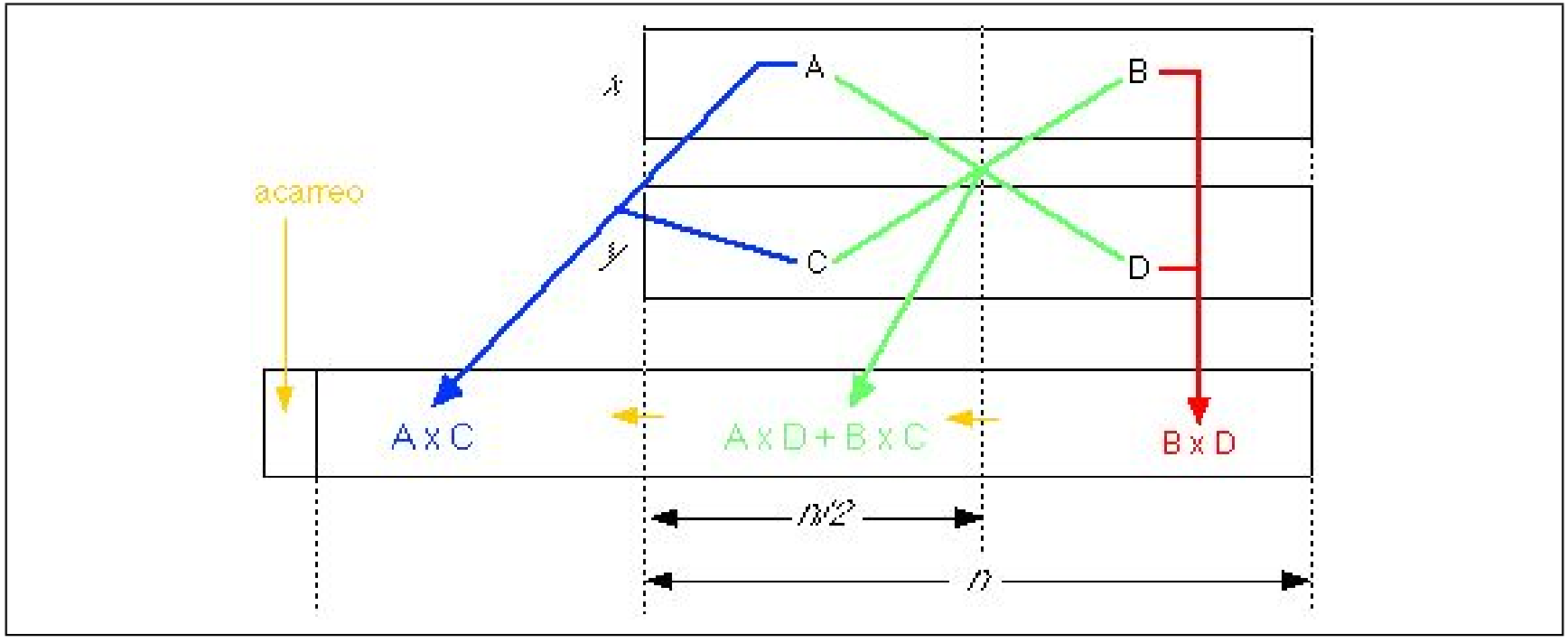
El algoritmo tradicional de multiplicación tiene coste $\Theta(n^2)$ para multiplicar dos enteros de n bits, ya la suma de dos enteros de $\Theta(n)$ bits tiene coste $\Theta(n)$.

```
231 x 659 = 9 x 231 + 5 x 231 x 10
          + 6 x 231 x 100
          = 2079 + 11550 + 138600
          = 152229
```

El mismo coste tiene el algoritmo de multiplicación a la rusa.

```
z = 0;                              //  x = X ∧ y = Y
while (x != 0) {    // Inv:         z + x · y = X · Y
     if (x % 2 == 0) {
          x /= 2; y *= 2;
     } else {
          --x; z += y;
     }
}
```

Supongamos que x e y son dos números positivos de $n = 2^k$ bits (si n no es una potencia de 2 ó x e y no tienen ambos la misma longitud, podemos añadir 0's por la izquierda para que así sea).
Una idea que no funciona:



$$x \cdot y = (A \cdot 2^{n/2} + B) \cdot (C \cdot 2^{n/2} + D)$$
$$= A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D.$$

Efectuada la descomposición $x = \langle A, B \rangle$ y $y = \langle C, D \rangle$, se calculan 4 productos ($A \cdot B$, $A \cdot D$, $B \cdot C$ y $B \cdot D$) y se combinan las soluciones mediante sumas y desplazamientos (shifts). El coste de estas operaciones es $\Theta(n)$. Por lo tanto, el coste de la multiplicación mediante este algoritmo divide y vencerás viene dado por la recurrencia

$$M(n) = \Theta(n) + 4 \cdot M(n/2), \quad n > 1$$

cuya solución es $M(n) = \Theta(n^2)$. En este caso $k = 1$, $a = 4$ y $b = 2$, y $k = 1 < \alpha = \log_2 4 = 2$.

El algoritmo de Karatsuba (1962) realiza la multiplicación dividiendo los dos números como antes pero se realizan las siguientes 3 multiplicaciones (recursivamente)

$$U = A \cdot C$$
$$V = B \cdot D$$
$$W = (A + B) \cdot (C + D)$$

Y el resultado se calcula a partir de las tres soluciones obtenidas

$$x \cdot y = U \cdot 2^n + (W - (U + V)) \cdot 2^{n/2} + V.$$

El algoritmo requiere realizar 6 adiciones (una de ellas es de hecho una resta) frente a las 3 adiciones que se empleaban en el algoritmo anterior. El coste mejora pues el coste de las funciones de división y de combinación del algoritmo de Karatsuba sigue siendo lineal y disminuye el número de llamadas recursivas.

$$M(n) = \Theta(n) + 3 \cdot M(n/2)$$

$$M(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1{,}5849625\ldots})$$

La constante oculta en la notación asintótica es grande y la ventaja de este algoritmo frente a los algoritmos básicos no se manifiesta hasta que n es relativamente grande (del orden de 200 a 250 bits por multiplicando).

```
void mult(const vector<bool>& x, const vector<bool>& y,
                  const vector<bool>& z,
                  int ix, int jx, int iy, int jy) {
// calcula el producto de x[ix..jx] por y[iy..jy]
// con jy - iy = jx - ix
     int n = jx - ix + 1;
     if (n < M)
        usar método simple aquí
     else {
        int mx = (ix + jx) / 2; int my = (iy + jy) / 2;
        vector<bool> U(n+1), V(n+1), W1(n+1), W2(n+1);
        mult(x, y, U, mx + 1, jx, my + 1, jy);
        mult(x, y, V, ix, mx, iy, my);
        decala_izquierda(U, n);
        W1 = x[ix..mx] + y[iy..my];
        W2 = x[mx + 1..jx] + y[my + 1..jy];
        // añadir un bit a W1 y W2 si es necesario
        vector<bool> W;
        W = mult(W1, W2, ...);
        W = W - (U + V);
        decala_izquierda(W, n / 2);
        Z = U + W + V;
     }
}
```

Ordenación por fusión

El algoritmo de ordenación por fusión (mergesort) fue uno de los primeros algoritmos eficentes de ordenación jamás propuestos. Diversas variantes de este algoritmo son particularmente útiles para la ordenación de datos residentes en memoria externa. El propio mergesort es un método muy eficaz para la ordenación de listas enlazadas.

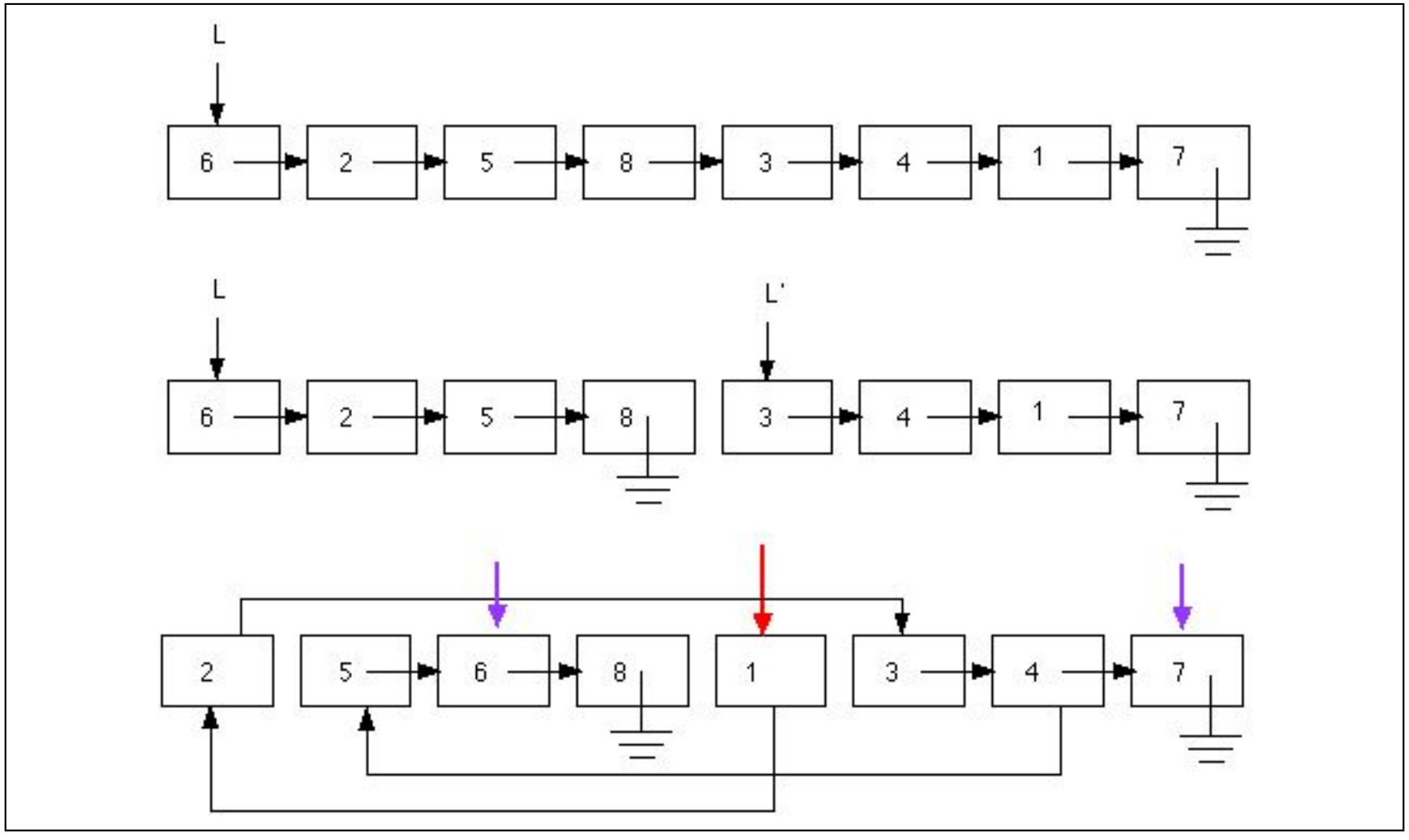
La idea básica es simple: se divide la secuencia de datos a ordenar en dos subsecuencias de igual o similar tamaño, se ordena recursivamente cada una de ellas, y finalmente se obtiene una secuencia ordenada fusionando las dos subsecuencias ordenadas. Si la secuencia es suficientemente pequeña puede emplearse un método más simple (y eficaz para entradas de tamaño reducido).

Supondremos que nuestra entrada es una lista enlazada L conteniendo una secuencia de elementos $x_1, \ldots, x_n$. Cada elemento se almacena en un nodo con dos campos: `info` contiene el elemento y `next` es un apuntador al siguiente nodo de la lista (indicaremos que un nodo no tiene sucesor con el valor especial `next` = 0). La propia lista L es, de hecho, un apuntador a su primer nodo.

```
void split(node<T>* L1, node<T>*& L2, int n) {
        node<T>* p = L1;
        while (n > 1) {
             p = p -> next;
             --n;
        }
        L2 = p -> next; p -> next = 0;
}
```

```
void mergesort(node<T>*& L, int n) {
     if (n > 1) {
        node<T>* L';
        int m = n / 2; split(L, L', m);
        mergesort(L, m);
        mergesort(L', n - m);
       // fusiona L y L' ``destructivamente'' sobre L
        L = merge(L, L');
}
```

```
node<T>* merge(node<T>*& L, node<T>*& L’) {
          if (L == 0) { return L’; }
          if (L’ == 0) { return L; }
          if (L -> info <= L’ -> info) {
              L -> next = merge(L -> next, L’);
              return L;
          } else {
              L’ -> next = merge(L, L’ -> next);
              return L’;
          }
}
```


L
6
2
5
8
3
4
1
7
L
L'
6
2
5
8
3
4
1
7
2
5
6
8
1
3
4
7

Cada elemento de L y L' es "visitado" exactamente una vez, por lo que el coste de la operación `merge` es proporcional a la suma de los tamaños de las listas L y L'; es decir, su coste es $\Theta(n)$.
El coste de mergesort viene descrito por la recurrencia

$$\begin{aligned} M(n) &= \Theta(n) + M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + M\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &= \Theta(n) + 2 \cdot M\left(\frac{n}{2}\right) \end{aligned}$$

cuya solución es $M(n) = \Theta(n \log n)$, aplicando el segundo caso del Teorema I.

Algoritmo de Strassen

El algoritmo convencional de multiplicación de matrices tiene coste $\Theta(n^3)$ para multiplicar dos matrices $n \times n$.

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Para abordar una solución con el esquema divide y vencerás se descomponen las matrices en bloques:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

donde $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$, etc.
Cada bloque de C requiere dos multiplicaciones de bloques de tamaño $n/2 \times n/2$ y dos sumas cuyo coste es $\Theta(n^2)$. El coste del algoritmo divide y vencerás así planteado tendría coste

$$M(n) = \Theta(n^2) + 8 \cdot M(n/2),$$

es decir, $M(n) = \Theta(n^3)$.
Para conseguir una solución más eficiente debemos reducir el número de llamadas recursivas.

Strassen (1969) propuso una forma de hacer justamente esto. En el libro de Brassard & Bratley se detalla una posible manera de hallar la serie de fórmulas que producen el resultado deseado. Un factor que complica las cosas es que la multiplicación de matrices no es conmutativa, a diferencia de lo que sucede con la multiplicación de enteros.
Se obtienen las siguientes 7 matrices $n/2 \times n/2$, mediante 7 productos y 14 adiciones/sustracciones

$$M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12})$$
$$M_2 = A_{11} \cdot B_{11}$$
$$M_3 = A_{12} \cdot B_{21}$$
$$M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{12})$$
$$M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$$
$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$$
$$M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$$

Mediante 10 adiciones/sustracciones más, podemos obtener los bloques de la matriz resultante C:

$$
\begin{aligned}
C_{11} &= M_2 + M_3 \\
C_{12} &= M_1 + M_2 + M_5 + M_6 \\
C_{21} &= M_1 + M_2 + M_4 - M_7 \\
C_{22} &= M_1 + M_2 + M_4 + M_5
\end{aligned}
$$

Puesto que las operaciones aditivas tienen coste $\Theta(n^2)$, el coste del algoritmo de Strassen viene dado por la recurrencia

$$M(n) = \Theta(n^2) + 7 \cdot M(n/2),$$

cuya solución es $\Theta(n^{\log_2 7}) = \Theta(n^{2{,}807\ldots})$.

