In questa lezione

Ordinamenti quadratici

[CLRS] par 2.2 e problema 2.2

Specifica del problema dell ordinamento

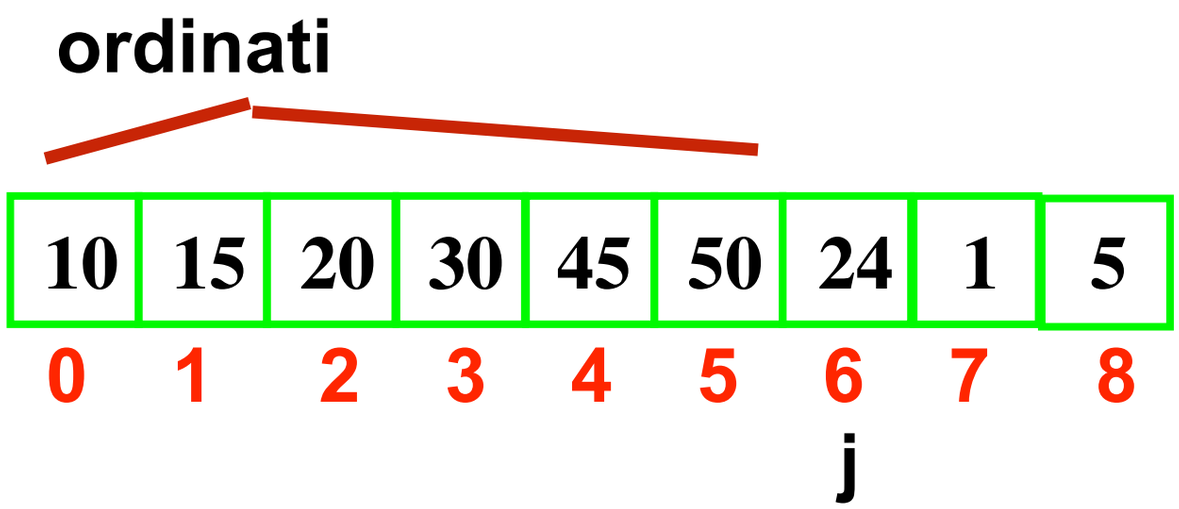
Per tutti gli algoritmi di ordinamento la precondizione è
"preCond: input A è un array di elementi su cui è definito un ordinamento totale, ≤."
e la postcondizione è
"postCond: A[0],…,A[n-1] è una permutazione degli elementi iniziali tale che A[0]≤…≤A[n-1] "

Non la scriveremo ogni volta per non appesantire.

Progettazione di un algoritmo di ordinamento

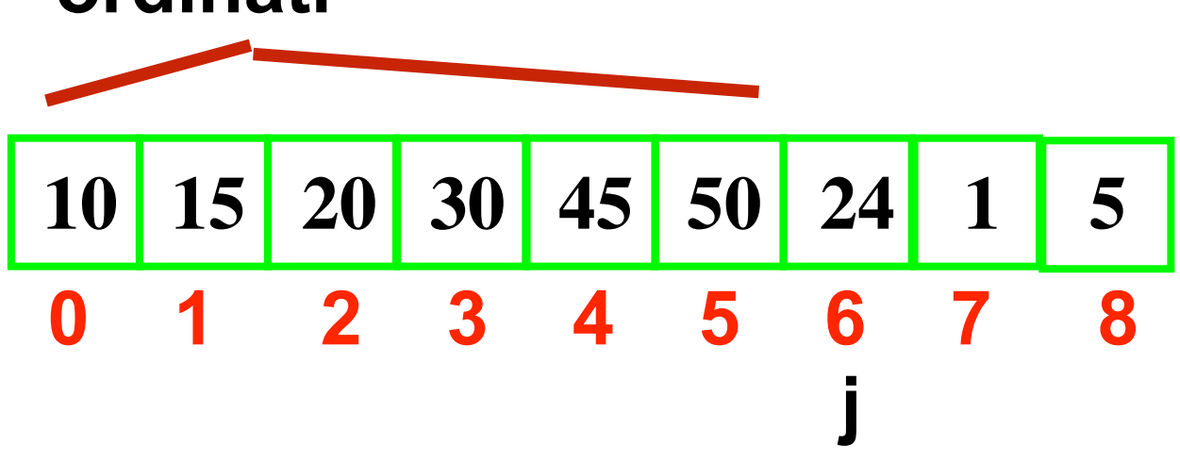
Il problema dell'ordinamento:
l'input è un array di numeri (interi), A[0:n-1],
la postcondizione (output) è: gli elementi di A[0:n-1] sono una permutazione ordinata crescente (per esempio) di quelli iniziali: A[0] ≤A[1]≤…≤A[n-1].

Supponiamo di arrivare alla postcondizione un passo alla volta, cioè incrementalmente, estendendo in ogni passo l'ordinamento a un nuovo elemento, utilizzando un indice j per scorrere la lista verso destra a partire da 0.



