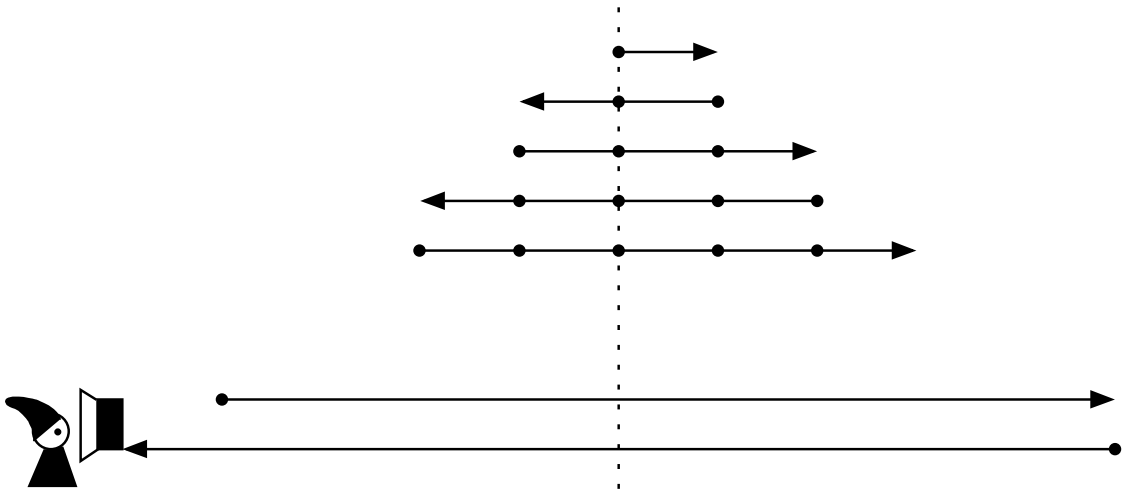
Divide y vencerás

Yolanda Ortega Mallén

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

La búsqueda de Gumersindo



$P(n) = \sum_{i=1}^{z(n)} i$ pasos

- $z(n) = 2n - 1$ si comienza en la dirección adecuada,
- $z(n) = 2n$ si comienza en la dirección contraria.

$P(n) \in \Theta(n^2)$.

$$\sum_{i=0}^{\lceil \log 2n \rceil + 1} 2^i = 2^{\lceil \log 2n \rceil + 2} - 1 \in \Theta(2^{\log 2n}) = \Theta(n).$$

Sumario

- Esquemas algorítmicos.
- Divide y vencerás: el ejemplo de la búsqueda binaria.
- Esquema general.
- Aplicaciones.

Bibliografía

- G. Brassard y P. Bratley. *Fundamentos de Algoritmia*, Prentice Hall, 1997.
  Capítulo 7
- R. Neapolitan y K. Naimipour. *Foundations of Algorithms*. Cuarta edición. Jones and Bartlett Publishers, 2010.
  Capítulo 2
- N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo López. *Estructuras de datos y métodos algorítmicos. 213 Ejercicios resueltos*. Segunda edición. Garceta Grupo Editorial, 2013.
  Capítulo 11

Esquemas algorítmicos

- Los esquemas algorítmicos son estrategias de resolución de problemas.
- Algoritmos genéricos que se aplican en la resolución de problemas que presentan unas características comunes.
- Esquemas algorítmicos más comunes:
  - Divide y vencerás,
  - Método voraz,
  - Programación dinámica,
  - Métodos de exploración exhaustiva: Vuelta atrás y Ramificación y poda.

Búsqueda binaria

Dado un vector ordenado $V[1..N]$ y un elemento x, buscar x en V y devolver, si x está en V, la posición donde se encuentra y, si no está, la posición donde debería estar, manteniendo el vector ordenado.

