Algoritmi e Strutture Dati

Algoritmi di ordinamento

Basato su materiale di C. Demetrescu, I. Finocchi, G.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 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

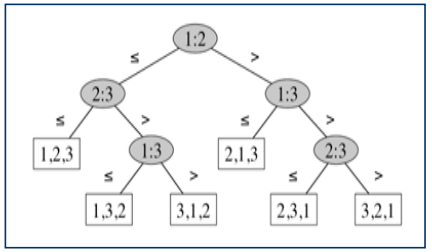
- In questo modello, per ordinare è possibile usare solo **confronti tra oggetti**
- Primitive quali operazioni aritmetiche (somme o prodotti), logiche (and e or), o altro (shift) sono proibite
- Sufficientemente generale per catturare le proprietà degli algoritmi più noti

Lower bound

- 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
- Dato un generico algoritmo *A*, che ordina eseguendo solo confronti:
  - Nel caso peggiore, *A* esegue $\Omega(n \log n)$ confronti

Albero di decisione

Descrive le varie sequenze di confronti che l'algoritmo *A* potrebbe fare su istanze di lunghezza *n*



- Per una particolare istanza, i confronti eseguiti da *A* su quella istanza rappresentano un **cammino radice – foglia**
- Il numero di confronti nel caso peggiore è pari **all'altezza dell'albero di decisione**

Proprietà

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
- $h(k) \geq 1+\log_2 (k/2) = \log_2 k$

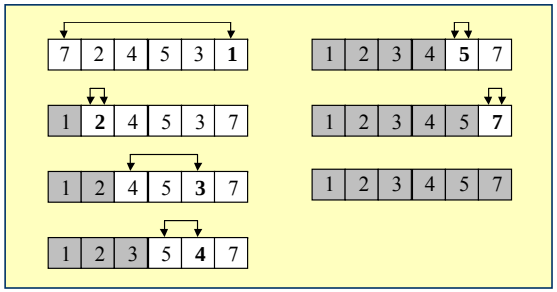
Il lower bound $\Omega(n \log n)$

- Un albero di decisione per l'ordinamento di *n* elementi contiene almeno **n!** foglie
  - una per ogni possibile permutazione degli *n* oggetti
- 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: **$\log (n!) \sim \theta(n \log n)$**

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: analisi

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

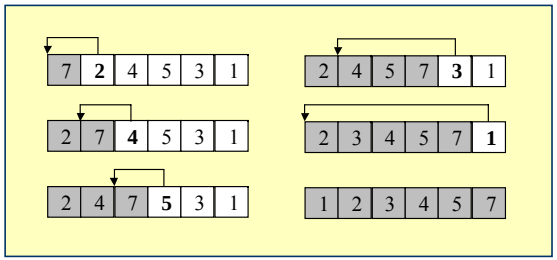
In totale, il tempo di esecuzione è pertanto:

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

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: 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:

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

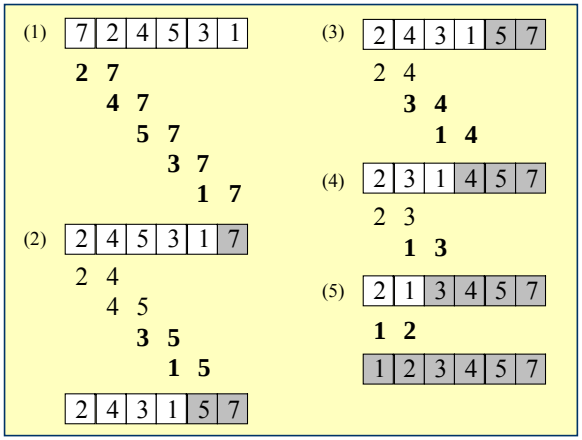
usando la serie aritmetica

BubbleSort

- Esegue una serie di scansioni dell'array
  - In ogni scansione confronta coppie di **elementi adiacenti**, scambiandoli se non sono in ordine
  - Dopo una scansione in cui non viene effettuato nessuno scambio l'array è ordinato
- Proprietà
  - *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