Progettazione di un algoritmo incrementale di ordinamento

Ci mettiamo in un passo generico:
consideriamo l'elemento A[j], sotto l'ipotesi che gli elementi di A[0:j-1] sono già ordinati: A[0] ≤A[1]≤...≤A[j-1], per j≤n.



Ora si tratta di decidere come estendere l' ordinamento anche ad a A[j]

Progettazione insertion sort

L'idea implementata nell'insertion sort per estendere l'ordinamento al prossimo elemento A[j], supponendo che A[0] ≤A[2]≤...≤A[j-1], per j≤n, è quella di

inserire A[j] al posto giusto tra i primi j.

Realizzato l'inserimento gli elementi di A[0:j] saranno ordinati:

A[0] ≤A[1]≤...≤A[j], per j≤n

Evidentemente il primo da inserire è il secondo elemento A[1], in quanto il primo da solo costituisce un insieme ordinato.

Vedremo due modi diversi di realizzare questo tipo di inserimento.

InsertionSort: pseudocodice

La funzione di inserimento è così specificata:
Ins(A,i,j)
input:A è un array di interi, e i e j indici in [0,len(A))
prec: A[i] ≤ A[i+1]≤ … ≤A[j-1]
post: A[j] è spostato in modo che A[i] ≤ A[i+1]≤ … ≤A[j]

```
InsertionSort(A)
n = len(A)
for (j = 1;j < n; j++)
In questo ciclo gli elementi di
A[0:j-1] sono una permutazione ordinata
crescente  di quelli iniziali,  A[0] ≤A[1]≤...≤A[j-1]
      Ins(A,0,j)
```

Inserimento in un array ordinato: soluzione 1

Esempio: inserimento di 80 nel posto giusto nell'array:

20	40	70	200	400	700	80
0	1	2	3	4	5	6

Basta confrontare l'elemento da inserire, k, con quelli che lo precedono nell'array, e scambiarlo nel caso risultino minori.

Possiamo usare un "trucco" per evitare di fare gli scambi, che costano 3 operazioni in generale, copiando l'elemento che si vuole inserire in una nuova variabile e usando la sua entrata nell'array per spostare gli elementi a destra

x = 80

20	40	70	80	200	400	700
0	1	2	3	4	5	6

Inserimento in un array ordinato: soluzione 1

Esempio: inserimento di 80 nel posto giusto nell'array:

20	40	70	200	400	700	80
0	1	2	4	5	6	7

```
InsOrd(A,i,j)
input:A è una lista di interi, e A[j] è l'elemento da
inserire tra gli elementi A[i],...,A[j-1]
prec: A[i:j-1] è  ordinato
output: l'array A[i:j] è ordinato
x = A[j]
h = j-1
while (h ≥ i and x < A[h])
      {A[h+1] = A[h]
// spostamento di una posizione a destra
      h = h - 1 }
 A[h+1] = x
```

// Se all'uscita h = i - 1, vuol dire che il controllo x < A[h] è risultato vero per h= j-1 fino a i, quindi x viene correttamente inserito in A[i].
Se l'uscita è determinata da x ≥ A[h], allora, poichè A è ordinato, tutti quelli che precedono A[h] sono minori o uguali di A[h] e quindi di x, mentre i successivi sono più grandi. Quindi h+1 è il posto giusto per x in A.

Inserimento in un array ordinato: complessità soluzione 1

```
InsOrd(A,i,j)
input:A è una lista di interi, e A[j] è l'elemento da
inserire tra gli elementi A[i],...,A[j-1]
prec: A[i:j-1] è  ordinato
output: l'array A[i:j] è ordinato
x = A[j]
h = j-1
while (h ≥ i and x < A[h])
      {A[h+1] = A[h]
// spostamento di una posizione a destra
      h= h - 1 }
  A[h+1] = x
```

Sia n il numero di elementi di A[i:j]

1. **nel caso migliore non si entra nel *ciclo while* perché *x ≥ A[j-1],* x si trova già nella posizione giusta e allora tutta l'operazione è eseguita in Θ(1).**

2. **nel caso peggiore *x < A[i]* e allora l'operazione è eseguita n volte, quindi la complessità di tempo nel caso peggiore su un input di dimensione n è $TMAX_{InsOrd}(n) = \Theta(n)$**

3. **in generale l'operazione su un input di n elementi è eseguita in un numero di passi superiormente limitato da una funzione lineare, $T_{InsOrd}(n) = O(n)$**

Inserimento in un array ordinato: soluzione 2

Data una lista A di interi ordinata in ordine crescente, cioè in modo tale che A[i] ≤ A[i+1], per 0≤i<n-1, vogliamo inserire un elemento, in modo da ottenere ancora una lista di interi ordinata crescente.

Possiamo usare l'algoritmo della ricerca binaria o dicotomica, per determinare la posizione di inserimento dell'elemento.
Una volta determinata la corretta posizione, i, possiamo inserire l'elemento spostando a destra di una posizione tutti gli elementi dall'i-simo all'ultimo.

RBisect: pseudocodice

RBisect(A,i,j,k)

INPUT: una lista A di interi, l'indice di inizio e fine della porzione di array d esaminare e l'elemento da cercare, k.

