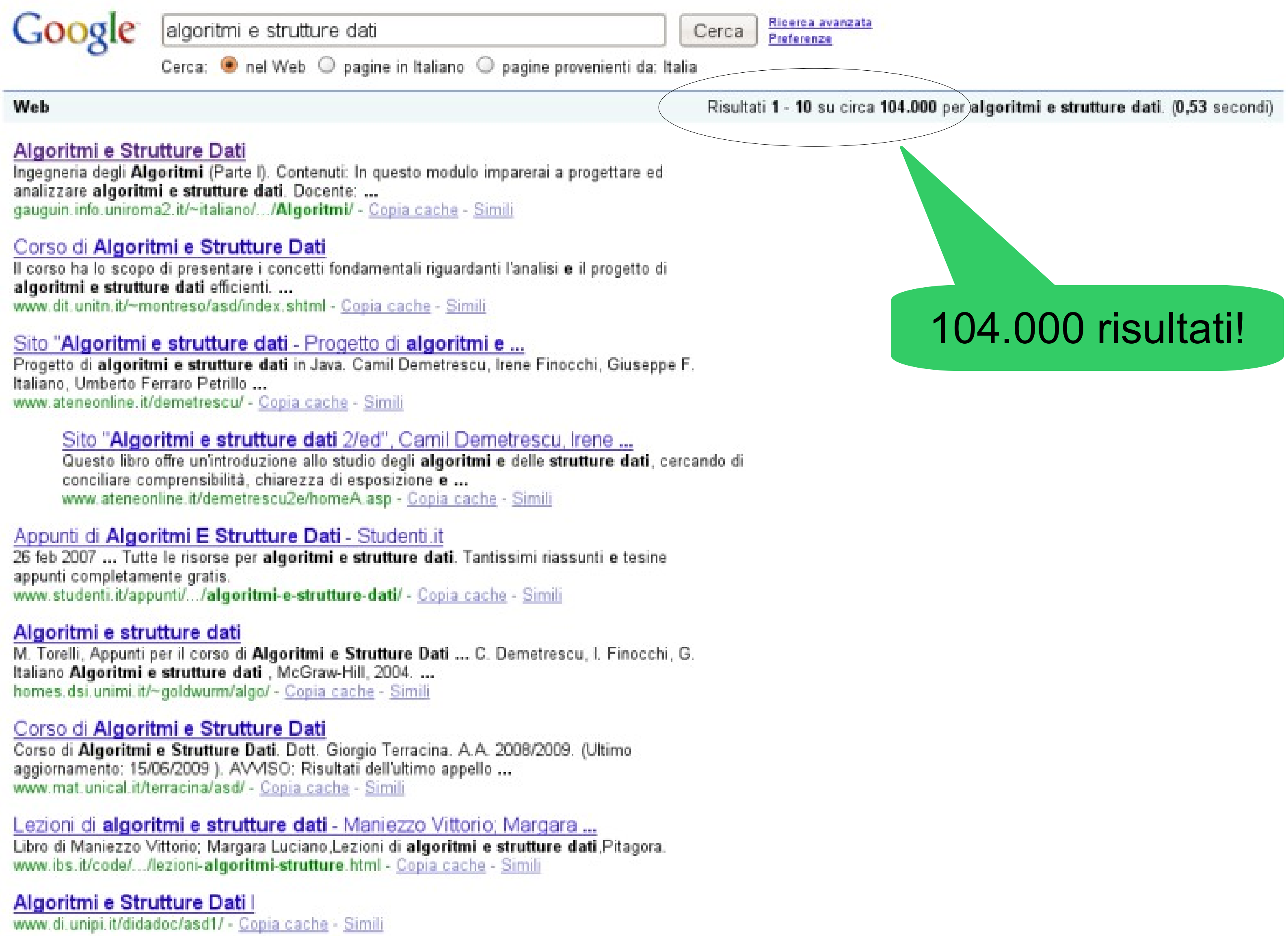
Selezione del k-esimo minimo

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Selezione del k-esimo minimo

- Consideriamo il seguente problema:
  - Selezione del k-esimo minimo: dato un array A[1..n] di valori distinti e un valore $1 \leq k \leq n$, trovare l'elemento che è maggiore di esattamente k-1 elementi
- Caso particolare:
  - Mediano: il valore che occuperebbe la posizione (n/2) se l'array fosse ordinato

Selezione del k-esimo minimo: a che serve?



