Divide y vencerás

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Octubre 2008

Bibliografía

- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulo 2
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
  Capítulo 3
- N. Martí Oliet, C. Segura Díaz y J. A. Verdejo López. *Especificación, derivación y análisis de algoritmos: Ejercicios resueltos*. Colección Prentice Practica, Pearson/Prentice Hall, 2006.
  Capítulo 11

Divide y vencerás

- La técnica divide y vencerás consiste en descomponer el problema que haya que resolver en una serie de subproblemas más pequeños, resolver estos subproblemas y combinar después los resultados para obtener la solución del problema original.
- Lo importante es que los subproblemas son del mismo tipo que el problema original, y se resuelven usando la misma técnica. Algoritmo recursivo.
- Así, se van generando problemas del mismo tipo cada vez más pequeños, hasta llegar a subproblemas *suficientemente pequeños* para ser resueltos sin división.

Esquema general

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

Costes

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 tenemos

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

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

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

Búsqueda binaria

Dado un vector ordenado $V[1..n]$ y un elemento x, el problema consiste en 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.

Por tanto hay dos posibles resultados:

ÉXITO: el elemento x está en la posición p, $V[p] = x$, y

FALLO: el elemento x no está en el vector, y p es la posición que tendría en caso de estar.

fun búsqueda-binaria$(V[1..n]$ **de** *ent*$, x :$ *ent*$, c, f :$ *nat*$)$ **dev** $\langle b :$ *bool*$, p :$ *nat*$\rangle$
  si $c > f$ **entonces** $\langle b, p \rangle := \langle \texttt{falso}, c \rangle$
  si no
    $m := (c + f) \operatorname{div} 2$;
    casos
      $x < V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria$(V, x, c, m-1)$
      $\square\ x = V[m] \rightarrow \langle b, p \rangle := \langle \texttt{cierto}, m \rangle$
      $\square\ x > V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria$(V, x, m+1, f)$
    fcasos
  fsi
ffun

Ejemplo

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V	-5	-3	1	6	15	16	27	41	48	53	72	79	84	99

Si buscamos el entero 99, los parametros i, j de las sucesivas llamadas, y la variable m toman los siguientes valores:

i	j	m	
1	14	7	
8	14	11	
12	14	13	
14	14	14	encontrado

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V	-5	-3	1	6	15	16	27	41	48	53	72	79	84	99

Si buscamos el entero -4, tendremos

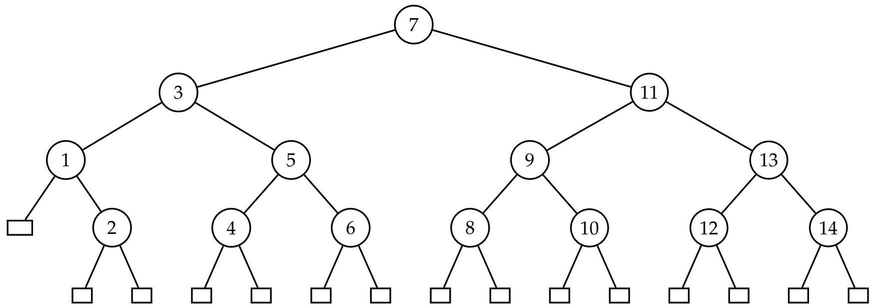
i	j	m	
1	14	7	
1	6	3	
1	2	1	
2	2	2	
2	1		no encontrado

Y si buscamos el entero 15, obtendremos

i	j	m	
1	14	7	
1	6	3	
4	6	5	encontrado

Número de comparaciones

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
V	-5	-3	1	6	15	16	27	41	48	53	72	79	84	99
N° comp.	3	4	2	4	3	4	1	4	3	4	2	4	3	4



Caso peor: 4

Caso medio (éxito): $\frac{45}{14} \simeq 3{,}21$

Caso medio (fallo): $\frac{59}{15} \simeq 3{,}93$

Búsqueda binaria: caso peor

- Cuando el tamaño n del vector cumple $2^{k-1} \leq n < 2^k$, se realizan como mucho k comparaciones en una búsqueda con ÉXITO, y $k-1$ o k en caso de FALLO.
- Si $2^{k-1} \leq n < 2^k$, los nodos circulares, que representan éxitos, en el árbol de decisiones correspondiente, se encuentran en los niveles $1, 2, \ldots, k$, y son necesarias i comparaciones para terminar en un nodo ÉXITO en el nivel i
- Los nodos cuadrados, fallos, se encuentran en los niveles k o $k+1$, y para terminar en un nodo FALLO en el nivel i se necesitan $i-1$ comparaciones.
- El tiempo del algoritmo está en $\Theta(\log n)$ en el caso peor.

