```
    algoritmo select1(array A, intero k) → elem
1.      scegli un elemento x in A
2.      partiziona A rispetto ad x calcolando:
3.          A1 = { y ∈ A : y < x }
4.          A2 = { y ∈ A : y = x }
5.          A3 = { y ∈ A : y > x }
6.      quickSort(A1)
7.      quickSort(A3)
8.      return k-esimo elemento della concatenazione di A1, A2 ed A3
```

Figura 5.5 Adattamento dell'algoritmo `quickSort` al problema della selezione.

Dimostrazione. Basta osservare che le procedure `heapify`, `deleteMax` e `max` richiedono rispettivamente tempo $O(n)$, $O(\log n)$ ed $O(1)$, come dimostrato nel Paragrafo 4.3.1 del Capitolo 4. □

Per $k = O(n/\log n)$, il tempo di esecuzione dell'algoritmo `heapSelect` è pertanto $O(n)$. Questo algoritmo è dunque interessante per piccoli valori di k. Purtroppo, nel caso $k = \lceil n/2 \rceil$, questo approccio ha lo stesso costo di un ordinamento, e quindi non risolve ancora il problema del calcolo del mediano.

5.2 Selezione randomizzata

Per risolvere il problema della selezione per qualunque valore di k, ritorniamo all'idea originale: ricorriamo ad un algoritmo di ordinamento e poi restituiamo il k-esimo elemento dell'array ordinato. In particolare, consideriamo l'algoritmo `quickSort` visto nel Paragrafo 4.5 del Capitolo 4, che possiamo facilmente adattare al problema della selezione come mostrato in Figura 5.5. Si noti che, per efficienza e per semplicità di analisi, l'algoritmo `select1` partiziona l'array A in tre sottoinsiemi, anziché due come nel caso del `quickSort`. È comunque facile modificare la procedura `partition` vista nel Paragrafo 4.5 in modo da produrre i tre sottoinsiemi, lavorando comunque *in loco* ed in tempo $O(n)$.

Il tempo di esecuzione atteso dell'algoritmo `select1` è $O(n \log n)$, esattamente come per il `quickSort`. Osserviamo però che, se k è minore della lunghezza di A_1, `select1`(A, k) restituirà sempre un oggetto di A_1: in tal caso non importa se A_3 venga ordinato o meno, e la chiamata `quickSort`(A_3) risulta del tutto superflua. Allo stesso modo, se k è maggiore della somma delle lunghezze di A_1 ed A_2, `select`(A, k) restituirà sempre un oggetto di A_3, ed in questo caso è la chiamata `quickSort`(A_1) a risultare superflua. In entrambi i casi, possiamo risparmiare tempo facendo solo una delle due chiamate ricorsive. Otteniamo quindi `select2`, una variante dell'algoritmo `select1` mostrata in Figura 5.6.

Pur facendo meno chiamate al `quickSort`, il tempo di esecuzione atteso dell'algoritmo `select2` rimane comunque $O(n \log n)$: la chiamata al `quickSort`

```
    algoritmo select2(array A, intero k) → elem
1.      scegli un elemento x in A
2.      partiziona A rispetto ad x calcolando:
3.          A1 = { y ∈ A : y < x }
4.          A2 = { y ∈ A : y = x }
5.          A3 = { y ∈ A : y > x }
6.      if ( k ≤ |A1| ) then
7.          quickSort(A1)
8.          return k-esimo elemento di A1
9.      else if ( k > |A1| + |A2| ) then
10.             quickSort(A3)
11.             return (k − |A1| − |A2|)-esimo elemento di A3
12.     else return x
```

Figura 5.6 Una variante dell'algoritmo di selezione mostrato in Figura 5.5.

(su A_1 o su A_3) può infatti richiedere comunque tempo $O(n \log n)$. Per migliorare ulteriormente l'algoritmo, possiamo osservare che le chiamate al `quickSort` ci servono solo per ordinare un sottoarray e poi restituire un elemento in posizione specificata: ma questo è esattamente il problema da cui siamo partiti! Applichiamo quindi lo stesso ragionamento (ovvero, fare una sola chiamata ricorsiva) anche alle chiamate `quickSort`(A_1) e `quickSort`(A_3): possiamo ottenere questo effetto sostituendo le chiamate al `quickSort` con opportune chiamate ricorsive all'algoritmo di selezione stesso. L'algoritmo così ottenuto, che chiamiamo `quickSelect`, è mostrato in Figura 5.7.

