Algoritmi di ricerca, ordinamento, selezione

Alice Cortinovis, Giorgio Audrito

Online, 8 aprile 2021

Algoritmi di ricerca

Problema: dato un array A lungo N, vogliamo cercare se c'è un certo elemento x

1. Array non ordinato $\Rightarrow$ ricerca lineare

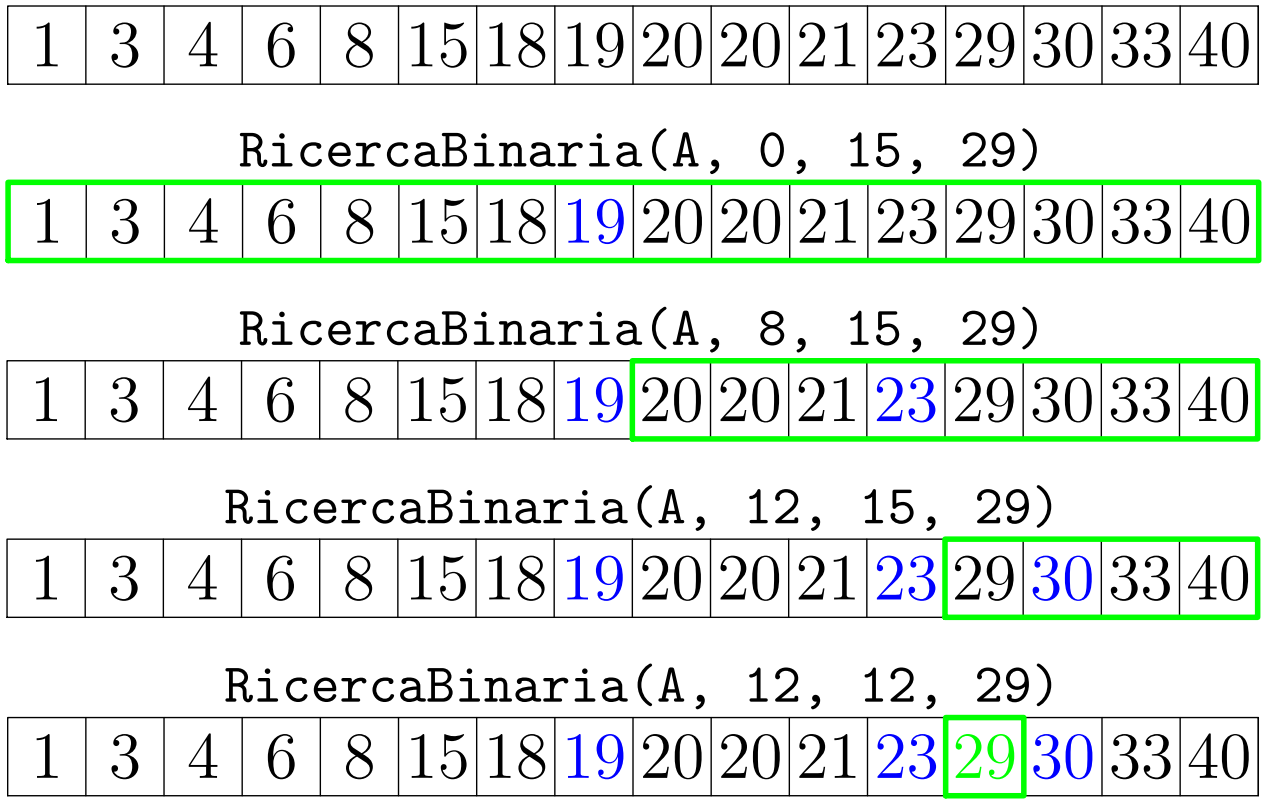
Scorro tutti gli elementi uno per uno, fino a quando trovo x
oppure se arrivo alla fine dell'array vuol dire che x non c'è

2. Array ordinato $\Rightarrow$ ricerca binaria

Questa funzione ritorna una posizione in cui c'è x *oppure* -1 se non c'è:

```
1    RicercaBinaria(int[] A, int inizio, int fine, int x) {
2       if (inizio > fine) return -1;
3       int centro = (inizio + fine)/2;
4       if (A[centro] == x) return centro;
5       else if (A[centro] < x) return RicercaBinaria(A, centro+1, fine, x);
6       else return RicercaBinaria(A, inizio, centro-1, x);
7    }
```

Esempio di ricerca binaria



Complessità

Ricerca lineare

Scandisco tutti gli elementi $\Rightarrow \mathcal{O}(n)$

Ricerca binaria

```
1    RicercaBinaria(int[] A, int inizio, int fine, int x) {
2       if (inizio > fine) return -1;
3       int centro = (inizio + fine)/2;
4       if (A[centro] == x) return centro;
5       else if (A[centro] < x) return RicercaBinaria(A, centro+1, fine, x);
6       else return RicercaBinaria(A, inizio, centro-1, x);
7    }
```

$$T(n) = \mathcal{O}(1) + T(n/2) \quad \Rightarrow \quad T(n) \in \mathcal{O}(n)$$

Algoritmi di ordinamento

Lenti $\mathcal{O}(n^2)$:

- Selection sort
- Insertion sort
- Bubble sort

Veloci $\mathcal{O}(n \log n)$:

- Merge sort
- Quick sort (soprattutto randomizzato)
- Heapsort (non lo vedremo)

Speciali $\approx \mathcal{O}(n)$:

- Counting sort
- Radix sort

Selection sort

- Metto al primo posto l'elemento più piccolo dell'array
- Metto al secondo posto l'elemento più piccolo tra quelli che rimangono
- Metto al terzo posto l'elemento più piccolo tra quelli che rimangono
- ...

```
1       int A[MAXN];
2       int N;
3
4       for (int i=0; i<N-1; i++) {
5           int m = i;
6           for (int j = i+1; j<N; j++)
7               if (A[j] < A[m])
8                   m = j;
9           swap(A[i], A[m]);
10      }
```

Esempio di selection sort

8	3	7	1	5
8	3	7	1	5
1	3	7	8	5
1	3	7	8	5
1	3	7	8	5
1	3	5	8	7
1	3	5	8	7
1	3	5	7	8
1	3	5	7	8

Complessità del selection sort

Quante operazioni fa l'algoritmo su un array A di N elementi?

