Algoritmi di ordinamento

Ordinamento per selezione.

Esegue $n-1$ passi. Al passo generico $i = 0, 1, \ldots, n-2$, viene selezionato l'elemento che occuperà la posizione i nella sequenza ordinata.

Pertanto, a conclusione del passo i-esimo, i primi $(i+1)$ elementi della sequenza sono già ordinati.

L'algoritmo, al passo i, seleziona il minimo (massimo per un ordinamento in senso decrescente) tra gli elementi che si trovano dalla posizione i in poi.

```
SelectionSort (a)
  FOR (i = 0;  i < n-1;  i = i+1){
    minimo = a[i];
    indiceMinimo = i;
    FOR (j = i+1;  j < n;  j = j+1){
      IF (a[j] < minimo){
        minimo = a[j];
        indiceMinimo = j;
      }
    }
    a[indiceMinimo] = a[i];
    a[i] = minimo;
  }
```

... $\forall$ posizione i ...

... poni temporaneamente minimo = a[i], e ind... = i

... della posizione successiva j fino alla fine ...

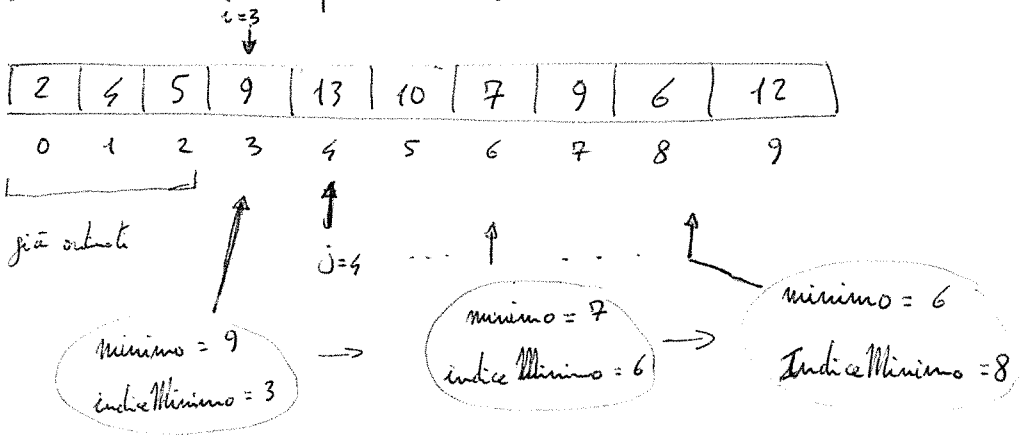
se a[j] è più piccolo ...

... ricordati e

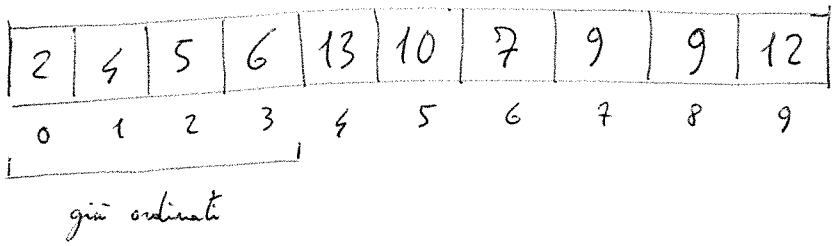
... segnati la posizione ...

... sposta a[i] al posto del minimo che hai trovato e metti il minimo in a[i]

Esempio di esecuzione. Al passo $i = 3$.



Configurazione di partenza.



Configurazione al termine del passo $i = 3$.

Teorema. Il SelectionSort ordina un array di n elementi in tempo $\Theta(n^2)$.

Dimostrazione. L'algoritmo esegue $(n-1)$ iterazioni del ciclo esterno. (non n perché l'elemento che resta è più grande di tutti gli altri)

Il costo t_i della i-esima iterazione è dato da alcune operazioni di costo $O(1)$ più il costo delle operazioni del ciclo interno che cerca il minimo. Il numero di operazioni del ciclo interno è proporzionale a $n - i$. Pertanto, il numero totale di operazioni eseguite è dato da

$$\sum_{i=0}^{n-2} (n-i) = \left(\begin{array}{l} \text{Ponendo } K = n-i, \text{ si ha che, per } i=0,\ K=n \\ \text{e, per } i = n-2,\ K = 2 \end{array} \right) =$$

$$= \sum_{K=2}^{n} K = \left(\text{Ricordando che } \sum_{K=1}^{n} K = \frac{n(n+1)}{2} \right)$$

$$= \frac{n(n+1)}{2} - 1 = \Theta(n^2)$$

■

Osservazione: Tale complessità è sempre raggiunta per qualsiasi sequenza di input. Non esistono sequenze particolarmente vantaggiose.