Per calcolare il tempo di esecuzione dell'algoritmo `quickSelect`, osserviamo innazitutto che la chiamata ricorsiva è sempre fatta su un problema più piccolo. Se ogni chiamata ricorsiva dimezzasse la dimensione del problema otterremmo la

```
    algoritmo quickSelect(array A, intero k) → elem
1.      scegli un elemento x in A
2.      partiziona A rispetto ad x calcolando:
3.          A1 = { y ∈ A : y < x }
4.          A2 = { y ∈ A : y = x }
5.          A3 = { y ∈ A : y > x }
6.      if ( k ≤ |A1| )
7.      then return quickSelect(A1, k)
8.      else if ( k > |A1| + |A2| )
9.          then return quickSelect(A3, k − |A1| − |A2|)
10.         else return x
```

Figura 5.7 L'algoritmo randomizzato `quickSelect`.

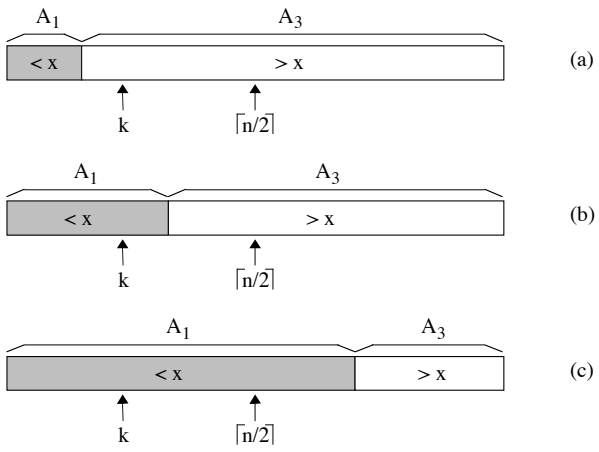

Figura 5.8 Il calcolo del mediano rappresenta il caso peggiore per il problema di selezione, poiché si ricorre sempre sull'array di dimensione maggiore.

relazione di ricorrenza

$$T(n) = O(n) + T(n/2)$$

che ha soluzione $T(n) = O(n)$, come si può facilmente vedere applicando il teorema fondamentale delle ricorrenze (Teorema 2.1 del Capitolo 2). Ovviamente, non sempre siamo così fortunati. Nel caso peggiore potremmo fare una chiamata ricorsiva ad un sottoproblema di $(n-1)$ elementi: infatti A_3 potrebbe essere vuoto ed A_2 potrebbe contenere un solo elemento, il perno x. In questo caso l'equazione di ricorrenza diventerebbe

$$T(n) = O(n) + T(n-1)$$

e, come sappiamo già dall'analisi del `quickSort`, avrebbe soluzione $T(n) = O(n^2)$. Quindi, nel caso peggiore, l'algoritmo `quickSelect` potrebbe essere molto inefficiente. Fortunatamente, il numero atteso di confronti è di molto inferiore.

Analisi probabilistica. Assumiamo ora che il perno x sia scelto in maniera casuale e calcoliamo il numero atteso di confronti. Per iniziare, poniamoci nel caso $k = \lceil n/2 \rceil$ e consideriamo l'insieme S_1 di elementi minori o uguali a x, e l'insieme S_2 di elementi maggiori o uguali a x: il mediano si trova sicuramente nel più grande di questi insiemi, che avrà dimensione $\geq n/2$. Quindi, se non restituiamo x come mediano (nel qual caso l'algoritmo termina), sicuramente eliminiamo $|A2| + \min\{|A1|, |A3|\}$ elementi dalla chiamata ricorsiva. Vediamo ora una delimitazione inferiore al numero di elementi eliminati nel caso in cui $k \neq \lceil n/2 \rceil$.

Lemma 5.2 *Il caso peggiore nell'esecuzione dell'algoritmo* `quickSelect` *si ha per* $k = \lceil n/2 \rceil$.