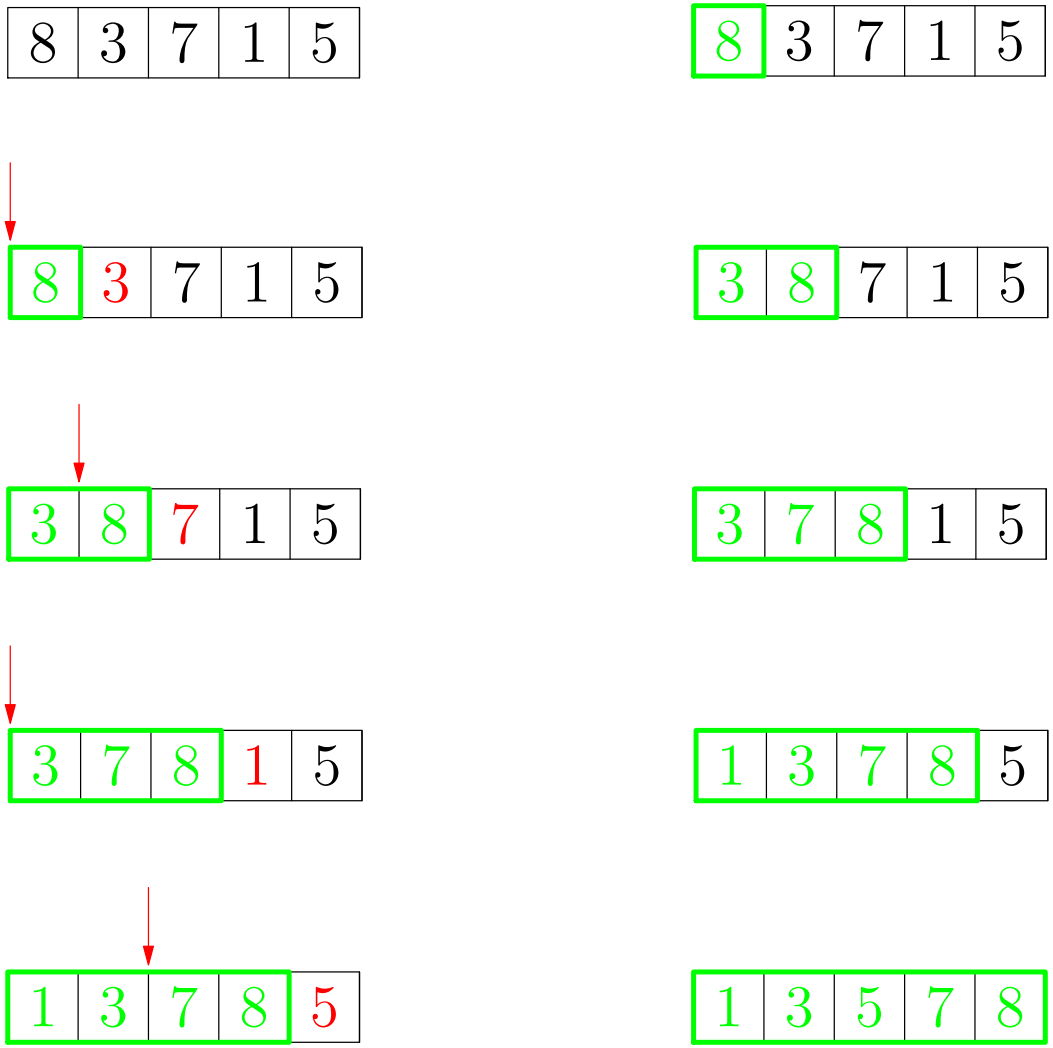
$$\mathcal{O}(N) \times \mathcal{O}(N) = \mathcal{O}(N^2)$$

Insertion sort

L'array contenente solo il primo elemento è ordinato. Induttivamente, se l'array è ordinato fino alla posizione $i-1$, guardo dove va messo l'elemento $A[i]$ e ce lo metto, spostando in avanti gli elementi più grandi di lui.

```
1        int A[MAXN];
2        int N;
3
4        for (int i=1; i<N; i++) {
5            int j = 0;
6            while (j < i && A[j] <= A[i]) j++;
7            int tmp = A[i];
8            for (int k=i; k>j; k--)
9                A[k] = A[k-1];
10           A[j] = tmp;
11       }
```

Esempio di insertion sort



Complessità dell'insertion sort

Quante operazioni fa l'algoritmo su un array A di N elementi?