Búsqueda binaria: caso medio

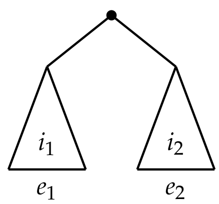
- Tenemos que relacionar el tamaño del árbol de decisiones con el número de comparaciones hechas por el algoritmo.
- La distancia de un nodo a la raíz es uno menos que su nivel.
- Llamamos longitud del camino interno, I, a la suma de las distancias a la raíz de todos los nodos internos, y longitud del camino externo, E, a la suma de las distancias de todos los nodos externos a la raíz.
- Para cualquier árbol binario con i nodos internos, se cumple que

$$E = I + 2i.$$

Lo demostramos por inducción sobre el número de nodos internos.

Base: Cuando $i = 0$ tenemos solo un nodo (externo), y se cumple $E = 0$ e $I = 0$, por lo que tenemos $E = I + 2i$.

Paso inductivo: Supongamos un árbol con $i > 0$ nodos internos, y e nodos externos, repartidos así



donde

$$\begin{aligned} i &= i_1 + i_2 + 1 \\ e &= e_1 + e_2 \end{aligned}$$

Por hipótesis de inducción,

$$\begin{aligned} E_1 &= I_1 + 2i_1 \\ E_2 &= I_2 + 2i_2 \end{aligned}$$

Al formar el nuevo árbol, la distancia de todos los nodos en los hijos izquierdo y derecho ha aumentado en 1. Por tanto,

$$\begin{aligned} E &= E_1 + E_2 + e_1 + e_2 \\ I &= I_1 + I_2 + i_1 + i_2 + 0 \end{aligned}$$

y tenemos,

$$\begin{aligned} I + 2i &= I_1 + I_2 + i_1 + i_2 + 2(i_1 + i_2 + 1) \\ &= I_1 + I_2 + 3i_1 + 3i_2 + 2 \\ &= E_1 + E_2 + i_1 + i_2 + 2 \\ &=^* E_1 + E_2 + e_1 + e_2 \\ &= E \end{aligned}$$

* Un árbol no vacío con i nodos internos tiene $e = i + 1$ nodos externos.

$NM_E(n)$ = número medio de comparaciones en caso de éxito

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

$NM_F(n)$ = número medio de comparaciones en caso de fallo

$$NM_F(n) = \frac{E}{n+1}.$$

Los nodos externos están situados a una distancia proporcional a la altura del árbol de decisiones, que por ser equilibrado está en $\Theta(\log n)$, siendo n el número de nodos internos.

Por tanto, E es proporcional a $n \log n$, y $NM_F(n) \in \Theta(\log n)$.

Para el caso de éxito, tenemos

$$NM_E(n) = 1 + \frac{I}{n} = 1 + \frac{E - 2n}{n} = \frac{E}{n} - 1 \in \Theta(\log n)$$

Búsqueda binaria: complejidad

Resumimos los resultados en la siguiente tabla:

	MEJOR	MEDIO	PEOR
ÉXITO	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
FALLO	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Buscar el máximo y el mínimo

Sea $V[1..n]$ un vector cuyos elementos se pueden comparar entre sí. Suponiendo que n es una potencia de 2 mayor que 2, tenemos que encontrar el máximo y el mínimo de V realizando menos de $2n-3$ comparaciones entre elementos.

- Encontrar el máximo realizando $n-1$ comparaciones,
- determinar el mínimo realizando $n-2$ comparaciones adicionales;
- un total de $2n-3$ comparaciones entre elementos.

$$\begin{array}{l}
\langle \mathit{máx}_1, \mathit{mín}_1 \rangle := \texttt{máx-mín}(V[1..n/2]) \\
\langle \mathit{máx}_2, \mathit{mín}_2 \rangle := \texttt{máx-mín}(V[n/2+1..n]) \\
\mathit{máx} := \texttt{máx}(\mathit{máx}_1, \mathit{máx}_2) \\
\mathit{mín} := \texttt{mín}(\mathit{mín}_1, \mathit{mín}_2)
\end{array}$$