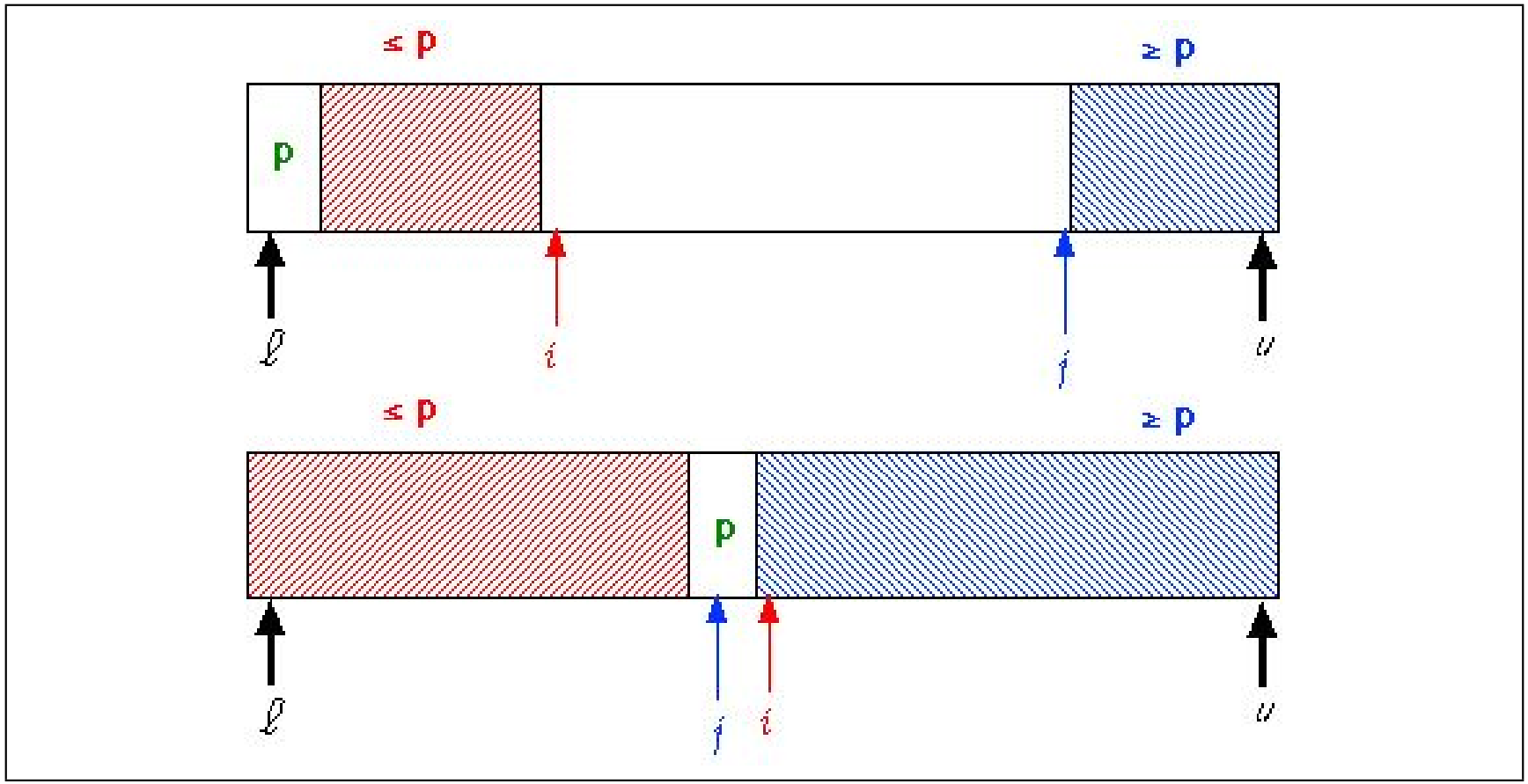
El algoritmo de Strassen tuvo un enorme impacto teórico ya que fue el primer algoritmo de multiplicación de matrices cuya complejidad era $o(n^3)$; lo cual tenía también implicaciones en el desarrollo de algoritmos más eficientes para el cálculo de matrices inversas, de determinantes, etc. Además era uno de los primeros casos en que las técnicas algorítmicas superaban lo que hasta el momento parecía una barrera infranqueable.

En años posteriores se han obtenido algoritmos todavía más eficientes. El más eficiente conocido es el de Coppersmith y Winograd (1986) cuyo coste es $\Theta(n^{2,376\ldots})$.

El algoritmo de Strassen no es competitivo en la práctica, excepto si n es muy grande ($n \gg 500$), ya que las constantes y términos de orden inferior del coste son muy grandes.

Quicksort

Quicksort (Hoare, 1962) es un algoritmo de ordenación que usa el principio de divide y vencerás, pero a diferencia de los ejemplos anteriores, no garantiza que cada subejemplar tendrá un tamaño que es fracción del tamaño original.



El procedimiento de partición sitúa a un elemento, el pivote, en su lugar apropiado. Luego no queda más que ordenar los segmentos que quedan a su izquierda y a su derecha. Mientras que en mergesort la división es simple y el trabajo se realiza durante la fase de combinación, en quicksort sucede lo contrario.
Para ordenar el segmento $A[\ell..u]$ el algoritmo queda así

```
void quicksort(vector<T>& A, int l, int u) {
       if (u - l + 1 <= M)
          usar un algoritmo sencillo de ordenación
       else {
          int k;
          particion(A, l, u, k);
          quicksort(A, l, k - 1);
          quicksort(A, k + 1, u);
       }
}
```

En vez de usar un algoritmo de ordenación simple (p.e. ordenación por inserción) con cada segmento de M o menos elementos, puede ordenarse mediante el algoritmo de inserción al final:

$\texttt{quicksort}(A, 0, A.\texttt{size}() - 1)$
$\texttt{insercion}(A, 0, A.\texttt{size}() - 1)$

Puesto que el vector A está quasi-ordenado tras aplicar `quicksort`, el último paso se hace en tiempo $\Theta(n)$, donde $n = A.\texttt{size}()$.
Se estima que la elección óptima para el umbral o corte de recursión M oscila entre 20 y 25.

Existen muchas formas posibles de realizar la partición. En Bentley & McIlroy (1993) se discute un procedimiento de partición muy eficiente, incluso si hay elementos repetidos. Aquí examinamos un algoritmo básico, pero razonablemente eficaz.
Se mantienen dos índices i y j de tal modo que $A[\ell + 1..i - 1]$ contiene elementos menores o iguales que el pivote p, y $A[j + 1..u]$ contiene elementos mayores o iguales. Los índices barren el segmento (de izquierda a derecha, y de derecha a izquierda, respectivamente), hasta que $A[i] > p$ y $A[j] < p$ o se cruzan ($i = j + 1$).

```
void particion(vector<T>& A, int l, int u, int& k) {
     int i = l; int j = u + 1; T pv = A[l];
     for ( ; ; ) {
          do ++i; while (A[i] < pv);
          do --j; while (A[j] > pv);
          if (i >= j) break;
          swap(A[i], A[j]);
     }
     swap(A[l], A[j]); k = j;
}
```

El coste de quicksort en caso peor es $\Theta(n^2)$ y por lo tanto poco atractivo en términos prácticos. Esto ocurre si en todos o la gran mayoría de los casos uno de los subsegementos contiene muy pocos elementos y el otro casi todos, p.e. así sucede si el vector está ordenado creciente o decrecientemente. El coste de la partición es $\Theta(n)$ y entonces tenemos

$$\begin{aligned} Q(n) &= \Theta(n) + Q(n-1) + Q(0) \\ &= \Theta(n) + Q(n-1) = \Theta(n) + \Theta(n-1) + Q(n-2) \\ &= \cdots = \sum_{i=0}^{n} \Theta(i) = \Theta\left(\sum_{0 \le i \le n} i\right) \\ &= \Theta(n^2). \end{aligned}$$

Sin embargo, en promedio, el pivote quedará más o menos centrado hacia la mitad del segmento como sería deseable —justificando que quicksort sea considerado un algoritmo de divide y vencerás.

Para analizar el comportamiento de quicksort sólo importa el orden relativo de los elementos. También podemos investigar exclusivamente el número de comparaciones entre elementos, ya que el coste total es proporcional a dicho número. Supongamos que cualquiera de los $n!$ ordenes relativos posibles tiene idéntica probabilidad, y sea q_n el número medio de comparaciones.

$$
\begin{aligned}
q_n &= \sum_{1\le j\le n} \mathbb{E}[\text{núm. compar.}|\text{pivote es } j\text{-ésimo}] \\
&\qquad \times \Pr\{\text{pivote es } j\text{-ésimo}\} \\
&= \sum_{1\le j\le n} (n-1+q_{j-1}+q_{n-j}) \times \frac{1}{n} \\
&= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1\le j\le n} (q_{j-1}+q_{n-j}) \\
&= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0\le j<n} q_j
\end{aligned}
$$

Multiplicamos $(n+1)$ por q_{n+1} y sustraemos $n \cdot q_n$ para eliminar los factores $1/n$ y los sumatorios:

$$(n+1)q_{n+1} - nq_n = (n+1)n + 2 \sum_{0 \le j < n+1} q_k - n(n-1) - 2 \sum_{0 \le j < n} q_k$$
$$= 2n + 2q_n.$$

Por lo tanto

$$(n+1)q_{n+1} = 2n + (n+2)q_n.$$

Pasando $(n+1)$ al otro lado y "desplegando"

$$q_{n+1} = \frac{2n}{n+1} + \frac{n+2}{n+1}\frac{2(n-1)}{n} + \frac{n+2}{n}q_{n-1}$$

$$= \frac{2n}{n+1} + \frac{n+2}{n+1}\frac{2(n-1)}{n} + \frac{n+2}{n}\frac{2(n-2)}{n-1} + \frac{n+2}{n-1}q_{n-2}$$

$$= \ldots = \frac{2n}{n+1} + 2(n+2)\sum_{i=1}^{k}\frac{n-i}{(n-i+1)(n-i+2)} + \frac{n+2}{n-k+1}q_{n-k}.$$

Tomando $k = n$, $(n+2)q_0 = 0$ y reorganizando la suma

$$q_{n+1} = \frac{2n}{n+1} + 2(n+2)\sum_{i=0}^{n-1} \frac{i}{(i+1)(i+2)}.$$

Puesto que

$$\frac{i}{(i+1)(i+2)} = \frac{2}{i+2} - \frac{1}{i+1},$$

tenemos

$$q_{n+1} = \frac{2n}{n+1} + 2(n+2)\left(2\sum_{i=2}^{n+1}\frac{1}{i} - \sum_{i=1}^{n}\frac{1}{i}\right).$$

Usando $H_n = \sum_{i=1}^{n} 1/i$ podemos expresar q_{n+1} como

$$\begin{aligned} q_{n+1} &= \frac{2n}{n+1} + 2(n+2)(2H_{n+1} - 2 - H_n) \\ &= \frac{2n}{n+1} + 2(n+2)\left(\frac{1}{n+1} + H_{n+1} - 2\right) \\ &= \frac{2n + 2(n+2)}{n+1} + 2(n+2)H_{n+1} - 4(n+2) \\ &= 4 + 2(n+2)H_{n+1} - 4(n+2) = 2(n+2)H_{n+1} - 4(n+1). \end{aligned}$$

Luego

$$q_n = 2(n+1)H_n - 4n = 2n\ln n + \mathcal{O}(n).$$

Quickselect

El problema de la selección consiste en hallar el j-ésimo de entre n elementos dados. En concreto, dado un vector A con n elementos y un rango j, $1 \leq j \leq n$, un algoritmo de selección debe hallar el j-ésimo elemento en orden ascendente. Si $j = 1$ entonces hay que encontrar el mínimo, si $j = n$ entonces hay que hallar el máximo, si $j = \lfloor n/2 \rfloor$ entonces debemos hallar la mediana, etc.

Es fácil resolver el problema con coste $\Theta(n \log n)$ ordenando previamente el vector y con coste $\Theta(j \cdot n)$, recorriendo el vector y manteniendo los j elementos menores de entre los ya examinados. Con las estructuras de datos apropiadas puede rebajarse el coste a $\Theta(n \log j)$, lo cual no supone una mejora sobre la primera alternativa si $j = \Theta(n)$.
Quickselect (Hoare, 1962), también llamado Find y one-sided quicksort, es una variante del algoritmo quicksort para la selección del j-ésimo de entre n elementos.

Supongamos que efectuamos una partición de un subvector $A[\ell..u]$, conteniendo los elementos ℓ-ésimo a u-ésimo de A, y tal que $\ell \leq j \leq u$, respecto a un pivote p. Una vez finalizada la partición, supongamos que el pivote acaba en la posición k.
Por tanto, en $A[\ell..k-1]$ están los elementos ℓ-ésimo a $(k-1)$-ésimo de A y en $A[k+1..u]$ están los elementos $(k+1)$-ésimo a u-ésimo. Si $j = k$ hemos acabado ya que hemos encontrado el elemento solicitado. Si $j < k$ entonces procedemos recursivamente en el subvector de la izquierda $A[\ell..k-1]$ y si $j > k$ entonces encontraremos el elemento buscado en el subvector $A[k+1..u]$.

```
int quickselect(vector<T>& A, int l, int j, int u) {
     if (l == u) return A[l];
     int k;
     particion(A, l, u, k);
     if (j == k) return A[k];
     if (j < k)
         return quickselect(A, l, j, k - 1);
     else
         return quickselect(A, k+1, j, u);
}
```

Puesto que `quickselect` es recursiva final es muy simple obtener una versión iterativa eficiente que no necesita espacio auxiliar.
En caso peor, el coste de `quickselect` es $\Theta(n^2)$. Sin embargo, su coste promedio es $\Theta(n)$ donde la constante de proporcionalidad depende del cociente j/n.
Knuth (1971) ha demostrado que $C_n^{(j)}$, el número medio de comparaciones necesarias para seleccionar el j-ésimo de entre n es:

$$C_n^{(j)} = 2((n+1)H_n - (n+3-j)H_{n+1-j} - (j+2)H_j + n + 3)$$

El valor máximo se alcanza para $j = \lfloor n/2 \rfloor$; entonces $C_n^{(j)} = 2(\ln 2 + 1)n + o(n)$.

Algoritmo de selección de Rivest y Floyd

Podemos obtener un algoritmo cuyo coste en caso peor sea lineal si garantizamos que cada paso el pivote escogido divide el vector en dos subvectores cuya talla sea una fracción de la talla del vector original, con coste $\mathcal{O}(n)$ (incluyendo la selección del pivote).
Entonces, en caso peor,

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

donde $p < 1$. Puesto que $\log_{1/p} 1 = 0 < 1$ concluímos que $C(n) = \mathcal{O}(n)$. Por otra parte, es bastante obvio que $C(n) = \Omega(n)$; luego, $C(n) = \Theta(n)$.
Este es el principio del algoritmo de selección de Rivest y Floyd (1970).

La única diferencia entre el algoritmo de selección de Hoare y el de Rivest y Floyd reside en la manera en que elegimos los pivotes. El algoritmo de Rivest y Floyd obtiene un pivote de "calidad" empleando el propio algoritmo, recursivamente, para seleccionar los pivotes!

- El algoritmo de Rivest y Floyd calcula una pseudomediana y elige ésta como pivote.

- El algoritmo de Rivest y Floyd calcula una pseudomediana y elige ésta como pivote.
- Se subdivide el subvector $A[\ell..u]$ en bloques de q elementos (excepto posiblemente el último bloque), con q constante e impar, y para cada uno de ellos se obtiene su mediana. El coste de esta fase es $\Theta(n)$ y como resultado se obtiene un vector con $\lceil n/q \rceil$ elementos.

