

Emiddio Ingenito - Algoritmi e strutture dati

Algoritmo e programma

Visione matematica

Sintesi e analisi di algoritmi:

Esempio: algoritmo iterativo per la moltiplicazione

Esempio: Moltiplicazione "alla russa"

Confronto complessità algoritmica: moltiplicazione iterativa - moltiplicazione "alla russa"

Algoritmo "Potenza"

Algoritmo "Potenza" ricorsiva

Record di Attivazione della Chiamata Ricorsiva

Algoritmo "potenza ricorsiva" V2

Confronto tra algoritmi

Notazioni asintotiche:

Proprietà della notazione asintotica

Operazioni elementari

Macchina ad accesso diretto (RAM)

Complessità di algoritmi

Criterio di costo uniforme

Algoritmo "potenza riflessiva" (costo uniforme)

Criterio di costo logaritmico

Algoritmo "potenza riflessiva" (costo logaritmico)

Algoritmi praticabili

Esprire la complessità di problemi

Array

Ricerca sequenziale

Ricerca binaria (dicotomica)

Complessità temporale:

Complessità spaziale

Ricerca dicotomica iterativa

Costi operazioni su array ordinato

Inserimento

Ricerca

Eliminazione

Algoritmo di ordinamento

Stabilità

Complessità degli algoritmi di ordinamento

Algoritmi di ordinamento interno

Elementari

Selection sort

Insertion sort

Bubble sort

Avanzati

Merge sort

Algoritmo di "fusione" di due array:

Implementazione Merge sort, con tecnica "divide-et-impera"

Merge Sort Ottimizzato

QuickSort

Formule utili

Tecnica "divide-et-impera"

Tipo di variabile

Dizionario

Struttura dati

Collezioni:

Strutture indicizzate (Array)

Strutture collegate

Linked lists

Ricerca elemento in base alla posizione (indice)

Ricerca elemento in base alla chiave

Ricerca elemento in base alla chiave, in una Lista Ordinata

Inserimento in lista ordinata

[Pila \(Stack\)](#)

[Implementazione mediante liste concatenate](#)

[Metodi](#)

[Coda](#)

[Implementazione mediante liste concatenate](#)

[Metodi](#)

[Alberi Binari](#)

[Visita in ampiezza \(implementata utilizzando Coda C\)](#)

[DFS](#)

[DFS Pre-order](#)

[DFS In-order](#)

[DFS Post-order](#)

[Alberi generici](#)

[Rappresentazione tramite "vettore dei padri"](#)

[Rappresentazione tramite vettore dei figli](#)

[Rappresentazioni collegate](#)

[Relazione tra numero di nodi e altezza in un albero binario](#)

[Alberi binari completi](#)

[Alberi binari "quasi completi"](#)

[Heap](#)

[Rappresentazione di un heap](#)

[Risistemare uno heap](#)

[Costruzione di uno heap, soluzione iterativa](#)

[Costruzione di uno heap, soluzione ricorsiva](#)

[Da Array ad albero binario quasi completo, a Max Heap.](#)

[Heap Sort](#)

[Costruzione di un heap, in loco](#)

[Operazioni su heap](#)

[Trovare l'elemento di chiave massima/minima](#)

[Cancellare l'elemento di chiave massima](#)

[Inserimento di un nuovo elemento](#)

[Cancellazione di un elemento](#)

[Modificare la chiave di un elemento](#)

[Code con priorità](#)

[Algoritmi di ordinamento senza confronti tra chiavi](#)

[IntegerSort \(Counting Sort\)](#)

[BucketSort](#)

[RadixSort](#)

[Insiemistica](#)

[Union Find](#)

[Implementazione QuickFind](#)

[Operazioni:](#)

[QuickUnion](#)

[Bilanciamento in altezza \(Union by RANK\)](#)

[Compressione di cammino](#)

[Grafi](#)

[Archi](#)

[Grado di un nodo.](#)

[Cammino](#)

[Ciclo](#)

[Catena](#)

[Differenza tra cammino e catena:](#)

[Circuito](#)

[Circuito Hamiltoniano](#)

[Circuito euleriano](#)

[Grafo connesso](#)

[Grafo fortemente connesso](#)

[Sottografo](#)

[Cricca](#)

[Alberi](#)

[Rappresentazione di grafi](#)

[Lista di archi](#)

[Lista di adiacenza](#)
[Lista di incidenza](#)
[Matrice di adiacenza](#)
[Matrice di incidenza](#)

Visite sui grafi

[BFS](#)
[DFS](#)

Grafi pesati

[Problemi di ottimizzazione](#)
[Tecnica risolutiva greedy](#)

Spanning tree

[Minimum spanning tree](#)
[Algoritmo di Kruskal](#)
[Implementazione algoritmo di Kruskal \(UNION-FIND / MFSET\)](#)

[Algoritmo di Prim](#)

[Implementazione algoritmo di Prim:](#)

Risoluzione di problemi con strategia bottom-up.

Programmazione dinamica

[Sottovettore di somma massima](#)
[Cammini di valore minimo su matrici](#)

Cammini minimi

[Rappresentazione di grafi pesati:](#)

[Lista di adiacenza con pesi](#)
[Matrice dei pesi](#)

[Cammini minimi tra ogni coppia di vertici \(Algoritmo di Floyd e Warshall\)](#)

Cammino minimo tra un vertice e un altro

Cammini minimi tra un vertice e tutti gli altri

[Algoritmo di Bellman & Ford:](#)
[Algoritmo di Dijkstra](#)

Alberi binari di ricerca (ABR)

[Trovare il nodo con chiave max/min](#)
[Ricerca dicotomica ricorsiva](#)
[Inserimento ricorsivo](#)
[Cancellazione di un nodo](#)

[Esercizio: Cancellazione nodi](#)
[Complessità temporale](#)
[Relazione tra altezza e numero di nodi:](#)

Alberi perfettamente bilanciati

Alberi bilanciati in altezza (AVL)

[Inserimento e bilanciamento](#)
[Sbilanciamento dell'albero a sinistra, nel sottoalbero sinistro](#)
[Sbilanciamento dell'albero a destra, nel sottoalbero sinistro](#)
[Riepilogo costo operazioni](#)

Alberi 2-3

[Alberi 2-3 di ricerca](#)
[Ricerca di un nodo:](#)
[Inserimento](#)
[Cancellazione](#)
[Riepilogo costo operazioni](#)

Memorizzazione dei dati su memoria secondaria

B-Alberi

[Ricerca di un nodo](#)
[Inserimento di un nodo](#)
[Cancellazione](#)

[Accessi a memoria secondaria](#)

Dizionari

Tabelle hash

[Fattore di carico:](#)
[Tabella hash ad accesso diretto](#)
[Funzione hash perfetta](#)
[Collisioni](#)
[Funzioni hash](#)

[Gestione delle collisioni](#)
[Gestione esterna \(utilizza strutture dati esterne\): Liste di collisione](#)
[Gestione interna: Indirizzamento aperto](#)
[Inserimento di un elemento \(Hash Table\)](#)
[Ricerca \(Hash Table\)](#)
[Re-hashing](#)
[Risorse necessarie e sufficienti](#)
[Limitazione superiore \(Upper bound\)](#)
[Limitazione inferiore \(Lower bound\)](#)
[Complessità computazionale](#)
[Classi di complessità](#)
[Classe P](#)
[Tipologie di problemi](#)
[Classe TIME e SPACE](#)
[Relazione tempo-spazio](#)
[Relazione spazio-tempo](#)

Algoritmo e programma

Algoritmo - Sequenza ordinata e finita di passi eseguibili, non ambigui, che definiscono un procedimento che termina.

Caratteristiche Fondamentali di un Algoritmo:

- **Non ambiguità:** Le istruzioni che compongono un algoritmo devono essere chiare, univoche e comprensibili per chi deve eseguirlo. Non devono esserci dubbi o interpretazioni multiple che potrebbero portare a risultati errati.
- **Eseguibilità:** Ogni singola istruzione contenuta nell'algoritmo deve essere realizzabile e concretamente eseguibile dall'esecutore, che sia una persona, un computer o un altro tipo di sistema.
- **Finitezza:** L'esecuzione dell'algoritmo deve terminare in un tempo finito, producendo un risultato o arrivando a un punto di stallo definito. Non deve essere un processo infinito o che si blocca senza fornire una conclusione.
- **Correttezza:** L'algoritmo deve funzionare correttamente per qualsiasi insieme di dati di input ammessi, producendo il risultato desiderato o previsto.

Può essere descritto secondo linguaggio naturale (spiegato a parole) o pseudocodice.

Programma - Implementazione di un algoritmo, in un linguaggio comprensibile da parte del calcolatore (Linguaggio di programmazione).

Possiamo dire che mentre un algoritmo è un'astrazione di un procedimento (a parole, o pseudocodice), un programma ne è la sua concreta implementazione in un linguaggio di programmazione.

Visione matematica

Dal punto di vista matematico, un algoritmo è una funzione da un dominio "degli input", ad un codominio delle "soluzioni"

Sintesi e analisi di algoritmi:

Sintesi di un algoritmo - Progettazione di un algoritmo che risolva uno specifico problema.

Analisi di un algoritmo - Ne si valuta:

- Correttezza - Dato un algoritmo a che risolve un problema p , dimostrare che effettivamente a risolve p .
- Efficienza - Valutazione della quantità di risorse (tempo, spazio...) utilizzate.

Questa valutazione può essere effettuata "a posteriori" (da un algoritmo, ne implemento il programma, ed effettuo test sul programma in esecuzione), oppure "a priori" (secondo metodi matematici, durante la fase di progettazione, evitando

l'implementazione, che può essere dispendiosa di tempo e denaro, oltre a fornire una valutazione più precisa, in quanto gli input non possono essere testati tutti, potrebbero essere infiniti)

Esempio: algoritmo iterativo per la moltiplicazione

Dati $a, b \geq 0$

```
moltiplicazione(intero a, intero b) -> intero
1 prod <- 0
2 while b>0 DO
3     prod <- prod + a
4     b <- b-1
5 return prod
```

Notazione Pseudocodice:

- Prima riga, viene indicato il nome dell'algoritmo, i parametri in input tra parentesi, tipo di valore restituito in output
- La freccia \leftarrow indica un'operazione di memorizzazione

Per effettuare l'analisi dell'algoritmo, non possiamo considerare il tempo d'esecuzione, perché varia in base alle prestazioni della macchina.

Approssimiamo il tempo, in base al numero di istruzioni eseguite, e al variare di esse, al variare dei parametri in input.

- **Caso $b=0$** , vengono eseguite le righe: 1,2,5.

Complessità d'esecuzione = 3.

- **Caso $b>0$** , vengono eseguite:

Righe 1,5 per una volta

Righe 3,4 per b volte

Riga 2 per $b+1$ volte

Per ogni blocco, **tempo = nRighe * nVolte**

$$\text{Tempo totale} = (2*1) + (2*b) + (1*(b+1)) = 2 + 2b + b + 1 = 3b + 3$$

Complessità spaziale: Oltre allo spazio necessario per la memorizzazione dei parametri in ingresso, è necessaria una variabile "prod" aggiuntiva. Spazio costante.

Esempio: Moltiplicazione "alla russa"



Andando verso il basso, nella colonna dei moltiplicandi, moltiplico per 2, nella colonna dei moltiplicatori, divido per 2.

L'Algoritmo termina quando ottengo moltiplicatore pari a 1.

Il risultato della moltiplicazione, si ottiene sommando i moltiplicandi che appaiono in corrispondenza dei moltiplicatori (b) dispari.

Questo algoritmo è particolarmente efficiente quando implementato all'interno del calcolatore, poiché moltiplicare e dividere per due, significa "shiftare" A e B a destra o sinistra. Molto meno dispendioso rispetto alla somma iterativa.

```
moltiplicazione(intero a, intero b) -> intero
1 prod <- 0
2 while b>0 DO
3   if b IS ODD THEN
4     prod <- prod + a
5   a <- a/2
6   b <- b/2
7 RETURN prod
```

Se considero il parametro U come il numero di iterazioni del ciclo while

Riga 1,7 vengono eseguite una volta

Riga 3,5,6 vengono eseguite U volte

Riga 4, non possiamo sapere quante volte viene eseguita, dipende da b, ma sappiamo che verrà eseguita per un numero $\leq U$ volte. Consideriamo per il calcolo della complessità computazionale, il caso peggiore in cui la riga 4 viene eseguita U volte.

Riga 2, U+1 volte.

Tempo totale = $(2*1) + (3*U) + (1*U) + (1*(U+1)) = 2 + 3U + U + U + 1 = 5U + 3$ (al più, considerando il caso peggiore).

Ora studiamo come varia U al variare di B.

b	U (Numero iterazioni while)
0	0
1	1
2	2
3	2
4	3
5	3
6	3
7	3
8	4

U, è uguale a $\log_2(b) + 1$.

Sostituendo U nell'espressione precedentemente calcolata, $T(b) \leq 5(\log_2(b) + 1) + 3 = 5 \log_2(b) + 8$

Confronto complessità algoritmica: moltiplicazione iterativa - moltiplicazione "alla russa"

Moltiplicazione iterativa:

$$T(b) = 3b + 3$$

Moltiplicazione "alla russa":

$$T(b) \leq 5(\log_2(b) + 1) + 3 = 5\log_2(b) + 8$$

Analizziamo le prestazioni dei due algoritmi, all'aumentare di **b**:

b	1	2	8	1000	1000000
Moltiplicazione Iterativa	6	9	27	3003	3000003
Moltiplicazione "alla russa"	8	13	23	53	103

La moltiplicazione "**alla russa**", risulta estremamente più efficiente.

Algoritmo "Potenza"

Dati x, y numeri interi:

```
ALGORITMO potenza (intero x,intero y) -> intero
1   power <- 1
2   while y>0 DO
3       power <- power * x
4       y <- y-1
5   return power
```

Righe 1,5 vengono eseguite una volta.

Riga 2 viene eseguita $y+1$ volte.

Righe 3,4 vengono eseguite y volte.

Tempo di esecuzione dell'algoritmo "Potenza": $T(x, y) = (2 * 1) + (1 * y + 1) + (2 * y) = 2 + y + 1 + 2y = 3y + 3$

Dove **y** è l'esponente (secondo parametro dell'algoritmo).

La complessità di questo algoritmo è lineare rispetto all'esponente.

Algoritmo "Potenza" ricorsiva

```
ALGORITMO potenza_ricorsiva(intero x,intero y) -> intero
1   if y=0 THEN
2       return 1
3   ELSE
4       power <- potenza(x,y/2)
5       power <- power * power
6       if y è dispari THEN
7           power <- power * x
8   return power
```

- **Se $y=0$**

Riga 1,2 eseguite una volta.

Tempo totale: 2

- **Se $Y > 0$**

Riga 1,4,5,6,8, eseguite una volta

Riga 7, eseguita al più una volta. (0 o 1 volte).

La chiamata ricorsiva, riga 4, non costa 1 tuttavia, ma varrà $1 + T(x, y/2)$, questo perché la funzione viene chiamata ricorsivamente con l'esponente dimezzato.

Tempo totale: $(5*1) + 1 + T(x, y/2) \leq 6 + T(x, y/2)$



La riga in cui compare "ELSE", non conta nel conteggio della complessità computazionale, in quanto non è richiesto alcun calcolo o operazione per essere eseguita.

Quanto vale la chiamata ricorsiva? Calcoliamo il valore del "parametro" $T(x, y/2)$.

$$T(x,y) \leq 6 + T(x, y/2)$$

$$\leq 6 + (6 + T(x, y/4))$$

$$\leq 6 + 6 + 6 + T(x, y/8))$$

...

$$6*K + T(x, y/2^k)$$

Sostituisco $T(x, y/2^k)$, con qualcosa che conosco, per esempio con $T(x, 1)$ che so essere 8.

Quand'è che $y/2^k = 1$? quando $2^k = y$, cioè quando $k = \lfloor \log_2(y) \rfloor$

Sostituendo $k = \lfloor \log_2(y) \rfloor$, otteniamo: $6 * \lfloor \log_2(y) \rfloor + T(x, 1) = 6 * \lfloor \log_2(y) \rfloor + 8$.

Quindi, il tempo totale dell'algoritmo "Potenza ricorsiva" è: $T(y) = 6 * \lfloor \log_2(y) \rfloor + 8$

Questa complessità logaritmica è significativamente più efficiente rispetto alla versione iterativa, specialmente per valori elevati di y .

Record di Attivazione della Chiamata Ricorsiva

Il record di attivazione (o frame di attivazione) per la chiamata ricorsiva dell'algoritmo "Potenza ricorsiva" contiene le seguenti informazioni:

- **Parametri della funzione:** x e y
- **Variabili locali:** power
- **Indirizzo di ritorno:** L'indirizzo dell'istruzione successiva alla chiamata ricorsiva
- **Spazio per il valore di ritorno:** Dove verrà memorizzato il risultato della chiamata ricorsiva

Ad ogni chiamata ricorsiva, un nuovo record di attivazione viene creato e impilato sullo stack delle chiamate. Questo processo continua fino a quando non si raggiunge il caso base ($y = 0$).

Ecco un esempio di come lo stack delle chiamate potrebbe apparire per `potenza_ricorsiva(2, 5)`:

```
| potenza_ricorsiva(2, 0) | - Caso base
| potenza_ricorsiva(2, 1) |
```

```
| potenza_ricorsiva(2, 2) |
| potenza_ricorsiva(2, 5) | <- Chiamata iniziale
```

Dato y , possiamo calcolare il numero di record di attivazione che verranno impilati sullo stack, che è pari a $2 + \lfloor \log_2(y) \rfloor$.

Ogni record, necessita di memorizzare: quindi una dimensione variabile di bytes, pari alle variabili $x, y, power$, e una porzione variabile di memoria, per i meccanismi di ricorsione (indirizzo di ritorno, spazio per il valore di ritorno).

Algoritmo “potenza ricorsiva” V2

```
ALGORITMO potenza (intero x,intero y) -> intero
1   power <- 1
2   IF y>0 THEN
3       power <- potenza(x,y/2)
4       power <- power * power
5       IF y è dispari THEN
6           power <- power*x
7   return power
```

- **Se $Y=0$**

Riga 1,2,7 eseguite una volta.

Tempo totale: $3*1 = 3$.

- **Se $Y>0$**

Riga 1,2,3,4,5,7 eseguite una volta.

Riga 6 eseguita al più una volta

La riga 3 non può valere 1, è una chiamata ricorsiva, vale $1 + T(x,y/2)$.

Tempo totale: $(6*1) + (AI MAX 1) + T(x,y/2)$

$$\leq 7 + T(x,y/2)$$

Quanto vale la chiamata ricorsiva? Calcoliamo il valore del “parametro” $T(x, y/2)$.

$$T(x,y) \leq 7 + T(x, y/2)$$

$$\leq 7 + 7 + T(x,y/4)$$

$$\leq 7 + 7 + 7 + T(x,y/8)$$

...

$$\leq 7*K + T(x, y/2^K)$$

Provo a sostituire $T(x, y/2^K)$ con qualcosa che conosco, per esempio $T(x,1)$, che so essere 10 ($T(x,1) \leq 7 + T(x,0) \leq 7+3 \leq 10$).

Quand’è che $y/2^K$ sarà uguale a 0 = quando $2^K = y$, cioè $K = \log_2 Y$

Tempo totale $\leq 7 * \log_2 Y + T(x,0) \leq 7 * \log_2 Y + 10$.

Confronto tra algoritmi

Dati due algoritmi, vogliamo valutarne l’ordine di grandezza del loro tempo di esecuzione, e delle loro necessità in termini di memoria, per poter definire quale di questi è più efficiente.

La disciplina che si occupa della quantificazione di tali grandezze è la **complessità computazionale asintotica**:

Notazioni asintotiche:

Date due funzioni (algoritmi) $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$, introduciamo le seguenti notazioni per eseguirne un confronto.

- Limitazione superiore - O-grande

$f(n)$ è O-grande di $g(n)$, se $\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0 : f(n) \leq c \cdot g(n)$

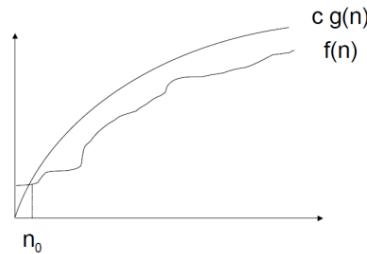
In altre parole, esiste un multiplo della funzione g , per cui la funzione f , per ogni valore superiore a n_0 , è inferiore alla funzione g .

Esempio 2.1

Sia $f(n) = 3n + 3$

$f(n)$ è un $O(n^2)$ in quanto, posto $c = 6$, $cn^2 \geq 3n + 3$ per ogni $n \geq 1$.

Ma $f(n)$ è anche un $O(n)$ in quanto $cn \geq 3n + 3$ per ogni $n \geq 1$ se $c \geq 6$, oppure per ogni $n \geq 3$ se $c \geq 4$.



In particolare, ci interesserà successivamente, determinare il più piccolo valore di c , per cui $f(n)$ sia sempre più piccola di $c * g(n)$.

Possiamo pensare alla notazione $f(n) = O(g(n))$ come ad una sostituzione del simbolo di minore, asintoticamente parlando.

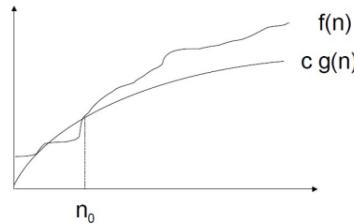
Questo perché sarebbe scorretto dire che $f(n)$ è minore di $g(n)$, notiamo infatti che nella prima fase delle curve nel disegno sopra, $f(n)$ sta sopra a $g(n)$.

Asintoticamente parlando però, cioè quando si va verso valori molto grandi, in particolare superato il valore n_0 , $f(n)$ rimane sempre sotto $g(n)$.

- Limitazione inferiore

$f(n)$ è Omega-grande di $g(n)$, se $\exists c > 0, \exists n_0 \in \mathbb{N}, \forall n > n_0 : f(n) \geq c \cdot g(n)$

Questo significa che $f(n)$ cresce almeno tanto velocemente (o più) quanto $g(n)$, moltiplicata per una costante positiva c , per tutti i valori n sufficientemente grandi (maggiori di n_0).

