Algoritmi e Strutture Dati

by Andrea Falbo

Indice:

Algoritmi e Strutture Dati

Introduzione

- **Algoritmo**:
  - Sequenza finita di istruzioni che, eseguita a partire da un insieme di dati, produce, in un n° finito di passi, altri dati.
  - I dati devono essere rappresentati in modo finito, ma non devono necessariamente essere di quantità finita.
  - Ogni istruzione non deve essere ambigua: aggiungere 16 g di sale ✓
    aggiungere sale q.b. ✗
  - I dati prodotti sono in relazione con i dati a partire dai quali l'esecuzione di istruzioni si applica, dunque mediante una funzione $f: I \to O$.
  - L'algoritmo deve essere dunque deterministico, cioè con gli stessi dati e le stesse istruzioni produce sempre lo stesso output

- **Problema Computazionale**:
  - Data un'istanza, trovare una soluzione attraverso la funzione:
    f: Istanza → Soluzione

  Es: Istanza $x \in \mathbb{N}$, Soluzione $f(x): \mathbb{N} \times \mathbb{N}$ t.c. $\forall x \in \mathbb{N}, f(x) = 2^x$

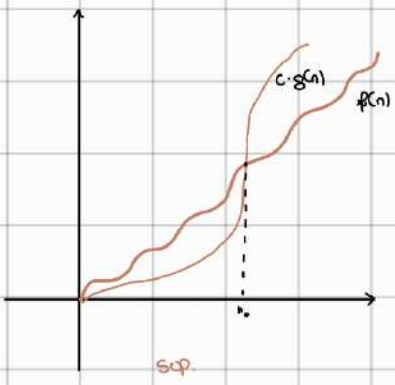
- **Tempi di Calcolo**: Dato x di dimensione h, come varia il tempo al variare di h

Algoritmi e Strutture Dati

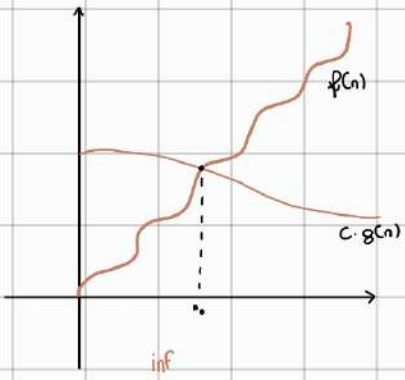
Notazioni Asintotiche

- Limite Superiore Asintotico: $O(g(n)) = \{f : \exists c, n_0 > 0, \forall n > n_0 \quad 0 \leq f(n) \leq c \cdot g(n)\}$
- Limite Inferiore Asintotico: $\Omega(g(n)) = \{f : \exists c, n_0 > 0, \forall n > n_0 \quad 0 \leq c \cdot g(n) \leq f(n)\}$
- Limite Asintotico Stretto: $\Theta(g(n)) = \{f : \exists c_1, c_2, n_0 > 0, \forall n > n_0 \quad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$

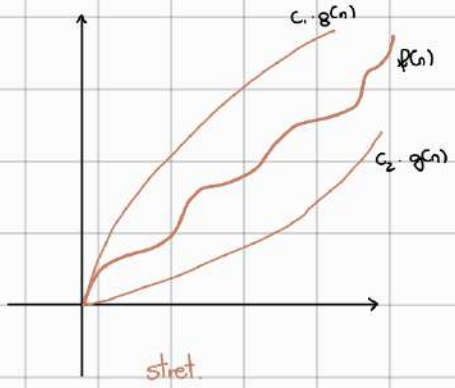
Scalas di Asintoti: $\underbrace{1}_{\text{costanti}}, \log n, n, n \log n, n^2, n^3, \ldots, \underbrace{2^n, 3^n, \ldots, n!, n^n}_{\text{non trattabili in termini teorici}}$



sup.



inf



stret.

Complessità Computazionale: Numero di istruzioni, eventualmente pesate, eseguite da un algoritmo su un input. Dipende da:

- Dimensione input
- Specificità della struttura dell'input

Casi:

- Caso peggiore: $T_A : \mathbb{N} \to \mathbb{R}^* \quad \forall n \in \mathbb{N}, T_A(n) = \max\{T_A^*(x) : x \in I, \dim(x) = n\}$
- Caso migliore: $t_A : \mathbb{N} \to \mathbb{R}^* \quad \forall n \in \mathbb{N}, t_A(n) = \min\{t_A^*(x) : x \in I, \dim(x) = n\}$
- Caso medio: $\bar{t}_A : \mathbb{N} \to \mathbb{R}^* \quad \forall n \in \mathbb{N}, \bar{t}_A(n) = \text{med}\{t_A^*(x) : x \in I, \dim(x) = n\}$

$T_A^*(x)$ è il numero di istruzioni eseguite da A con input x

$x \in I$ è dimensione input

$n \in \mathbb{N}$ n° di istruzioni eseguite da A su input $n \in \mathbb{N}$

Algoritmi e Strutture Dati

Esempio

1. $f(n) = 2^5$

   $g(n) = n^6$

   $f(n) = O(n^6)$?

   $\exists\, c>0, n_0>0$ tc $\forall n > n_0$ $2n^5 \leq c \cdot n^6$?

   $c = 2$

   $2n^5 \leq 2n^6$

   $n_0 = 1 \rightarrow n \geq 1$ omissione

   $1 \leq n$

   $n \geq 1$? Sì ✓

2. $f(n) = 2^5$

   $g(n) = n^5$

   $f(n) = O(n^5)$?

   $\exists\, c>0, n_0>0$ tc $\forall n > n_0$ $2n^5 \leq c \cdot n^5$?

   $c = 2$

   $2n^5 \leq 2n^5$

   $n_0 = 1 \rightarrow n \geq 1$ omissione

   $1 \leq 1$ ✓

Problema dell'ordinamento di numeri interi

$I = \{ v \in \bigcup \mathbb{Z}^n \}$, $\Theta = I$

$f: I \rightarrow \Theta$

$\forall v \in I$, $f(v)\, v' \in \Theta$ tc.

- v' permutazione di v
- $v_1' \leq v_2' \leq \dots \leq v_n'$

Istanza $v \in I$

Soluzione $f(v) \in \Theta$

Vogliamo trovare uno o più algoritmi che realizzano questa funzione:

- Selection Sort
- Bubble-Sort
- Insertion Sort
- Merge-Sort (ricorsivo)

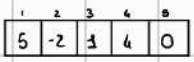
Definizioni per Algoritmi

- In loco: Se utilizza un numero costante di variabili indipendentemente dalla dimensione dell'input. Es: QuickSort in loco.
- Stabile: Se mantiene inalterato l'ordine relativo di elementi uguali: $z_a < z_b < z_c$. Es: Merge-Sort stabile

Algoritmi e Strutture Dati

Selection Sort

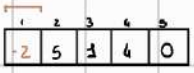
Metodo Risolutivo: Sia n la lunghezza di V.



n=5

- Trovo il minimo in V[1,...,n]

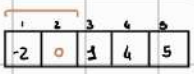
- Scambio con V[1]



V[1,...,1] ordinato

- Trovo il minimo in V[2,...,n]

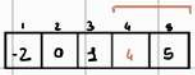
- Scambio con V[2]



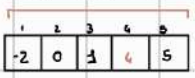
V[1,...,2] ordinato

- ...

- Trovo il minimo in V[n-1,n]



- Scambio con V[n-1]



V[1,...,n] ordinato

Algoritmo: SELECTION-SORT(v)

```
n = length(v)
for i = 1 to n-1                    // per la ricerca dei minimi
    posmin = i                      // assumo che sia nella 1° casella
    for j = i+1 to n                // scorrimento per inserimento
        if V[j] < V[posmin]         // se l'elemento successivo è più piccolo del minimo
            posmin = j              // aggiorno
    if posmin ≠ i                   // se posmin non è il valore di default
        tmp = V[i]                  /*
        V[i] = V[posmin]              effettuo sostituzione
        V[posmin] = tmp                                    */
```

Algoritmi e Strutture Dati

Selection Sort

Tempi di calcolo: SELECTION-SORT(v)

	Caso migliore	Caso peggiore
n = length(v)	1	1
for i = 1 to n-1	n	n
posmin = i	n-1	n-1
for j = i+1 to n	*	*
if V[j] < V[posmin]	**	**
posmin = j	∅	***
if posmin ≠ i	n-1	n-1
tmp = V[i]	∅	n-1
V[i] = V[posmin]	∅	n-1
V[posmin] = tmp	∅	n-1

* Ciclo for annidato:

$i = 1 \rightarrow j = 2..n$

$i = 2 \rightarrow j = 3..n$

$\rightarrow i, j = \overbrace{i+1}^{\text{da}}..\overbrace{n}^{\text{a}} \longrightarrow$ b - a + 1 $\rightarrow n-(i+1)+1+1 = n-i+1$ (uscire dal ciclo)

$$\sum_{i=1}^{n-1}(n-i+1) = (n-1)\cdot n - \frac{(n-1)\cdot n}{2} + n - 1 = \frac{1}{2}(n-1)\cdot n + n - 1$$

** Contenuto del ciclo for annidato, quindi: *-1

$$\sum_{i=n}^{n-1}(n-i) = \frac{1}{2}(n-1)\cdot n$$

*** $i \rightarrow n-(2i-1)$

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

Algoritmi e Strutture Dati

Selection Sort

Casi:

Caso migliore: Vettore già ordinato. $\{\sigma \in I \mid v_1 \leq v_2 \leq \dots \leq v_n\}$

$t(n) = 1 + n + n-1 + n^2 - 1 + n - 1 = \Theta(n^2)$

Caso peggiore: Vettore ordinato al contrario $\{\sigma \in I \mid v_1 > v_2 > \dots > v_n\}$

$T(n) = 1 + n + n-1 + n^2 - 1 + n - 1 + n-1 + n-1 + n-1 = \Theta(n^2)$

→ Caso medio: $\bar{t}(n) = \Theta(n^2)$

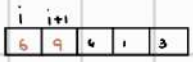
Note: L'algoritmo non capisce quando il vettore è già ordinato. Poca differenza tra caso migliore e peggiore.

Algoritmi e Strutture Dati

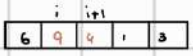
Bubble Sort

Metodi Risolutivi: Sia n la lunghezza di V.

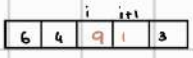
Scorro V[1,...,n-1] e sostituisco con successivo se quest'ultimo inferiore. Ottengo ogni volta l'elemento maggiore in fondo.



