Algoritmi e Strutture Dati

Capitolo 4
Ordinamento

Ordinamento

Dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S

- Esempi: ordinare una lista di nomi alfabeticamente, o un insieme di numeri, o un insieme di compiti d'esame in base al cognome dello studente

- Subroutine in molti problemi
- È possibile effettuare ricerche in array ordinati in tempo $O(\log n)$

Il problema dell'ordinamento

- **Input:** una sequenza di *n* numeri $<a_1,a_2,\ldots,a_n>$
- **Output:** una permutazione (riarrangiamento) $<a_1',a_2',\ldots,a_n'>$ della sequenza di input tale che $a_1' \leq a_2' \leq \ldots \leq a_n'$

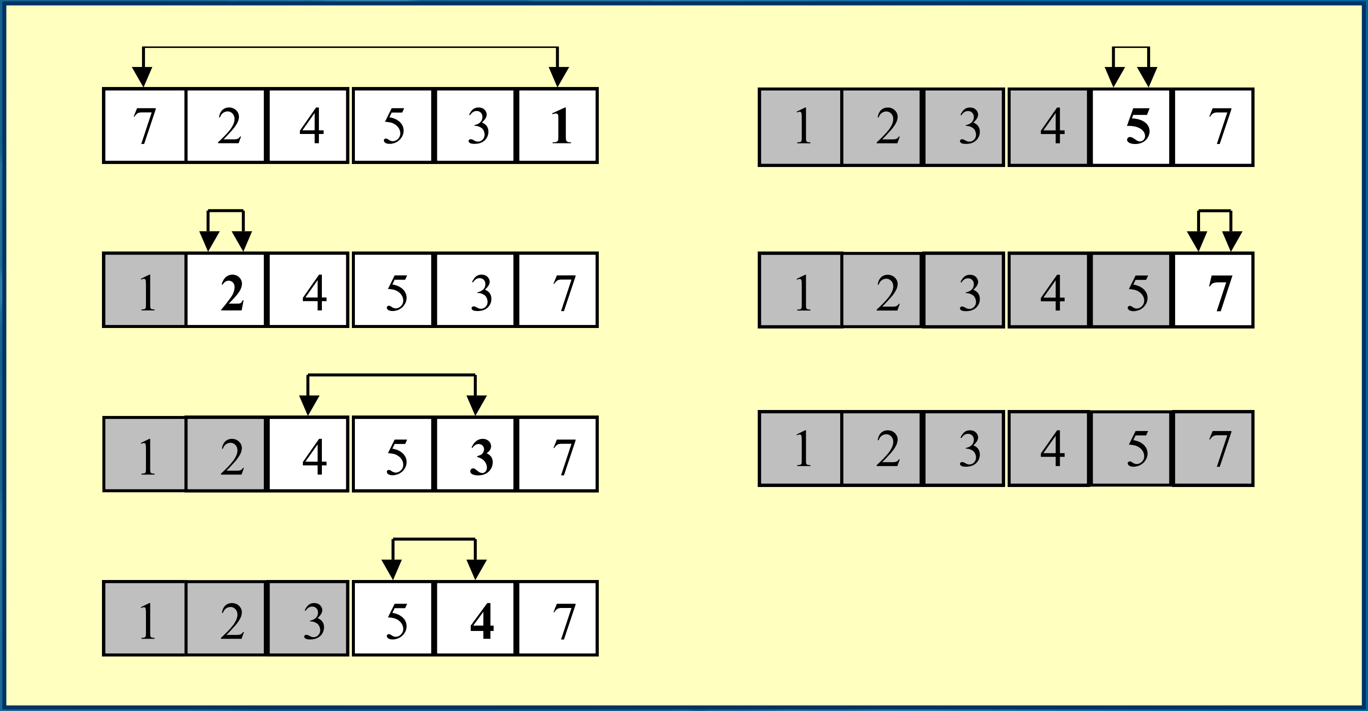
Ordinare in tempo quadratico

Un algoritmo semplice,
intuitivo, facile da
programmare.

E inefficiente.

SelectionSort

Approccio incrementale: estende l’ordinamento da k a k+1 elementi, scegliendo il minimo degli n-k elementi non ancora ordinati e mettendolo in posizione k+1



```
SelectionSort (A)
1.    for k=0 to n-2 do
2.        m = k+1
3.        for j=k+2 to n do
4.            if (A[j] < A[m]) then m=j
5.        scambia A[m] con A[k+1]
```

- al generico passo k, A[1],…,A[k] sono già ordinati
- linee 2-4: ricerca del minimo fra gli elementi A[k+1],…,A[n]
- m è l’indice dell’array in cui si trova il minimo
- il minimo è messo in posizione k+1

Corretto?

- E’ facile convincersi che l’algoritmo mantiene le seguenti *invarianti*: dopo il generico passo k (k=0,…,n-2) abbiamo che:
  - (i) i primi k+1 elementi sono ordinati e
  - (ii) sono i k+1 elementi più piccoli dell’array

Suggerimento: ragionare per invarianti è uno strumento utile per dimostrare la correttezza di un algoritmo, perché permette di isolare proprietà dell’algoritmo, spiegarne il funzionamento, capire a fondo l’idea su cui si basa.

Complessità temporale (analisi)

$T(n) =$ #operazioni elementari sul modello RAM a costi uniformi eseguite dall'algoritmo nel caso peggiore su istanze di dimensione n.

Complessità: un upper bound

```
SelectionSort (A)
    for k=0 to n-2 do
        m = k+1
        for j=k+2 to n do
            if (A[j] < A[m]) then m=j
        scambia A[m] con A[k+1]
```

eseguite al più n volte per ogni ciclo esterno

ciclo esterno eseguito al più n volte

ogni linea di codice costa tempo O(1)

$$T(n) \leq 5\, n^2\, O(1) = \Theta(n^2) \quad \Rightarrow \quad T(n) = O(n^2)$$

L'analisi è stretta? Cioè, T(n) è $\Theta(n^2)$?

Complessità: un lower bound

```
SelectionSort (A)
1.  for k=0 to n-2 do
2.      m = k+1
3.      for j=k+2 to n do
4.          if (A[j] < A[m]) then m=j
5.      scambia A[m] con A[k+1]
```

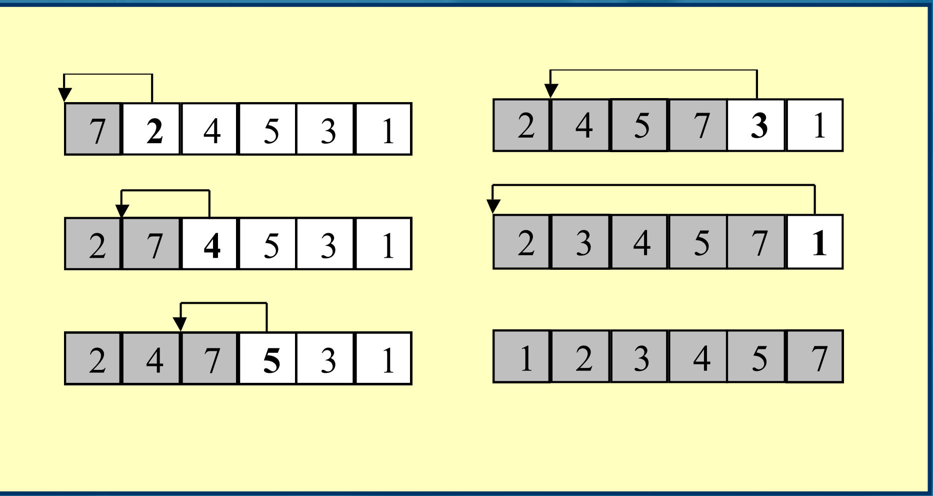
$\}$ n-k-1 confronti

Idea: conto solo i confronti fra elementi

$$T(n) \geq \sum_{k=0}^{n-2} (\text{n-}k\text{-1}) = \sum_{k=1}^{n-1} k = \text{n(n-1)/2} = \Theta\,(n^2)$$