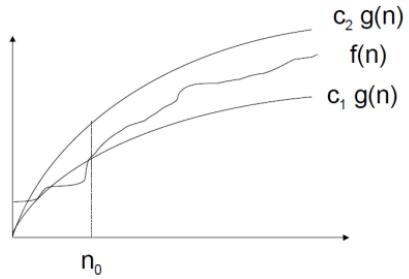


In questo caso, cercheremo la più grande $c * g(n)$ tale per cui $f(n)$ è Omega-grande di $g(n)$

- Stesso ordine di grandezza

$f(n)$ è Theta-grande di $g(n)$, se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Cioè esistono due costanti c_1 e c_2 , tali per cui posso "incapsulare" $f(n)$ tra $c_1 * g(n)$ e $c_2 * g(n)$ (Per n superiori a n_0).



Proprietà della notazione asintotica

- $f(n) = O(g(n)) \Rightarrow k \cdot f(n) = O(g(n))$ per ogni costante $k > 0$
- **Somma:** $O(f(n) + g(n)) = O(\max(f(n), g(n)))$
- **Prodotto:** $O(f(n) * g(n)) = O(f(n) * g(n))$ (Svolgo semplicemente la moltiplicazione)

Queste proprietà sono fondamentali per semplificare l'analisi della complessità degli algoritmi e per confrontare diverse funzioni di crescita.

Operazioni elementari

Quali sono le operazioni elementari? Cioè, quali possono essere eseguite con complessità $O(1)$?

Dipende dall'architettura dell'elaboratore su cui eseguo i miei algoritmi.

Macchina ad accesso diretto (RAM)

Modello di calcolo astratto, in cui l'accesso a memoria, avviene in un quanto di tempo costante, indipendentemente dalla posizione del dato nella memoria.

Pertanto, Load e Store possono essere considerate operazioni elementari.

Altre operazioni elementari:

- Operazioni aritmetico-logiche su interi ($+, -, *, /, \%, \text{AND}, \text{OR}, \text{NOT}, \text{XOR}...$)
- Operazioni di confronto su interi ($>, <, \geq, \leq$)
- Salto

Complessità di algoritmi

Dato un algoritmo A , e un'istanza I (istanza = insieme di input), studieremo la crescita del tempo impiegato dall'algoritmo A , al crescere della lunghezza dell'input (numero di bit).

Analizzando il tempo impiegato da un algoritmo, in relazione alla lunghezza dell'input, può essere interessante effettuare due valutazioni:

- **Tempo impiegato dall'algoritmo su input di lunghezza N**, nel caso peggiore, definito come $T(n) = \max\{\text{tempo}(I) : |I| = n\}$, cioè il tempo massimo che l'algoritmo impiega tra tutte le istanze di lunghezza n .
- **Tempo medio impiegato dall'algoritmo su input di lunghezza N**: $T(\text{avg}) = \sum_{|I|=n} \text{Prob}(I) * \text{tempo}(I)$, cioè per ogni istanza di lunghezza n , il tempo che l'algoritmo impiega per essere eseguito, moltiplicato per la probabilità che l'istanza appaia.

Criterio di costo uniforme

- **Tempo** - Ogni istruzione elementare viene considerata con complessità $O(1)$, indipendentemente dalla grandezza degli operandi.
- **Spazio** - Ogni variabile elementare, utilizza un'unità di spazio, indipendentemente dal valore contenuto.

Può essere applicato quando i valori trattati dall'algoritmi sono di grandezza limitata (int, float, double...)

Non posso però pensare di approssimare ad O(1) il confronto tra stringhe per esempio, poiché esso "include" under the hood, al più N confronti, dove N è la lunghezza della stringa.

Algoritmo "potenza riflessiva" (costo uniforme)

```
algoritmo xx (intero x) -> intero
1   p <- 1
2   for i<-1 TO x DO
3       p <- p*x
4   return p
```

Riga 1,4 eseguite una volta

Riga 2 viene eseguita x volte (Riga 2 include un'operazione di incremento e confronto)

Riga 3 viene eseguita x volte (Riga 3 include un'operazione di prodotto e assegnamento)

Tralasciando righe 1,4, le complessità asintotiche di righe 2 e 3 valgono Theta(x) (ogni riga vale 2^x , perché includono due operazioni ciascuna, ciò vuol dire che segue il comportamento di x)

Complessità totale (secondo il criterio del costo uniforme):

$$\Theta(x) + \Theta(x) = (\Omega(x) + O(x)) + (\Omega(x) + O(x)) = \Theta(x)$$

Criterio di costo logaritmico

- Tempo - il tempo di calcolo è proporzionale alla lunghezza degli operandi, il costo è pari al logaritmo delle rappresentazioni.
- Spazio - Si tiene conto della lunghezza della rappresentazione del dato, il cui costo è pari al logaritmo della lunghezza del dato



I Logaritmi, vanno considerati sempre in base 2.

Algoritmo "potenza riflessiva" (costo logaritmico)

```
algoritmo xx (intero x) -> intero
1   p <- 1
2   for i<-1 TO x DO
3       p <- p*x
4   return p
```

All'aumentare di X, qual'è il blocco di codice che manda in crisi il mio algoritmo? La riga 3.

La riga 2 può essere trascurata, per X=20 per esempio, il for itererà da 1 a 20, trascurabile.

Il vero problema è calcolare 20^20 .

Tempo (logaritmo delle rappresentazioni):

- All'i-esima iterazione, viene calcolato $(x^{i-1}) * x$, e viene inserito all'interno di p.

Costo del calcolo ⇒ Pari al costo della rappresentazione del dato calcolato

Dato un numero n , il numero di bit necessari a rappresentare n è $\log_2(n)$.

Tempo totale = $\log(x^{i-1}) + \log(x) = (i-1)\log(x) + \log(x) = (i\log(x)) - \log(x) + \log(x) = i\log(x)$

Costo dell'assegnamento ⇒ Pari al costo della rappresentazione del dato da memorizzare

Per memorizzare x^i , mi servono $\log(x^i)$ bit = $i\log(x)$

L' i -esima iterazione, costa allora:

Costo calcolo + costo assegnamento = $i\log(x) + i\log(x) = 2 * i * \log(x) =$

$$\Theta(i\log(x))$$

I coefficienti NUMERICI posso essere trascurati.

- Tempo totale:

$$\sum_{i=1}^x \Theta(i\log(x))$$

La sommatoria, rappresenta il for, cioè ripetuto per x iterazioni.

Algoritmi praticabili

Un algoritmo è detto "praticabile" o "ragionevole", se $T(\text{tempo}(n))$, su input di lunghezza n , è un polinomio.

Gli algoritmi che utilizzano tempo esponenziale, sono considerati "impraticabili", "inefficienti".

Esprimere la complessità di problemi

Dato un problema P , quanto tempo è necessario per risolvere P ?

- Limitazione superiore - P è risolubile in tempo

$$O(T(n))$$

Cioè esiste uno specifico algoritmo, che risolve il problema, e usa al massimo $T(n)$ tempo.

- Limitazione inferiore - $P = \Omega(T(n))$

Cioè ogni algoritmo che risolve quel problema, usa almeno $T(n)$ tempo.

Array

Collezione omogenea di elementi, accessibili in base alla posizione (indice, o offset rispetto al base address).

Memorizzato in una porzione contigua di memoria, pertanto la sua struttura è statica (se voglio aggiungere elementi, potrebbe essere necessaria una riallocazione)

Il Tempo d'accesso, è indipendente dalla posizione del dato (per il concetto di RAM)

Ricerca sequenziale

Input: Array A, elemento x

Output: Indice i t.c. $A[i] = x$, -1 se A non contiene x

```
ALGORITMO ricercaSequenziale(array A[0....N-1], elemento x) -> indice
1   i<-0
2   while i<n AND A[i]!=x DO
3     i<-i+1
4   IF i=n THEN
5     RETURN -1
6   ELSE
7     RETURN i
```

Tempo di esecuzione:

$$\Theta(N), \text{ con } N = \text{len}(A)$$

Ricerca binaria (dicotomica)

Si assume che l'array in input sia ordinato.

Caso base: se l'array ha 0 elementi, l'esecuzione termina

Passo induttivo: confronto l'elemento ricercato con l'elemento "di mezzo".

Se non l'ho trovato, scelgo se continuare la ricerca nella parte sinistra o destra.

```
FUNZIONE RicercaRicorsiva(Array A, indice sx, indice dx, elemento x) -> indice
IF dx<=sx THEN RETURN -1 //se l'array ha 0 elementi, caso base
ELSE
  indiceMedio = (sx+dx)/2 //parte intera, risolvo il problema di lunghezza pari o dx
  if A[indiceMedio] = x //caso base in caso l'array avesse un solo elemento.
    RETURN indiceMedio
  ELSE IF x<A[indiceMedio]
    RicercaRicorsiva(A, sx, m, x)
  ELSE
    RicercaRicorsiva(A, m+1, dx, x)

ALGORITMO ricercaBinaria(Array A, elemento x) -> indice
RETURN RicercaRicorsiva(A, 0, N, x)
```

Complessità temporale:

Ad ogni chiamata ricorsiva, l'intervallo di ricerca viene dimezzato: in particolare, dall'array originale di dimensione N, andrò ad operare su sottoparti di dimensioni $N/2, N/4, N/8...$

All'i-esima chiamata ricorsiva, stiamo considerando una sottoparte dell'array originale, di lunghezza $N/2^{i-1}$

Notiamo che nel codice ci sono due casi base: se l'array è vuoto, viene restituito -1, se invece ha un solo elemento, viene subito restituito l'indiceMedio.

Dato un array di dimensioni N, quante chiamate ricorsive devo fare per arrivare a considerare un caso base con una sottoparte dell'array con un solo elemento?

Risolvo:

$$N/2^{i-1} = 1$$

$$N = 2^{i-1}$$

$$i = 1 + \log_2 n$$

Per arrivare ad un caso base con 0 elementi, dovrei fare una chiamata in più, $i = 2 + \log_2 n$

Ogni chiamata effettua un numero costante di operazioni, possiamo quindi concludere che la complessità temporale è $\Theta(\log N)$

Complessità spaziale

L'algoritmo applicato ad un array di dimensione N, esegue un numero logaritmico di chiamate ricorsive.

Ogni chiamata, genera un frame d'attivazione che viene impilato sullo stack delle chiamate, ed occupa lo spazio necessario a memorizzare i parametri della funzione Array A, indice sx, indice dx, elemento x.

Essendo l'array A passato per riferimento, esso non occupa spazio. Consideriamo che ogni chiamata ricorsiva occupi spazio pari a 3.

Spazio totale: Numero di frame di attivazione * Dimensione frame = $\log_2 n * 3 = \Theta(\log n)$

Ricerca dicotomica iterativa

```
ALGORITMO ricercaBinariaIterativa(Array A[0..n-1], elemento x) -> indice
    sx <- 0
    dx <- n
    pos <- -1
    WHILE sx < dx AND pos = -1 DO
        m <- (sx + dx) / 2
        IF A[m] = x THEN
            pos <- m
        ELSE IF x < A[m] THEN
            dx <- m
        ELSE
            sx <- m + 1
    RETURN pos
```

L'algoritmo iterativo ha prestazioni:

- Tempo $\Theta(\log(n))$
- Spazio $O(1)$

Costi operazioni su array ordinato

Inserimento

Per inserire un elemento in un array ordinato, è necessaria un'operazione di ricerca iniziale, per determinare la posizione in cui inserire il nuovo elemento, ed un numero massimo di n spostamenti per "far spazio" al nuovo elemento da inserire.

L'inserimento in se, ha costo costante.

Tra le operazioni da effettuare, quella più onerosa sono gli spostamenti, quindi il costo è $\Theta(n)$.

Ricerca

Come abbiamo visto precedentemente, la ricerca dicotomica richiede un numero logaritmico di operazioni.

Eliminazione

Per eliminare un elemento da un array ordinato, è necessaria un'operazione di ricerca per determinare l'indice in cui l'elemento da eliminare si trova. Infine, devono essere eseguite una serie di $n - 1$ operazioni di spostamento (n è la dimensione dell'array pre-eliminazione) per "chiudere il buco" lasciato dall'eliminazione del nuovo elemento.

Costo pari a $\Theta(n)$.

Algoritmo di ordinamento

- **Ordinamento interno** - i dati da ordinare sono in memoria centrale
 - Accesso diretto e sequenziale agli elementi
 - **Ordinamento esterno** - i dati da ordinare si trovano su memoria di massa
 - Lentezza hardware dato dall'accesso alle periferiche
-

Stabilità

Un algoritmo di ordinamento è detto stabile, se esso durante l'ordinamento di un insieme di elementi, preserva l'ordine relativo tra record con la medesima chiave.

La stabilità di un algoritmo di ordinamento è la proprietà dello stesso di lasciare nelle posizioni *relative* originarie gli elementi che sono *di per sé distinti* ma che invece sono uguali per quanto concerne il criterio di ordinamento.

Esempio: in un gruppo di persone di varia altezza, Mario e Giuseppe hanno la stessa identica altezza e Mario si trova prima di Giuseppe.

Ordinando il gruppo per altezza crescente con un algoritmo **stabile**, Mario si troverà prima di Giuseppe anche dopo (i.e. l'algoritmo di ordinamento ha preservato le loro posizioni relative).

Usando un algoritmo non stabile, Mario **potrebbe** ritrovarsi **dopo** Giuseppe (non lo sappiamo per certo — dipende dai casi).

Complessità degli algoritmi di ordinamento

Valutata secondo:

- **Spazio** - Memoria aggiuntiva alla struttura da ordinare (eventuali altre strutture d'appoggio, stack nel caso di algoritmi ricorsivi)
- **Tempo** - Stimato in base al numero di confronti tra chiavi

Tempo di esecuzione = numero di confronti * tempo utilizzato da ciascun confronto

La complessità temporale degli algoritmi di ordinamento è stimata in base al numero di confronti, in quanto sono le operazioni "più costose" effettuate da questi algoritmi.

Se i confronti vengono effettuati in tempo costante (esempio: confronti tra interi), allora la stima dei confronti fornisce una stima del tempo di calcolo.

In caso di confronti tra stringhe per esempio, in cui un confronto tra esse non avviene in tempo costante, ma in tempo $O(m)$, dove m è la lunghezza della stringa, allora il numero di confronti sovrasta il tempo di calcolo.

Nota: il **limite inferiore degli algoritmi di ordinamento basati su confronti è $O(nlogn)$** , ciò significa che ogni algoritmo d'ordinamento basato esclusivamente su confronti tra elementi, ha una complessità temporale minima di $O(nlogn)$.

Algoritmi di ordinamento interno

Elementari

Sorting Algorithm	Time complexity	Space Complexity
Bubble sort	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$O(1)/O(n)$
Merge sort	$O(n \log(n))$	$O(n)$

Selection sort

Ad ogni iterazione del ciclo principale, identifico l'elemento relativo minore, e lo scambio con la posizione del primo elemento ancora non ordinato.

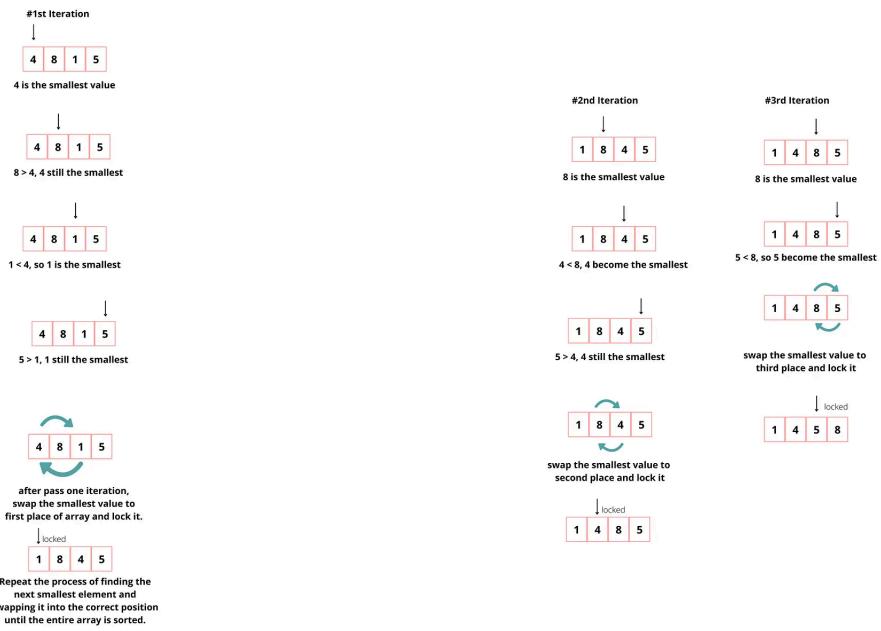
Al passo principale k , $k = 0, \dots, n - 1$, viene selezionato l'elemento che deve essere collocato nella posizione k . L'elemento viene collocato in tale posizione, dalla quale l'elemento non sarà più spostato.

Più in dettaglio:

1. Prima del passo principale k , i primi k elementi dell'array sono al loro posto definitivo, cioè sono ordinati tra loro e minori o uguali degli elementi successivi, i.e., $A[0] \leq A[1] \leq \dots \leq A[k - 1]$ e $A[k - 1] \leq A[j]$ per $j \geq k$;
2. si seleziona l'elemento che andrà collocato in posizione k , cioè il minimo della parte non ordinata (quindi il minimo tra $A[k], \dots, A[n - 1]$),
3. lo si colloca in posizione k , scambiandolo con l'elemento ivi presente,
4. in questo modo, dopo il passo principale k , i primi k elementi risultano collocati nella loro posizione definitiva.

Si può facilmente osservare che dopo il passo $n - 2$ la parte non ordinata contiene solo un elemento e, in base al punto 1, questo è maggiore o uguale dei precedenti e, dunque, si trova nella sua posizione definitiva. Pertanto non è necessario eseguire il passo $n - 1$.

Questo algoritmo non è stabile, l'Insertion Sort e il Bubble Sort invece sì.
Esempio: Ordinare [10a, 10b, 5] produce come risultato [5, 10b, 10a].



Esempio di implementazione in “GoLang”

```
package main

import "fmt"

func SelectionSort(arr []int) {
    for i := 0; i < len(arr)-1; i++ {
        minIndex := i
        for j := i + 1; j < len(arr); j++ {
            if arr[minIndex] > arr[j] {
                minIndex = j
            }
        }
        arr[i], arr[minIndex] = arr[minIndex], arr[i]
    }
}

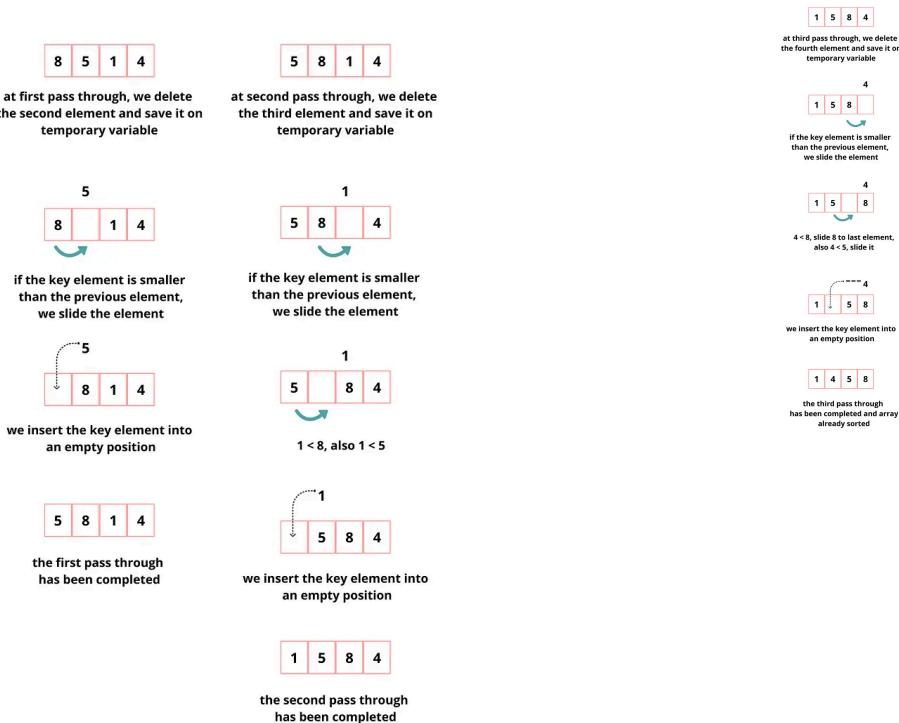
func main() {
    arr := []int{14, 12, 3, 4, 43, 11}
    SelectionSort(arr)
    fmt.Println(arr)
}
```

Il ciclo esterno, viene eseguito $n - 2$ volte.

Ad ogni iterazione k , vengono eseguiti $n - k - 1$ confronti.

Numero totale di confronti: $\sum_{k=0}^{n-2} (n - k - 1) = \frac{(n-1)*n}{2} = \Theta(n^2)$

Insertion sort



Esempio di implementazione in "GoLang"

```

package main

import "fmt"

func InsertionSort(arr []int) {
    for i := 1; i < len(arr); i++ {
        // Salva l'elemento corrente
        key := arr[i]
        j := i - 1

        // Sposta gli elementi a destra finché sono maggiori di "key"
        for j >= 0 && arr[j] > key {
            arr[j+1] = arr[j]
            j--
        }

        // Inserisci "key" nella posizione corretta
        arr[j+1] = key
    }
}

func main() {
    arr := []int{14, 12, 3, 4, 43, 11}
    InsertionSort(arr)
}

```

```

    fmt.Println(arr)
}

```

Ad ogni iterazione k, io sposto (e quindi effettuo confronti) al massimo k elementi.

Il ciclo esterno effettua n-1 iterazioni, pertanto il numero di confronti totale è $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2)$

Quanto allo spazio, vengono utilizzate sole due variabili aggiuntive, quindi lo spazio aggiuntivo è costante.

Se nel for principale, sostituissi $arr[j] > key$ con $arr[j] \geq key$, effettuerai scambi anche quando le chiavi sono uguali, perdendo la stabilità.