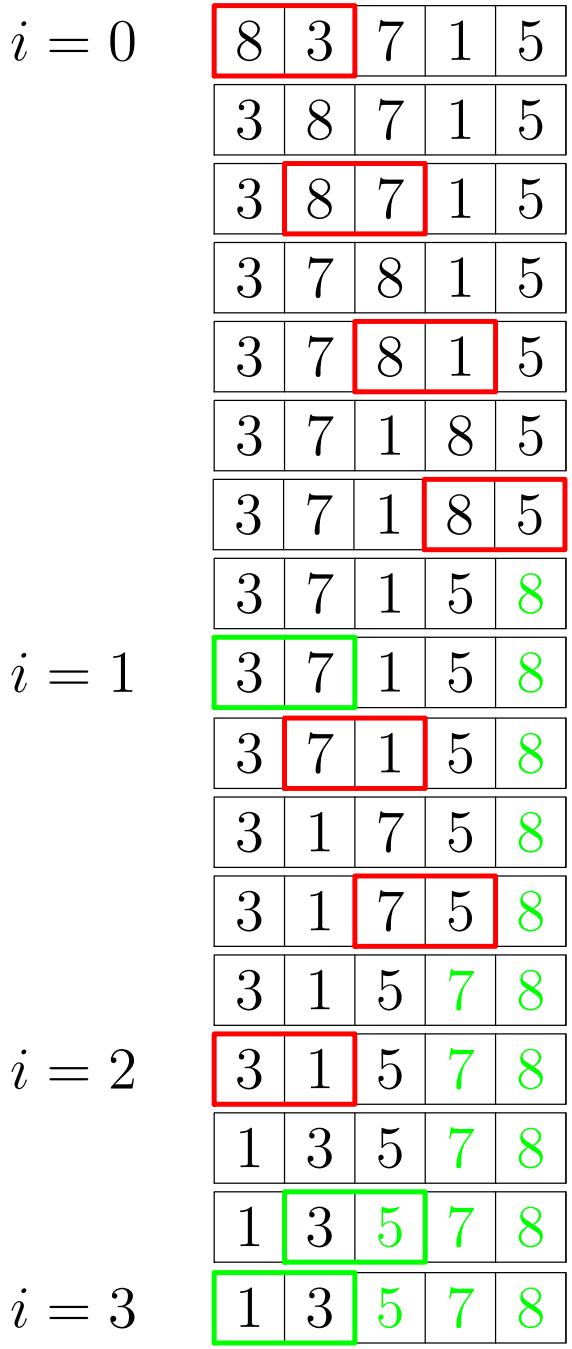
$$\mathcal{O}(N) \times \mathcal{O}(N) = \mathcal{O}(N^2)$$

Bubble sort

Idea: Ogni volta che due elementi vicini hanno ordine relativo sbagliato, li scambio.

```
1        int A[MAXN];
2        int N;
3
4        for (int i=0; i<N-1; i++)
5            for (int j=1; j<N-i; j++)
6                if (A[j-1] > A[j])
7                    swap(A[j-1], A[j]);
```

Perché basta fare $N-1$ passate di scambi?
E perché il ciclo interno si ferma a $N-i$?
perché a ogni passata uno degli elementi in fondo va a posto


$i = 0$
8 3 7 1 5
3 8 7 1 5
3 8 7 1 5
3 7 8 1 5
3 7 8 1 5
3 7 1 8 5
3 7 1 8 5
3 7 1 5 8
$i = 1$
3 7 1 5 8
3 7 1 5 8
3 1 7 5 8
3 1 7 5 8
3 1 5 7 8
$i = 2$
3 1 5 7 8
1 3 5 7 8
1 3 5 7 8
$i = 3$
1 3 5 7 8

Complessità del bubble sort

1. Quante operazioni fa l'algoritmo su un array A di N elementi?

$$\mathcal{O}(N) \times \mathcal{O}(N) = \mathcal{O}(N^2)$$

2. Sapete trovare una situazione in cui dovete scambiare elementi *tutte* le volte?

$$A = [5, 4, 3, 2, 1]$$

Merge-sort

L'idea è di dividere l'array da ordinare in due parti della stessa lunghezza, ordinare ricorsivamente le due parti, e poi unirle (*merge*) in modo da rispettare l'ordine.

```
1        void MergeSort(int left, int right) {
2           if (left < right) {
3              int center = (left + right)/2;
4              MergeSort(left, center);
5              MergeSort(center+1, right);
6              Merge(left, center, right);
7           }
8        }
9
10       MergeSort(0, N-1);
```

```
1   void Merge(int left, int center, int right) {
2       int i = left;
3       int j = center+1;
4       int k = 0;
5       int tmp[MAXN];
6       while (k < right - left + 1) {
7           while (i <= center && (j > right || A[i] <= A[j])) {
8               tmp[k] = A[i];
9               i++,
10              k++;
11          }
12          while (j <= right && (i > center || A[j] <= A[i])) {
13              tmp[k] = A[j];
14              j++;
15              k++;
16          }
17      }
18      for (int i=0; i < right - left + 1; i++) A[left+i] = tmp[i];
19  }
```

Esempio di merge sort

`MergeSort(0, 7):`

A :

2	8	0	4	6	1	7	9

`MergeSort(0, 3):`

A :

2	8	0	4	6	1	7	9

A :

