ROMA TRE
UNIVERSITÀ DEGLI STUDI

INGEGNERIA
DIPARTIMENTO DI ECCELLENZA

Algoritmi e Strutture di Dati

Il problema dell'ordinamento
Algoritmi greedy e algoritmi iterativi

m.patrignani

Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Panoramica

- il problema dell'ordinamento
- gli algoritmi greedy
  - l'algoritmo selection sort
- gli algoritmi iterativi
  - l'algoritmo insertion sort

Il problema dell'ordinamento

- **input**: una sequenza di *n* numeri $<a_1, a_2, ..., a_n>$
- **output**: una permutazione $<a_1', a_2', ..., a_n'>$ della sequenza tale che $a_1' \leq a_2' \leq ... \leq a_n'$
- esempio
  - un'istanza

    <31, 41, 59, 26, 41, 58>
  - la soluzione dell'istanza qui sopra

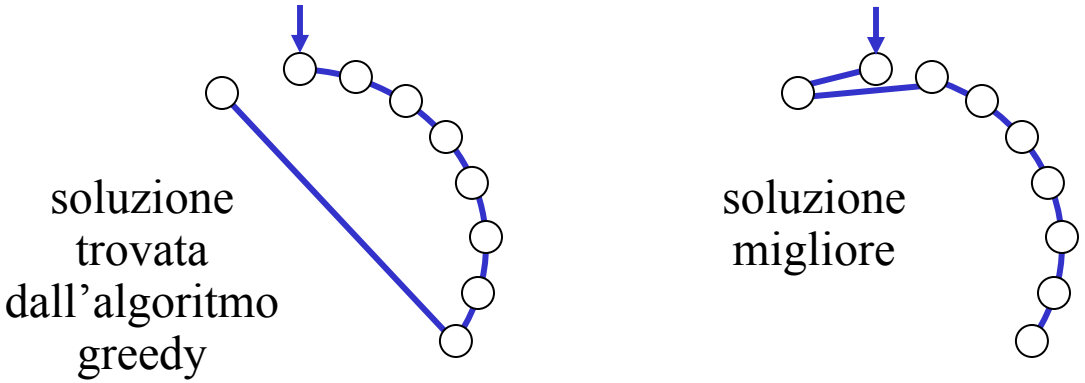
    <26, 31, 41, 41, 58, 59>
- nel seguito supporremo che l'istanza sia fornita tramite un array A con *n* posizioni

Selection sort

La tecnica greedy

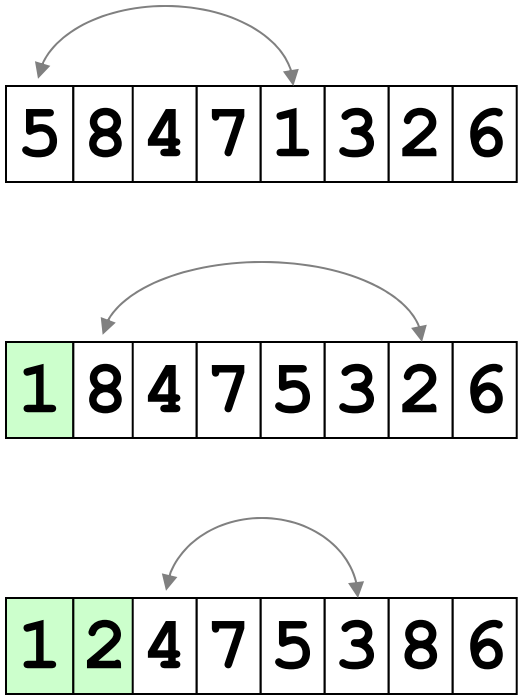
La tecnica greedy

- la tecnica greedy (golosa) consiste nel scegliere sempre l'alternativa che al momento sembra più appetibile
  - corrisponde ad eseguire una scelta localmente ottima
  - ciò non sempre comporta una scelta globalmente ottima
- esempio in cui greedy non dà la soluzione ottima
  - trovare il cammino più breve tra *n* città
  - algoritmo greedy: muoviti sempre verso la città più vicina non ancora visitata



