Selezione

Luca Tagliavini

April 7-8, 2021

Contents

0.1 Il problema della selezione

Il problema della *selezione del k-esimo* e' il problema che rappresenta l'ordinamento e la selezione dell'elemento k dopo aver ordinato l'array in cui esso e' contenuto. La domanda che ci facciamo e', *si ha un modo migliore per selezionare l'elemento anzi che ordinare l'intero array*?

Avremo dunque un array $A[1..n]$ di valori *distinti*, un valore $1 \leq k \leq n$ e dovremo trovare l'elemento che e' maggiore esattamente di $k-1$ elementi. Esempio trovare *il mediano*: valore che occuperebbe la posizione $n/2$ se l'array fosse ordinato.

0.1.1 Caso particolare: selezione del minimo

Ricerca del k-esimo con $k = 1$:

```
algorithm minimum(T[1..n] A) -> T
  min := A[1]
  for i := 2 to n do
    if A[i] < min then
      min = A[i]
    endif
  endfor
  return min
endalgorithm
```

Costo: $O(n-1) = \Theta(n)$

0.1.2 Caso particolare: selezione del *secondo* minimo

Ricerca del k-esimo con $k = 2$:

```
algorithm second_minimum(T[1..n] A) -> T
  min1 := A[1]
  min2 := A[2]
  for i := 3 to n do
    if A[i] < min2 then
      min2 = A[i]
      if min2 < min1 then
        swap(min1, min2)
      endif
    endif
  endfor
  return min
endalgorithm
```

Costo: $O(2(n-2)+1)$

0.2 Generalizzazione: ricerca (non ottimale) del k-esimo

Questo algoritmo si ispira fortemente al Selection Sort, in quanto cerca k volte il minimo e lo pone in cima all'array, per poi ritornare $A[k]$ che per quello che abbiamo detto poc'anzi sara' il k-esimo valore minimo

```
algorithm k_minimum(T[1..n] A, int k) -> T
  for i := 1 to k do
    minIndex := i
    minValue := A[i]
    for j := i+1 to n do
      if A[i] < minValue then
        minIndex := j
        minValue := A[j]
      endif
    endfor
    swap(A[i], A[minIndex])
  endfor
  return A[k]
endalgorithm
```

Costo: $O(kn)$

0.3 Soluzione al problema del k-esimo con heap

Si puo' avere una soluzione simile all'algoritmo dell'Heap Sort, che funziona tramite una struttura arborescente Heap che posizione in testa all'array il valore minore (in tempo lineare) e ne rimuove via via k volte il piu' piccolo valore per ottenere il minimo di nostro interesse (con tempositiche dell'ordine $\log_2 n$). Eccone una implementazione:

```
algorithm k_minimum(T[1..n] A, int k) -> T
  min_heapify(A)   // O(n)
  for i := 1 to k-1 do
    delete_min(A) // O(log n)
  endfor
  return find_min(A) // O(1)
endalgorithm
```

Costo: $O(n + k \log_2 n)$

Questa versione ispirata all'Heap Sort e' migliore di quella precedente basata sul Selection Sort in quanto, per valori di k molto grandi (vicini a n) questo e' migliore.

In particolare se interessati a un $k \leq O(\frac{n}{\log_2 n})) = O(\frac{n}{\log_2 n})$ avremo un tempo $O(n + \frac{n}{\log_2 n} \log_2 n) = O(2n) = O(n)$. Ha chiaramente un costo migliore dell'altro (che sarebbe stato $O(\frac{n^2}{\log_2 n})$.

Analogamente, per casi scomodi come la ricerca del mediano dove si ha $k = n/2$. Facendo i conti avremmo un costo nell'ordine di $O(n \log_2 n)$, stesso costo di un algoritmo di ordinamento completo.

0.4 Metodo ancora migliore

I spirandoci al Quick Sort possiamo tuttavia trovare una soluzione divide-et-impera che ci portera' ad avere performance migliori delle soluzioni precedenti:

```
algorithm select1(T[1..n] A, int k) -> T
  x := un elemento in A
  A1 := { y in A | y < x }
  A2 := { y in A | y = x }
  A3 := { y in A | y > x }
  quicksort(A1);
  quicksort(A3);
  ritorna merge(A1, A2, A3)[k];
endalgorithm
```

Ora possiamo osservare che, visto che il nostro obbiettivo non e' ordinare l'intero array ma sono la parte che conterra' il k-esimo, possiamo vedere dove andra' a cadere k guardando le grandezze delle partizioni e ordinderemo solo codeste partizioni. In questo modo avremo un costo vantaggioso. Eccone una implementazione:

```
algorithm quick_select(T[1..n] A, int k) -> T
  x := un elemento in A
  A1 := { y in A | y < x }
  A2 := { y in A | y = x }
  A3 := { y in A | y > x }
  if k <= len(A1) then
    return quick_select(A1, k)
  elif k <= len(A1) + len(A2) then
    return x
  else
    return quick_select(A3, k - len(A1) - len(A2))
  endif
endalgorithm
```

Calcolo del costo:

- caso ottimo (ogni chiamata dimezziamo): $T(n) = T(n/2) + n = \Theta(n)$ caso 3 del Master Theorem. (NOTA: $+n$ e' dovuto alla divisione in 3 sottoarray)
- caso pessimo (rimuoviamo sempre solo il pivot): $T(n) = T(n-1) + n = \Theta(n^2)$
  dimostrabile come nel Quick Sort
- caso medio: Ad ogni iterazione elimino $len(A2) + min(len(A1), len(A3)$ elementi, insomma assumo sempre che andro' a cercare nel sottoinsieme

piu' grande. Il numero di elementi *scartati* sara' dunque $1 \leq \cdot \leq n/2$.
La propobabilita' di ricadere su un segmento di lunghezza i e' di $\approx \frac{1}{n/2} = 2/n$ per $i = n/2, n/(2+1), \ldots, n-1$

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(n') + n & \text{con } \frac{n}{2} \leq n' < n \end{cases}$$

Tramite la tecnica della sostituzione proviamo il seguente teorema: Il costo $T(n) \leq 4n$, quindi $T(n) = O(n)$.