Selezione del k-esimo minimo: a che serve?

- I motori di ricerca producono molti risultati a fronte di una singola query
- I risultati vengono mostrati in pagine, in ordine decrescente di rilevanza
  - Nella prima pagina i risultati più rilevanti
  - Nella seconda quelli meno, e così via
- È inutile ordinare tutti i risultati in base alla rilevanza
  - Quanti vanno frequentemente oltre la quarta pagina di risultati della ricerca?
- È quindi utile selezionare i primi k risultati, e via via i successivi, se l'utente seleziona le altre pagine

Selezione: casi particolari Ricerca del minimo

```
algorithm minimum(array A[1..n]) → elem
  min := A[1];
  for i := 2 to n do
    if (A[i] < min) then
      min = A[i];
    endif
  endfor
  return min;
```

- $T(n) = n-1 = \Theta(n)$ confronti

Selezione: casi particolari

- Ricerca del secondo minimo
  - Trovare il secondo elemento più piccolo dell'array A[]
  - Costo: 2n-3 confronti nel caso peggiore (il caso peggiore si verifica quando i valori sono in ordine decrescente)

```
algorithm minimum2(array A[1..n]) → elem
  min1 := A[1];
  min2 := A[2];
  if (min2 < min1) then
     swap(min1, min2);
  endif
  for i:= 3 to n do
    if (A[i] < min2) then
      min2 = A[i];
      if (min2 < min1) then
        swap(min1, min2);
      endif
    endif
  return min2;
```

Selezione del k-esimo minimo

```
algorithm select(array A[1..n], int k) → elem
   for i:=1 to k do
      minIndex := i;
      minValue := A[i];
      for j:=i+1 to n do
          if (A[j] < minValue) then
              minIndex := j;
              minValue := A[j];
          endif
      endfor
      swap A[i] and A[minIndex];
    endfor
    return A[k];
```

- In sostanza un Selection Sort incompleto
  - Si ferma al k-esimo elemento
- Costo $\Theta(kn)$