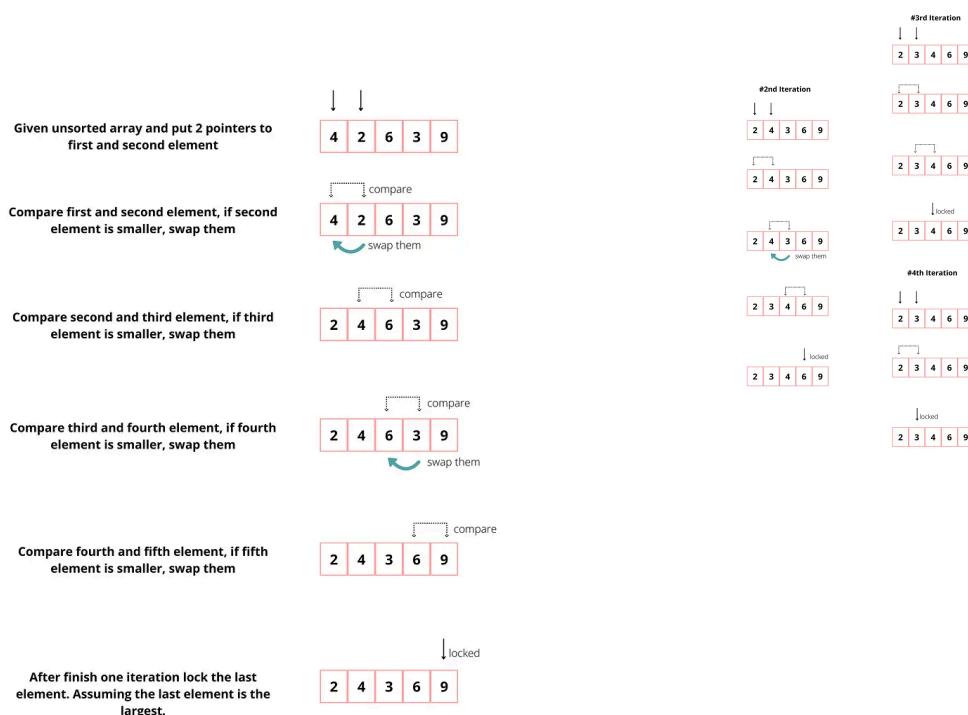
Bubble sort

Confronto gli elementi da sinistra verso destra, a due a due, scambiandoli quando l'elemento di sinistra è minore di quello di destra, o viceversa.

Dopo ogni iterazione i, gli ultimi i elementi saranno già nella loro posizione definitiva, pertanto non sarà necessario effettuare scambi.

Per questo motivo, il ciclo principale effettua n-1 iterazioni (gli ultimi n-1 elementi sono già ordinati, quell'ultimo elemento che rimane sarà automaticamente nella sua posizione definitiva)

Il ciclo principale, termina quando non sono stati effettuati scambi.



Esempio di implementazione in "GoLang"

```

package main

import "fmt"

func BubbleSort(arr []int) {
    n := len(arr)
    for passo := 0; passo < n-1; passo++ {

```

```

scambiato := false
for i := 0; i < n-1-passo; i++ { // Limita i confronti
    if arr[i] > arr[i+1] {
        arr[i], arr[i+1] = arr[i+1], arr[i]
        scambiato = true
    }
}
if !scambiato {
    break
}
}

func main() {
    arr := []int{14, 12, 3, 4, 43, 11}
    BubbleSort(arr)
    fmt.Println(arr)
}

```

Nel caso peggiore, il ciclo principale viene eseguito $n-1$ volte, e per ogni i -esima iterazione vengono effettuati $n-i-1$ confronti.
 Numero totale di confronti: $(n - 1) + (n - 2) + (n - 3) \dots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$

Bubble sort, potrebbe terminare prima delle $n-1$ iterazioni, se l'array viene ordinato prima.
 Nel caso migliore, il ciclo principale esegue una sola iterazione, e soli $n-1$ confronti.

Esempio:

Array: [1, 2, 3, 5, 4]

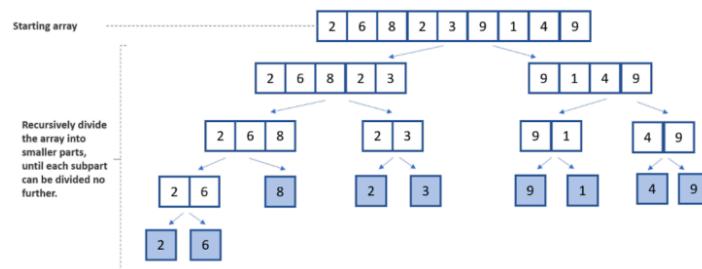
1. Iterazione 1: scambia 5 e 4 → [1, 2, 3, 4, 5]
2. Iterazione 2: nessun scambio → algoritmo **termina anticipatamente**

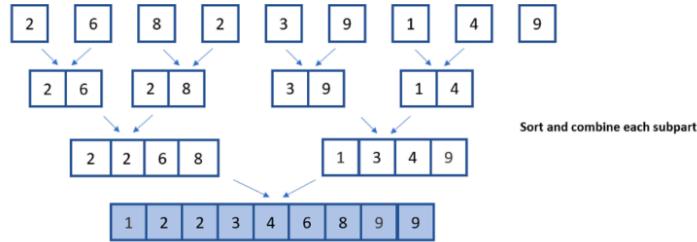
Avanzati

Merge sort

Basato sul principio "divide and conquer", divido ricorsivamente l'array da ordinare in due, finché non ottengo gruppi da singoli elementi.

La ricorsione poi risale, combinando tra loro ad ogni iterazione, due array già ordinati.

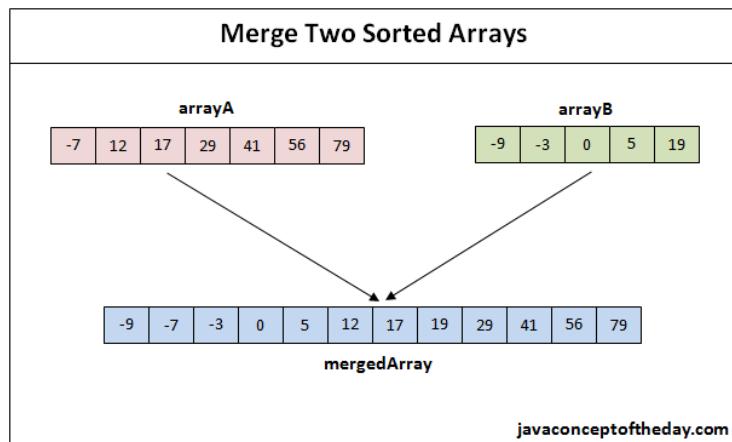




Algoritmo di “fusione” di due array:

Dati due array A e B da unire, considero due indici $i - j$ uno per ogni array A e B.

Ad ogni passo, confronto $A[i]$ con $B[j]$, accodo in un array ausiliario X l’elemento maggiore/minore, e incremento l’indice dell’array da cui ho preso l’elemento ($i o j$)



```

ALGORITMO merge(Array B[0...lenB-1], Array C[0...lenC-1]) -> Array
sia X[0 ... lenB + lenC - 1] un array
(quello che conterrà alla fine, tutti gli elem. di B e C)

indiceB <- 0, indiceC <- 0, k <- 0

WHILE indiceB < lenB AND indiceC < lenC DO
// itero fino a quando nessuno dei due array è finito
| IF B[indiceB] <= C[indiceC] THEN
| | X[k] <- B[indiceB]
| | indiceB <- indiceB + 1
| ELSE
| | X[k] <- C[indiceC]
| | indiceC <- indiceC + 1
| k <- k + 1

//Se ci sono ancora elementi di B o C rimasti.
WHILE indiceB < lenB DO
| X[k] <- B[indiceB]
| indiceB <- indiceB + 1
| k <- k + 1

WHILE indiceC < lenC DO
| X[k] <- C[indiceC]
  
```

```

| indiceC <- indiceC + 1
| k <- k + 1

```

```
RETURN X
```

Complessità temporale: Dati due array B e C da unire, il numero di confronti effettuati dalla funzione merge nel caso peggiore, è $C_{merge} = n - 1$, dove $n = \text{len}(B) + \text{len}(C)$

Complessità spaziale: si utilizzano due indici (variabili intere), ed un array aggiuntivo. Lo spazio è costante.

Implementazione Merge sort, con tecnica “divide-et-impera”

Caso base: se l'array A ha lunghezza 0 o 1, è sicuramente già ordinato.

Altrimenti, ricorsivamente:

- Dividi l'array A in 2 parti
- Ordinalo separatamente (ricorsione)
- “Fondi” le due parti.

```

ALGORITMO MergeSort(Array A[0...n-1])
  IF n > 1 THEN
    m <- n/2
    B <- A[0...m-1] //prima metà dell'array
    C <- A[m...n-1] //seconda metà dell'array

    //ordino le due parti
    MergeSort(B)
    MergeSort(C)

    //combino le due soluzioni
    A <- merge(B,C)

```

Complessità temporale dell'algoritmo MergeSort:

Analizziamo l'algoritmo MergeSort:

- Se $n = 0, n = 1$, complessità pari a 0. ($C_{MergeSort}(n) = 0$)
- Se $n > 1$, la complessità di MergeSort è pari al numero di confronti da effettuare per ordinare la parte sinistra, sommata al numero di confronti per ordinare la parte destra, sommata al numero di confronti necessari per fondere le due parti:

$$C_{MergeSort}(n) = C_{MergeSort}\left(\frac{n}{2}\right) + C_{MergeSort}\left(\frac{n}{2}\right) + C_{Merge}(n) = 2C_{MergeSort}\left(\frac{n}{2}\right) + (n - 1)$$

Con $C_{MergeSort}\left(\frac{n}{2}\right)$ pari al numero di confronti sull'array di dimensione dimezzata, $n - 1$ il numero di confronti della procedura “merge”. Il numero di confronti da effettuare per ordinare la parte SX e DX dell'array sono uguali solo se n è pari. Quindi $C_{MergeSort}\left(\frac{n}{2}\right) + C_{MergeSort}\left(\frac{n}{2}\right) = 2C_{MergeSort}\left(\frac{n}{2}\right)$ solo per n pari.

$$\begin{aligned} C(n) &= 2C\left(\frac{n}{2}\right) + n - 1 = \\ &= 2[2C\left(\frac{n}{2^2}\right) + \frac{n}{2} - 1] + n - 1 = 2^2C\left(\frac{n}{2^2}\right) + n - 2 + n - 1 \\ &= 2^2[2C\left(\frac{n}{2^3}\right) + \frac{n}{2^2} - 1] + n - 2 + n - 1 \end{aligned}$$

$$\begin{aligned} &\vdots \\ &2^kC\left(\frac{n}{2^k}\right) + kn - \sum_{i=0}^{k-1} 2^i \\ &2^kC\left(\frac{n}{2^k}\right) + kn - 2^k + 1 \end{aligned}$$

Proviamo a sostituire $C\left(\frac{n}{2^k}\right)$ con qualcosa che conosciamo, per esempio $C(1)$, che sappiamo essere 0.

Se $\frac{n}{2^k}$ fosse 1, cioè $k = \log_2 n$, $C(1) = 0$.

Sostituiamo $k = \log_2 n$, rimane $n \log_2 n - n + 1 = \Theta(n \log n)$

Complessità spaziale:

L'altezza dello stack di ricorsione H è

- Pari a 1, nel caso base (Lunghezza dell'array pari a 0 o 1)
- Pari a $1 + \max(H(\lfloor \frac{n}{2} \rfloor), H(\lceil \frac{n}{2} \rceil))$ se $n > 1$

Se n è una potenza di 2, $H(n) = 1 + \log_2 n = \Theta(\log(n))$

L'algoritmo utilizza però un array ausiliario, che occupa $\Theta(n)$, maggiore della dimensione dello stack $\Theta(\log(n))$, pertanto lo spazio utilizzato è $\Theta(n)$.

Merge Sort Ottimizzato

Con l'implementazione classica di MergeSort, ad ogni chiamata della procedura su array A di lunghezza > 1:

- Si creano due nuovi array B e C (Spreco di memoria)
- Si devono copiare in essi, le due metà dell'array A (Spreco di tempo)

Possiamo modificare l'algoritmo, in modo che operi con un solo array ausiliario X.

```
ALGORITMO MergeSort(Array A[0...lenA-1])
    Sia X un Array di lunghezza pari a quella di A
    MergeSort(A,0,lenA,X)

PROCEDURA MergeSort(Array A, indiceInizio, indiceFine, Array X){
    IF indiceFine-indiceInizio > 1 THEN
        indiceMedio = (indiceInizio+indiceFine)/2
        MergeSort(A,indiceInizio,indiceMedio,X)
        MergeSort(A,indiceMedio,indiceFine,X)
        merge(A,indiceInizio,indiceMedio,indiceFine,X)
    }

ALGORITMO merge(Array A[0...indiceFine-1], indiceInizio, indiceMedio, IndiceFine, Array X[0...Indic
eFine-1])
    indiceB <- indiceInizio, indiceC <- indiceMedio, k <- 0

    WHILE indiceB < indiceMedio AND indiceC < IndiceFine DO
        // Itero fino a quando entrambi i segmenti hanno elementi
        | IF A[indiceB] <= A[indiceC] THEN
        | | X[k] <- A[indiceB]
        | | indiceB <- indiceB + 1
        | ELSE
        | | X[k] <- A[indiceC]
        | | indiceC <- indiceC + 1
        | k <- k + 1

        // Copia degli elementi rimanenti del segmento sinistro (se presenti)
        WHILE indiceB < indiceMedio DO
            | X[k] <- A[indiceB]
            | indiceB <- indiceB + 1
```

```

| k <- k + 1

// Copia degli elementi rimanenti del segmento destro (se presenti)
WHILE indiceC < IndiceFine DO
| X[k] <- A[indiceC]
| indiceC <- indiceC + 1
| k <- k + 1

A=X //copio il contenuto di X, in A (For loop)

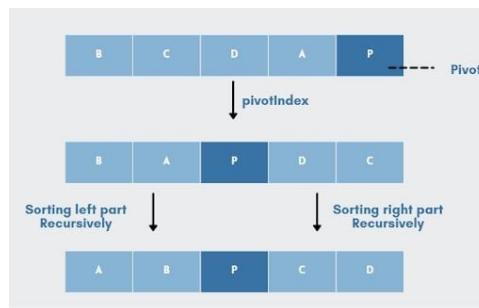
```

QuickSort

Algoritmo ricorsivo.

Fa uso di una procedura "partiziona" il cui scopo è individuare, un pivot dell'array da ordinare (pari al primo o ultimo elemento dell'array), cioè un valore per cui tutti gli elementi di indice minore ad esso, sono minori del pivot, e tutti gli elementi di indice maggiore, sono maggiori o uguali al pivot.

Individuato e posizionato correttamente il pivot, si procede ricorsivamente ad ordinare le due metà dell'array a sinistra e destra del pivot.



```

ALGORITMO quickSort(Array A[0...n-1])
    quickSort(A,0,n)

PROCEDURA quickSort(Array A, indiceInizio, indiceFine)
    IF indiceFine-indiceInizio>1 THEN
        indicePivot <- partiziona(A,indiceInizio,indiceFine) //trova pivot
        quickSort(A,indiceInizio,indicePivot)
        quickSort(A,indicePivot+1,indiceFine)

```

Procedura "partiziona" per individuare il pivot (Primo elemento dell'array) e posizionarlo correttamente:

```

ALGORITMO partiziona (Array A, indiceInizio, indiceFine) -> indice
    perno <- A[indiceInizio]
    sx<-indiceInizio
    dx<-indiceFine
    WHILE sx<dx DO
        DO dx <- dx-1 WHILE A[dx]>perno
        DO sx <- sx+1 WHILE sx<dx AND A[sx]<=perno
        IF sx<dx THEN
            Scambia A[sx] con A[dx]
        Scambia A[indiceInizio] con A[dx]
    RETURN dx

```

Numero di confronti $C(n)$:

Se Numero di elementi nell'array è ≤ 1 , 0 confronti.

Altrimenti, dato un pivot in posizione k , il numero di confronti è pari a:

$$C_{QuickSort}(n) = C_{partiziona}(n) + C_{QuickSort}(k) + C_{QuickSort}(n - k - 1)$$

Dove:

k = Posizione del pivot

$C(k)$ Numero Confronti per ordinare parte SX del pivot

$C(n - k - 1)$ Numero confronti per ordinare parte DX del pivot

$C_{partiziona}(n) = N$ (ogni elemento, a sx o dx del perno, va confrontato col perno).

Caso peggiore, si ottiene quando la divisione tra parte sx e dx del pivot è completamente sbilanciata, cioè se scelgo come pivot l'elemento minore o massimo dell'array (rispettivamente in posizione 0 o $n - 1$ nell'array finale). Ipotizzando di scegliere il primo elemento come pivot, in posizione $k = 0$, sostituisco k nell'equazione ottenendo:

$$C_{worst}(n) = n + \max(C_{worst}(k) + C_{worst}(n - k - 1) | k = 0 \dots n - 1) = n + C_{worst}(n - 1)$$

Equazione di ricorrenza:

$$C_{worst} = n + C_{worst}(n - 1) = n + n - 1 + C_{worst}(n - 2) = n + (n - 1) + (n - 2) + (n - 3) \dots + C_{worst}(1)$$

L'equazione di ricorrenza si riduce alla somma di tutti i numeri da 1 a N , che è uguale al semiprodotto $\frac{n*(n-1)}{2}$, che è circa $\theta(n^2)$

Per evitare il caso peggiore, va scelto come pivot un elemento che non sia né l'ultimo né il primo: si può scegliere o un elemento random, oppure la mediana dei tre elementi primo, ultimo e centrale.

Caso migliore, quando la divisione tra parte sx e dx del pivot è bilanciata, cioè quando il pivot è in mezzo: se $k = \frac{n}{2}$, allora:

$$C_{best} = n + \min(C_{best}(k) + C_{best}(n - k - 1) | k = 0 \dots n - 1) = n + 2C_{best}(\frac{n}{2}), \text{ cioè circa } n\log n.$$

Caso medio:

$$C_{medio} = \frac{\sum_{k=0}^{n-1} [n + C(k) + C(n - k - 1)]}{n}, \text{ che si dimostra essere sempre circa } n\log n.$$

Formule utili

Trovare la più piccola potenza di 2, che contenga un numero binario:

Esempio:

11010 (2) → shift a sinistra (aggiungere uno zero a destra) → 110100(2) → considero solo la prima cifra a 1, pongo tutte le altre a 0 → 100000(2)

La più piccola potenza di 2, che contiene 11010(2) è 100000(2).

Tecnica “divide-et-impera”

Data un'istanza di un problema, posso dividerla in istanze più piccole, risolverle singolarmente, combino le soluzioni.

In informatica, questa tecnica è spesso implementata **ricorsivamente**.

```

ALGORITMO risolviProblema (Istanza I) -> Soluzione //istanza del problema
  IF |I| <= C THEN
    //Se la lunghezza dell'istanza del problema, è già "abbastanza corta" (minore di una costante)
    RETURN Soluzione
  ELSE
    dividi I in I1, I2, Im, con |Ij| < |I| per j=1,2...m //dividi in istanze di lunghezza minore di I originale
    Sol1 <- risolviProblema(I1)
    Sol2 <- risolviProblema(I2)
    *
    *
    *
    Solm <- risolviProblema(Im)

  RETURN combina(Sol1,Sol2...Solm)

```

$$T(I) = \text{costante, se } |I| \leq C$$

Altrimenti

$$T(I) = T(\text{Dividi } I) + T(I1) + T(I2)...T(Im) + T\text{combina}(Sol1, Sol2...Solm)$$

Tipo di variabile

Attributo che specifica l'insieme di valori una variabile può assumere, e le operazioni effettuabili su di essa.

Dizionario

Collezione di elementi, in cui ognuno è identificato univocamente mediante chiave.

Struttura dati

Organizzazione delle informazione che permette di implementare un tipo di dati. (Strutture indicizzate, collegate, ...)

Stesso tipo, non significa stessa struttura dati.

Esempio: Un dizionario, può essere implementato utilizzando un array ordinato, o uno non ordinato.

Collezioni:

Strutture indicizzate (Array)

Allocate in porzione di memoria contigua, tempo di accesso indipendente dalla posizione del dato (accesso diretto), dimensione statica, definita a compile-time.

Strutture collegate

Composte da elementi (nodi) connessi tra loro.

Non è necessario che questi siano memorizzati in porzioni di memoria contigue, dimensione variabile dinamicamente
In base alla tipologia di collegamenti, posso implementare liste, alberi, pile, coda, grafi.

Linked lists

Insieme di nodi, connessi linearmente l'uno dopo l'altro.

Ogni nodo contiene due informazioni:

- Un dato della collezione (che potrebbe essere a sua volta una collezione di dati)
- Chiave (facoltativo, per identificare il nodo)
- Informazione per l'accesso al nodo successivo (puntatore).

Ricerca elemento in base alla posizione (indice)

```
FUNZIONE elemento(Lista L, intero i) -> Nodo
    p<-L
    WHILE p!=null AND i>0 DO
        p<-p.next
        i<-i-1
    return p
```

Ricerca elemento in base alla chiave

```
FUNZIONE elementoChiave(Lista L, tipoChiave K) -> Nodo
    p<-L
    WHILE p!=null AND p.chiave!=k DO
        p<-p.next
    return p
```

Ricerca elemento in base alla chiave, in una Lista Ordinata

```
FUNZIONA trova(ListaOrdinata L, tipoChiave k) -> Nodo
    p<-L
    while p!=null AND p.chiave < k DO //vado avanti nella lista, finchè la chiave del nodo corrente, è minore di quella da cercare.
        p<-p.pros
    IF p=null OR p.chiave>k THEN //se sono fuori dal while, o l'elemento non esiste, oppure lo resto intuisco
        RETURN null
    ELSE
        RETURN p
```

Inserimento in lista ordinata

```
FUNZIONE inserisci(ListaOrdinata L, elemento d) -> ListaOrdinata
    k <- d.chiave
    p <- L
    prec <- null
    WHILE p!=null AND p.chiave<k DO
        prec<-p
        p<-p.pros
    newNode <- nuovo nodo vuoto
```

```

newNode.chiave <- k
newNode.next <- p

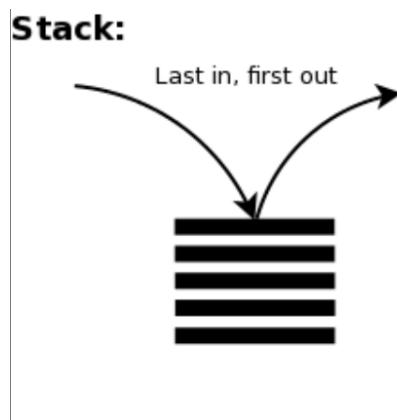
if prec=null THEN
    L<-newNode
ELSE
    prec.next <- newNode

RETURN L

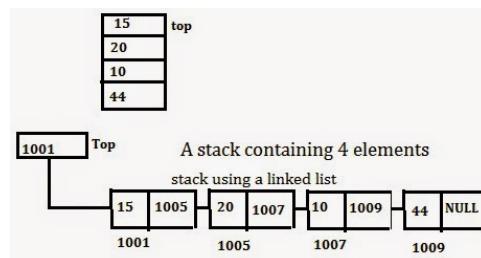
```

Pila (Stack)

Struttura dati di tipo LIFO (Last In - First Out).



