Algoritmi e Strutture Dati

Capitolo 4
Ordinamento

Camil Demetrescu, Irene Finocchi,
Giuseppe F. Italiano

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
- E' possibile effettuare ricerche in array ordinati in tempo $O(\log n)$

Algoritmi e tempi tipici

- Numerosissimi algoritmi
- Tre tempi tipici: $O(n^2)$, $O(n \log n)$, $O(n)$

n	10	100	1000	10^6	10^9
$n \log_2 n$	~ 33	~ 665	$\sim 10^4$	$\sim 2 \cdot 10^7$	$\sim 3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Modello basato su confronti

- In questo modello, per ordinare è possibile usare solo confronti tra oggetti
- Primitive quali operazioni aritmetiche (somme o prodotti), logiche (and e or), o altro (shift) sono proibite
- Sufficientemente generale per catturare le proprietà degli algoritmi più noti

Lower Bound

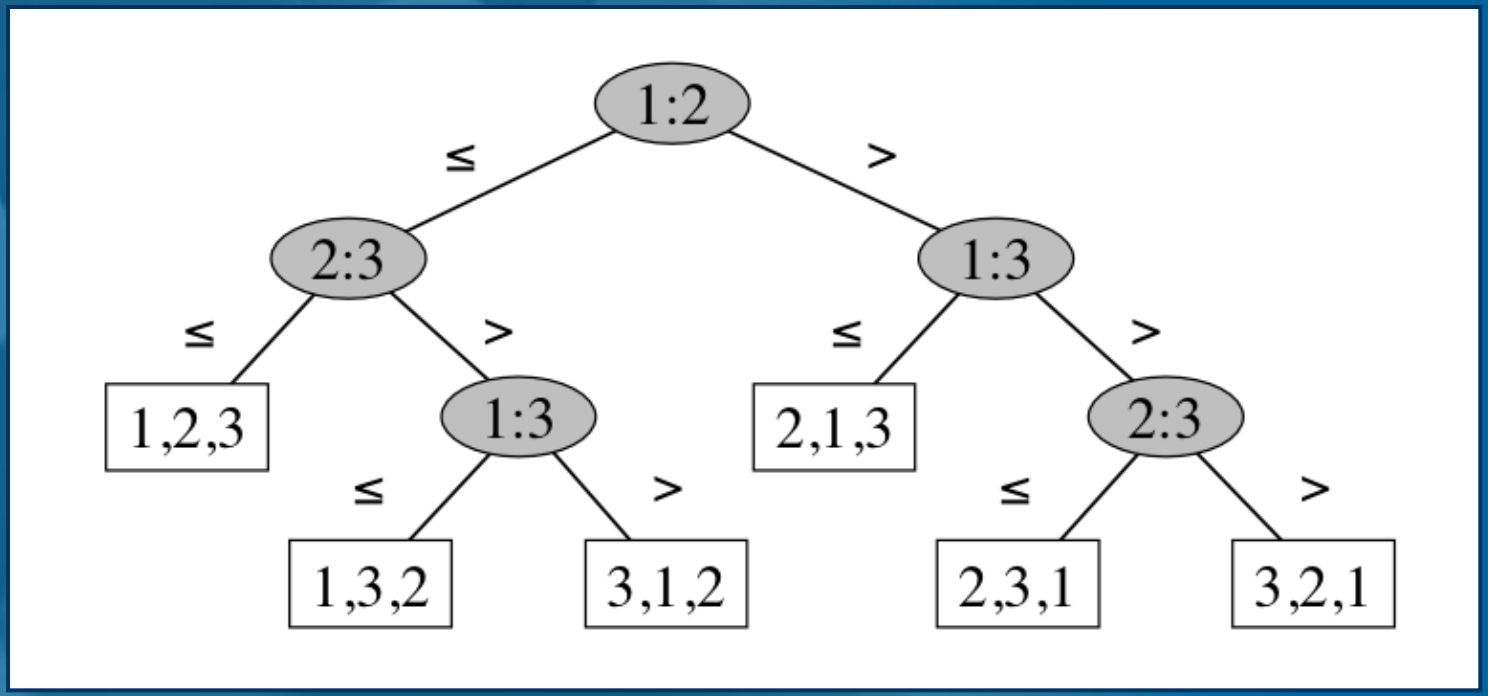
- Si dimostra che nessun algoritmo di sorting basato su confronti può eseguire meno di $O(n \log n)$ istruzioni

Lower bound

- Delimitazione inferiore alla quantità di una certa risorsa di calcolo necessaria per risolvere un problema
- $\Omega(n \log n)$ è un lower bound al numero di confronti richiesti per ordinare n oggetti
- Consideriamo un generico algoritmo $\mathcal{A}$, che ordina eseguendo solo confronti: dimostreremo che $\mathcal{A}$ esegue $\Omega(n \log n)$ confronti

Alberi di decisione

- Descrive le diverse sequenze di confronti che $\mathcal{A}$ potrebbe fare su istanze di lunghezza n



Proprietà

- Per una particolare istanza, i confronti eseguiti da $\mathcal{A}$ su quella istanza rappresentano un cammino radice - foglia
- Il numero di confronti nel caso peggiore è pari all'altezza dell'albero di decisione
- Un albero di decisione per l'ordinamento di n elementi contiene almeno n! foglie

  (una per ogni possibile permutazione degli n oggetti)

Altezza in funzione delle foglie

- Un albero binario con k foglie tale che ogni nodo interno ha esattamente due figli ha altezza almeno $\log_2 k$
- Dimostrazione per induzione su k:
  - Passo base: $0 \geq \log_2 1$
  - $h(k) \geq 1+h(k/2)$ poiché uno dei due sottoalberi ha almeno metà delle foglie
  - $h(k/2) \geq \log_2 (k/2)$ per ipotesi induttiva
  - $h(k) \geq 1+\log_2 (k/2) = \log_2 k$

Il lower bound $\Omega(n \log n)$

- L'altezza dell'albero di decisione è almeno $\log_2 (n!)$
- Formula di Stirling: $n! \sim (2\pi n)^{1/2} \cdot (n/e)^n$
- Quindi: $\log (n!) \sim n \log n - O(n)$

Proprietà degli algoritmi di ordinamento

- Algoritmi incrementali: ad ogni passo si incrementa di un'unità la sottosequenza ordinata.
- Algoritmi *in loco*: utilizzano una quantità di memoria supplementare (oltre a quella necessaria per contenere l'input) di ordine costante.
- Algoritmi adattivi: la misura della loro efficienza varia al variare dell'input

