Select

Realizzare un algoritmo `Select(A,k)` che dato un array non ordinato di interi `A[1,n]` ed un indice $1 \leq k \leq n$, restituisce l'elemento dell'array che sarebbe in posizione k nell'array ordinato (o meglio, uno degli elementi che potrebbe essere in posizione k, dato che se ci sono ripetizioni, l'elemento non è univocamente determinato). Ad esempio, se $A = [1, 4, 2, 8, 5]$ si vuole `Select(A,3)= 4` mentre `Select(A,5)= 8`.

Proporre soluzioni di con complessità:

1. $O(n \log n)$
2. $O(n + k \log n)$
3. $O(n \log k)$

Soluzione

1. Si può ottenere una soluzione di tempo $O(n \log n)$ banalmente, ordinando l'array e quindi ritornando l'elemento di posizione k. Ovvero

```
Select(A, k)
   n = A.length
   Mergesort(A,1,n)
   return A[k]
```

2. Per ottenere una soluzione di costo $O(n + k \log n)$ posso trasformare A in un min-heap ed estrarre k volte l'elemento più piccolo, ovvero:

```
Select(A, k)
   BuildMinHeap(A)
   for i=1 to k-1
      ExtractMin(A)
   return A[1]
```

   L'invariante di ciclo, utile a dimostrare la correttezza del procedimento, è che `A[1,n-i+1]` contiene gli $n - i + 1$ elementi più grandi di A.

   Il costo di `BuildMinHeap(A)` è $\Theta(n)$, mentre `ExtractMin(A)` costa $O(\log n)$ e viene eseguita k volte, quindi, in totale $O(n + k \log n)$.

3. Per ottenere una soluzione con complessità $O(n \log k)$ si può procedere nel modo seguente:

   - Si costruisce un max-heap che contenga i primi k elementi di A. Per questo si utilizza `BuildMaxHeap(A,k)` dove il parametro addizionale k indica il numero di elementi di A che vanno inclusi nel max-heap (si opera come se la dimensione di A fosse k).
   - Ognuno dei rimanenti $n - k$ elementi di A è confrontato con il massimo dei k elementu nel max-heap correnti, e se più piccolo, lo sostituisce.
   - Alla fine, il max-heap conterrà i k elementi più piccoli di A, il cui massimo $A[1]$ è l'elemento che si troverebbe in posizione k nell'array ordinato, ovvero l'elemento cercato.

   Si ottiene dunque

```
Select(A, k)
   BuildMaxHeap(A,k)      // costo Θ(k)

   for i=k+1 to n         // costo O((n−k)log k) quindi O(n log k)
      if A[i] < A[1]
         A[i] <-> A[1]    // scambia A[i] e A[1]
         MaxHeapify(A,1)
   return A[k]
```

L'invariante di ciclo è

- $A[1,k]$ max-heap
- $A[k+1, i-1] \geq A[1,k]$, ovvero il frammento esplorato della parte rimanente di array, contiene elementi tutti tutti maggiori o uguali a quelli in $A[1,k]$.

Quindi, quando il ciclo termina $A[k+1,n] \geq A[1,k]$, ovvero $A[1,k]$ contiene i k elementi più piccoli dell'array. Pertanto il massimo, in $A[1]$, è l'elemento in posizione k nell'array ordinato, e quindi è l'elemento cercato.

Il costo complessivo è $O(k + n \log k) = O(n \log k)$, dato che il termine $n \log k$ domina k.

4. *Selezione in tempo medio lineare: QuickSelect*
   L'idea è qui quella di sfruttare la procedura di partizionamento vista per il Quicksort. Se la posizione q del pivot è esattamente l'indice k cercato abbiamo concluso. Altrimenti possiamo ricorrere su una delle due parti a seconda che k sia minore o maggiore di q. Esplicitamente:

```
QuickSelect(A, p, r, k)
   q = Partition(A,p,r)                    // costo  an + b

   if (q == k)
      return A[q]
   else if (k < q)
      return QuickSelect(A, p, q-1, k)
   else
      return QuickSelect(A, q+1, r, k)
```

   È facile vedere che, come nel Quicksort, nel caso peggiore, ovvero quando si ottenga sistematicamente una delle due partizioni vuote, si ottiene la ricorrenza

$$T(n) = T(n-1) + an + b$$

   con soluzione $T(n) = \Theta(n^2)$

   Quando invece sistematicamente la partizione sia perfettamente bilanciata, si ottiene la ricorrenza

$$T(n) = T(n/2) + an + b$$

   che, utilizzando il Master Theorem, ha come soluzione $T(n) = \Theta(n)$.

   Si può verificare che anche il caso medio porta a complessità lineare. In un approccio semplificato, possiamo assumere che con uguale probabilità, si

termini o si ricorra su di una partizione di dimensione 1, 2, ..., $n-1$. Il valore medio della complessità può dunque essere espresso dalla ricorrenza:

$$T(n) = \frac{1}{n}(\sum_{i=0}^{n-1} T(i)) + an + b$$

E si può verificare per sostituzione che $T(n) = O(n)$, ovvero che $T(n) \leq cn$ asintoticamente per un'opportuna costante $c > 0$.

Infatti si ha

$$\begin{aligned}
T(n) &= \frac{1}{n}(\sum_{i=0}^{n-1} T(i)) + an + b \\
&\leq \frac{1}{n}(\sum_{i=0}^{n-1} ci) + an + b \qquad \text{[ipotesi induttiva]} \\
&= c\frac{1}{n}\frac{n(n-1)}{2} + an + b \\
&= c\frac{(n-1)}{2} + an + b \\
&= \frac{cn}{2} - \frac{c}{2} + an + b \\
&= (\frac{c}{2} + a)n - \frac{c}{2} + b \\
&\leq cn
\end{aligned}$$

dove l'ultima disuguaglianza vale, asintoticamente, se $\frac{c}{2} + a < c$ e quindi per $c > 2a$. Questo conclude la prova.

Così come nel caso del QuickSort, il caso medio può essere indotto anche quando l'input non sia uniformemente distribuito, con un scelta randomizzata del pivot. La presenza di duplicati richiederà di utilizzare una tripartizione.

5. *Selezione in tempo lineare*
   Si veda il libro di testo.