0	2	4	8	6	1	7	9

`MergeSort(4, 7):`

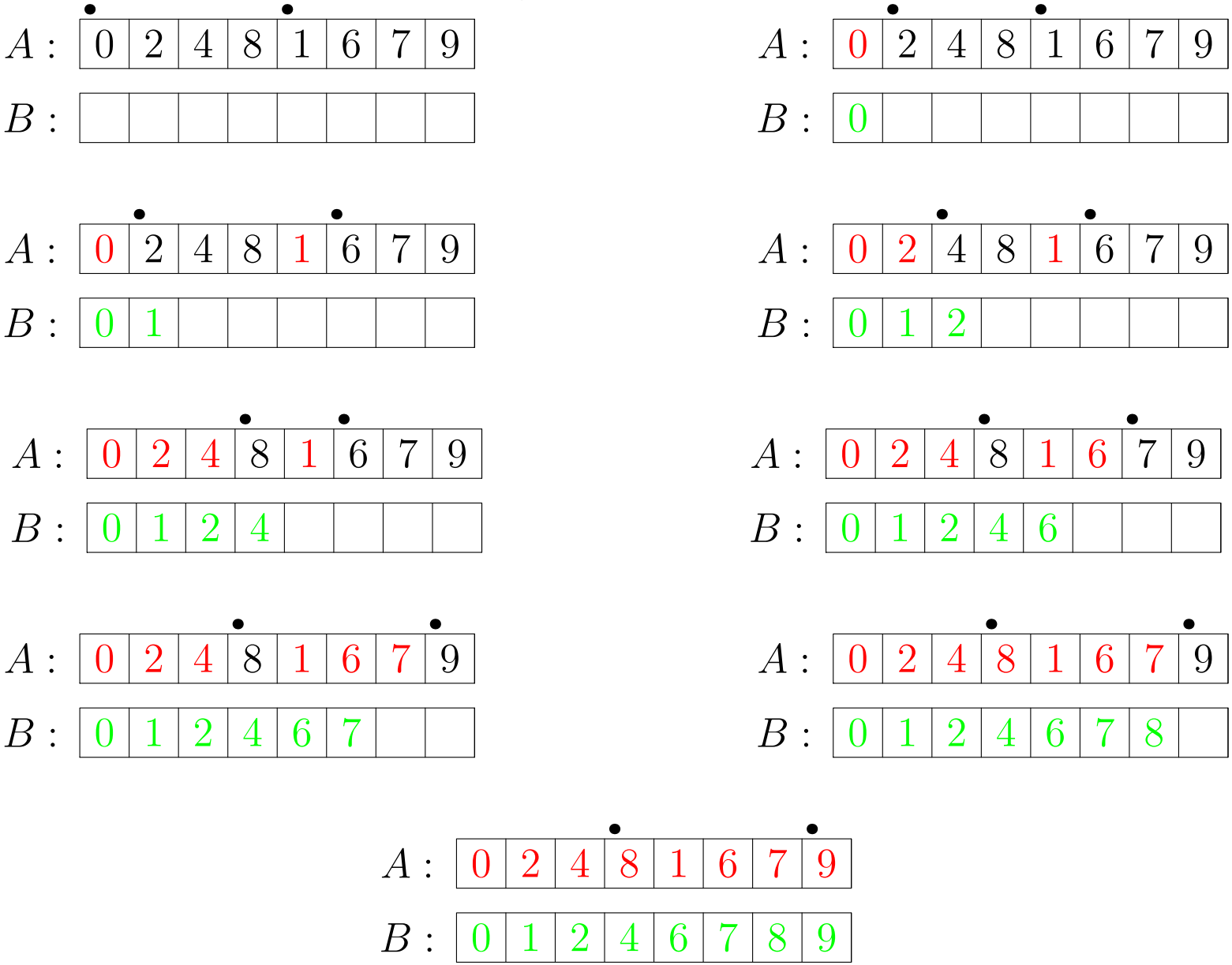
A :

0	2	4	8	6	1	7	9

A :

0	2	4	8	1	6	7	9

`Merge(0, 3, 7):`


A : 0 2 4 8 1 6 7 9
B :
A : 0 2 4 8 1 6 7 9
B : 0
A : 0 2 4 8 1 6 7 9
B : 0 1
A : 0 2 4 8 1 6 7 9
B : 0 1 2
A : 0 2 4 8 1 6 7 9
B : 0 1 2 4
A : 0 2 4 8 1 6 7 9
B : 0 1 2 4 6
A : 0 2 4 8 1 6 7 9
B : 0 1 2 4 6 7
A : 0 2 4 8 1 6 7 9
B : 0 1 2 4 6 7 8
A : 0 2 4 8 1 6 7 9
B : 0 1 2 4 6 7 8 9

Complessità del merge sort

Quante operazioni fa l'algoritmo su un array A di N elementi?

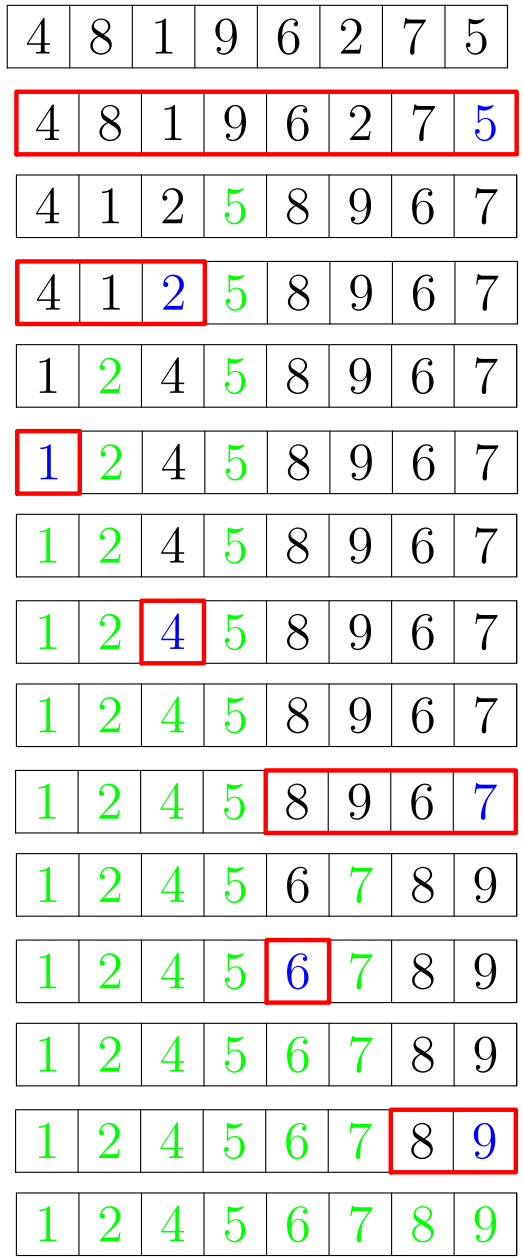
$$T(N) = \mathcal{O}(N) + 2T(N/2) \quad \Rightarrow \quad T(N) = \mathcal{O}(N \log N)$$