$\{\ ord(V,c,f)\ \wedge\ 1 \leq c \leq f+1 \leq N+1\ \}$

fun búsqueda-binaria$(V[1..N]$ **de** *ent*$, x$: *ent*$, c, f$: *nat*$)$ **dev** $\langle\, b$: *bool*$, p$: *nat* $\rangle$

  si $c > f$ **entonces** $\langle\, b, p \,\rangle := \langle$ falso$, c \,\rangle$

  si no

    $m := (c+f)$ div 2 ;

    casos

      $x < V[m] \ \rightarrow\ \langle\, b, p \,\rangle :=$ búsqueda-binaria$(V, x, c, m-1)$

    [] $x = V[m] \ \rightarrow\ \langle\, b, p \,\rangle := \langle$ cierto$, m \,\rangle$

    [] $x > V[m] \ \rightarrow\ \langle\, b, p \,\rangle :=$ búsqueda-binaria$(V, x, m+1, f)$

    fcasos

  fsi

ffun

$\{\ (b \rightarrow c \leq p \leq f\ \wedge\ V[p] = x)\ \wedge\ (\neg b \rightarrow c \leq p \leq f+1\ \wedge\ V[c..p) \prec x \prec V[p..f])\ \}$

Coste: $\Theta(\log n)$

Divide y vencerás

Divide et impera - Julio César (100 a.C.-44 a.C)

- Descomponer el problema a resolver en una colección de subproblemas más pequeños, resolver estos subproblemas y combinar los resultados para obtener la solución del problema original.
- Los subproblemas son del mismo tipo que el problema original, y se resuelven usando la misma técnica. Algoritmo recursivo.
- Los subproblemas han de tener un tamaño fracción del tamaño original (un medio, un tercio, etc.).
- Los subproblemas se generan exclusivamente a partir del problema original.
- Los casos base no son necesariamente los casos triviales.

Esquema de divide y vencerás

fun `divide-y-vencerás`$(x : \textit{problema})$ **dev** $y : \textit{solución}$
  si `pequeño`(x) **entonces**
    $y :=$ `método-directo`(x)
  si no
    $\{$ descomponer x en $k \geq 1$ problemas más pequeños $\}$
    $\langle x_1, x_2, \ldots, x_k \rangle :=$ `descomponer`(x)
    $\{$ resolver recursivamente los subproblemas $\}$
    para $j = 1$ **hasta** k **hacer**
      $y_j :=$ `divide-y-vencerás`(x_j)
    fpara
    $\{$ combinar los y_j para obtener una solución y para x $\}$
    $y :=$ `combinar`$(y_1, \ldots, y_k)$
  fsi
ffun

Para el esquema general

$$T(n) = \begin{cases} g(n) & n \leq n_0 \\ \left(\sum_{j=1}^{k} T(n_j) \right) + f(n) & n > n_0 \end{cases}$$

En particular, para muchos algoritmos

$$T(n) = \begin{cases} c & 0 \leq n < b \\ aT(n/b) + f(n) & n \geq b \end{cases}$$

Si $f(n) \in \Theta(n^k)$ entonces:

$a < b^k$	$T(n) \in \Theta(n^k)$
$a = b^k$	$T(n) \in \Theta(n^k \log n)$
$a > b^k$	$T(n) \in \Theta(n^{\log_b a})$

Ejemplos de aplicación del esquema con éxito

Ordenación por mezclas (*mergesort*) la operación `descomponer` divide el vector en dos mitades y la operación `combinar` mezcla las dos mitades ordenadas en un vector final.

- $b = 2$ Tamaño mitad de cada subvector.
- $a = 2$ Siempre se generan dos subproblemas.
- $k = 1$ Coste lineal de mezclar.

Coste total $\Theta(n \log n)$.

Ordenación rápida (*quicksort*) la operación `descomponer` elige el pivote, particiona el vector con respecto a él y lo divide en dos mitades. La operación `combinar` es vacía.

- $b = 2$ Tamaño mitad de cada subvector (en el caso mejor).
- $a = 2$ Siempre se generan dos subproblemas.
- $k = 1$ Coste lineal de la partición.

Coste total $\Theta(n \log n)$.