- El algoritmo de Rivest y Floyd calcula una pseudomediana y elige ésta como pivote.
- Se subdivide el subvector $A[\ell..u]$ en bloques de q elementos (excepto posiblemente el último bloque), con q constante e impar, y para cada uno de ellos se obtiene su mediana. El coste de esta fase es $\Theta(n)$ y como resultado se obtiene un vector con $\lceil n/q \rceil$ elementos.
- Sobre el vector de medianas se aplica el algoritmo de selección para hallar la mediana.

- El algoritmo de Rivest y Floyd calcula una pseudomediana y elige ésta como pivote.
- Se subdivide el subvector $A[\ell..u]$ en bloques de q elementos (excepto posiblemente el último bloque), con q constante e impar, y para cada uno de ellos se obtiene su mediana. El coste de esta fase es $\Theta(n)$ y como resultado se obtiene un vector con $\lceil n/q \rceil$ elementos.
- Sobre el vector de medianas se aplica el algoritmo de selección para hallar la mediana.
- Como el pivote seleccionado es la pseudomediana de los n elementos originales, ello garantiza que al menos $n/2q \cdot q/2 = n/4$ elementos son mayores que el pivote.

Por lo tanto, el coste $C(n)$ en caso peor satisface

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4).$$

Puede demostrarse que $C(n) = \mathcal{O}(n)$ si

$$\frac{3}{4} + \frac{1}{q} < 1.$$

Rivest y Floyd usaron el valor $q = 5$ en su formulación original de este algoritmo.

Segmento de suma máxima

Dado un vector $A[1..n]$ de enteros, determinar cúal es el segmento de suma máxima en A, es decir, determinar i y j, $i \leq j+1$, tales que el segmento $A[i..j]$ tiene suma máxima. Un segmento vacío ($i = j+1$) es la respuesta válida si todos los elementos de A son negativos.
Denotaremos $\sigma_{i,j}$ la suma del segmento $A[i..j]$.
Una primera solución ingenua tiene coste $\Theta(n^3)$:

```
imax = 0; jmax = -1; max = 0;
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j) {
        int sum = 0;
        for (int k = i; k <= j; ++k)
            sum += A[k];
        // sum == σ_{i,j}
        if (sum > max) {
            imax = i; jmax = j; max = sum;
        }
    }
```

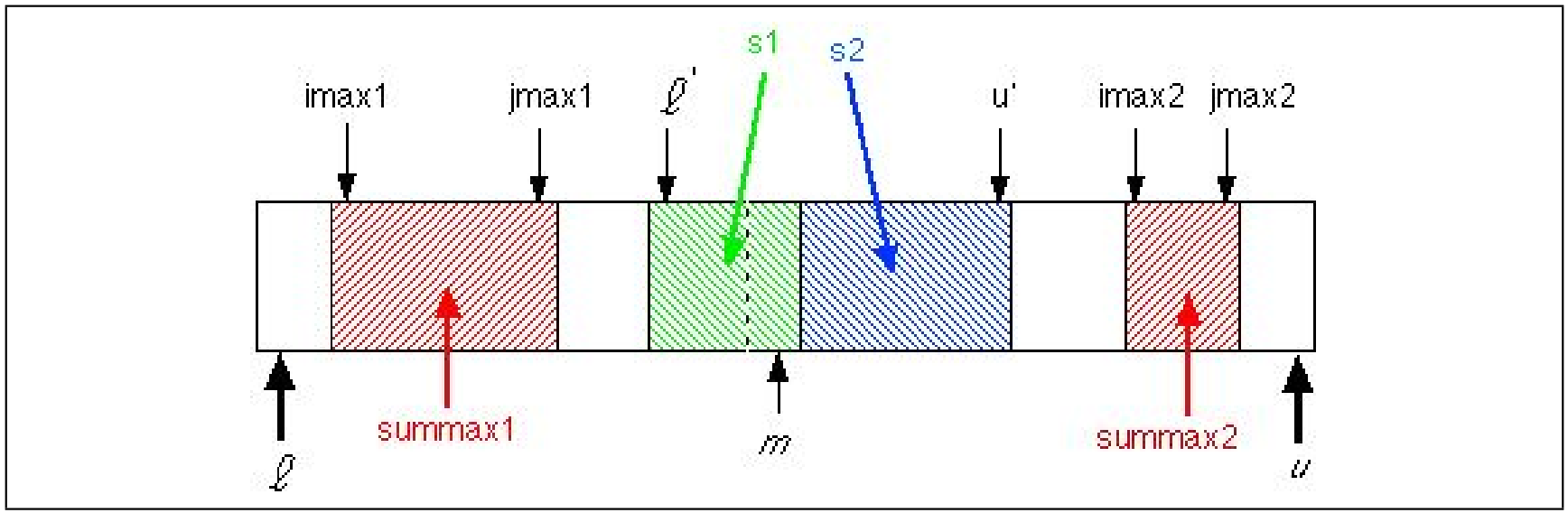
Se puede obtener una mejora sustancial sobre la solución previa observando que

$$\sigma_{i,j+1} = \sigma_{i,j} + A[j+1]$$

```
imax = 0; jmax = -1; max = 0;
for (int i = 0; i < n; ++i) {
     int sum = 0;
     for (int j = i; j < n; ++j) {
          // sum == σ_{i,j-1}
          sum += A[j];
          // sum == σ_{i,j}
          if (sum > max) {
             imax = i; jmax = j; max = sum;
          }
     }
}
```

Esta nueva solución tiene coste $\Theta(n^2)$. Usando divide y vencerás podemos obtener la solución de un modo más eficiente.

Supongamos que dividimos el segmeto en curso $A[\ell..u]$ (con más de un elemento, en otro caso podemos dar la solución directa) en dos mitades (o casi) y resolvemos el problema de manera independiente para cada una de ellas. El segmento de suma máxima en $A[\ell..u]$ será uno de los dos segmentos obtenidos recursivamente o bien un segmento no vacío que contiene elementos de las dos mitades en que se dividió $A[\ell..u]$, "atravesando" el punto de corte ...



Para determinar el segmento de suma máxima en $A[\ell..u]$, siendo el punto de corte m, $\ell \leq m < u$, hemos de averiguar cúal es el segmento de la forma $A[r..m]$, es decir, que necesariamente acaba en la posición m, de suma máxima en $A[\ell..m]$. Sea s_1 su suma y ℓ' su extremo inferior (su otro extremo es, por definición, m).

Análogamente, deberemos averiguar cúal es el segmento de la forma $A[m+1..s]$ de suma máxima en $A[m+1..u]$, siendo su extremo superior u' y suma s_2 . Entonces el segmento de suma máxima en $A[\ell..u]$ es el de mayor suma entre el segmento de suma máxima en $A[\ell..m]$, el segmento de suma máxima en $A[m+1..u]$, y el segmento $A[\ell'..u']$ (su suma es s_1+s_2).

Tanto s_1 y s_2 como sus extremos se pueden obtener mediante dos sencillos bucles, pues $A[\ell'..m]$ tiene suma máxima entre todos los segmentos de la forma $A[r..m]$, $\ell \leq r \leq m$, y $A[m+1..u']$ tiene suma máxima entre todos los que son de la forma $A[m+1..s]$, $m+1 \leq s \leq u$.

```
void ssm(const vector<T>& A, int l, int u,
         int& imax, int& jmax, int& summax) {
    if (u - l + 1 <= 1) {
       if (u - l + 1 == 0) {
         imax = l; jmax = u; summax = 0;
       } else if (A[l] >= 0) {
         imax = l; jmax = u; summax = A[l];
       } else {
         imax = l; jmax = l - 1; summax = 0;
       }
       return;
    }
    int m = (l + u) / 2;
    int imax1, jmax1, imax2, jmax2, summax1, summax2;
    ssm(A, l, m, imax1, jmax1, summax1);
    ssm(A, m + 1, u, imax2, jmax2, summax2);
```

→

```
    int l' = m; int s1 = 0;
    for (int sum = 0, int r = m; r >= l; --r) {
            sum += A[r];
            if (sum > s1)  { l' = r; s1 = sum; }
    }

    int u' = m; int s2 = 0;
    for (int sum = 0, int s = m + 1; s <= u; ++r) {
            sum += A[s];
            if (sum > s2)  { u' = s; s2 = sum; }
    }

    summax = max(summax1, summax2, s1 + s2);
    imax = ...; // imax1, imax2 ó l'
    jmax = ...; // jmax1, jmax2 ó u'
}
```

Puesto que el coste de dividir y combinar es $\Theta(n)$, el coste del algoritmo divide y vencerás es

$$S(n) = \Theta(n) + 2 \cdot S(n/2),$$

es decir, $S(n) = \Theta(n \log n)$.

Pero podemos mejorar la eficiencia hasta conseguir que sea óptima, utilizando una inmersión de eficiencia. En concreto, la nueva versión de `ssm`, aplicada a un segmento $A[\ell..u]$ retornará:

1. Los extremos del segmento de suma máxima $imax$ y $jmax$ y su valor $summax$;
2. El extremo $ider$ del segmento de suma máxima de entre aquellos que incluyen al extremo superior u, y su suma $sder$;
3. El extremo $jizq$ del segmento de suma máxima de entre aquellos que incluyen al extremo inferior ℓ, y su suma $sizq$;
4. La suma sum de todos los elementos del segmento.

Si efectuamos las llamadas

```
m = (l + u) / 2;
ssm(A, l, m, imax1, jmax1, summax1,
        ider1, sder1, jizq1, sizq1, sum1);
ssm(A, m+1, u, imax2, jmax2, summax2,
        ider2, sder2, jizq2, sizq2, sum2);
```

podemos calcular los valores correspondientes a $A[\ell..u]$ teniendo en cuenta lo siguiente:

- El segmento de suma máxima será el correspondiente a

$$\text{máx}(summax1, summax2, sder1 + sizq2)$$

- El segmento de suma máxima con extremo superior en u será el correspondiente a

$$\text{máx}(sder2, sder1 + sum2)$$

- El segmento de suma máxima con extremo inferior en ℓ será el correspondiente a

$$\max(sizq1, sum1 + sizq2)$$

- La suma del segmento es $sum1 + sum2$.

Los valores de los extremos $imax$, $jmax$, $ider$ y $jizq$ se calculan en consonancia: p.e. si $\max(sizq1, sum1 + sizq2)$ es $sizq1$ entonces $jizq = jizq1$, sino $\max(sizq1, sum1 + sizq2) = sum1 + sizq2$ y por tanto $jizq = jizq2$.
Gracias a la inmersión de eficiencia, el coste no recursivo es ahora $\Theta(1)$ y ya que

$$S(n) = \Theta(1) + 2 \cdot S(n/2),$$

el coste del nuevo algoritmo es $S(n) = \Theta(n)$ (usar el caso 3 del Teorema 1).

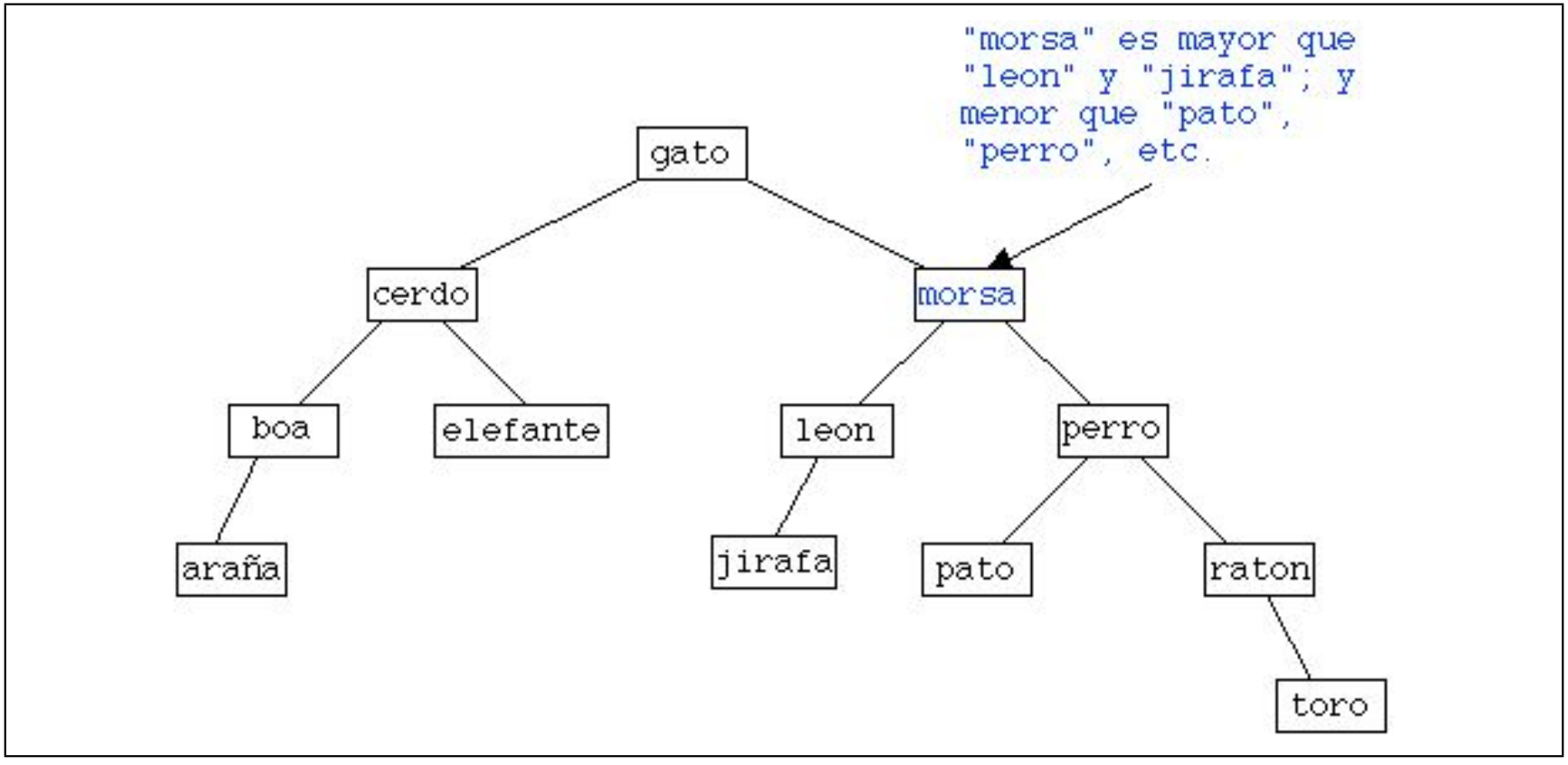
1. Análisis de Algoritmos
2. Divide y Vencerás
3. Estructuras de Datos
   - Árboles binarios de búsqueda
   - Colas de prioridad
4. Algoritmos Voraces

Árboles Binarios de Búsqueda

Definición

Un árbol binario de búsqueda T es un árbol binario tal que o es vacío o bien contiene un elemento x y satisface

1. Los subárboles izquierdo y derecho, L y R, respectivamente, son árboles binarios de búsqueda.
2. Para todo elemento y de L, $\texttt{clave}(y) < \texttt{clave}(x)$, y para todo elemento z de R, $\texttt{clave}(z) > \texttt{clave}(x)$.


"morsa" es mayor que
"leon" y "jirafa"; y
menor que "pato",
"perro", etc.
gato
cerdo
morsa
boa
elefante
leon
perro
araña
jirafa
pato
raton
toro

Lema

Un recorrido en inorden de un árbol binario de búsqueda T visita los elementos de T por orden creciente de clave.