Dimostrazione

$$N + 2\left(\frac{N}{2}\log\frac{N}{2}\right) = N + N(\log N - 1) = N \log N$$

Quick-sort

- Nell'array da ordinare prendo un elemento a caso (l'ultimo?), che sarà detto *pivot*.
- Partiziono l'array in modo che a sinistra del pivot ci stiano i numeri minori di lui, e a destra quelli maggiori.
- Ricorsivamente ordino la parte destra e sinistra dell'array (se non sono vuote).


4 8 1 9 6 2 7 5
4 8 1 9 6 2 7 5
4 1 2 5 8 9 6 7
4 1 2 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 8 9 6 7
1 2 4 5 6 7 8 9
1 2 4 5 6 7 8 9
1 2 4 5 6 7 8 9
1 2 4 5 6 7 8 9

Esempio di implementazione del quick-sort

```
1   int PARTITION(int left, int right) {
2       int pivot = A[right];
3       int i = left-1;
4       for (int j = left; j < right; j++)
5           if (A[j] < pivot) {
6               i++;
7               swap(A[i], A[j]);
8           }
9       swap(A[i+1], A[right]);
10      return i+1;
11  }
12
13  void QUICKSORT(int left, int right) {
14      if (left < right) {
15          int q = PARTITION(left, right);
16          QUICKSORT(left, q-1);
17          QUICKSORT(q+1, right);
18      }
19  }
20
21  QUICKSORT(0, N-1);
```

Complessità del quick sort

Quante operazioni fa l'algoritmo su un array A di N elementi?

In media fa $\mathcal{O}(N \log N)$ operazioni

se per ogni pivot divido circa a metà, la ricorrenza diventa la stessa del merge sort:

$$T(N) = \mathcal{O}(N) + 2T(N/2) \quad \Rightarrow \quad T(N) = \mathcal{O}(N \log N)$$

al caso pessimo fa $\mathcal{O}(N^2)$ operazioni

Per esempio... se prendiamo l'ultimo elemento come pivot in un array già ordinato!

In pratica, scegliendo un pivot a caso è molto veloce

Lower bound per l'ordinamento

Esiste un algoritmo di ordinamento che ci mette meno di $\mathcal{O}(N \log N)$?
Dipende.

Definizione Diciamo che un algoritmo di ordinamento è *per confronto* se l'ordine degli elementi dell'array è basato soltanto sul confronto di elementi.

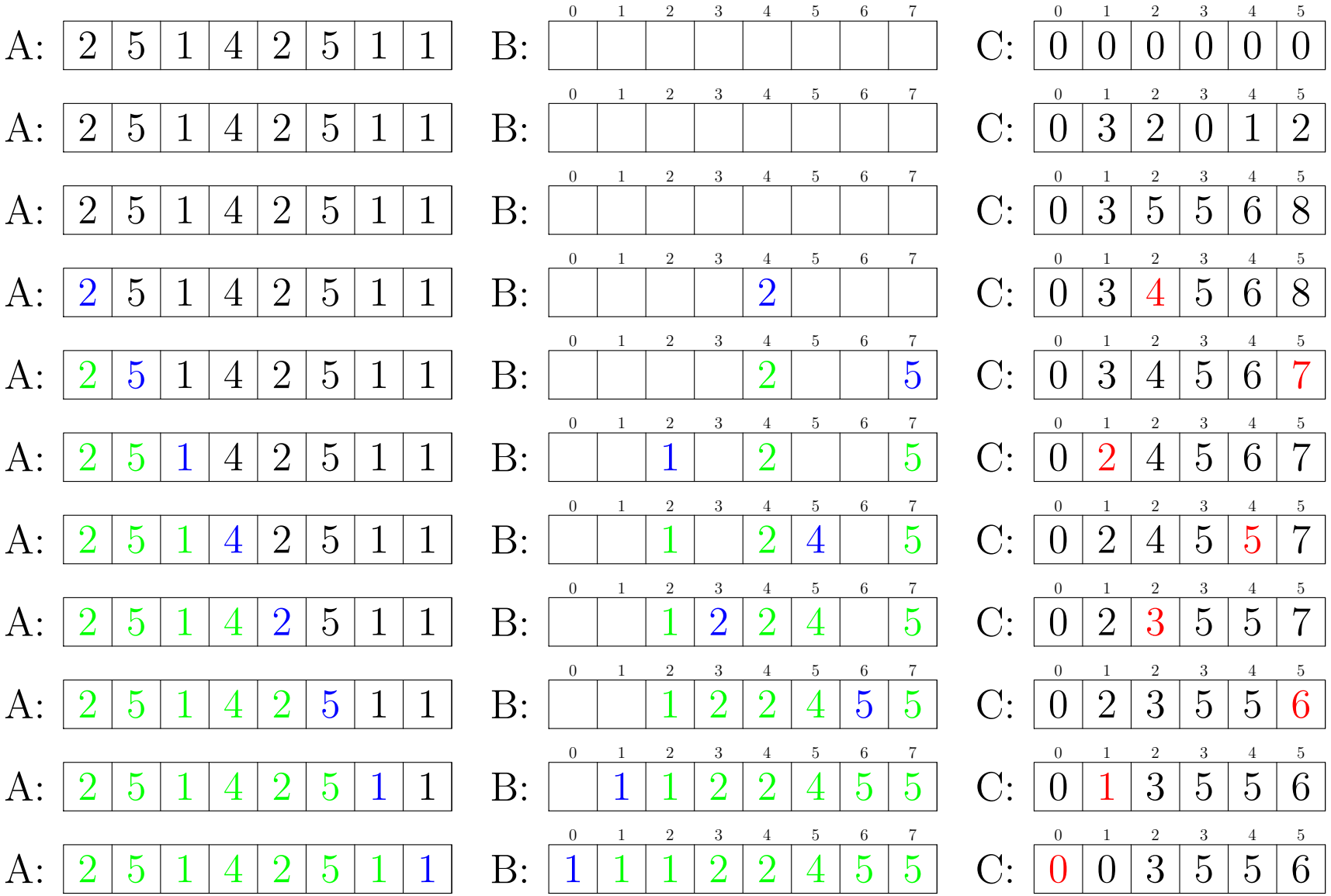
Un algoritmo per confronto richiede sempre almeno $\Omega(N \log N)$ confronti nel caso pessimo.
Dimostrazione
L'*albero di decisione* ha $N!$ foglie dunque ha altezza