PREC: A[i]≤A[i+1]≤...≤A[*j*-1],

OUTPUT: restituisce la posizione nella quale inserire k se non occorre nella sequenza, l'indice dell'elemento immediatamente a destra dell'ultima occorrenza di k altrimenti.

1. finchè i < j

%nel cicli che segue l'intervallo della ricerca è sempre determinato dagli indici i,…,j-1

2. m = i + (j-i)/2

3. se k < A[m] poni j=m

4. se k ≥ A[m] poni i=m+1

5. return i

RBisect: analisi

RBisect(A,i,j,k)

INPUT: una lista A di interi, l'indice di inizio e fine della porzione di array d esaminare e l'elemento da cercare, k.

PREC: A[i]≤A[i+1]≤...≤A[*j*-1],

OUTPUT: restituisce la posizione nella quale inserire k se non occorre nella sequenza, l'indice dell'elemento immediatamente a destra dell'ultima occorrenza di k altrimenti.

1. finchè i < j

%nel cici che segue l'intervali della ricerca è sempre determinato dagli indici i,…,j-1

2. m = i + (j-i)/2

3. se k < A[m] poni j=m

4. se k ≥ A[m] poni i=m+1

5. return i

$T_{RBisect}(n) = TMAX_{RBisect}(n) = \Theta(\lg n)$

Qui infatti il numero di esecuzioni del ciclo è lo stesso su ogni input di n elementi.

InsOrd2

```
InsOrd2(A,i,j)
input:A è una lista di interi, e A[j] è l'elemento da inserire tra gli elementi
A[i],...,A[j-1]
prec: A[i:j-1] è  ordinato
output: l'array A[i:j] è ordinato

h = RBisect(A,i,j,A[j])
d = j-1
x = A[j]
while d ≥ h
        {A[d+1] = A[d]
 // spostamento di una posizione a destra
         d = d-1 }
 A[h] = x
```

Il tempo di esecuzione asintotico della ricerca della posizione in cui inserire l'elemento è $T_{RBisect}(n) = \Theta(\lg n)$

Nel caso peggiore, cioè quando inserisco al posto del primo elemento, per l' operazione di spostamento a destra e inserimento si ha $\Theta(n)$.

Caso peggiore: $TMAX_{InsOrd2}(n) = \Theta(\lg n) + \Theta(n) = \Theta(n)$,
sempre chiamando n il numero degli elementi della porzione di array considerata.
Caso migliore: $TMIN_{InsOrd2}(n) = \Theta(\lg n) + \Theta(1) = \Theta(\lg n)$,

Confronto di complessità

Abbiamo determinato che

$$TMAX_{insOrd2}(n) = T_{RBisect}(n) + TMAX_{insPos}(n) = \Theta(n) + \Theta(\lg n) = \Theta(n)$$

Mentre per la prima soluzione sempre nel caso peggiore:

$$TMAX_{InsOrd}(n) = \Theta(n)$$

Siamo ora in grado di decidere quale dei due algoritmi è meglio?

Nel caso peggiore, con InsOrd facciamo n confronti e n assegnamenti per "fare spazio", mentre con InsOrd2 facciamo lg n confronti e n spostamenti per "fare spazio".

Se i confronti sono molto costosi e n è molto grande è preferibile l'algoritmo che ne fa di meno, cioè quello che usa la ricerca binaria, altrimenti meglio il più semplice.

Ins(A,0,j)

Abbiamo visto due algoritmi diversi per realizzare l'inserimento, entrambi di complessità di tempo asintotica lineare:

1. In InsOrd, si fanno scorrere gli elementi maggiori di quello da inserire di una posizione a destra fino a che si trova un elemento minore o uguale. Complessità Θ(n) nel caso peggiore, Θ(n) confronti e Θ(n) assegnamenti

2. Il secondo algoritmo, InsOrd2, usa RBISECT per individuare la giusta posizione per l'inserimento e poi esegue gli spostamenti degli elementi a destra di questa posizione. Complessità Θ(n) nel caso peggiore, Θ(lg n) confronti e Θ(n) assegnamenti

InsertionSort: con InsOrd

```
InsertionSort(A)
n = len(A)                                   Θ(1)
 for (j = 1;j < n; j++)                      Eseguito n volte
In questo cici gli elementi di
A[0:j-1] sono una permutazione
ordinata crescente  di quelli iniziali,
A[0] ≤A[1]≤...≤A[j-1]
      InsOrd(A,0,j)                          Tra Θ(1) e jΘ(1)
```

Nel caso peggiore si tratta di sommare $j\Theta(1)$, per j =1 fino a n-1, ottenendo $\Theta(n(n-1)/2)) = \Theta(n^2)$

Nel caso migliore si tratta di sommare $\Theta(1)$ n-1 volte, ottenendo $\Theta(n)$

InsertionSort2, con InsSort2

Se si usa InsOrd2, il caso migliore è Θ(lg n), perchè Rbisect ha tempo di esecuzione Θ(lg n) e l'inserimento in un array, data la posizione, nel caso migliore è eseguito in tempo costante.