Selezione per piccoli valori di k

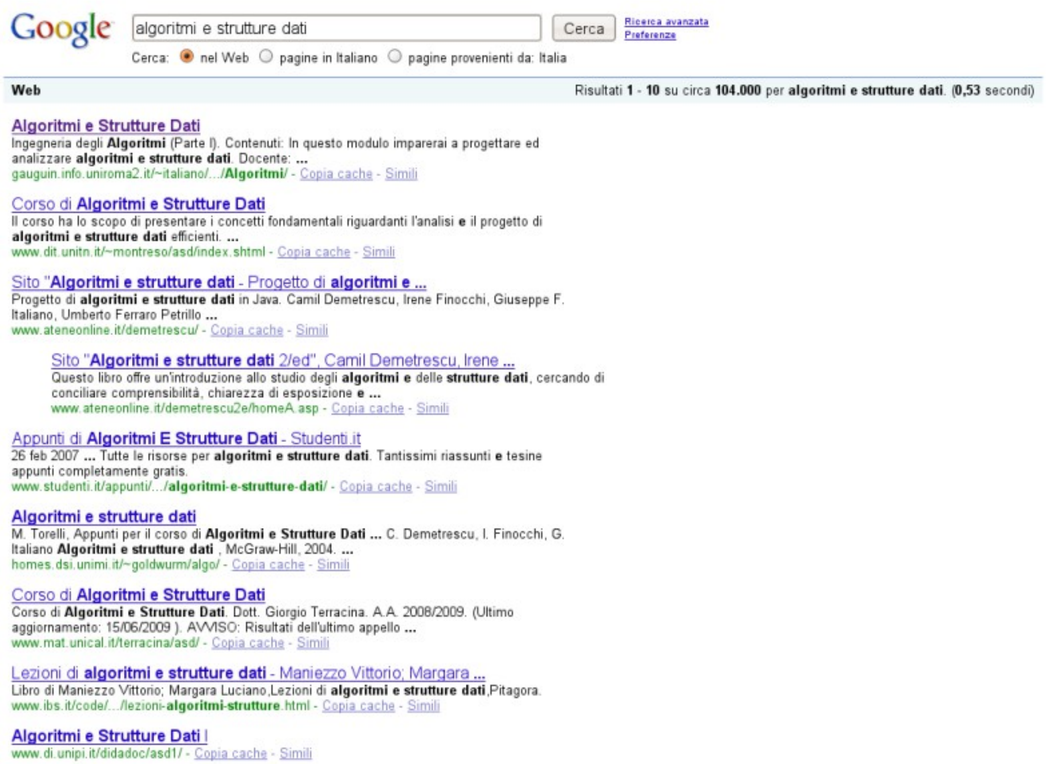
- Costruisco un min-heap a partire dai valori
  - Costo O(n)
- Estraggo per *k-1* volte il minimo
  - Costo O(k log n)
- Il k-esimo minimo è l'elemento minimo che rimane
  - Costo complessivo: $O(n + k \log n)$

```
algorithm heapselect(array A[1..n], int k) → elem
  min-heapify(A);
  for i := 1 to k-1 do
      deleteMin(A);
  endfor
  return findMin(A);
```

Questa funzione costruisce un min-heap

Esempio

- Estrarre i primi k elementi da una query che ha fornito n match costa O(n + k log n) = O(n) se k è O(n/log n)
  - Esempio: k=10, n=104000



Esempio

- Le cose vanno meno bene se *k* è del medesimo ordine di grandezza di *n*
- Esempio: voglio calcolare il valore mediano
  - Cioè il valore che occuperebbe la posizione centrale se l'array fosse ordinato
- In questo caso $k = n/2$, e per valori sufficientemente grandi di *n* il costo è

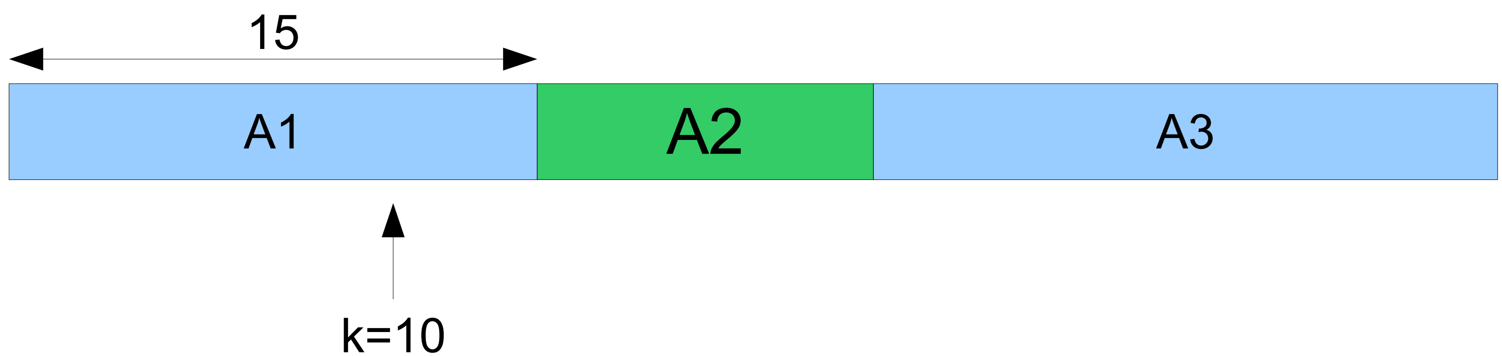
$$O(n + k \log n) = O(n + (n/2) \log n) = O(n \log n)$$