Algoritmo selection sort

- utilizza una tecnica greedy per ordinare un array
- strategia generale
  - seleziona l'elemento più piccolo e mettilo al primo posto
  - seleziona l'elemento più piccolo dei rimanenti e mettilo al secondo posto
  - …



Algoritmo SELECTION_SORT

- pseudocodice dell'algortimo
  - si fa uso di due cicli annidati

```
SELECTION_SORT(A)
1.  for i = 0 to A.length-2
2.      min = i          ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1    ▷ scorro l'array
4.          if A[j] < A[min]           ▷ devo aggiornare min
5.              min = j
6.      temp = A[i]                    ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
```

Complessità del SELECTION_SORT

```
SELECTION_SORT(A)
1.  for i = 0 to A.length-2
2.      min = i          ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1     ▷ scorro l'array
4.          if A[j] < A[min]            ▷ devo aggiornare min
5.              min = j
6.      temp = A[i]                     ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
```

- i valori dell'input non modificano il numero delle iterazioni del ciclo esterno e del ciclo interno
  - quindi il caso migliore, il caso peggiore ed il caso medio hanno la stessa complessità

Complessità del SELECTION_SORT

```
SELECTION_SORT(A)
1.  for i = 0 to A.length-2
2.      min = i          ▷ indice elemento minimo in A[i..n-1]
3.      for j = i + 1 to A.length-1     ▷ scorro l'array
4.          if A[j] < A[min]            ▷ devo aggiornare min
5.              min = j
6.      temp = A[i]                     ▷ scambio A[i] con A[min]
7.      A[i] = A[min]
8.      A[min] = temp
```

- l'algoritmo esegue $O(n)$ cicli esterni e $O(n)$ cicli interni
  - dunque SELECTION-SORT ha complessità $O(n^2)$
- la riga 4 viene eseguita $(n\text{-}1)+(n\text{-}2)+\ldots+1 = [n(n\text{-}1)]/2$ volte
  - dunque SELECTION-SORT ha complessità $\Omega(n^2)$
- il tempo di esecuzione dell'algoritmo è $\Theta(n^2)$

Algoritmi che operano *in loco*

- gli algoritimi che operano *in loco* non necessitano di copiare l'input in strutture di dati diverse da quella utilizzata per l'input
  - questa caratteristica è utile per input di grosse dimensioni
- l'algoritmo `SELECTION_SORT` opera in loco
  - gli elementi vengono solo scambiati
  - l'algoritmo necessita di una quantità di memoria costante oltre a quella per memorizzare A
    - la memoria utilizzata dall'algoritmo è O(1)

Algoritmi di ordinamento *stabili*

- un algoritmo di ordinamento si dice *stabile* se non modifica l'ordine degli elementi che hanno lo stesso valore
  - in alcune applicazioni ciò può essere utile
    - quando agli elementi sono collegati dei dati satellite
    - quando gli elementi con la stessa chiave hanno una posizione reciproca significativa
- l'algoritmo `SELECTION_SORT` è stabile
  - se A[j] = A[k] con $j < k$, allora A[j] viene selezionato per primo

Insertion sort

Un algoritmo incrementale

Gli algoritmi incrementali

- si basano sulla seguente osservazione
  - la soluzione di un'istanza di dimensione *n*-1 può essere utile per risolvere un'istanza di dimensione *n*
- esempio