$TMAX_{InsSort2}(n) = \Theta(n^2)$, caso peggiore

$T_{InsSort2}(n) = O(n^2)$ e $T_{InsSort2}(n) = \Omega(n \lg n)$, in tutti i casi.

Sostituzione codice: insertionSort standard

```
InsertionSort(A)
n = len(A)
for (j = 1;j < n; j++)
      InsOrd(A,0,j)
```

più leggibile

```
InsOrd(A,i,j)
x = A[j]
h = j-1
while (h ≥ i and x < A[h])
      {A[h+1] = A[h]
// spostamento di una posizione a
destra
      h = h - 1 }
 A[h+1] = x
```

più veloce

```
InsertionSort(A)
n = len(A)
for (j = 1; j < n; j++)
%in ogni esecuzione del cici gli elementi di A[0:j-1]
sono una permutazione ordinata crescente
di quelli iniziali,  A[0] ≤A[1]≤...≤A[j-1]
     x = A[j]
     i = j-1
     while i ≥ 0 and x < A[i] do
           A[i+1] = A[i]
           i = i - 1
 A[i+1] = x
```

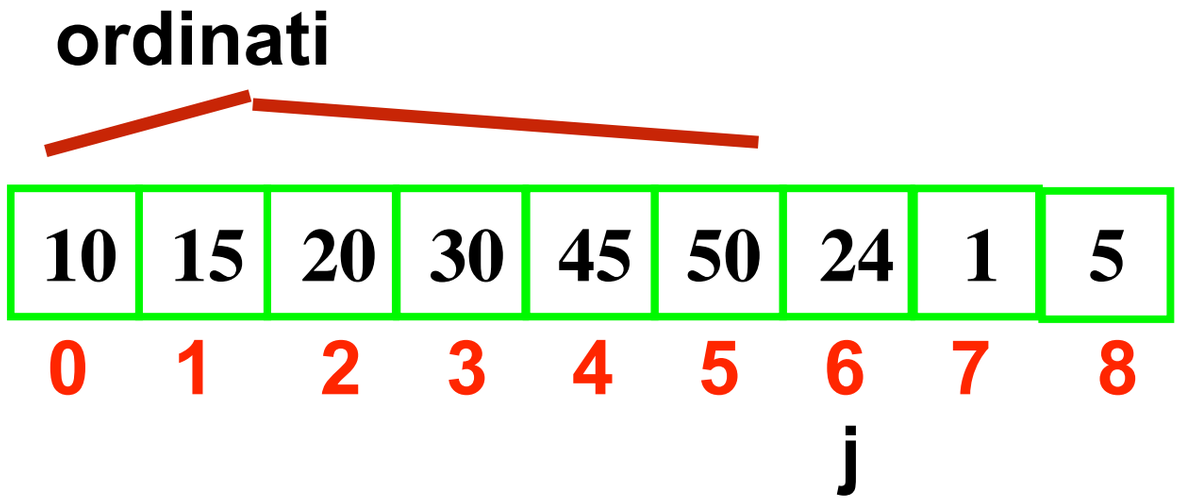
Progettazione di un algoritmo di ordinamento 2

Ritorniamo al problema dell'ordinamento (caso crescente)

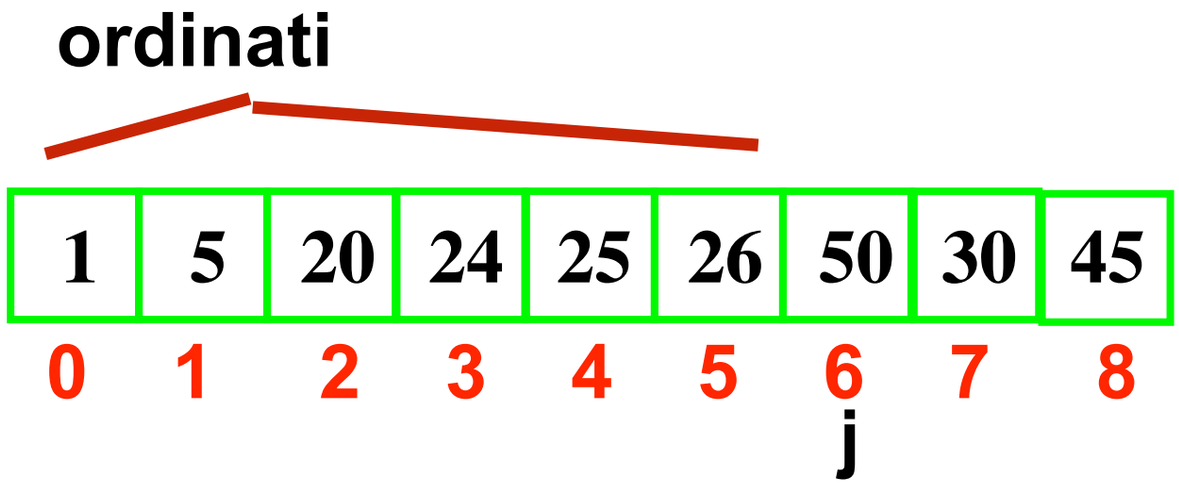
l'input è una lista di numeri (interi), A[0:n-1],
la postcondizione (o output) è: gli elementi di A[0:n-1] sono una permutazione ordinata crescente di quelli iniziali:

$$A[0] \leq A[1] \leq \ldots \leq A[n-1].$$

Come prima, supponiamo di arrivare alla postcondizione incrementalmente, estendendo in ogni passo l'ordinamento a un nuovo elemento, utilizzando un indice j per scorrere il vettore a partire da 0.