La correttezza discende immediatamente osservando che, al termine dell'iterazione i-esima del ciclo esterno, i primi $(i+1)$ elementi coincidono con i primi $(i+1)$ elementi della sequenza finale ordinata.

Infatti, ragionando per induzione sul numero di iterazioni:

Base $it = 1$. (corrisponde al ciclo all'iterazione $i=0$) L'algoritmo cerca il minimo tra gli n elementi e lo pone in posizione $a[0]$. Le iterazioni successive non spostano più l'elemento. Coincide con il primo elemento nella sequenza finale ordinata.

Ipotesi induttiva $it = i$. (corrisponde all'ado all'iterazione $i-1$) I primi i elementi coincidono con i primi i elementi della sequenza ordinata.

Passo induttivo $it = i+1$ (corrisponde nel ciclo all'iterazione i) L'algoritmo cerca il minimo tra l'elemento in posizione i e i successivi. Trovandolo lo pone in posizione i, cioè in $a[i]$. Non sarà più spostato.

Poiché non è stato scelto prima è maggiore di tutti i precedenti, $a[0] \dots a[i-1]$.

Pertanto, alla fine dell'iterazione, i primi $(i+1)$ elementi della sequenza coincidono con i primi $(i+1)$ elementi della sequenza finale ordinata

■

Note: a) l'algoritmo ordina in LOCO (non ricopia gli elementi in un altro array)

b) l'algoritmo NON è stabile (nel senso che elementi uguali non preservano l'ordine con cui sono presenti nella sequenza di input in quella finale ordinata)
(es. il "9" nell'esempio di applicazione descritto)

Ordinamento per inserimento

Esegue $n-1$ passi. Al passo generico $i = 1, 2, \dots, n-1$, l'elemento in posizione i viene inserito al posto giusto tra i primi $i+1$ elementi.

Pertanto, a conclusione del passo i-esimo, gli $(i+1)$ elementi sistemati fino a quel momento sono tra di loro ordinati.

Non coincidono necessariamente con gli $(i+1)$ elementi dell'ordinamento finale.

L'algoritmo, al passo i, confronta l'elemento in posizione i con i precedenti, a partire dal vicino, fino a trovare "il posto giusto" in cui inserire l'elemento. Ad ogni confronto, l'elemento che risulta più grande dell'elemento i-esimo, viene spostato di una posizione a destra.

```
InsertionSort (a)
  FOR ( i=1;  i<n;  i=i+1){
      prossimo = a[i];
      j = i;
      WHILE (j>0 && (a[j-1] > prossimo)){
          a[j] = a[j-1];
          j = j - 1;
      }
      a[j] = prossimo;
  }
```

... $\forall$ posizione i ...

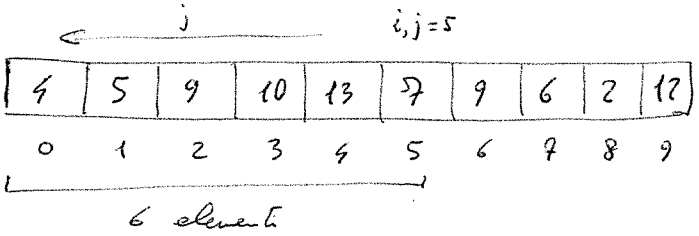
... copio $a[i]$ in una variabile esterna ... posso scrivere in $a[i]$

... fino a quando non raggiungo la fine del vettore a sinistra o trovo un elemento più piccolo di $a[i]$

