

# Algoritmi (modulo di laboratorio)

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano



- Lezioni: Martedì 8.30 - 10.30 in aula 8 Mercoledì 10.30 - 13.30 in aula 309  
Giovedì 16.30 - 18.30 in aula 307 Venerdì 10.30 - 12.30 in aula 4
- Ricevimento: su appuntamento (Dipartimento di Informatica)
- E-mail: roberto.cordone@unimi.it
- Pagina web: <http://homes.di.unimi.it/~cordone/courses/2024-algo/2024-algo.html>
- Sito Ariel: <https://mgoldwurmasd.ariel.ctu.unimi.it>

Lezione 14: Algoritmi di ordinamento "efficienti"

Milano, A.A. 2023/24

# Il problema dell'ordinamento

Sia  $U$  un insieme dotato di un ordine debole  $\preceq$  (sono ammessi i doppieni)

Il **problema dell'ordinamento** ha come

- istanza: qualsiasi vettore  $V$  su  $U$
- soluzione: il vettore  $V'$  permutazione di  $V$  tale che

$$V[i] \preceq V[j] \text{ per ogni } i \leq j$$

Esempio:

$$V = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 5 & 2 & 8 & 4 & 7 & 1 & 3 & 6 \\ \hline \end{array}$$

$$V' = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline \end{array}$$

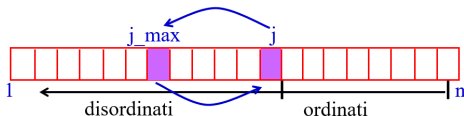
# SelectionSort

L'algoritmo *SelectionSort* divide il vettore  $V$  in due tabelle:

- il sottovettore iniziale contiene gli elementi più piccoli disordinati
- il sottovettore finale contiene gli elementi più grandi ordinati

La seconda tabella è vuota all'inizio, poi cresce un elemento alla volta:

- l'elemento massimo della prima tabella si sposta nella seconda e diventa il suo primo elemento



# SelectionSort: complessità

```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)            $\sum_{j=1}^n (\dots)$ 
  {
    i = TrovaIndiceMassimo(V,j);     $g(j)$ 
    Scambia(&V[i],&V[j]);            $\Theta(1)$ 
  }
}
```

```
TrovaIndiceMassimo(V,n)              $g(j) = \dots$ 
{
  iMax = 1;                           $\Theta(1)$ 
  for (i = 2; i <= n; i++)            $\sum_{i=1}^j (\dots)$ 
    if (V[i] > V[iMax]) iMax = i;     $\Theta(1)$ 
  return iMax;
}
```

*SelectionSort* è un algoritmo quadratico:  $T(n) \in \Theta(n^2)$

# Inefficienza di *SelectionSort*

La ricerca dell'elemento massimo rende *SelectionSort* inefficiente

*Se richiedesse tempo costante, la complessità scenderebbe a  $O(n)$*

La struttura dati astratta *max-heap* consente di

- gestire un insieme (compresi inserimenti e cancellazioni)
- determinarne l'elemento massimo in tempo costante

Si può usarla per rappresentare la prima tabella, ma occorre

- ① costruirla al principio quando coincide con l'intero vettore
- ② mantenerla aggiornata quando perde l'elemento massimo

Queste operazioni sono però abbastanza efficienti

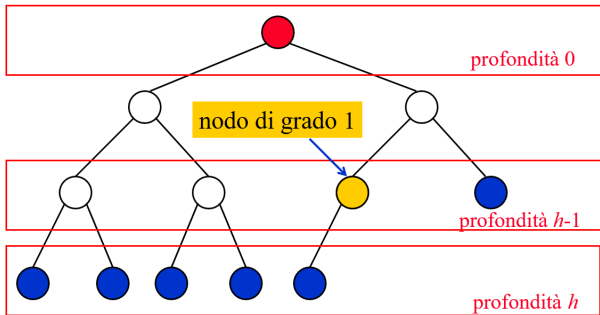
- aggiungere un elemento nuovo
- modificare un elemento
- cancellare un elemento (*ordinare richiede solo questa operazione*)

richiedono tempo logaritmico

# La struttura *max-heap*

Un *max-heap* è caratterizzato da due proprietà:

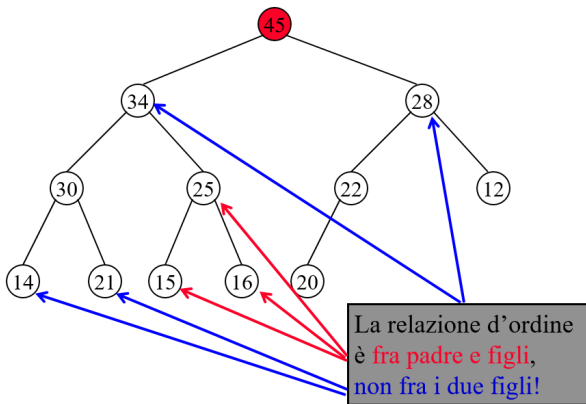
- 1 è un **albero binario** (*Lezione 12*) **quasi completo**, cioè con
  - foglie di profondità  $h$  o  $h - 1$
  - al più un nodo di grado 1
    - a profondità  $h$
    - col solo figlio sinistro
    - con tutti i nodi alla sua destra nello stesso livello di grado nullo



# La struttura *max-heap*

Un *max-heap* è caratterizzato da due proprietà:

- 2 i nodi sono etichettati con valori tratti da un insieme  $U$  ordinato e l'etichetta di ogni nodo non precede quelle degli eventuali nodi figli



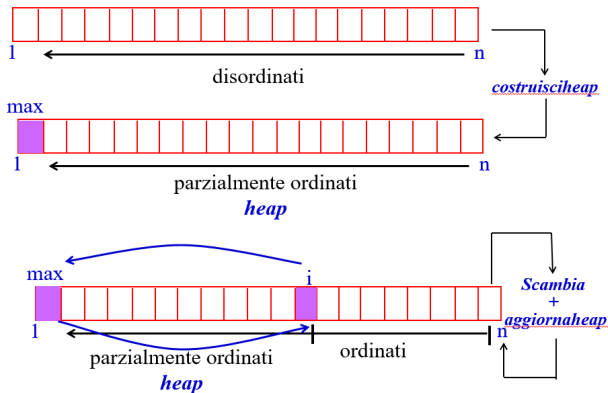
# Heap e ordinamenti parziali

Un *heap* rappresenta un **ordinamento parziale**, cioè una relazione binaria

- riflessiva
- transitiva
- antisimmetrica

Questo suggerisce che possa essere utile per il problema dell'ordinamento

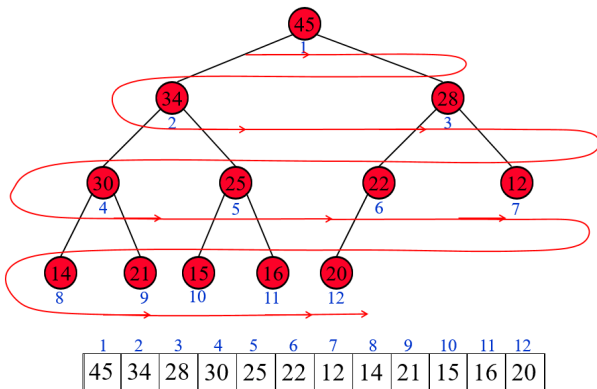
*Manca solo la completezza!*





# Heap e tabella

Un *heap* può facilmente essere rappresentato come una tabella



La quasi completezza consente di **sostituire puntatori espliciti con operazioni aritmetiche sugli indici**:

- il figlio sinistro di  $V[i]$  sta in  $V[2i]$
- il figlio destro di  $V[i]$  sta in  $V[2i + 1]$
- il padre di  $V[i]$  sta in  $V[i/2]$