Verso un altro algoritmo



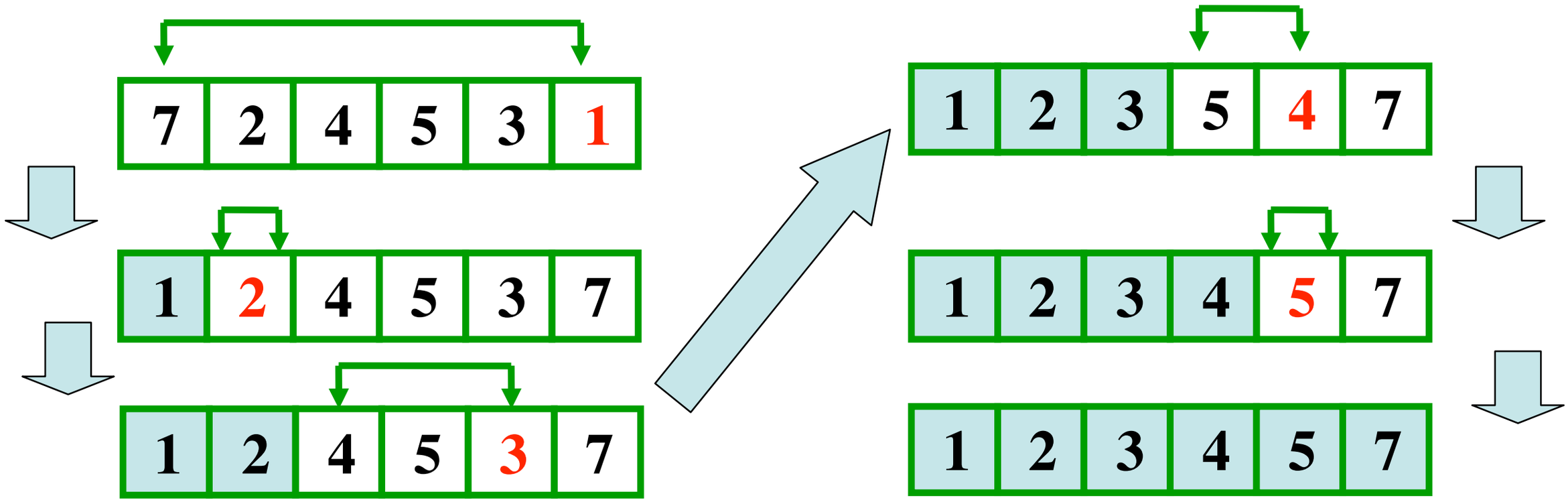
Si può estendere l'ordinamento già ottenuto su A[0:j-1] a A[0:j] in un altro modo?
Osserviamo che se i primi j elementi sono già ordinati e sono anche più piccoli dei rimanenti n - j, allora si potrebbe estendere l'ordinamento prendendo il minimo dei rimanenti e spostandolo nella posizione j

Progettazione del selectionSort

All'inizio si calcolerà il minimo in A[0:n-1] e lo si scambierà con l'elemento nella posizione 0, procedendo poi con il minimo in A[1:n-1], …

Così facendo avremo che l'ipotesi che gli elementi già ordinati siano più piccoli di quelli ancora da ordinare viene rispettata, legittimando il procedimento.

Un' esecuzione del selectionSort



Osserviamo che l'ultimo minimo da calcolare è tra A[n-2] e A[n-1].

L'algoritmo

La funzione Min è qui specificata:

Min(A,i,j)

Input: A è una lista di interi e i e j sono indici in [0,len(A))
Output: restituisce l'indice del minimo tra A[i],…,A[j]

```
SelSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
% gli elementi in A[0,j-1] sono
ordinati e minori degli elementi in
A[j,n-1]. In ogni passo il minimo tra
gli elementi A[j],…,A[n-1] va in A[j]
      minInd = Min(A,j,n-1)
      scambia A[j] con A[minInd]
```

Complessità

```
SelSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
% in ogni passo il minimo  tra gli elementi A[j],
…,A[n-1] va in A[j]
      minInd = Min(A,j,n-1)        Θ(n-j)
      scambia A[j] con A[minInd]   Θ(1)
```

La funzione Min è chiamata la prima volta su n elementi, poi n-1, e così via decrescendo, quindi complessivamente il numero di queste chiamate è la somma dei primi n interi.

Quindi, nel caso peggiore SelectionSort ha una complessità pari a $\Theta(n^2)$.

Concludiamo che $TMAX_{SelSort}(n) = \Theta(n^2)$.