a) istanza di dimensione cinque: | 5 | 2 | 4 | 6 | 1 |

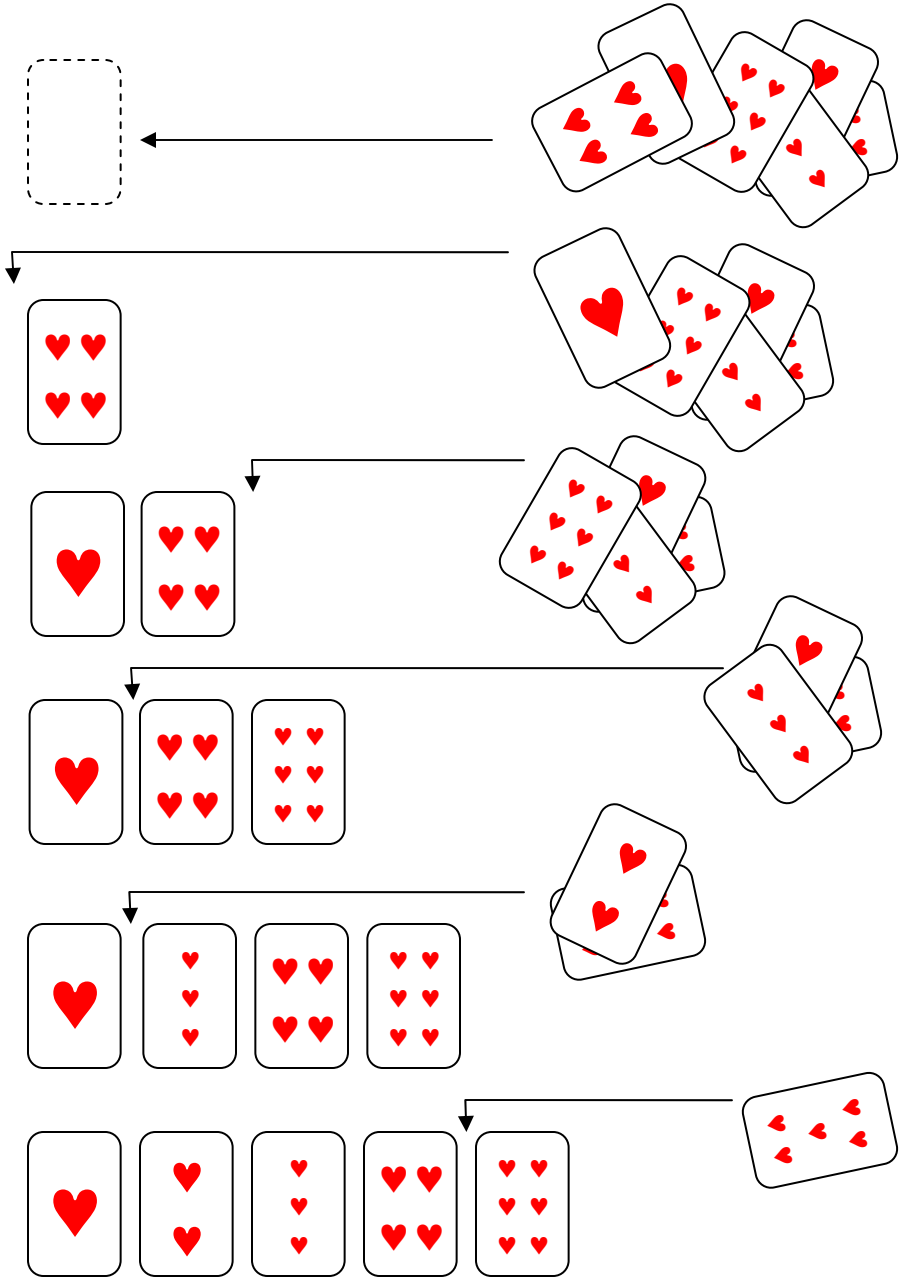
b) istanza di dimensione sei: | 5 | 2 | 4 | 6 | 1 | + | 3 |

la soluzione di (a) mi aiuta a risolvere (b) ?

| 1 | 2 | 4 | 5 | 6 | + | 3 | = | ? |

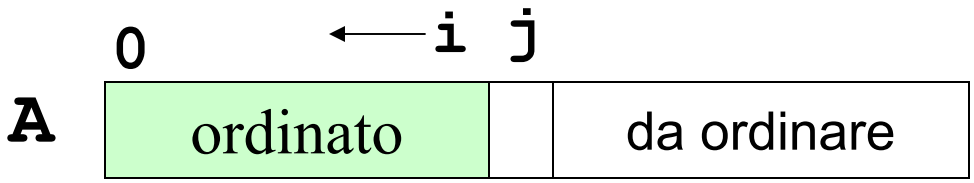
Insertion sort

- manteniamo un sottoinsieme ordinato di elementi
  - cominciando da un singolo elemento
- inseriamo un elemento alla volta
  - il sottoinsieme ordinato cresce
  - i suoi elementi vengono traslati per far posto al nuovo elemento
- quando tutti gli elementi sono inseriti il problema è risolto