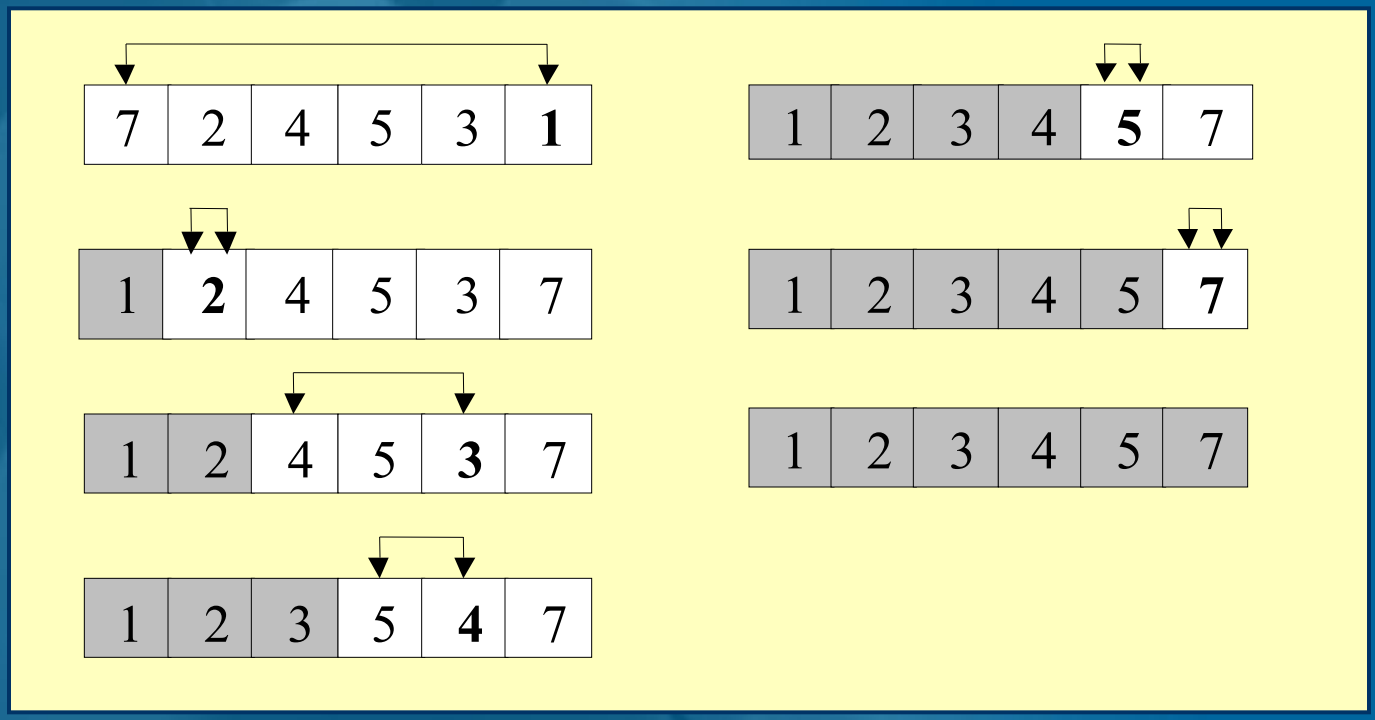
Proprietà degli algoritmi di ordinamento

- Algoritmi stabili: sono gli algoritmi tali che, se nell'insieme da ordinare sono contenuti elementi con chiavi uguali, essi appariranno alla fine dell'algoritmo nello stesso ordine relativo che avevano prima che l'algoritmo venisse applicato.
- Importanti quando si ordinano record rispetto a un campo.

Ordinamenti quadratici

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

```
algoritmo selectionsort(array A)
for k=1 to n-1 do
      m←k
      for j=k+1 to n do
            if (A[j]<A[m]) then m←j
      scambia A[m] con A[k]
```

SelectionSort: analisi

La k-esima estrazione di minimo costa tempo $O(n-k)$

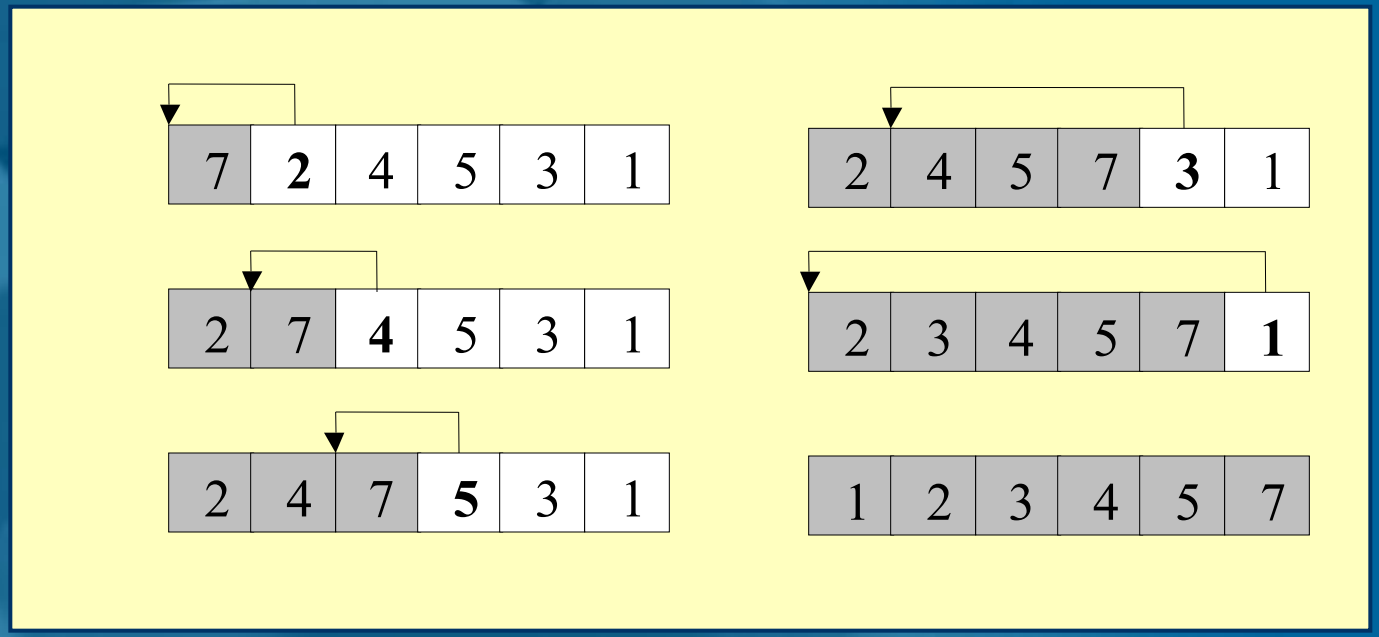
In totale, il tempo di esecuzione è pertanto:

$$\sum_{k=1}^{n-1} O(n-k) = O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$$

usando il cambiamento di variabile $i = n-k$ e la serie aritmetica

InsertionSort

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



Insertionsort

```
algoritmo insertionsort(array A)
for k=1 to n-1 do
      x←A[k+1]
      for j=1 to k+1 do
            if(A[j]>x)then break
      if(j<k+1)then
            for t=k downto j do A[t+1]← A[t]
            A[j]←x
```