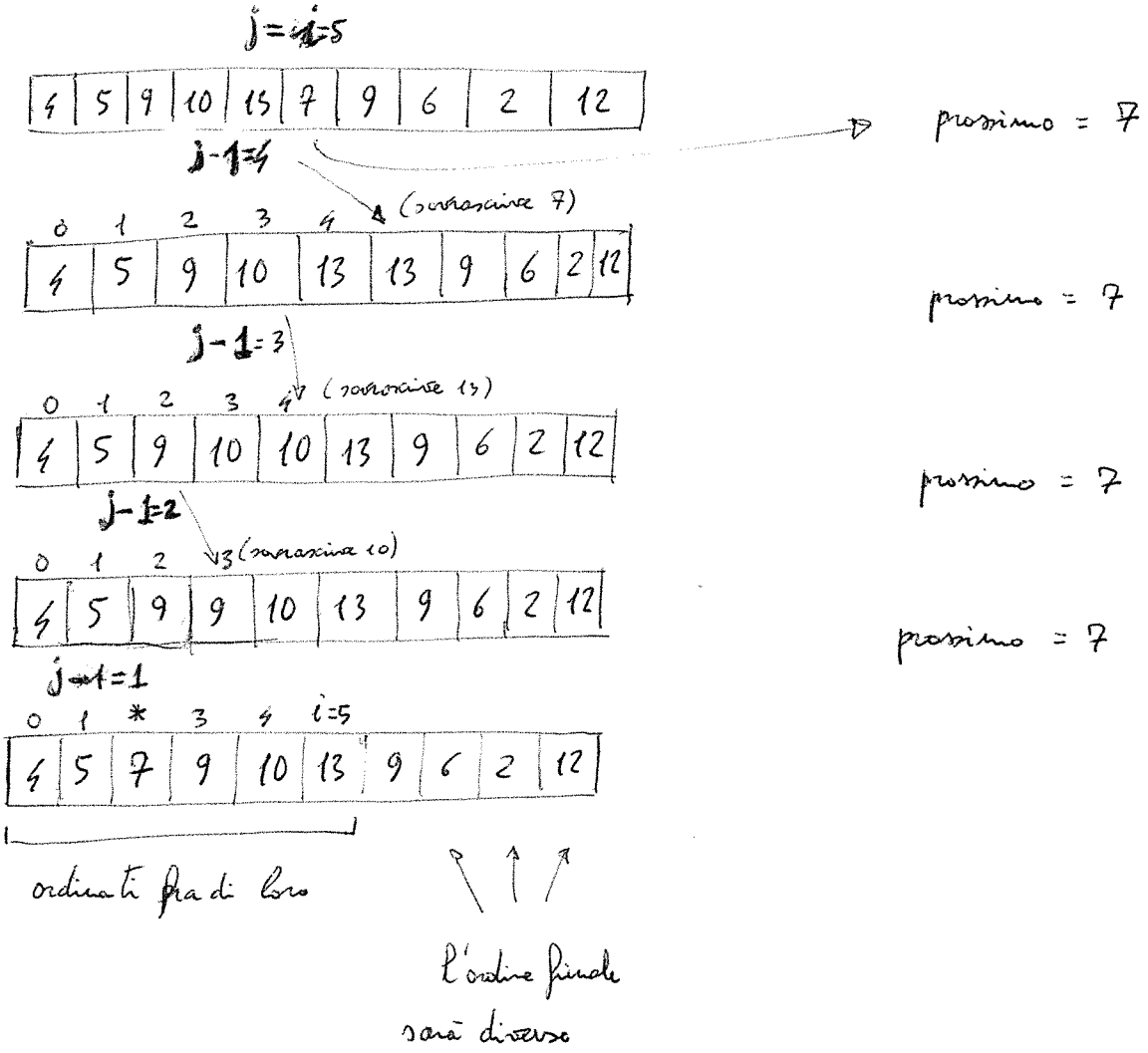
sposto l'elemento a destra di una posizione ...

... inserisco $a[i]$ al posto giusto (fino ad ora) ...