$\Rightarrow$ $T(n) = \Omega(n^2)$ $\Rightarrow$ $\boxed{T\,(n) = \Theta\,(n^2)}$

altri algoritmi di ordinamento con tempo $O(n^2)$

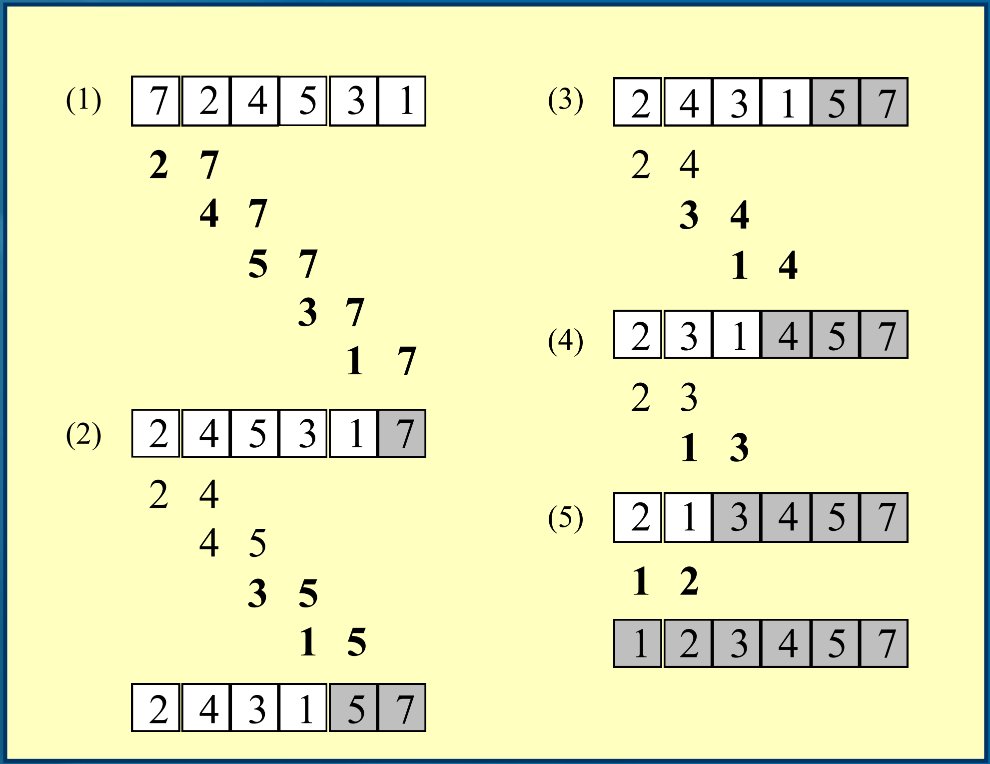


Insertion Sort

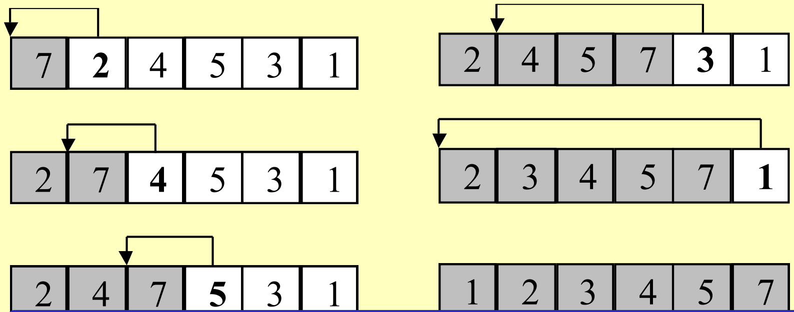
Approccio incrementale: estende l'ordinamento da k a k+1 elementi, posizionando l'elemento (k+1)-esimo nella posizione corretta rispetto ai primi k elementi

Bubble Sort

Approccio incrementale: esegue n-1 scansioni. Ad ogni scansione guarda coppie di elementi adiacenti e li scambia se non sono nell'ordine corretto.



Insertion Sort



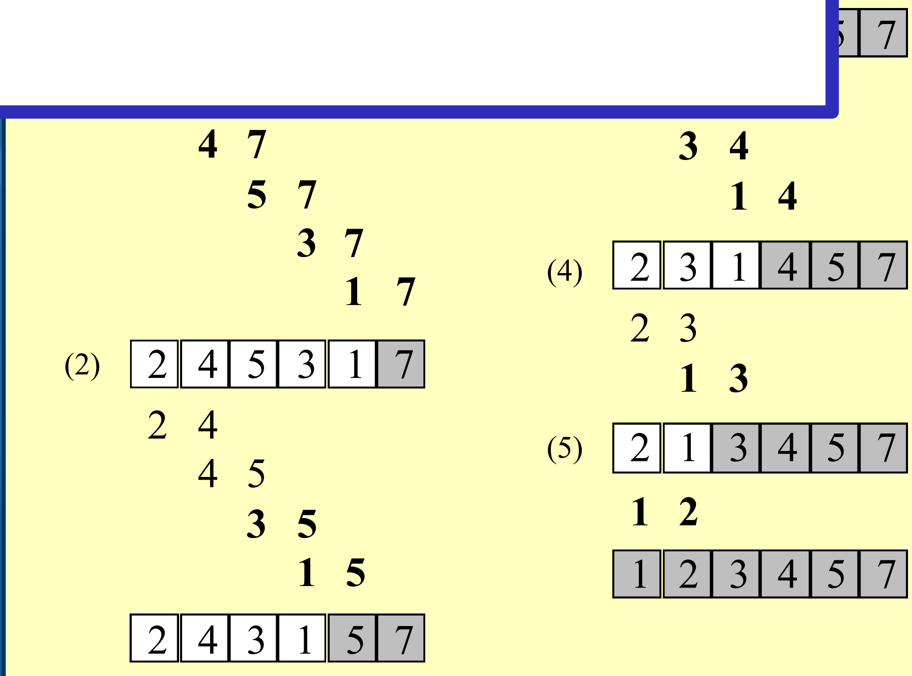
Approccio incrementale: estende l'ordinamento da k a k+1 elementi, posizionando l'elemento (k+1)-

... a

Esercizio

Scrivere lo pseudocodice dei due algoritmi e fare l'analisi della complessità temporale nel caso peggiore.

Approccio incrementale: esegue n-1 scansioni. Ad ogni scansione guarda coppie di elementi adiacenti e li scambia se non sono nell'ordine corretto.



Ordinare in tempo meno che quadratico

Un algoritmo semplice, un po' meno intuitivo, facile da programmare.

E temporalmente efficiente.

Tecnica: Divite et Impera

MergeSort

- Usa la tecnica del divide et impera:

1 Divide: dividi l'array a metà

2 Risolvi i due sottoproblemi ricorsivamente

3 Impera: fondi le due sottosequenze ordinate

```
MergeSort (A, i, f)
1.    if (i < f) then
2.         m = ⌊(i+f)/2⌋
3.         MergeSort(A,i,m)
4.         MergeSort(A,m+1,f)
5.         Merge(A,i,m,f)
```