# Operazioni fondamentali sugli *heap*

Le operazioni fondamentali sugli *heap* sono

- **costruzione** a partire da un semplice vettore non ordinato

$$\text{creaheap} : \mathcal{V}_{n,U} \rightarrow \mathcal{H}_{n,U}$$

- **aggiornamento**, cioè ripristino dell'etichettatura corretta in un nodo, assumendo che essa valga in ogni altro nodo

$$\text{aggiornaheap} : \mathcal{H}_{n,U}^{(1)} \times \mathbb{N}^+ \rightarrow \mathcal{H}_{n,U}$$

intendendo con  $\mathcal{H}_{n,U}^{(1)}$  l'insieme degli alberi binari quasi completi con al più una violazione sulle etichette (eventualmente, nessuna)

# Aggiornamento

L'aggiornamento di un *heap* in un nodo dato consiste nel

- determinare il figlio con l'etichetta massima
- confrontare tale etichetta con quella del nodo dato:  
se superiore,
  - scambiare le due etichette
  - applicare ricorsivamente la procedura al nodo figlio

```
aggiornaheap(V,n,i)                                T(h)
{
  s := 2 * i; { figlio sinistro }                  Θ(1)
  d := 2 * i + 1; { figlio destro }                Θ(1)
  iMax := i;                                       Θ(1)
  if (s <= n && A[s] > A[iMax]) iMax = s;           Θ(1)
  if (d <= n && A[d] > A[iMax]) iMax = d;           Θ(1)
  if (iMax != i)                                    Θ(1)
  {
    Scambia(&A[i],&A[iMax]);                       Θ(1)
    aggiornaheap(V,n,iMax);                         T(h - 1)
  }
}
```

Quindi complessità  $\Theta(\log n)$

La costruzione di un *heap* su un vettore dato consiste nel

- aggiornare gli *heap* costituiti dai suoi sottoalberi
- in ordine inverso, perché l'aggiornamento presuppone la correttezza di tutti gli *heap* ai livelli inferiori (quindi con indici successivi)
- trascurando le foglie, perché sono certamente *heap* corretti

```
creaheap(V,n)
{
    for (i = n/2; i >= 1; i--)
        aggiornaheap(V,n,i);
}
```

Parte da  $n/2$  anziché da  $n$  perché un nodo ha figli se e solo se  $2i \leq n$  e i nodi senza figli certamente rispettano le proprietà

L'analisi di complessità (non elementare) mostra che  $T(n) \in \Theta(n)$

Si può quasi dire che sia lo stesso algoritmo con strutture dati diverse

```
SelectionSort(V,n)
```

```
{  
  
  for (j = n; j > 1; j--)  
  {  
    i = TrovaIndiceMassimo(V,j);  
    Scambia(&V[i],&V[j]);  
  }  
}
```

```
HeapSort(V,n)
```

```
{  
  creaheap(V,n);  
  for (j = n; j > 1; j--)  
  {  
    i = 1;  
    Scambia(&V[i],&V[j]);  
    aggiornaheap(V,j-1,1);  
  }  
}
```

così che la complessità temporale scende da  $\Theta(n^2)$  a  $\Theta(n \log n)$

# Ordinamento per fusione (*MergeSort*)

Dato un vettore  $V$ , sia  $V[s, d]$  il sottovettore contenente gli elementi di  $V$  con indici compresi fra  $s$  e  $d$

L'algoritmo *MergeSort* applica la strategia detta *divide et impera*

- **divide**: suddivide il vettore in due sottovettori di  $n/2$  elementi calcolando l'indice mediano (*è il modo più semplice, non l'unico!*)
- **impera**: ordina i sottovettori ricorsivamente, lasciando invariati quelli di lunghezza  $\leq 1$  (**caso base**)
- **combina**: fonde i due sottovettori producendone uno solo ordinato

# MergeSort: pseudocodice

```
MergeSort(V, s, d)
{
  if (s < d)
  {
     $m = \left\lfloor \frac{s + d}{2} \right\rfloor$ ;
    MergeSort(V, s, m);
    MergeSort(V, m+1, d);
    Merge(V, s, m, d);
  }
}
```