Ma anche che $T_{SelSort}(n) = \Theta(n^2)$ visto che il minimo si calcola in ogni caso in $\Theta(n - j)$.

Algoritmo per il minimo

```
Min(A,i,j)
Input A è una lista di interi i e  j sono indici
preCond:i e j sono compresi tra 0 il numero
degli elementi di A.
Output: restituisce l'indice del  minimo  tra A[i],
…,A[j]
minInd = i
for (k = i+1;k ≤ j; k++)
%minInd è l'indice del minimo tra
gli elementi già esaminati A[i],…,A[k-1]
     if A[k] < A[minInd ]
     then minInd = k
return minInd
```

Sostituzione codice: selectionSort standard

```
SelSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
        minInd = Min(A,j,n-1)
        scambia A[j] con A[minInd]
```

```
Min(A,i,j)
minInd = i
for (k = i+1;k ≤ j; k++)
     if A[k] < A[minInd ]
     then minInd = k
return minInd
```

più leggibile

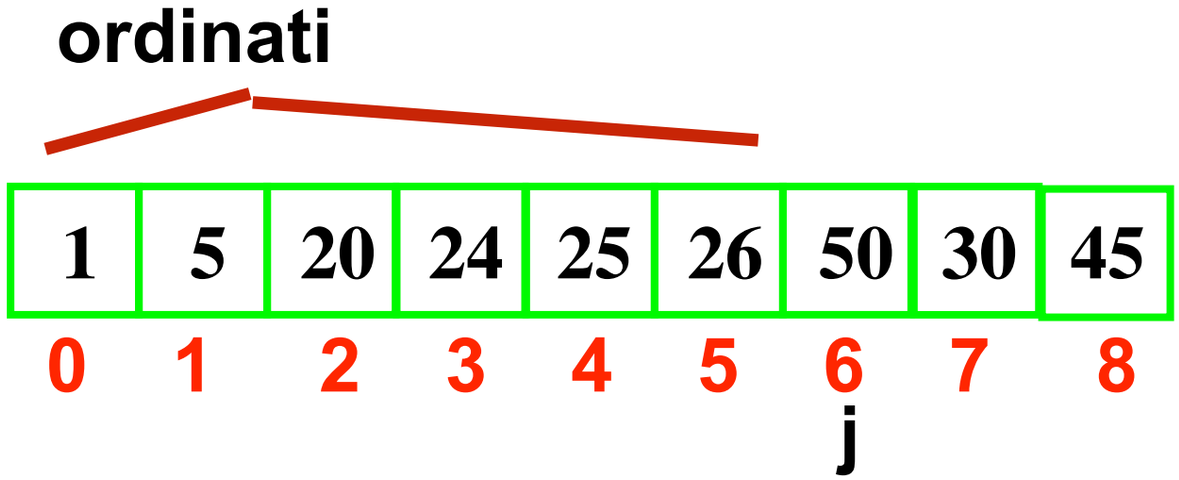
più veloce

```
SelSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
        minInd = j
        for (k = j+1;k ≤ n-1; k++)
              if A[k] < A[minInd ]   then minInd = k
        scambia A[j] con A[minInd]
```

Progettazione di un algoritmo di ordinamento

Ritorniamo al problema dell'ordinamento

l'input è una lista di numeri (interi), A[0:n-1],
la postcondizione è: gli elementi di A[0:n-1] sono una permutazione ordinata crescente (per esempio) di quelli iniziali: A[0] ≤A[1]≤…≤A[n-1].



Possiamo estendere l'ordinamento già ottenuto su A[0:j-1] a A[0:j] in ancora un altro modo?

Progettazione del BubbleSort

L'idea è la stessa del Selection Sort, ma usando un'altra funzione per il calcolo del minimo.

L'idea è ancora quella di estendere l'ordinamento già ottenuto su A[0:j-1] spostando nella posizione j il minimo tra gli elementi ancora da ordinare.

La differenza consiste nel fatto che si scambiano elementi vicini, nel caso quello a destra sia minore.

In generale si procede spostando nella posizione j il minimo tra gli elementi compresi tra quelli nella j-sima posizione e l'ultima, per tutti i valori di j=0,…,n-1.

Calcoi e spostamente del minimo

Scriviamo una funzione, Min2(A,i,j), che realizza i spostamento del minimo, tra gli elementi nella porzione di lista tra i e j, nella posizione i, in modo alternativo a quello visto.

Lo spostamento avviene esaminando l'array dalla fine destra e scambiando il più piccolo tra due elementi successivi.

Poiché di volta in volta si sposta il più piccolo incontrato questo sarà il minimo tra gli elementi già esaminati.

Il risultato dell'esecuzione di Min2(A,i,j) è allora A[i] = min{A[i],...,A[j]} e gli elementi di A sono una permutazione di quelli iniziali.