La transformada rápida de Fourier (FFT)

- Un problema históricamente famoso es la Transformada Discreta de Fourier (DFT), que puede resolverse con el algoritmo de tipo *divide y vencerás* conocido como Transformada Rápida de Fourier (FFT), de J.W. Cooley y J.W. Tukey (1965).
- La DFT convierte un conjunto de muestras de amplitud de una señal, en el conjunto de frecuencias que resultan del análisis de Fourier de la misma.
- Esta transformación y su inversa (utilizando el mismo algoritmo DFT) tienen gran interés práctico para filtrar frecuencias indeseadas (p.e. ruido) y mejorar la calidad de las señales de audio o de vídeo.
- La DFT en esencia multiplica una matriz $n \times n$ de números complejos por un vector de longitud n de coeficientes reales, y produce otro vector de la misma longitud. El algoritmo clásico tiene un coste en $O(n^2)$.
- La FFT descompone el vector original en dos vectores de tamaño $n/2$, realiza la FFT de cada uno, y combina los resultados. Las dos partes no recursivas tienen coste lineal, estando en $O(n \log n)$ el coste total de FFT.
- Se utilizó por primera vez para analizar un temblor de tierra que tuvo lugar en Alaska en 1964. El algoritmo clásico empleó 26 minutos en analizar la muestra, mientras que la FFT de Cooley y Tukey lo hizo en 6 segundos.

Ordenación por mezclas: Vector de índices

Vector de índices: la componente i-ésima indica la posición (en el vector de entrada) del elemento que debe ocupar el i-ésimo lugar en la ordenación.

	1	2	3	4	5	6	7	8
vector	3	4	1	0	7	9	6	2

	1	2	3	4	5	6	7	8
índices	4	3	8	1	2	7	5	6

proc `mergesort-índices`(**e** $V[1..n]$ **de** *elem*, *índices*$[1..n]$ **de** $1..n$, **e** c, f : *nat*)

casos

$c > f \rightarrow$ **nada**

$\square$ $c = f \rightarrow$ *índices*$[c] := c$

$\square$ $c < f \rightarrow$ $m := (c + f)$ div 2

`mergesort-índices`$(V, índices, c, m)$

`mergesort-índices`$(V, índices, m+1, f)$

`mezclar-índices`$(V, índices, c, m, f)$

fcasos

fproc

$\{\ 1 \leq c \leq m < f \leq n\ \wedge\ V[I[c]] \leq \ldots \leq V[I[m]]\ \wedge\ V[I[m+1]] \leq \ldots \leq V[I[f]]\ \}$

proc `mezclar-índices`(**e** $V[1..n]$ **de** *elemento*, $I[1..n]$ **de** $1..n$, **e** $c, m, f : 1..n$)

var $aux[1..n]$ **de** $1..n$

$i := c\ ;\ j := m+1\ ;\ k := c$

mientras $i \leq m\ \wedge\ j \leq f$ **hacer**

si $V[I[i]] \leq V[I[j]]$ **entonces**

$aux[k] := I[i]\ ;\ i := i+1$

si no

$aux[k] := I[j]\ ;\ j := j+1$

fsi

$k := k+1$

fmientras

si $i > m$ **entonces** { copiar elementos del segundo subvector }

$aux[k..f] := I[j..f]$

si no { $j > f$, copiar elementos del primer subvector }

$aux[k..f] := I[i..m]$

fsi

$I[c..f] := aux[c..f]$

fproc

$\{\ V[I[c]] \leq \ldots \leq V[I[f]]\ \}$

Tornillos y tuercas

Paquita Manitas ha tenido un percance: se le han mezclado todos los tornillos y tuercas que tenía de varios tamaños. Separar las tuercas de los tornillos ha sido fácil, pero ahora quiere emparejar cada tuerca con un tornillo del tamaño correspondiente y a ojo no consigue distinguir los tamaños, así que la única comparación posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tornillo.

proc `emparejar`$(tornillos[1..n], tuercas[1..n]$ **de** nat^+, **e** $c,f : nat)$
  si $c < f$ **entonces**
    `partición`$(tuercas, c, f, tornillos[c], i, j)$
    `partición`$(tornillos, c, f, tuercas[i], k, l)$
    $\{ i = k \wedge j = l \}$
    `emparejar`$(tornillos, tuercas, c, i-1)$
    `emparejar`$(tornillos, tuercas, j+1, f)$
  fsi
fproc

$<$	$=$	$>$

Tornillos y tuercas: Coste

Sea $n = f - c + 1$

$$T(n) = \begin{cases} c'_0 & n = 0 \\ T(p) + T(q) + c'_1 n & n > 0 \end{cases}$$

con $p + q = n - 1$.