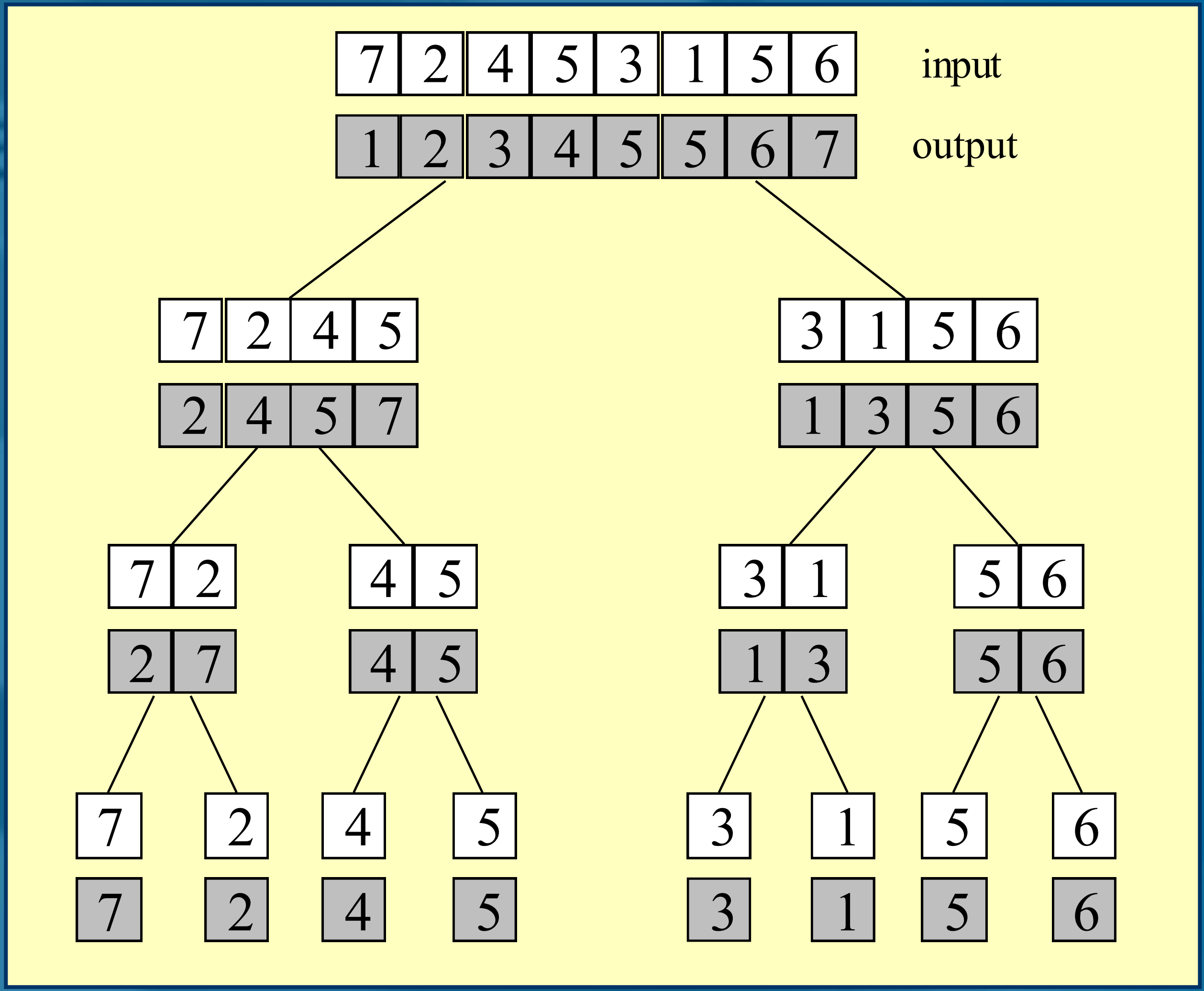
ordina A[i;f]

chiamata iniziale:
MergeSort(A,1,n)

fonde A[i;m] e A[m+1;f]
output in A[i;f]

Notazione: dato un array A e due indici $x \leq y$, denotiamo con A[x;y] la porzione di A costituita da A[x], A[x+1],…,A[y]