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

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

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

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

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

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

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

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

Mergesort

proc $\texttt{mergesort}(V[1..n]$ **de** *elem*, **e** c, f : *nat*)
    si $c < f$ **entonces**
        $m := (c + f) \operatorname{div} 2$
        $\texttt{mergesort}(V, c, m)$
        $\texttt{mergesort}(V, m + 1, f)$
        $\texttt{mezclar}(V, c, m, f)$
    fsi
fproc

proc $\texttt{mezclar}(V[1..n]$ **de** *elem*, **e** c, m, f : *nat*)
var $W[1..n]$
    $i := c$; $j := m + 1$; $k := c$
    mientras $i \leq m \wedge j \leq f$ **hacer**
        si $V[i] \leq V[j]$ **entonces**
            $W[k] := V[i]$; $i := i + 1$
        si no
            $W[k] := V[j]$; $j := j + 1$
        fsi
        $k := k + 1$
    fmientras

si $i > m$ **entonces** { quedan elementos en la segunda mitad }
para $l = j$ **hasta** f **hacer**
$W[k] := V[l]$
$k := k+1$
fpara
si no { quedan elementos en la primera mitad }
para $l = i$ **hasta** m **hacer**
$W[k] := V[l]$
$k := k+1$
fpara
fsi
{ copiamos el resultado en V }
para $l = c$ **hasta** f **hacer**
$V[l] := W[l]$
fpara
fproc

Mergesort: coste

Si $T(n)$ representa el tiempo en el caso peor del algoritmo `mergesort` cuando el tamaño del vector a ordenar es n, $T(n)$ se define con la siguiente recurrencia:

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

donde $c_1 n$ es el tiempo del algoritmo `mezclar`.

Aplicando el teorema de la división sabemos que

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

Mergesort con enlaces

Podemos ahorrarnos las copias que hace `mezclar` entre el vector V y el vector auxiliar utilizando *listas enlazadas*.

Cada elemento se representa por su posición en el vector inicial. Los subvectores ya ordenados se representan mediante listas de posiciones.

La implementación de estas listas se puede hacer sobre un vector de *enlaces*, de forma que cada entrada en el vector de enlaces indica la posición del siguiente elemento, y el valor 0 indica el final de una lista.

	1	2	3	4	5	6	7	8
enlaces	3	4	0	1	7	8	6	0

El entero 2 denota el comienzo de la lista $l_1 = (2,4,1,3)$, y el entero 5 el comienzo de la lista $l_2 = (5,7,6,8)$. Si interpretamos las listas como que están describiendo el orden en el vector $V[1..8]$, la conclusión es que $V[2] \leq V[4] \leq V[1] \leq V[3]$ y $V[5] \leq V[7] \leq V[6] \leq V[8]$.

proc `mergesort-enlaces`($\mathbf{e}\ V[1..n]$ **de** $\mathit{elem}, \mathit{enlaces}[0..n]$ **de** $0..n$, **e** $c, f : \mathit{nat}, p : 0..n$)
{ p será la posición inicial de la lista correspondiente a $V[c..f]$ }
  casos
    $c > f \ \rightarrow$ **nada**
  □ $c = f \ \rightarrow \ p := c$
  □ $c < f \ \rightarrow$
      $m := (c+f)$ div 2
      `mergesort-enlaces`$(V, \mathit{enlaces}, c, m, p_1)$
      `mergesort-enlaces`$(V, \mathit{enlaces}, m+1, f, p_2)$
      `mezclar-enlaces`$(V, \mathit{enlaces}, p_1, p_2, p)$
  fcasos
fproc

Con llamada inicial:

$\mathit{enlaces} := [0, \ldots, 0]$
`mergesort-enlaces`$(V, \mathit{enlaces}, 1, n, p)$
`ordenar-enlaces`$(V, \mathit{enlaces}, p)$

$\{ \, p_1$ es el comienzo de la primera lista, p_2 es el de la segunda y $\}$
$\{ \, p$ será el de la lista mezclada $\}$

```
proc mezclar-enlaces(e V[1..n] de elem, enlaces[0..n] de 0..n, e p1, p2 : 0..n, p : 0..n)
    i := p1 ; j := p2 ; k := 0
    { k es el último en la lista que se construye }
    mientras i ≠ 0 ∧ j ≠ 0 hacer
        si V[i] ≤ V[j] entonces
            enlaces[k] := i ; k := i ; i := enlaces[i]
        si no
            enlaces[k] := j ; k := j ; j := enlaces[j]
        fsi
    fmientras
    si i = 0 entonces enlaces[k] := j    { enlazar resto del segundo vector }
    si no enlaces[k] := i    { enlazar resto del primer vector }
    fsi
    p := enlaces[0]    { comienzo lista construida }
fproc
```


Mergesort con enlaces

```
proc ordenar-enlaces(V[1..n] de elem, e enlaces[0..n] de 0..n, e p : 0..n)
var W[1..n] de elem
    i := p
    para k = 1 hasta n hacer
        W[k] := V[i] ; i := enlaces[i]
    fpara
    V := W
fproc
```