- Cuando se divide siempre por la mitad

$$T(n) = \begin{cases} c'_0 & n = 0 \\ 2T(n/2) + c'_1 n & n > 0 \end{cases}$$

  cuya solución está en $\Theta(n \log n)$ (teorema de la división): caso mejor.
- Cuando el pivote queda sistemáticamente colocado en uno de los extremos

$$T(n) = \begin{cases} c'_0 & n = 0 \\ T(n-1) + c'_0 + c'_1 n & n > 0 \end{cases}$$

  cuya solución está en $\Theta(n^2)$ (teorema de la resta).
- ¿Puede haber casos peores todavía con otras elecciones de p y q?

Tornillos y tuercas: Coste en el caso peor

$T(n) \in O(n^2)$ para cualquier elección de p y q

Demostración: por inducción constructiva que $\forall n : n \geq 1 : T(n) \leq Cn^2$.

Caso base Para $n = 1$ tenemos que

$$T(1) = T(0) + T(0) + c'_1 = 2c'_0 + c'_1 \leq C$$

Paso inductivo Supongamos que la propiedad es cierta para cualquier $m < n$, entonces para $n > 1$:

$$T(n) = T(p) + T(q) + c'_1 n \leq^{h.i.} Cp^2 + Cq^2 + c'_1 n.$$

Como $p + q = n - 1$, $p^2 + q^2 = (n-1)^2 - 2pq \leq (n-1)^2$, por lo que

$$T(n) \leq C(n-1)^2 + c'_1 n = Cn^2 - 2Cn + C + c'_1 n.$$

De aquí, $Cn^2 - 2Cn + C + c'_1 n \leq Cn^2 \Leftrightarrow C + (c'_1 - 2C)n \leq 0$.

Según la primera restricción, podemos tomar $C = 2c'_0 + c'_1$, y comprobamos que cumple también la última restricción:

$$(2c'_0 + c'_1) + (c'_1 - 2(2c'_0 + c'_1))n = (2c'_0 - 4c'_0 n) + (c'_1 - c'_1 n) \overset{n>1}{<} 0 + 0 = 0$$

Basta tomar $C = 2c'_0 + c'_1$ para que $\forall n : n \geq 1 : T(n) \leq Cn^2$.

Tornillos y tuercas: Coste en el caso medio

$T(n) \in O(n \log n)$

Demostración:

p y q entre 0 a $n-1$, siempre cumpliendo $p+q = n-1$, es decir, $q = n-p-1$.

$$
\begin{aligned}
T_m(n) &= \frac{1}{n} \sum_{p=0}^{n-1} (T_m(p) + T_m(n-p-1) + (n-1)) \\
&= \frac{1}{n} \left(\sum_{p=0}^{n-1} T_m(p) + \sum_{p=0}^{n-1} T_m(n-p-1) + \sum_{p=0}^{n-1} (n-1) \right) \\
&= \frac{1}{n} \left(\sum_{p=0}^{n-1} T_m(p) + \sum_{j=0}^{n-1} T_m(j) \right) + (n-1) \\
&= (n-1) + \frac{2}{n} \sum_{p=0}^{n-1} T_m(p) \\
&= (n-1) + \frac{2}{n} \sum_{p=2}^{n-1} T_m(p)
\end{aligned}
$$

ya que $T_m(0) = T_m(1) = 0$.

Es una recurrencia con historia.

$$nT_m(n) = n(n-1) + 2\sum_{p=2}^{n-1} T_m(p)$$

$$(n-1)T_m(n-1) = (n-1)(n-1-1) + 2\sum_{p=2}^{n-2} T_m(p)$$

y restando,

$$\begin{aligned} nT_m(n) - (n-1)T_m(n-1) &= 2T_m(n-1) + 2(n-1) \\ nT_m(n) &= (n+1)T_m(n-1) + 2(n-1) \end{aligned}$$

Dividiendo por $n(n+1)$:

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

$$S(n) = \frac{T_m(n)}{n+1} \qquad S(n) = \begin{cases} 0 & n \leq 1 \\ S(n-1) + \frac{2(n-1)}{n(n+1)} & n \geq 2 \end{cases}$$

como se cumple $1 > \frac{n-1}{n+1}$ podemos simplificar $S(n)$,

$$S(n) \leq \begin{cases} 0 & n \leq 1 \\ S(n-1) + \frac{2}{n} & n \geq 2 \end{cases}$$

$$\begin{aligned} S(n) &\leq S(n-1) + \frac{2}{n} \\ &\leq S(n-2) + \frac{2}{n-1} + \frac{2}{n} \\ &\leq S(n-3) + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} \\ &\cdots \\ &\leq S(n-i) + 2\sum_{j=n-i+1}^{n} \frac{1}{j} \end{aligned}$$