InsertionSort: analisi

L'inserimento del k-esimo elemento nella posizione corretta rispetto ai primi k richiede tempo $O(k)$ nel caso peggiore

In totale, il tempo di esecuzione è pertanto:

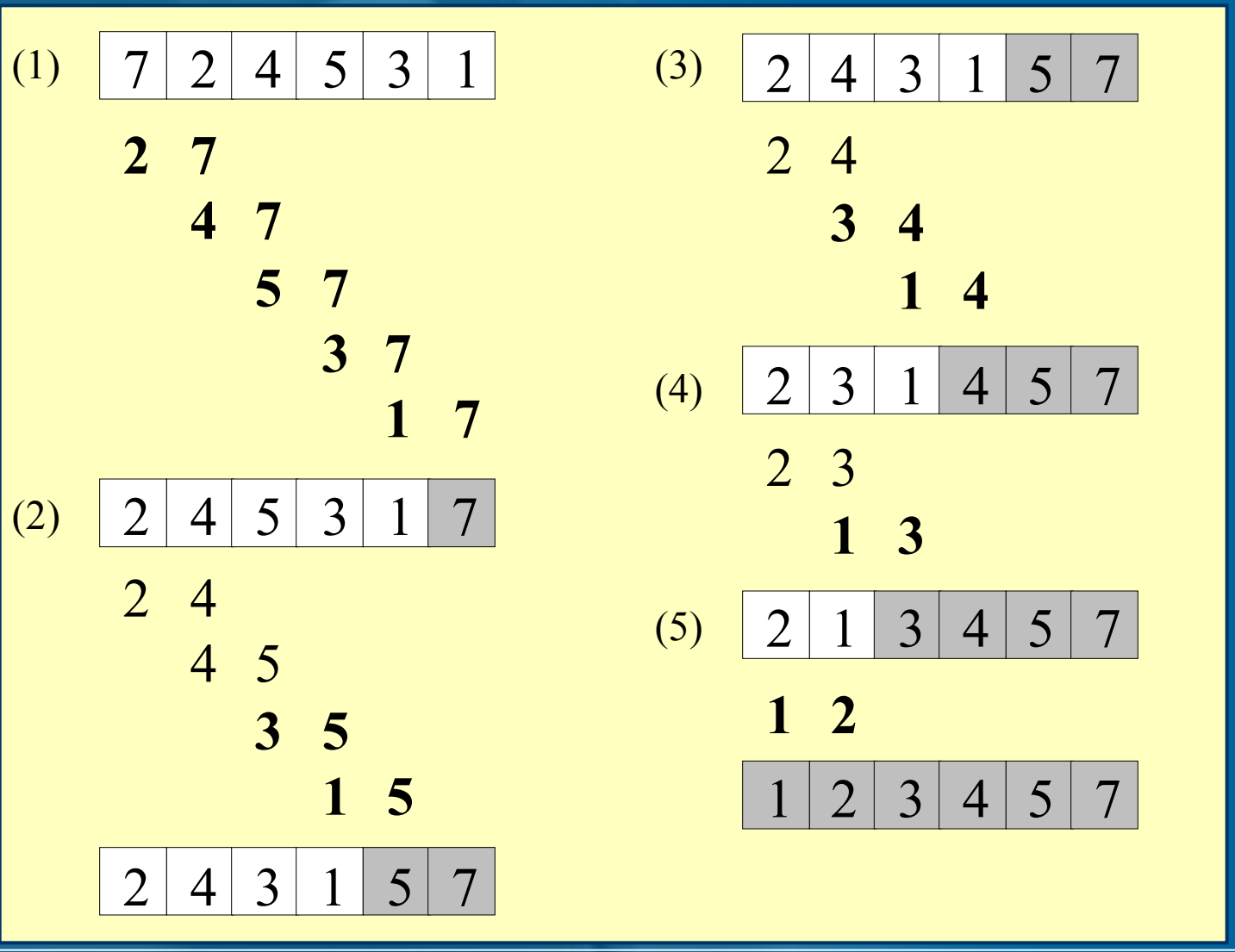
$$O\left(\sum_{k=1}^{n-1} k\right) = O(n^2)$$

usando la serie aritmetica

BubbleSort

- Esegue una serie di scansioni dell'array
  - In ogni scansione confronta coppie di elementi adiacenti, scambiandoli se non sono nell'ordine corretto
  - Dopo una scansione in cui non viene effettuato nessuno scambio l'array è ordinato
- Dopo la k-esima scansione, i k elementi più grandi sono correttamente ordinati ed occupano le k posizioni più a destra ⇒ correttezza e tempo di esecuzione $O(n^2)$

Esempio di esecuzione



Bubblesort

```
algoritmo bubblesort(array A)
for i=1 to n-1 do
      for j=2 to n-i+1 do
            if(A[j-1]>A[j])then
                  scambia A[j-1] e A[j]
      if(non ci sono più scambi)then break
```

BubbleSort: analisi

il ciclo esterno viene effettuato n-1 volte.

il ciclo interno i-esimo nel caso peggiore si effettuano n-i confronti.