Ejemplo de ordenación por mezclas con enlaces

	1	2	3	4	5	6	7	8
V	24	12	80	7	15	43	29	20

			0	1	2	3	4	5	6	7	8	
		V	-	24	12	80	7	15	43	29	20	
		enlaces	0	0	0	0	0	0	0	0	0	
p_1	p_2	p										
1	2	2	2	0	1	0	0	0	0	0	0	$(12,24)$
3	4	4	4	0	1	0	3	0	0	0	0	$(12,24),(7,80)$
2	4	4	4	3	1	0	2	0	0	0	0	$(7,12,24,80)$
5	6	5	5	3	1	0	2	6	0	0	0	$(7,12,24,80),(15,43)$
7	8	8	8	3	1	0	2	6	0	0	7	$(7,12,24,80),(15,43),(20,29)$
5	8	5	5	3	1	0	2	8	0	6	7	$(7,12,24,80),(15,20,29,43)$
4	5	4	4	7	5	0	2	8	3	6	1	$(7,12,15,20,24,29,43,80)$

Quicksort

proc `quicksort`$(V[1..n]$ **de** *elem*, **e** c, f : *nat*$)$
    si $c < f$ **entonces**
        `partición`(V, c, f, p)
        `quicksort`$(V, c, p-1)$
        `quicksort`$(V, p+1, f)$
    fsi
fproc

partición

proc partición($V[1..n]$ **de** *elem*, **e** c, f : *nat*, p : *nat*)
{ p es la posición donde queda colocado el pivote }
    $piv := V[c]$ { se toma como pivote el primer elemento }
    $i := c + 1$
    $d := f$
    mientras $i \neq d + 1$ **hacer**
        mientras $i \leq d \wedge V[i] \leq piv$ **hacer**
            $i := i + 1$
        fmientras
        mientras $i \leq d \wedge V[d] \geq piv$ **hacer**
            $d := d - 1$
        fmientras
        si $i < d$ **entonces**
            intercambiar(V, i, d)
            $i := i + 1$
            $d := d - 1$
        fsi
    fmientras
    intercambiar(V, c, d)
    $p := d$ { pivote colocado }
fproc

Quicksort: coste

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

con $p + q = n - 1$.

Cuando se divide siempre por la mitad, la recurrencia se aproxima de la forma

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

cuya solución está en $\Theta(n \log n)$ por el teorema de la división.

Cuando el pivote queda sistemáticamente colocado en uno de los extremos, la recurrencia queda

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

cuya solución está en $\Theta(n^2)$ por el teorema de la resta.

¿Puede haber casos peores todavía con otras elecciones de p y q?

Quicksort: caso peor

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

Demostramos mediante *inducción constructiva* que $\forall n : n \geq 1 : T(n) \leq Cn^2$.

En el caso básico $n = 1$ tenemos que

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

Supongamos que la propiedad es cierta para cualquier $m < n$.

Entonces, en el paso de inducción tenemos 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, $Cn^2 - 2Cn + C + c'_1 n \leq Cn^2 \Leftrightarrow C + (c'_1 - 2C)n \leq 0$.

De acuerdo con la primera restricción, podemos tomar como constante $C = 2c'_0 + c'_1$, y comprobamos que esta elección satisface 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 $\forall n : n \geq 1 : T(n) \leq Cn^2$.

Quicksort: caso medio

El caso medio está determinado por la variación en los tamaños de los vectores con los que se hacen las llamadas recursivas, p y q. Ambos pueden variar de 0 a $n-1$, pero siempre cumpliendo $p + q = n - 1$, o $q = n - p - 1$.

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

donde el último paso se puede dar ya que $T_m(0) = T_m(1) = 0$.

Es una recurrencia con historia. La resolvemos calculando el término para n, para $n-1$ y restando. La recurrencia para n es

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

para $n-1$,

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

y restando,

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

Dividiendo por $n(n+1)$, queda:

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

Definimos

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

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

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

Ahora, desplegando,

$$\begin{array}{rcl} S(n) & \leq & S(n-1) + \dfrac{2}{n} \\ & \leq & S(n-2) + \dfrac{2}{n-1} + \dfrac{2}{n} \\ & \leq & S(n-3) + \dfrac{2}{n-2} + \dfrac{2}{n-1} + \dfrac{2}{n} \\ & \vdots & \\ & \leq & S(n-i) + 2\displaystyle\sum_{j=n-i+1}^{n} \frac{1}{j} \end{array}$$

La recurrencia desaparece cuando $n - i = 1$, o $i = n - 1$. En ese caso,

$$S(n) \leq S(1) + 2\sum_{j=2}^{n} \frac{1}{j} = 2\sum_{j=2}^{n} \frac{1}{j} \leq 2\int_{1}^{n} \frac{1}{x}\mathrm{d}x = 2\ln n$$

Y por tanto

$$\begin{aligned} T_m(n) &= (n+1)S(n) \\ &\leq (n+1)2\ln n \\ &= \frac{2}{\log e}(n+1)\log n \\ &\approx 1{,}386(n+1)\log n \\ &\in O(n\log n) \end{aligned}$$

¿Se puede mejorar quicksort?