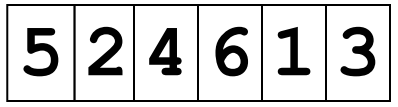


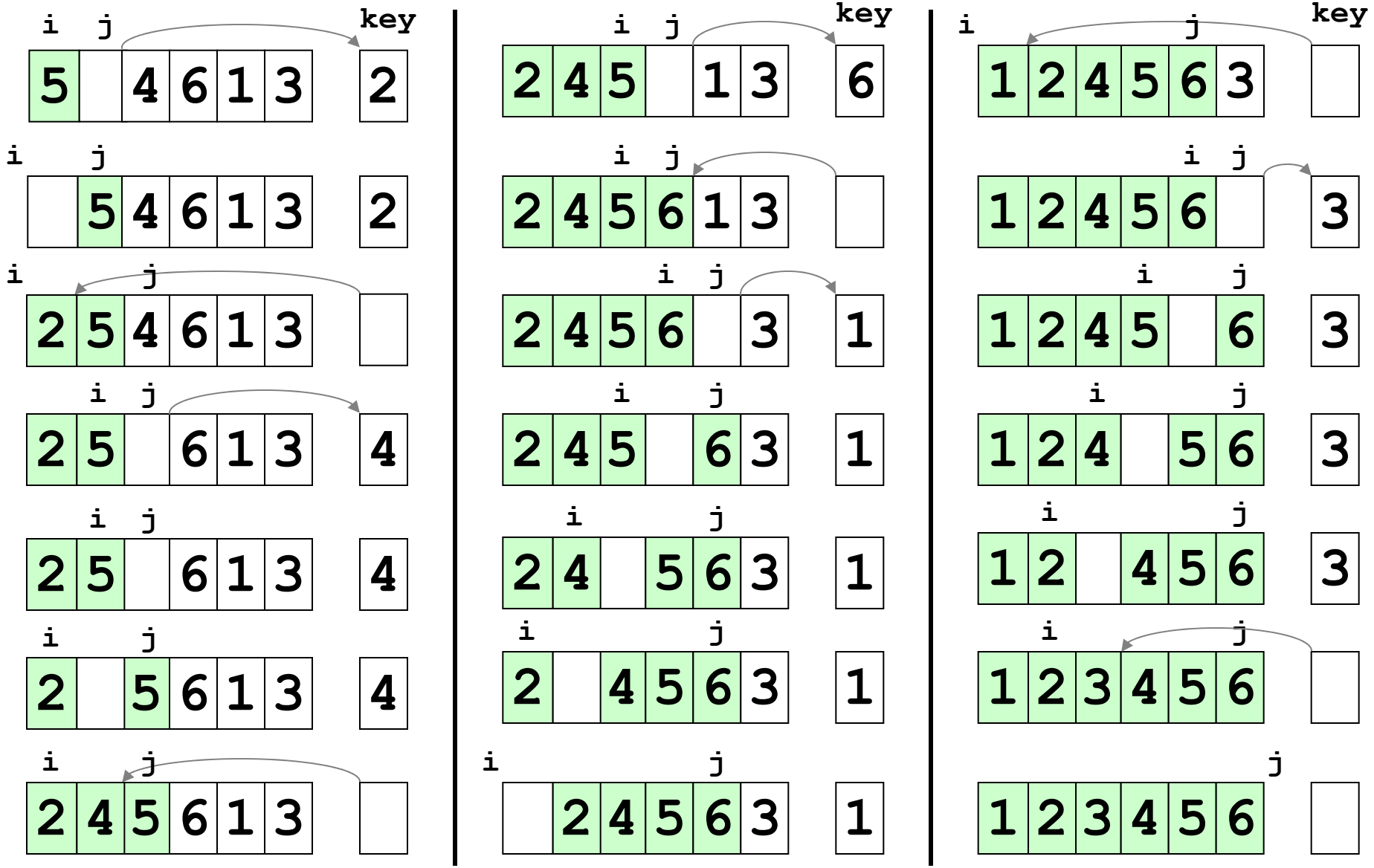
INSERTION_SORT

```
INSERTION_SORT(A)
1.  for j = 1 to A.length-1
2.          key = A[j]
3.          ▷ inserisce key nella sequenza ordinata A[0..j-1]
4.          i = j-1
5.          while i>-1 and A[i]>key
6.                  A[i+1] = A[i]
7.                  i = i-1
8.          A[i+1] = key
```