Il numero di confronti è dato da

$$\sum_{i=1}^{n-1} (n-i) = n(n-1)/2 = O(n^2)$$

usando la serie aritmetica

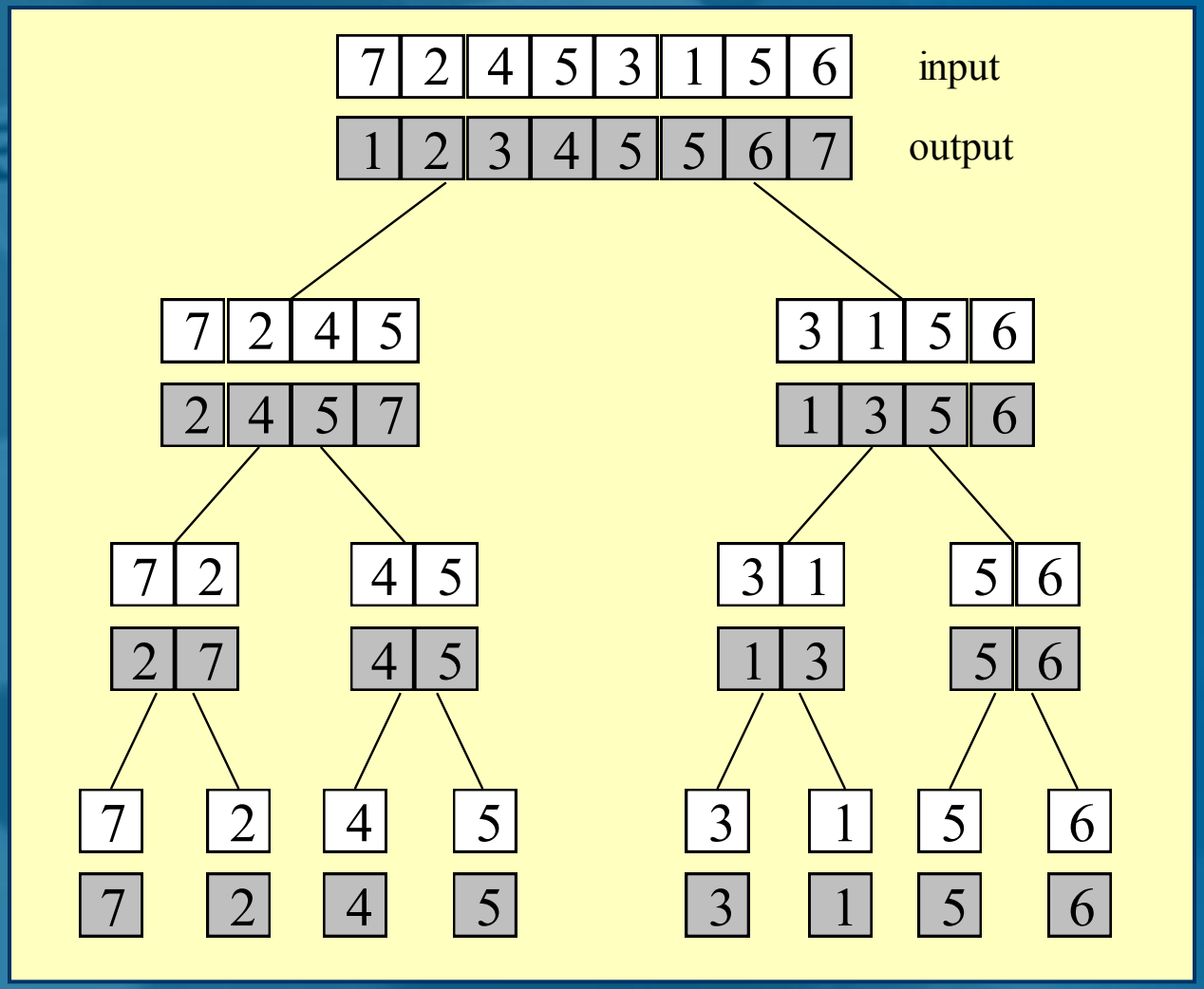
Ordinamenti ottimi
(nel modello basato su confronti)

MergeSort

- Usa la tecnica del divide et impera:

1. Divide: dividi l'array a metà
2. Risolvi il sottoproblema ricorsivamente
3. Impera: fondi le due sottosequenze ordinate

Esempio di esecuzione



Input ed output 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
- Tempo: O(n), dove n è il numero totale di elementi

Procedura Merge

```
Procedura merge(array A, interi r,m,l)
k ← r; i ← r; j ← m+1
while (i<=m, e j<=l) do
      if A[i]<=A[j] then X[k] ← A[i]
                         incrementa i e k
      else X[k] ← A[j]
           incrementa j e k
if (i<=m)then
      copia A[i,m] alla fine di X
else copia A[j,l] alla fine di X
Copia X in A[r,l]
```

MergeSort

```
Algoritmo mergesort(array A, interi r,l)
     if (r>=l) then return
     m←(r+l)/2
     mergesort(A, r, m)
     mergesort(A,m+1, l)
     merge(A,r,m,l)
```

Tempo di esecuzione

- Il numero di confronti del MergeSort è descritto dalla seguente relazione di ricorrenza:

$$C(n) = 2C(n/2) + O(n)$$

- Usando il Teorema Master si ottiene

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

HeapSort

- Stesso approccio incrementale del selectionSort
- Usa però una struttura dati per estrarre in tempo O(log n) il massimo ad ogni iterazione

- Struttura dati heap associata ad un insieme S = albero binario radicato con le seguenti proprietà:
  1) completo fino al penultimo livello
  2) gli elementi di S sono memorizzati nei nodi dell'albero
  3) chiave(padre(v)) ≥ chiave(v) per ogni nodo v