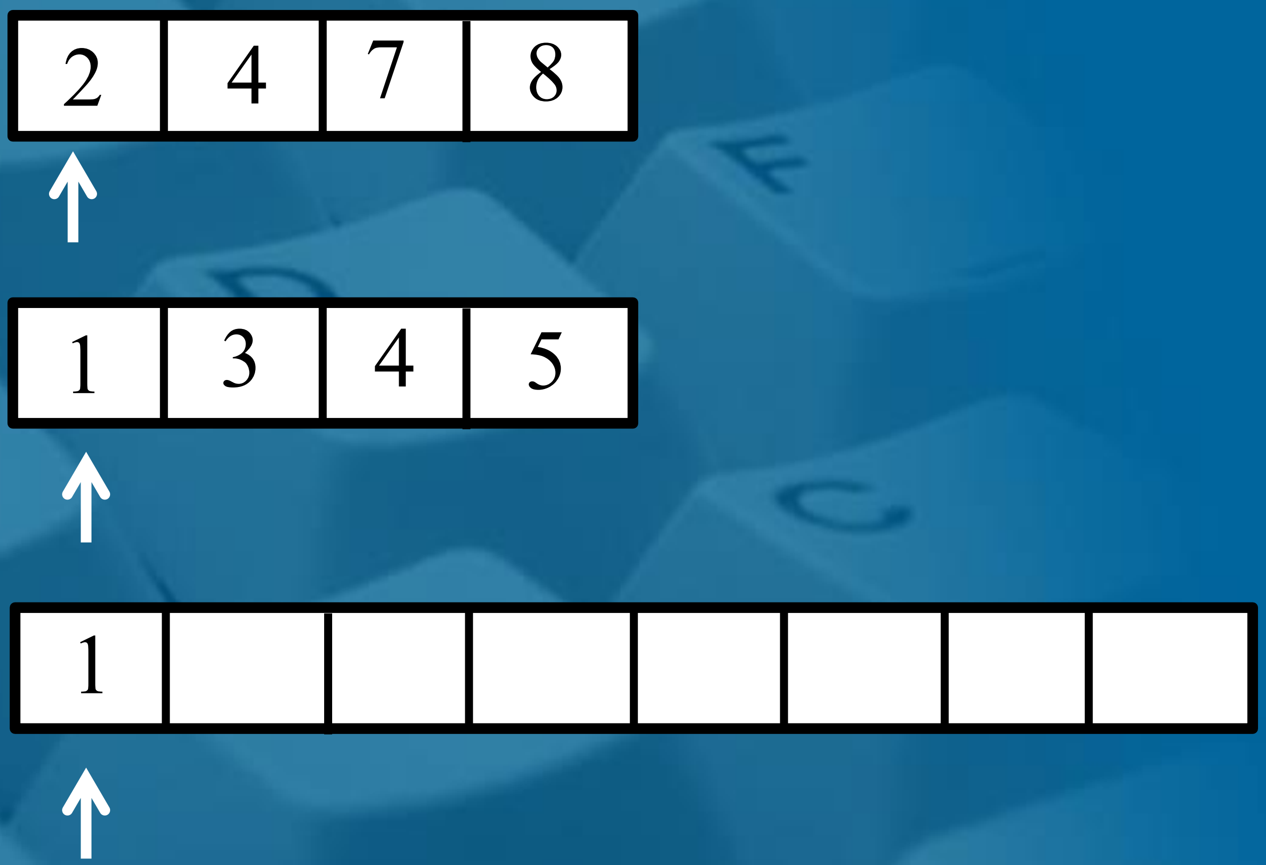
Esempio di esecuzione

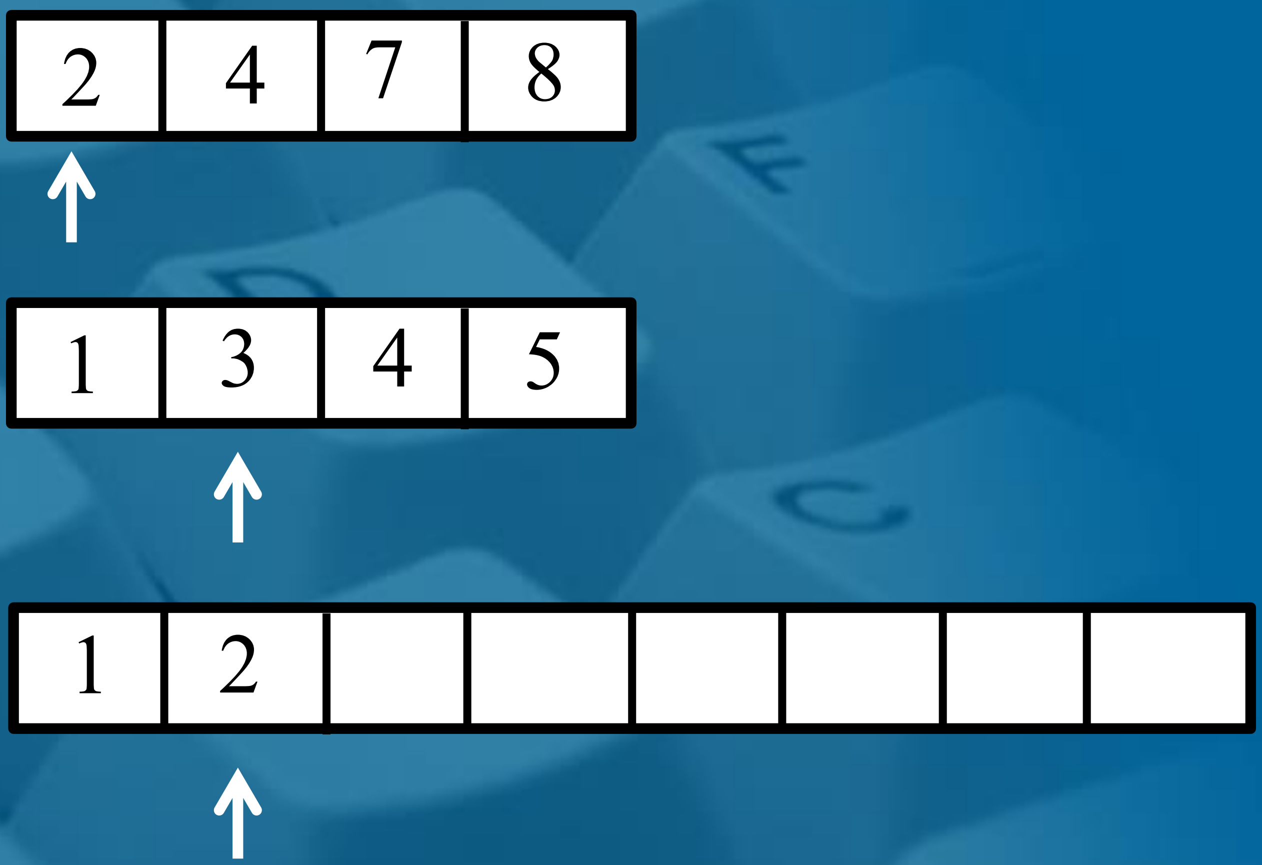


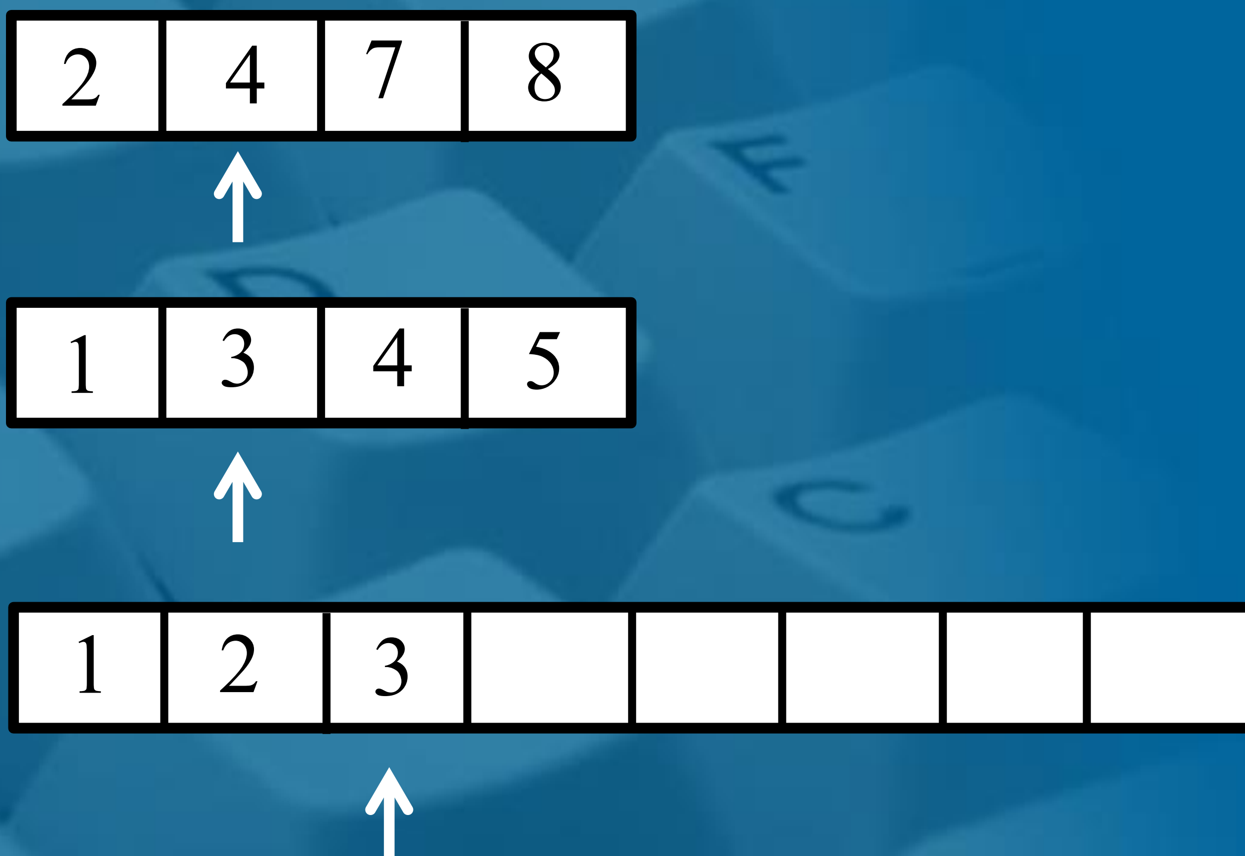
Albero delle chiamate ricorsive

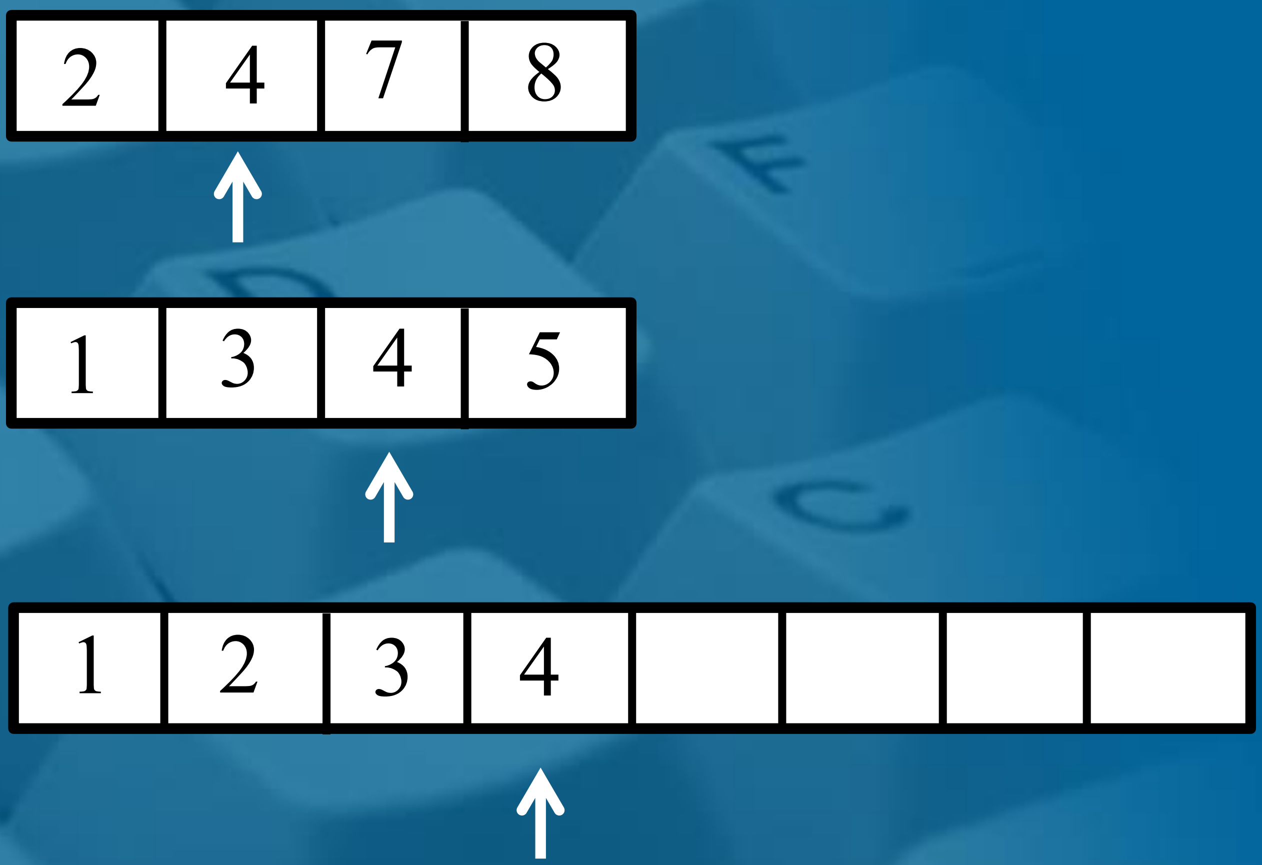
Procedura Merge

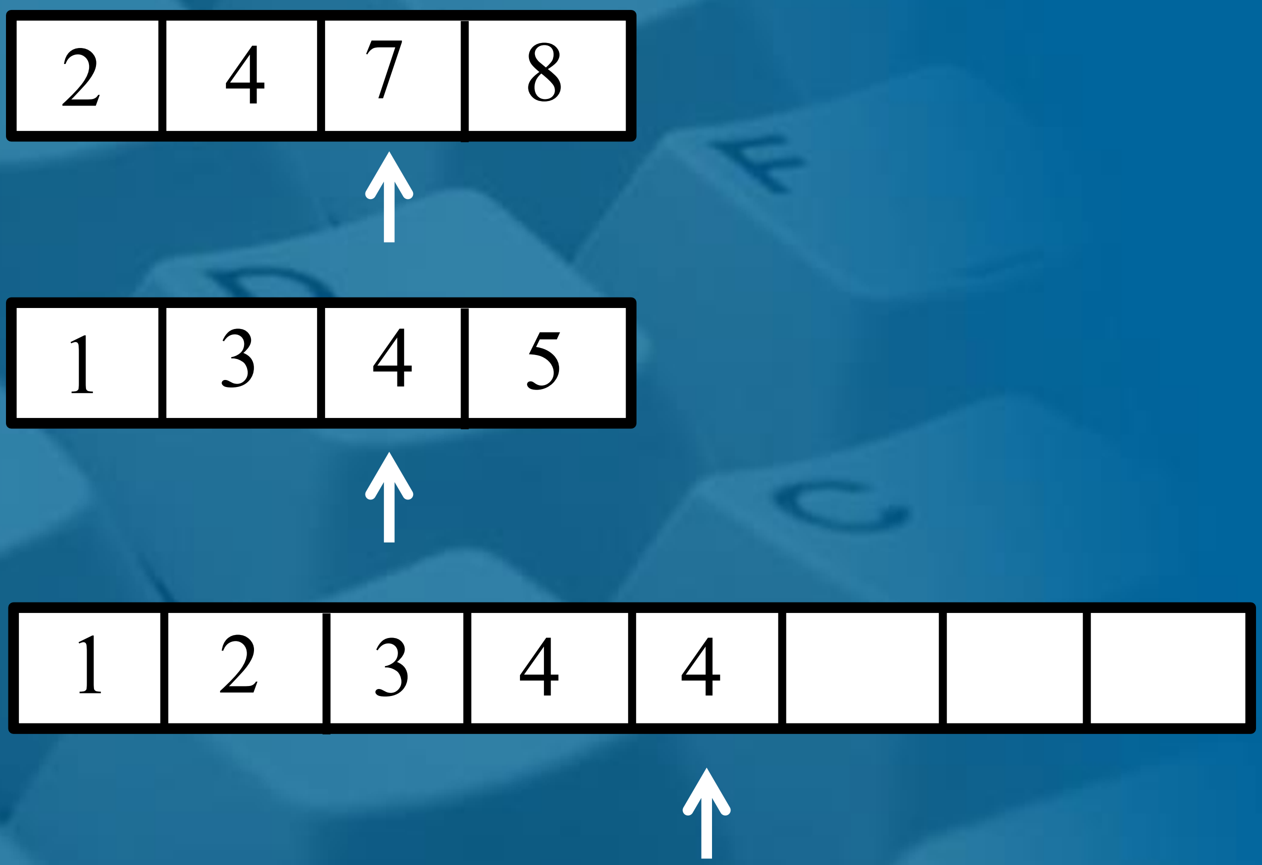
- Due array ordinati A e B possono essere fusi rapidamente:
  - estrai ripetutamente il minimo di A e B e copialo nell'array di output, finché A oppure B non diventa vuoto
  - copia gli elementi dell'array non vuoto alla fine dell'array di output