La recurrencia desaparece cuando $n - i = 1$ $(i = n-1)$:

$$S(n) \leq S(1) + 2\sum_{j=2}^{n} \frac{1}{j} \leq 2\int_1^n \frac{1}{x}\mathrm{d}x = 2\ln n \Rightarrow T_m(n) = (n+1)S(n) \in O(n \log n)$$

Problema de selección

Dado un vector $V[1..n]$ de elementos que se pueden ordenar, y un entero k, $1 \leq k \leq n$, encontrar el k-ésimo menor elemento.

¿Ordenar el vector V y tomar $V[k]$? Coste en $O(n \log n)$.

¿Utilizar el algoritmo de `partición`?

$i \leq k \leq j$ el elemento $V[k]$ es el k-ésimo.

$k < i$ buscar el k-ésimo en $V[c..i-1]$.

$k > j$ buscamos el $(k-p)$-ésimo en $V[j+1..f]$.

```
{ k representa una posición absoluta, c ≤ k ≤ f }
proc selección1(V[1..n] de elem, e c,f,k : nat, k-ésimo : elem)
    si c = f entonces k-ésimo := V[c]
    si no
        partición(V,c,f,V[c],i,j)
        casos
              k < i  →  selección1(V,c,i-1,k,k-ésimo)
           □ i ≤ k ∧ k ≤ j  →  k-ésimo := V[k]
           □ k > j  →  selección1(V,j+1,f,k,k-ésimo)
        fcasos
    fsi
fproc
```

Caso peor: (el pivote queda siempre en un extremo) $\Theta(n^2)$.
Caso medio: $\Theta(n)$.

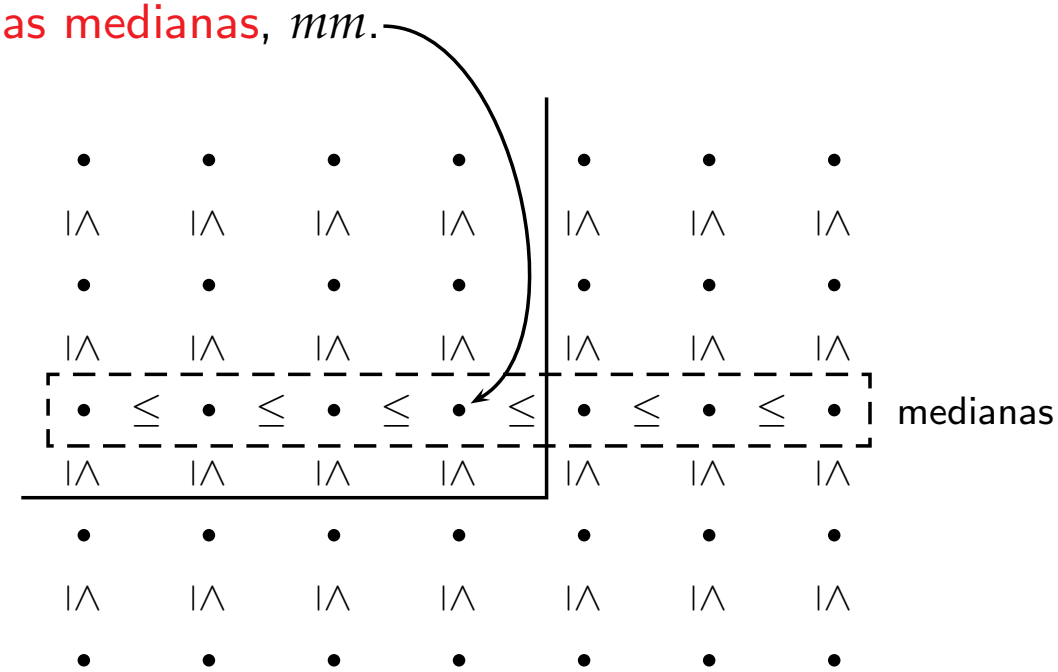
Problema de selección: Mediana de las medianas

¿Cómo asegurar que el pivote no va a quedar en un extremo?