Adattamento di quicksort al problema della selezione

```
algorithm select1( array A[1..n], int k ) -> elem
   scegli un elemento x 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 il k-esimo elemento della concatenazione di A1, A2, A3
```

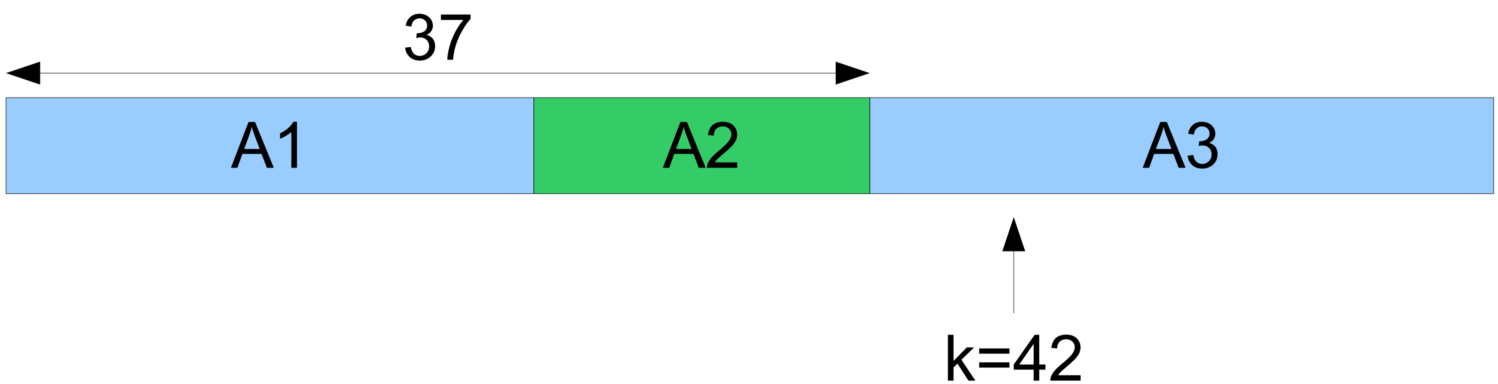
- Idea
  - Approccio divide-et-impera simile al QuickSort...
  - ...però essendo un problema di ricerca, non è necessario cercare in tutte le partizioni, basta cercare in una sola

Adattamento di quicksort al problema della selezione



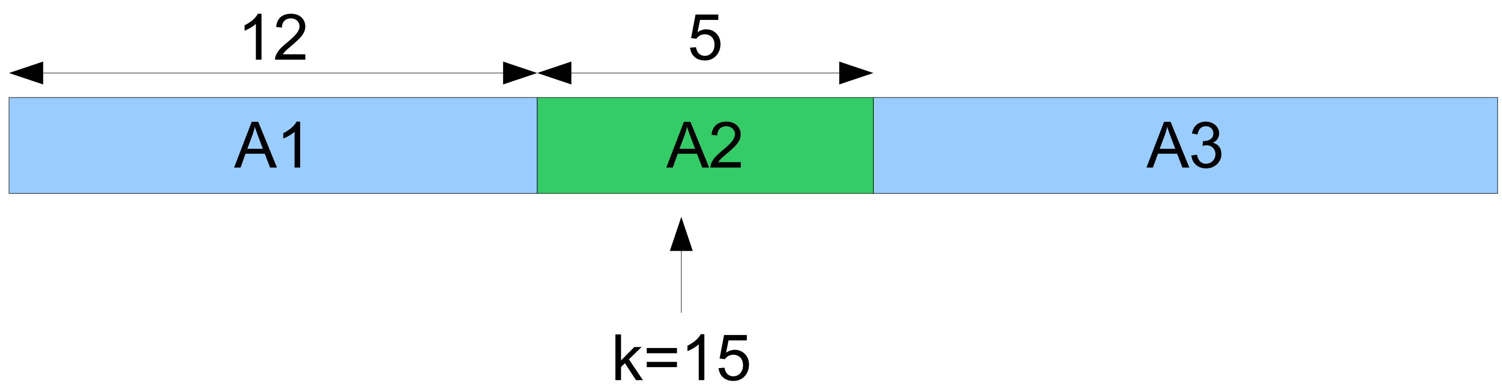
- In realtà non serve considerare tutto il vettore!
  - Supponiamo di cercare il 10mo minimo;
  - Supponiamo che A1 abbia 15 elementi
  - Il valore cercato sicuramente sta in A1