- ¿Se puede mejorar `quicksort` para que requiera un tiempo que esté en $O(n \log n)$ incluso en el caso peor? Sí, *pero no*.
- Una solución sería seleccionar la *mediana* (el elemento que ocuparía la posición $(n + 1)$ div 2 del vector si este se ordenara) de $V[c..f]$ como pivote, lo cual se puede hacer en un tiempo lineal, como veremos.
- Pero seguimos requiriendo un tiempo $O(n^2)$ en el caso peor, cuando todos los elementos son iguales.
- Podemos utilizar un algoritmo (`partición2`) que divida el vector en tres partes: elementos menores que el pivote, elementos iguales al pivote y elementos mayores que el pivote; y hacer las llamadas recursivas con los elementos menores y mayores.

V	$<$	$=$	$>$

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

partición2

proc `partición2`$(V[1..n]$ **de** *elem*, **e** c,f : *nat*, **e** *pivote* : *elem*, i,j : *nat*$)$
  { los índices i y j son parámetros de salida, y describen la partición }
  $i := c\ ;\ \ k := c\ ;\ \ j := f$
  { Inv: $(\forall r : c \le r < i : V[r] < pivote) \ \wedge\ (\forall s : i \le s < k : V[s] = pivote)$ }
  { $\quad \wedge\ (\forall t : j < t \le f : V[t] > pivote) \ \wedge\ i \le k \le j$ }
  mientras $k \le j$ **hacer**
    casos
      $V[k] < pivote \ \rightarrow\ $ `intercambiar`$(V,k,i)\ ;\ \ i := i+1\ ;\ \ k := k+1$
      □ $V[k] = pivote \ \rightarrow\ k := k+1$
      □ $V[k] > pivote \ \rightarrow\ $ `intercambiar`$(V,k,j)\ ;\ \ j := j-1$
    fcasos
  fmientras
  { $(\forall r : c \le r < i : V[r] < pivote) \ \wedge\ (\forall s : i \le s \le j : V[s] = pivote)$ }
  { $\wedge\ (\forall t : j < t \le f : V[t] > pivote)$ }
fproc

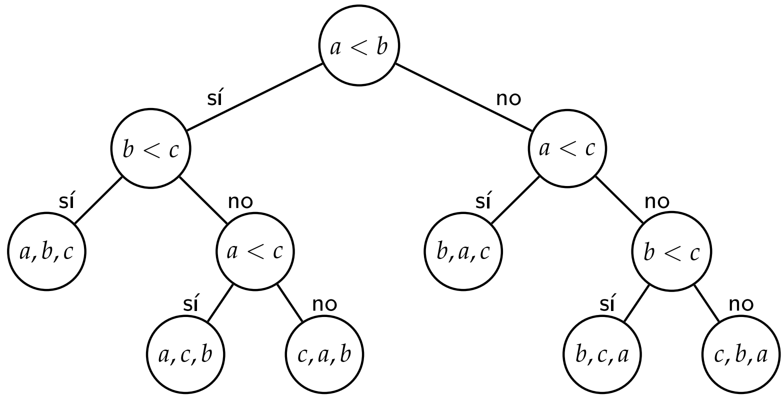
Cota inferior de la ordenación basada en comparaciones

Vamos a ver que el número de comparaciones necesarias para ordenar n valores distintos está en $\Omega(n \log n)$, si para ordenar nos limitamos a utilizar comparaciones entre elementos.

Primero consideremos el siguiente algoritmo que ordena tres valores:

proc `ordenar-tres`(a,b,c)
  si $a < b$ **entonces**
    si $b < c$ **entonces** $\langle a,b,c \rangle := \langle a,b,c \rangle$
    si no si $a < c$ **entonces** $\langle a,b,c \rangle := \langle a,c,b \rangle$
        si no $\langle a,b,c \rangle := \langle c,a,b \rangle$ **fsi**
    fsi
  si no { $b < a$ }
    si $a < c$ **entonces** $\langle a,b,c \rangle := \langle b,a,c \rangle$
    si no si $b < c$ **entonces** $\langle a,b,c \rangle := \langle b,c,a \rangle$
        si no $\langle a,b,c \rangle := \langle c,b,a \rangle$ **fsi**
    fsi
  fsi
fproc

Podemos asociar un árbol binario al procedimiento `ordenar-tres` que indique cómo se van haciendo las comparaciones.