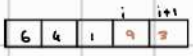
$6 < 9$? si → avanzo i



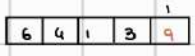
$9 < 4$? no → scambio 9 con succ



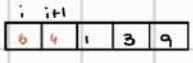
$9 < 1$? no → scambio 9 con succ



$9 < 3$? no → scambio 9 con succ



9 è il maggiore, riporto da capo



$6 < 4$? no → scambio 6 con succ.

...

Vettore ordinato

Algoritmo: BUBBLE-SORT(v)

```
n = length(v)
for i = 1 to n-1                    // scansione per aggiornamento posizioni
    for j = 1 to n-i                // scansione per scambio elemento
        if V[j+1] < V[j]            // Se l'elemento successivo è più piccolo
            tmp = V[j+1]            /*
            V[j+1] = V[j]           scambio
            V[j] = tmp              */
```

Algoritmi e Strutture Dati

Bubble Sort

Tempi di Calcolo : BUBBLE-SORT(V)

	Caso migliore	Caso peggiore
n = length(V)	1	1
for i = 1 to n-1	n	n
for j = 1 to n-i	*	*
if V[j+1] < V[j]	**	**
tmp = V[j+1]	ø	**
V[j+1] = V[j]	ø	**
V[j] = tmp	ø	**

* $i, j = 1, \dots, n-i = n-i-1+1+1 = n-i+1$

$$\sum_{i=1}^{n-1} (n-i+1) = \frac{1}{2}(n-1)\cdot n + n-1$$

** il contenuto quindi come $*-1$

$$\sum_{i=1}^{n} (n-i) = \frac{1}{2}(n-1)\,n$$

Casi:

Caso migliore: vettore già ordinato $\{v \in I \mid v_1 \le v_2 \le \dots \le v_n\}$

$$t(n) = 3 + n + \frac{1}{2}(n-1)\cdot n + n - 1 + \frac{1}{2}(n-1)\cdot n = \Theta(n^2)$$

Caso peggiore: vettore ordinato al contrario $\{v \in I \mid v_1 > v_2 > \dots > v_n\}$

$$T(n) = 3 + n + \frac{1}{2}(n-1)\cdot n + n - 1 + \frac{1}{2}(n-1)\cdot n + 3\left(\frac{1}{2}(n-1)n\right) = \Theta(n^2)$$

Caso medio: $\bar{t}(n) = \Theta(n^2)$

Note: Non abbiamo migliorato. Ma se dopo il 1° ciclo ci accorgiamo di non aver cambiato nulla, il vettore sarebbe già ordinato e, interrompendo, avremmo caso migliore $= \Theta(n)$

Algoritmi e Strutture Dati

Insertion Sort

Metodo Risolutivo: Sia n la lunghezza di V

Considero $K = V[2]$

Inserisco in $V[1,..,1]$ in modo tale che $V[1,..,2]$ sia ordinato.

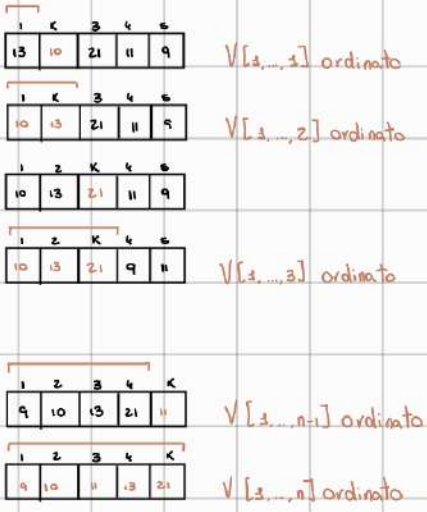
Considero $K = V[3]$

Inserisco in $V[1,..2]$ in modo tale che $V[1,..,3]$ sia ordinato

...

Considero $K = V[n]$

Inserisco in $V[1,..n-1]$ in modo tale che $V[1,..,n]$ sia ordinato.



Algoritmo:

```
INSERTION-SORT(V)
    n = length(V)
    for i = 2 to n
        K = V[i]
        j = i - 1                       // elemento per confronto, il precedente
        while j > 0 and V[j] > K        // se non sono in 1° posizione (j valido) ed è > K
            V[j+1] = V[j]               // sposto in avanti j
            j--                         // rifaccio tutto guardando l'elemento primo: j-1.
        V[j+1] = K
```

Algoritmi e Strutture Dati

Insertion Sort

Tempi di Calcolo: INSERTION-SORT(v)

	Caso migliore	Caso peggiore
n = length(v)	1	1
for i = 2 to n	n	n
K = V[i]	n-1	n-1
j = i - 1	n-1	n-1
while j > 0 and V[j] > K	*	*
V[j+1] = V[j]	*-1	*-1
j--	*-1	*-1
V[j+1] = K	n-1	n-1

* : dato i, T_i è quante volte viene eseguito il test del while

$$\sum_{i=2}^{n} T_i$$

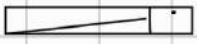
Algoritmi e Strutture Dati

Insertion Sort

Casi:

Caso migliore: $\{v \in I \mid v_1 \leq v_2 \leq \dots \leq v_n\}$

$t_i = 1$, perché $K > V[j]$



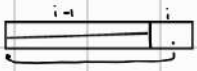
* $\sum_{i=2}^{n} t_i = n \cdot 2 + 1 = n-1$

* -1 : $t_i - 1 = 0$

$$t(n) = 1 + n + n - 1 + n - 1 + n - 1 + 0 + 0 + n - 1 = \Theta(n)$$

Caso peggiore: $\{v \in I \mid v_1 > v_2 > \dots > v_n\}$

$t_i = i - 1 + 1 = i$



* $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$

* -1 $\sum_{i=2}^{n} i - 1 = \frac{n(n+1)}{2} - n$

$$T(n) = 1 + n + n - 1 + n - 1 + \frac{n(n+1)}{2} - 1 + 2\left(\frac{n(n+1)}{2} - n\right) + n - 1 = \Theta(n^2)$$

Caso medio: $\overline{t}(n) = \frac{A}{2} = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) = \Theta(n^2)$

Note: Ora nel caso migliore abbiamo n e n^2. Questo grazie al while che non cicla un n° finito di volte ma solo in determinati casi.

Algoritmi e Strutture Dati

Divide et Imperas.

Prima di parlare del merge-sort, è bene capire come funziona la tecnica di ordinamento che utilizza: la divide et impera (et combina)

P è un problema computazionale, x ∈ I

- Divide: Dividere P in K sottoproblemi, ciascuno dei quali non è altro che P applicato su un input più piccolo di x e consiste di una delle K parti di cui x è composto.

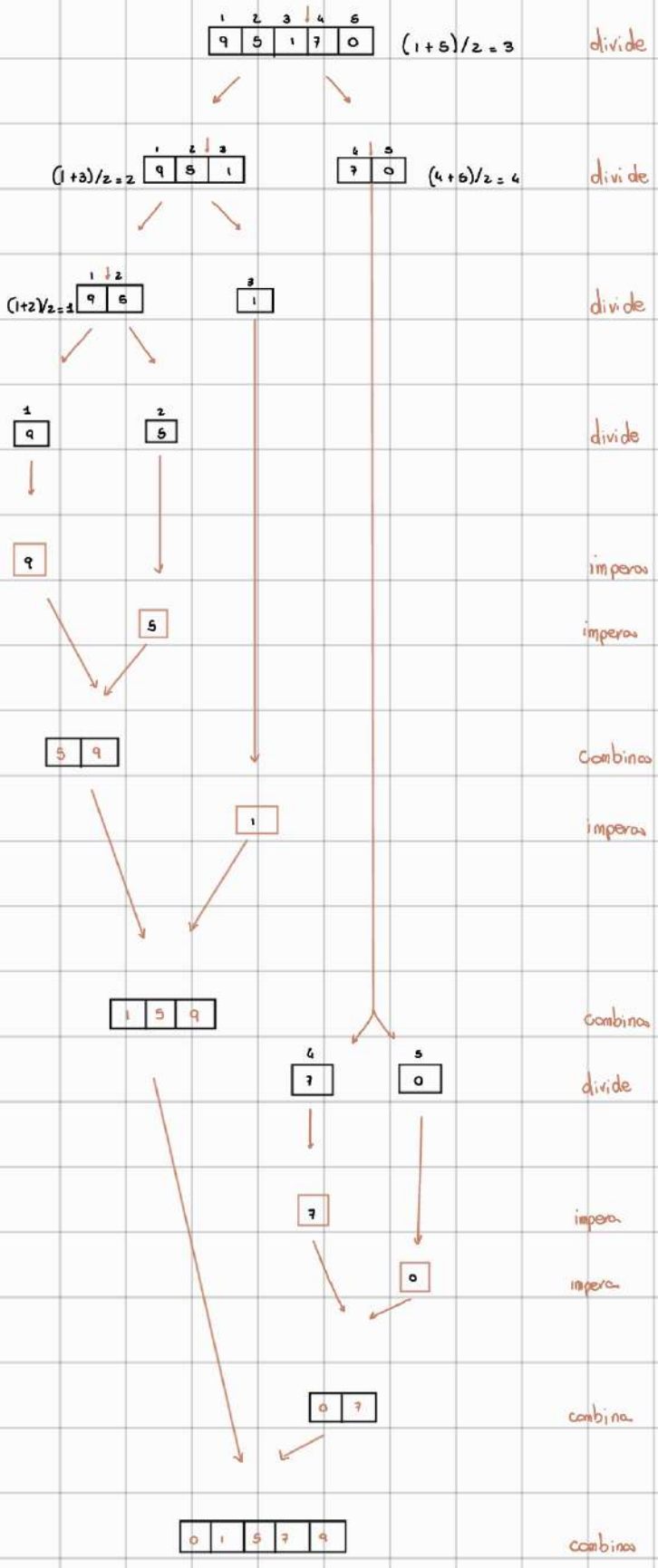
- Impera: Risolvere ricorsivamente i K sottoproblemi utilizzando questi a passi invocati K volte ciascuna su una delle K parti di cui x è composto.

- Combina: Si combinano opportunamente le K soluzioni dei K sottoproblemi per costruire la soluzione P su x.