Esempio di esecuzione. Al passo $i = 5$



All'inizio della iterazione $i = 5$, si cerca di posizionare $a[5] = 7$, al posto giusto tra i primi 6 elementi. Quindi



Teorema. L'InsertionSort ordina un array di n elementi in tempo $O(n^2)$

Dimostrazione. L'algoritmo esegue $n-1$ passi (ciclo esterno).
(non n perché il "confronto" comincia dal 2° elemento dell'ARRAY)

Il costo t_i della i-esima iterazione, a meno di operazioni di costo $O(1)$, dipende dal numero di posizioni precedenti esaminate. È <u>al più</u> pari ad $(i+1)$: confronto con tutti gli i elementi nelle posizioni $a[i-1], a[i-2], \dots a[0]$ più l'ultima verifica della condizione $j>0$ che porta all'uscita del ciclo WHILE. Pertanto, il numero totale di operazioni è proporzionale a ($\leq$ per la precisione)

$$\sum_{i=1}^{n-1} (i+1) = \left(\begin{array}{l}\text{Ponendo } K = i+1, \text{ e notando che, per } i=1 \\ K=2, \text{ e per } i=n-1,\ K=n\end{array}\right)$$

$$= \sum_{K=2}^{n} K = \left(\begin{array}{l}\text{Per quanto già visto} \\ \text{nell'analisi precedente}\end{array}\right)$$

$$= \frac{n(n+1)}{2} - 1 = O(n^2)$$

∎

La correttezza discende immediatamente osservando che, al termine della i-esima iterazione del ciclo esterno, i primi $(i+1)$ elementi sono ordinati tra di loro. Pertanto, al termine della $(n-1)$-esima iterazione, gli n elementi sono ordinati tra di loro.

Infatti, ragionando per induzione sul numero di iterazioni:

Base $it = 1$. Il secondo elemento dell'array viene confrontato con il primo. Se minore, i due vengono scambiati di posto. Pertanto, al termine della prima iterazione, i primi due elementi sono ordinati tra di loro.

Ipotesi induttiva $it = i > 1$. Al termine della i-esima iterazione i primi $(i+1)$ elementi sono ordinati tra di loro.

Passo induttivo $it = n-1$. L'algoritmo cerca di posizionare l'elemento $a[n-1]$, nella posizione giusta tra i primi $n-1$ elementi che, per ipotesi induttiva, sono già ordinati tra di loro.

Il ciclo WHILE sposta gli elementi precedenti fino a trovare un elemento minore di $a[n-1]$ e ad inserirlo alla sua destra.

In questa configurazione:

- tutti gli elementi a destra sono già ordinati e minori
- quelli a sinistra sono maggiori e già ordinati (sono stati semplicemente traslati di un posto ...)

Pertanto, la sequenza finale risulta ordinata.

Osservazioni. A differenza del Selection Sort, però richiede un tempo significativamente inferiore. Se la sequenza è già ordinata il tempo diviene $O(n)$. Infatti il ciclo WHILE viene eseguito sempre una sola volta e quindi

$$\sum_{i=1}^{n-1} 1 = n-1 = O(n).$$

la proprietà si mantiene anche se <u>solo un numero costante di elementi</u> è fuori ordine.

D'altra parte, se la sequenza è già ordinata <u>in senso decrescente</u>, il tempo di esecuzione è $\Theta(n^2)$.

Infatti, sia $a[0] > a[1] > \dots > a[n-1]$. L'inserimento di $a[i]$ richiede sempre di raggiungere la prima posizione e, quindi, $(i+1)$ valutazioni della condizione del ciclo WHILE.

Pertanto, il tempo di esecuzione è proporzionale a ($\geq$ per la precisione)

$$\sum_{i=1}^{n-1} (i+1) = \frac{n \cdot (n+1)}{2} - 1 = \Theta(n^2) \quad \begin{pmatrix} \Omega(n^2) \\ O(n^2) \end{pmatrix}$$

In conclusione anche la complessità in tempo dell'Insertion-Sort è $\Theta(n^2)$ (esistono configurazioni dell'input come visto che richiedono $\Omega(n^2)$ passi), ma <u>il comportamento dipende da quanto la sequenza di input sia già ordinata.</u>

L'algoritmo ordina in LOCO.

L'algoritmo è stabile.

Inoltre si noti che, nel caso di due applicazioni che vengono eseguite "in concorrenza", in cui la prima ordina gli elementi e la seconda li elabora, l'algoritmo SelectionSort è utilizzabile per "far lavorare contemporaneamente" le due applicazioni (gli elementi già ordinati non vengono più spostati), mentre l'InsertionSort no (l'ordine intermedio non è quello finale).