Dimostrazione. Se $k \neq \lceil n/2 \rceil$, le possibilità di ricorrere sul più piccolo dei due insiemi S_1 ed S_2 aumentano, come mostrato per $k < \lceil n/2 \rceil$ in Figura 5.8: per calcolare il mediano, la ricorsione ha luogo sempre sull'array di dimensione maggiore, ma per calcolare il k-esimo elemento, nel caso (b) si ricorre sull'array di dimensione minore. Il caso $k > \lceil n/2 \rceil$ è simmetrico. □

Grazie al Lemma 5.2, per analizzare `quickSelect` possiamo evitare di scrivere una relazione di ricorrenza randomizzata con due parametri, k ed n, scrivendo invece una relazione di ricorrenza con la sola variabile n. Come ci aspettiamo da un'analisi di valore atteso, otterremo una ricorrenza con una somma, su tutte le possibili scelte casuali, della probabilità di fare quella scelta moltiplicata per il tempo di esecuzione che deriva dalla scelta stessa (vedi Paragrafo 2.6).

Abbiamo detto che ad ogni passo eliminiamo sempre $|A2| + \min\{|A1|, |A3|\}$ elementi dalla chiamata ricorsiva. Questo valore è un qualsiasi numero tra 1 ed $n/2$, con la stessa probabilità: ciò deriva dal fatto che ogni elemento ha la stessa probabilità di essere scelto come perno, e quindi ogni partizione dell'array A è equiprobabile. Di conseguenza, la chiamata ricorsiva a `quickSelect` avviene in maniera equiprobabile su array che hanno una qualsiasi dimensione tra $n/2$ e $(n-1)$: più precisamente, la probabilità di ricorrere su un array di dimensione i è $1/(n/2) = 2/n$, per ogni $i \in [n/2, n-1]$. Ricordando dal Paragrafo 4.5 del Capitolo 4 che la partizione intorno al perno può essere effettuata con $(n-1)$ confronti ed usando la formula per l'analisi del tempo di esecuzione atteso data nel Paragrafo 2.6 del Capitolo 2, otteniamo la seguente relazione di ricorrenza per il numero atteso di confronti:

$$C(n) = n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} C(i) \tag{5.1}$$

Lemma 5.3 *La relazione di ricorrenza (5.1) ha soluzione $C(n) \leq 4n$.*

Dimostrazione. Dimostreremo che $C(n) \leq 4 \cdot n$ usando il metodo di sostituzione. Supponiamo, per ipotesi induttiva, che $C(i) \leq c \cdot n$, per ogni $i < n$ e per una opportuna costante c. Risulta quindi:

$$C(n) = n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} C(i) \leq n - 1 + \sum_{i=n/2}^{n-1} \frac{2}{n} c \cdot i = n - 1 + \frac{2c}{n} \sum_{i=n/2}^{n-1} i =$$

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

Usando la serie aritmetica $\sum_{k=1}^{n} k = n(n+1)/2$ otteniamo:

$$C(n) = n - 1 + \frac{2c}{n} \left(\frac{n^2}{2} - \frac{n^2}{8} - \frac{n}{4} \right) = \left(1 + \frac{3c}{4} \right) n - 1 - \frac{c}{2} \leq \left(1 + \frac{3c}{4} \right) n$$

```
    algoritmo sampleSelect(array A, interi n e k) → elem
1.      if ( |A| ≤ 10 ) then
2.          ordina A
3.          return k-esimo elemento di A
4.      scegli due parametri m e j "opportunamente"
5.      S ← sottoinsieme random di A contenente m elementi
6.      x ← sampleSelect(S, m, j)
7.      partiziona A rispetto ad x calcolando:
8.          A1 = { y ∈ A : y < x }
9.          A2 = { y ∈ A : y = x }
10.         A3 = { y ∈ A : y > x }
11.     if ( k ≤ |A1| ) then return sampleSelect(A1, |A1|, k)
12.     else if ( k > |A1| + |A2| ) then
13.             return sampleSelect(A3, |A3|, k − |A1| − |A2|)
14.     else return x
```

Figura 5.9 L'algoritmo `sampleSelect`.