Algoritmi e Strutture Dati

Merge Sort

Simulazione Esecuzione:



Algoritmi e Strutture Dati

Merge Sort

Algoritmo Merge-Sort: MERGE-SORT (V, p, q)

```
if p<q
    m = ⌊(p+q)/2⌋                    // divide
    MERGE-SORT (V, p, m)             // impero
    MERGE-SORT (V, m+1, q)           // impero
    MERGE (V, p, m, q)               // combino
```

Algoritmo Merge (combina): MERGE (V, p, m, q) // per il combina ci servirà un array di appoggio e due puntatori

```
sx=p , dx = m+1 , i=p                // sx e dx contano le frecce
while sx ≤ m AND dx ≤ q              // fino a quando siamo nelle sottometà
    if V[sx] ≤ V[dx]                 // se il valore a sx ≤ dx
        W[i] = V[sx]                 // lo inserisco nell array di appoggio
        sx ++                        // sposto di 1 pos. la freccia
    else                             // altrimenti  V[dx] < V[sx]
        W[i] = V[dx]                 // inserisco il valore di V[dx]
        dx ++                        // incremento freccia
    i ++                             // incremento posizione nell'array
while sx ≤ m                         // se dx ha finito le posizioni ma sx non ancora
    W[i] = V[sx]                     // inserisco gli elementi
    sx ++, i++
while dx ≤ q                         // se sx ha finito le posizioni ma dx non ancora
    W[i] = V[dx]                     // inserisco gli elementi
    dx ++ , i++
for j=p to q                         // scorro tutte le posizioni
    V[j] = W[j]                      // per ricopiare nel mio array i valori ordinati
```

Algoritmi e Strutture Dati

Merge Sort

Tempi di Calcolo: L'algoritmo è fatto in modo tale che le procedure vengano sempre eseguite.

$t(n) = \bar{t}(n) = T(n)$

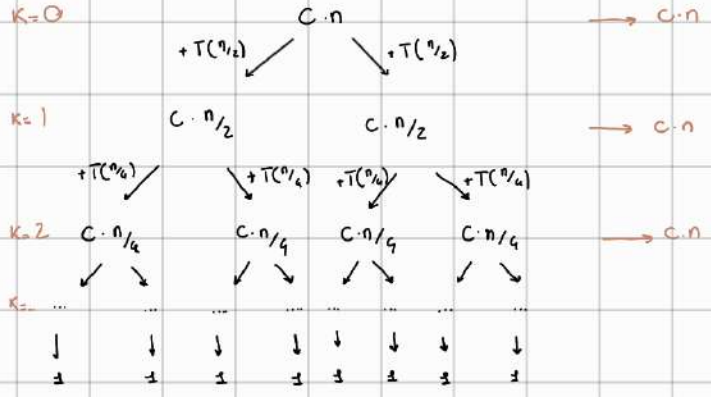
$T(n)$ costo dell'invocazione su $V[1...,n]$

- Caso base: $n = 1 \rightarrow T(n) = 1$
- Caso passo: $n > 1 \rightarrow T(n) = 1 + 1 + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$

(controllo, m, divide, divide, merge)

Risolviamo l'equazione di ricorrenza:

$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n/2) + c \cdot n \end{cases}$$



quindi: $K = 1 \rightarrow \frac{n}{2} \cdot 2$, $K = 2 \rightarrow \frac{n}{2^2} \cdot 2^2$, $K = 3 \rightarrow \frac{n}{2^3} \cdot 2^3$

$\frac{n}{2^K} = 1 \rightarrow 2^K = n \rightarrow K = \log_2 n$

$$T(n) = c \cdot n \cdot \log_2 n = \Theta(n \cdot \log n)$$

N.b.: Si poteva dimostrare che fosse $\Theta(n \log n)$ attraverso il teorema dell'esperto, esposto alla prossima pagina.

Algoritmi e Strutture Dati

Teorema dell' Esperto

Dato $T(n) = a \cdot T(n/b) + f(n)$ con $n \in \mathbb{N}$, $a \geq 1$, $b > 1$:

1. se $\exists\, \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon}) \rightarrow T(n) = \Theta(n^{\log_b a})$
2. se $f(n) = \Theta(n^{\log_b a}) \rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$ (merge-sort)
3. se $\exists\, \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon})$ e $\exists\, c < 1 : f(\frac{n}{b}) \leq c \cdot f(n)$ definitivamente $\rightarrow T(n) = \Theta(f(n))$

eg. $T(n) = 2T(n/2) + 3n - 1$

$a = 2$

$b = 2$

$f(n) = 3n - 1$

$\log_b a = \log_2 2 = 1 \rightarrow$ · $3n - 1 \stackrel{?}{=} O(n^{1-\varepsilon})$ $\nexists\, \varepsilon$

· $3n - 1 \stackrel{?}{=} \Theta(n^1)$ ✓ $\rightarrow \Theta(n^1 \cdot \log_n) \rightarrow \Theta(n \log n)$

Algoritmi e Strutture Dati

QuickSort

Metodo Risolutivo: Tecnica divide et impera.

1. Sceglie un elemento di V detto pivot, chiamato K.
2. Partiziona l'array V[l,...,r] in V[l,...,q] e V[q,...,r] con q da determinare in modo tale che, riarrangiando gli elementi di V mediante scambi si ha che:
   - tutti gli elementi in V[l,...,q] siano $\leq K$
   - tutti gli elementi in V[q,...,r] siano $\geq K$
3. Ordina le due parti richiamandosi
4. Combina non fa nulla

Algoritmo: QUICKSORT(V,l,r)

```
if l<r
    q = PARTITION(V,l,r)
    QUICKSORT(V,l,q)
    QUICKSORT(V,q,r)
```

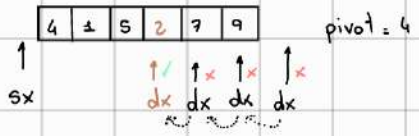
Partizionamento: Vedremo la tecnica di Hoare. Ci sono partizionamenti migliori, come quello di Lomuto, ma non lo vedremo

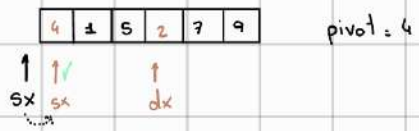
- 1 Impone K=V[l]
- 2 Scansiona V da destra a sinistra, fermandosi sul 1° el. $\leq K$.
- 3 Scansiona V da sinistra a destra, fermandosi sul 1° el. $\geq K$.
- 4 Scambia gli elementi e riprende la scansione sse i due indici non si sono sovrapposti.

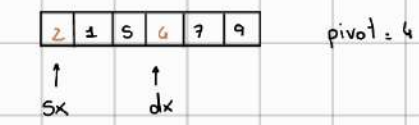
Algoritmi e Strutture Dati

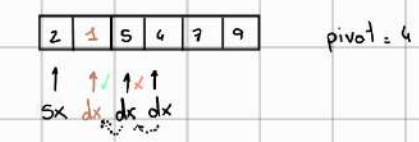
QuickSort

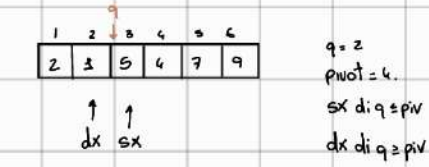
Dimostrazione

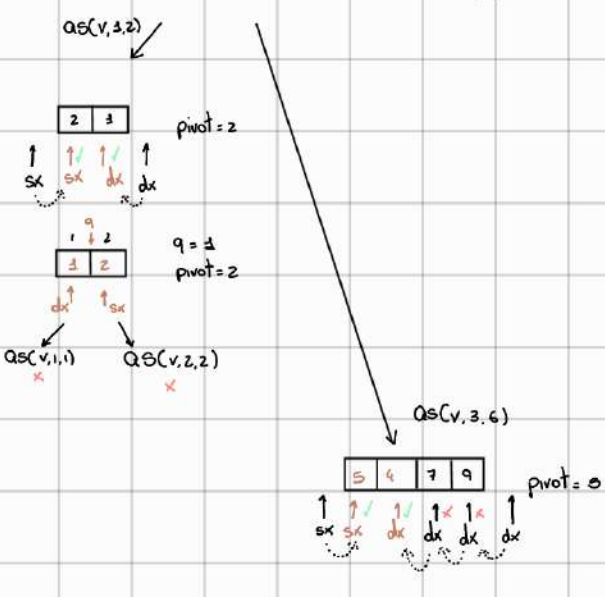


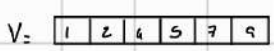


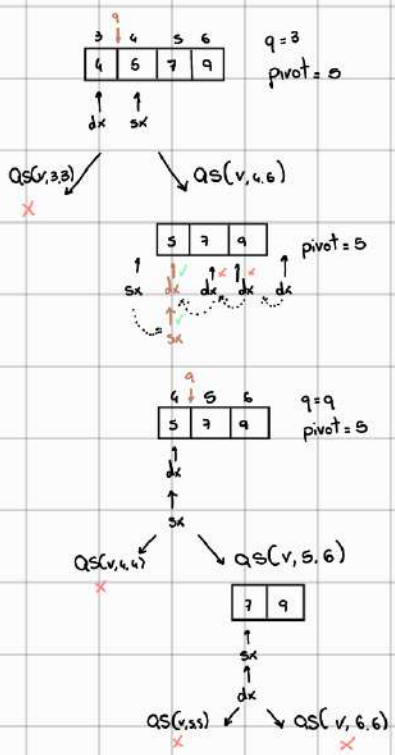












Algoritmi e Strutture Dati

QuickSort

Algoritmo QuickSort : QUICKSORT (V, l, r)

```
if l < r
    q = PARTITION (V, l, r)
    QUICKSORT (V, l, q)
    QUICKSORT (V, q, r)
```

Algoritmo Partition Hoare : PARTITION (V, l, r)

```
pivot = V[l], sx = l - 1, dx = r + 1
while sx ≤ dx
    do                                  // scansioni parziali per ottenere l'elemento da dx
        dx --
    while V[dx] > pivot
    do                                  // scansioni parziali per ottenere l'el da sx
        sx ++
    while V[sx] < pivot
    if sx < dx                          // se non ho sovrapposto gli indici
        scambia (V[sx], V[dx])          // scambio
return dx
```

Algoritmi e Strutture Dati

QuickSort

Tempi di Calcolo: $\Theta(n \cdot k + 1)$

1

lungh. spostamenti = lungh. array

partition: $\Theta(n - 1 + 1) = \Theta(n)$

$T(n) = T(q) + T(n-q) + \Theta(n)$

V più sbilanciata possibile 

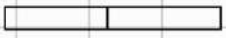
$T(n) = T(1) + T(n-1) + \Theta(n)$

$T(n-1) = T(1) + T(n-2) + \Theta(n-1)$

$T(n-2) = \dots$

$= \underbrace{T(1) + \dots + T(1)}_{n-1 \text{ volte}} + \underbrace{\Theta(2) + \Theta(3) + \Theta(n-1) + \Theta(n)}_{n-2+1}$

$$\sum_{i=2}^{n} \Theta(i) = \Theta\left(\frac{n(n+1)}{2} - 1\right) = \Theta(n^2)$$