A este árbol se le llama árbol de decisión, porque en cada nodo se toma una decisión sobre cuál es el siguiente nodo.

En el árbol hay una hoja por cada posible ordenación.

El árbol de decisión se dice que es válido para ordenar n valores si para cada permutación de los n valores hay un camino desde la raíz a una hoja que ordena la permutación. Debe tener al menos $n!$ hojas.

El número de comparaciones en el caso peor hechas por un árbol de decisiones es igual a su profundidad.

Encontrar una cota inferior de la profundidad de un árbol binario que contiene $n!$ hojas.

Si m es el número de hojas de un árbol binario y d es su profundidad, entonces se cumple

$$d \geq \lceil \log m \rceil.$$

Demostramos primero que para un árbol binario no vacío con m hojas y profundidad d se cumple $m \leq 2^d$, por inducción sobre d.

Base: Cuando $d = 0$, solo existe un nodo que es una hoja, y $1 \leq 2^0$.

Paso inductivo: Suponemos cierto que para todo árbol binario con m hojas y profundidad d se cumple $m \leq 2^d$, y probémoslo para $d + 1$. Tenemos que probar que $m' \leq 2^{d+1}$, donde m' es el número de hojas. Si borramos todas las hojas obtemos un árbol de profundidad d cuyas hojas son los padres de las hojas del árbol original. Si m es el número de estos padres, entonces por hipótesis de inducción,

$$m \leq 2^d.$$

Ya que cada padre puede tener como mucho dos hijos,

$$m' \leq 2m.$$

Combinando estas dos desigualdades, obtenemos

$$m' \leq 2m \leq 2^{d+1}.$$

Tomando ahora logaritmos, obtenemos $d \geq \log m$ y como d es entero, esto implica $d \geq \lceil \log m \rceil$.

De todo lo anterior se deduce que cualquier algoritmo que ordene n valores distintos utilizando exclusivamente comparaciones, debe realizar en el caso peor al menos $\lceil \log(n!) \rceil$ comparaciones.

¿Cómo de grande es $\log(n!)$?

$$\begin{aligned}
\log(n!) &= \log[n(n-1)(n-2)\cdots(2)1] \\
&= \sum_{i=2}^{n} \log i \\
&\geq \int_1^n \log x dx = \frac{1}{\ln 2}\Big[x \ln x - x\Big]_{x=1}^{x=n} \\
&= \frac{1}{\ln 2}(n \ln n - n + 1) \\
&\geq n \log n - 1{,}45n.
\end{aligned}$$

Selección

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

En particular, encontrar la mediana consiste en encontrar el $\lceil \frac{n}{2} \rceil$-ésimo elemento, es decir, el elemento que ocupa la posición $(n+1)$ div 2 del vector $V[1..n]$ cuando este se ordena.

Una primera idea para resolver este problema consiste en ordenar el vector V y tomar $V[k]$. Esto tiene una complejidad $O(n \log n)$. ¿Lo podemos hacer mejor?

Otra posibilidad es utilizar el algoritmo `partición`.

- Si el índice p, donde se coloca el pivote, es igual a k, el elemento $V[p]$ es el k-ésimo.
- Si $k < p$, podemos pasar a buscar el k-ésimo en $V[c..p-1]$, ya que estos elementos son menores o iguales a $V[p]$ y $V[p]$ es el p-ésimo.
- Si $k > p$, buscamos el $(k-p)$-ésimo en $V[p+1..f]$, ya que estos elementos son mayores o iguales que el p-ésimo, y hemos quitado p elementos por la izquierda.


```
{ 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
```

El caso peor del algoritmo tiene lugar cuando el pivote queda siempre en un extremo del subvector correspondiente: $\Theta(n^2)$.

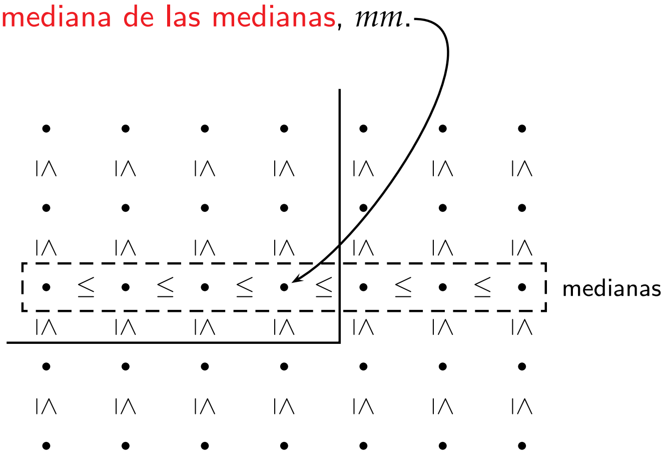
El coste en el caso medio está en $\Theta(n)$.