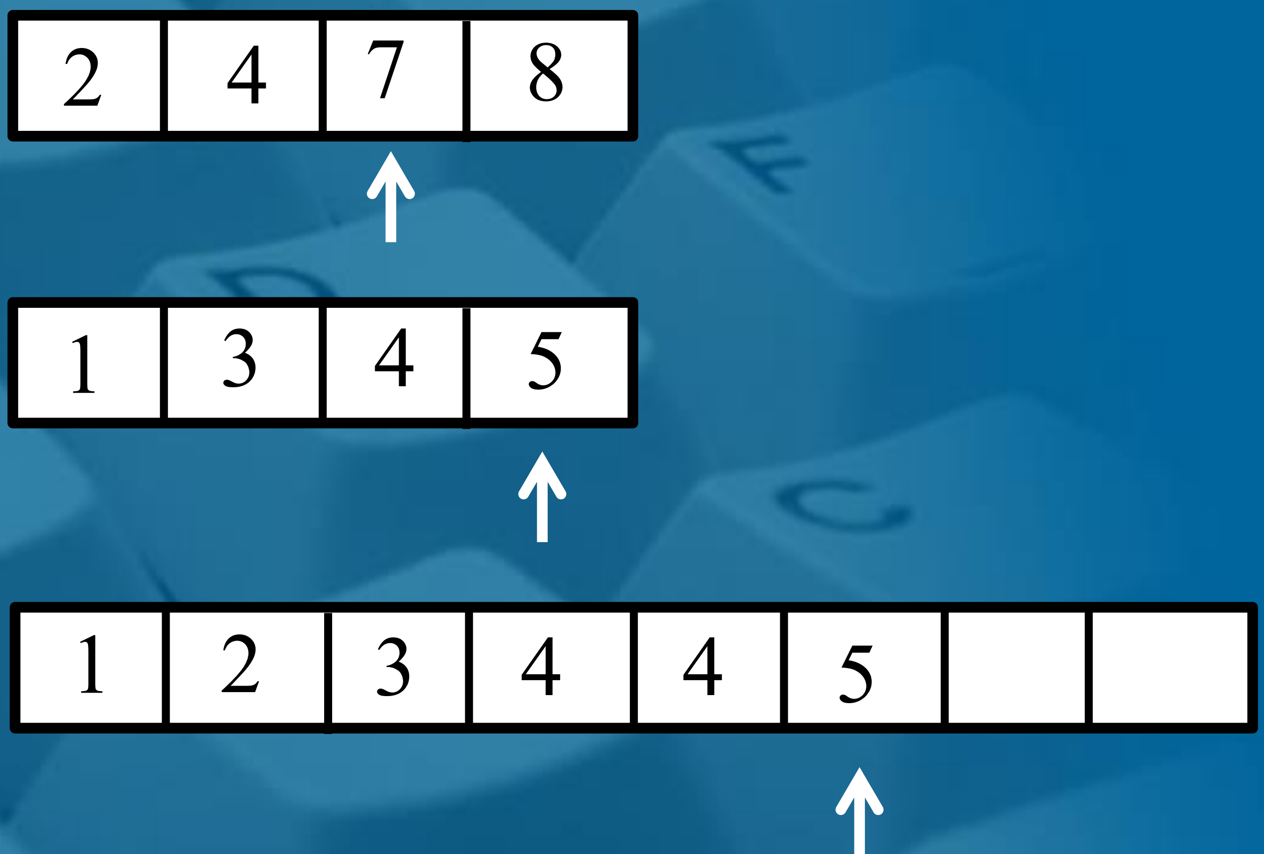

2 4 7 8
1 3 4 5
1

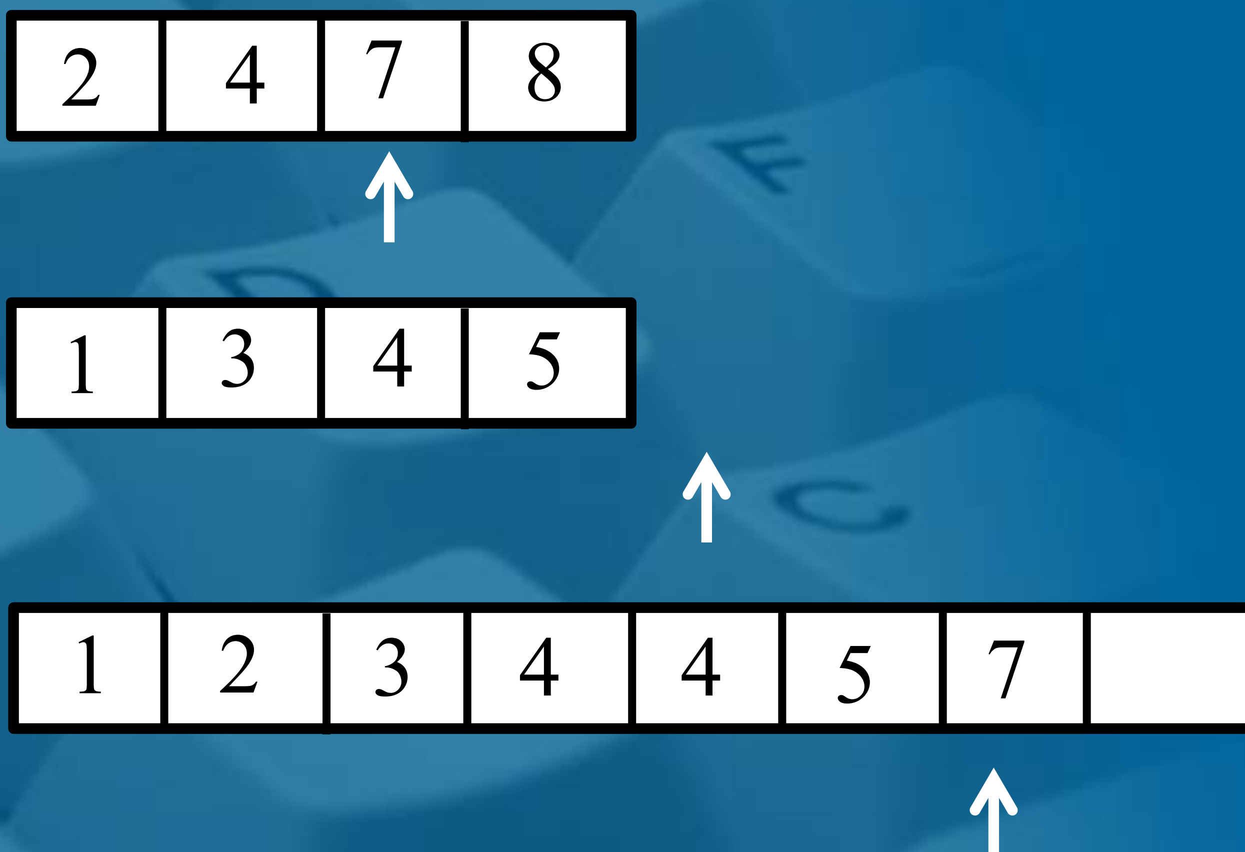

2 4 7 8
1 3 4 5
1 2


2 4 7 8
1 3 4 5
1 2 3


2 4 7 8
1 3 4 5
1 2 3 4


2 4 7 8
1 3 4 5
1 2 3 4 4


2 4 7 8
1 3 4 5
1 2 3 4 4 5


2 4 7 8
1 3 4 5
1 2 3 4 4 5 7

2	4	7	8

1	3	4	5

1	2	3	4	4	5	7	8

2	4	7	8

1	3	4	5

1	2	3	4	4	5	7	8

```
Merge (A, i1, f1, f2)
    Sia X un array ausiliario di lunghezza f2-i1+1
    i=1; k1=i1
    k2=f1+1
    while (k1≤ f1 e k2 ≤ f2) do
        if (A[k1] ≤ A[k2])
        then X[i]=A[k1]
                incrementa i e k1
        else X[i]=A[k2]
                incrementa i e k2
    if (k1≤f1) then copia A[k1;f1] alla fine di X
    else copia A[k2;f2] alla fine di X
    copia X in A[i1;f2]
```

fonde $A[i_1;f_1]$ e $A[f_1+1;f_2]$
output in $A[i_1;f_2]$

Osservazione: sto usando un array ausiliario

Lemma

La procedure Merge fonde due sequenze ordinate di lunghezza n_1 e n_2 in tempo $\Theta(n_1 + n_2)$.

dim

Ogni confronto “consuma” un elemento di una delle due sequenze. Ogni posizione di X è riempita in tempo costante. Il numero totale di elementi è $n_1 + n_2$. Anche la Linea 12 (copia del vettore ausiliario) costa $\Theta(n_1 + n_2)$. ■

```
MergeSort (A, i, f)
    if (i  < f) then
        m = ⌊(i+f)/2⌋
        MergeSort(A,i,m)
        MergeSort(A,m+1,f)
        Merge(A,i,m,f)
```

Corretto?

Sì.

- chiamate ricorsive ordinano le due metà
- il Merge le fonde correttamente

Complessità?

Tempo di esecuzione

- La complessità temporale del MergeSort è descritto dalla seguente relazione di ricorrenza:

$$T(n) = 2\,T(n/2) + O(n)$$

- Usando il Teorema Master si ottiene

$$T(n) = O(n \log n)$$

$a=b=2$, $f(n)=O(n)$ $\rightarrow$ caso 2

Quanta memoria (ausiliaria) usiamo?

- La complessità spaziale del MergeSort è $\Theta(n)$
  - la procedura Merge usa memoria ausiliaria pari alla dimensione di porzione da fondere;
  - non sono mai attive due procedure di Merge contemporaneamente;
  - ogni chiamata di MergeSort usa memoria costante (esclusa quella usata dalla procedura Merge);
  - numero di chiamate di MergeSort attive contemporaneamente sono $O(\log n)$;

- Il MergeSort non ordina *in loco*
  - occupazione di memoria ausiliaria (oltre input) pari a $\Theta(n)$

Ancora un algoritmo di ordinamento che usa la tecnica del ***divide et impera***: il **Quick Sort**

Efficiente?

Caso peggiore, caso medio e versione randomizzata

QuickSort

- Usa la tecnica del divide et impera:

1 Divide: scegli un elemento x della sequenza (perno) e partiziona la sequenza in elementi $\leq$ x ed elementi >x

2 Risolvi i due sottoproblemi ricorsivamente

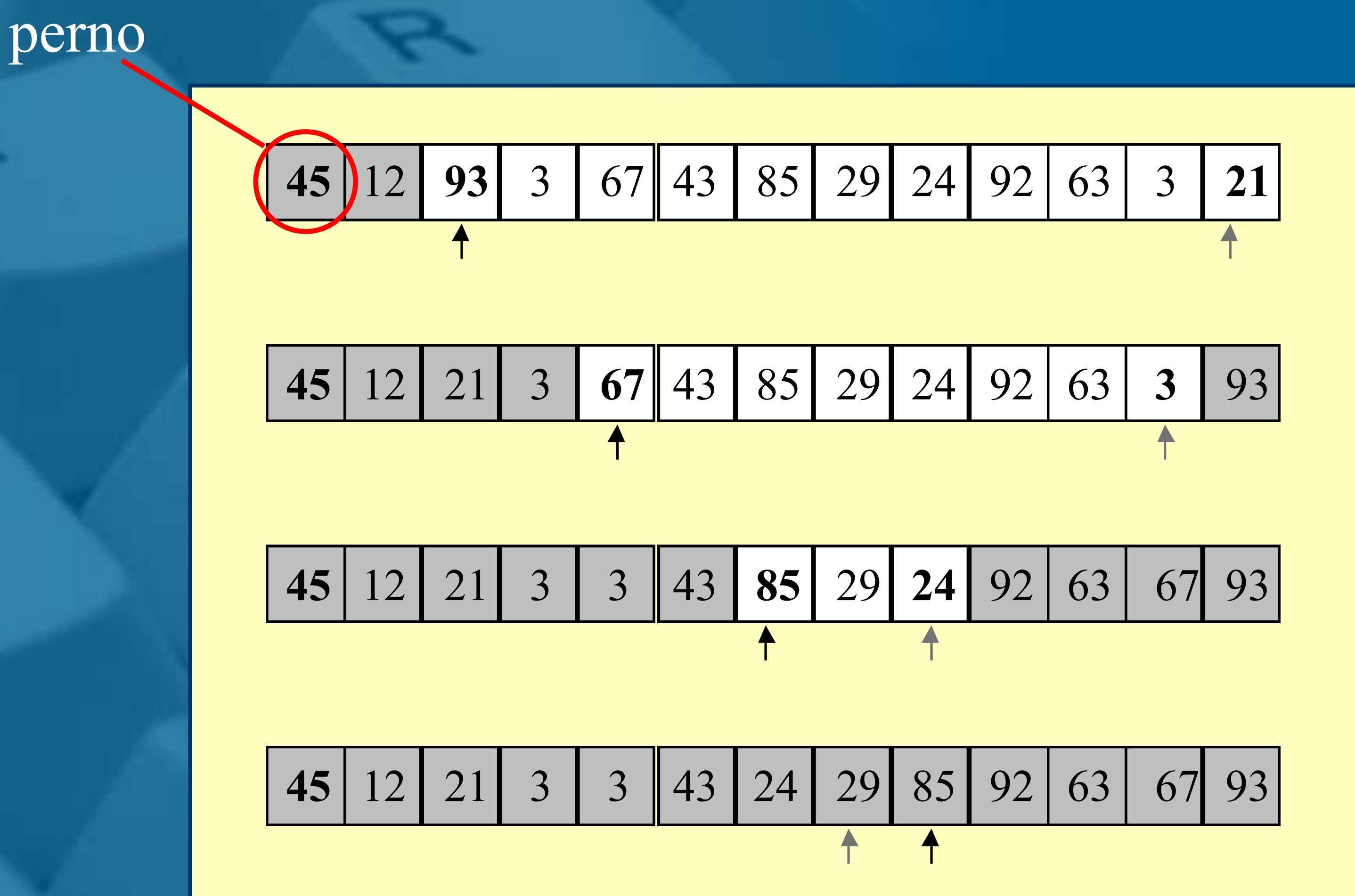
3 Impera: restituisci la concatenazione delle due sottosequenze ordinate