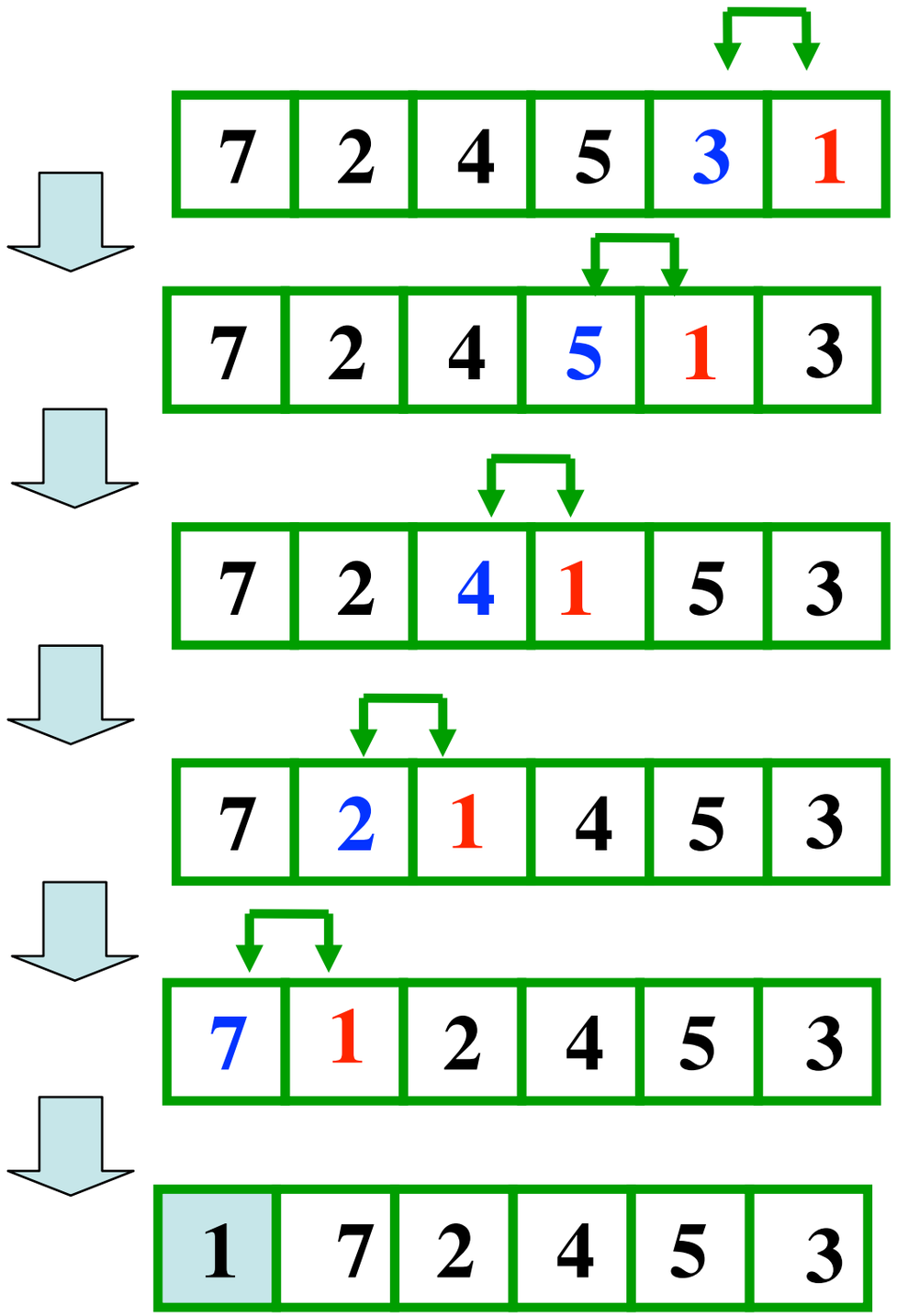
Calcolo e spostamente del minimo

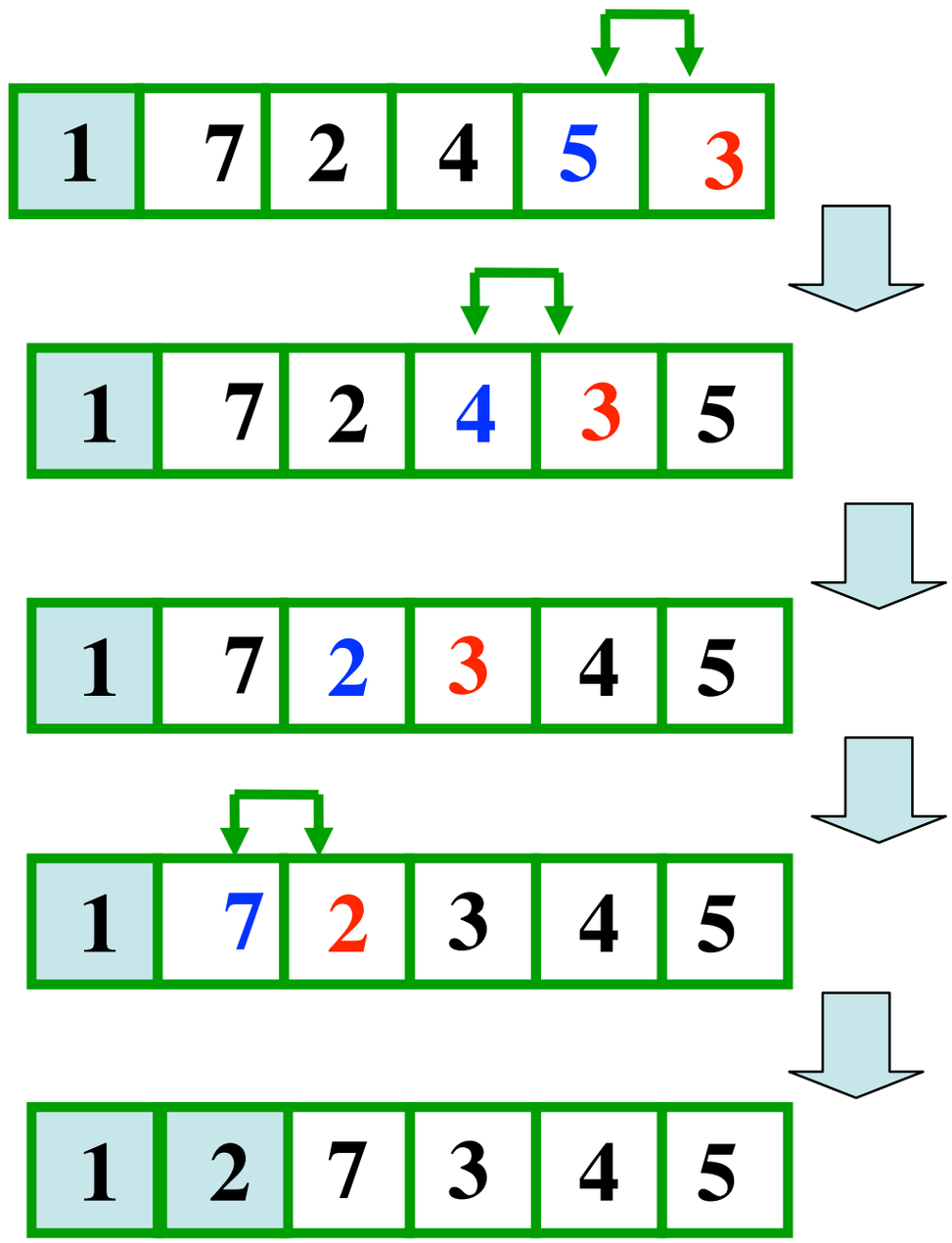
min2(A,i,j)
input: A è una lista di interi, i e j in [0,len(A))
output: A[i] = min{A[i],…,A[j]} e gli elementi A[i], …,A[j] sono una permutazione degli elementi presenti all'inizio del ciclo

```
for k = j downto i+1 do
        if A[k] < A[k-1] then
        scambia A[k] e A[k-1]
```

Il tempo di esecuzione di min2(A,i.j) è Θ(j-i).

Una parziale esecuzione del BubbleSort





E così via…

sostituzione del codice: bubbleSort standard

```
BubbleSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
      min2(A,j,n-1)
```

```
min2(A,i,j)
for k = j downto i+1 do
      if A[k] < A[k-1] then
      scambia A[k] e A[k-1]
```

```
BubbleSort(A)
n = len(A)
for (j = 0; j ≤ n-2; j++)
     for k = n-1 downto j+1 do
          if A[k] < A[k-1] then  scambia A[k] e A[k-1]
```

$T_{BubbleSort}(n) = \Theta(n^2)$ visto che il minimo si calcola in ogni caso in $\Theta(n - j)$!

N.B. **Con una modifica si può rendere lineare nel caso migliore: esercizio.**

Popolarità del bubblesort