I due sottovettori sono sempre più piccoli del vettore  $V$  grazie a

- l'arrotondamento per difetto
- la divisione in  $V[s, m]$  e  $V[m + 1, d]$

Senza questa combinazione la ricorsione proseguirebbe all'infinito (esaurendo lo *stack*)

(per esempio, si consideri  $s = 5$  e  $d = 6$ )

L' algoritmo funziona per induzione matematica (forte)

- nel caso base ( $s \geq d$ ), il vettore  $V$  è ordinato  
Dunque l' algoritmo funziona su vettori di  $n \leq 1$  elementi
- nel caso ricorsivo
  - ① i due sottovettori sono strettamente più corti  
Dunque  $n_1 < n$  e  $n_2 < n$
  - ② l' ipotesi induttiva garantisce che siano ordinati  
Se l' algoritmo ordina tutti i vettori di qualsiasi lunghezza  $\ell < n \dots$
  - ③ la funzione *Merge* garantisce che il risultato sia ordinato  
 $\dots$  l' algoritmo ordina anche qualsiasi vettore di lunghezza  $n$

L' algoritmo ordina qualsiasi vettore  $V$  di qualsiasi lunghezza  $n$



La procedura ricombina le soluzioni dei due sottoproblemi

Parte con le due metà del vettore  $V$  ordinate al proprio interno e un vettore ausiliario  $B$  vuoto:

- 1 finché i due sottovettori sono non vuoti
  - confronta i loro elementi minimi
  - sposta il minore dei due in fondo al vettore ausiliario  $B$
- 2 quando uno dei due sottovettori è vuoto
  - copia l'altro in fondo al vettore ausiliario  $B$
- 3 quando sono vuoti entrambi
  - ricopia il vettore ausiliario  $B$  sul vettore iniziale  $V$

La complessità temporale è ovviamente lineare  $\Theta(n)$

# MergeSort: complessità

```
MergeSort(V, s, d)
{
  if (s < d)                                 $\Theta(1)$ 
  {
     $m = \left\lfloor \frac{s+d}{2} \right\rfloor;$            $\Theta(1)$ 
    MergeSort(V, s, m);                        $T_{MS}(n_1)$ 
    MergeSort(V, m+1, d);                      $T_{MS}(n_2)$ 
    Merge(V, s, m, d);                          $\Theta(n)$ 
  }
}
```

È facile mostrare che  $n = d - s + 1 = n_1 + n_2$  con

$$n_1 = m - s + 1 = \left\lfloor \frac{n}{2} \right\rfloor \quad n_2 = d - m = \left\lceil \frac{n}{2} \right\rceil$$

# MergeSort: complessità

Per semplicità consideriamo sequenze in cui  $n$  è una potenza di 2

- i due sottovettori hanno entrambi lunghezza  $n/2$

Di conseguenza

- quando  $n = 1$ , il tempo di calcolo è  $T(1) \in \Theta(1)$
- quando  $n > 1$ , il tempo di calcolo  $T(n)$  è la somma di
  - 1  $\Theta(1)$  per il calcolo dell'indice mediano (divide)
  - 2  $2T(n/2)$  per la soluzione dei due sottoproblemi (impera)
  - 3  $\Theta(n)$  per la ricomposizione della soluzione (combina)

$$T_{MS}(n) = \begin{cases} \Theta(1) & \text{per } n = 1 \\ 2T_{MS}(n/2) + \Theta(n) & \text{per } n > 1 \end{cases}$$

Si dimostra che  $T_{MS}(n) \in \Theta(n \log n)$

# Ordinamento “veloce” (*QuickSort*)

Dato un vettore  $V$ , sia  $V[s, d]$  il sottovettore contenente gli elementi di  $V$  con indici compresi fra  $s$  e  $d$