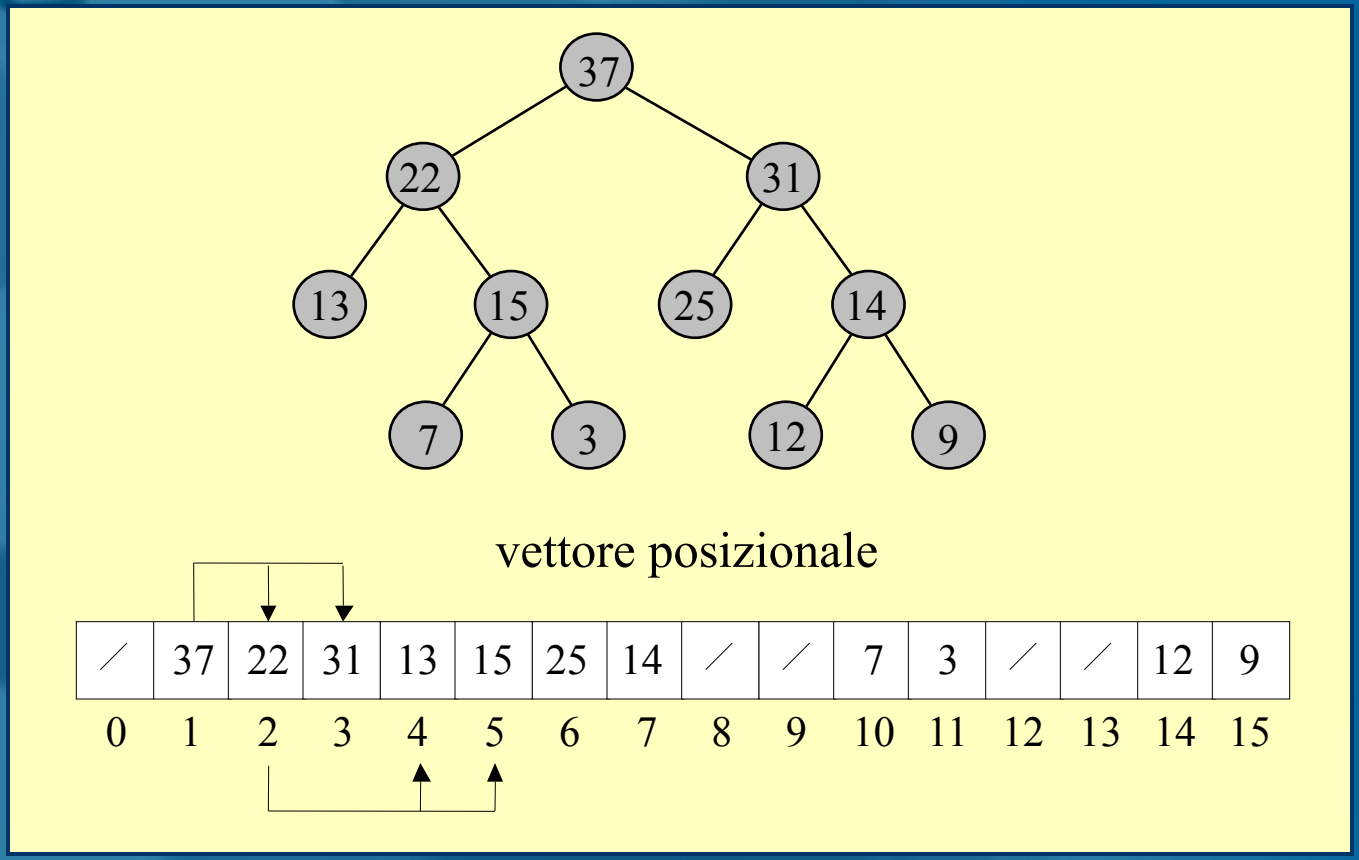
Operazioni su heap

Riguardo agli heap ci interessa effettuare le seguenti operazioni:

- Dato un array, costruire un heap H con quegli elementi
- Trovare il più grande oggetto in H
- Cancellare il più grande oggetto in H

Struttura dati heap

Rappresentazione ad albero e con vettore posizionale

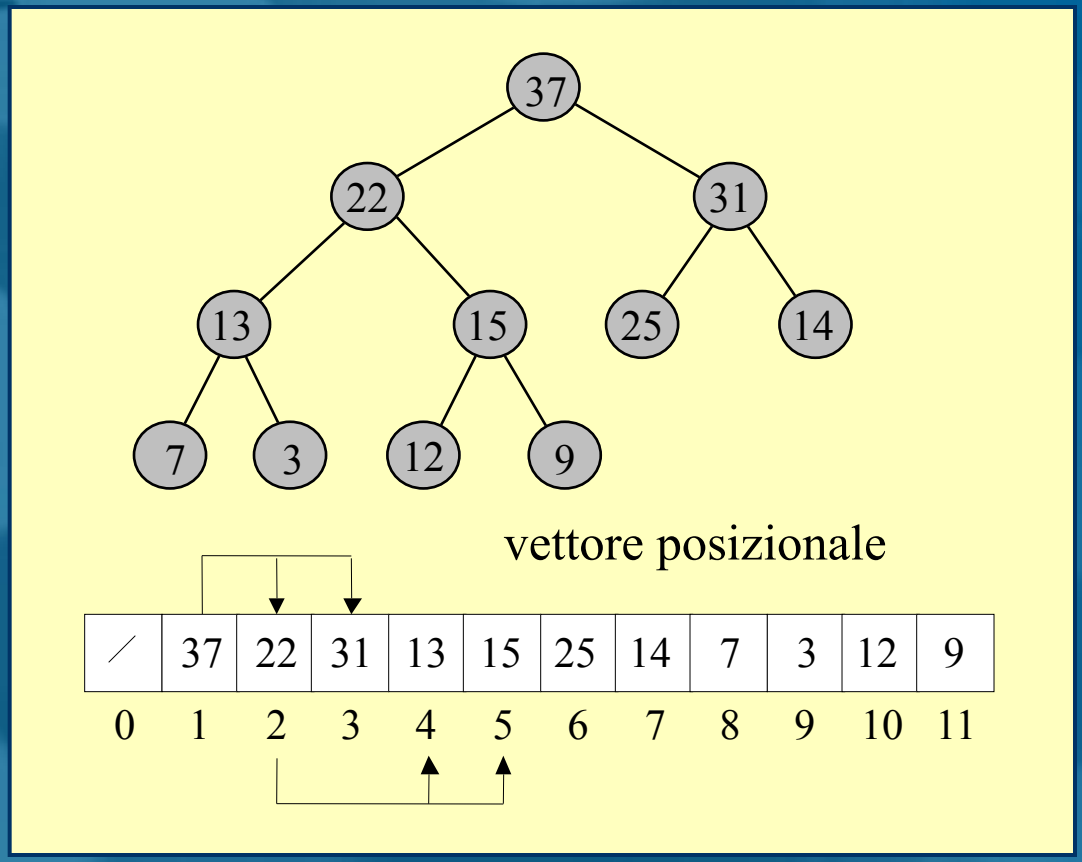


sin(i) = 2i

des(i) = 2i+1

Heap con struttura rafforzata

Le foglie nell'ultimo livello sono compattate a sinistra



Il vettore posizionale ha esattamente dimensione n

Proprietà salienti degli heap

1) Il massimo è contenuto nella radice

2) L'albero ha altezza O(log n)

3) Gli heap con struttura rafforzata possono essere rappresentati in un array di dimensione pari a n

La procedura fixHeap

Se tutti i nodi di H tranne v soddisfano la proprietà di ordinamento a heap, possiamo ripristinarla come segue:

```
fixHeap(nodo v, heap H)
   if (v è una foglia) then return
   else
      sia u il figlio di v con chiave massima
      if (chiave(v) < chiave(u)) then
         scambia chiave(v) e chiave(u)
         fixHeap(u,H)
```

Tempo di esecuzione: O(log n)

Estrazione del massimo

- Estrai il valore che si trova nella radice
- Copia nella radice la chiave contenuta nella la foglia più a destra dell'ultimo livello
- Rimuovi la foglia
- Ripristina la proprietà di ordinamento a heap richiamando fixHeap sulla radice