V più bilanciata possibile 

$T(n) = 2T(n/2) + \Theta(n)$ ← (come merge-sort)

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

CASO PESSIMO $\Theta(n^2)$

sbilanciata

CASO MEDIO $\Theta(n^2)$

anche se meno sbilanciato,

non lo è per un n° costante

CASO MIGLIORE $\Theta(n \log n)$

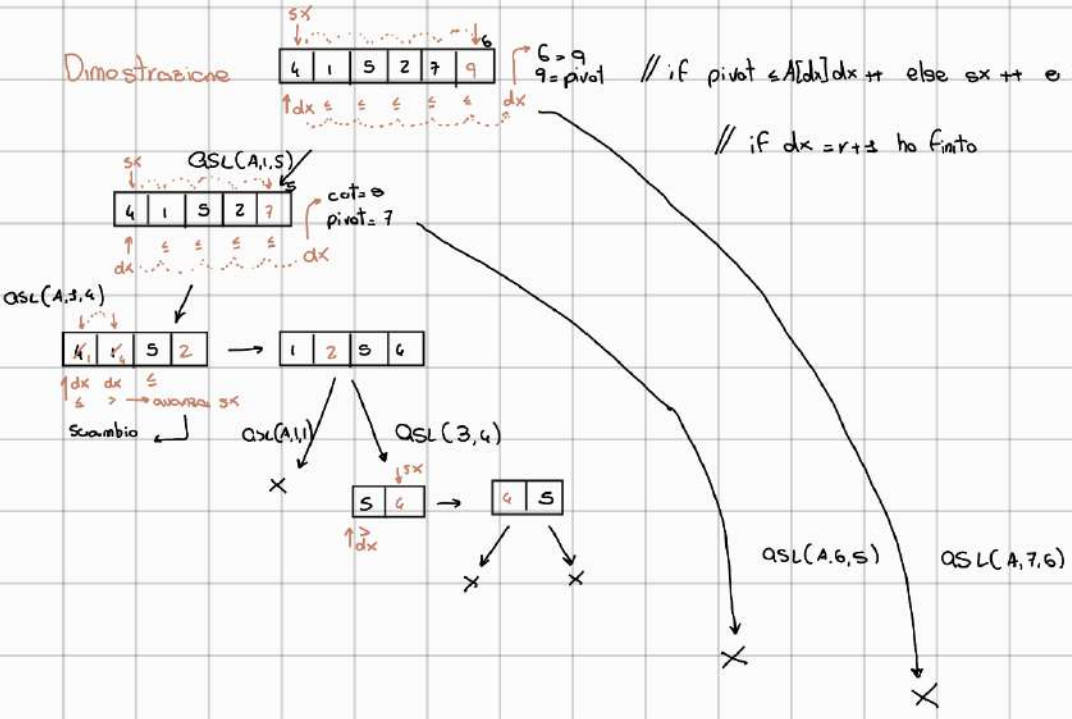
bilanciato

Perché: QuickSort > Merge-Sort? Perché scegliere il 1° se il 2° è sempre $\Theta(n \log n)$?

1. Costanti minori
2. QuickSort in Loco.
3. Cambiando pivot si migliora (es. usando come pivot la mediana oppure un valore random per evitare di aggiungere complessità)

Algoritmi e Strutture Dati

QuickSort con Lomuto



Algoritmo: `CONTO(A, l, r)`

```
pivot = A[r]
sx = l - 1
dx = l
while dx ≤ r
    if A[dx] ≤ pivot
        sx++
        scambio A[dx] con A[sx]
    dx++
return sx
```

```
QUICKSORT (A, l, r)
    if l < r
        q = CONTO (A, l, q)
        QUICKSORT (A, l, q-1)
        QUICKSORT (A, q+1, r)
```

Algoritmi e Strutture Dati

Strutture Dati

Insiemi dinamici, ovvero che variano nel tempo. Sono manipolati da algoritmi in 2 modi:

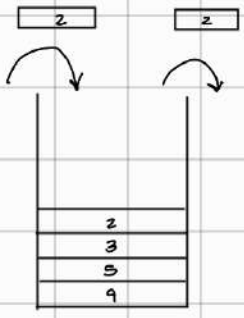
1. operazioni di modifica
2. interrogazioni o query

Algoritmi e Strutture Dati

Pila - Stack

Insieme dinamico su dominio D. Può essere identificato come una sequenza S dove

- $S = <>$ stack vuoto
- $S = <a_1, \dots, a_n>$ sequenza di n elementi tc $\forall i \in \{1, \dots, n\}$, $a_i \in D$ e a_n elemento in cima allo stack



Politica LIFO: Last in First Out.

Operazioni su pila/stack

Sia S l'insieme di tutte le pile su un dominio D.

- Inserimento: PUSH: $S \times D \to S$

  $\forall (s, x) \in S \times D$

  - se $S = <>$, $PUSH(S, x) = <x> \in S$
  - se $S = <a_1, \dots, a_n>$ $PUSH(S, x) = <a_1, \dots a_n, x> \in S$

- Cancellazione: POP: $S \setminus \{<>\} \to S \times D$

  $\forall s \in S$ con $s \neq <>$

  - $POP(S) = (<a_1, \dots a_{n-1}>, a_n)$

  con abuso di notazione, con POP ci riferiremo solo all'el. rimosso, non alla coppia generata

- Interrogazione: STACK-EMPTY: $S \to \{true, false\}$

  $\forall s \in S$

  - STACK-EMPTY(S) $\begin{cases} \text{True} & \text{se } S = <> \\ \text{False} & \text{altrimenti} \end{cases}$

- Interrogazione: TOP: $S \setminus \{<>\} \to D$

  $\forall s \in S$ con $s \neq <>$

  - $TOP(S) = a_n$

Algoritmi e Strutture Dati

Code - Queue

Insieme dinamico su dominio D. Può essere identificato come una sequenza Q dove

- $Q = <>$ coda vuota
- $Q = <a_1, \dots, a_n>$ sequenza di n elementi tc $\forall i \in \{1, \dots, n\}$, $a_i \in D$. a_1 = head di Q, a_n = tail di Q

Politica FIFO: First in First Out.

Operazioni su code / queue

Sia Q l'insieme di tutte le code su un dominio D.

- Inserimento: ENQUEUE: $Q \times D \to Q$

  $\forall (q, x) \in Q \times D$

  se $q = <>$, $\text{ENQUEUE}(q, x) = <x>$

  se $q = <a_1, \dots, a_n>$ $\text{ENQUEUE}(q, x) = <a_1, \dots a_n, x> \in Q$

- Cancellazione: DEQUEUE: $Q \setminus \{<>\} \to Q \times D$

  $\forall q \in Q$ con $q \neq <>$

  $\text{DEQUEUE}(q) = (<a_2, \dots, n>, a_1)$

  con abuso di notazione, con DEQUEUE ci riferiremo solo all'el. rimosso, non alla coppia generata.

- Interrogazione: EMPTY-QUEUE : $Q \to \{true, false\}$

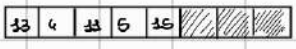
  $\forall q \in Q$

  $\text{empty-queue}(q) \begin{cases} \text{True} & \text{se } q = <> \\ \text{False} & \text{altrimenti} \end{cases}$

Algoritmi e Strutture Dati

Implementazione stack mediante array

Stack $S = \langle a_1, \dots, a_n \rangle$ con al più m elementi ($n \leq m$)

- $A[1 \dots m]$ array con $m = A.length$
- $A.top$ = ultimo el. inserito
- $A[1 \dots A.top]$ memorizza S.
- $A[1]$ elemento in fondo.
- $A.top = 0 \longleftrightarrow S = \langle\rangle$
- Esempio: $S = \langle 13, 4, 11, 5, 15 \rangle$, $m = 8$ $A =$ 


```
STACK-EMPTY(S)
    if A.top = 0
        return true
    return false
```

```
PUSH(S, x)
    if A.top = m
        error "overflow"
    A.top ++
    A[top] = x
```

```
POP(S)
    if Stack-Empty(S)
        error "underflow"
    A.top --
    return A[top+1]
```

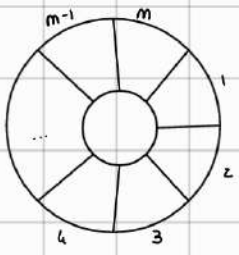
Algoritmi e Strutture Dati

Implementazione code mediante array

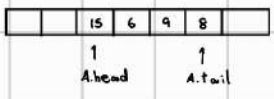
Coda $Q = \langle a_1, \dots, a_n \rangle$ con al più $m-1$ elementi ($n \leq m-1$)

- $A[1 \dots m]$ array con $m = A.length$
- $A.head$ = indice del 1° el. inserito
- $A.tail$ = indice del prossimo el. da inserire
- $A[A.head \dots A.tail - 1]$ memorizza Q
- la posizione 1 segue m secondo un ordine circolare
- $A.head = A.tail \longleftrightarrow Q = \langle\rangle$
- Coda piena sse $n = m-1$ || $A.head = A.tail + 1$ || $A.head = 1$ AND $A.tail = A.length$

Graficamente:



Dimostrazione:

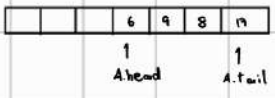


$Q = \langle 15, 6, 9, 8 \rangle$, $m = 7$

ENQUEUE(Q, 17)

$Q = \langle 15, 6, 9, 8, 17 \rangle$ $m = 7$

DEQUEUE(Q)



$Q = \langle 6, 9, 8, 17 \rangle$ $m = 7$

```
EMPTY-QUEUE(Q)
    return A.tail = A.head
```

```
ENQUEUE(Q, x)
    if (A.head = 1 AND A.tail = A.length) OR (A.head = A.tail + 1)
        error "overflow"
    A.[A.tail] = x
    if A.tail = A.length
        A.tail = 1
    else
        A.tail ++
```

```
DEQUEUE(Q)
    if A.head = A
        error "underflow"
    x = A[A.head]
    if A.head = A.length
        A.head = 1
    else
        A.head ++
    return x
```

Algoritmi e Strutture Dati

Esercizi pile

RICERCAELEMENTO(S, x)

```
trovato = false
while not stack_empty(S) ∧ not trovato
    R = pop(S)
    if R = x
        trovato = TRUE
return trovato
```

CONSIDERAZIONI

$T(n) = O(n)$

stiamo distruggendo lo stack!