Implementazione mediante liste concatenate



Metodi

- **isEmpty() → Boolean**

```

FUNZIONE isEmpty() -> boolean
IF Top = null
    RETURN true
ELSE
    RETURN false

```

- **push(value)**

```

PROCEDURA push (valore x)
newNode <- NuovoNodoVuoto
newNode.dat0 <- x
newNode.next = Top
Top <- newNode

```

- **pop() → value**

```
FUNZIONE pop() -> value
    elementoDaRestituire <- Top.dato
    Top <- Top.next
    RETURN elementoDaRestituire
```

- **top () → value**

```
FUNZIONE top() -> value
    RETURN Top.dato
```

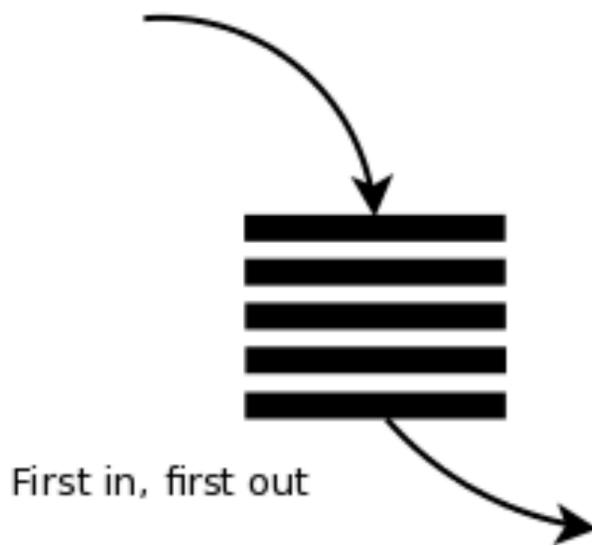
Tutte le operazioni eseguibili sulla pila, vengono eseguite in tempo $O(1)$

Posso implementare una pila anche tramite un array, inserendo gli elementi in coda, togliendo dalla coda.

Coda

Struttura dati di tipo FIFO (First In - First Out).

Queue:



Implementazione mediante liste concatenate

Se io considerassi una lista concatenata, in cui tengo traccia del solo valore del primo elemento inserito (cima della lista), io posso eseguire l'operazione di cancellazione di un elemento in $O(1)$, ma dovrei scorrere la lista fino a trovare l'ultimo elemento (coda della lista), per "accodare" un elemento.

Per far sì che sia l'aggiunta che la rimozione possano essere eseguite in $O(1)$, tengo traccia di due puntatori, testa e coda della lista.

Una coda vuota, ha entrambi i puntatori `puntatoreTesta` e `puntatoreCoda` pari a `null`.

Rimuovo dalla testa (testa si intende il primo elemento inserito nella coda), aggiungo in coda

Metodi

- `isEmpty() → Boolean`

```
FUNZIONE isEmpty() -> Boolean
    if puntatoreTesta == null
        return true
    ELSE
        return false
```

- `First() → value`

```
FUNZIONE First() -> value
    RETURN puntatoreTesta.dato
```

- `deQueue() → value`

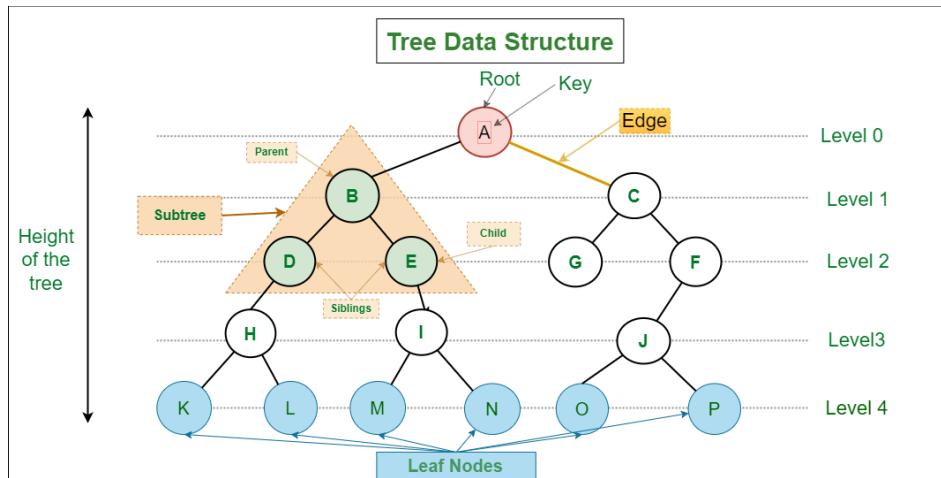
```
FUNZIONE deQueue() -> value
    valorePrimoElemento <- puntatoreTesta.dato
    puntatoreTesta <- puntatoreTesta.next
    IF puntatoreTesta == null THEN
        puntatoreCoda = null
    RETURN valorePrimoElemento
```

- `enqueue(value)`

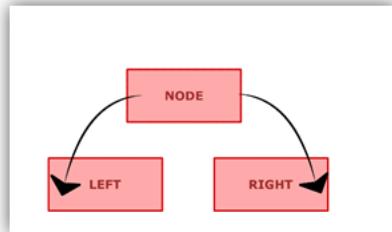
```
PROCEDURA enqueue(value x)
    newNode <- NuovoNodoVuoto
    newNode.dato <- x
    newNode.next <- NULL //sarà il nuovo nodo di coda.

    IF puntatoreTesta == null THEN //se la lista è vuota, sia la testa che la coda puntano allo
    stesso elemento
        puntatoreTesta <- newNode
        puntatoreCoda <- newNode //l'ultimo viene aggiornato
    ELSE
        puntatoreCoda.next <- newNode //il "penultimo", punta ora a newNode (ultimo), non più a
    null
        puntatoreCoda <- newNode //l'ultimo viene aggiornato
```

Alberi Binari



Implementati secondo liste lineari, in cui ogni nodo include due riferimenti, ad un figlio destro e sinistro (che ricorsivamente, rappresentano le radici di un sottoalbero sinistro e destro).



Visita in ampiezza (implementata utilizzando Coda C)

```

ALGORITMO visitaInAmpiezza(AlberoBinario radice)
    C<-coda vuota, in cui andranno i nodi "ancora da visitare"
    C.enqueue(radice)

    WHILE not(C.isEmpty()) DO
        nodoDaVisitare <- C.dequeue()
        IF nodoDaVisitare != null
            visita nodo "nodoDaVisitare" (stampo, eseguo operazioni sul nodo...)
            C.enqueue(nodoDaVisitare.sx)
            C.enqueue(nodoDaVisitare.dx)
    
```

DFS

DFS è una tecnica di ricerca, che consiste nell'esplorare prima i nodi a profondità maggiore, prima di retrocedere e proseguire nella ricerca per livelli di profondità minori.

In base all'ordine di visita dei nodi durante la DFS, distinguo 3 tipi di ricerca:

DFS Pre-order

Visito prima la radice, poi il sottoalbero sinistro, poi il destro.

Implementazione con Lista

```

ALGORITMO visitaInProfondità(AlberoBinario radice)
    <-Lista vuota, in cui andranno i nodi "ancora da visitare"
    L.push(radice)

    WHILE not(L.isEmpty()) DO
        nodoDaVisitare <- L.pop()
        IF nodoDaVisitare != null
            visita nodo "nodoDaVisitare" (stampo, eseguo operazioni sul nodo...)
            L.push(nodoDaVisitare.sx)
            L.push(nodoDaVisitare.dx)

```

Implementazione ricorsiva

```

ALGORITMO visitaInProfonditàRecursive(AlberoBinario radice)
    IF radice != null THEN //se l'albero non è vuoto
        //visita della radice
        visitaInProfonditàRecursive(radice.sx)
        visitaInProfonditàRecursive(radice.dx)

```

DFS In-order

Visito prima il sottoalbero sinistro, poi la radice, poi il destro.

Implementazione ricorsiva

```

ALGORITMO visitaInProfonditàRecursive(AlberoBinario radice)
    IF radice != null THEN //se l'albero non è vuoto
        visitaInProfonditàRecursive(radice.sx)
        //visita della radice
        visitaInProfonditàRecursive(radice.dx)

```

DFS Post-order

Visito prima il sottoalbero sinistro, poi il destro, poi la radice.

Implementazione ricorsiva

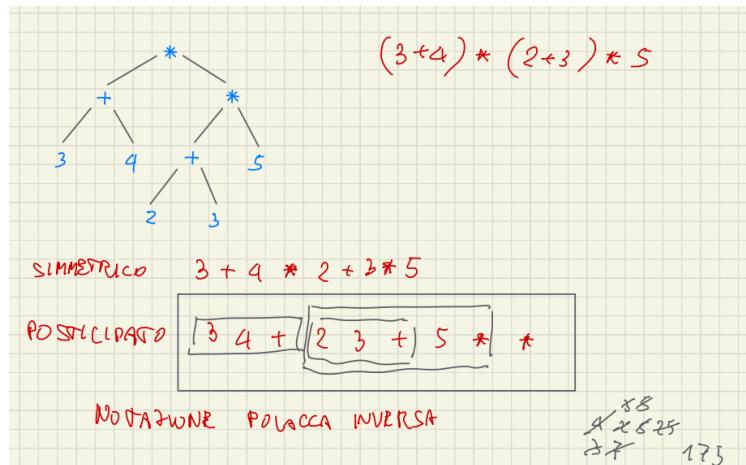
```

ALGORITMO visitaInProfonditàRecursive(AlberoBinario radice)
    IF radice != null THEN //se l'albero non è vuoto
        visitaInProfonditàRecursive(radice.sx)
        visitaInProfonditàRecursive(radice.dx)
        //visita della radice

```

La visita post-order di un albero di un'espressione, produce la sua notazione polacca inversa.

Questa permette di calcolare espressioni matematiche correttamente, senza l'uso delle parentesi



Interpretare la **RPN**, prevede l'uso di una pila:

Leggo la notazione da SX a DX

- Quando trovo un numero, lo "pusho" sullo stack
- Quando trovo un operatore, estraggo gli ultimi due elementi dello stack, ne applico l'operatore, e pusho il risultato sullo stack.

Iterazione 1	Iterazione 2	Iterazione 3	Iterazione 4	Iterazione 5	Iterazione 6	Iterazione 7	Iterazion
			3		5		175
	4		2	5	5	25	
3	3	7	7	7	7	7	

Alberi generici

Definizione ricorsiva di un albero: Un albero con radice è:

- Una struttura vuota
- Un nodo a cui sono associati $K \geq 0$ alberi.

Profondità di un nodo:

- 0 se è la radice
- $k + 1$ per i nodi figli di un nodo a profondità k

Grado di un nodo: numero dei suoi figli

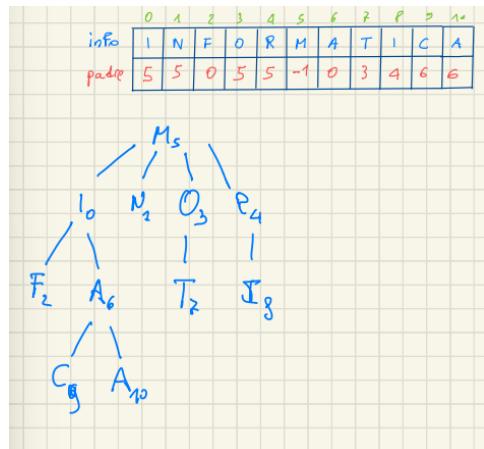
Grado di un albero: massimo grado tra tutti i suoi nodi

Altezza di un albero: percorso più lungo per andare dalla radice alla sua foglia più profonda.

Rappresentazione tramite "vettore dei padri"

Due array:

- Array con i valori dei nodi
- Array contenente la posizione (nell'array precedente) del padre di ciascun nodo (-1 per la radice)



Il nodo "I", di posizione 0, ha il padre in posizione 5 (M)

Il nodo "N", di posizione 1, ha padre in posizione 5 (M)

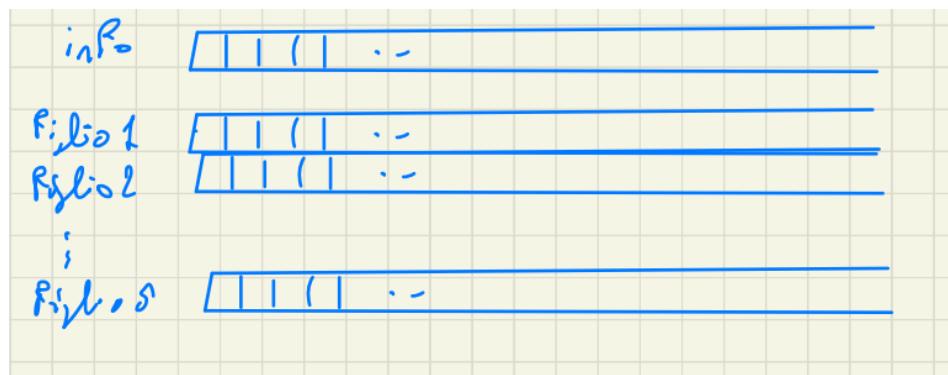
...

Costruisco così l'albero.

Rappresentazione tramite vettore dei figli

Si tengono in memoria:

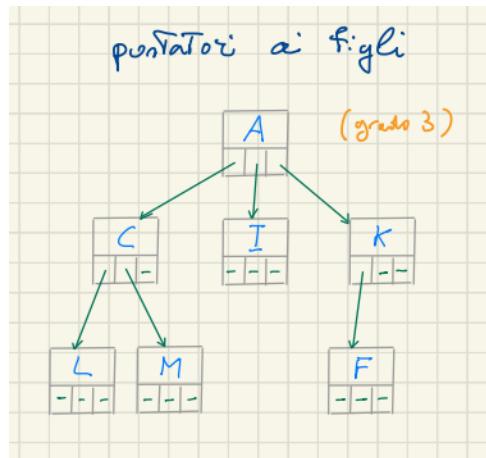
- Un array A con i valori dei nodi
- k array (dove k è il grado dell'albero), che indicano per ogni nodo in posizione i nell'array A, i suoi figli.



Rappresentazioni collegate

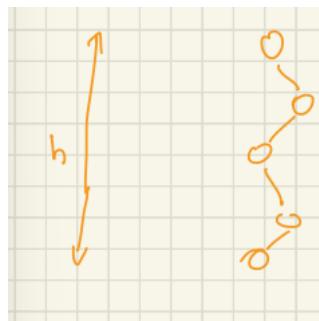
Implementati come liste, con puntatori.

E' necessario conoscere il grado massimo dei nodi, per poterne definire la struttura (sapere quanti puntatori mettere nella struct del nodo)



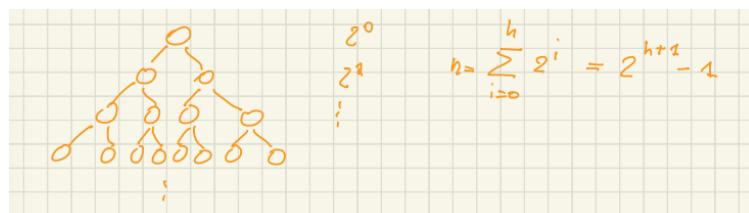
Relazione tra numero di nodi e altezza in un albero binario

Considerando la radice a profondità 0, il numero minimo di nodi in un albero di altezza h è $h + 1$.



$$h = 4, n = h + 1 = 5$$

Numero massimo di nodi per alberi di altezza h



$$\text{Da queste due, abbiamo che } h = \log_2(n)$$

Alberi binari completi

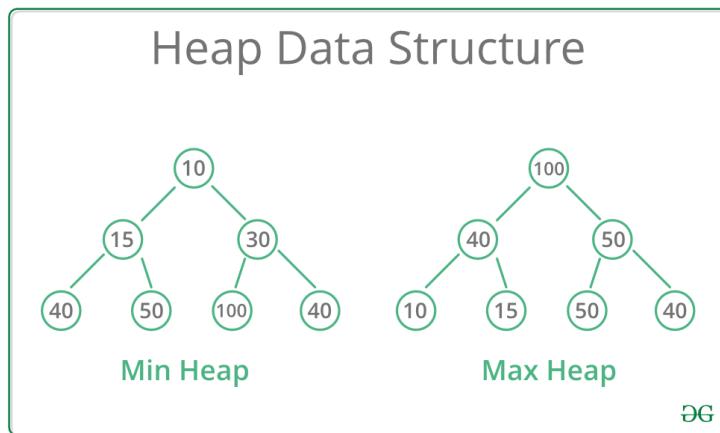
Un albero binario è completo se tutte le fogli sono alla stessa profondità, e ogni nodo interno ha entrambi i figli.

Alberi binari “quasi completi”

Un albero binario è quasi completo, se ogni nodo, tranne quelli di ultimo e penultimo livello, possiede entrambi i figli. (se è completo fino al penultimo livello)

Heap

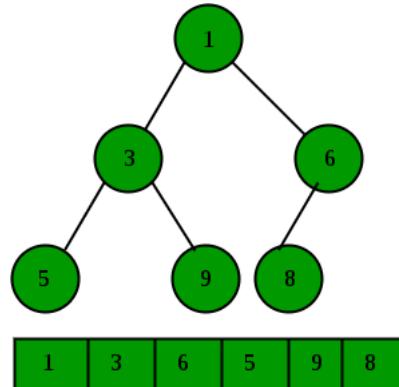
Uno heap (mucchio), è un albero binario quasi completo, in cui la chiave contenuta in ciascun nodo, è maggiore/minore o uguale delle chiavi contenute nei figli.



Per comodità, consideriamo heap in cui le foglie dell'ultimo livello si trovano più a sinistra possibile.

Rappresentazione di un heap

Uno heap è generalmente rappresentato in memoria come un array, utilizzando un ordinamento per livelli:



Risistemare uno heap

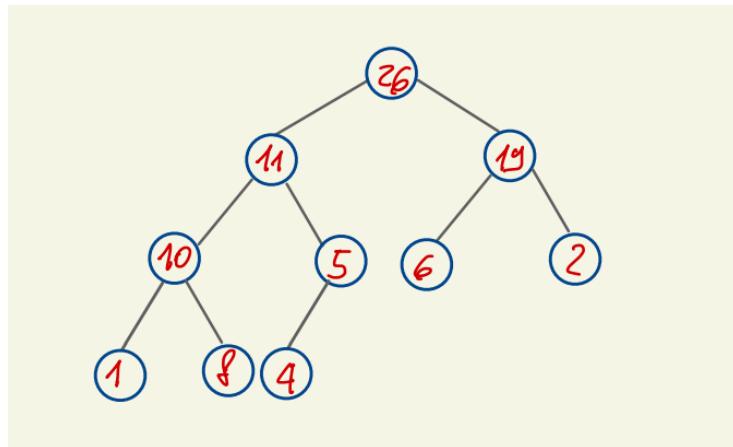
Dato uno heap, è possibile costruire un array ordinato:

- Estraggo la radice
- Sostituisco la radice con la foglia più a destra dell'ultimo livello (considerando l'albero come a "strati orizzontali"), ottenendo uno heap sbagliato con la sola radice da risistemare.
La procedura risistema(Albero) sistema uno heap con la sola radice sbagliata.

La strategia consiste nel far scendere la radice verso il basso, scambiandola man mano col figlio più grosso (se più grosso di essa)

Sostituisco la radice con la foglia più a destra dell'ultimo livello, perchè considerando l'heap rappresentato come un array, se io riempissi il "buco" lasciato dalla radice con un qualsiasi elemento, dovrei poi shiftare tutti gli altri elementi per risanare il buco. Sostituendo invece con l'ultimo elemento, il buco ce l'ho in coda, quindi più facile da risolvere.

Esempio:



Estraggo la radice "26", la sostituisco con la foglia più a destra, dell'ultimo livello (Ultimo livello: 1,8,4. Foglia più a destra dell'ultimo livello: 4).

```

PROCEDURA risistema(Heap H)
v <- H
x <- v.chiave
y <- v.campi
da_collocare <- true
WHILE da_collocare DO
    IF v è una foglia THEN
        da_collocare <- false
    ELSE
        u <- figlio di v di valore massimo
        IF u.chiave > x THEN
            v.chiave <- u.chiave
            v.campi <- u.campi
            v <- u
        ELSE
            da_collocare <- false
        v.chiave <- x
        v.campi <- y
  
```

Nel caso peggiore la radice scende all'ultimo livello.

Poichè si fanno 2 confronti per figlio, in un albero alto h il numero di confronti è $\Theta(h)$.

Ricordando che in un albero binario quasi completo con n nodi l'altezza è massimo $h = \lceil \log_2(n) \rceil$, si conclude che il numero di confronti è $\Theta(\log(n))$ con n numero di nodi.

Costruzione di uno heap, soluzione iterativa

Dato un albero binario quasi completo si vogliono riordinare i nodi in modo tale che questo rispetti la definizione di heap.

```

PROCEDURA creaHeap(Albero T)
    h <- altezza di T
    FOR p <- h DOWN TO 0 DO
        FOREACH nodo x di profondità p DO
            risistema(x)

```

Costruzione di uno heap, soluzione ricorsiva

```

PROCEDURA creaHeap(Albero T)
    /* Trasforma l'albero binario T in uno heap */
    if T != albero vuoto then
        creaHeap(T.sx)
        creaHeap(T.dx)
        risistema(T)

```

Numero di confronti di creaHeap:

La procedura creaHeap (iterativa), richiama la procedura risistema un certo numero di volte, per alberi di altezza via via crescendo.