Tempo di esecuzione: O(log n)

Costruzione dell'heap

Algoritmo ricorsivo basato sul divide et impera

```
heapify(heap H)
   if (H è vuoto) then return
   else
      heapify(sottoalbero sinistro di H)
      heapify(sottoalbero destro di H)
      fixHeap(radice di H,H)
```

Tempo di esecuzione: $T(n)= 2T(n/2)+O(\log n)$

$T(n) = O(n)$ dal Teorema Master

L'algoritmo HeapSort

- Costruisce un heap tramite heapify — O(n)
- Estrae ripetutamente il massimo per n-1 volte — O(n log n)
  - ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata

⇒ ordina in loco in tempo O(n log n)

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

Algoritmo Quicksort

Algoritmo Quicksort(array A)

Scegli un elemento x in A (o un elemento nel range di definizione di A)

Partiziona A rispetto a x calcolando

$A_1=\{y \in A: y \leq x\}$

$A_2=\{y \in A: y > x\}$

if ($|A_1|>1$) **then** quicksort(A_1)

if ($|A_2|>1$) **then** quicksort(A_2)

copia la concatenazione di A_1 e A_2 in A

Partizione in loco

- 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

Tempo di esecuzione: O(n)

Partizione in loco

```
Procedura Partition(array A, indici l,r)->indice
  x←A[l]; inf←l; sup←r+1
  do
     do(inf←inf+1) while (A[inf]≤x)
     do(sup←sup-1) while (A[sup]>x)
     if (inf<sup) scambia A[inf] e A[sup]
     inf←inf+1; sup←sup-1
  while (inf<sup)
  scambia A[l] e A[sup]
  return sup
```

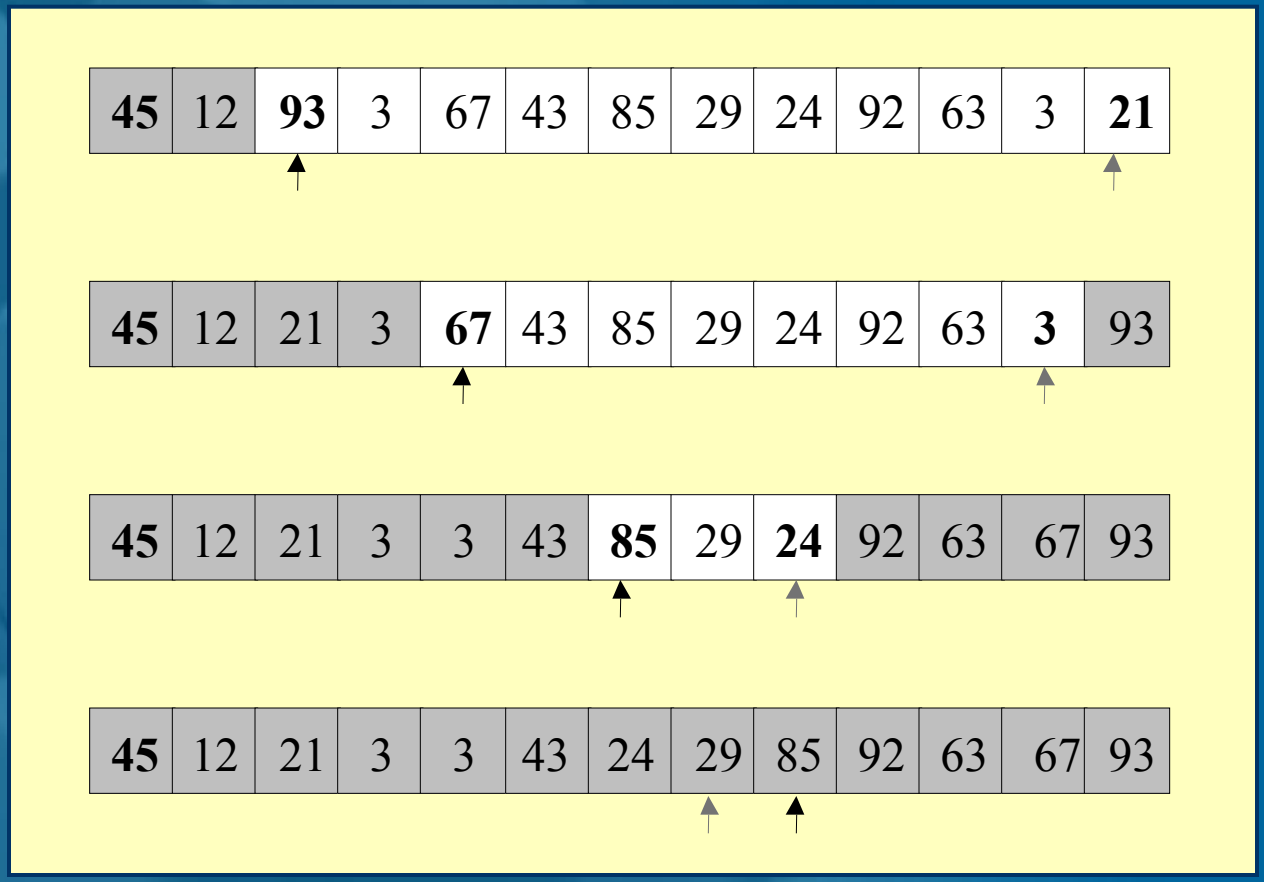
In questo procedimento si sceglie come pivot il primo elemento dell'array
Alla fine viene scambiato con l'elemento in posizione sup, per evitare che alla
Chiamata successiva di quicksort venga scelto lo stesso elemento

Osservazioni e Proprietà

- In questo procedimento si sceglie come pivot il primo elemento dell'array
- Alla fine viene scambiato con l'elemento in posizione sup, per evitare che alla chiamata successiva di quicksort venga scelto lo stesso elemento
- In ogni istante gli elementi A[l]…A[inf-1] sono minori o uguali al pivot, mentre gli elementi A[sup+1]…A[r] sono maggiori del pivot

Partizione in loco: un esempio

Pivot 45



Quicksort in loco

```
Algoritmo quicksort(array A, indici l,r)
    if (l>r) then return;
    m←partition(A,l,r)
    quicksort(A,l,m-1)
    quicksort(A,m+1,r)
```