Per la sua semplicità il bubbleSort è spesso usato per introdurre al concetto di algoritmo. Ma alcuni ricercatori hanno addirittura proposto che non venga più insegnato.

Già Donald Knuth, nel suo libro The Art of Computer Programming, aveva notato che "il bubbleSort non sembra avere nulla per cui lo si debba raccomandare, eccetto un nome orecchiabile e il fatto che porta a qualche interessante problema teorico."

Il bubbleSort è qualificato come un cattivo algoritmo.

Pur essendo infatti asintoticamente equivalente all' l'insertionSort sperimentalmente mostra un comportamento molto peggiore.

Risultati sperimentali di Astrachan in Java mostrano che il bubble sort è all'incirca 5 volte più lento dell' insertionSort e 40% più lento del selectionSort.

E' molto popolare in computer grapjcs per la capacità di individuare piccoli errori (due elementi scambiati) in array quasi ordinati e di rimediare all'errore in tempo lineare.

Animazioni

Due siti dove attivare un animatore didattico

http://www.uwosh.edu/faculty_staff/naps/sos-the-sequel/demo.htm

http://www.site.uottawa.ca/~stan/csi2514/applets/sort/sort.html

Qui un film storico! Prodotto nel 1981, è una curiosità, ma è anche molto istruttivo

https://www.youtube.com/watch?v=HnQMDkUFzh4