Per effettuare l'analisi nel caso peggiore, supponiamo di applicare creaHeap ad un albero binario completo, di altezza h .

Per ogni nodo n a profondità p , il nodo n è radice di un sottoalbero di altezza $h - p$ (p va da h a 0)

Quindi per ogni nodo n a profondità p , la procedura risistema effettua $O(h - p)$ confronti.

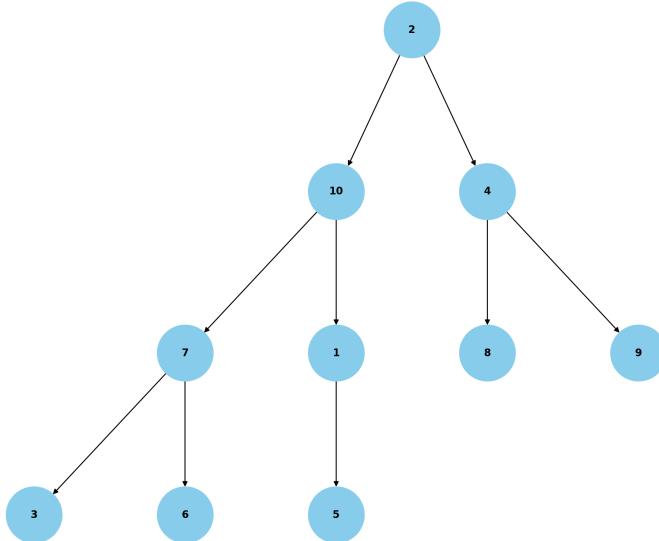
Per ogni livello di profondità p , ci sono 2^p nodi, quindi per ogni livello si eseguono $O((h - p)2^p)$ confronti.

Per l'intero albero di altezza h si effettuano allora $O(\sum_{p=0}^h (h - p)2^p) = O(n)$ confronti, con n pari al numero di nodi nello heap.

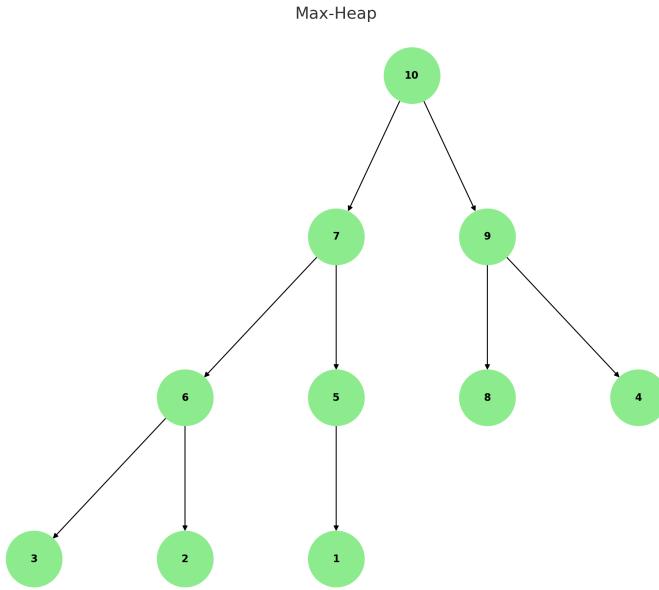
Da Array ad albero binario quasi completo, a Max Heap.

Dall'array [2, 10, 4, 7, 1, 8, 9, 3, 6, 5], costruisco un albero binario quasi completo, inserendo i numeri in ordine da sinistra a destra, collocandoli per livelli da sinistra verso destra.

Albero Binario Quasi Completo



Da questo, utilizzo la procedura creaHeap, che iterativamente, partendo dai nodi di livello più basso, fino alla radice, esegue la procedura `risistema`, su alberi di altezza via via maggiori.



Da questo, procedo con l'algoritmo di sorting vero e proprio, estraendo la radice, e risistemando l'albero.

Heap Sort

L'algoritmo HeapSort, si basa sulla struttura dati heap.

Da un array non ordinato, si costruisce un albero binario quasi completo, lo si trasforma in un heap, e si procede poi a svuotarlo rimuovendo i valori dalla radice, e risistemandolo finché non rimane vuoto.

Possiamo ottimizzare l'algoritmo, lavorando in loco sull'array che rappresenta l'heap da ordinare.

Nella rappresentazione ad array, per ogni nodo in posizione i , i due figli sono rispettivamente in posizione $2i + 1$ e $2i + 2$.

Considerato quindi l'albero binario quasi completo, rappresentato tramite un array, trasformiamolo in uno heap, cambiando l'implementazioni di `risistema` e `creaHeap`.

```

PROCEDURA risistema(Array A[0..n-1], Intero l, Intero r)
v <- r
x <- A[v]
da_collocare <- true
WHILE da_collocare DO
  IF 2 * v + 1 >= l THEN
    da_collocare <- false
  ELSE
    u <- 2 * v + 1
    IF u + 1 < l AND A[u + 1] > A[u] THEN
      u <- u + 1
    IF A[u] > x THEN
      A[v] <- A[u]
      v <- u
    ELSE
      da_collocare <- false
    A[v] <- x
  
```

Costruzione di un heap, in loco

```

PROCEDURA creaHeap(Array A[0..n-1])
FOR i <- n / 2 DOWNT0 0 DO
    risistema(A, i, n)

```

L'implementazione di HeapSort, diventa a questo punto banale:

```

PROCEDURA heapSort(Array A[0..n-1])
creaHeap(A)
FOR l <- n - 1 DOWNT0 1 DO // l'indice l tiene conto dell'ultima foglia dello heap
    A[0], A[l] <- A[l], A[0]
    risistema(A, 0, l)

```

Complessità temporale:

creaHeap esegue $O(n)$ confronti.

Il ciclo principale, chiama la procedura risistema n volte.

risistema, esegue $O(\log n)$ confronti.

Quindi l'algoritmo esegue in totale $O(n) + O(n \log n)$ confronti = $O(n \log n)$.

L'algoritmo opera in loco, quindi non utilizza spazio.

Operazioni su heap

Dato un (max/min)heap, si descrivono una serie di operazioni effettuabili su di esso:

Trovare l'elemento di chiave massima/minima

In un max heap, l'elemento di chiave massima è la radice.

L'operazione viene eseguita in tempo costante.

Cancellare l'elemento di chiave massima

Si estrae la radice, sostituisce con l'ultima foglia a destra, si richiama la procedura risistema() sullo heap.

La parte più onerosa dell'operazione, è la chiamata alla procedura risistema(), che esegue $O(\log n)$ confronti.

Inserimento di un nuovo elemento

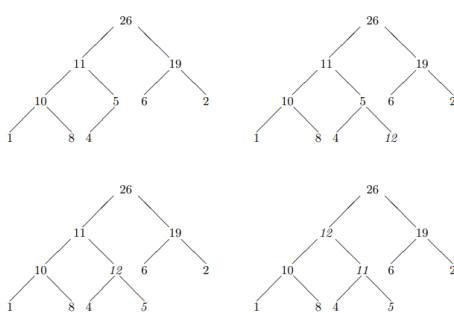
Per inserire un nuovo elemento, posso aggiungere una nuova foglia nella prima posizione libera più a sinistra dell'ultimo livello.

Se l'ultimo livello è completo, aggiungo una nuova foglia in un nuovo livello (sempre a sinistra).

Devo poi risistemare l'heap, per far sì che il nuovo elemento inserito sia correttamente posizionato.

Si "fa salire" il nuovo elemento inserito, scambiandolo via via col padre, ogni volta che il padre risulta minore di esso. A questo scopo, introduciamo una procedura risistemaDalBasso.

Esempio: inserimento di un nuovo elemento "12"



Nel caso peggiore, il nuovo nodo inserito arriverà a scambiarsi con la radice, eseguendo un numero logaritmico di passi $O(\log n)$.

Cancellazione di un elemento

Per cancellare un elemento, lo si sostituisce con la foglia più a destra dell'ultimo livello, che viene rimossa.

La nuova chiave inserita al posto dell'elemento eliminato, potrebbe però essere maggiore o minore della chiave dell'elemento eliminato.

Se la nuova chiave è minore di quella dell'elemento eliminato, allora essa potrebbe essere anche minore di una o entrambe delle chiavi dei figli. Potrebbe essere necessario "farla scendere" verso i nodi foglia.

Si richiama allora la procedura risistema, applicandola al sottoalbero di radice pari alla nuova chiave.

Se la nuova chiave invece è maggiore di quella dell'elemento eliminato, essa potrebbe essere maggiore anche del suo nodo padre.

Potrebbe allora essere necessario "far salire" la nuova chiave verso la radice.

Si richiama la procedura risistemaDalBasso.

In entrambi i casi, nel caso peggiore, potrei poter eseguire un numero logaritmico di scambi, verso l'alto o verso il basso $O(\log n)$.

Modificare la chiave di un elemento

Come per il caso precedente, mi trovo in due casi:

Se il nuovo valore è maggiore della chiave precedente, potrebbe esser necessario farla salire.

Altrimenti, potrebbe essere necessario farla scendere verso i nodi foglia

In entrambi i casi, nel caso peggiore, potrei poter eseguire un numero logaritmico di scambi, verso l'alto o verso il basso $O(\log n)$.

Code con priorità

Particolare tipo di coda FIFO, contenente elementi a cui è assegnata una chiave che ne indica la priorità.

Gli elementi nella coda, sono ordinati per valore di priorità crescente (minore è il valore di priorità associato, più alta è la priorità)

Operazioni:

- `findMin()` - Restituisce l'elemento minimo (con priorità massima) della coda, senza rimuoverlo.
- `deleteMin()` - Rimuove l'elemento minimo della coda, restituendolo
- `insert(elemento e, chiave k)` - Inserisce nella coda un elemento e, con una priorità associata k
- `delete(elemento e)` - Cancella l'elemento e
- `changeKey(elemento e, chiave d)` - Modifica la priorità dell'elemento e, con il valore d.

Implementando una coda con priorità tramite un min-heap, le operazioni assumono i seguenti costi:

- `findMin()` - $O(1)$ (l'elemento a priorità minore, è la radice)

- `deleteMin()` - $\Theta(\log n)$ (elimino radice, e risistemo heap)
- `insert()` - $\Theta(\log n)$ (risistemo dal basso)
- `delete()` - $\Theta(\log n)$ (risistemo dall'alto o dal basso)
- `changeKey()` $\Theta(\log n)$ (risistemo dall'alto o dal basso)

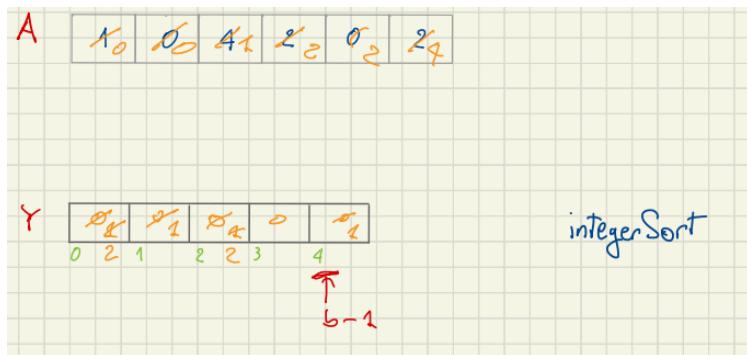
Algoritmi di ordinamento senza confronti tra chiavi

IntegerSort (Counting Sort)

Problema: Ordinare un array A di N interi, di valori compresi tra 0 e K-1 (Mi serve quindi sapere il valore massimo dell'array).

Creo un array ausiliario Y di dimensione K (K è non K-1 per tener traccia anche dello 0), e lo inizializzo a 0.

Scorro l'array A da SX a DX, per ogni elemento A_i , incremento l'elemento $Y[A_i]$ di 1.



L'array ordinato, si costruisce leggendo l'array ausiliario, che mi dice che ci sono 2 zero, un uno, 2 due, 0 tre, 1 quattro.

Array ordinato -> 001224

Pseudocodice:

```

ALGORITMO integerSort(Array A[0..n-1], Intero k)
    Array Y[0..k-1]
    FOR i ← 0 TO k-1 DO
        Y[i] ← 0

    FOR i ← 0 TO n-1 DO
        Y[A[i]] ← Y[A[i]] + 1 // Conta le occorrenze

    j ← 0
    FOR i ← 0 TO k-1 DO
        WHILE Y[i] > 0 DO
            A[j] ← i
            Y[i] ← Y[i] - 1
            j ← j + 1
    
```

Complessità temporale:

Ci sono 3 cicli for: il primo esegue k iterazioni, per inizializzare a 0 l'array aggiuntivo Y.

Il secondo, esegue n iterazioni, per scorrere l'array A, e incrementare le posizioni dell'array Y.

Il terzo, esegue k iterazioni, e al termine delle k iterazioni, sono stati scritti n valori (quelli contenuti nell'array A, ma in ordine).

Quello più oneroso, è quindi il terzo. $O(n + k)$

Dipende tutto da quanto è grande k (Numero massimo contenuto nell'array) rispetto ad n (numero di elementi nell'array).

Se $k = O(n)$ (cioè è lineare rispetto a n), allora $O(n + O(n)) = O(n)$

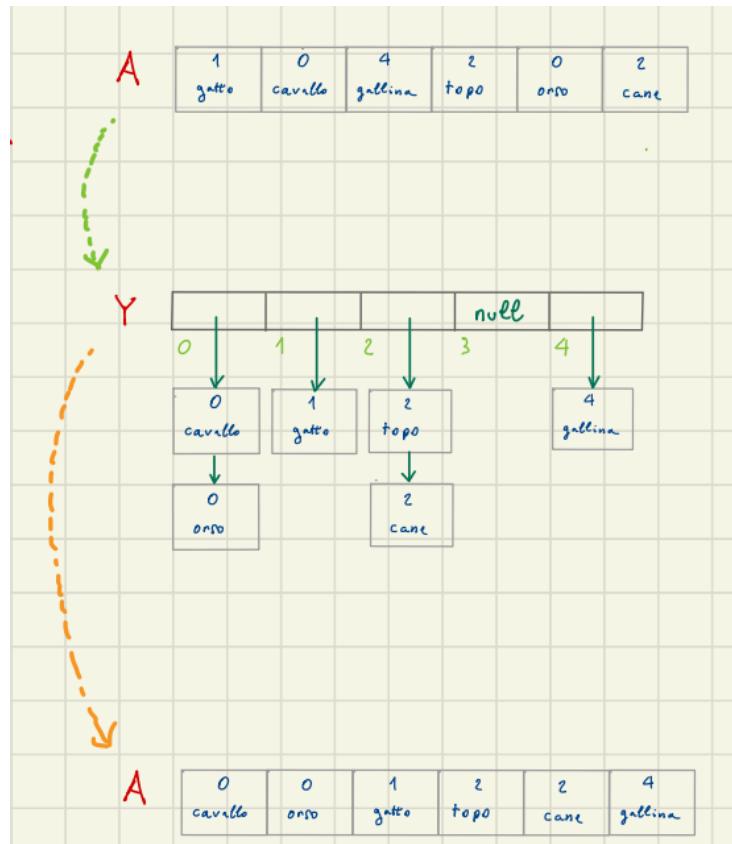
Ma se k fosse molto grande, ipotizziamo $k = O(n^2)$, allora $O(n + O(n^2)) = O(n^2)$

Utilizzando gli elementi dell'array originale come indici per accedere ad un array ausiliario, capiamo che l'algoritmo funziona solo con interi non negativi.

BucketSort

Come IntegerSort, ma permette di ordinare non solo interi, ma N record con chiavi **intero non negativo** da 0 a $B-1$

Y è un array di puntatori, ognuno testa di una coda, di elementi di chiavi pari all'indice dell'array Y



```
ALGORITMO bucketSort(Array A[0...N-1], intero b) //B = valore max in A, N = len di A
```

Sia $Y[0...B-1]$ un array di puntatori

FOR $i \leftarrow 0$ TO $B-1$ DO

$Y[i] \leftarrow$ coda vuota //inizializzo ogni puntatore a null

FOR $i \leftarrow 0$ TO $N-1$ DO //scorro A

$Y[A[i].Chiave].enqueue(A[i])$

$j \leftarrow 0$

FOR $i \leftarrow 0$ TO $B-1$ DO //scorro l'array di puntatori

 WHILE NOT $Y[i].isEmpty()$ DO //per ogni bucket, lo svuoto in A

$A[j] \leftarrow Y[i].dequeue()$

$j \leftarrow j+1$

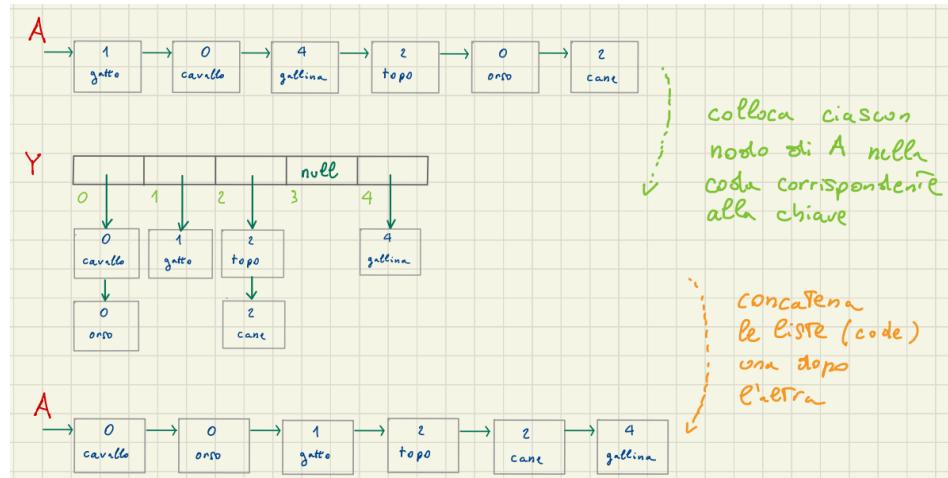
Tempo totale: $\theta(n + b)$ passi

Se $b = \theta(n)$, il tempo totale diventa $\theta(n + \theta(n)) = \theta(n)$

Se $b = \theta(n^2)$, il tempo totale diventa $\theta(n + \theta(n^2)) = \theta(n^2)$

Dipende tutto da b ! se è troppo grande, meglio usare Heapsort.

Bucketsort, può anche essere implementato per produrre una lista contenente i valori ordinati, concatenando direttamente le liste dell'array Y , una dopo l'altra.



Collego quindi "0 orso" a "1 gatto", poi "1 gatto" a "2 topo", "2 cane" a "4 gallina".

Se ogni coda è implementata con un puntatore a head e uno a tail, non devo nemmeno iterare attraverso ogni coda per collegarle, molto efficiente.

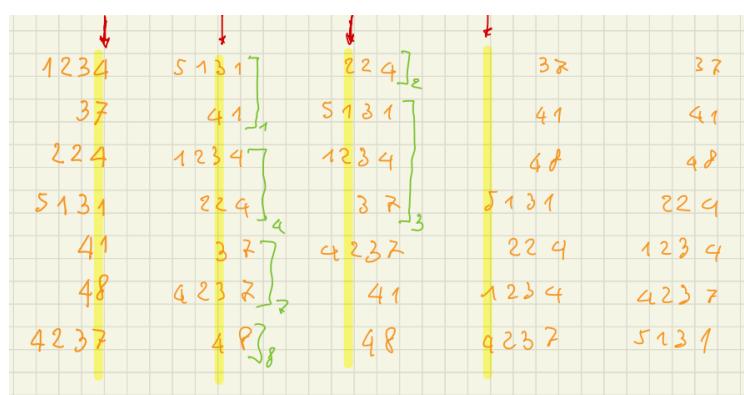
RadixSort

Dati n numeri in base b , compresi in un intervallo di valori da 0 a $k - 1$, se applicassi bucketSort, mi servirebbero k bucket.

Se k diventa molto grande, bucketSort diventa molto sconveniente.

Se però accedo alle singole cifre degli n numeri, ipotizzando un numero in base 10 da 4 cifre, ogni cifra può assumere 10 valori (0-9), cioè k diventa pari a b .

Posso allora applicare bucketSort iterativamente, utilizzando come chiavi le singole cifre degli n numeri, riducendo per ogni iterazione il numero di bucket da utilizzare a b .



Applico bucketSort ordinando secondo la prima cifra, poi per la seconda, terza, quarta, partendo da destra.

Per uniformare tutti i valori alla stessa lunghezza, considerare tutti i numeri con eventuali 0 a sinistra.

```

ALGORITMO radixSort(Array A[0...N-1]
    t<-0
    WHILE(ESISTE CHIAVE K in A T.C. (K/b^t) != 0)
        bucketSort(A,B,t)
        t<-t+1

```

BucketSort, dovrà essere implementato in modo da estrarre la cifra di posizione T per ogni chiave di A , tramite la formula $(x/b^t) \text{ MOD } b$ dove x è il numero da cui estrarre la cifra t , e b è la base in cui è espresso x .

```

PROCEDURA bucketSort(Array A[0...N-1], intero B, intero T)
//B indica il valore massimo della cifra considerata (=base),
//T indica la posizione della cifra che uso come criterio di ordinamento
    Sia Y un array di B elementi.
    FOR i=0 TO B-1
        Y[i]<-Coda vuota

    FOR i=0 TO N-1
        cifraT <- (A[i].chiave/B^T) MOD B
        Y[cifraT].enqueue(A[i])

    j<-0
    FOR i=0 TO B-1
        WHILE !Y[i].isEmpty()
            A[j] <- Y[i].dequeue()
            j<-j+1

```

Richiede di eseguire $\log_b k$ passate di BucketSort, ognuna a costo $O(n)$.