L'induzione funziona quindi quando $(1 + 3c/4) \leq c$, ossia per $c \geq 4$. □

Riassumiamo quindi le prestazioni dell'algoritmo `quickSelect`:

Teorema 5.3 *Per ogni k, $1 \leq k \leq n$, l'algoritmo* `quickSelect` *randomizzato trova il k-esimo elemento di un array disordinato di lunghezza n eseguendo $O(n^2)$ confronti nel caso peggiore. Il numero atteso di confronti è invece $O(n)$.*

Selezione tramite campionamento. Floyd e Rivest [9] hanno osservato che una scelta più accorta del perno x rende l'algoritmo più efficiente, proponendo una tecnica simile a quella usata degli istituti demoscopici per effettuare indagini statistiche. Questa tecnica, analizzata in [4], si basa sull'idea che l'algoritmo è tanto più efficiente, quanto più il perno x è vicino alla k-esima posizione. Scegliere il perno come k-esimo elemento di un opportuno campione (invece di sceglierlo casualmente) può fornirci qualcosa di più vicino alla k-esima posizione. Lo pseudocodice dell'algoritmo di selezione basato su campionamento è mostrato in Figura 5.9. Le scelte di m e j influiscono naturalmente sul tempo di esecuzione, ma un'analisi dettagliata è al di là dello scopo di questo libro e rimandiamo il lettore interessato al già citato [4].

5.3 Selezione deterministica

Come abbiamo visto nel Paragrafo 5.2, l'algoritmo `quickSelect` è un algoritmo randomizzato per il calcolo del mediano e, in generale, per la selezione del k-esimo elemento: il valore atteso dei numeri di confronti è $O(n)$, ed anche in

```
    algoritmo select(array A, intero k) → elem
1.      if ( |A| ≤ 10 ) then
2.          ordina A
3.          return k-esimo elemento di A
4.      partiziona A in ⌈n/5⌉ sottoinsiemi S_i di (al più) 5 elementi ciascuno
5.      for i = 1 to ⌈n/5⌉ do m_i ← mediano di S_i
6.      M ← select({m_i : i ∈ [1, ⌈n/5⌉]}, ⌈n/10⌉)
7.      partiziona A rispetto ad M calcolando:
8.          A_1 = { y ∈ A : y < M }
9.          A_2 = { y ∈ A : y = M }
10.         A_3 = { y ∈ A : y > M }
11.     if ( k ≤ |A_1| ) then return select(A_1, k)
12.     else if ( k > |A_1| + |A_2| ) then return select(A_3, k − |A_1| − |A_2|)
13.     else return M
```

Figura 5.10 L'algoritmo deterministico `select`.

pratica `quickSelect` risulta molto veloce. Dal punto di vista teorico, però, sarebbe più soddisfacente avere anche un algoritmo lineare deterministico: Blum, Floyd, Pratt, Rivest e Tarjan [2] hanno risolto per primi questo problema in un modo molto elegante. Ricordiamo che l'algoritmo randomizzato `quickSelect` sceglie a caso un perno x, partiziona l'array in elementi più grandi e più piccoli di x, e richiama se stesso ricorsivamente su uno dei due sottoarray. L'algoritmo più veloce `sampleSelect` opera in modo simile, ma sceglie il perno in un modo leggermente più complicato, richiamando se stesso ricorsivamente su un "campione" statistico dell'input scelto casualmente. L'algoritmo deterministico che presenteremo in questo paragrafo segue la stessa idea, ovvero sceglie x tramite una chiamata ricorsiva, ma riesce a selezionare opportunamente il campione in modo deterministico.

Se potessimo scegliere il perno x come mediano di tutti i valori, ogni chiamata ricorsiva sarebbe su al più la metà dei valori: ma ovviamente, se sapessimo come scegliere il mediano, avremmo già risolto il nostro problema, senza necessità di ulteriori chiamate ricorsive. Il progetto sembra quindi troppo ambizioso. Rilassiamo le nostre pretese, e cerchiamo di selezionare come perno qualcosa di vicino al mediano, ad esempio, un valore che dista al più $n/4$ da esso. In tal caso, ogni chiamata ricorsiva sarebbe su una frazione dell'input al più grande $3n/4$, e questo sarebbe comunque sufficiente per ottenere un algoritmo lineare.