Rispetto al MergeSort, divide complesso ed impera semplice

Partizione (in loco)

- Scegli il perno
- Scorri l'array "in parallelo" da sinistra verso destra e da destra verso sinistra
  - da sinistra verso destra, ci si ferma su un elemento maggiore del perno
  - da destra verso sinistra, ci si ferma su un elemento minore del perno
- Scambia gli elementi e riprendi la scansione

Partizione in loco: un esempio



```
Partition (A, i, f )
x=A[i]
inf =i
sup= f + 1
while (true) do
    do (inf=inf + 1) while (inf ≤ f e A[inf] ≤ x)
    do (sup=sup-1) while (A[sup] > x)
    if (inf < sup) then scambia A[inf] e A[sup]
    else break
scambia A[i] e A[sup]
return sup
```

partiziona A[i;f] rispetto a A[i]

Tempo di esecuzione: $O(n)$

mette il perno "al centro"

restituisce l'indice del "centro"

Proprietà (invariante):

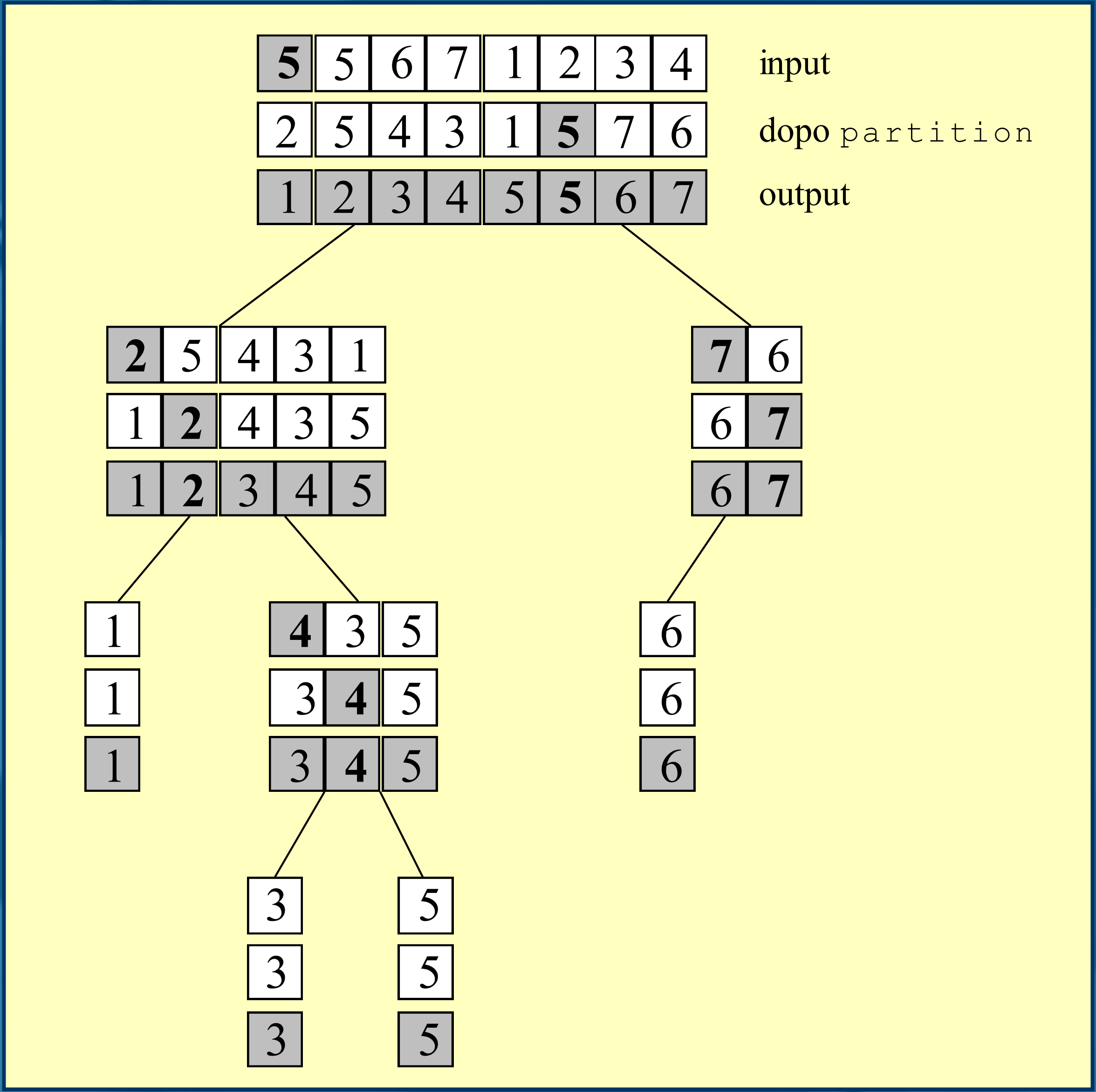
In ogni istante, gli elementi A[i],…,A[inf-1] sono ≤ del perno, mentre gli elementi A[sup+1],…,A[f] sono > del perno

```
QuickSort (A, i, f )
1.     if (i < f) then
2.          m=Partition(A,i,f)
3.          QuickSort(A,i,m-1)
4.          QuickSort(A, m +1,f)
```

ordina A[i;f]

chiamata iniziale:
QuickSort(A,1,n)

Esempio di esecuzione



L'albero delle chiamate ricorsive può essere sbilanciato

QuickSort (A, i, f)

1. **if** (i < f) **then**

2. m=Partition(A,i,f)

3. QuickSort(A,i,m-1)

4. QuickSort(A, m +1,f)

Corretto?

Sì.

- dopo Partition:

A[i:m-1] contiene elem ≤ del perno, A[m] il perno, A[m+1:f] elementi > del perno

- le chiamate ricorsive ordinano A[i:f]

Complessità?

Analisi nel caso peggiore

- Ogni invocazione di Partition posizione almeno un elemento in modo corretto (il perno)
- Quindi dopo n invocazioni di Partition, ognuna di costo $O(n)$ ho il vettore ordinato. Il costo complessivo è quindi $O(n^2)$