Costo: $O(n \log_b k)$

Insiemistica

Dato un insieme A , una partizione di A è un insieme di parti (sottoinsiemi di A), $a_1, a_2 \dots a_k \subseteq A$ T.C

- $a_i \neq \emptyset, \forall i = \{1 \dots k\}$
- $a_i \cap a_j = \emptyset, \text{ per } i \neq j$
- $a_1 \cup a_2 \dots \cup a_k = A$

Una partizione di A , è un insieme di sottoinsiemi di A tali per cui:

- Ogni sottoinsieme non è vuoto
- Ogni sottoinsieme non ha valori in comune con gli altri
- L'unione dei sottoinsieme, forma l'insieme A

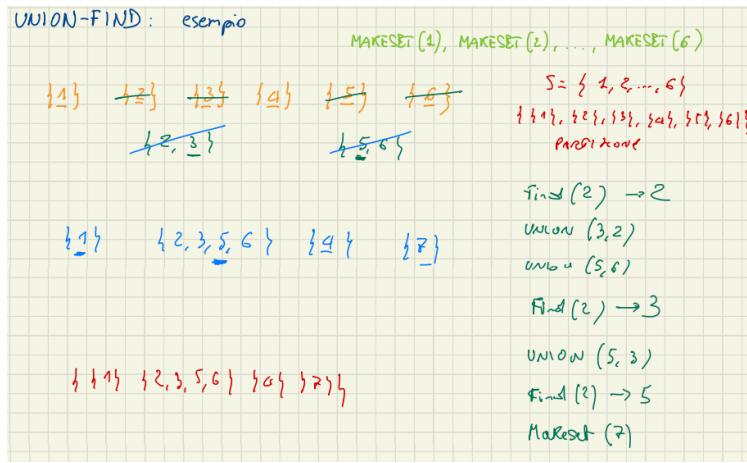
Union Find

Union-find, is a data structure that categorizes objects into different sets and lets checking out if two objects belong to the same set.

Union find, è una struttura dati che rappresenta una partizione di un insieme finito.

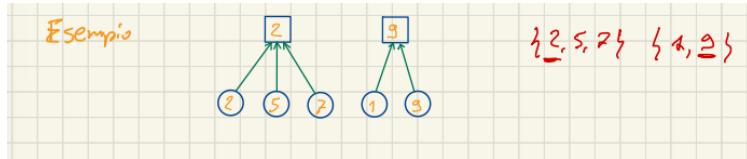
Mantiene quindi una collezione di sottoinsiemi disgiunti, fornendo le operazioni:

- UNION(A,B) → Restituisce un insieme $A \cup B$, di nome A (il nome di un sottoinsieme, è il valore sottolineato)
- FIND(x) → Restituisce il nome dell'insieme a cui appartiene x
- MAKESET(x) → Crea un nuovo insieme con un solo elemento $\{x\}$



Implementazione QuickFind

Ogni insieme è rappresentato con un albero di altezza 1 con radice pari al nome dell'insieme, in cui i nodi sono elementi dell'insieme. Il nodo radice è duplicato anche nei figli.



Se una partizione è un insieme di sottoinsiemi, e ogni insieme è un albero, una partizione è una foresta di alberi.

Operazioni:

- Makeset(e) → Crea un albero di questo tipo

Costo: $T(n) = O(1)$ (deve solo allocare un elemento)

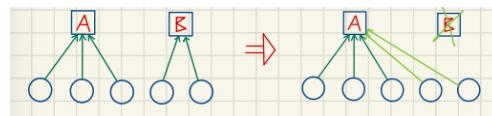


- Find(elemento e) → NomeInsieme

Basta salire dall'elemento "e" al puntatore della radice

Costo $T(n) = O(1)$

- Union(nome A, nome B) → Devo spostare tutti i puntatori di B, per far sì che puntino alla radice di A, e cancellare B.



Costo $T(n) = O(n)$, dove n è il numero di elementi di B .

Implementazione con **bilanciamento**: Nella radice di ogni albero, memorizzo anche il numero di figli.

Quando faccio Union(A,B), scelgo l'albero con meno figli, e sposto i suoi figli nell'altro albero (sposto meno figli).

In caso A fosse quello con meno figli, sposto i suoi puntatori in B, poi rinomino la radice di B in A (perchè compare per primo nella notazione Union).

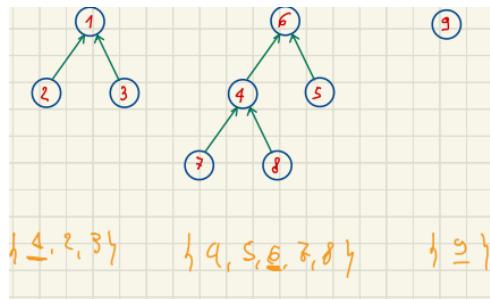
Costo best case: $O(1)$ (se sposto un solo nodo)

Costo worst case $O(n)$ (se gli alberi hanno stessa lunghezza)

Il costo ammortizzato dell'operazione risulta $O(\log(n))$, poichè si può dimostrare che se capitano operazioni di unione costose, allora quelle dopo non lo saranno.

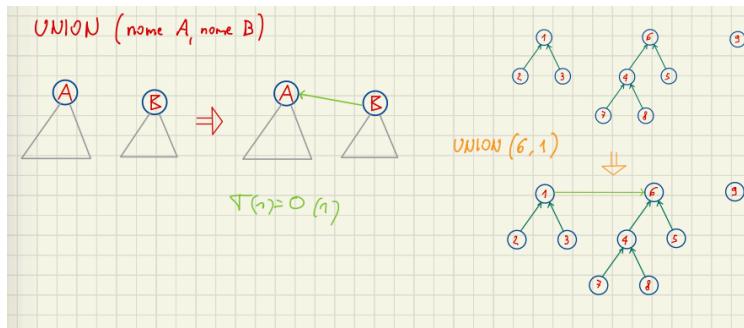
QuickUnion

Alberi di varie altezze, la radice rimane il nome dell'insieme, ma questa volta non è ridondata



L'operazione Union, è implementata tramite meccanismo dei puntatori.

Esempio: Union (A,B), vuol dire metti gli elementi di B in A, semplicemente faccio in modo che il puntatore "padre" della radice di B, punti alla radice di A.



Molto più efficiente! $T(n) = O(1)$

La Find però sarà più inefficiente, perchè essendo gli alberi di altezza variabile, io dovrei, nel caso peggiore, dover risalire tutta l'altezza dell'albero, per poter determinare la radice partendo da un nodo foglia.

Albero peggiore:

MakeSet(1), MakeSet(2), MakeSet(3) ... MakeSet(n)

Union(1,2), Union (2,3), Union(3,4) ... Union(n-1,n)

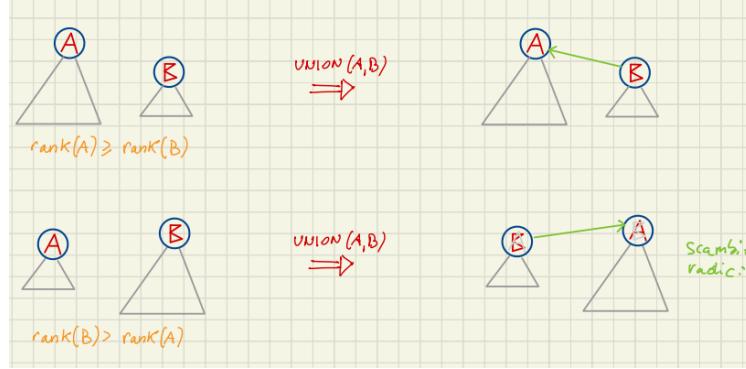
Altezza albero: $n - 1$

$T(n) = O(n - 1) = O(n)$

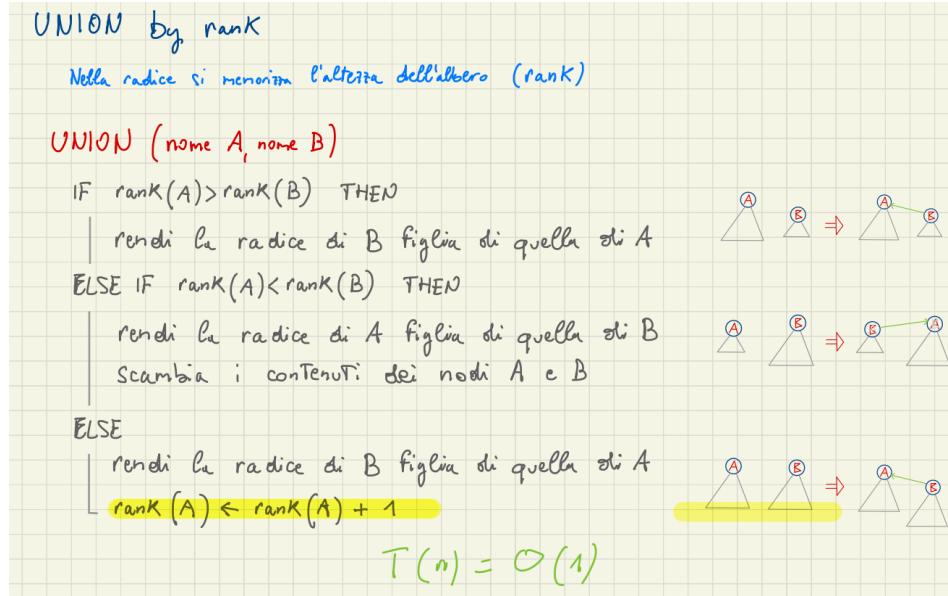
Dobbiamo tenere d'occhio l'altezza dell'albero.

Bilanciamento in altezza (Union by RANK)

Tengo d'occhio le altezze (RANK) degli alberi (che memorizzo nella radice), attacco l'albero più basso sotto la radice dell'albero più alto, ed eventualmente scambio le radici tra i due alberi.



Se $\text{rank}(b) = \text{rank}(a)$, allora si produce un albero di altezza $\text{rank}(b) + 1$ (perchè l'albero che ho attaccato sotto la radice, sarà un livello sotto rispetto alla radice).



Proviamo a calcolare il costo dell'operazione FIND, con il bilanciamento in altezza.

Nel caso peggiore, per l'operazione FIND, devo scorrere l'intera altezza dell'albero, per arrivare alla radice. Ma quanto vale $O(h)$?

Un albero quickUnion x di altezza h bilanciato in altezza, ha almeno $n \geq 2^h$ nodi.

E ciò è vero per induzione. Supponiamo che ogni albero di altezza h abbia almeno 2^h nodi. Allora un albero di altezza $h + 1$ ha almeno 2^{h+1} nodi.

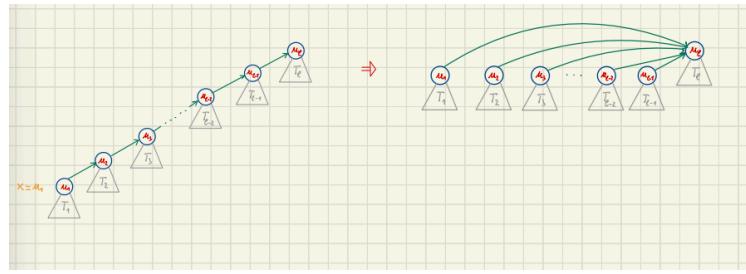
Per ottenere un albero di altezza $h + 1$, devo unire due alberi di altezza h , quindi il nuovo albero di altezza $h + 1$ avrà almeno $2^h + 2^h = 2^{h+1}$ nodi.

Da qui, Se $n \geq 2^h$, allora $h \leq \log_2 n$

Quindi nel caso peggiore, la find richiede $O(\log_2 n)$ passi.

Compressione di cammino

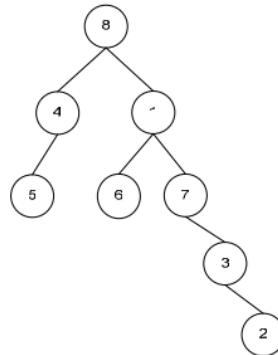
Per migliorare i tempi di esecuzione, vogliamo diminuire l'altezza degli alberi il più possibile.



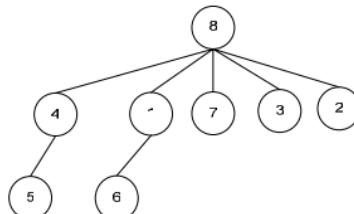
Assegnamo all'operazione di Find, un ulteriore compito.

Facendo uso dell'euristica di Path Compression, l'operazione FIND(x), ristruttura l'albero, facendo in modo di rendere il nodo padre di ogni nodo incontrato durante il percorso tra x e la radice dell'albero quickUnion, pari alla radice dell'albero.

Sia considerato il seguente albero Quick-Union:



l'applicazione dell'algoritmo di Find(2) modificato secondo l'euristica di path compression ristruttura l'albero nel seguente modo:

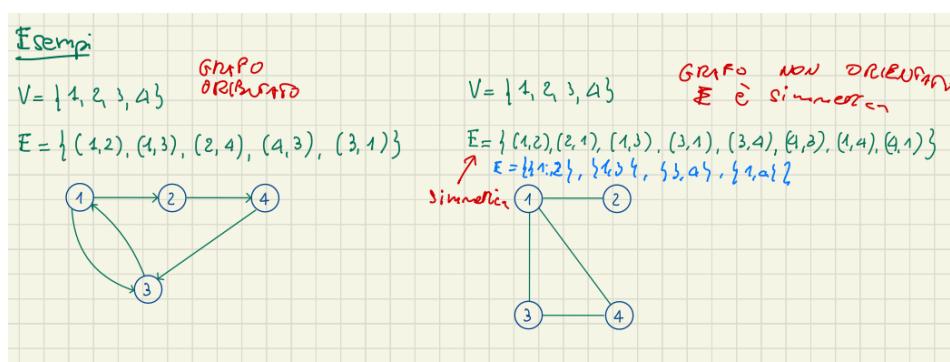


Dopo n makeset, $O(n)$ union e find, ha tempo totale $O(n \log^* n)$, quindi il costo ammortizzato di find è $O(\log^* n)$.

Grafi

Matematicamente definiti come $G = (V, E)$, dove V è l'insieme dei nodi, $E \subseteq V \times V$ l'insieme degli archi.

Se l'insieme E è simmetrico, il grafo si dice non orientato (se ogni arco può essere percorso in entrambe le direzioni).



Archi

Un elemento dell'insieme E , (x, y) , indica che esiste un arco che esce da x ed entra in y .

L'arco $(x, y) \in E$ si dice incidente sui nodi x e y .

I Vicini di un nodo, sono tutti i nodi ad esso collegati.

Grado di un nodo.

Dato un nodo v , il suo grado $\delta_{in}(v) / \delta_{out}(v)$ (in ingresso o in uscita), è il numero di archi uscenti/entranti.

$$\delta(v) = \delta_{in}(v) + \delta_{out}(v)$$

Cammino

Un cammino da x a y , è una sequenza di nodi, collegati da archi, che porti da x a y .

Formalmente: è una sequenza di nodi $v_0, v_1 \dots v_k \in V$ t.c $v_0 = x$, $v_k = y$, e $(v_{i-1}, v_i) \in E$ per $i = 1, \dots, k$.

Lunghezza del cammino = numero di archi

Un cammino è semplice, se non contiene nodi ripetuti.

y è raggiungibile da x , se esiste un cammino da x a y .

Ciclo

Cammino da x a x stesso.

Un ciclo è semplice, se l'unico nodo ripetuto è l'inizio/fine.

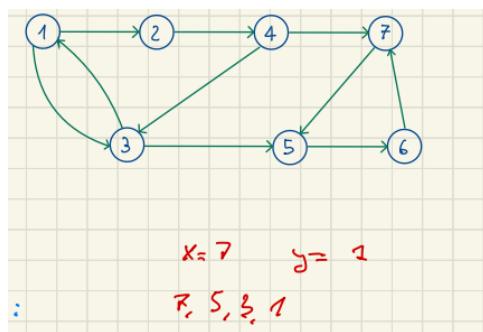
Catena

Una catena da x a y , è un cammino da x a y , che non considera l'orientamento degli archi.

Formalmente: è una sequenza di nodi $v_0, v_1 \dots v_k \in V$ t.c $v_0 = x$, $v_k = y$, $(v_{i-1}, v_i) \in E \vee (v_i, v_{i-1}) \in E$ per $i = 1, \dots, k$.

Differenza tra cammino e catena:

- **Cammino:** Segue rigorosamente l'orientamento degli archi. In un grafo orientato, un cammino da x a y richiede che tutti gli archi nella sequenza siano orientati in direzione $x \rightarrow y$.
- **Catena:** Non tiene conto dell'orientamento degli archi. Può utilizzare un arco in entrambe le direzioni, $x \rightarrow y$ o $y \rightarrow x$, purché colleghi due vertici.



Non considero orientamento archi, quindi 7,5,3,1 è valido, anche se l'arco tra 5 e 3 ha orientamento contrario.

Circuito

Catena da x a x stesso.

Le definizioni da ricordare, sono essenzialmente quelle di "cammino" e "ciclo". Quelle di "catena" e "circuito", si ricavano semplicemente ignorando l'orientamento tra archi.

Circuito Hamiltoniano

Circuito (percorso da x a x , senza considerare orientamento archi) che passa per ogni vertice del grafo, una e una sola volta.

Circuito euleriano

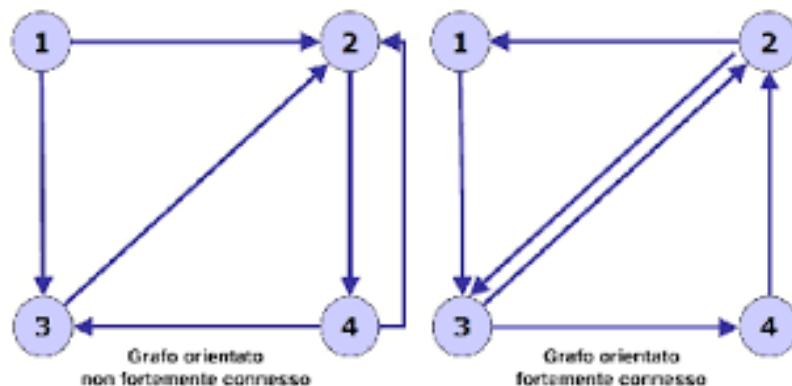
Circuito che passa per ogni arco del grafo, una e una sola volta.

Grafo connesso

Se tra ogni coppia di nodi, esiste una catena che li collega (quindi per ogni coppia di nodi, c'è un collegamento, indipendentemente dall'orientamento dell'arco)

Grafo fortemente connesso

Se tra ogni coppia di nodi, esiste un cammino che li collega (un collegamento, che è coerente con l'orientamento degli archi)

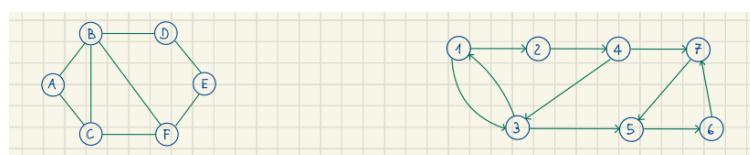


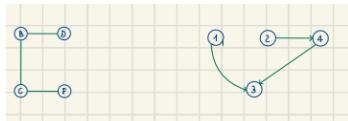
Sottografo

Un sottografo $G' = (V', E')$ di un grafo $G = (V, E)$, è ottenuto prendendo un sottoinsieme di vertici $V' \subseteq V$, e un sottoinsieme di archi $E' \subseteq E$.

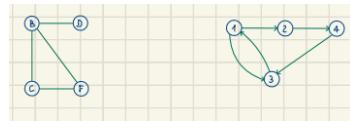
Un sottografo è indotto, se dato l'insieme di vertici V' , il grafo contiene anche tutti gli archi del grafo originale, che collegano i vertici in V' .

Esempio: Dati i due grafi





Sottografi, non sono contenuti tutti gli archi tra i nodi considerati (manca B-F per esempio)



Sottografi indotti, dato il sottoinsieme di vertici, ci sono anche tutti gli archi del grafo originale che connettono tali vertici.

Cricca

Grafo in cui ogni coppia di vertici è collegata direttamente da un arco.

In altre parole, è un grafo non orientato completo.

Una cricca in un grafo invece, è un sottografo completo.

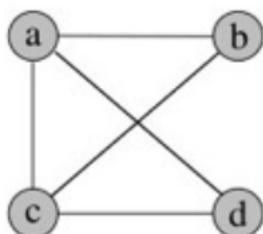
Da ogni nodo, posso raggiungere tutti gli altri con un solo arco.

Alberi

Grafo non orientato, aciclico, connesso → tra ogni coppia di vertici, esiste uno e un solo cammino.

Rappresentazione di grafi

Lista di archi

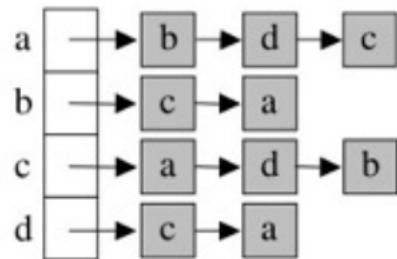
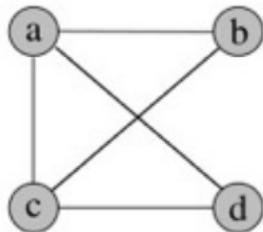


Utilizza spazio $O(n + m)$, dove n è il numero di nodi, m il numero di archi.

Se m prevale, cioè è maggiore di n , $O(m)$.

(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

Lista di adiacenza



Utilizzo un array di n vertici, in cui ognuno è in realtà testa di una lista di vertici adiacenti.

Nel caso di grafi non orientati:

Utilizza n liste, la cui somma delle lunghezze è $2m$. (Le liste contengono in totale 10 elementi = $2*m=2*5=10$, ogni arco è rappresentato due volte, non orientato)

$$\text{Spazio totale} = O(n) + O(2m) = O(n + m)$$

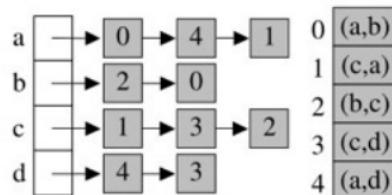
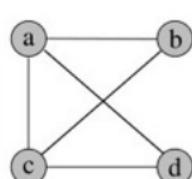
Nel caso di grafi orientati

Utilizzo n liste, la cui somma delle lunghezze è m

$$\text{Spazio totale} = O(n + m)$$

Lista di incidenza

Data la lista di archi, costruisco una lista di adiacenza, in cui per ogni nodo, associo l'indice dell'arco contenuto nella lista di archi.

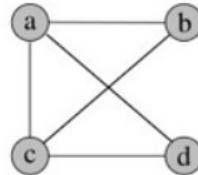


Esempio: da a , partono 3 archi, che sono rappresentati nella lista di adiacenza, dagli elementi in posizione 0 (da a a b), posizione 4 (da a a d), posizione 1 (da a a c).

$$\text{Spazio totale} = O(n + m)$$

Matrice di adiacenza

Creo una matrice M , di dimensioni $n * n$, in cui dove compare un 1, vuol dire che c'è un collegamento tra l'elemento in riga, a quello in colonna.