5.3.1 Mediano dei mediani

Come ottenere efficientemente un valore vicino al mediano? Come nel caso dell'algoritmo `sampleSelect`, invece di trovare il mediano dell'intero insieme, cerchiamo il mediano di un opportuno campione. E come scegliere il campione?

Ancora una volta usando i mediani! L'algoritmo, il cui pseudocodice è mostrato in Figura 5.10, può essere descritto informalmente come segue:

Passo 1. Raccogli gli elementi in gruppi di 5. Per semplicità, chiameremo l'i-esimo gruppo S_i, con $i \in [1, \lfloor n/5 \rfloor]$. Se n non è divisibile per 5, crea un ulteriore gruppo $S_{\lceil n/5 \rceil}$ contenente gli elementi rimanenti (al più quattro).

Passo 2. Trova il mediano m_i di ogni gruppo S_i.

Passo 3. Tramite una chiamata ricorsiva, trova il mediano M dei mediani m_i individuati nel Passo 2.

Passo 4. Usa M come perno e richiama l'algoritmo ricorsivamente sull'array opportuno, esattamente come in `quickSelect`.

Nello pseudocodice, per motivi di efficienza abbiamo fermato la ricorsione quando A è sufficientemente piccolo (al più 10 elementi). Osserviamo inoltre che la costante 5 nel Passo 1 dell'algoritmo (riga 4 in Figura 5.10) non è cruciale: come si evincerà dall'analisi, potremmo infatti usare senza problemi anche altre costanti ≥ 5, come ad esempio 7 o 9: rimandiamo all'Esercizio 5.2 per ulteriori considerazioni. La seguente proprietà dell'algoritmo `select` sarà poi utile in sede di analisi.

Proprietà 5.1 *Il calcolo dei mediani m_i delle quintuple S_i (riga 5 in Figura 5.10) può essere implementato eseguendo al più $6\lceil n/5 \rceil$ confronti.*

Il mediano di ogni gruppo di 5 elementi può essere infatti calcolato eseguendo al più 6 confronti, come il Problema 5.1 richiede di dimostrare. Il calcolo del mediano dei mediani M (riga 6 in Figura 5.10) è implementato tramite una chiamata ricorsiva su un insieme di $\lceil n/5 \rceil$ elementi, e richiede pertanto tempo $T(\lceil n/5 \rceil)$. La maggiore difficoltà nell'analisi sembra nel Passo 4: non sappiamo, infatti, quanto è grande l'insieme su cui si procede ricorsivamente, data la scelta del perno M effettuata dall'algoritmo. Per dire qualcosa sulla dimensione di questo insieme, ragioniamo in termini degli elementi scartati, cioè non inclusi nella chiamata ricorsiva.

Lemma 5.4 *La chiamata ricorsiva nel Passo 4 dell'algoritmo* `select` *viene effettuata su un insieme contenente al più $(7n/10 + 3)$ elementi.*

Dimostrazione. Ad ogni passo, dopo aver partizionato intorno al mediano dei mediani M, scartiamo sempre o tutti i valori più grandi o tutti i valori più piccoli di M (oltre a quelli uguali). Nel primo caso la chiamata ricorsiva del Passo 4 viene effettuata su A_1, nel secondo caso su A_3: supponiamo senza perdere di generalità di ricorrere su A_1, e quindi di scartare i valori di $A_3 \cup A_2$.