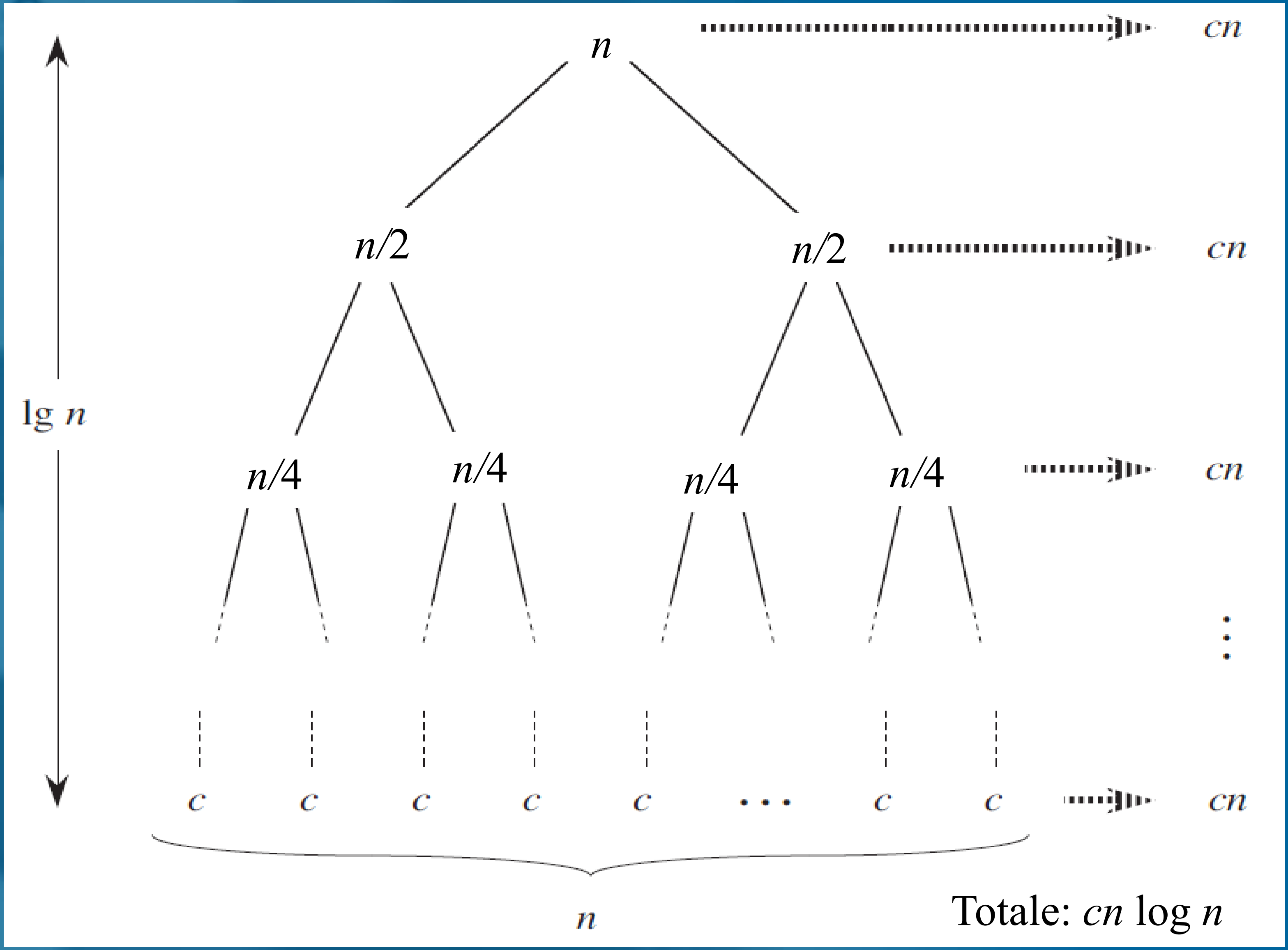
- Il caso peggiore si verifica quando il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array
- La complessità in questo caso è:

$$\begin{aligned} T(n) &= T(n-1)+T(0)+O(n) \\ &= T(n-1)+O(1)+O(n) \\ &= T(n-1)+O(n) \end{aligned}$$

$$\Rightarrow \boxed{T(n)=O(n^2)}$$

complessità nel caso migliore?

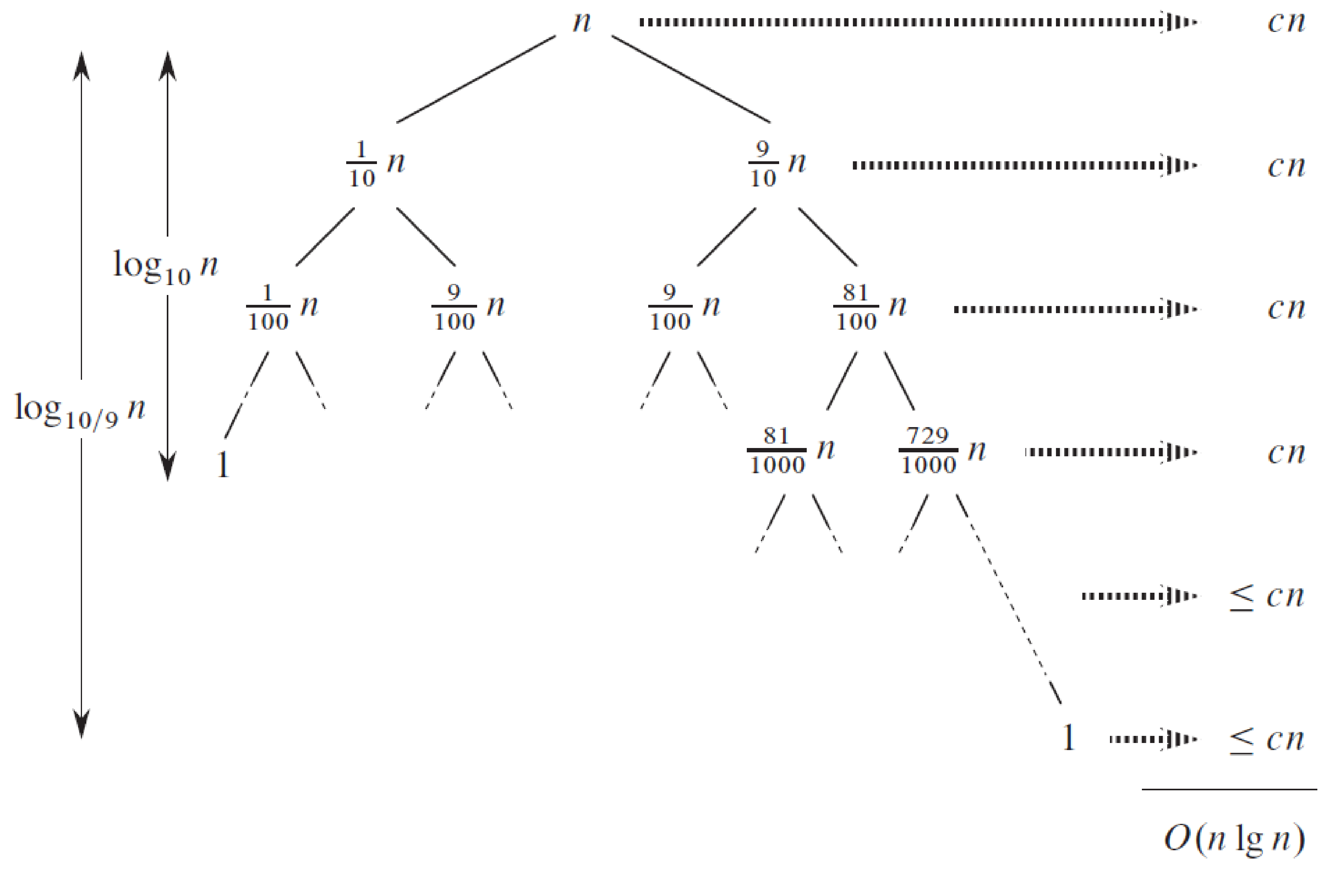
Caso migliore: O(n log n), partizionamento sempre bilanciato



…intuizioni sul caso medio…
(penso al caso di istanze equiprobabili)

- problema: la partizione può essere sbilanciata
- la probabilità che ad ogni passo si presenti la partizione peggiore è molto bassa
- per partizioni che non sono "troppo sbilanciate" l'algoritmo è veloce
- domanda: quale è la complessità dell'algoritmo supponendo che l'algoritmo di partizionamento produca sempre una partizione proporzionale 9-a-1?
- E se la partizione fosse sempre proporzionale a 99-a-1?
- Nota: sembrano partizioni piuttosto sbilanciate…

…la complessità è ancora O(n log n)



…e se le istanze non sono equiprobabili?

Versione randomizzata: scegli il perno x a caso fra gli elementi da ordinare

Teorema

L'algoritmo quickSort randomizzato ordina in loco un array di lunghezza n in tempo $O(n^2)$ nel caso peggiore e $O(n \log n)$ tempo atteso

…e se le istanze non sono equiprobabili?

Versione randomizzata: scegli il perno x a caso fra gli elementi da ordinare

Teorema

L'algoritmo quickSort randomizzato ordina in loco un array di lunghezza n in tempo $O(n^2)$ nel caso peggiore e $O(n \log n)$ con alta probabilità, ovvero con probabilità almeno 1-1/n.

quickSort randomizzato
(randomizzazione ≠ caso medio)

- nessuna assunzione sulla distribuzione di probabilità delle istanze
- nessun input specifico per il quale si verifica il caso peggiore
- il caso peggiore determinato solo dal generatore di numeri casuali

Analisi e progettazione di algoritmi randomizzati: ampia e importante area di studio e ricerca