Búsqueda en BSTs

```
template <typename K, typename V>
class BST {
        ...
        void lookup(const K& key, V& val, bool& enc) const;
        void insert(const K& key, const V& val);
        void delete(const K& key);
        ...
private:
        struct nodo {
                 K clave_;
                V valor_;
                nodo *izq_, *der_;
        };
        nodo* raiz_;
        void lookup(nodo* p, const K& key, V& val,
                                  bool& enc) const;
        ...
};
```

Búsqueda en BSTs (versión recursiva)

```
template <typename K, typename V>
void BST<K,V>::lookup(const K& key, V& val, bool& enc) {
        lookup(raiz_, key, val, enc);
}

template <typename K, typename V>
void BST<K,V>::lookup(nodo* p, const K& key, V& val,
                                                bool& enc) {
        if (p == NULL) {
                enc = false; return;
        }
        if (p -> clave_ == key) {
                enc = true; val = p -> valor_;
        }
        else if (key < p -> clave_)
                lookup(p -> izq_, key, val, enc);
        else
                lookup(p -> der_, key, val, enc);
}
```

Puesto que el algoritmo es recursivo final es inmediato obtener una versión iterativa.

Búsqueda en BSTs (versión iterativa)

```
template <typename K, typename V>
void BST<K,V>::lookup(const K& key, V& val, bool& enc) {
        nodo* p = raiz_; enc = false;
        while (p != NULL && !enc) {
                if (key < p -> clave_)
                        p = p -> izq_;
                else if (p -> clave_ < key)
                        p = p -> der_;
                else
                        enc = true;
        }
}
```

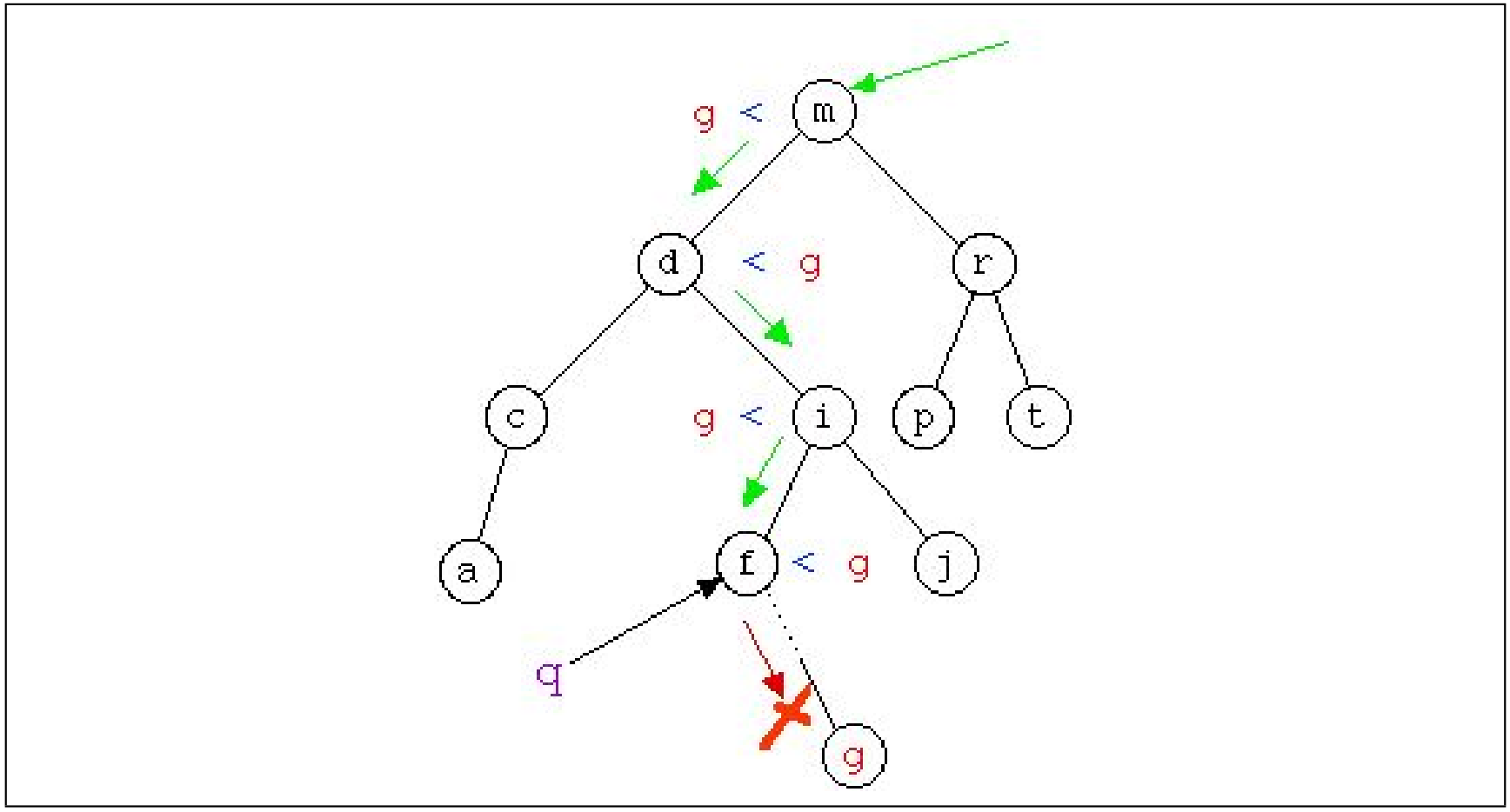
Inserción en BSTs (versión recursiva)

```
// constructora de nodos
template <typename K, typename V>
BST<K,V>::nodo::nodo(const K& key, const V& val) :
clave_(key), valor_(val), izq_(NULL), der_(NULL) {}

template <typename K, typename V>
void BST<K,V>::insert(const K& key, const V& val) {
        raiz_ = insert(raiz_, key, val);
}
```

Inserción en BSTs (versión recursiva)

```
template <typename K, typename V>
BST<K,V>::nodo* BST<K,V>::insert(nodo* p, const K& key,

        if (p == NULL)
                return new nodo(key, val);
        if (key < p -> clave_)
                p -> izq_ = insert(p -> izq_, key, val);
        else if (p -> clave_ < key)
                p -> der_ = insert(p -> der_, key, val);
        return p;
}
```

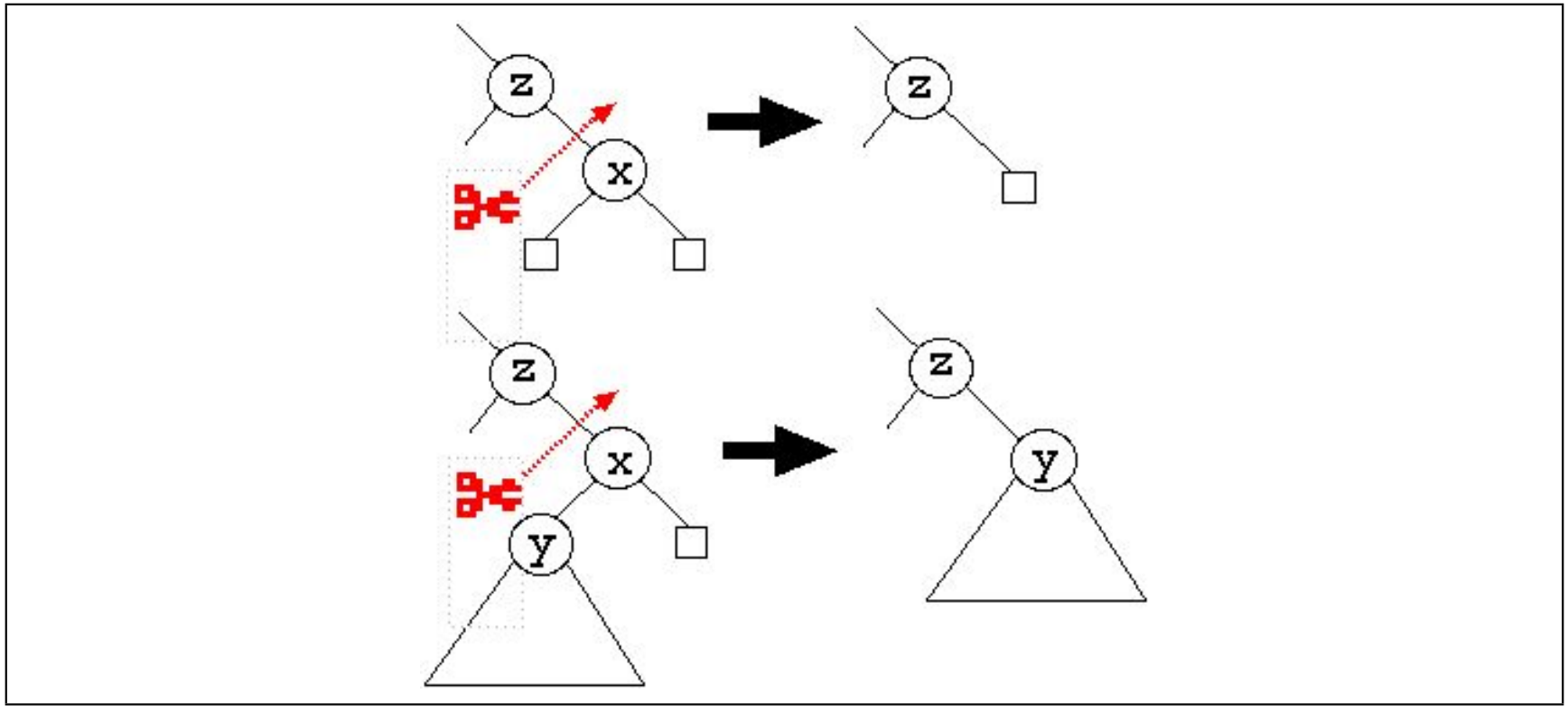
La versión iterativa es más compleja, ya que además de localizar la hoja en la que se ha de realizar la inserción, deberá mantenerse un apuntador *q* al que será padre del nuevo nodo.



Inserción en BSTs (versión iterativa)

```
template <typename K, typename V>
void BST<K,V>::insert(const K& key, const V& val) {
        nodo* p = raiz_; nodo* q = NULL;
        while (p != NULL && p -> clave_ != key) {
               q = p;
               if (key < p -> clave_) p = p -> izq_;
               else                   p = p -> der_;
        }
        if (p == NULL) {
           if (q != NULL) {
               if (key < q -> clave_)
                    q -> izq_ = new nodo(key, val);
               else
                    q -> der_ = new nodo(key, val);
           } else {  // q == NULL, insercion en BST vacío
                raiz_ = new nodo(key, val);
           }
        }
}
```

Sólo nos queda por considerar la eliminación de elementos en BSTs. Si el elemento ha eliminar se encuentra en un nodo cuyos dos subárboles son vacíos Basta eliminar el nodo en cuestión. Otro tanto sucede si el nodo x a eliminar sólo tiene un subárbol no vacío: Basta hacer que la raíz del subárbol no vacío quede como hijo del padre de x.



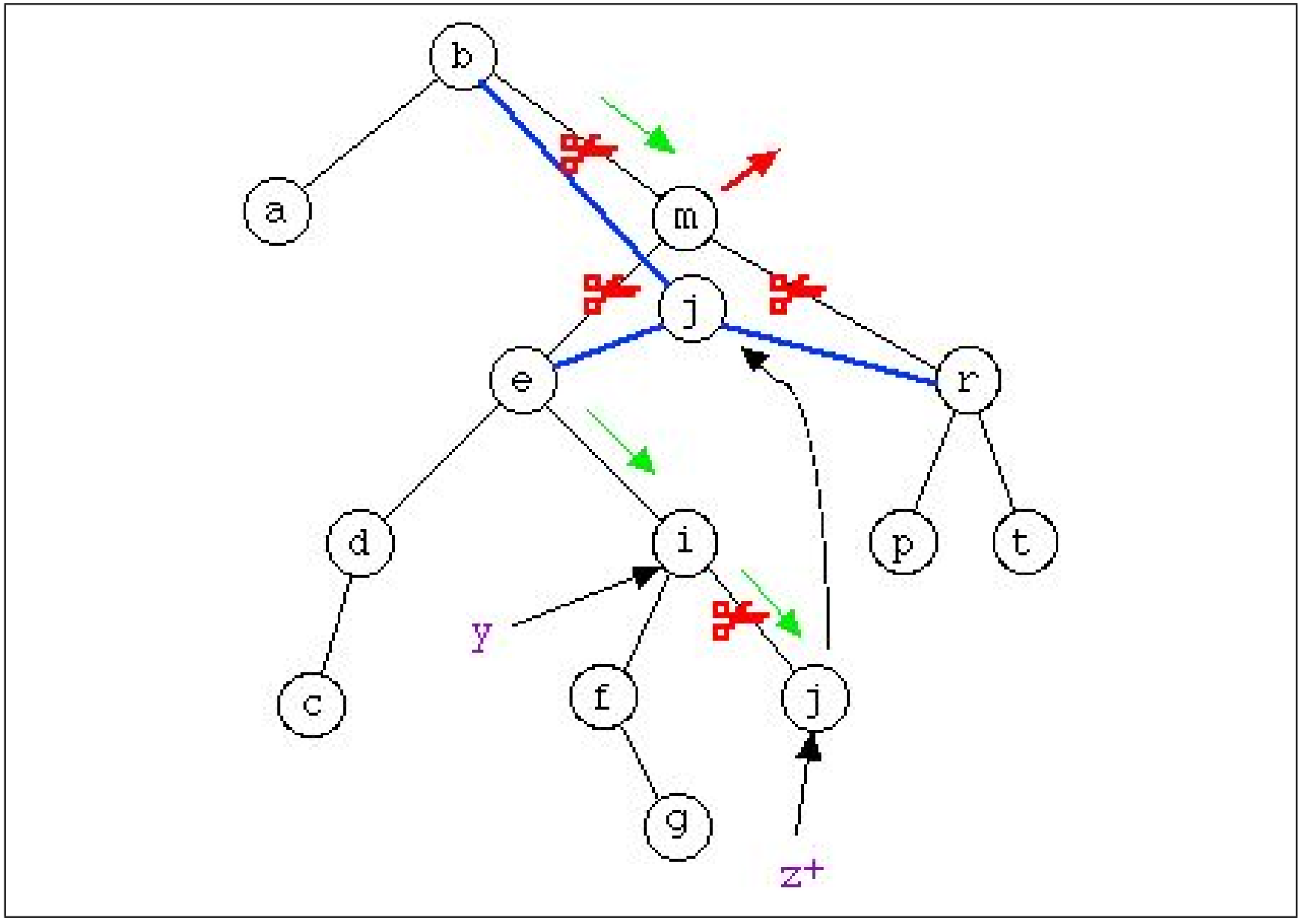
El problema surge si hay que eliminar un nodo que contiene dos subárboles no vacíos. Podemos reformular el problema de la siguiente forma: dados dos BSTs T_1 y T_2 tales que todas las claves de T_1 son menores que las claves de T_2 obtener un nuevo BST que contenga todas las claves: $T = \texttt{juntar}(T_1, T_2)$. Obviamente:

$$\texttt{juntar}(T, \square) = T$$
$$\texttt{juntar}(\square, T) = T$$

En particular, $\texttt{juntar}(\square, \square) = \square$.

Borrado en BSTs

```
template <typename K, typename V>
void BST<K,V>::delete(const K& key) {
         raiz_ = delete(raiz_, key);
}

template <typename K, typename V>
BST<K,V>::nodo* BST<K,V>::delete(nodo* p, const K& key) {
      if (p == NULL) return NULL;
      if (key < p -> clave_)
          p -> izq_ = delete(p -> izq_, key);
      else if (p -> clave_ < key)
          p -> der_ = delete(p -> der_, key);
      else {
          nodo* aux = p;
          p = juntar(p -> izq_, p -> der_);
          delete aux;
      }
      return p;
}
```


b
a
m
j
e
r
d
i
p
t
y
c
f
j
g
z+

Sea z^+ la mayor clave de T_1. Puesto que es mayor que todas las demás en T_1 pero al mismo tiempo menor que cualquier clave de T_2 podemos construir T colocando un nodo raíz que contenga al elemento de clave z^+, a T_2 como subárbol derecho y al resultado de eliminar z^+ de T_1 —llamémosle T_1'— como subárbol izquierdo. Además puesto que z^+ es la mayor clave de T_1 el correspondiente nodo no tiene subárbol derecho, y es el nodo "más a la derecha" en T_1, lo que nos permite desarrollar un procedimiento ad-hoc para eliminarlo.