$$H \geq \log_2 N! \geq \log_2 \left(\left(\frac{N}{2} \right)^{N/2} \right) = \frac{N}{2} \cdot \log_2 \frac{N}{2} = \Omega(N \log N)$$

Counting sort

Si può usare quando i valori di $A[i]$ sono limitati.

```
1         int A[MAXN], B[MAXN+1];
2         int N, K;
3         int C[MAXK+1];
4
5         for (int i=0; i<N; i++) C[A[i]]++;
6         for (int i=1; i<=K; i++) C[i] += C[i-1];
7         for (int i=N-1; i>=0; i--) {
8            B[C[A[i]]-1] = A[i];
9            C[A[i]]--;
10        }
```


A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
C: 0 1 2 3 4 5
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
C: 0 1 2 3 4 5
0 3 2 0 1 2
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
C: 0 1 2 3 4 5
0 3 5 5 6 8
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
2
C: 0 1 2 3 4 5
0 3 4 5 6 8
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
2 5
C: 0 1 2 3 4 5
0 3 4 5 6 7
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 2 5
C: 0 1 2 3 4 5
0 2 4 5 6 7
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 2 4 5
C: 0 1 2 3 4 5
0 2 4 5 5 7
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 2 2 4 5
C: 0 1 2 3 4 5
0 2 3 5 5 7
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 2 2 4 5 5
C: 0 1 2 3 4 5
0 2 3 5 5 6
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 1 2 2 4 5 5
C: 0 1 2 3 4 5
0 1 3 5 5 6
A: 2 5 1 4 2 5 1 1
B: 0 1 2 3 4 5 6 7
1 1 1 2 2 4 5 5
C: 0 1 2 3 4 5
0 0 3 5 5 6

Complessità del counting sort

Quante operazioni fa l'algoritmo su un array A di N elementi in cui l'elemento massimo è K?