Analisi nel caso peggiore

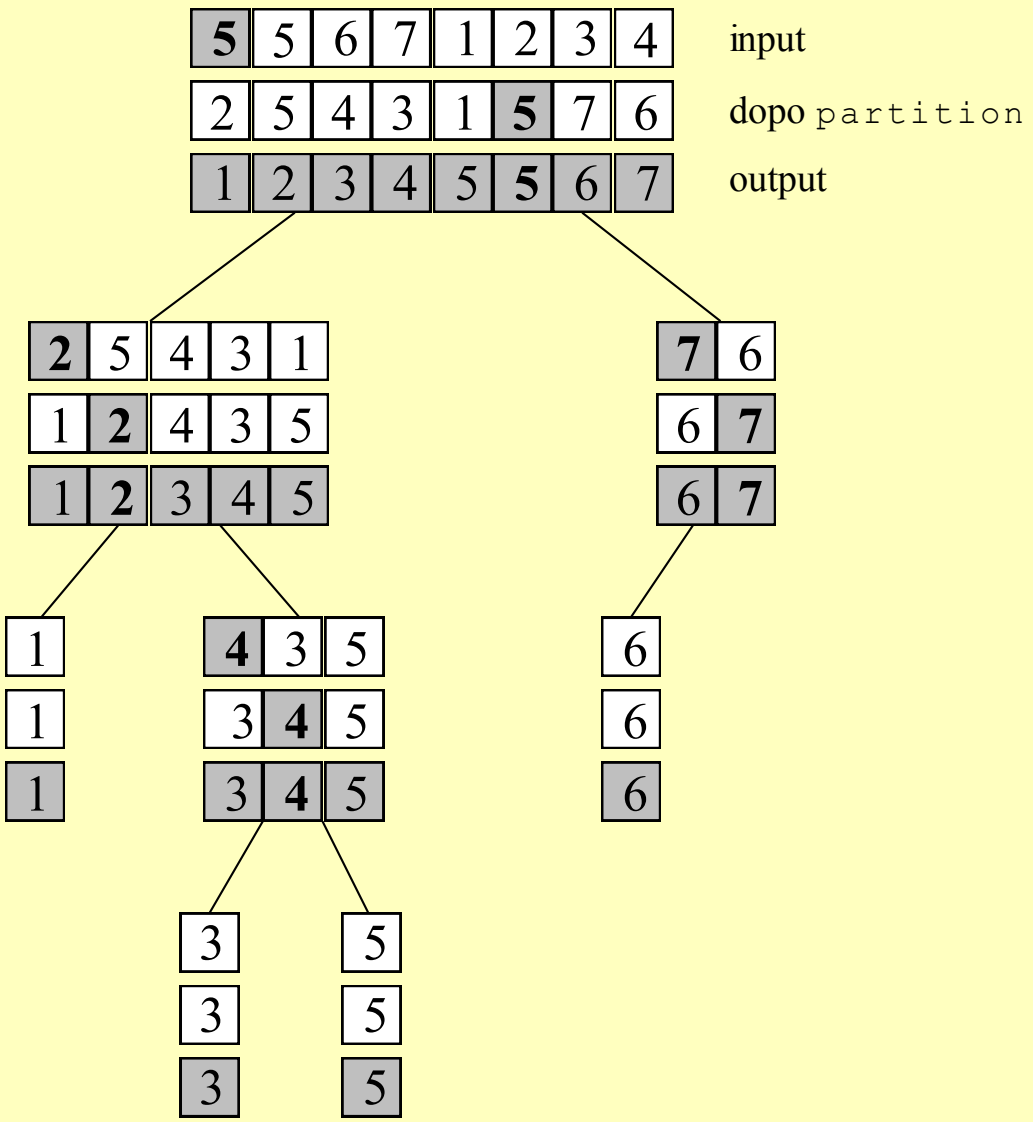
- Nel caso peggiore, il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array
- Il numero di confronti è pertanto:

$$C(n)=C(n-1) + O(n)$$

- Svolgendo per iterazione si ottiene

$$C(n) = O(n^2)$$

Esempio di esecuzione



L'albero delle chiamate ricorsive può essere sbilanciato

Randomizzazione

- Possiamo evitare il caso peggiore scegliendo come perno un elemento a caso
- Poiché ogni elemento ha la stessa probabilità, pari a 1/n, di essere scelto come perno, il numero di confronti nel caso atteso è:

$$C(n)= \sum_{a=0}^{n-1} \frac{1}{n}\left[n-1+C(a)+C(n-a-1)\right] = n-1+\sum_{a=0}^{n-1} \frac{2}{n} C(a)$$

dove a e (n-a-1) sono le dimensioni dei sottoproblemi risolti ricorsivamente

Analisi nel caso medio

La relazione di ricorrenza $C(n) = n-1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$

ha soluzione $C(n) \leq 2\, n \log n$

Dimostrazione per sostituzione

Assumiamo per ipotesi induttiva che $C(i) \leq 2\, i \log i$

$$\Rightarrow \quad C(n) \leq n-1 + \sum_{i=0}^{n-1} \frac{2}{n} 2\, i \log i \leq n-1 + \frac{2}{n} \int_{2}^{n} x \log x \, dx$$

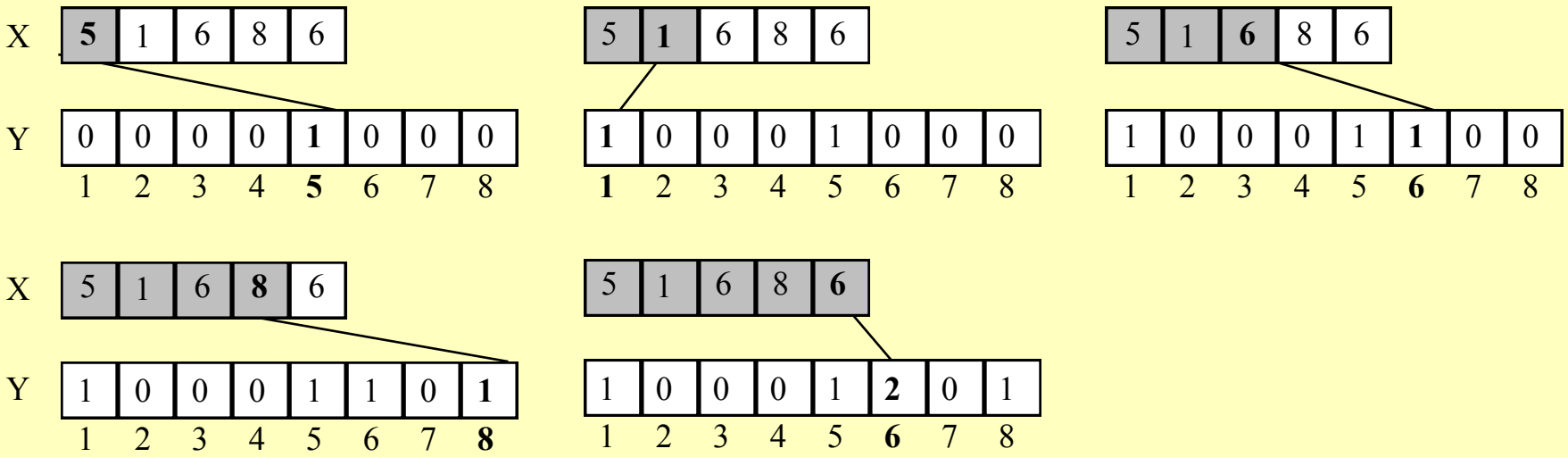
Integrando per parti si dimostra che $C(n) \leq 2\, n \log n$