Borrado en BSTs

```
template <typename K, typename V>
BST<K,V>::nodo* BST<K,V>::juntar(nodo* T1, nodo* T2) {
     if (T1 == NULL) return T2;
     if (T2 == NULL) return T1;
     nodo* z;
     z = elimina_max(T1);
     z -> izq_ = T1; z -> der_ = T2;
     return z;
}

template <typename K, typename V>
BST<K,V>::nodo* BST<K,V>::eliminar_max(nodo*& T) {
     nodo* z = T; nodo* zp = NULL;
     while (z -> der_ != NULL) {
          zp = z; z = z -> der_;
     }
     if (zp == NULL) T = T -> izq_;
     else            zp -> der_ = NULL;
     return z;
}
```

Un razonamiento análogo nos lleva a una versión de `juntar` en la que se emplea la clave mínima z^- del árbol T_2 para que ocupe la raíz del resultado, en el caso en que T_1 y T_2 no son vacíos.
Se ha evidenciado experimentalmente que conviene alternar entre el predecesor z^+ y el sucesor z^- del nodo a eliminar en los borrados (por ejemplo, mediante una decisión aleatoria o con bit que va pasando alternativamente de 0 a 1 y de 1 a 0) y no utilizar sistemáticamente una de las versiones.
Existen otros algoritmos de borrado pero no se estudiarán en la asignatura.

Análisis de la eficiencia de BSTs

Un BST de n elementos puede ser equivalente a una lista, pues puede contener un nodo sin hijos y $n-1$ con sólo un hijo (p.e. si insertamos una secuencia de n elementos con claves crecientes en un BST inicialmente vacío). Un BST con estas características tiene altura n. En caso peor, el coste en caso peor de una Búsqueda, inserción o Borrado en dicho árbol es por lo tanto $\Theta(n)$. En general, una Búsqueda, inserción o Borrado en un BST de altura h tendrá coste $\Theta(h)$ en caso peor. Como función de n, la altura de un árbol puede llegar a ser n y de ahí que el coste de las diversas operaciones es, en caso peor, $\Theta(n)$.

Pero normalmente el coste de estas operaciones será menor. Supongamos que cualquier orden de inserción de los elementos es equiprobable. Para búsquedas con éxito supondremos que buscamos cualquiera de las n claves con probabilidad $1/n$. Para búsquedas sin éxito o inserciones supondremos que finalizamos en cualquiera de las $n+1$ hojas con igual probabilidad.
El coste de una de estas operaciones va a ser proporcional al número de comparaciones que habrá que efectuar entre la clave dada y las claves de los nodos examinados. Sea $C(n)$ el número medio de comparaciones y $C(n;j)$ el número medio de comparaciones si la raíz está ocupada por la j-ésima clave.

$$C(n) = \sum_{1 \leq j \leq n} C(n;j) \times \mathbb{P}\Big[\text{raíz es } j\text{-ésima}\Big] .$$

$$
\begin{aligned}
C(n) &= \frac{1}{n} \sum_{1 \le j \le n} C(n;j) \\
&= 1 + \frac{1}{n} \sum_{1 \le j \le n} \left(\frac{1}{n} + \frac{j-1}{n} C(j-1) + \frac{n-j}{n} C(n-j) \right) \\
&= 1 + \frac{1}{n^2} \sum_{0 \le k < n} (k \cdot C(k) + (n-1-j) \cdot C(n-1-j)) \\
&= 1 + \frac{2}{n^2} \sum_{0 \le j < n} k \cdot C(j).
\end{aligned}
$$

Otra forma de plantear esto es calcular $I(n)$, el valor medio de la longitud de caminos internos (IPL). Dado un BST su longitud de caminos internos es la suma de las distancias desde la raíz a cada uno de los nodos.

$$
C(n) = 1 + \frac{I(n)}{n}
$$

La longitud media de caminos internos satisface la recurrencia

$$I(n) = n - 1 + \frac{2}{n} \sum_{0 \leq j < n} I(j), \qquad I(0) = 0.$$

Esto es así porque cada nodo que no sea la raíz contribuye al IPL total 1 más su contribución al IPL del subárbol en que se encuentre. Otra razón por la que resulta interesante estudiar el IPL medio es porque el coste de construir un BST de tamaño n mediante n inserciones es proporcional al IPL.
La recurrencia satisfecha por $I(n)$ es idéntica a la recurrencia del coste esperado de Quicksort. Por lo tanto

$$I(n) = \Theta(n \log n), C(n) = \Theta(\log n).$$

Podemos asociar a cada ejecución de quicksort un BST como sigue. En la raíz se coloca el pivote de la fase inicial; los subárboles izquierdo y derecho corresponden a las ejecuciones recursivas de quicksort sobre los subvectores a la izquierda y a la derecha del pivote. Consideremos un subárbol cualquiera de este BST. Todos los elementos del subárbol, salvo la raíz, son comparados con el nodo raíz durante la fase a la que corresponde ese subárbol. Y recíprocamente el pivote de una determinada fase ha sido comparado con los elementos (pivotes) que son sus antecesores en el árbol y ya no se comparará con ningún otro pivote. Por lo tanto, el número de comparaciones en las que interviene un cierto elemento no siendo el pivote es igual a su distancia a la raíz del BST. Por lo tanto, el IPL medio coincide con el número medio de comparaciones que hace quicksort

Otras operaciones

Los BSTs permiten otras varias operaciones siendo los algoritmos correspondientes simples y con costes (promedio) razonables. Algunas operaciones como las de búsqueda o borrado por rango (e.g. busca el 17^o elemento) o de cálculo del rango de un elemento dado requieren una ligera modificación de la implementación estándar, de tal forma que cada nodo contenga información relativa a tamaño del subárbol en él enraizado. Concluimos esta parte con un ejemplo concreto: dadas dos claves k_1 y k_2, $k_1 < k_2$ se precisa una operación que devuelve una lista ordenada de todos los elementos cuya clave k está comprendida entre las dos dadas, esto es, $k_1 \leq k \leq k_2$.

Si el BST es vacío, la lista a devolver es también vacía. Supongamos que el BST no es vacío y que la clave en la raíz es k. Si $k < k_1$ entonces todas las claves buscadas deben encontrarse, si las hay, en el subárbol derecho. Análogamente, si $k_2 < k$ se proseguirá la búsqueda recursivamente en el subárbol izquierdo. Finalmente, si $k_1 \leq k \leq k_2$ entonces puede haber claves que caen dentro del intervalo tanto en el subárbol izquierdo como en el derecho. Para respetar el orden creciente en la lista, deberá buscarse recursivamente a la izquierda, luego listar la raíz y finalmente buscarse recursivamente a la derecha.

Búsqueda por rango

```
template <typename K, typename V>
void BST<K, V>::range_search(nodo* p,
          const K& k1, const K& k2, list<pair<K, V> >& L) {
     if (p == NULL) return;
     if (k1 <= p -> clave_)
          range_search(p -> izq_, k1, k2, L);
     if (k1 <= p -> clave_ && p -> clave_ <= k2)
          L.push_back(make_pair(p -> clave_, p -> valor_));
     if (p -> clave_ <= k2)
          range_search(p -> der_, k1, k2, L);
}
```

Colas de prioridad

Una cola de prioridad (cat: cua de prioritat; ing: priority queue) es una colección de elementos donde cada elemento tiene asociado un valor susceptible de ordenación denominado prioridad.

- Inserción de un elementos

Colas de prioridad

Una cola de prioridad (cat: cua de prioritat; ing: priority queue) es una colección de elementos donde cada elemento tiene asociado un valor susceptible de ordenación denominado prioridad.

- Inserción de un elementos
- Consulta del elemento de mínima (o máxima) prioridad

Colas de prioridad

Una cola de prioridad (cat: cua de prioritat; ing: priority queue) es una colección de elementos donde cada elemento tiene asociado un valor susceptible de ordenación denominado prioridad.

- Inserción de un elementos
- Consulta del elemento de mínima (o máxima) prioridad
- Eliminación del elemento de mínima (o máxima) prioridad

Algunas aplicaciones de las colas de prioridad:

- Algoritmos de Kruskal y Prim para el cálculo del árbol de expansión mínimo de un grafo etiquetado.
- Algoritmo de Dijkstra para el cálculo de caminos mínimos en un grafo etiquetado.
- Construcción de códigos de Huffman (códigos binarios de longitud media mínima).

Otra tareas para las que obviamente podemos usar una cola de prioridad son la ordenación y la selección del k-ésimo de un conjunto.

Ordenación mediante una cola de prioridad

```
...
struct Info { Clave clave; ... };
vector<Info> A;
Pqueue<Info, Clave> P;

for (int i = 0; i < A.size(); ++i)
     P.insert(A[i], A[i].clave);
int i = 0;
while (!P.empty()) {
       A[i++] = P.min();
       P.delete_min();
}
```

Implementación:

- Listas enlazadas ordenadas por prioridad
  Ventajas: Consulta y eliminación del mínimo triviales, con coste $\Theta(1)$
  Inconvenientes: Inserciones con coste lineal, tanto en caso peor como en promedio

Implementación:

- Listas enlazadas ordenadas por prioridad
  Ventajas: Consulta y eliminación del mínimo triviales, con coste $\Theta(1)$
  Inconvenientes: Inserciones con coste lineal, tanto en caso peor como en promedio
- Árboles de Búsqueda
  Ventaja: Inserciones y eliminación del mínimo con coste $\Theta(\log n)$ en caso peor (AVLs) o en caso promedio (BST)

- Si el conjunto de posibles prioridades es reducido entonces será conveniente emplear una tabla de listas, correspondiendo cada lista a una prioridad o intervalo reducido de prioridades.

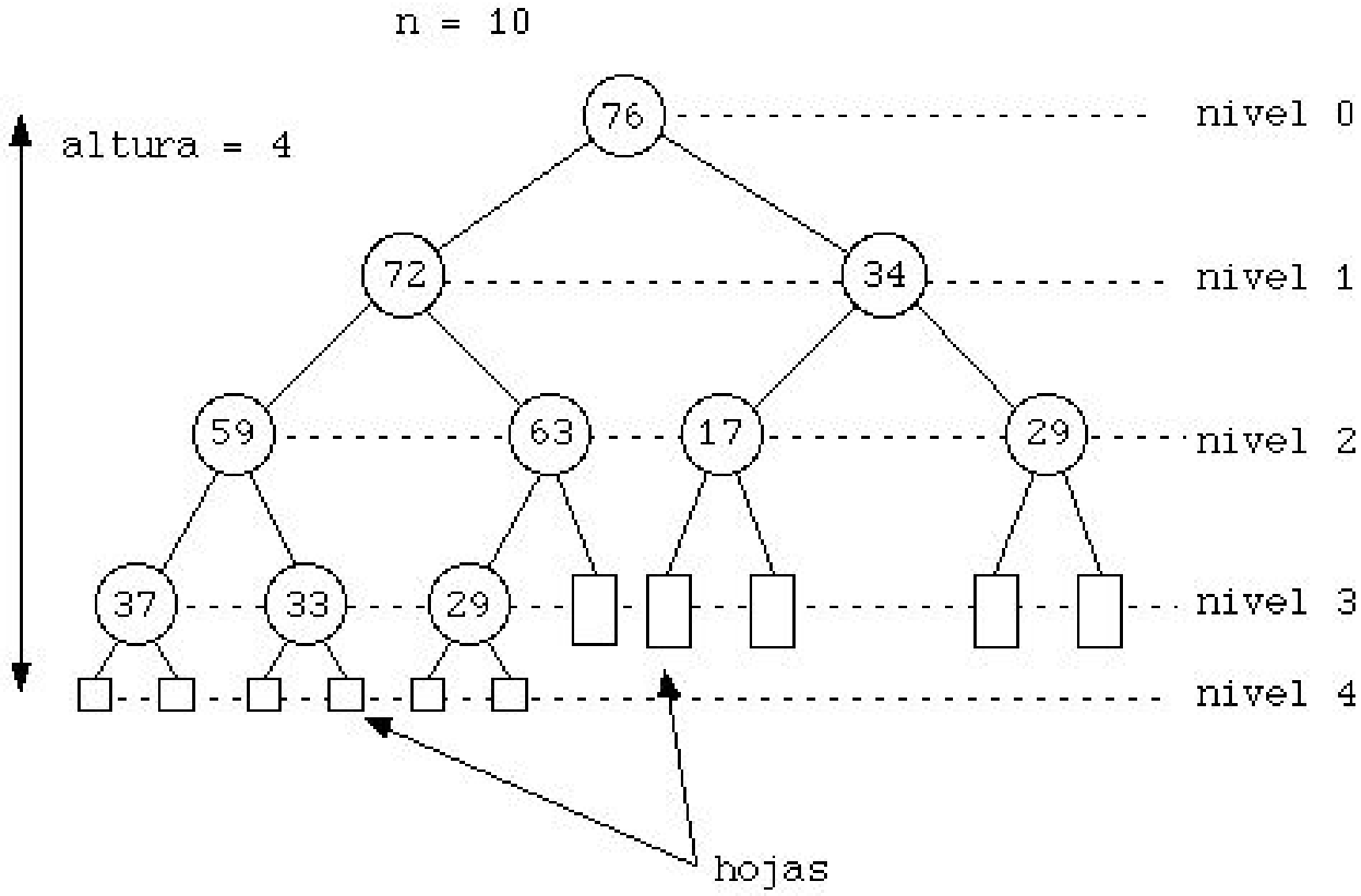
- Si el conjunto de posibles prioridades es reducido entonces será conveniente emplear una tabla de listas, correspondiendo cada lista a una prioridad o intervalo reducido de prioridades.
- En lo que resta estudiaremos una técnica específica para la implementación de colas de prioridad basada en los denominados montículos.

Definición

Un montículo (ing: heap) es un árbol binario tal que

1. todos las hojas (subárboles son vacíos) se sitúan en los dos últimos niveles del árbol.
2. en el penúltimo nivel existe a lo sumo un nodo interno con un sólo hijo, que será su hijo izquierdo, y todos los nodos a su derecha en el mismo nivel son nodos sin hijos.
3. el elemento (su prioridad) almacenado en un nodo cualquiera es mayor (menor) o igual que los elementos almacenados en sus hijos izquierdo y derecho.

Se dice que un montículo es un árbol binario quasi-completo debido a las propiedades 1-2. La propiedad 3 se denomina orden de montículo, y se habla de max-heaps o min-heaps según que los elementos sean $\geq$ ó $\leq$ ques sus hijos. En lo sucesivo sólo consideraremos max-heaps.