Tomar como pivote la mediana del subvector (el $\lceil \frac{n}{2} \rceil$-ésimo elemento).

¡Pero calcular la mediana es un caso particular del problema de selección: $k = (c + f) \text{ div } 2$!

Nos conformamos con una aproximación suficientemente buena de la mediana: la mediana de las medianas, mm.



Dividir el vector en grupos de 5 elementos;
calcular la mediana de cada grupo;
mm es la mediana de esas $n \text{ div } 5$ medianas.

Problema de selección: Mediana de las medianas

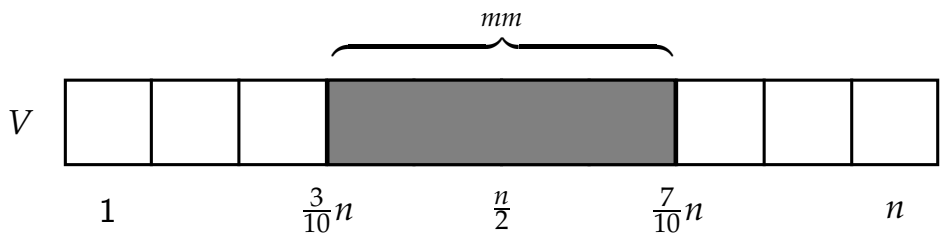
Utilizar mm como pivote para particionar el vector V.

Al menos $\frac{3(n \text{ div } 5)}{2}$ elementos de V son menores o iguales que mm.

Como $n \text{ div } 5 \geq \frac{n-4}{5}$, concluimos que al menos $\frac{3n-12}{10}$ elementos de V son menores o iguales que mm, y, por tanto, como mucho $\frac{7n+12}{10}$ elementos son estrictamente mayores que mm.

Lo mismo para elementos mayores o iguales y estrictamente menores.

$\frac{7n+12}{10}$ es cota superior del número de elementos para las llamadas recursivas.



Problema de selección: Mediana de las medianas

Los pasos del algoritmo `selección2`$(V, c, f, k, elemento)$ son:

1. calcular la mediana de cada grupo de 5 elementos.
   Son n div 5 medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y escoger el tercero.
2. calcular mm, con una llamada recursiva a `selección2` con n div 5 elementos.
3. llamar a `partición`(V, c, f, mm, i, j) utilizando como pivote mm.
4. hacer una distinción de casos similar a la de `selección1`:

casos

$$\begin{array}{rcl} k < i & \rightarrow & \texttt{selección2}(v, c, i-1, k, elemento) \\ \square\ i \leq k \leq j & \rightarrow & elemento := mm \\ \square\ k > j & \rightarrow & \texttt{selección2}(v, j+1, f, k, elemento) \end{array}$$

fcasos

- Las llamadas recursivas se realizan con $\frac{7n+12}{10}$ elementos como mucho.
- El tiempo requerido por `selección2` es lineal en el caso peor (demostración por inducción constructiva).

Problema de selección: Mediana de las medianas, Implementación

$\{\ c \leq k \leq f\ \}$

proc `selección2`$(V[1..n]$ **de** *elem*, **e** c, f, k : *nat*, *k-ésimo* : *elem*$)$

  $t := f - c + 1$

  si $t \leq 12$ **entonces**

    `ordenar`(V, c, f) ; $k\text{-ésimo} := V[k]$

  si no

    $s := t$ div 5

    para $l = 1$ **hasta** s **hacer**

      `ordenar`$(V, c + 5 * (l - 1), c + 5 * l - 1)$

      $pm := c + 5 * (l - 1) + 5$ div 2

      `intercambiar`$(V[c + l - 1], V[pm])$

    fpara

    $\{$ medianas situadas en $V[c, c + s - 1]$ $\}$

    `selección2`$(V, c, c + s - 1, c + (s - 1)$ div $2, mm)$ $\{$ mediana de las medianas $\}$

    `partición`(V, c, f, mm, i, j)

    casos

      $k < i \rightarrow$ `selección2`$(V, c, i - 1, k, k\text{-ésimo})$

      $\square$ $i \leq k \wedge k \leq j \rightarrow k\text{-ésimo} := mm$

      $\square$ $k > j \rightarrow$ `selección2`$(V, j + 1, f, k, k\text{-ésimo})$

    fcasos

  fsi

fproc

Ordenación rápida de Hoare: Mejora

¿Se puede mejorar `quicksort` para que requiera un tiempo que esté en $O(n \log n)$ incluso en el caso peor? Sí, *pero no*.