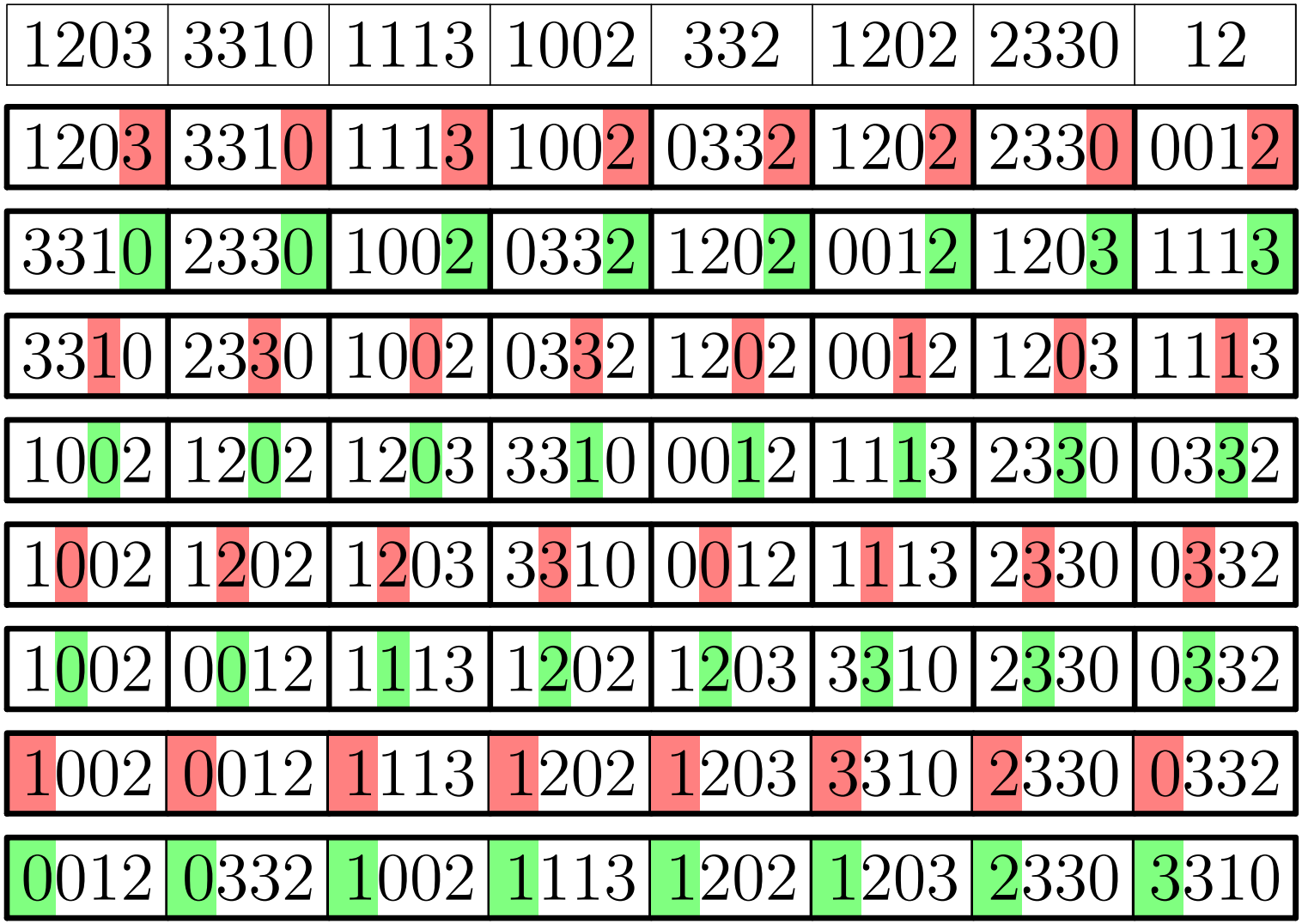
$$\mathcal{O}(N + K)$$

Radix sort

Ordinate rispetto alla cifra meno significativa utilizzando un ordinamento stabile (ad esempio il counting sort visto prima). Poi ordinate rispetto alla seconda cifra meno significativa, e così via.

Definizione Un ordinamento su un array A si dice *stabile* se quando due elementi $A[i]$ e $A[j]$ sono "uguali" e $i < j$, dopo l'ordinamento stanno ancora nello stesso ordine relativo nel'array A.

Esempio di radix sort



Esempio di implementazione del radix sort:

```
1   int A[MAXN], B[MAXN+1];
2   int N, K;
3   int C[10];
4
5   void stable_sort(int cifra) {
6      for (int i=0; i<=9; i++) C[i] = 0;
7      int p = round(pow(10, cifra));
8
9      for (int i=0; i<N; i++) C[(A[i]/p) % 10]++;
10     for (int i=1; i<=9; i++) C[i] += C[i-1];
11     for (int i=N-1; i>=0; i--) {
12        B[C[(A[i]/p) % 10]] = A[i];
13        C[(A[i]/p) % 10]--;
14     }
15
16     for (int i=0; i<N; i++) A[i] = B[i+1];
17  }
18
19  for (int i=0; i<numero_massimo_cifre; i++)
20     stable_sort(i);
```

Complessità del radix sort

Quante operazioni fa l'algoritmo su un array A di N elementi che hanno al massimo C cifre in base B?

$$\mathcal{O}(C \cdot (N + B)) \quad \Rightarrow \quad \mathcal{O}(N \log K) \text{ se numeri}$$

Può convenire con numeri molto ripetuti, ma più spesso si applica a stringhe

Algoritmi di selezione

Problema: dato un array non ordinato A di lunghezza N e un intero $0 \leq k \leq N-1$, trovare il k-esimo elemento di A in ordine crescente

Prima idea: ordinare l'array e prendere $A[k] \Rightarrow$ Tempo $O(N \log N)$.

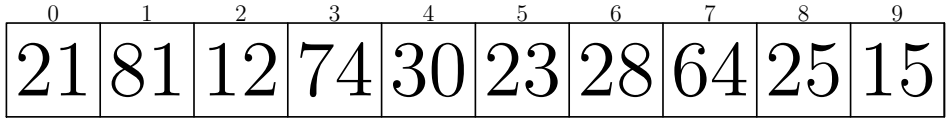
Vorremmo fare più veloce...

Randomized select

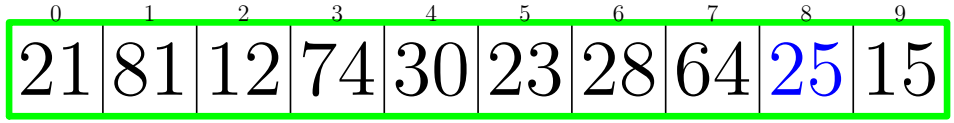
Usiamo una versione modificata del quicksort randomizzato:

```
1   RandomizedSelect(A, left, right, k) {
2       if (left == right) return A[left];
3
4       // Chiamo q l'indice del pivot dopo la partizione
5       int q = RandomizedPartition(A, left, right);
6
7       int x = q - left;
8       if (x == k) return A[x];
9       else if (x < k) return RandomizedSelect(A, q+1, right, k-x-1);
10      else return RandomizedSelect(A, left, q-1, k);
11  }
```