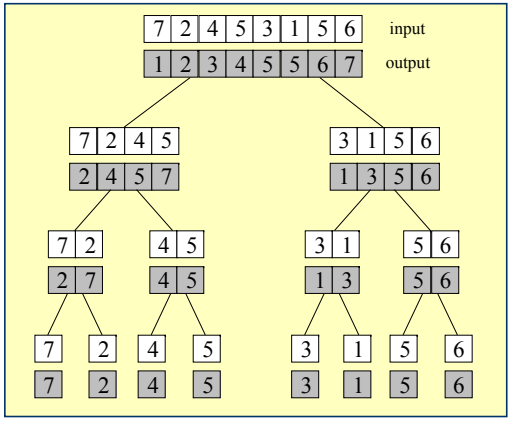
Ordinamenti ottimi

(nel modello basato su confronti)

MergeSort

- Usa la tecnica del divide et impera
  - Risolve il problema su una data istanza, affrontando un insieme di problemi dello stesso tipo su istanze di dimensione inferiore
- Può essere implementato con la ricorsione
  1. **Partizionamento ("Divide")**: dividi l'array a metà
  2. Risolvi i due sottoproblemi ricorsivamente
  3. **Ricombinazione ("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

Tempo di esecuzione

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

$$C(n) = 2\,C(n/2) + k\,n$$

- La soluzione può essere trovata mediante iterazione

$$\begin{aligned} C(n) &= 2 \bullet C(n/2) + k \bullet n \\ &= k \bullet n + 2\,k \bullet (n/2) + 4 \bullet C(n/4) \\ &= k \bullet n\,(1 + 2 \bullet 1/2 + \ldots\ 2^{i} \bullet 1/2^{i} + 2^{\log n} \bullet 1/2^{\log n}) \\ &= k \bullet n \bullet \log n \\ &= O(n \bullet \log n) \end{aligned}$$

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:

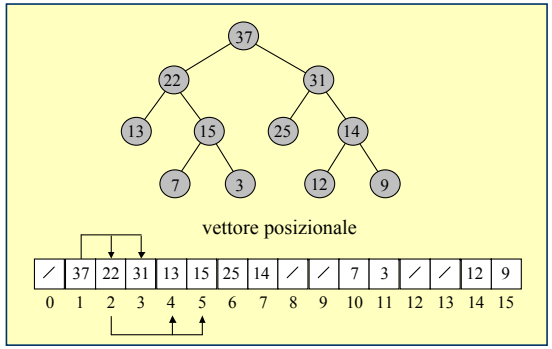
  1) pieno fino al penultimo livello

  2) gli elementi di S sono memorizzati nei nodi dell'albero

  3) chiave(padre(v)) ≥ chiave(v) per ogni nodo v

Struttura dati heap

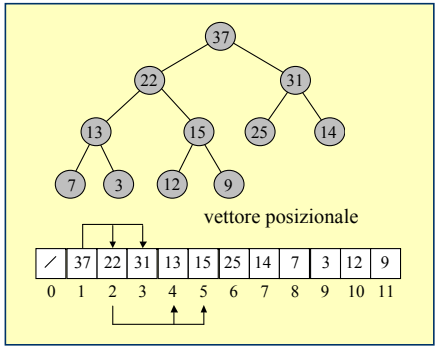
Rappresentazione ad albero e con vettore posizionale



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

- Copia nella radice la chiave contenuta nella 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

Approccio naive:

- 1) Creo un heap vuoto
- 2) Inserisco uno per volta tutti gli *n* elementi nell'heap (ogni
- inserimento costa al più *log n*)

Complessità: O(*n logn*)

Costruzione dell'heap: un metodo più sofisticato

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
  - naive **O(n log n)**
  - oppure con *heapfy* **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 ≤ *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