Adattamento di quicksort al problema della selezione



- In realtà non serve considerare tutto il vettore!
  - Supponiamo di cercare il 42mo minimo;
  - Supponiamo che A1 e A2 abbiano 37 elementi
  - Il valore cercato è il (42-37)=5 di A3

Adattamento di quicksort al problema della selezione

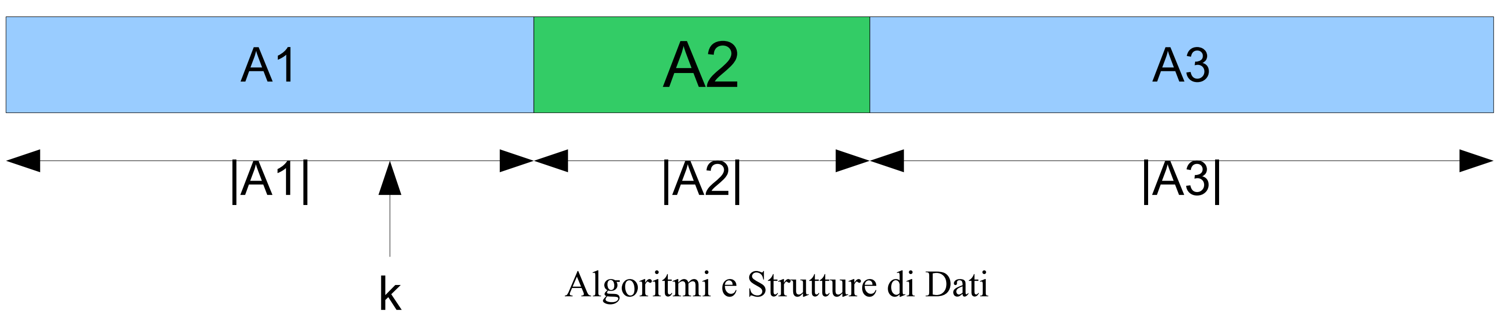


- In realtà non serve considerare tutto il vettore!
  - Supponiamo di cercare il 15mo minimo;
  - Supponiamo che A1 abbia 12 elementi e A2 ne abbia 5
  - Il valore cercato si trova in A2

Algoritmo quickSelect()

```
algorithm quickSelect( Array A, int k ) → elem
    scegli un elemento x in A
    A1 := {y in A: y < x }
    A2 := {y in A: y = x }
    A3 := {y in A: y > x }
    if ( k ≤ |A1| ) then
        return quickSelect( A1, k );
    else
        if ( k > |A1| + |A2| ) then
            return quickSelect( A3, k - |A1| - |A2| );
        else
            return x;
        endif
    endif
```

A1, A2, e A3 possono essere determinati con l'algoritmo della “bandiera nazionale”, una modifica della “partition” di quicksort che raggruppa i valori uguali al pivot sempre in O(n)