Ordinamenti lineari
(per dati di input con proprietà particolari)

IntegerSort: fase 1

Per ordinare n interi con valori in [1,k]

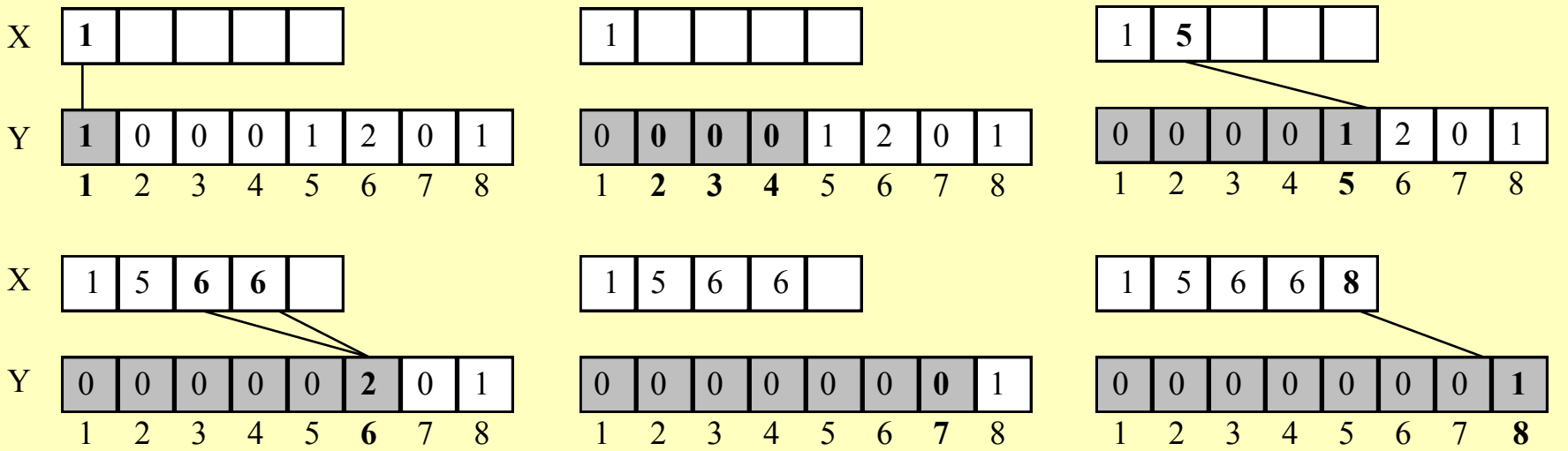
Mantiene un array Y di k contatori tale che Y[x] = numero di volte che il valore x compare nell'array di input X



(a) Calcolo di Y

IntegerSort: fase 2

Scorre Y da sinistra verso destra e, se Y[x]=k, scrive in X il valore x per k volte



(b) Ricostruzione di X

Counting sort

```
Algoritmo countingsort(array A)
   for i=1 to k cont[i]←0
   for j=1 to n cont[A[j]]←cont[A[j]]+1
   j←1
   for i=1 to k do
       while (cont[i]>0) do
             A[j]←i
             j←j+1
             cont[i]←cont[i]-1
```

Stabilità

- Ma che succede se gli interi considerati sono le chiavi di un record, di cui vogliamo in qualche modo mantenere l'ordinamento precedente? L'algoritmo visto non è stabile.

Counting sort stabile

```
Algoritmo countingsort(array A)
  for i=1 to k cont[i]←0
  for j=1 to n cont[A[j]]←cont[a[j]]+1
  for i=2 to k do cont[i]←cont[i-1]+cont[i]
  for j=n downto 1 do
     B[cont[A[j]]]←A[j]
     cont[A[j]]←cont[A[j]]-1
```

IntegerSort: analisi

- Tempo O(k) per inizializzare Y a 0
- Tempo O(n) per calcolare i valori dei contatori
- Tempo O(n+k) per ricostruire X

O(n+k)

Tempo lineare se k=O(n)

BucketSort

Per ordinare n record con chiavi intere in [1,k]

- Basta mantenere un array di liste, anziché di contatori, ed operare come per IntegerSort
- La lista Y[x] conterrà gli elementi con chiave uguale a x
- Concatenare poi le liste

Tempo O(n+k) come per IntegerSort

Stabilità

- Un algoritmo è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave
- Il BucketSort può essere reso stabile appendendo gli elementi di X in coda alla opportuna lista Y[i]

RadixSort

- Cosa fare se il massimo valore k>n, ad esempio se $k = \Theta(n^c)$?
- Rappresentiamo gli elementi in base b, ed eseguiamo una serie di BucketSort
- Partiamo dalla cifra meno significativa verso quella più significativa

Per b=10

2397	5924	5924	4368	2397
4368	2397	4368	2397	4368
5924	4368	2397	5924	5924

Correttezza

- Se x e y hanno una diversa t-esima cifra, la t-esima passata di BucketSort li ordina
- Se x e y hanno la stessa t-esima cifra, la proprietà di stabilità del BucketSort li mantiene ordinati correttamente

Dopo la t-esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle t cifre meno significative

Tempo di esecuzione

- $O(\log_b k)$ passate di bucketsort
- Ciascuna passata richiede tempo $O(n+b)$

$$O((n+b)\log_b k)$$

Se $b = \Theta(n)$, si ha $O(n \log_n k) = O\left[n \left[\frac{\log k}{\log n}\right]\right]$

Tempo lineare se $k=O(n^c)$, c costante

Riepilogo

- Nuove tecniche:
  - Divide et impera (MergeSort, QuickSort)
  - Randomizzazione (QuickSort)
  - Strutture dati efficienti (HeapSort)
- Proprietà particolari dei dati in ingresso possono aiutare a progettare algoritmi più efficienti: algoritmi lineari
- Alberi di decisione per la dimostrazione di delimitazioni inferiori