	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

Spazio totale = $O(n^2)$

Posso controllare la presenza di un collegamento in tempo molto breve (accesso alla riga in tempo costante, massimo n scorrimenti per iterare tra le colonne).

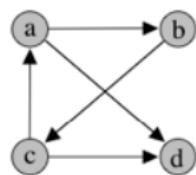
Se il grafo è non orientato, la matrice sarà simmetrica rispetto alla diagonale.

Matrice di incidenza

Righe rappresentano i vertici, colonne rappresentano gli archi

Per ogni casellina, avrò:

- -1 - Arco entrante
- 0 - Nodo non incidente (cioè l'arco nella colonna, non interessa quel nodo)
- 1 - Arco uscente



(a) Grafo orientato G

	(a,b)	(c,a)	(b,c)	(c,d)	(a,d)
a	1	-1	0	0	1
b	-1	0	1	0	0
c	0	1	-1	1	0
d	0	0	0	-1	-1

(f) Matrice di incidenza di G

Visite sui grafi

BFS

Si inizia visitando un nodo S (start), poi visito i vertici adiacenti ad S, e così via, ricorsivamente

```

ALGORITMO visitaInAmpiezza(Grafo G, vertice S) -> Albero
    C <- Coda vuota
    T <- Albero formato solo da S
    Marco s come raggiunto
    C.enqueue(s)
    WHILE NOT C.isEmpty() DO
        u<-C.dequeue() //prelevo un vertice
        FOR EACH (u,v) in E DO //guardo tutti gli archi che escono dal vertice u
            IF v non è marcato come raggiunto THEN
                aggiungi v ed (u,v) a T
                marca v come raggiunto
                C.enqueue(v)
    Return T

```

Restituisce l'albero ricoprente del grafo.

Esegue $O(n * m)$ passi.

DFS

Da un nodo di partenza S, esploro i nodi in profondità.

```

ALGORITMO visitaInProfondita(Grafo G, Vertice S) -> Albero
    Albero T formato solo da S
    visitaRicorsiva(G, S, T)
    RETURN T

PROCEDURA visitaRicorsiva(Grafo G, Vertice u, Albero T)
    marca u come raggiunto
    FOR EACH (u, v) IN E DO
        IF NOT v è marcato come visitato THEN
            aggiungi v e (u, v) a T
            visitaRicorsiva(G, v, T)

```

Grafi pesati

Associo alla rappresentazione $G = (V, E)$, una funzione $w : E \rightarrow R$, che dato un arco in ingresso, ne restituisce il peso.

Sui grafi pesati, identifichiamo alcuni problemi, cui:

- Cammini minimi
- Commesso viaggiatore
- Albero Ricoprente Minimo (Per grafi pesati non orientati)

Problemi di ottimizzazione

Problemi in cui lo scopo è minimizzare la spesa / massimizzare il guadagno.

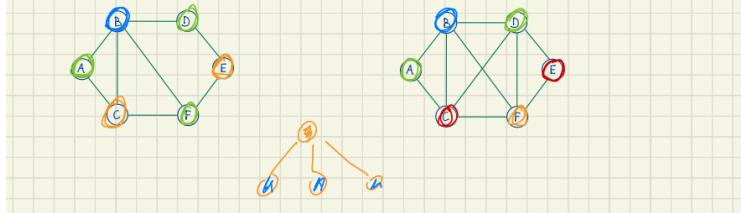
Esempio: Colorazione di grafi

PROBLEMI DI OTTIMIZZAZIONE: esempi

COLORAZIONE DI GRAFI

Istanza Grafo non orientato $G = (V, E)$

Problema Trovare il minimo numero di colori da attribuire ai vertici di G in modo che vertici adiacenti abbiano colori differenti



PROBLEMI DI OTTIMIZZAZIONE: esempi

COLORAZIONE DI GRAFI

Istanza Grafo non orientato $G = (V, E)$

Problema Trovare il minimo numero di colori da attribuire ai vertici di G in modo che vertici adiacenti abbiano colori differenti

colorazione: $c: V \rightarrow \text{Colori}$ t.c. $(x, y) \in E \Rightarrow c(x) \neq c(y)$
Vincolo

obiettivo: minimizzare $\# f_m(c)$

Soluzione ammissibile \rightarrow vertici adiacenti hanno colori diversi

Soluzione ottima \rightarrow sol. amm. con n° di colori minimo

Soluzione ammissibile, è ogni soluzione che soddisfa i vincoli del problema.

Soluzione ottima, soddisfa i vincoli del problema, e massimizza/minimizza un criterio specifico.

Esempio: Problema dello zaino

PROBLEMI DI OTTIMIZZAZIONE: esempio

ZAINO MONODIMENSIONALE

Istanza Zaino di altezza h , K contenitori c_1, c_2, \dots, c_K di altezze h_1, h_2, \dots, h_K

Problema Scegliere quali contenitori collocare nello zaino, in modo da riempirlo il più possibile

Soluzione ammissibile sottoinsieme $S \subseteq \{c_1, c_2, \dots, c_K\}$ t.c.

$$\underbrace{f(S) \leq h}_{\text{Vincolo}} \quad \text{con} \quad f(S) = \sum_{c_i \in S} h_i$$

Soluzione ottima sottoinsieme ammissibile S^* t.c.

$$f(S^*) \geq f(S) \quad \forall S \text{ ammissibile}$$

PROBLEMI DI OTTIMIZZAZIONE: esempio

ZAINO MONODIMENSIONALE

Intanza Zaino di altezza h , K contenitori c_1, c_2, \dots, c_K di altezza h_1, h_2, \dots, h_K .
Problema: scegliere quali contenitori collocare nello zaino in modo da riempire il più possibile.

Soluzione ammessa: soluzione $S \subseteq \{c_1, c_2, \dots, c_K\}$ t.c. $f(S) \leq h$ con $f(S) = \sum_{c_i \in S} h_i$.

Soluzione ottima: soluzione ammessa $S^* \subseteq \{c_1, c_2, \dots, c_K\}$ t.c. $f(S^*) \geq f(S)$ per ogni S ammesso.

Esempio:

$h = 6 \quad K = 4$
 $h_1 = 1.5 \quad h_2 = 2.5 \quad h_3 = 3 \quad h_4 = 5$

$S_1 = \{c_1, c_2\} \quad f(S_1) = 6.5$ non amm.
 $S_2 = \{c_1\} \quad f(S_2) = 5$ amm.
 $S_3 = \{c_1, c_3\} \quad f(S_3) = 4.5$ amm.
 $S_4 = \{c_2, c_3\} \quad f(S_4) = 5.5$ amm.

Tecnica risolutiva greedy

Risolve problemi di ottimizzazione.

Dato C un insieme di candidati a formulare una soluzione, bisogna trovare $S \subseteq C$, un sottoinsieme di candidati, che formino una soluzione ottima.

Strategia

- Inizialmente, $S \leftarrow \emptyset$
- Ad ogni passo
 - Prelevo da C , l'elemento x "migliore"
 - Se $S \cup \{x\}$ è una soluzione ammessa, aggiungo x a S e lo elimino dall'insieme dei candidati, altrimenti, scarto il candidato, e non lo riconSIDERO più.
- Termino quando ho esaminato tutti i candidati

```

ALGORITMO greedy(insieme C) -> Soluzione
  S <- ∅
  while NOT C.isEmpty DO
    x <- selezione(C) "migliore"
    C<-C\{x} //tolgo x dall'insieme di candidati.
    IF S U {x} è ammmissible THEN
      S<-S U {x}
  RETURN S

```

Tecnica greedy, applicata al problema dello zaino:

ZAINO MONODIMENSIONALE

$$F_1 \quad h = 20$$

$$C_1 \quad h_1 = 8$$

$$C_2 \quad h_2 = 7$$

$$C_3 \quad h_3 = 6$$

$$C_4 \quad h_4 = 3$$

$$C_5 \quad h_5 = 2$$

$$C_6 \quad h_6 = 1$$

$$\overline{\text{TOT}} \quad 20$$

OTTIMA

Istanza Zaino di altezza h , K contenitori c_1, c_2, \dots, c_n di altezza h_1, h_2, \dots, h_n .
Problema scegliere quali contenitori collocare nello zaino in modo da riempire il più possibile.

Soluzione ammessa: sottoinsieme $S \subseteq \{c_1, c_2, \dots, c_n\}$ t.c.
 $f(S) \leq h$ con $f(S) = \sum_{c \in S} h_c$

Soluzione ottima: sottoinsieme ammesso $S^* \subseteq \{c_1, c_2, \dots, c_n\}$ t.c.
 $f(S^*) \geq f(S)$ per ogni S ammesso.

$$C = \{C_1, C_2, \dots, C_K\}$$

CANDIDATI = CONTENITORI

STRATEGIA GREEDY:

ispezioniamo i
contenitori in
ordine di altezza,
dal più alto al
più basso

Ha identificato la soluzione ottima

ZAINO MONODIMENSIONALE

$$F_2 \quad h = 20$$

$$C_1 \quad h_1 = 8$$

$$C_2 \quad h_2 = 7$$

$$C_3 \quad h_3 = 6$$

$$C_4 \quad h_4 = 3$$

$$C_5 \quad h_5 = 3$$

$$C_6 \quad h_6 = 1$$

$$\overline{\text{TOT}} \quad 19$$

$$\{C_1, C_2, C_3, C_4, C_5, C_6\}$$

OTTIMA

Istanza Zaino di altezza h , K contenitori c_1, c_2, \dots, c_n di altezza h_1, h_2, \dots, h_n .
Problema scegliere quali contenitori collocare nello zaino in modo da riempire il più possibile.

Soluzione ammessa: sottoinsieme $S \subseteq \{c_1, c_2, \dots, c_n\}$ t.c.
 $f(S) \leq h$ con $f(S) = \sum_{c \in S} h_c$

Soluzione ottima: sottoinsieme ammesso $S^* \subseteq \{c_1, c_2, \dots, c_n\}$ t.c.
 $f(S^*) \geq f(S)$ per ogni S ammesso.

$$C = \{C_1, C_2, \dots, C_K\}$$

CANDIDATI = CONTENITORI

STRATEGIA GREEDY:

ispezioniamo i
contenitori in
ordine di altezza,
dal più alto al
più basso

La tecnica greedy porta a una soluzione in cui lo zaino è riempito a 19/20, ma avrei potuto fare meglio

ZAINO MONODIMENSIONALE

Fs 3 $h = 20$

c_1 $h_1 = 10$

c_2 $h_2 = 8$

c_3 $h_3 = 8$

c_4 $h_4 = 3$

c_5 $h_5 = 3$

TOT 18

$\{c_1, c_5, c_4\}$ 19

OTTIMA

Istanza Zaino di altezza h , K contenitori c_1, c_2, \dots, c_K di altezza h_1, h_2, \dots
Problema Scegliere quali contenitori collocare nello zaino in modo da
riempire il più possibile

Soluzione ammessa: soluziose $S \subseteq \{c_1, c_2, \dots, c_K\}$ t.c.
 $f(S) \leq h$ con $f(S) = \sum_{c_i \in S} h_i$

Soluzione ottima: soluziose ammessa $S^* \subseteq \{c_1, c_2, \dots, c_K\}$ t.c.
 $f(S^*) \geq f(S)$ per ogni S ammesso

$$C = \{c_1, c_2, \dots, c_K\}$$

CANDIDATI = CONTENITORI

STRATEGIA GREEDY:

ispezioniamo i
contenitori in
ordine di altezza,
dal più alto al
più basso

Ancora, la tecnica greedy non identifica la soluzione ottima.

Problema del sacco del ladro

Esempio: ZAINO CON VALORI (o Sacco del Ladro)

Istanza Sacco che può portare al massimo peso P

c_i \nearrow p_i : peso
 v_i : valore

K oggetti: c_1, c_2, \dots, c_K di peso p_1, p_2, \dots, p_K e valore v_1, v_2, \dots, v_K , rispettivamente
Problema Scegliere quali oggetti collocare nel sacco, in modo da massimizzare
il valore totale, evitando che il sacco si rompa

Obiettivo guadagnare il più possibile

Vincolo non superare la capacità del sacco

Esempio: ZAINO CON VALORI (o Sacco del Ladro)

Istanza Sacco che può portare al massimo peso P

K oggetti: c_1, c_2, \dots, c_K di peso p_1, p_2, \dots, p_K e valore v_1, v_2, \dots, v_K , rispettivamente

Problema Scegliere quali oggetti collocare nel sacco, in modo da massimizzare
il valore totale, evitando che il sacco si rompa

$$C = \{c_1, c_2, \dots, c_K\}$$

Candidati

$$S \subseteq C \rightarrow \text{val}(S) = \sum_{c_i \in S} v_i$$

$$\text{peso}(S) = \sum_{c_i \in S} p_i$$

Soluzione ammessa $S \subseteq C$ è ammessa se $\text{peso}(S) \leq P$

Soluzione ottima $S^* \subseteq C$ è ottima se (1) S^* ammessa

(2) $\forall S$ ammessa $\text{val}(S^*) \geq \text{val}(S)$

ZAINO CON VALORI: strategia greedy

ispeziona gli oggetti in ordine di valore

"selezione": scegli l'oggetto di maggiore valore

Ese p=20

	P	V
C ₁	12	100
C ₂	3	80
C ₃	6	75
C ₄	6	70
C ₅	3	30
TOT	18	210

{C₂ C₃ C₄ C₅} 255

SOL. OTTIMA

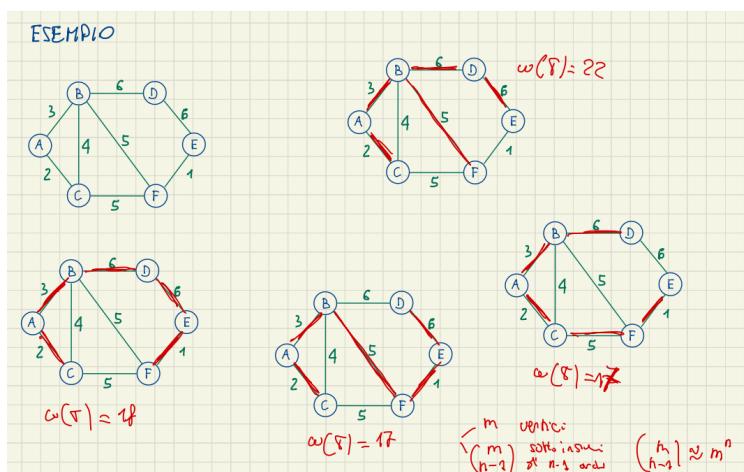
Ipotesi: il sacco supporta peso massimo=20, massimizzare il valore. Ad ogni iterazione, "prendo" l'oggetto c di valore maggiore, e controllo se la soluzione è ammissibile.

Spanning tree

Dato un grafo $G = (V, E)$, non orientato connesso, il suo albero ricoprente è un albero $G' = (V', E')$, in cui $V' = V$, e $E' \subseteq E$, cioè un albero con gli stessi nodi, ma un sottoinsieme di archi (\subseteq).

Minimum spanning tree

Dato un grafo $G = (V, E)$, non orientato connesso, e in cui ogni arco ha un peso, il suo albero ricoprente minimo $G' = (V', E_T)$ di peso minimo.



Algoritmo di Kruskal

Utilizza lista di archi.

Lavora correttamente anche in presenza di archi a costo negativo.

Utilizza Tecnica risolutiva greedy:

Ad ogni passo, seleziono l'arco di peso minimo, se l'aggiunta di questo arco, non forma cicli con gli archi già scelti, lo aggiungo alla soluzione

ALGORITMO Kruskal (grafo $G = (V, E, \omega)$) → albero

ordina E in maniera non decrescente in base al pes.

$T \leftarrow (V, \emptyset)$

FOR EACH $(x, y) \in E$ secondo l'ordine DO

 IF x e y non sono connessi in T THEN

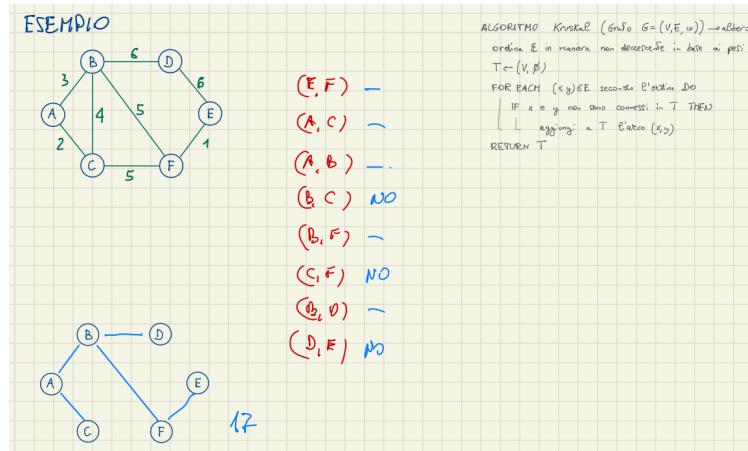
 aggiungi a T l'arco (x, y)

RETURN T

ω è la funzione peso di un arco.

T è un grafo, che all'inizio non ha nessun arco.

Per ogni arco in $(x, y) \in E$, in ordine crescente, se i nodi x e y non sono connessi (cioè se non si formano cicli), lo aggiungo a T .



L'algoritmo identifica sempre la soluzione ottima, lavora correttamente anche in presenza di archi di peso negativo.

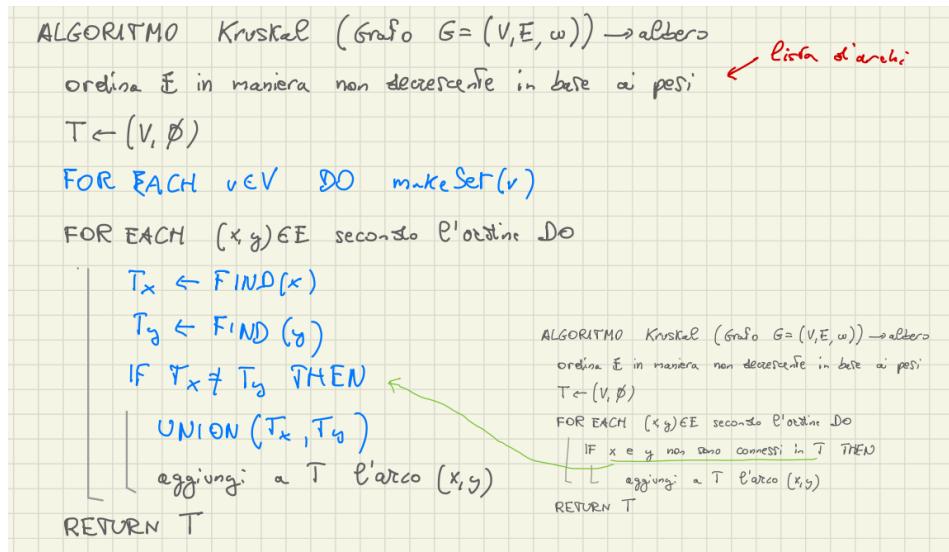
Implementazione algoritmo di Kruskal (UNION-FIND / MFSET)

Partizione di A , è un insieme di "parti" tali per cui:

- $a_i \neq \emptyset, \forall i = \{1 \dots k\}$
- $a_i \cap a_j = \emptyset$, per $i \neq j$
- $a_1 \cup a_2 \dots \cup a_k = A$

Consideriamo una partizione dell'insieme dei nodi V , in cui:

- Ogni elemento è un albero (pertanto l'unione di tutti gli elementi, cioè la partizione, forma una foresta)
- Un nodo è connesso ad un altro, se e solo se sono nello stesso elemento della partizione (se i due nodi sono in uno stesso albero). (IF FIND(x) == FIND(y))
- Aggiungere un arco, significa unire due alberi (UNION)
- La partizione inizialmente è formata da alberi contenenti un solo elemento (cioè ci sono solo i singoli nodi, che man mano verranno collegati). (MAKESET)



Implementando l'algoritmo di Kruskal utilizzando QuickUnion con bilanciamento in altezza, le cui operazioni vengono svolte in:

- MakeSet - $O(1)$
- Find - $O(\log n)$
- Union - $O(1)$

Tempo totale $T(n, m)$:

- Per l'ordinamento dell'insieme degli archi, possiamo usare HeapSort o MergeSort, con complessità temporale pari a $O(m \log m)$ (Nel caso migliore, medio e peggiore)
- Il ForEach in blu, scorre l'insieme dei nodi, e crea un albero per ogni elemento, Makeset ha costo $O(1)$, tempo ForEach pari a $O(n)$.
- Il secondo ForEach, cicla sull'insieme degli archi m volte, per ogni iterazione esegue due Find(), ed un'eventuale Union(). Al termine delle m iterazioni, avrà eseguito $O(n)$ Union().
- Costo secondo ForEach: $O(m \log n) + O(n)$

Tempo totale = $O(m \log m) + O(n) + O(m \log n) + O(n) = O(m \log n)$

Dove m è il numero degli archi, n il numero di nodi.

Utilizzando RadixSort e utilizzando la compressione di cammino, il costo scende a $O(m \log^* n)$

Algoritmo di Prim

Strategia Greedy alternativa, per identificare l'albero ricoprente minimo.