Esempio

(nota: il partizionamento è fatto in modo arbitrario)

pivot

k=6

3	75	4	14	28	16	22	5	67	54	37	91	12	17

Cerchiamo il k=5 in A3

3	75	4	14	28	16	22	5	67	54	37	91	12	17

k=5

75	4	14	28	16	22	5	67	54	37	91	12	17

Cerchiamo il k=5 in A1

4	14	28	16	22	5	67	54	37	12	17	75	91

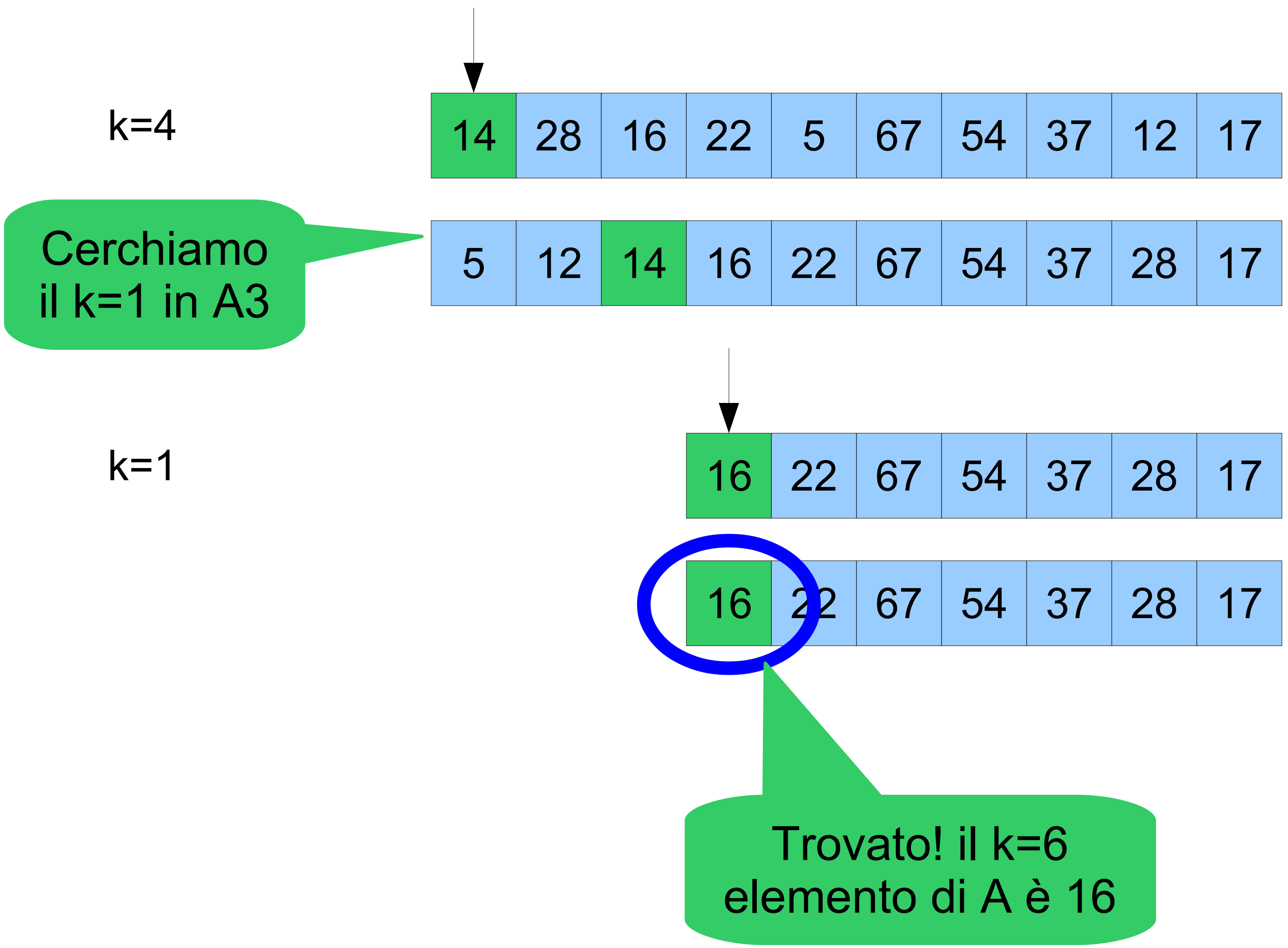
k=5

4	14	28	16	22	5	67	54	37	12	17

Cerchiamo il k=4 in A3

4	14	28	16	22	5	67	54	37	12	17

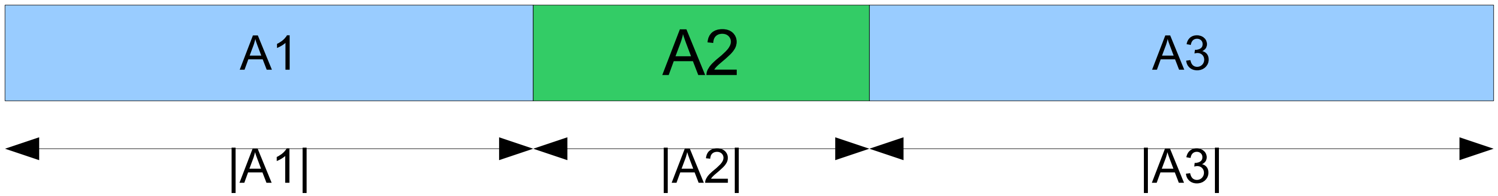
Esempio (cont.)



Analisi dell'algoritmo quickSelect()

- Costo nel caso ottimo
  - $T(n) = \Theta(n)$
  - Esecuzione del partizionamento
- Costo nel caso pessimo
  - $T(n) = T(n-1) + n = \Theta(n^2)$
  - Dimostrazione come nel caso di Quick Sort
- ...e nel caso medio?

Analisi del caso medio



- Vedremo che, anche sotto ipotesi “pessimista”, il costo medio sarà lineare
- Ipotesi “pessimista”: si effettua chiamata ricorsiva su un vettore di lunghezza max(|A1|, |A3|)
  - Scartiamo A2 (altrimenti termineremmo subito) e il sottovettore più corto tra A1 e A3
- Il numero di valori su cui si ricorre è fra n/2 e n-1:
  - Assumiamo uguale frequenza per ogni valore fra n/2 e n-1. Ogni valore appare quindi con frequenza 1/(n/2) = 2/n

Analisi del caso medio

- Si ha quindi la seguente relazione di ricorrenza per esprimere il numero T(n) di confronti richiesti:

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

- Teorema: si mostra, tramite tecnica di sostituzione, che la soluzione all'equazione di ricorrenza di cui sopra è T(n) ≤ 4n

Costosto del partizionamento usando la modifica di partition() che raggruppa gli elementi uguali al pivot

Dimostrazione

- Per sostituzione, dimostriamo che T(n) ≤ cn per una opportuna costante c

$$
\begin{aligned}
T(n) &= n-1+\frac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\
&\leq n-1+\frac{2}{n}\sum_{i=n/2}^{n-1} ci \\
&= n-1+\frac{2\mathrm{c}}{n}\left(\sum_{i=1}^{n-1} i-\sum_{i=1}^{n/2-1} i\right)
\end{aligned}
$$

Dimostrazione

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

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

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

$$\le \left(1 + \frac{3c}{4}\right)n \le cn$$

Ricordiamo che

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

- L'induzione funziona quando $(1+3c/4) \le c$, ossia $c \ge 4$

Possiamo fare ancora meglio?

- Sì, anche se non facilmente
- Esiste un algoritmo deterministico per la selezione del k-esimo elemento che ha costo O(n) nel caso peggiore
  - Trovate la descrizione dell'algoritmo (che usa una tecnica chiamata “mediano dei mediani”) nel libro di testo
  - Nota: il costo non dipende da k come nel caso di quickSelect