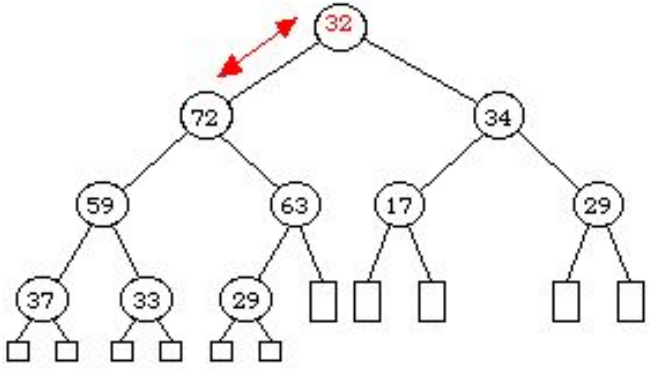

n = 10
altura = 4
76
72
34
59
63
17
29
37
33
29
nivel 0
nivel 1
nivel 2
nivel 3
nivel 4
hojas

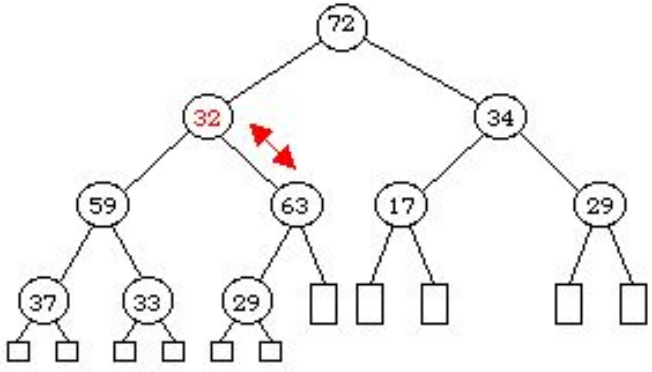
De las propiedades 1-3 se desprenden dos consecuencias importantes:

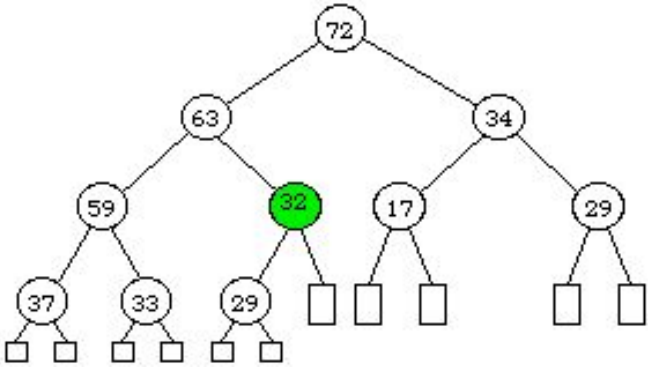
1. El elemento máximo se encuentra en la raíz.
2. Un heap de n elementos tiene altura $h = \lceil \log_2(n+1) \rceil$.

La consulta del máximo es sencilla y eficiente pues basta examinar la raíz.

¿Cómo eliminar el máximo? Un procedimiento que se emplea a menudo consiste en ubicar al último elemento del montículo (el del último nivel más a la derecha) en la raíz, sustituyendo al máximo; ello garantiza que se preservan las propiedades 1-2. Pero como la propiedad 3 deja eventualmente de satisfacerse, debe reestablecerse el invariante para lo cual se emplea un procedimiento privado denominado `hundir`.
Éste consiste en intercambiar un nodo con el mayor de sus dos hijos si el nodo es menor que alguno de ellos, y repetir este paso hasta que el invariante se haya reestablecido.


32
72
34
59
63
17
29
37
33
29


72
32
34
59
63
17
29
37
33
29


72
63
34
59
32
17
29
37
33
29

¿Cómo añadir un nuevo elemento? Una posibilidad consiste en colocar el nuevo elemento como último elemento del montículo, justo a la derecha del último o como primero de un nuevo nivel. Para ello hay que localizar al padre de la primera hoja y sustituirla por un nuevo nodo con el elemento a insertar. A continuación hay que reestablecer el orden de montículo empleando para ello un procedimiento `flotar`, que trabaja de manera similar pero a la inversa de `hundir`: el nodo en curso se compara con su nodo padre y se realiza el intercambio si éste es mayor que el padre, iterando este paso mientras sea necesario.

Puesto que la altura del heap es $\Theta(\log n)$ el coste de inserciones y eliminaciones es $\mathcal{O}(\log n)$. Se puede implementar un heap mediante memoria dinámica. La representación elegida debe incluir apuntadores al hijo izquierdo y derecho y también al padre, y resolver de manera eficaz la localización del último elemento y del padre de la primera hoja.

Una alternativa atractiva es la implementación de heaps mediante un vector. No se desperdicia demasiado espacio ya que el heap es quasi-completo. Las reglas para representar los elementos del heap en un vector son simples:

1. $A[1]$ contiene la raíz.
2. Si $2i \leq n$ entonces $A[2i]$ contiene al hijo izquierdo del elemento en $A[i]$ y si $2i + 1 \leq n$ entonces $A[2i + 1]$ contiene al hijo derecho de $A[i]$.
3. Si i **div** $2 \geq 1$ entonces $A[i$ **div** $2]$ contiene al padre de $A[i]$.

Implementación mediante heaps

```
template <typename Elem, typename Prio>
class Pqueue {
public:
       Pqueue() {
          _elems.push_back(Elem());
          _prios.push_back(Prios());
          _nelems = 0;
       }
       void insert(const Elem& x, const Prio& p);
       Elem max() const;
       void delete_max();
       bool empty() const { return _nelems == 0; }
private:
       vector<Elem> _elems;
       vector<Prio> _prios;
       int _nelems;
}
```

Implementación mediante heaps

```
template <typename Elem, typename Prio>
void Pqueue<Elem,Prio>::insert(const Elem& x,
                               const Prio& p) {
      ++_nelems;
      if (_nelems < _elems.size()) {
          _elems[_nelems] = x;
          _prios[_nelems] = p;
      } else {
          _elems.push_back(x);
          _prios.push_back(p);
      }
      flotar(_elems, _prios, _nelems);
}
```

Implementación mediante heaps (cont.)

```
template <typename Elem, typename Prio>
Elem Pqueue<Elem,Prio>::max() const {
     if (_nelems == 0) ...
     return _elems[1];
}

template <typename Elem, typename Prio>
void Pqueue<Elem,Prio>::delete_max() {
     if (_nelems == 0) ...
     _elems[1] = _elems[_nelems];
     _prios[1] = _prios[_nelems];
     --_nelems;
     hundir(_elems, _prios, 1, _nelems);
}
```

Hundir (versión recursiva)

```
template <typename Elem, typename Prio>
void hundir(vector<Elem>& elems, vector<Prio>& prios,
            int j, int n) {
    int hijo = 2 * j;
    if (hijo > n) return;
    if (hijo < n && _prios[hijo] < _prios[hijo + 1])
       ++hijo;
    if (_prios[j] < _prios[hijo]) {
       swap(_prios[j], _prios[hijo]);
       swap(_elems[j], _elems[hijo]);
    }
    hundir(_elems, _prios, hijo, n);
}
```

Flotar (versión iterativa)

```
template <typename Elem, typename Prio>
void flotar(vector<Elem>& elems, vector<Prio>& prios,
            int j) {
    while (j > 1) {
        int padre = j / 2;
        if (_prios[padre] < _prios[j]) {
            swap(_prios[padre], _prios[j]);
            swap(_elems[padre], _elems[j]);
        }
        else break;
        j = padre;
    }
}
```

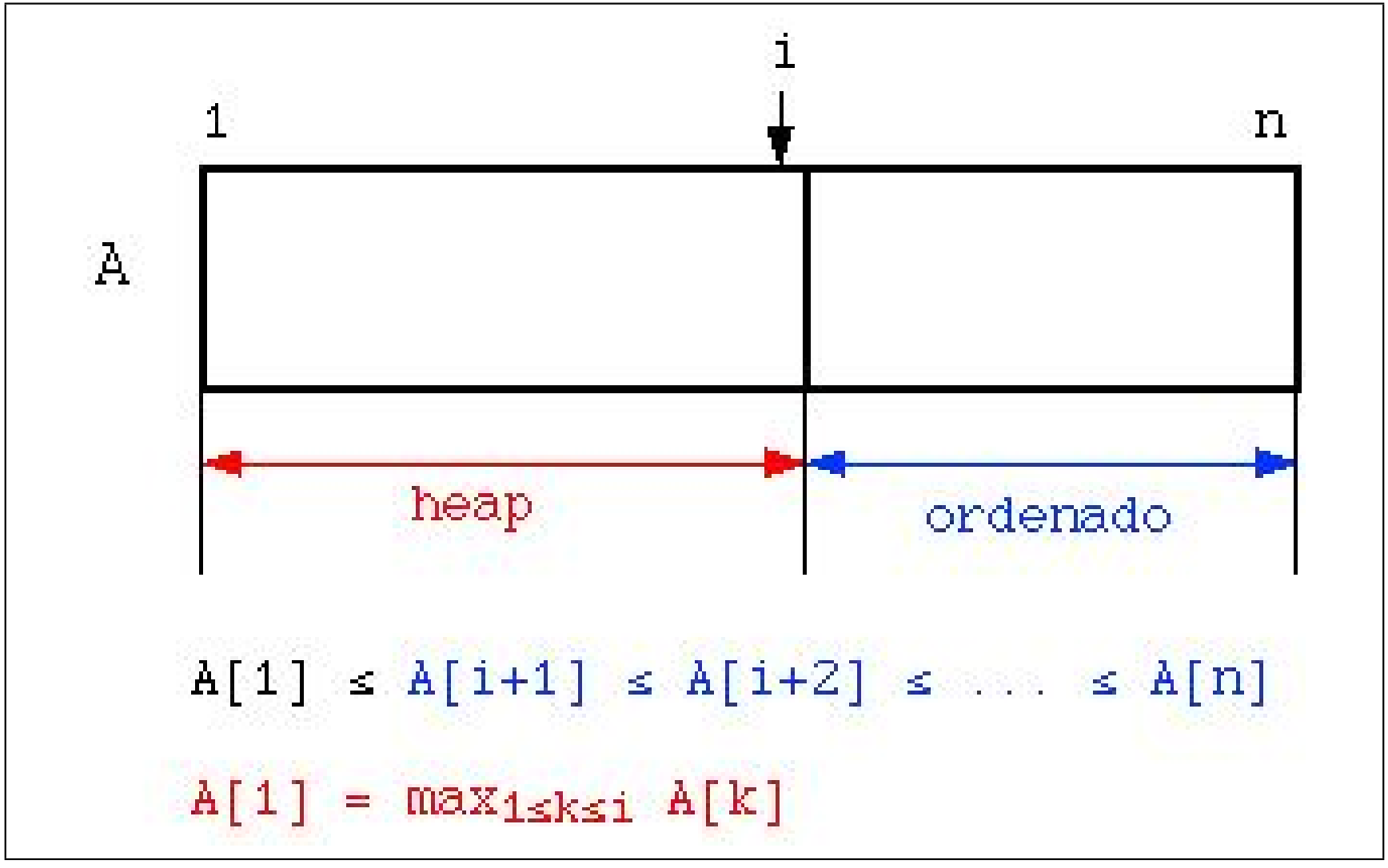
Heapsort

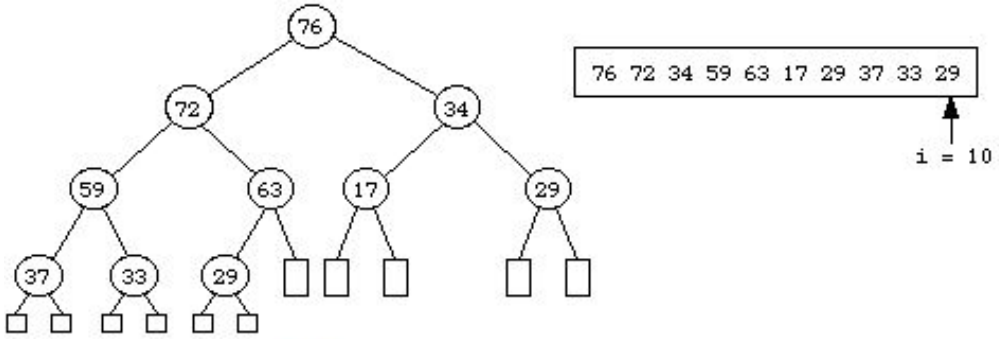
Heapsort (Williams, 1964) ordena un vector de n elementos construyendo un heap con los n elementos y extrayéndolos, uno a uno del heap a continuación. El propio vector que almacena a los n elementos se emplea para construir el heap, de modo que heapsort actúa in-situ y sólo requiere un espacio auxiliar de memoria constante. El coste de este algoritmo es $\Theta(n \log n)$ (incluso en caso mejor) si todos los elementos son diferentes.

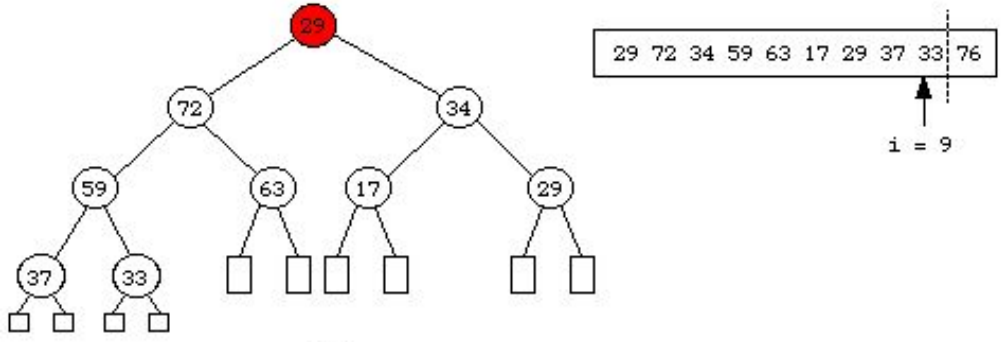
En la práctica su coste es superior al de quicksort, ya que el factor constante multiplicativo del término $n \log n$ es mayor.

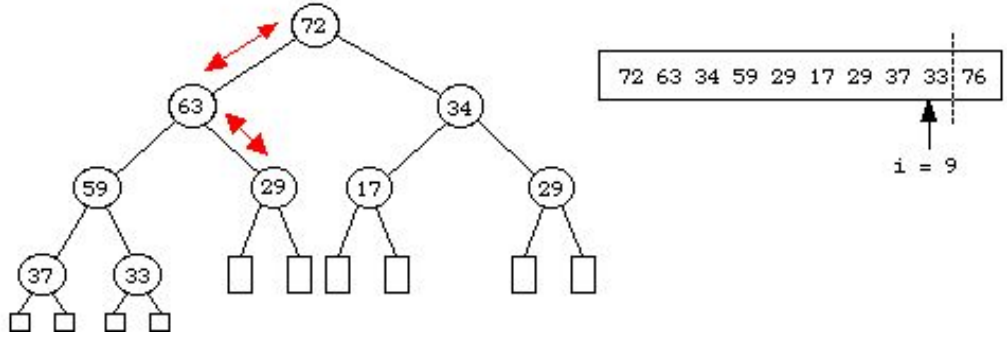
Heapsort

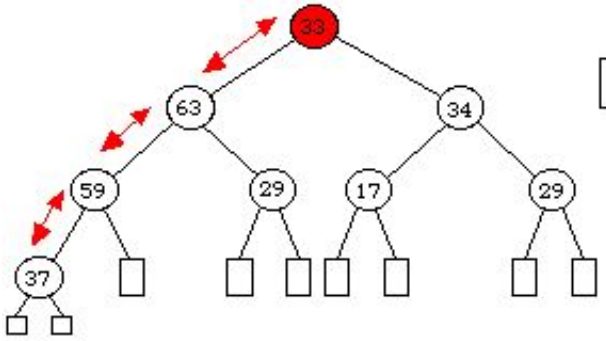
```
template <typename Elem>
void heapsort(vector<Elem>& A, int n) {
     crea_heap(A, n);
     for (int i = n; i > 0; --i) {
         swap(A[1], A[i]);
         hundir(A, 1, i - 1);
     }
}
```

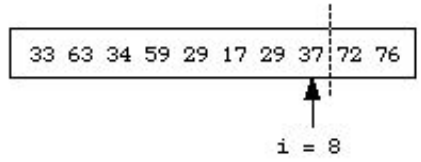

i
1
n
A
heap
ordenado
A[1] ≤ A[i+1] ≤ A[i+2] ≤ . . . ≤ A[n]
A[1] = max1≤k≤i A[k]