RICERCA_ELEMENTO(S, x)

```
trovato = FALSE
    while not stack_empty(S) ∧ not trovato
        R = pop(S)
        Push(S2, R)
        if R = x
            trovato = TRUE
    while not stack_empty(S2)
        R = pop(S2)
        Push(S, R)
    return trovato
```

CONSIDERAZIONI

$T(n) = O(n)$

ripristiniamo lo stack ma ci serve uno stack in più.

Algoritmi e Strutture Dati

Esercizi pile

STACKCRESCENTE(S_1, S_2) // S_1 e S_2 decrescenti

```
while not Stack_empty(S1) ∧ not Stack_Empty(S2)
    if TOP(S1) ≤ TOP(S2)
        R = POP(S1)
        PUSH(S3, R)
    else
        R = POP(S2)
        PUSH(S3, R)
while not Stack.Empty(S1)
    R = POP(S1)
    Push(S3, R)
while not Stack.Empty(S2)
    R = POP(S2)
    Push(S3, R)
while not Stack.empty(S3)      // inversione
    R = POP(S3)
    PUSH(S, R)
return S
```

PARENTESI(E)

```
for i = 1 to E.length
    if E[i] = ")" or "]" or "("
        if not Stack_Empty(S) and TOP(S) < E[i]
            return false
        Push(S, E[i])
    else
        if Stack Empty(S)
            return false
        R = POP(S)
        if not R ~ E[i]      (~ : compatibile)
            return FALSE
return Stack.Empty(S)   // se è vuoto parent. aperte = chiuse
                        altrimenti se ≥ no vuoto
```

CONSIDERAZIONI

$T(n+m) = O(n+m) = \Omega(n+m) = \Theta(n+m)$

Algoritmi e Strutture Dati

Esercizi code

CODA PARI(a)

```
while not Queue.Empty(a)
    R=dequeue(a)
    if R mod 2 = 0 then
        Enqueue(ar, R)
    return ar.
```

CODA PARI_IN_LOCO(a)

```
Enqueue(a, -1)      // metto un valore sentinella
while Head(a) ≠ -1
    R= Dequeue(a)
    if R mod 2=0
        Enqueue(a, R)
Dequeue(a)
```

CANCELLI COPPIE>10(a)

```
while not Queue-empty(a)
    R2= Dequeue(a)
    if stack_empty(Sapp)
        Push(Sapp, R2)
    else
        R1 = Pop(Sapp)
        if R1 + R2 ≤ 10
            Push(Sapp, R1)
            Push(Sapp, R2)
while not stack_empty(Sapp)
    R= Pop(Sapp)
    Push(Sapp, R)
while not Stack.Empty(S'app)
    R= Pop(S'app)
    Enqueue(ar, R)
```

Algoritmi e Strutture Dati

Lista / List

È un insieme dinamico su un dominio D.

A ciascun elemento della lista si accede tramite puntatore.

Lista doppiamente concatenata: sequenza di oggetti dove ciascuno contiene:

- Key: il valore
- 2 puntatori: prev e next, per andare nell'oggetto precedente/successivo.
- L.head è il puntatore al 1° oggetto della lista.

L'accesso alla lista è sequenziale, cioè bisogna scorrere tutti gli elementi, non c'è accesso diretto.

Lista semplicemente concatenata: sequenza di oggetti dove ciascuno contiene:

- Key: il valore
- 1 puntatore: next per andare nell'oggetto successivo.
- L.head è il puntatore al 1° oggetto della lista.

L'accesso alla lista è sequenziale, cioè bisogna scorrere tutti gli elementi, non c'è accesso diretto.

Definizione Lista Ricorsiva

1. $nil \in \mathcal{L}$ (lista vuota)
2. $L \in \mathcal{L}$ e $m \in D \rightarrow (m, L) \in \mathcal{L}$
3. Ciò che non è definito in 1 o 2 $\notin \mathcal{L}$

Algoritmi e Strutture Dati

Implementazioni di una Lista

SCAN(L)

```
x = L.head
while x ≠ NIL
    x.Key          // accesso elemento
    x = x.next     // accesso al next
```

LIST_SEARCH(L, K)

```
x = L.head
while x ≠ NIL AND x.key ≠ K
    x = x.next
return x
```

LIST_HEAD_INSERT(L, x)

```
x.next = L.head
if L ≠ <>
    L.head.prev = x
L.head = x
x.prev = NIL
```

LIST_DELETE(L, x)

```
if x.prev ≠ NIL                 // non è il primo
    x.prev.next = x.next
else                            // è il primo
    L.head = x.next
if x.next ≠ NIL                 // non è l'ultimo
    x.next.prev = x.prev
```

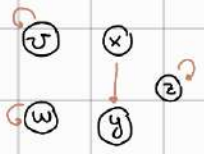
Algoritmi e Strutture Dati

Grafo

Definizione: (V, E) è un insieme finito V di elementi: nodi $E \subseteq V \times V$

E.g.: $V = \{v, w, x, y, z\}$

$E = \{(v,v), (w,w), (x,y), (z,z)\}$



Grafo non orientato: $\forall i, j$ si ha che:

1. $(i,j) \in E \rightarrow (j,i) \in E$
2. $(i,i) \notin E$

Cammino: v, v' di lunghezza K è $p = \langle v_0, v_1, \dots, v_n \rangle$ con $v_0 = v$ e $v_n = v'$ tc $\forall i \in \{0, \dots, K-1\}$ $(v_i, v_{i+1}) \in E$ con K = numero di archi

v' è **raggiungibile** da v se $\exists$ cammino da v a v': $v \overset{p}{\rightsquigarrow} v'$

Ciclo in grafo orientato: $\exists\ v \overset{p}{\rightsquigarrow} v'$ tc $v = v'$

Ciclo in grafo non orientato: $\exists\ v \overset{p}{\rightsquigarrow} v'$, $p = \langle v_0, \dots v_n \rangle$ con $K \geq 3$ e $v = v'$

Grafo fortemente connesso: tutti i nodi raggiungono tutti i nodi

Algoritmi e Strutture Dati

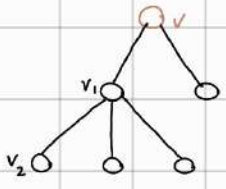
Albero

Definizione: É un grafo non orientato, connesso ed aciclico.

Sia $G(V, E)$, G é un albero se vale 1 di queste definizioni:

- G é un albero
- G connesso e $\forall e \in E$ $(V, E \setminus \{e\})$ non é connesso
- G connesso e $|e| = |v| - 1$
- G aciclico e $|e| = |v| - 1$
- G é aciclico e $\forall (u,v) \in V$ con $(u,v) \notin E$ si ha che $(V, E \cup \{(u,v)\})$ contiene un ciclo.

Albero radicato: (T, m) con T albero e $m \in V$ vertice



antenato i nodo v_1: - nodo v_2 presente cammino radice $\rightsquigarrow v_1$.

proprio se $v_1 \neq v_2$

- **discendente**: v_1 di v_2 se v_2 antenato di v_1
- **padre**: v_2 di v_1 se (v_2, v_1) ultimo arco nel cammino radice $\rightsquigarrow v_1$.
- **foglia**: nodi senza figli
- **interno**: nodo $\neq$ foglia
- **profondità** di un vertice s: lunghezza cammino radice $\rightsquigarrow s$
- **altezza** di un vertice s: n° archi del più lungo cammino da s ad una foglia.
- **albero K-ario**: ogni nodo ha al più K figli

Algoritmi e Strutture Dati

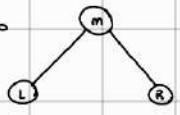
Albero Binario

Definizione: ogni nodo ha al più 2 figli

- pieno: ogni nodo ≠ foglie ha 2 figli
- completo: tutte le foglie hanno stessa profondità + è pieno.

Def. ricorsiva albero binario etichettato: detto $\mathcal{T}$ l'insieme di alberi su D:

- Albero vuoto è albero (-11)
- se $L, R \in \mathcal{T}$ e $m \in D \longrightarrow (L, m, R)$ è un albero

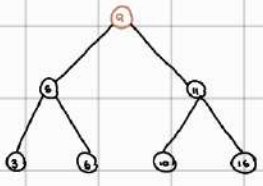


- nient'altro è Albero

Albero binario di ricerca: $B \in \mathcal{T}$ è un ABR sse:

- $\forall v$ di B
- $\forall$ chiave $b \in D$ presente nei nodi del sottoalbero sx di B con radice v
- $\forall$ chiave $c \in D$ presente nei nodi del sottoalbero dx di B con radice v

si ha che $b \leq a \leq c$, dove a è la chiave di v



Visite di un albero binario:

- Simmetrica (INORDER): sx > r > dx
- Anticipata (PREORDER): r > sx > dx
- Posticipata (POST ORDER): sx > dx > r

Algoritmi e Strutture Dati

Implementazioni Albero Binario

```
BTREE-SEARCH(K, x)
    if x = NIL or K = x.Key
        return x
    if K < x.Key
        return BTREE-SEARCH(x.left, K)
    else
        return BTREE-SEARCH(x.right, K)
```

```
BTREE-MIN(x)
    if x.left = NIL
        return x
    else
        return B-TREE-MIN(x.left)
```

```
BTREE-MAX(x)
    if x.right = NIL
        return x
    else
        return B-TREE-MAX(x.right)
```

```
BTREE-SUCC(x)
    if x.right = NIL
        return BTREE-MIN(x.right)
    else
        y = x.prev
        while y ≠ NIL AND x = y.right
            x = y
            y = y.prev
        return y
```

```
BTREE-INSERT(x, z)
    if x = NIL
        x = z
    else if z.key < x.key
        BTREE-INSERT(x.left, z)
    else
        BTREE-INSERT(x.right, z)
```

Algoritmi e Strutture Dati

Esercizi ricorsivi su albero binario

```
CONTADISPARI(x)
    if x = nil
        return 0
    else
        return CONTADISPARI(x.left) + CONTADISPARI(x.right) + (x.key mod 2)
```

```
CONTANODI(x)
    if x = nil
        return 0
    else
        return CONTANODI(x.left) + CONTANODI(x.right) + 1
```

```
CONTAFOGLIE(x)
    if x = NIL
        return 0
    if x ≠ nil
        if x.left = nil AND x.right = nil
            return 1
        else
            return CONTAFOGLIE(x.left) + CONTAFOGLIE(x.right)
```