- Seleccionar como pivote la mediana (en tiempo lineal).
- Sigue siendo $O(n^2)$ en el caso peor (todos los elementos son iguales).
- Utilizar el algoritmo de partición que divide el vector en tres partes y hacer las llamadas recursivas solo con los elementos menores y mayores.

```
proc quicksort-opt(V[1..N] de elem, e c,f : nat)
    si c < f entonces
        m := mediana(V,c,f)
        partición(V,c,f,m,i,j)
        quicksort-opt(V,c,i-1)
        quicksort-opt(V,j+1,f)
    fsi
fproc
```

- `quicksort-opt` requiere un tiempo $O(n \log n)$ incluso en el caso peor, pero la constante multiplicativa es tan alta que en la práctica sería peor que otros algoritmos en todos los casos.

Lo más caro y lo más barato

Acabas de hacer la compra en Merca-Madonna y tienes los precios de todos los productos que has comprado en un vector $V[1..N]$. Quieres saber cuál ha sido el producto más caro y cuál el más barato. Suponiendo que n es una potencia de 2 mayor que 2, tienes que averiguarlo realizando menos de $2N-3$ comparaciones entre precios.

Máximo: $N-1$ comparaciones,

Mínimo: $N-2$ comparaciones adicionales;

Total: $2N-3$ comparaciones entre elementos.

$$\langle máx_1, mín_1 \rangle := \texttt{máx-mín}(V[1..N/2])$$
$$\langle máx_2, mín_2 \rangle := \texttt{máx-mín}(V[N/2+1..N])$$
$$máx := \texttt{máx}(máx_1, máx_2)$$
$$mín := \texttt{mín}(mín_1, mín_2)$$

El número total de comparaciones entre elementos sería:
$2(\frac{N}{2}) - 3 + 2(\frac{N}{2}) - 3 + 1 + 1 = 2N - 4$.

fun máx-mín1$(V[1..N]$ **de** *elem*$, c, f$: *nat*$)$ **dev** $\langle$ *máx*$,$ *mín* : *elem* $\rangle$
  si $c = f$ **entonces** $\langle$ *máx*$,$ *mín* $\rangle := \langle\, V[c], V[c] \,\rangle$
  si no
    $m := (c+f)$ div 2
    $\langle$ *máx*$_1$, *mín*$_1$ $\rangle$:= máx-mín1(V, c, m)
    $\langle$ *máx*$_2$, *mín*$_2$ $\rangle$:= máx-mín1$(V, m+1, f)$
    máx := máx$($*máx*$_1$, *máx*$_2)$
    mín := mín$($*mín*$_1$, *mín*$_2)$
  fsi
ffun

Sea $n = f - c + 1$

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + 2 & n > 1 \end{cases}$$

$$T(n) = 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j$$

$$T(n) = 2^{\log n} T(1) + \sum_{j=1}^{\log n} 2^j = 2n - 2$$

fun máx-mín2$(V[1..N]$ **de** *elem*, c, f : *nat*$)$ **dev** $\langle$ *máx*, *mín* : *elem* $\rangle$

casos

$c = f \;\rightarrow\; \langle$ *máx*, *mín* $\rangle := \langle\, V[c], V[c] \,\rangle$

$\square$ $c + 1 = f \;\rightarrow\;$ { hay dos elementos }

si $V[c] < V[f]$ **entonces** $\langle$ *máx*, *mín* $\rangle := \langle\, V[f], V[c] \,\rangle$

si no $\langle$ *máx*, *mín* $\rangle := \langle\, V[c], V[f] \,\rangle$

fsi

$\square$ $c + 1 < f \;\rightarrow$

$m := (c + f)$ div 2

$\langle$ *máx*$_1$, *mín*$_1$ $\rangle :=$ máx-mín2(V, c, m)