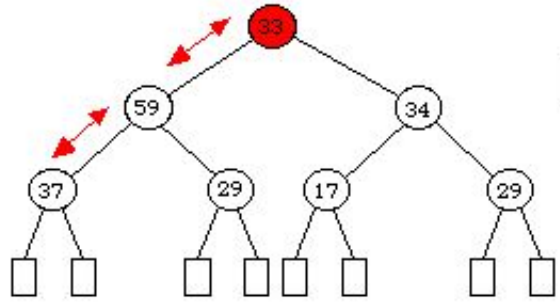

76
72
34
59
63
17
29
37
33
29
29
76 72 34 59 63 17 29 37 33 29
i = 10

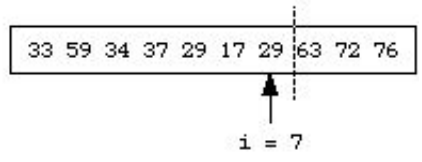

29
72
34
59
63
17
29
37
33
29 72 34 59 63 17 29 37 33 76
i = 9

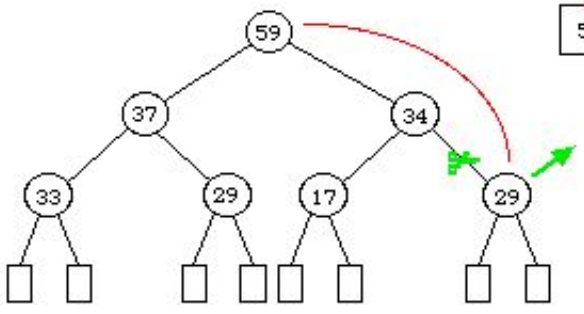

72
63
34
59
29
17
29
37
33
72 63 34 59 29 17 29 37 33 76
i = 9

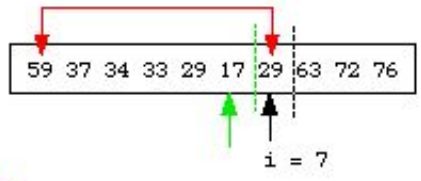

33
63
34
59
29
17
29
37


33 63 34 59 29 17 29 37 72 76
i = 8


33
59
34
37
29
17
29

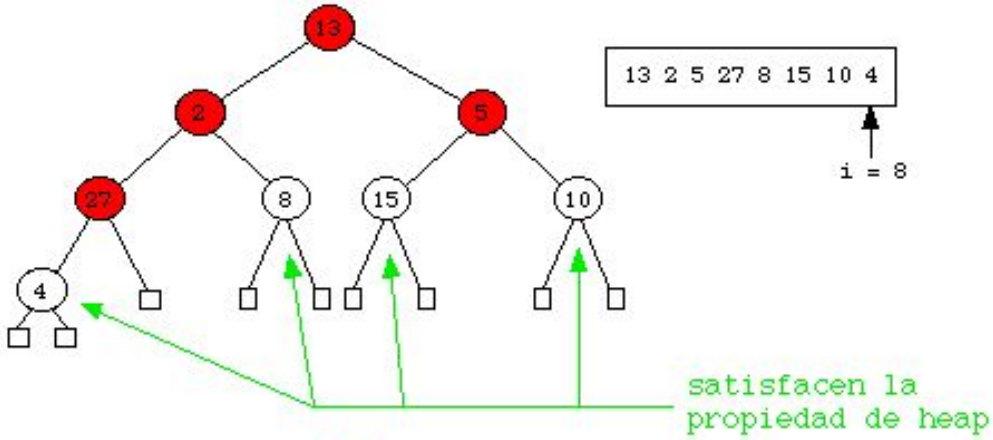

33 59 34 37 29 17 29 63 72 76
i = 7

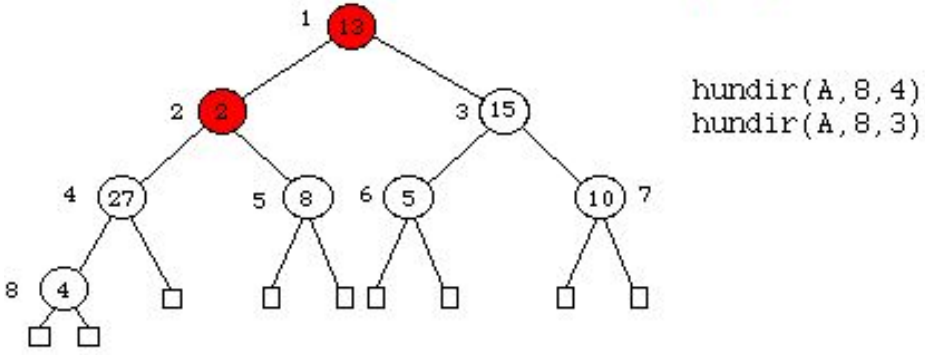

59
37
34
33
29
17
29


59 37 34 33 29 17 29 63 72 76
i = 7

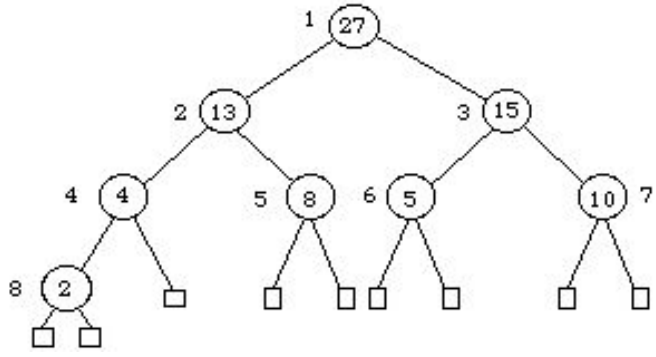
Crea_heap

```
void crea_heap(vector<Elem>& A, int n) {
     for (int i = n / 2; i > 0; --i)
        hundir(A, i, n);
}
```





hundir(A,8,4)
hundir(A,8,3)



hundir(A,8,2)
hundir(A,8,1)

Sea $H(n)$ el coste en caso peor de `heapsort` y $B(n)$ el coste de crear el heap inicial. El coste en caso peor de $\texttt{hundir}(A, 1, i-1)$ es $\mathcal{O}(\log i)$ y por lo tanto

$$\begin{aligned} H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\ &= B(n) + \mathcal{O}\left(\sum_{1 \le i \le n} \log_2 i\right) \\ &= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n) \end{aligned}$$

Un análisis "grueso" de $B(n)$ indica que $B(n) = \mathcal{O}(n \log n)$ ya que hay $\Theta(n)$ llamadas a `hundir`, cada una de las cuales tiene coste $\mathcal{O}(\log n)$. Podemos concluir por tanto que $H(n) = \mathcal{O}(n \log n)$. No es difícil construir una entrada de tamaño n tal que $H(n) = \Omega(n \log n)$ y por tanto $H(n) = \Theta(n \log n)$ en caso peor. La demostración de que el coste de `heapsort` en caso mejor[1] es también $\Theta(n \log n)$ es bastante más complicada.

[1] Siendo todos los elementos distintos.

Por otra parte, la cota dada para $B(n)$ podemos refinarla ya que

$$\begin{aligned} B(n) &= \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\ &= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\ &= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n). \end{aligned}$$

Puesto que $B(n) = \Omega(n)$, podemos afirmar que $B(n) = \Theta(n)$.

Otra forma de demostrar que $B(n)$ es lineal consiste en razonar del siguiente modo: Sea $h = \lceil \log_2(n+1) \rceil$ la altura del heap. En el nivel $h-1-k$ hay como mucho

$$2^{h-1-k} < \frac{n+1}{2^k}$$

nodos y cada uno de ellos habrá de hundirse en caso peor hasta el nivel $h-1$; eso tiene coste $\mathcal{O}(k)$.

Por lo tanto,

$$\begin{aligned} B(n) &= \sum_{0 \leq k \leq h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\ &= \mathcal{O}\left(n \sum_{0 \leq k \leq h-1} \frac{k}{2^k} \right) \\ &= \mathcal{O}\left(n \sum_{k \geq 0} \frac{k}{2^k} \right) = \mathcal{O}(n), \end{aligned}$$

ya que

$$\sum_{k \geq 0} \frac{k}{2^k} = 2.$$

En general, si $r < 1$,

$$\sum_{k \geq 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$

Aunque globalmente $H(n) = \Theta(n \log n)$, es interesante el análisis detallado de $B(n)$. Por ejemplo, utilizando un min-heap podemos hallar los k menores elementos de un vector (y en particular el k-ésimo) con coste:

$$\begin{aligned} S(n,k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

y si $k = \mathcal{O}(n/\log n)$ entonces $S(n,k) = \mathcal{O}(n)$.

Algoritmo de Dijkstra

Dado un grafo dirigido $G = \langle V, E \rangle$ etiquetado, el algoritmo de Dijkstra (1959) nos permite hallar los caminos mínimos desde un vértice $s \in V$ dado a todos los restantes vértices del grafo.

Si el grafo G contuviera ciclos de peso negativo la noción de camino mínimo no estaría bien definida para todo par de vértices, pero el algoritmo de Dijkstra puede fallar en presencia de arcos con peso negativo, incluso si no hay ciclos de peso negativo. Supondremos por lo tanto que todos los pesos $\omega : E \to \mathbb{R}^+$ son positivos.

Sea $\mathcal{P}(u, v)$ el conjunto de caminos entre dos vértices u y v de G. Dado un camino $\pi = [u, \ldots, v] \in \mathcal{P}(u, v)$ su peso es la suma de los pesos de los n arcos que lo forman:

$$\omega(\pi) = \omega(u, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{n-1}, v).$$

Sea $\Delta(u,v) = \min\{\omega(\pi) \mid \pi \in \mathcal{P}(u,v)\}$ y $\pi^*(u,v)$ un camino de $\mathcal{P}(u,v)$ cuyo peso es mínimo. Si $\mathcal{P}(u,v) = \emptyset$ entonces tomamos $\Delta(u,v) = +\infty$, por convenio. Consideraremos en primer lugar la versión del algoritmo de Dijkstra que calcula los pesos de los caminos mínimos desde un vértice a todos los restantes, y más tarde la versión que calcula adicionalmente los caminos propiamente dichos.

- $G = \langle V, E \rangle$ es un grafo dirigido con pesos positivos
- $s \in V$

`dijkstra`(G, s, D)

- Para todo $u \in V$, $D[u] = \Delta(s,u)$

- $G = \langle V, E \rangle$ es un grafo dirigido con pesos positivos
- $s \in V$

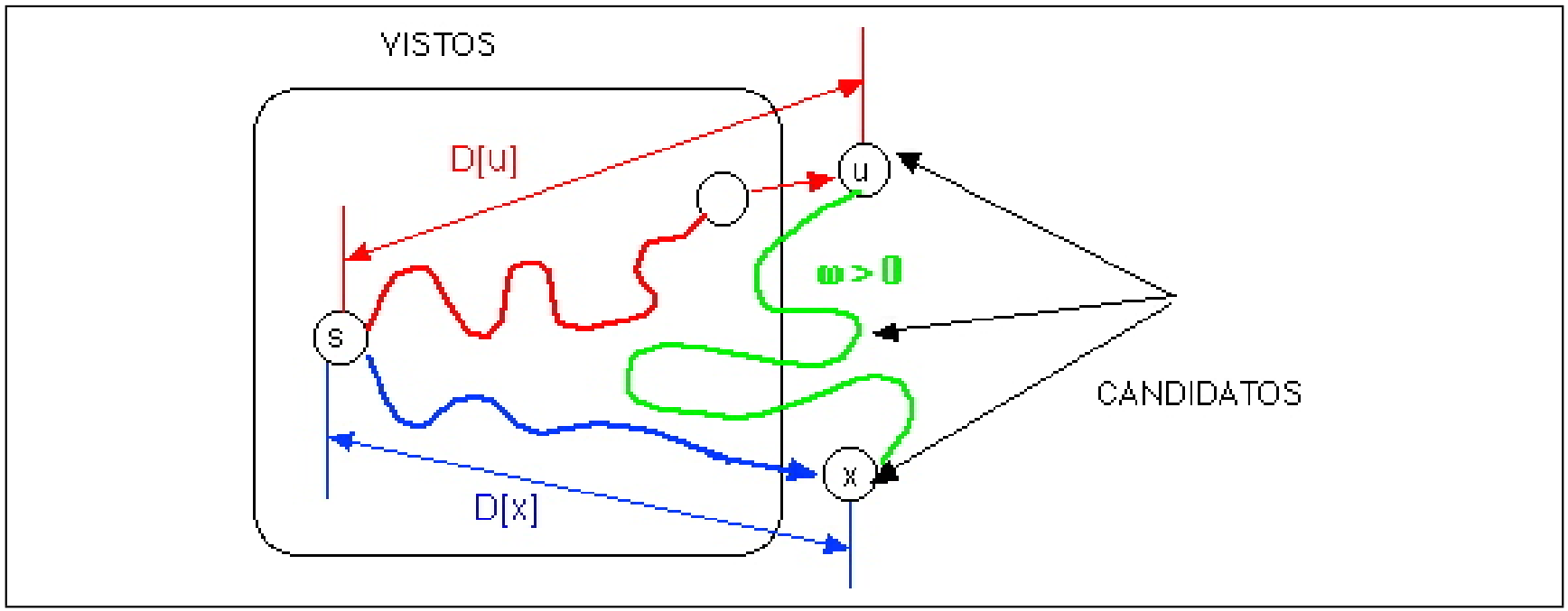
`dijkstra`(G, s, D, cam)

- Para todo $u \in V$, $D[u] = \Delta(s,u)$
- Para todo $u \in V$, $D[u] < +\infty \implies cam[u] = \pi^*(s,u)$

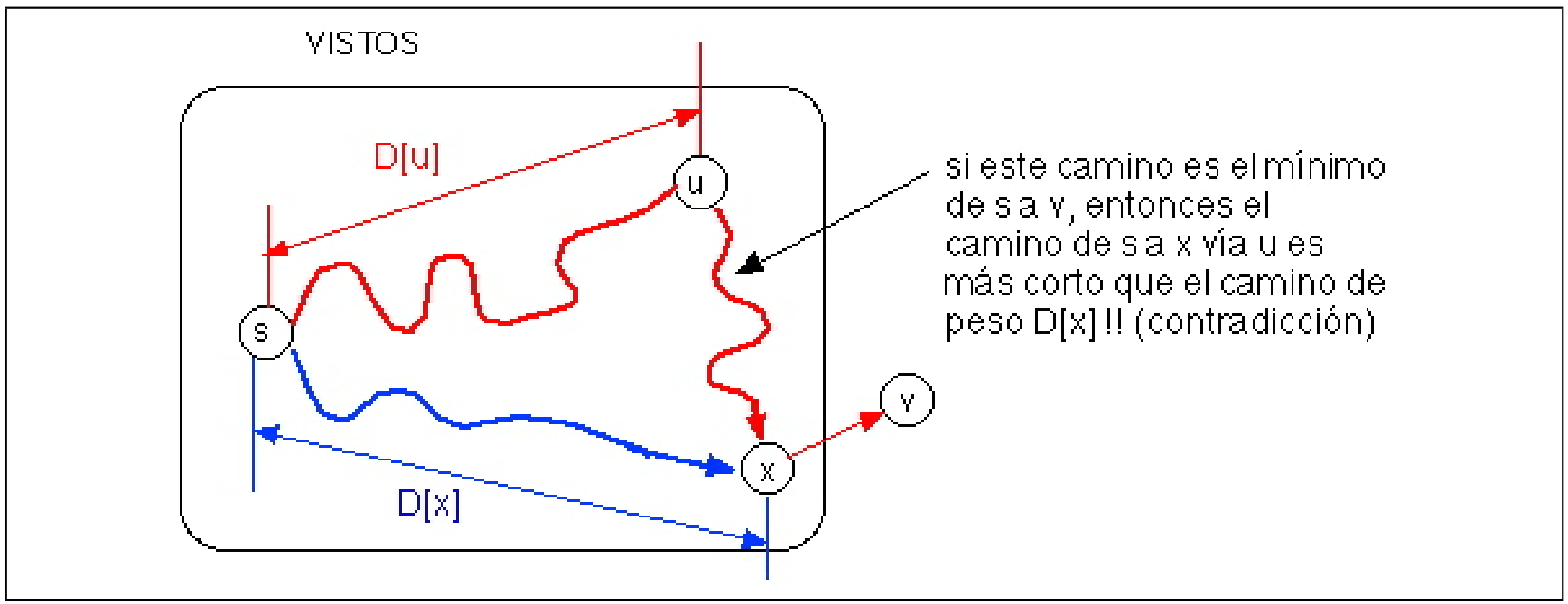
El algoritmo de Dijkstra actúa en una serie de etapas. En todo momento el conjunto de vértices V se divide en dos partes: los vértices vistos y los vértices no vistos o candidatos. Al inicio de cada etapa, si u es un vértice visto entonces $D[u] = \Delta(s, u)$; si u es un candidato entonces $D[u]$ es el peso del camino mínimo entre s y u que pasa exclusivamente por vértices intermedios vistos. Este es el invariante del algoritmo de Dijkstra.