Algoritmi e Strutture Dati

Esercizi ricorsivi su albero binario

```
VALORI_OLTRE_S(x)   // controlla se tutti i nodi hanno val > s
    if x = NIL
        return true
    else
        if x.key ≤ s
            return false
        else
            return VALORI_OLTRE_S(x.left) ∧ VALORI_OLTRE_S(x.right)
```

```
MINIMO(x)   x ≠ NIL
    m1 = x.key
    if x.left ≠ nil
        m2 = MINIMO(x.left)
        if m2 < m1
            m1 = m2
    if x.right ≠ nil
        m2 = MINIMO(x.right)
    return m1
```

```
MINIMO(x)   x ≠ NIL
    if m2 < m1
        m1 = m2
```

```
ALBERI_CONGRUENTI(x, y)
    if x = NIL ∧ y = NIL
        return true
    if x = NIL ∧ y ≠ NIL
        return false
    if x ≠ NIL ∧ y = NIL
        return false
    else
        if x.key ≠ y.key
            return false
        else
            return ALBERI_CONGRUENTI(x.left, y.left) ∧ ALBERI_CONGRUENTI(x.right, y.right)
```

```
ALBERI_SPECULARI(x, y)
    if x = NIL ∧ y = NIL
        return true
    if x = NIL ∧ y ≠ NIL
        return false
    if x ≠ NIL ∧ y = NIL
        return false
    else
        if x.key ≠ y.key
            return false
        else
            return ALB_SPECULARI(x.left, y.right) ∧ SPECULARI(x.right, y.left)
```

Algoritmi e Strutture Dati

Esercizi ricorsivi su abr

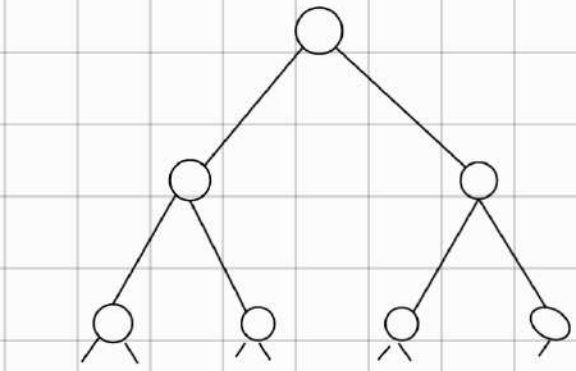
```
TROVAELEMENTO(x, K)
    if x=NIL
        return false
    else
        if x.key > K
            return TROVAELEMENTO(x.left,K)
        else
            return TROVAELEMENTO(x.right,K)
```

```
VALORIOLTRE5(x)  // controlla se tutti i nodi hanno val > s
    if x=NIL
        return true
    else
        if x.key ≤ s
            return false
        else
            return VALORIOLTRE5(x.left)  // quelli a dx saranno sicuri > s
```

Algoritmi e Strutture Dati

Esercizi ricorsivi su livelli di alberi

Graficamente:



sappiamo sicuri >=

NB: liv $\pm$ di x = liv 0 di x.left/right

liv l di x è l-$\pm$ di x.left/right

STAMPALIVELLO(x, l)

```
if x ≠ NIL
    if l = 0
        print (x.Key)
    else
        STAMPALIVELLO(x.left, l-1)
        STAMPALIVELLO(x.right, l-1)
```

SOMMALIVELLO(x, l)

```
if x = NIL
    return 0
else
    if l = 0
        return x.Key
    else
        return SOMMALIVELLO(x.left, l-1) + SOMMALIVELLO(x.right, l-1)
```

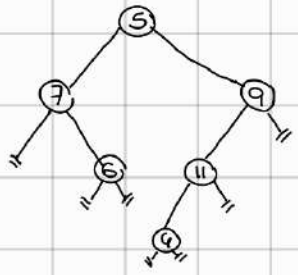
NODIPARI(x, l)

```
if x = NIL
    return true
else
    if l = 0
        if x.Key mod 2 = 0
            return true
        else return false
    else return NODIPARI(x.l, l-1) ∧ NODIPARI(x.r, l-1)
```

Algoritmi e Strutture Dati

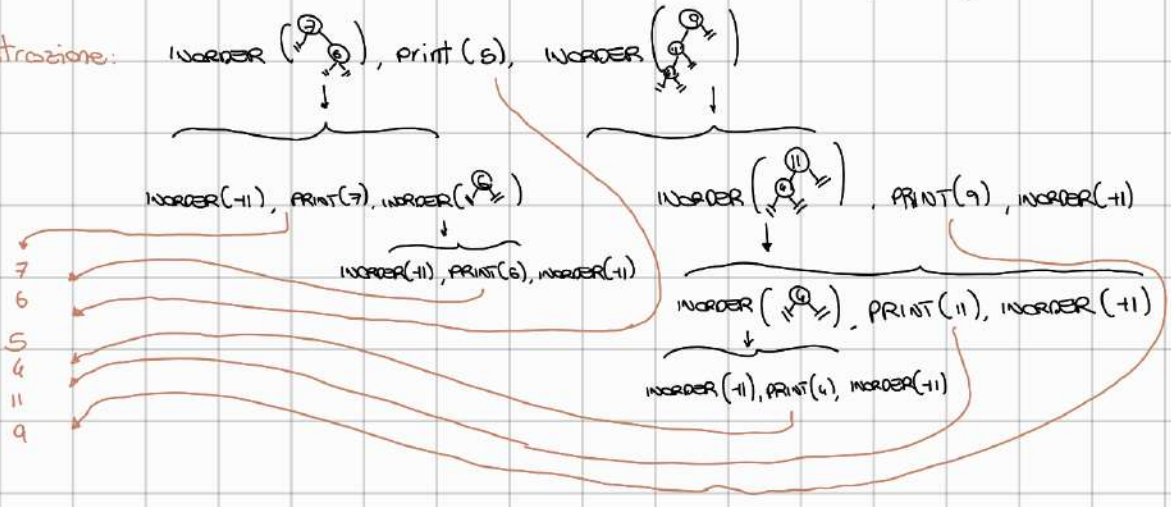
Visita Inorder

Visita inorder :




```
INORDER-VISIT(x)
if x ≠ NIL
    INORDER_VISIT (x.left)
    PRINT (x.Key)
    INORDER_VISIT (x.right)
```

Dimostrazione:

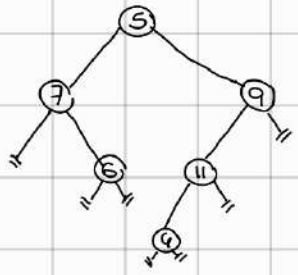


7 6 5 4 11 9

Algoritmi e Strutture Dati

Visita Anticipata

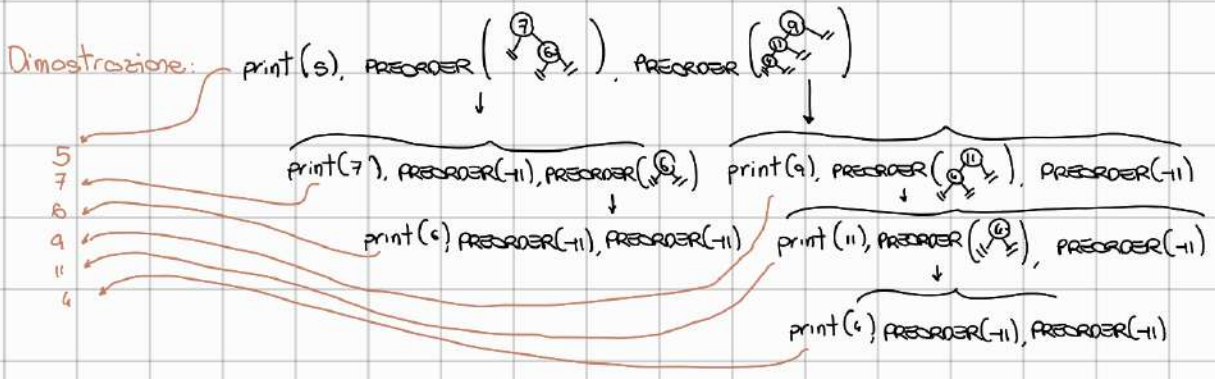
Visita preorder :




```
PRE-ORDER-VISIT(x)

if x ≠ NIL
    PRINT (x.Key)
    PREORDER_VISIT (x.left)
    PREORDER_VISIT (x.right)
```

Dimostrazione:

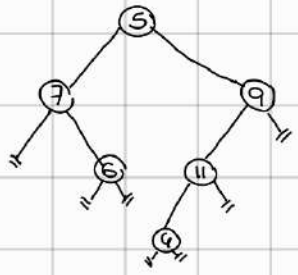


5 7 6 9 11 4

Algoritmi e Strutture Dati

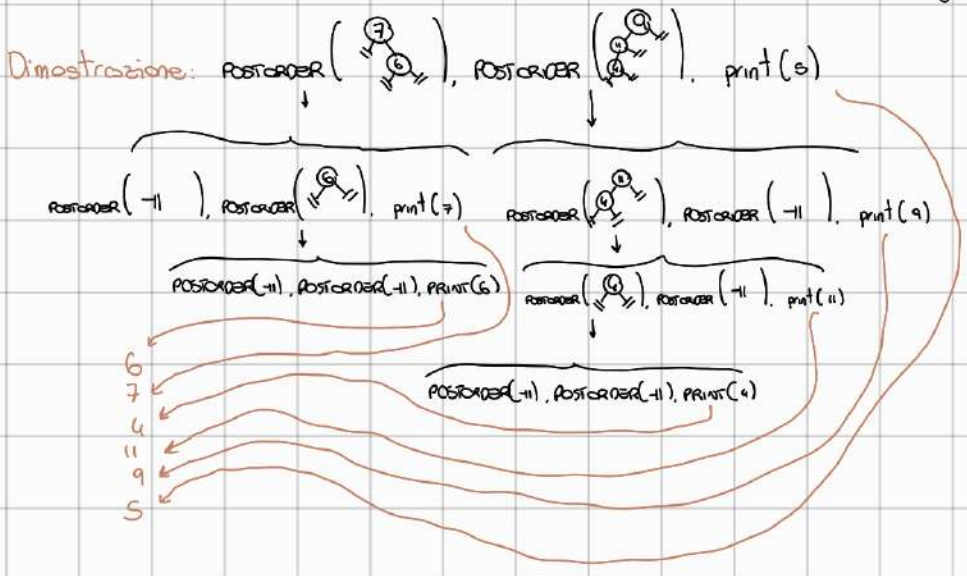
Visita posticipata

Visita post order:




```
POSTORDER-VISIT(x)
    if x ≠ NIL
        POSTORDER_VISIT (x.left)
        POSTORDER_VISIT (x.right)
        PRINT (x.Key)
```

Dimostrazione:



6 7 4 11 9 5

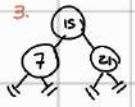
Algoritmi e Strutture Dati

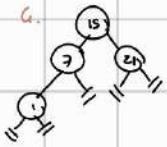
Inserimento e rimozioni elementi: Albero

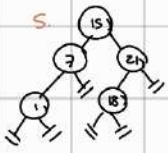
Costruzione: Valori: 15 21 7 1 18 31 23 36

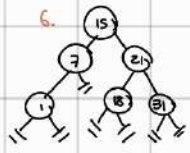
1. 