$\langle$ *máx*$_2$, *mín*$_2$ $\rangle :=$ máx-mín2$(V, m + 1, f)$

máx := máx$($*máx*$_1$, *máx*$_2)$

mín := mín$($*mín*$_1$, *mín*$_2)$

fcasos

ffun

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

$$T(n) = 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j$$

$$\begin{aligned} T(n) &= 2^{\log n - 1} T(2) + \sum_{j=1}^{\log n - 1} 2^j \\ &= \frac{n}{2} + 2^{\log n} - 2 = \frac{3}{2}n - 2 \\ &= \frac{3}{2}n - 2 + 1 - 1 \\ &< \frac{3}{2}n - 2 + \frac{1}{2}n - 1 = 2n - 3 \end{aligned}$$

Determinación del umbral

casos

$n \leq n_0 \rightarrow$ subalgoritmo básico

$n > n_0 \rightarrow$ dividir

llamadas recursivas

componer

fcasos

donde n_0 es el umbral.

La recursión exige más tiempo y espacio (pila) y además hay que dividir/componer (constante multiplicativa grande).
¿Cuándo merece la pena resolver el problema dividiendo? Determinar n_0.

- El umbral depende del algoritmo divide y vencerás, del subalgoritmo básico y del computador/compilador concreto.
- El umbral no afecta al orden del tiempo de ejecución, pero sí a la constante multiplicativa.

Umbral óptimo: n_0 tal que

$n \leq n_0$ es menos costoso llamar al subalgoritmo básico que dividir;

$n > n_0$ menos costoso dividir.

Pero este umbral óptimo no siempre existe.

Coches de colores: elemento mayoritario

Jaimito tiene una fantástica colección de coches en miniatura de diferentes colores y quiere averiguar si alguno de estos colores es mayoritario, es decir, si hay algún color tal que el número de coches de dicho color sea estrictamente mayor que $N/2$, siendo N la cantidad de coches de Jaimito.

```
fun mayoritario1(V[1..N] de elem, c, f : nat) dev ⟨ existe : bool, mayor : elem ⟩
    si c = f entonces ⟨ existe, mayor ⟩ := ⟨ cierto, V[c] ⟩
    si no
        m := (c + f) div 2
        ⟨ existe_1, mayor_1 ⟩ := mayoritario1(V, c, m)
        ⟨ existe_2, mayor_2 ⟩ := mayoritario1(V, m + 1, f)
        existe := falso
        si existe_1 entonces      { comprobamos el primer candidato }
            ⟨ existe, mayor ⟩ := ⟨ comprobar(V, mayor_1, c, f), mayor_1 ⟩
        fsi
        si ¬existe ∧ existe_2 entonces      { comprobamos el segundo candidato }
            ⟨ existe, mayor ⟩ := ⟨ comprobar(V, mayor_2, c, f), mayor_2 ⟩
        fsi
    fsi
ffun
```

fun comprobar$(V[1..N]$ **de** *elem*$, x$: *elem*$, c, f$: *nat*$)$ **dev** *válido* : *bool*
    veces $:= 0$
    para $i = c$ **hasta** f **hacer**
        si $V[i] = x$ **entonces** *veces* $:=$ *veces* $+ 1$ **fsi**
    fpara
    válido $:=$ *veces* $> (f - c + 1)$ div 2
ffun

Coste de `mayoritario1`:

$$T(n) = \begin{cases} c_0 & n = 1 \\ 2T(n/2) + c_1 n & n > 1 \end{cases}$$

$T(n) \in \Theta(n \log n)$

Elemento mayoritario

¿Se puede hacer mejor?

Anular entre sí elementos diferentes. Solo sobrevivirá el candidato a mayoritario.

```
fun mayoritario2(V[1..N] de elem) dev ⟨existe : bool, mayor : elem⟩
    candidato := V[1]
    contar := 1    { cuántas veces más ha aparecido el candidato }
    para i = 2 hasta N hacer
        si contar = 0 entonces { elegimos nuevo candidato }
            candidato := V[i] ; contar := 1
        si no { contar > 0 }
            si V[i] = candidato entonces contar := contar + 1
            si no contar := contar − 1
            fsi
        fsi
    fpara
    ⟨existe, mayor⟩ := ⟨comprobar(V, candidato, 1, N), candidato⟩
ffun
```

Coste: $\Theta(N)$