Insertion sort con input





Proprietà di INSERTION_SORT

- INSERTION_SORT è stabile
  - supponiamo che per $i < j$ si abbia A[i] = A[j]
  - quando l'algoritmo inserisce A[j], l'elemento A[i] è già stato inserito
  - l'algoritmo inserisce A[j] dopo A[i] preservando dunque la loro posizione reciproca originale
- INSERTION_SORT opera in loco
  - richiede O(1) memoria addizionale rispetto alla memoria utilizzata per l'input

Complessità nel caso peggiore

```
INSERTION_SORT(A)
1.  for j = 1 to A.length-1
2.          key = A[j]
3.          ▷ inserisce key nella sequenza ordinata A[0..j-1]
4.          i = j-1
5.          while i>-1 and A[i]>key
6.                  A[i+1] = A[i]
7.                  i = i-1
8.          A[i+1] = key
```

- il ciclo esterno viene eseguito $O(n)$ volte
- nel caso peggiore
  - l'elemento corrente deve essere inserito sempre al primo posto
    - l'array A in input è ordinato in maniera decrescente
  - il numero delle operazioni è $\Theta(n^2)$

Complessità nel caso migliore

```
INSERTION_SORT(A)
1.  for j = 1 to A.length-1
2.          key = A[j]
3.          ▷ inserisce key nella sequenza ordinata A[0..j-1]
4.          i = j-1
5.          while i>-1 and A[i]>key
6.                  A[i+1] = A[i]
7.                  i = i-1
8.          A[i+1] = key
```

- il ciclo esterno viene eseguito comunque $O(n)$ volte
- nel caso migliore
  - l'elemento corrente è già posizionato al punto giusto
    - l'array A in input è già ordinato
  - il numero delle operazioni è $\Theta(n)$

Complessità nel caso medio

```
INSERTION_SORT(A)
1.  for j = 1 to A.length-1
2.          key = A[j]
3.          ▷ inserisce key nella sequenza ordinata A[0..j-1]
4.          i = j-1
5.          while i>-1 and A[i]>key
6.                  A[i+1] = A[i]
7.                  i = i-1
8.          A[i+1] = key
```

- il ciclo esterno viene eseguito comunque $O(n)$ volte
- nel caso medio
  - l'elemento corrente va posizionato nel mezzo di $A[0\ldots j]$
  - il numero delle operazioni è $\Theta(n^2)$

Proprietà dell'insertion sort

- efficente su piccole istanze
  - più efficiente in pratica che il selection sort
  - nel caso migliore ha complessità lineare
  - si può calcolare che la complessità media è $n^2/4$
- adattivo
  - veloce su instanze già parzialmente ordinate
    - complessità $O(n + d)$ dove d è il numero delle inversioni
- stabile
  - non cambia l'ordine degli elementi che hanno lo stesso valore
- in loco
  - la memoria addizionale richiesta è $O(1)$
- online
  - può essere utilizzato quando i numeri arrivano uno alla volta

Algoritmi di ordinamento visti finora

	caso migliore	caso medio	caso peggiore	in loco	stabile
SELECTION-SORT	$\Theta(n^2)$			*si*	*si*
INSERTION-SORT	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	*si*	*si*