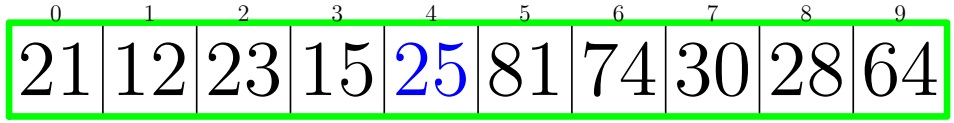
Esempio

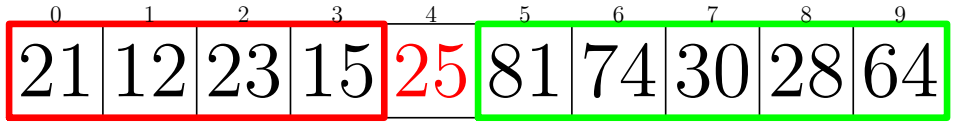


`RandomizedSelect(A, 0, 9, 6)`

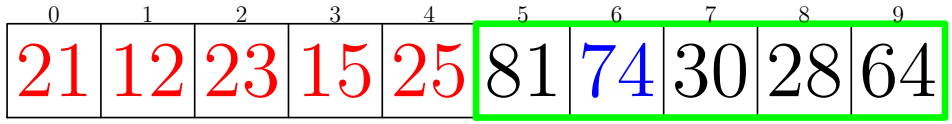


$q = 4;\ x = 4;$

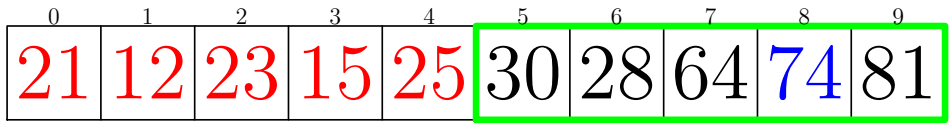


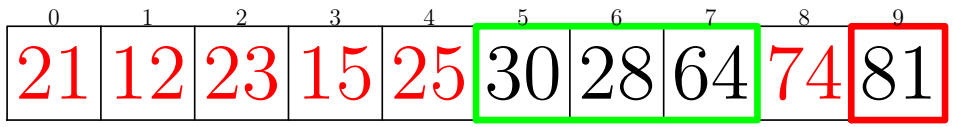


`RandomizedSelect(A, 5, 9, 1)`

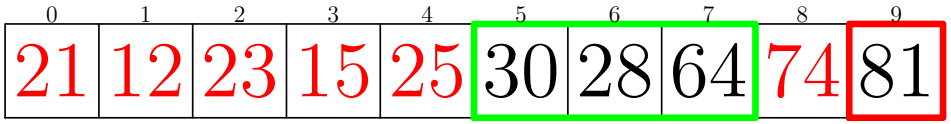


$q = 8;\ x = 3;$

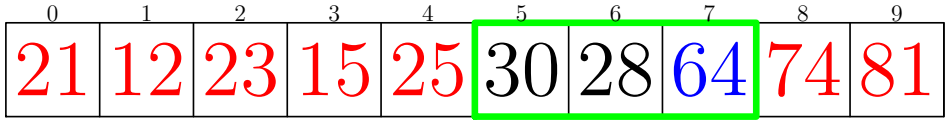




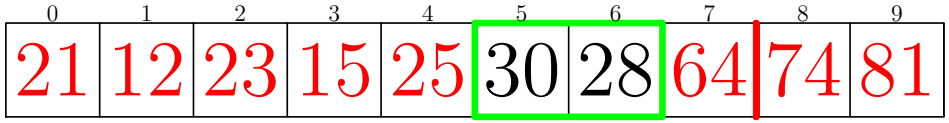
Esempio (continuazione)



`RandomizedSelect(A, 5, 7, 1)`

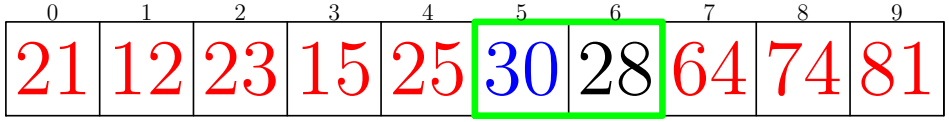


$q = 7;\ x = 2;$



`RandomizedSelect(A, 5, 6, 1)`

$q = 6;\ x = 1;$



Complessità del RandomizedSelect

Quante operazioni fa l'algoritmo di RandomizedSelect su un array A di N elementi?

Caso medio

$$T(N) = \mathcal{O}(N) + T(N/2) \quad \Rightarrow \quad T(N) = \mathcal{O}(N)$$

Dimostrazione: se $T(N) = 2N$, allora $T(N) = N + T(N/2)$

Caso pessimo

$$T(N) = \mathcal{O}(N) + T(N-1) \quad \Rightarrow \quad T(N) = \mathcal{O}(N^2)$$

Dimostrazione: se $T(N) = \frac{N(N-1)}{2}$, allora $T(N) = N + T(N-1)$

. . . comunque è randomizzato quindi non succede

Selezione in tempo lineare

1. Divido gli N elementi in $\lfloor n/5 \rfloor$ gruppi di 5 elementi ciascuno, e un gruppo con gli $n \bmod 5$ elementi rimasti.
2. Trovo la mediana di ciascun gruppo di elementi (ordinando ciascun gruppo con un insertion/bubble sort, tanto sono solo 5 elementi).
3. Uso lo stesso algoritmo, ricorsivamente, per trovare la mediana x delle $\lceil n/5 \rceil$ mediane trovate al passo precedente.
4. Faccio una partizione dell'array di input attorno a x. Ora, sia i il numero di elementi che stanno prima di x aumentato di 1, cioè tale che x sia l'i-esimo elemento di A in ordine crescente e ci siano $N - i$ elementi nella parte alta della partizione.
5. Se $k = i$, ritorno x. Altrimenti ripeto l'algoritmo per trovare il k-esimo elemento della parte bassa della partizione, o il $(N - k)$-esimo elemento della parte alta della partizione.

Selezione in tempo lineare

Complessità

$$U(N) = \mathcal{O}(N) + U(N/5) \quad \Rightarrow \quad U(N) = \mathcal{O}(N) \text{ per trovare } x$$

$$T(N) = U(N) + T(\frac{3}{10}N) \quad \Rightarrow \quad U(N) = \mathcal{O}(N) \text{ in tutto}$$

perché almeno $\lfloor n/10 \rfloor$ gruppi hanno una mediana maggiore di x,
e quindi tre elementi maggiori di x

Comunque in pratica è meglio il quickselect...