L'algoritmo *QuickSort* applica la strategia detta *divide et impera*

- **divide**: suddivide il vettore in due sottovettori  
(in modo diverso da *MergeSort*!)
- **impera**: ordina i sottovettori ricorsivamente, lasciando invariati quelli di lunghezza  $\leq 1$  (**caso base**)
- **combina**: fonde i due sottovettori producendone uno solo ordinato

La strategia è la stessa del *MergeSort*, ma i due sottovettori contengono

- 1 gli elementi non superiori a un valore di soglia
- 2 gli elementi superiori a un valore di soglia

Il valore soglia è il valore di un elemento specifico (elemento *pivot*), che viene estratto e poi reinserito in mezzo fra i due sottovettori ordinati

# QuickSort: pseudocodice

La suddivisione diversa implica che:

- 1 la fase "divide" è più sofisticata:  
sposta gli elementi raccogliendone alcuni al principio e altri alla fine
- 2 la fase "combina" è banale:  
basta concatenare i due sottovettori ordinati

```
QuickSort(V,s,d)
{
  if (s < d)
  {
    q = Partition(V,s,d);      { q è la posizione finale del pivot }
    QuickSort(V,s,q-1);
    QuickSort(V,q+1,d);
  }
}
```

I due sottovettori sono più piccoli di  $V$  perché non contengono il *pivot*  
(questo evita la ricorsione infinita)

## L'algoritmo funziona per induzione matematica (forte)

- nel caso base ( $s \geq d$ ), il vettore  $V$  è ordinato  
Dunque l'algoritmo funziona su vettori di  $n \leq 1$  elementi
- nel caso ricorsivo
  - ① escludendo il *pivot*, i sottovettori sono strettamente più corti di  $V$   
Dunque  $n_1 < n$  e  $n_2 < n$
  - ② la funzione *Partition* garantisce che i valori del primo sottovettore siano non superiori al *pivot* e quelli del secondo superiori
  - ③ l'ipotesi induttiva garantisce che siano ordinati  
Se l'algoritmo ordina tutti i vettori di qualsiasi lunghezza  $\ell < n \dots$
  - ④ il concatenamento crea banalmente un vettore ordinato  
 $\dots$  l'algoritmo ordina anche qualsiasi vettore di lunghezza  $n$

L'algoritmo ordina qualsiasi vettore

# QuickSort: Partition

La procedura sceglie un elemento *pivot* che faccia da separatore e poi crea sul vettore  $V$  due sottovettori

- uno con gli elementi più piccoli del *pivot*
- uno con gli elementi più grandi del *pivot*

*Ci sono molti modi di scegliere il pivot (per esempio  $V[s]$ )*

Parte con due sottovettori vuoti all'inizio di  $V$  (elementi piccoli e grandi)

Considera ogni elemento di  $V$  e lo sposta nella tabella giusta

- se maggiore del *pivot*, allarga la seconda tabella a includerlo
- se minore del *pivot*
  - 1 scambia l'elemento col primo della seconda tabella e allarga quest'ultima
  - 2 allarga la prima tabella a includere l'elemento togliendolo alla seconda

La complessità temporale è ovviamente lineare  $\Theta(n)$

# Uno schema mnemonico

I diversi algoritmi visti sinora sono legati da alcune idee di fondo

- 1 il modo di procedere; ci sono algoritmi che
  - a) **costruiscono un sottovettore ordinato e lo ricombinano con gli elementi esterni**, uno alla volta
  - b) **costruiscono due sottovettori, li ordinano e li ricombinano**
- 2 la distribuzione dello sforzo; ci sono algoritmi che impiegano
  - a) **poco sforzo nella costruzione dei sottovettori ordinati e molto sforzo nella ricombinazione**
  - b) **molto sforzo nella costruzione dei sottovettori ordinati e poco sforzo nella ricombinazione**

	Costruzione banale e ricombinazione sofisticata	Costruzione sofisticata e ricombinazione banale
Allargamento di un sottovettore	InsertionSort	SelectionSort (HeapSort)
Fusione di due sottovettori	MergeSort	QuickSort



Vedi file `InsSort-SelSort-Ricorsive.pdf`