Tra gli $\lceil n/5 \rceil$ valori m_i, la metà sono maggiori o uguali ad M, essendo M per definizione il mediano degli m_i. Per le proprietà della parte intera superiore

$$\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil = \left\lceil \frac{n}{10} \right\rceil$$

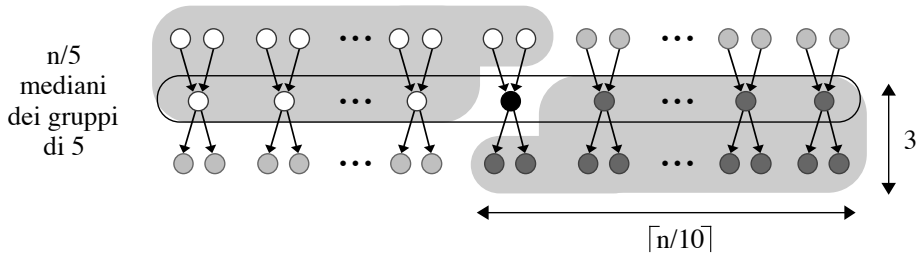

Figura 5.11 Dimostrazione visuale del Lemma 5.4.

elementi sono quindi maggiori o uguali ad M. Ricordiamo inoltre che m_i è il mediano dei 5 valori nel gruppo S_i, per ogni $i \in [1, \lfloor n/5 \rfloor]$: quindi, per ogni $m_i \geq M$ il cui gruppo contiene 5 elementi, altri due valori di S_i sono $\geq m_i \geq M$. Da quanto detto, almeno $\lceil n/10 \rceil - 1$ gruppi contengono almeno tre elementi maggiori o uguali ad M. Quindi:

$$|A_2 \cup A_3| \geq 3\left(\left\lceil \frac{n}{10} \right\rceil - 1\right) \geq \frac{3n}{10} - 3$$

elementi. Con un ragionamento analogo, possiamo dimostrare che

$$|A_1 \cup A_2| \geq \frac{3n}{10} - 3$$

La chiamata ricorsiva finale sarà pertanto su un array contenente al più $7n/10 + 3$ elementi. □

L'algoritmo `select`, per quanto dimostrato nel Lemma 5.4, ha la proprietà che desideravamo: ogni chiamata ricorsiva viene fatta su una frazione costante dei dati in ingresso, scelta in modo deterministico. Calcoliamo ora il numero di confronti effettuati.

Teorema 5.4 *L'algoritmo* `select` *esegue nel caso peggiore* $O(n)$ *confronti*.

Dimostrazione. Il numero $T(n)$ di confronti effettuati può essere trovato risolvendo la seguente relazione di ricorrenza

$$T(n) \leq \frac{11}{5}n + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 3\right) \tag{5.2}$$

che deriva dalle seguenti considerazioni.

- Per la Proprietà 5.1, il calcolo dei mediani m_i richiede al più $6\lceil n/5 \rceil \leq 6(n/5) + 1$ confronti.
- Il calcolo del mediano dei mediani è effettuato eseguendo una chiamata ricorsiva su un insieme contenente $\lceil n/5 \rceil$ elementi. Ciò richiede tempo $T(\lceil \frac{n}{5} \rceil)$.

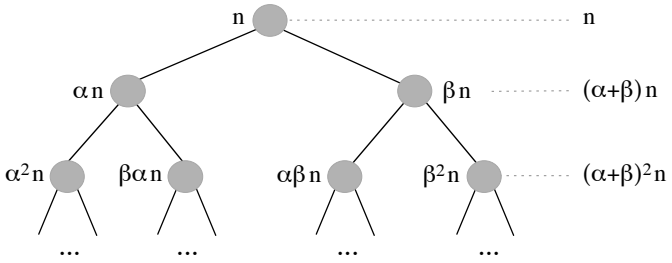

Figura 5.12 Albero della ricorsione corrispondente alla relazione di ricorrenza $T(n) \leq c \cdot n + T(\alpha\, n) + T(\beta\, n)$.

- Partizionare l'array A intorno al perno richiede al più $(n-1)$ confronti, come mostrato nel Paragrafo 4.5 del Capitolo 4.
- La chiamata ricorsiva che restituisce la soluzione viene eseguita su un insieme contenente al più $(\frac{7}{10}n + 3)$ elementi, come dimostrato nel Lemma 5.4.