En cada etapa un vértice pasa de ser candidato a ser visto. Y cuando todos los vértices son vistos entonces tenemos completamente resuelto el problema.

¿Qué vértice candidato debe seleccionarse en cada etapa para pasar a ser visto? Intuitivamente, aquél cuya D sea mínima de entre todos los candidatos. Sea u dicho vértice. Entonces $D[u]$ no sólo es el peso del camino mínimo entre s y u que sólo pasa por vistos (según el invariante), sino el peso del camino mínimo. En efecto, si el camino mínimo pasase por algún otro vértice no visto x, tendríamos que el peso de dicho camino es $D[x] + \Delta(x, u) < D[u]$, pero como $\Delta(x, u) \geq 0$ y $D[u]$ es mínimo llegamos a una contradicción.



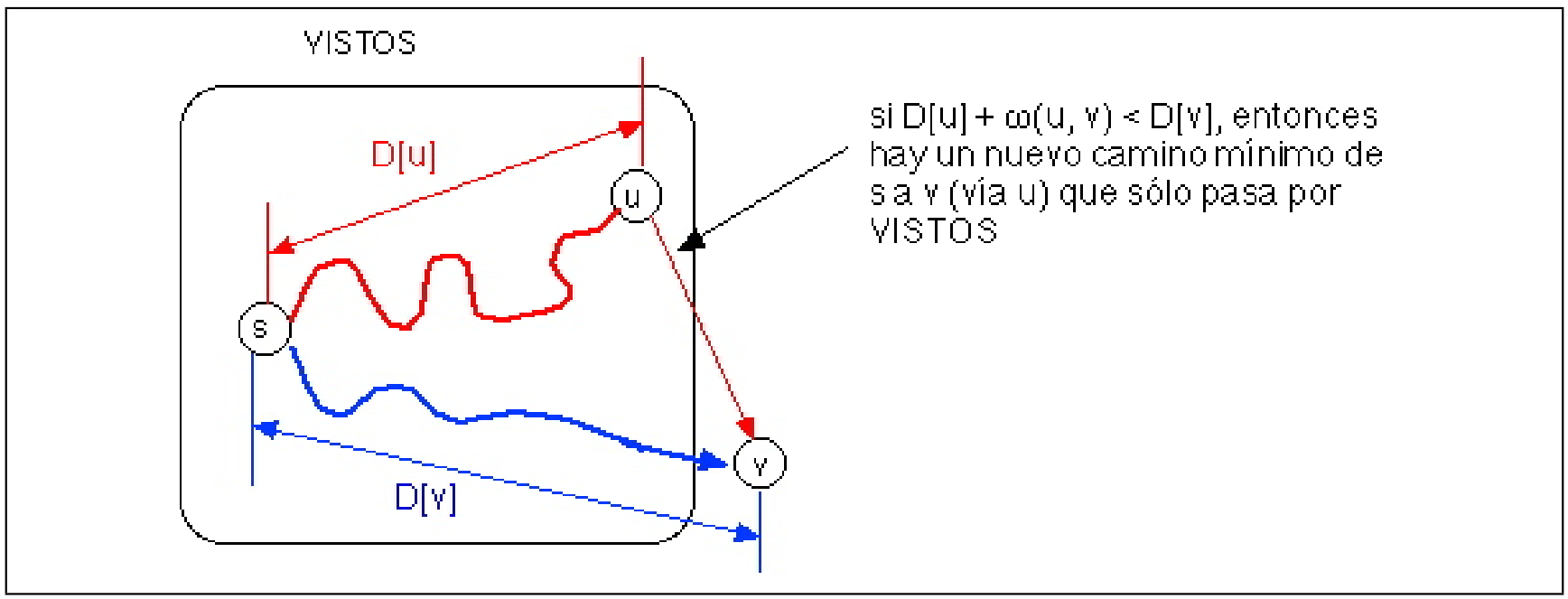
Ahora debemos pensar cómo mantener el resto del invariante. Para los vértices vistos D ya tiene el valor adecuado (incluído u, el vértice que pasa de candidatos a vistos, como acabamos de ver). Pero si v es un candidato, $D[v]$ tal vez haya de cambiar ya que tenemos un vértice adicional visto, el vértice u. Un sencillo razonamiento demuestra que si el camino mínimo entre s y v que pasa por vértices vistos incluye a u entonces u es el inmediato antecesor de v en dicho camino.



De ahí se sigue que el valor de D sólo puede cambiar para los vértices v sucesores de u. Esto ocurrirá si y sólo si

$$D[v] > D[u] + \omega(u, v).$$

Si v fuera sucesor de u pero ya estuviera visto la condición anterior no puede ser cierta.



```
procedure dijkstra(in g : grafo⟨vertice, real⟩;
                   in s : vertice;
                   out D : dict⟨vertice, real⟩)
    var cand : conjunto⟨vertice⟩
        u, v : vertice; d : real
    end

    for v ∈ V(g) do D[v] := +∞ end
    D[s] := 0
    cand := V(g)
    while cand ≠ ∅ do
        u := el vértice de cand con D mínima
        cand := cand \ {u}
        for v ∈ sucesores(g, u) do
            d := D[u] + etiqueta(g, u, v)
            if d < D[v] then D[v] := d fi
        end
    end
end
```

D						CANDIDATOS
1	2	3	4	5	6	
0	∞	∞	∞	∞	∞	$\{1,2,3,4,5,6\}$
0	3	6	∞	∞	∞	$\{2,3,4,5,6\}$
0	3	6	7	4	∞	$\{3,4,5,6\}$
0	3	6	7	4	9	$\{3,4,6\}$
0	3	6	7	4	8	$\{4,6\}$
0	3	6	7	4	8	$\{6\}$
						$\emptyset$

Para conseguir que el algoritmo compute los caminos mínimos propiamente dichos, empezamos observando que si el camino mínimo entre s y u pasa por x entonces la parte de ese camino que nos lleva de s a x ha de ser necesariamente el camino mínimo entre s y x. Por lo tanto, bastará con computar un árbol de caminos mínimos que implícitamente se representa mediante una tabla cam tal que:

$$cam[v] = \begin{cases} s & \text{si } v = s, \\ u & \text{si } (u,v) \text{ es el último arco de } \pi^*(s,v), \\ \perp & \text{si } \Delta(s,v) = +\infty, \end{cases}$$

donde $cam[v] = \perp$ indica que $cam[v]$ no está definido.

Claramente, los únicos cambios que serán necesarios son:

1. inicializar $cam[s] := s$.
2. incluir la actualización de cam en el bucle interno:

$$
\begin{array}{l}
\ldots \\
\textbf{for } v \in \texttt{sucesores}(g, u) \textbf{ do} \\
\quad d := D[u] + \texttt{etiqueta}(g, u, v) \\
\quad \textbf{if } d < D[v] \textbf{ then} \\
\quad\quad D[v] := d \\
\quad\quad cam[v] := u \\
\quad \textbf{fi} \\
\textbf{end} \\
\ldots
\end{array}
$$

Si queremos el camino completo de s a v podemos deshacer el recorrido a la inversa:

$$cam[u], cam[cam[u]], \ldots, cam[\cdots[cam[u]]\cdots]$$

hasta que lleguemos a s.

Sea n el número de vértices de g y m el número de arcos. Si el grafo g se implementa mediante matriz de adyacencias, el coste del algoritmo de Dijkstra es $\Omega(n^2)$ ya que se hacen n iteraciones del bucle principal y dentro de cada una de ellas se incurriría en un coste como mínimo $\Omega(n)$ para recorrer los sucesores del vértice seleccionado. Descartaremos esta posibilidad, y supondremos que el grafo se implementa mediante listas de adyacencia que nos darán mejores resultados tanto en tiempo como en espacio.

Supongamos, para simplificar, que $V(g) = \{1, \ldots, n\}$ y que implementamos el conjunto $cand$ y el diccionario D mediante sencillas tablas indexadas de 1 a n ($cand[i] = \texttt{cierto}$ si y sólo si el vértice i es un candidato). Entonces el coste del algoritmo de Dijkstra es $\Theta(n^2 + m) = \Theta(n^2)$ puesto que se hacen n iteraciones del bucle principal, buscar el mínimo de la tabla D en cada iteración tiene coste $\Theta(n)$ y actualizar la tabla D tiene coste proporcional al número de sucesores del vértice seleccionado (combinando la obtención de los vértices sucesores y la etiqueta de los arcos correspondientes).

Si $V(g)$ es un conjunto arbitrario, podemos conseguir el mismo rendimiento utilizando tablas de hash para implementar $cand$ y D. Y el problema sigue siendo la ineficiencia en la selección del vértice con D mínima. Para mejorar la eficiencia podemos convertir $cand$ en una cola de prioridad cuyos elementos son vértices y donde la prioridad de cada vértice es su correspondiente valor de D:

```
var cand : cola_prio⟨vertice, real⟩
    u, v : vertice; d : real
end
cand := vacia()
D[s] := 0
for v ∈ V(g) do D[v] := +∞ end
for v ∈ V(g) do inserta(cand, v, D[v]) end
while ¬ es_vacia(cand) do
    u := min(cand); elim_min(cand)
    for v ∈ sucesores(g, u) do
        ...
    end
end
```

El problema es que dentro del bucle que recorre los sucesores de u se puede modificar el valor de D (la prioridad) de vértices candidatos. Por ello deberemos dotar al TAD `COLA_PRIO` de una operación adicional que, dado un elemento, nos permita decrementar su prioridad (los valores de D se modifican siempre a la baja).

```
for v ∈ sucesores(g, u) do
    d := D[u] + etiqueta(g, u, v)
    if d < D[v] then
        D[v] := d; decr_prio(cand, v, d)
    fi
end
```

Si los costes de las operaciones sobre la cola de prioridad son $\mathcal{O}(\log n)$ entonces el coste del Bucle principal es

$$\begin{aligned} D(n) &= \sum_{v \in V(g)} \mathcal{O}(\log n) \cdot (1 + \#\text{ sucesores de } v) \\ &= \Theta(n \log n) + \mathcal{O}(\log n) \sum_{v \in V(g)} \#\text{ sucesores de } v \\ &= \Theta(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}((n+m) \log n) \end{aligned}$$

Por otra parte, el coste de las inicializaciones previas es $\mathcal{O}(n \log n)$.
Por lo tanto el coste que nos queda es $\mathcal{O}((n+m) \log n)$.
Puede demostrarse que, de hecho, el coste en caso peor del algoritmo de Dijkstra es $\Theta((m+n) \log n)$. Pero en muchos casos el coste es menor, porque no hay que usar `decr_prio` para todos los sucesores del vértice seleccionado y/o porque el coste de las operaciones sobre la cola de prioridad es frecuentemente menor que $\Theta(\log n)$.

Al crear $cand$ sabemos cuántos vértices tiene el grafo y podemos crear dinámicamente una estructura de datos para ese tamaño. Adicionalmente podemos evitar la redundancia de la tabla D, ya que para cada elemento de $cand$ tenemos su prioridad, que es su valor de D.

Necesitamos por lo tanto un TAD con la funcionalidad combinada típica de las colas de prioridad y de los diccionarios:

```
TAD COLA_PRIO_DIJKSTRA⟨ELEM,PRIO⟩
genero priodijks
ops
   crea: nat → priodijks
   inserta: priodijks elem prio → priodijks
   min: priodijks → elem
   elim_min: priodijks → priodijks
   prio: priodijks elem → prio
   esta: priodijks elem → bool
   decr_prio: priodijks elem prio → priodijks
   es_vacia: priodijks → bool
fops
```

```
var cand : priodijks⟨vertice, real⟩
    u, v : vertice; d, du : real
end
cand := crea(numero_vertices(g))
for v ∈ V(g) do
   inserta(cand, v, +∞) end
decr_prio(cand, s, 0)
while cand ≠ ∅ do
    u := min(cand); du := prio(cand, u)
    elim_min(cand)
    for v ∈ sucesores(g, u) do
        if esta(cand, v) then
            d := du + etiqueta(g, u, v)
            if d < prio(cand, v) then
                decr_prio(cand, v, d)
            fi
        fi
    end
end
```

Pueden conseguirse costes logarítmicos o inferiores en todas las operaciones sobre la cola de prioridad utilizando un heap implementado en vector.
Además necesitaremos una tabla de hash que nos permita "traducir" vértices a índices. Y una tabla de índices y otra de posiciones en el heap.

```
type priodijks =record
                    prio : array [...] of real
                    e : array [...] of elem
                    index : array [...] of
                    pos : array [...] of entero
                    nelems : entero
                    map : tabla_hash⟨elem, entero⟩
                end
end
```

Las tablas $prio$ y e, junto al contador $nelems$ representan al heap. Si $e[i] = e_j$ entonces $index[i] = j$ y $pos[j] = i$, es decir, pos nos da la posición en el heap del elemento e_j e $index$ su índice, siendo $\texttt{valor}(map, e_j) = j$.

La operación `vacia` reclama a la memoria dinámica las tablas $prio$, e, $index$ y pos con n componentes y crea la tabla de hash. Cada vez que se inserta un nuevo elemento se le asigna el índice $nelems + 1$, se inserta en map, se coloca en el heap en la posición $nelems + 1$, y se le hace flotar. La operación `flotar` se encarga de mantener actualizadas las tablas pos e $index$. Para `elim_min`, se intercambia el último nodo del heap con la raíz

$$prio[1] \leftrightarrow prio[nelems]; e[1] \leftrightarrow e[nelems]$$
$$pos[index[1]] \leftrightarrow pos[index[nelems]]$$
$$index[1] \leftrightarrow index[nelems]$$

A continuación se hunde la raíz, cuidando de mantener actualizadas las tablas pos e $index$.

La operación `decr_prio` requiere averiguar el índice j del elemento e_j cuya prioridad se va a decrementar usando la tabla de hash, usar la tabla $pos[j]$ para obtener la posición i del nodo que le corresponde en el heap, y una vez modificada la prioridad, flotar el nodo.
Es fácil comprobar que las operaciones del TAD, exceptuando `crea`, tienen coste $\Theta(1)$ (`es_vacia`, `min`, `prio`, `esta`) o $\mathcal{O}(\log n)$ (`inserta`, `elim_min`, `decr_prio`).