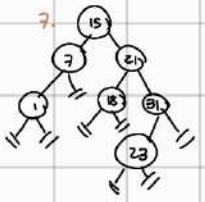
2. 

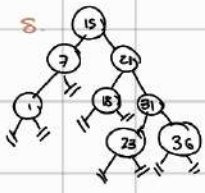
3. 

4. 

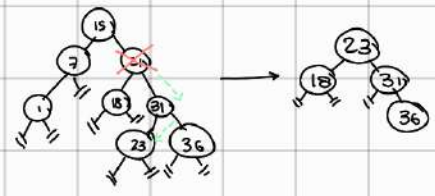
5. 

6. 

7. 

8. 

Rimozione elementi: tolgo 21



cerca il 5° successore

Algoritmi e Strutture Dati

Heap

Definizione: Struttura dati composta da un array A su un dominio D a cui corrisponde un albero binario, etichettato e quasi completo (solo l'ultimo livello non è completo, ma ha elementi a scorrimento da sx)

Ad ogni nodo è associata un indice tc:

- i=1 radice

se i è un nodo allora

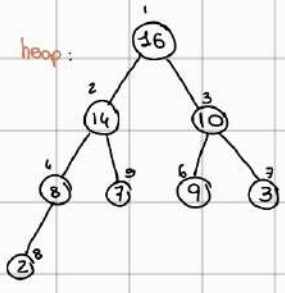
- $\lfloor i/2 \rfloor$ è il padre — parent(i)
- $2i$ è figlio sx — left(i)
- $2i+1$ è figlio dx — right(i)

- Il nodo di indice i contiene A[i]
- A.heap-size indica il nº di elementi memorizzati in A ⟶ $0 \leq$ A.heap_size $\leq$ A.length

Graficamente:

1	2	3	4	5	6	7	8
16	14	10	8	7	9	3	2

heap:



Proprietà: $\forall i \in \{2, \dots, A.heap.size\}$

- $A[parent(i)] \geq A[i]$ abbiamo un max_heap (ci concentreremo su questa)
- $A[parent(i)] \leq A[i]$ abbiamo un min_heap

Oss: un array decrescente è sempre un heap. Non vale il contrario (guarda es. esame)

Verifica Heap: Se abbiamo A array generico e vogliamo ottenere uno heap, dobbiamo utilizzare due algoritmi.

- Heapify
- Build-Heap

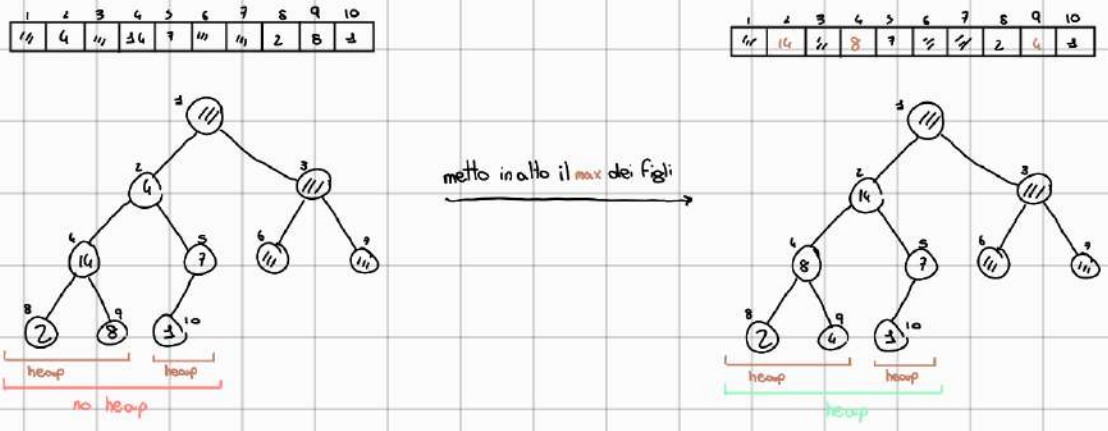
Algoritmi e Strutture Dati

Heapify e Build-Heap

Def heap: Data una struttura dove i due sottoalberi sono heap ma con la radice non lo è, lo trasforma in un heap. (1° step)

- Input: A tc left(i) e right(i) sono radici di heap ma l'albero con i non è heap
- Output: A heap
- Algoritmo: far scendere A[i] nella radice dei sottoalberi con valore maggiore

- Esempio:



- Algoritmo

```
HEAPIFY(A,i)
    l = left(i), r = right(i)
    if l ≤ A.heapsize AND A[l] > A[i]
        max = l
    else
        max = i
    if r ≤ A.heapsize AND A[r] > A[i]
        max = r
    else
        max = i
    if max ≠ i
        SCAMBIA(A[i], A[max])    // max contiene l o r
        HEAPIFY(A, max)          // ripeto nel sottoalbero modificato
```

Complessità:

- $O(h)$ se i è l'indice di un nodo di altezza h
- $O(\log_2 n)$ se n è il n° nodi nel sottoalbero con radice i

Algoritmi e Strutture Dati

Heapify e Build-Heap

Def build-heap: Utilizziamo heapify dal basso verso l'alto per convertire un Array $A[1,\dots,n]$ in uno HEAP.

Questo grazie al fatto che nel segmento $A[\lfloor n/2 \rfloor + 1, \dots, n]$ sono tutte foglie → sono heap

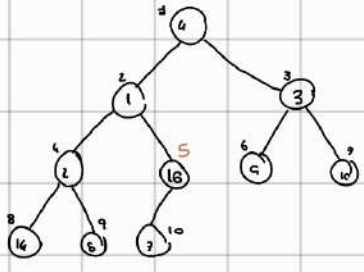
Algoritmo: BUILD-HEAP (A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ down to 1
    HEAPIFY(A, i)
```

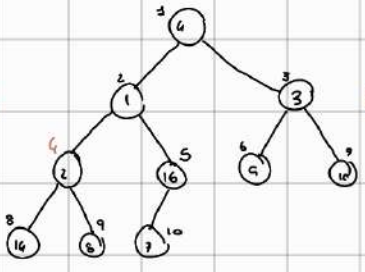
Step. Heapify + Build heap

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7

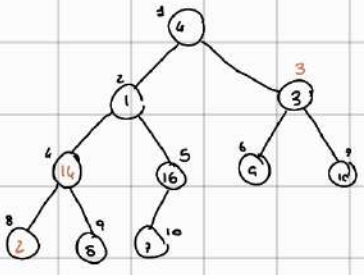
$i = \lfloor 10/2 \rfloor = 5$ → $A[6,\dots,10]$ foglie



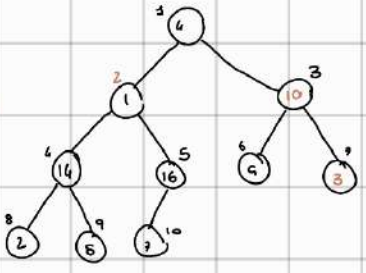
A.heapify(5) →



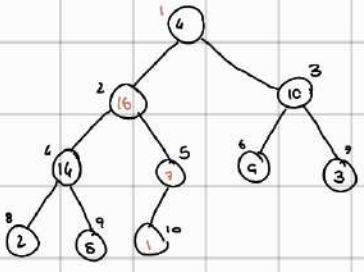
A.heapify(4) →



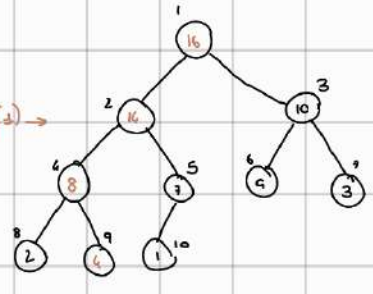
A.heapify(3) →



A.heap(2) →



A.heapify(1) →



Tempi di calcolo: $O(n \log n) \to O(n)$

Algoritmi e Strutture Dati

Heap-Sort

Introduzione: Modificando un array in un array che genera heap (heapify) posso ordinare un array, in quanto la chiave di val. max è sempre nella radice, ossia in A[1].

Metodo Risolutivo: Per ordinare un Array A di lunghezza n:

1. BUILD-HEAP(A) ho in A[1] max
2. SCAMBIO A[1] ↔ A[n] perché mi serve il max nell'ultima pos

   Questo comporta che A[1,...,n-1] non è detto sia heap ora. Mentre i due sottoalberi sono heap perché non sono stati modificati
3. A.heapsize -- (tolgo ultimo elemento = max)
4. Heapify(A,1) ora in A[1] c'è il max di A[1,...,n-1]
5. Ripeto dal punto 2 fino allo scambio A[1] ↔ A[2]

Algoritmo: HEAP-SORT(A)