Dimostriamo ora, usando il metodo di sostituzione introdotto nel Paragrafo 2.5.2 del Capitolo 2, che $T(n) \leq (c\, n - 4c)$ per una opportuna constante c. Nel nostro caso abbiamo:

$$T(n) \leq \frac{11}{5}n + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 3\right) \leq \frac{11}{5}n + T\left(\frac{n}{5} + 1\right) + T\left(\frac{7}{10}n + 3\right)$$

Per ipotesi induttiva risulta quindi

$$T(n) \leq \frac{11}{5}n + c\left(\frac{n}{5} + 1\right) - 4c + c\left(\frac{7n}{10} + 3\right) - 4c = n\left(\frac{11}{5} + \frac{9c}{10}\right) - 4c$$

Se vogliamo che questo sia al più $c \cdot n - 4c$, abbiamo bisogno che valga la disuguaglianza

$$n\left(\frac{11}{5} + \frac{9c}{10}\right) \leq c \cdot n$$

ovvero

$$\frac{11}{5} + \frac{9c}{10} \leq c$$

che implica $11/5 \leq c/10$ e quindi vale per $c \geq 22$. □

La dimostrazione del Teorema 5.4 mostra che l'algoritmo `select` esegue al più $22n$ confronti, ovvero richiede tempo lineare nel caso peggiore. Con tecniche più complesse, si può ridurre il numero dei confronti a $2.95n$ [7], che è meno del doppio dei confronti richiesti dalla selezione randomizzata, ma l'algoritmo è molto più complicato e meno efficiente in pratica.

Concludiamo questo paragrafo con alcune considerazioni sul tipo di relazione di ricorrenza che ci siamo trovati a risolvere per analizzare l'algoritmo `select`.

La relazione di ricorrenza (5.2) non ha infatti la forma adatta per poter utilizzare il teorema fondamentale delle ricorrenze (Teorema 2.1 nel Capitolo 2), ma sembra comunque essere generata da un algoritmo di tipo *divide et impera* che divide il problema originario in due sottoproblemi in maniera non bilanciata. Tralasciando gli addendi costanti ed usando costanti moltiplicative generiche, possiamo considerare la ricorrenza (5.2) come una caso particolare della seguente relazione:

$$T(n) \leq c \cdot n + T(\alpha\, n) + T(\beta\, n) \tag{5.3}$$

dove $\alpha + \beta < 1$. Mostreremo ora, con un procedimento basato sull'analisi dell'albero della ricorsione, come risolvere questa ricorrenza. Il fatto che la somma delle parti, $\alpha\, n + \beta\, n$, sia strettamente minore della dimensione originaria, n, giocherà un ruolo determinante nell'analisi.

Teorema 5.5 *La relazione di ricorrenza (5.3), con $\alpha + \beta < 1$, ha soluzione*

$$T(n) \leq \frac{c}{1 - \alpha - \beta}\, n$$

Dimostrazione. Consideriamo l'albero delle chiamate ricorsive mostrato in Figura 5.12, indicando le dimensioni dei problemi di ogni chiamata ricorsiva, ed analizziamo la dimensione totale dei problemi ad ogni livello dell'albero. Nel nostro caso al primo livello abbiamo $(\alpha + \beta)\, n$, al secondo $(\alpha + \beta)^2\, n$, e così via. Il tempo di esecuzione totale è quindi dato da:

$$\begin{aligned} T(n) &\leq \left(c \cdot n + c\,(\alpha + \beta)\, n + c\,(\alpha + \beta)^2\, n + ...\right) = \\ &= c \cdot n \left(1 + \ (\alpha + \beta) \ + \ (\alpha + \beta)^2 \ + \ (\alpha + \beta)^3 \ + ...\right) \leq \\ &\leq c \cdot n \sum_{i=0}^{\infty} (\alpha + \beta)^i = c \cdot n \frac{1}{1 - (\alpha + \beta)} = \frac{c}{1 - \alpha - \beta}\, n \end{aligned}$$

Osserviamo che la precedente uguaglianza è una semplice applicazione della serie geometrica riportata in Appendice:

$$\sum_{i=0}^{k} d^i = \frac{d^{k+1} - 1}{d - 1}$$

con $|d| = \alpha + \beta < 1$ e per $k \to \infty$. □

5.4 Esercizi

Esercizio 5.1 Progettare un algoritmo che, dato un insieme di n interi distinti (con $n > 2$), restituisca in tempo $O(1)$ un elemento dell'insieme che non è né il minimo né il massimo.