Selección: mediana de las medianas

La forma de mejorar este algoritmo es asegurarnos de que el pivote elegido no va a quedar en un extremo.

La mejor elección sería tomar como pivote la *mediana* del subvector, pero calcular la mediana es precisamente un caso particular del problema de selección, cuando $k = (c+f)$ div 2.

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



Utilizaremos 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 la mediana de las medianas, 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 podemos calcular para elementos mayores o iguales y estrictamente menores.

Por tanto, $\frac{7n+12}{10}$ es cota superior del número de elementos con los que haremos las llamadas recursivas.

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

1. calcular la mediana de cada grupo de 5 elementos. En total n div 5 medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero.
2. calcular la mediana de las medianas, mm, con una llamada recursiva a `selección2` con n div 5 elementos.
3. llamar a `partición2`(V, c, f, mm, i, j), utilizando como pivote mm.
4. hacer una distinción de casos similar a la de `selección1`:

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

   donde las llamadas recursivas se realizan con $\frac{7n+12}{10}$ elementos como mucho.

Selección: mediana de las medianas, coste

El tiempo requerido por `selección2`, $T(n)$, es lineal en el caso peor.
Utilizamos inducción constructiva para probar que

$$\exists c \text{ tal que } T(n) \leq cn \quad \forall n \geq 1.$$

Base: Tenemos que probar que $\forall n : 1 \leq n \leq n_0 : T(n) \leq cn$, donde n_0 es el valor que separa el caso básico del recursivo (al menos, $n_0 \geq 5$).
Hay un número finito de condiciones, por lo que la primera restricción sobre c es

$$c \geq \text{máx}\left\{\frac{T(n)}{n} \mid 1 \leq n \leq n_0\right\}.$$

Caso recursivo: Para $n > n_0 \geq 5$, tenemos

$$\begin{array}{lll} T(n) & \leq & dn + T(n \text{ div } 5) + \text{máx}\left\{T(m) \mid m \leq \frac{7n+12}{10}\right\} \\ & \leq^{h.i.} & dn + c\frac{n}{5} + \text{máx}\left\{cm \mid m \leq \frac{7n+12}{10}\right\} \\ & = & dn + c\frac{n}{5} + c\frac{7n+12}{10} \\ & = & \frac{9cn}{10} + dn + \frac{6c}{5} \\ & = & cn - \left(\frac{c}{10} - d - \frac{6c}{5n}\right) n \end{array}$$

Se sigue que $T(n) \leq cn$ siempre y cuando

$$\begin{array}{rcl} \left(\frac{c}{10} - d - \frac{6c}{5n}\right) & \geq & 0 \\ \left(1 - \frac{12}{n}\right) c & \geq & 10d \end{array}$$

Esto es posible siempre y cuando $n \geq 13$, en cuyo caso

$$c \geq \frac{10d}{1 - \frac{12}{n}}.$$

Teniendo en cuenta que $n \geq n_0$, cualquier elección de $n_0 \geq 12$ será aceptable, siempre y cuando c se seleccione en consecuencia. Por ejemplo, el paso de inducción es correcto si tomamos $n_0 = 12$ y $c \geq 130d$.
Reuniendo las restricciones sobre c, y tomando $n_0 = 12$, basta tomar

$$c = \text{máx}(130d, \text{máx}\{T(m) \mid 1 \leq m \leq 12\}).$$

Selección: mediana de las medianas, algoritmo

$\{ c \leq k \leq f \}$
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-inserción`(V, c, f) ; *k-ésimo* $:= V[k]$
  si no
    $s := t \text{ div } 5$
    para $l = 1$ **hasta** s **hacer**
      `ordenar-inserción`$(V, c + 5 * (l - 1), c + 5 * l - 1)$
      $pm := c + 5 * (l - 1) + 5 \text{ div } 2$
      $\langle V[c + l - 1], V[pm] \rangle := \langle V[pm], V[c + l - 1] \rangle$
    fpara
    `selección2`$(V, c, c + s - 1, c + (s - 1) \text{ div } 2, mm)$ { mediana de las medianas }
    `partición2`(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

Determinación del umbral

Divide y vencerás hace una distinción de casos

casos

$n \leq n_0 \;\longrightarrow\;$ subalgoritmo básico
□ $n > n_0 \;\longrightarrow\;$ 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 concreto.