```
BUILD-HEAP(A)
for i = A.lenght down to 2
    SCAMBIA(A[1], A[i])
    A.heap.size --
    HEAPIFY(A, 1)
```

Tempi di calcolo: $O(n \log n)$: n-1 chiamate di HEAPIFY ciascuna di costo $O(\log n)$

Proprietà: È in loco, non è stabile.

Nel caso medio è meglio quicksort

Algoritmi e Strutture Dati

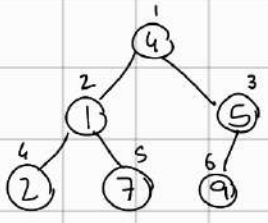
Heap-Sort simple

Simulazione: A = 4, 1, 5, 2, 7, 9

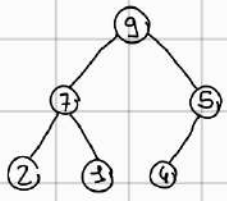
Build Heap

		i=3	i=2	i=1
1	4	4	~~4~~ 9	9	9
2	1	~~1~~ 7	7	7	7
3	~~5~~ 9	9	~~9~~ ~~4~~ 5	5	5
4	2	2	2	2	2
5	7	~~7~~ 1	1	1	1
heap.size 6	~~9~~ 5	5	~~5~~ 4	4	4

Albero di partenza



Albero finale



Algoritmo

$i = \lfloor size/2 \rfloor = 3$

l r max

Heapify(A,3)	6(9)	7(oob)	6(9)
Heapify(A,6)	12	13	6

l r max

i = 2

Heapify(A,2)	4	5	5
Heapify(A,5)	10	11	5

l r max

i = 1

Heapify(A,1)	2	3	3
Heapify(A,3)	6	7	6
Heapify(A,6)	12	13	6

Algoritmi e Strutture Dati

Heap Sort simpl 2

Simulazione: A = 4, 1, 5, 2, 7, 9

Build Heap

A =

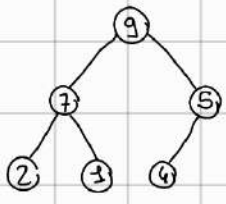
		i=3	i=2	i=1
1	4	4	4 9	9
2	1	~~1~~ 7	7	7
3	~~5~~ 9	9	9 ~~4~~ 5	5
4	2	2	2	2
5	7	~~7~~ 1	1	1
6	9 5	5	~~5~~ 4	4

A =

	i=6	i=5	i=4	i=3		
1	9	~~4~~ 7	~~7~~ 5	~~2~~ 4	~~5~~ 2	1	1
2	7	~~7~~ 4	4	~~4~~ 2	~~2~~ 1	2	
3	5	5	~~5~~ 1	1	4		
4	2	2	2	5			
5	1	1	7				
6	4	size = 9					

heap.size

Albero finale

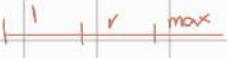


Heap Sort:

i=6
- Heapify (A, 1) 2 3 2
- Heapify (A, 2) 4 5 2

i=5
- Heapify (A, 1) 2 3 3
- Heapify (A, 3) 6 7 3

i=4
- Heapify (A, 1) 2 3 2
- Heapify (A, 2) 4 5 2



i=3
- Heapify (A, 3) 2 3 2
- Heapify (A, 2) 4 5 2

i=2
- Heapify (A, 1) 2 3 1

Algoritmi e Strutture Dati

Applicazione di Heap: code con priorità

Definizione: Struttura dati che

- Memorizza un insieme dinamico su un dominio D
- Consente di accedere al max in tempo costante $\Theta(1)$ (il max è $A[1]$)
- Rende disponibili:
  - Estrazione del max con rimozione con tempo $\Theta(\log n)$ ($A[1] \leftrightarrow A[A.heap.size]$, A.heap.size--, HEAPIFY(A,1))
  - Inserimento di un elemento k: A.heap.size++, $A[A.heapsize]=k$, "faccio salire" k fino alla pos. giusta tramite Heapify. Tempi: $\Theta(\log n)$

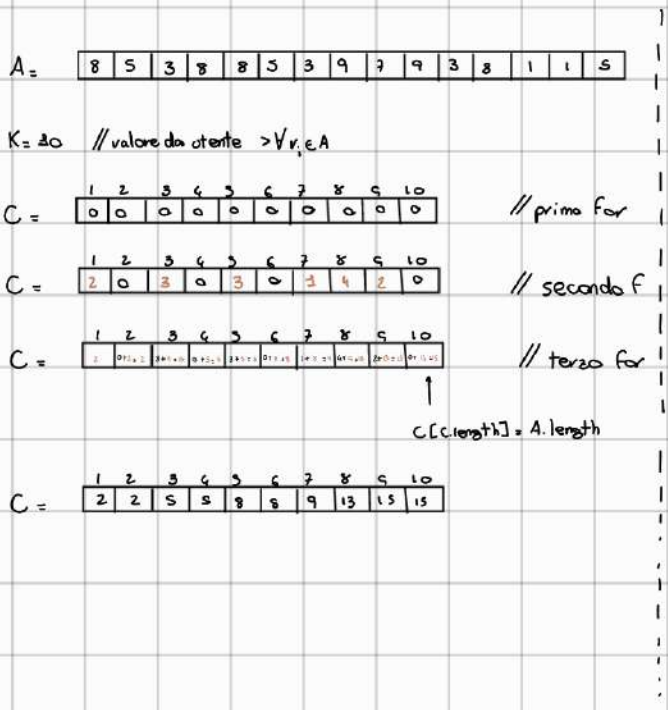
Algoritmi e Strutture Dati

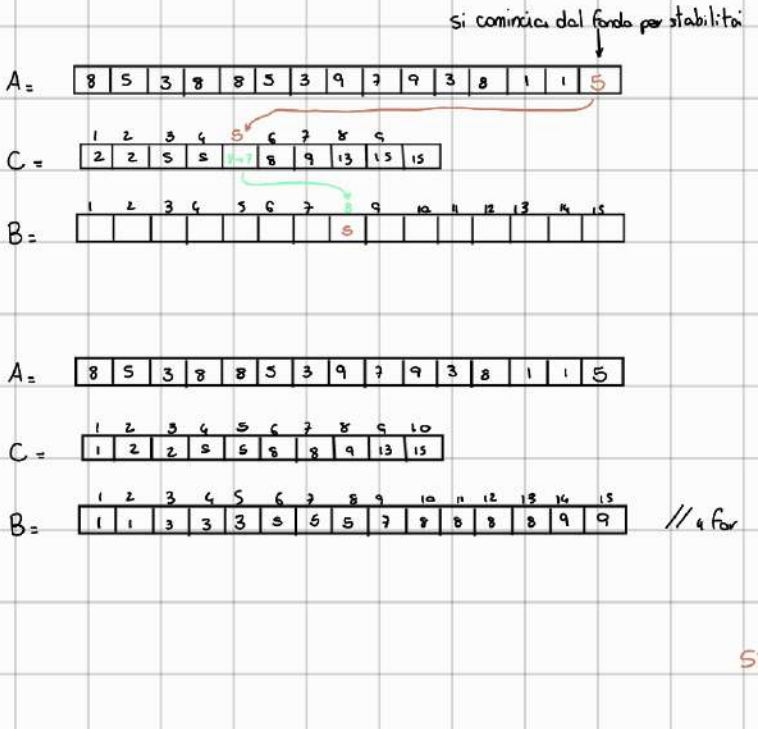
Counting Sort

COUNTING-SORT : Non è un algoritmo che utilizza i confronti !

```
COUNTING SORT (A, K)                    // K = max
    for i = 1 to C.length               // creo array di appoggio
        C[i] = 0
    for j = 1 to A.length               // conto in C[i] quante volte i è presente in A
        C[a[j]] = C[A[j]] + 1
    for i = 2 to C.length               // somme cumulate
        C[i] = C[i-1] + C[i]
    for j = A.length down to 1
        B[c[A[i]]] = A[j]
        C[A[j]] = C[A[j]] - 1
    return B
```

Dimostrazione :





Algoritmi e Strutture Dati

Radix Sort

RADIX-SORT: Utilizzato in ordinamenti con più campi chiave, ad esempio date, codice fiscale.

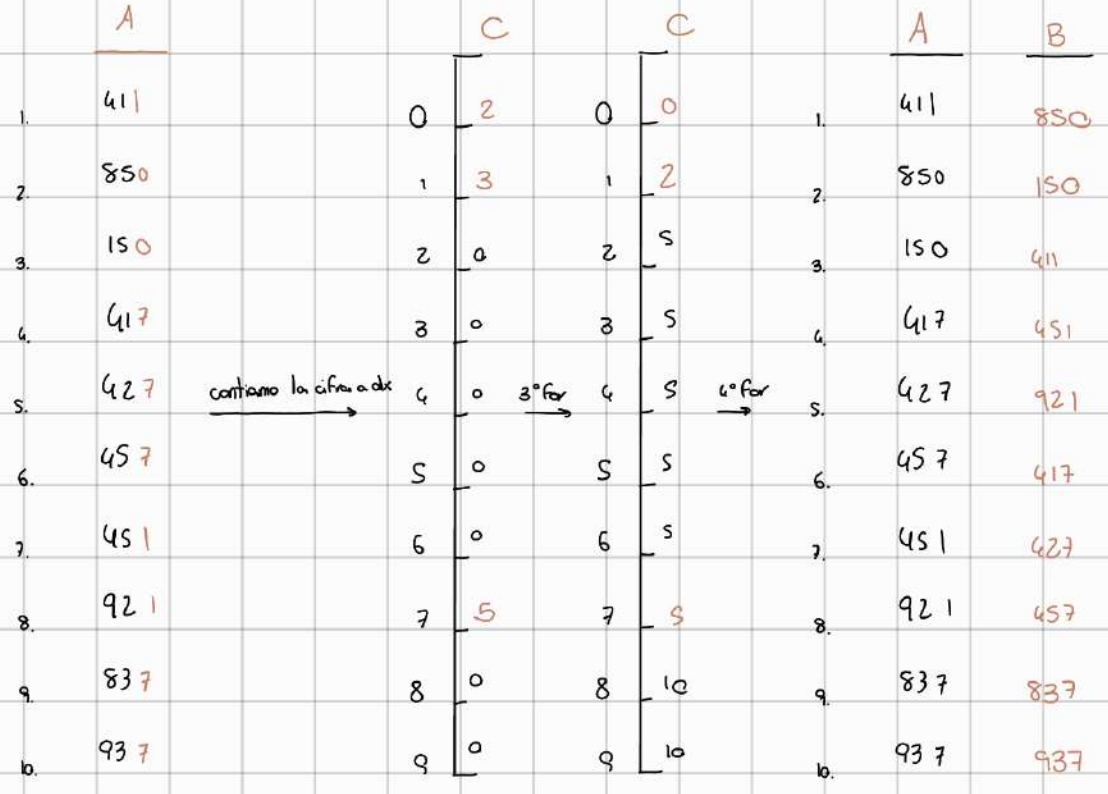
Supponendo un array A di n elementi con d cifre, dove d è quella di ordine + alto, abbiamo

Algoritmo: RADIX-SORT(A,d)

for i=1 to d

ordinamento stabile per ordinare l'Array A sulla cifra i

Dimostrazione: (sfrutto Counting Sort)



con lunghezza 10 (n° cifre)

ripetere per decine e centinaia.