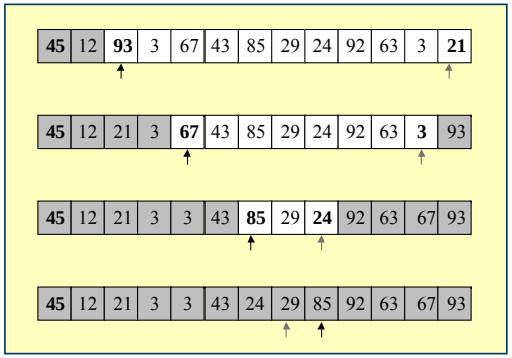
Può essere applicato a strutture non indicizzate!

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: un esempio



Analisi nel caso peggiore

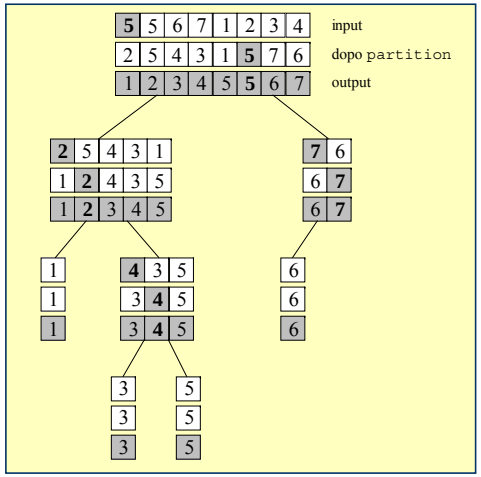
- Nel caso peggiore, il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array
- Il numero di confronti è pertanto:

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

- Svolgendo per iterazione si ottiene

$$\mathbf{C(n) = O(n^2)}$$

Esempio di esecuzione



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 $C(i) \leq 2\, i \log i$

$$\sum_{a=0}^{n-1} \frac{2}{n} 2\, i \log i \leq n-1+ \frac{2}{n} \int_2^n x \log x \, dx$$

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

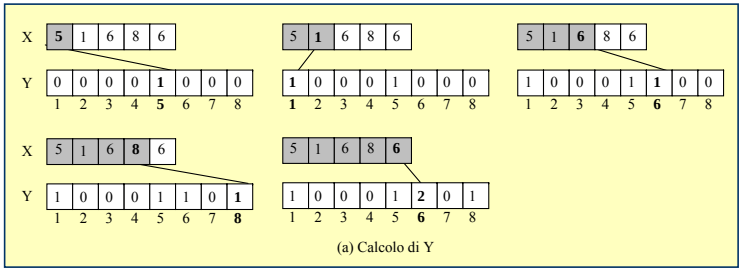
Ordinamenti lineari

(per dati di input con proprietà particolari)

IntegerSort: fase 1

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

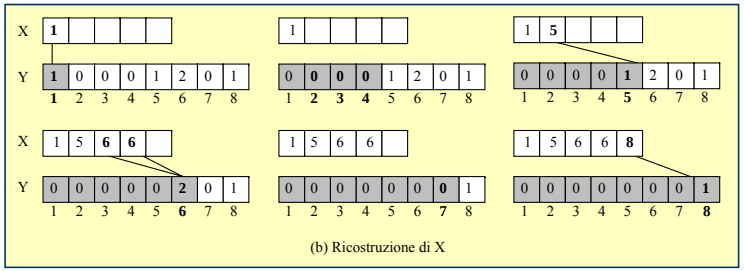
Mantiene un array *Y* di *k* contatori tale che *Y[x]* = numero di volte che il valore *x* compare nell'array di input *X*



(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

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

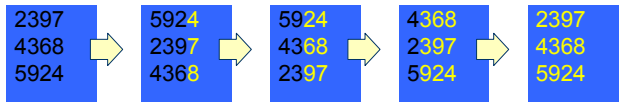
Stabilità

- 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 valore massimo *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



Correttezza

- Se *x* e *y* hanno 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 $O(n \log_n k) = O(n)$

 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