El ideal sería que existiera un *umbral óptimo* n_0 tal que
$n \leq n_0 \;\longrightarrow\;$ es tan rápido llamar al subalgoritmo básico como dividir
$n > n_0 \;\longrightarrow\;$ más rápido dividir.

Pero este umbral óptimo no siempre existe.

¿Por qué es interesante determinar el umbral? No afecta al orden del tiempo de ejecución, pero sí a la constante multiplicativa.

Veamos un ejemplo de determinación del umbral. Comparamos mergesort con inserción, intentando optimizar el caso peor.

Complejidad de mergesort

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1$$

Supongamos que necesita $32n\mu s$ para dividir y componer (cálculo de la mitad, operaciones con la pila de llamadas recursivas, etc.)

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 32n\ \mu s$$

Si suponemos n potencia de 2 y $T(1) = 0\ \mu s$ (no hace prácticamente nada), tenemos

$$T(n) = 32n \log n\ \mu s$$

Supongamos que inserción tarda

$$\frac{n(n-1)}{2}\ \mu s$$

Parece que lo que tenemos que hacer es encontrar un n tal que

$$\frac{n(n-1)}{2} < 32n \log n$$

Esto ocurre si $n < 257$. Pero es INCORRECTO, no es el umbral óptimo. Hemos obtenido que es mejor usar inserción para $n < 257$ si mergesort divide hasta $n = 1$!!

Nuestro objetivo es saber hasta cuándo hay que dividir en mergesort.

$$T(n) = \begin{cases} \frac{n(n-1)}{2}\ \mu s & n \leq n_0 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + 32n\ \mu s & n > n_0 \end{cases}$$

¿Valor óptimo de n_0? Cuando

$$T\left(\left\lceil \frac{n_0}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n_0}{2} \right\rfloor\right) + 32n_0 = \frac{n_0(n_0-1)}{2}$$

Ya que $\lceil \frac{n_0}{2} \rceil$ y $\lfloor \frac{n_0}{2} \rfloor \leq n_0$,

$$T\left(\left\lceil \frac{n_0}{2} \right\rceil\right) = \frac{\lceil \frac{n_0}{2} \rceil \left(\lceil \frac{n_0}{2} \rceil - 1\right)}{2}$$

$$T\left(\left\lfloor \frac{n_0}{2} \right\rfloor\right) = \frac{\lfloor \frac{n_0}{2} \rfloor \left(\lfloor \frac{n_0}{2} \rfloor - 1\right)}{2}$$

Si n_0 es par, entonces $\lceil \frac{n_0}{2} \rceil = \lfloor \frac{n_0}{2} \rfloor = \frac{n_0}{2}$, y obtenemos $\boxed{n_0 = 128.}$

Si n_0 es impar, entonces $\lceil \frac{n_0}{2} \rceil = \frac{(n_0+1)}{2}$ y $\lfloor \frac{n_0}{2} \rfloor = \frac{(n_0-1)}{2}$, y obtenemos $\boxed{n_0 = 128{,}008.}$

Por lo que tomamos $n_0 = 128$.

Elemento mayoritario

Dado un vector $V[1..n]$ de n elementos (no necesariamente ordenables), se dice que un elemento x es mayoritario en V cuando el número de veces que x aparece en V es estrictamente mayor que $n/2$.

```
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
```

El tiempo de ejecución $T(n)$ de mayoritario1 se describe mediante la recurrencia

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

de donde se deduce que $T(n) \in \Theta(n \log n)$.

Si los elementos del vector se pueden ordenar, ¿se puede hacer mejor?

fun $\texttt{mayoritario2}(V[1..n]$ **de** *elem*$)$ **dev** $\langle existe$: *bool*$, mayor$: *elem*$\rangle$
var $W[1..n]$ **de** *elem*
  $W := V$
  $pm := (n+1) \text{ div } 2$ { posición de la mediana }
  $\texttt{selección2}(W, 1, n, pm, mediana)$
  $\langle existe, mayor \rangle := \langle \texttt{comprobar}(W, mediana, 1, n), mediana \rangle$
ffun

Los costes en tiempo y en espacio de `mayoritario2` están en $\Theta(n)$.

¿Se puede hacer lineal aunque los elementos no se pueden ordenar?

fun $\texttt{mayoritario3}(V[1..n]$ **de** *elem*$)$ **dev** $\langle existe$: *bool*$, mayor$: *elem*$\rangle$
  $candidato := V[1]$
  $contar := 1$ { 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
  $\langle existe, mayor \rangle := \langle \texttt{comprobar}(V, candidato, 1, n), candidato \rangle$
ffun

El coste de `mayoritario3` está en $\Theta(n)$.