Parto da un albero T costituito da un solo vertice qualunque, ad ogni passo, espando T , aggiungendo l'arco di costo minimo, che collega un nodo appartenente a T , a un altro nodo non appartenente a T .

```

ALGORITMO Prim(G = (V, E)) -> Albero
Albero T con un unico vertice s qualunque
WHILE T ha meno di n vertici DO
    (x, y) <- vertice di peso minore con x in T e y non in T
    T.vertici <- T.vertici + {y}
    T.archi <- T.archi + {(x, y)}
RETURN T

```

Implementazione algoritmo di Prim:

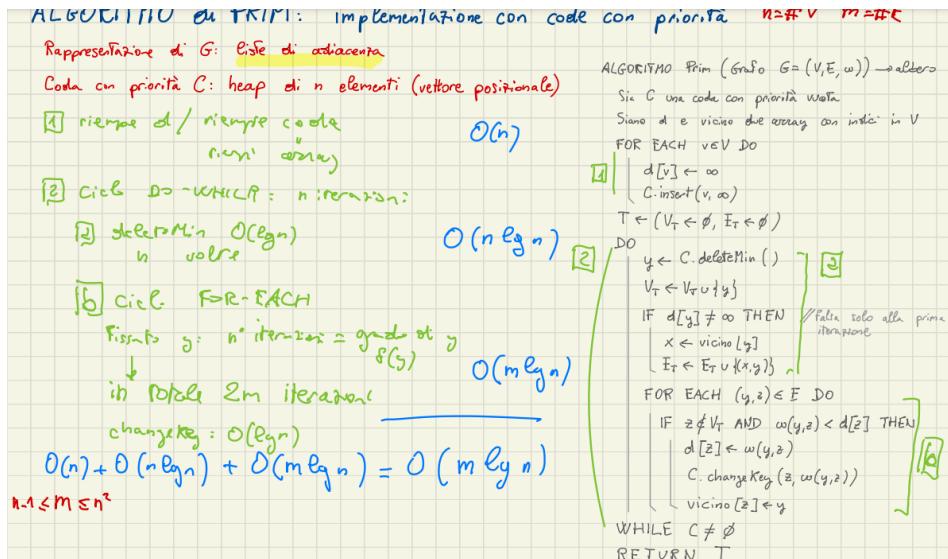
d è un array, in cui per ogni elemento $d[v]$ è memorizzato il minimo peso tra gli archi che collegano v a un vertice in T . vicino[v], mi dice qual'è il nodo (e quindi il singolo arco) che collega v a T , con peso $d[v]$

Utilizzando code con priorità:

```

ALGORITMO Prim(Grafo G=(V, E)) -> Albero
    C <- Coda con priorità vuota
    Siano d, vicino due array.
    //Riempio coda e array
    FOREACH v IN V DO
        d[v] <- infinito
        C.insert(v, infinito) //vertice-distanza
    T(V_T, E_T) <- Albero vuoto

    //Finchè ci sono vertici non inclusi nell'albero.
    DO
        y <- C.deleteMin() //elemento con distanza minima (all'inizio, sarà un nodo iniziale a caso, perchè tutti avranno distanza infinito)
        V_T <- V_T + {y} //aggiungo all'albero.
        IF d[y] != infinito DO //cioè se NON sono alla prima iterazione (ho scelto un nodo a distanza infinita)
            x <- vicino[y]
            E_T <- E_T + {(x, y)}
            FOREACH (y, z) IN E DO
                IF (NOT z IN V_T) AND (peso(y, z) < d[z]) THEN
                    d[z] <- peso(y, z)
                    vicino[z] <- y
                    C.changeKey(z, peso(y, z))
        WHILE NOT C.isEmpty()
        RETURN T
    
```



Risoluzione di problemi con strategia bottom-up.

Supponiamo di calcolare il numero di fibonacci, seguendo una strategia dividi et impera (top-down, ricorsivamente)

```

ALGORITMO Fibonacci(intero n) -> intero
    IF n=1 OR n=2 THEN RETURN 1
    ELSE RETURN Fibonacci(n-1)+Fibonacci(n-2)
    
```

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \text{ o } n = 2 \\ 2 + T(n-1) + T(n-2) & \text{se } n > 2 \end{cases}$$

Valore che tende circa a $T(n) = 1,6^n$, troppo!

Applicazione della memoizzazione, tecnica tipica della programmazione dinamica, che consiste nel salvare in memoria i valori restituiti da una funzione, in modo da averli a disposizione per un riutilizzo successivo, senza doverli ricalcolare

```
ALGORITMO Fibonacci(intero n) -> intero
    Sia Fib(1...n) un array
    Fib[1]<-1
    Fib[2]<-1
    FOR i<-3 to n DO
        Fib[i] <- Fib[i-1]+Fib[i-2]
    RETURN Fib[n]
```

Un programma banale per calcolare i numeri di Fibonacci è

```
fib(n) {
    if n is 1 or 2, return 1;
    return fib(n-1) + fib(n-2);
}
```

Poiché `fib()` è più volte ricalcolato sullo stesso argomento, il tempo di esecuzione per questo programma è $\Omega(1.6^n)$. Se, invece, memoizziamo (salviamo) il valore di `fib(n)` la prima volta che viene calcolato il tempo di esecuzione scende a $\Theta(n)$.

```
alloca array per memo, settando tutti i valori a zero;
inizializza memo[1] e memo[2] a 1;

fib(n) {
    if memo[n] diverso da 0, return memo[n];
    memo[n] = fib(n-1) + fib(n-2);
    return memo[n];
}
```

$T(n) = O(n)$, molto meglio!

Programmazione dinamica

Tecnica che permette di risolvere un problema, partendo dalla risoluzione di sottoproblemi più semplici.

- Si individuano i sottoproblemi del problema dato, i cui risultati verranno salvati in una tabella
- Si definiscono i valori iniziali della tabella (casi base)
- Si risolvono i sottoproblemi a partire dai più semplici
 - Sottoproblemi base - soluzione immediata, memorizzata direttamente in tabella
 - Sottoproblemi avanzati - risolti consultando la tabella, e memorizzando il risultato finale.

La soluzione finale, sarà ricavata dalle soluzioni dei sottoproblemi.

Sottovettore di somma massima

Dato un vettore V di interi in Z, trovare un sottovettore di somma massima:

Esempio: Dato 1, 2, -4, 8, -2, 3, -1.

Possiamo adottare una soluzione "banale", con due indici, uno di inizio e uno di fine, e due for innestati.

```

ALGORITMO sottovettoreMax (Array V[1..n]) -> (inizio, fine)
    max <- V[1], inizio <- 1, fine <- 1
    FOR i <- 1 TO n DO
        FOR f <- i TO n DO
            somma <- somma di V[i..f]
            IF somma > max THEN
                max <- somma
                inizio <- i
                fine <- f
    RETURN (inizio, fine)

    1 2 -4 8 -2 3 -1
    ↑   ↑   ↑   ↑
    i   f   i   f
    6
    somma <- 0
    FOR k <i TO f DO
        somma <- somma + V[k]

    Temp O(n^3)

```

L'istruzione "somma di tutti gli elementi da i a f ", è implementata con un altro loop, quindi la complessità è $O(n^3)$.

Potrei ridurre questa complessità, sommando all'avanzare di f , il nuovo valore $V[f]$, ma comunque la complessità non scenderebbe sotto $O(n^2)$

Soluzione avanzata:

Trovare il sottovettore di somma massima.

Sottoproblema "vincolato": Trovare il sottovettore di somma massima, che termina in posizione i (scomposizione del problema)

Alla fine, sceglierò tra un insieme di sottovettori $P(1), P(2) \dots P(n)$, dei sottovettori di somma massima che terminano in i , quello a valore maggiore.

Soluzione:

```

ALGORITMO sottovettoreMax(Array V[1...n]) ->
intero,intero
    Sia S[1..n] un vettore
    S[1]<-V[1] //soluzione "base"

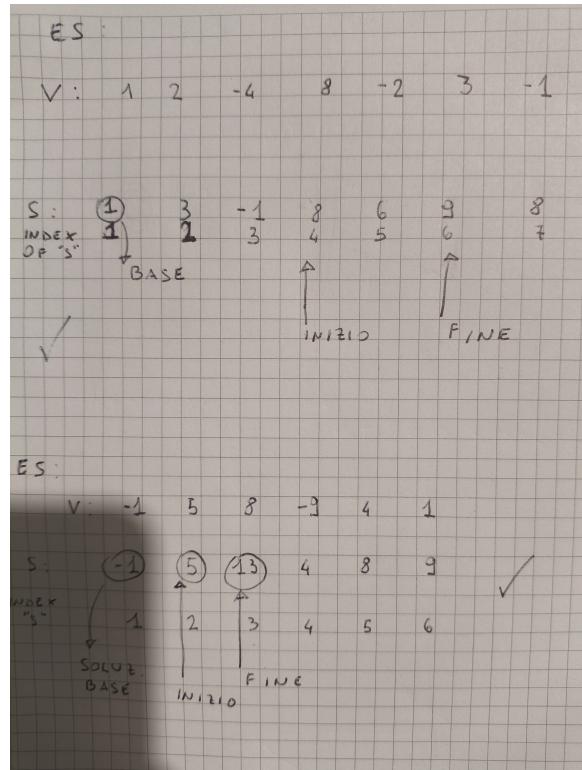
    //Costruisco S
    FOR i<-2 TO n DO
        IF S[i-1]>=0
            S[i]<-S[i-1]+V[i]
        ELSE
            S[i]<-V[i]

    //Trovo il valore massimo in S, mi segno
    la sua posizione
        max <- S[1]
        fine <- 1

        FOR i<-2 TO n DO
            IF S[i]>max THEN
                max<-S[i]
                fine <- i

    //dall'indice di fine della sequenza,
    //torno indietro finché non trovo l'inizio
    //o della sequenza.
        inizio <- fine
        WHILE S[inizio]!=V[inizio] DO
            inizio <- inizio -1

```



```
RETURN(inizio,fine)
```

Cammini di valore minimo su matrici

Data una matrice M, NxN, trovare un cammino di valore minimo tra la prima e l'ultima colonna.

Da ogni casella della matrice, posso spostarmi:

- In diagonale in alto
- In diagonale in basso
- Sulla stessa riga

Inizializzo una nuova matrice C, in cui ogni $C[i,j]$ rappresenta il valore del cammino minimo tra una casella della colonna uno, e la casella di posizione $[i,j]$.

La prima colonna della matrice C, sarà uguale a quella della matrice M.

Per ogni colonna successiva alla prima, il valore di $C[i,j]$ è scelto come $M[i,j] + \min\{C[i-1,j-1], C[i,j-1], C[i+1,j-1]\}$.

$$\begin{aligned}
 & \text{1^a colonna} \quad j=1 \\
 & C[1,1] = M[1,1] \quad i=1..n \\
 & \text{j-esima colonna} \\
 & C[i,j] = M[i,j] + \min_{j>1} \{ C[i-1,j-1], C[i,j-1], C[i+1,j-1] \} \\
 & \min \{ C[i,n] \mid i=1..n \}
 \end{aligned}$$

$M = \begin{pmatrix} 1 & 3 & 8 & 5 \\ 4 & 2 & 7 & 6 \\ 2 & 1 & 5 & 1 \\ 3 & 8 & 2 & 3 \end{pmatrix}$
input
 $C = \begin{pmatrix} 1 & 4 & 11 & 15 \\ 4 & 3 & 10 & 14 \\ 2 & 3 & 8 & 6 \\ 9 & 10 & 5 & 8 \end{pmatrix}$

$$\begin{aligned}
 & C = \begin{pmatrix} 1 & 4 & 11 & 15 \\ 4 & 3 & 10 & 14 \\ 2 & 3 & 8 & 6 \\ 9 & 10 & 5 & 8 \end{pmatrix} \\
 & M = \begin{pmatrix} 1 & 3 & 8 & 5 \\ 4 & 2 & 7 & 6 \\ 2 & 1 & 5 & 1 \\ 3 & 8 & 2 & 3 \end{pmatrix}
 \end{aligned}$$

input

```

ALGORITMO camminoMinimo (Matrice M (n*n)) ->
intero
Sia C una matrice n*n

//Prima colonna di C = Prima colonna di M
FOR i<-1 TO n
    C[i,1] = M[i,1]

//Inizializzo matrice
FOR j<-2 TO n
    FOR i<-2 to n
        min<-C[i,j-1]
        IF i>1 AND C[i-1,j-1] < min
            min = C[i-1,j-1]
        IF i<n AND C[i+1,j-1] < min
            min = C[i+1,j-1]
        C[i, j] = M[i, j]+min
    //Trovo valore del cammino minimo

    min<-C[1,n]
    FOR i<-2 TO n

```

```

IF C[i,n] < min
    min <- C[i,n]

```

```
RETURN min
```

Cammini minimi

Dato un grafo orientato $G = (V, E)$, con associata una funzione peso $\omega : E \rightarrow R$, definiamo un cammino da v_0 a v_k come $\pi = \langle v_0, v_1 \dots v_k \rangle$.

$\omega(\pi)$ è il peso del cammino, cioè la somma dei pesi degli archi che collegano v_o a v_k .

Se π è un cammino minimo da x a y , allora $\omega(\pi) \leq \omega(\pi')$, per ogni π' che porta da x a y .

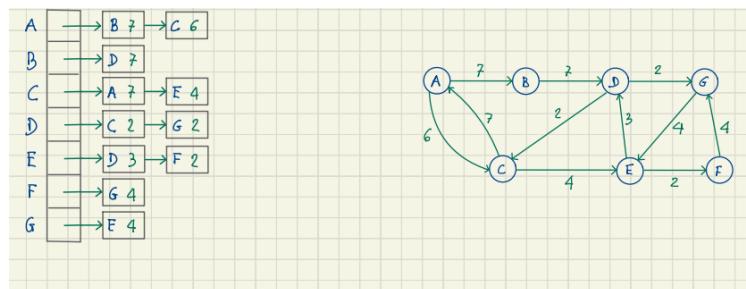
Proprietà:

- Se π è un cammino minimo da x a z , passando per y , allora contiene anche i percorsi minimi da x a y , e da y a z .

Ogni sottocammino di π è esso stesso un cammino minimo (Principio di ottimalità, una soluzione ottima è data dalle combinazioni delle sottosoluzioni ottime dei problemi più piccoli)
- Se tutti i pesi sono positivi, ogni cammino minimo è semplice (non ci sono nodi ripetuti)
- Se ci sono pesi negativi, e non ci sono cicli negativi, allora tra ogni coppia di vertici esiste sempre un cammino minimo semplice. Trovare un cammino minimo semplice in grafi con cicli negativi, è un problema NP-completo.

Rappresentazione di grafi pesati:

Lista di adiacenza con pesi



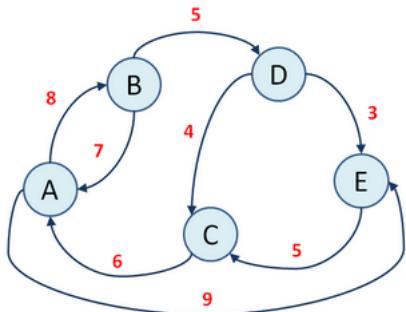
Matrice dei pesi

MATRICE DEI PESI							
	A	B	C	D	E	F	G
A	∞	7	6	∞	∞	∞	∞
B	∞	∞	∞	7	∞	∞	∞
C	7	∞	∞	∞	4	∞	∞
D	∞	∞	2	∞	∞	∞	2
E	∞	∞	∞	3	∞	2	∞
F	∞	∞	∞	∞	∞	4	∞
G	∞	∞	∞	∞	4	∞	∞

Ogni elemento (i, j) della matrice rappresenta il peso dell'arco che va dal nodo i al nodo j .

Se $i=j$, oppure non esiste un arco tra i e j , l'elemento (i, j) della matrice viene impostato a 0, oppure ad un valore molto grande, esempio MAX_INT (infinito).

L'algoritmo di Floyd Warshall, utilizza una variante della matrice dei pesi, che differisce nodi a distanza 0 ($i=j$), e nodi a distanza infinita, per cui non esiste (oppure non è ancora stato scoperto) un percorso.



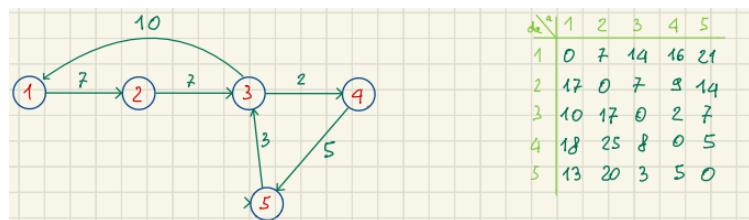
	A	B	C	D	E
A	0	8	$+\infty$	$+\infty$	9
B	7	0	$+\infty$	5	$+\infty$
C	6	$+\infty$	0	$+\infty$	$+\infty$
D	$+\infty$	$+\infty$	4	0	3
E	$+\infty$	$+\infty$	5	$+\infty$	0

Cammini minimi tra ogni coppia di vertici (Algoritmo di Floyd e Warshall)

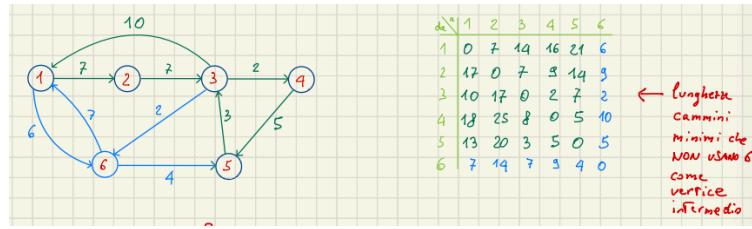
Se posso scegliere come rappresentare, utilizzare matrice dei pesi.

Utilizza tecnica di programmazione dinamica.

L'algoritmo funziona correttamente sia con grafi orientati che non: in caso di grafo non orientato, la matrice in ingresso sarà simmetrica.



Dato un grafo, e una tabella che riassume i costi dei cammini dai nodi sulle righe, a quelli sulle colonne, come cambiano i pesi dei cammini, se aggiungessi un nuovo "pezzo di grafo"? (aggiungo un nodo e degli archi, immaginiamola come una rete stradale: se viene costruita una nuova strada, le lunghezze dei cammini migliorano?)

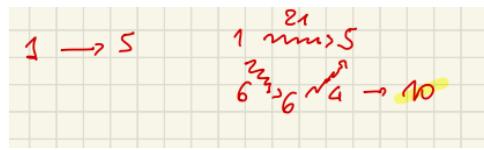


Aggiungo il nodo "6", aggiorno la tabella aggiungendo i cammini da e verso 6.

Ora ci chiediamo, passando per il nodo "6", riusciamo a ridurre i costi di altri cammini?

Per ogni coppia di nodi, calcoliamo il percorso passando da "6" come nodo intermedio, se è minore del costo attualmente nella tabella, aggiorno.

Esempio: da 1 a 5 costa 21, passando per 6, riesco a fare di meglio?



Da 1 a 6, costa 6.

Da 6 a 5, costa 4.

Il costo da 1 a 5 passa da 21 a 10! Aggiorno nella tabella.

$d_{ij}^{(k)}$	1	2	3	4	5	6
1	0	7	14	16	21	6
2	17	0	7	9	14	3
3	10	17	0	2	7	2
4	18	25	8	0	5	10
5	13	20	3	5	0	5
6	7	14	7	9	4	0

← lunghezza cammini minimi che NON usano 6 come vertice intermedio

$d_{ij}^{(k)}$	1	2	3	4	5	6
1	0	7	13	15	10	6
2	16	0	7	9	13	3
3	9	16	0	2	6	2
4	17	24	8	0	5	10
5	12	19	3	5	0	5
6	7	14	7	9	4	0

← usando anche 6 come vertice intermedio

Formalizziamo il problema:

Dato un insieme di nodi $V = \{v_1, v_2, \dots, v_n\}$, e sia d_{ij} il peso del cammino da v_i a v_j , vogliamo determinare d_{ij} per ogni coppia di nodi in V .

Consideriamo il problema vincolato ad un parametro k , $d_{ij}^{(k)}$, che indica il peso del cammino minimo da v_i a v_j , passando esclusivamente per nodi intermedi di indice $\leq k$.

Esempio

$$M = \begin{pmatrix} \infty & 5 & \infty & \infty \\ \infty & \infty & 7 & 4 \\ \infty & 1 & \infty & \infty \\ 2 & \infty & 2 & \infty \end{pmatrix}$$

matrice dei pesi

$$D_0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ \infty & 0 & 7 & 4 \\ \infty & 1 & 0 & \infty \\ 2 & \infty & 2 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & 42 & 9 \\ \infty & 0 & 7 & 4 \\ \infty & 1 & 0 & 5 \\ 2 & 7 & 2 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 5 & 42 & 9 \\ \infty & 0 & 7 & 4 \\ \infty & 1 & 0 & 5 \\ 2 & 3 & 2 & 0 \end{pmatrix}$$

$$d_{ij}^{(k)} = \min \left\{ \begin{array}{l} w(v_i, v_j) \\ d_{ij}^{(k-1)} \\ d_{ij}^{(k-1)} + w(v_{k-1}, v_j) \end{array} \right\}$$

Allora:

- Se $k = 0$, non si può passare per nodi intermedi, quindi è un cammino diretto.
 - $d_{ij}^{(0)}$ vale 0, se $v_i = v_j$.
 - Vale il peso dell'arco tra i due, se esiste un arco che li collega.
 - Vale ∞ , se non esiste un arco che li collega.
- Se $k > 0$, $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$, cioè il minimo tra la strada "vecchia" da i a j , prima dell'inserimento del nuovo nodo k , e la nuova strada passando per k (il cui peso è $d_{ik} + d_{kj}$).

```

ALGORITIMO FloydWarshall(Grafo G=(V, E, w)) -> Matrice
Matrice D[1..n, 1..n]

// Riempio matrice di programmazione dinamica D
FOR i <- 1 TO n DO
  FOR j <- 1 TO n DO
    IF i = j THEN //arco da un nodo a se stesso
      D[i, j] <- 0
    ELSE IF (V_i, V_j) in E THEN //arco diretto
      D[i, j] <- w(i, j)
    ELSE
      D[i, j] <- infinity

//Riempio matrice di programmazione dinamica, per valori di k diversi da 0
//(caso base, in cui considero solo archi diretti e cammini da x a x).
FOR k <- 1 TO n DO //considero nodi intermedi, uno ad uno
  FOR i <- 1 TO n DO
    FOR j <- 1 To n DO
      IF D[i, k] + D[k, j] < D[i, j] THEN
        D[i, j] <- D[i, k] + D[k, j]

RETURN D

```

L'algoritmo trova la soluzione ottima anche in presenza di archi a costo negativo (non devono esserci cicli negativi, in tal caso il problema è mal posto, io potrei continuare all'infinito a percorrere un ciclo negativo, e minimizzare il costo del cammino).

Si nota che la seguente implementazione dell'algoritmo, permette solo di conoscere il costo dei cammini minimi tra un vertice di indice i sulle righe, e un vertice di indice j sulle colonne. Se volessi anche ricostruire il percorso tra v_i e v_j , si introduce una matrice ausiliaria P , i cui valori vengono riempiti durante l'esecuzione dell'algoritmo in questo modo:

$P[i, j] = 0$ se $i = j$ oppure non esiste un cammino tra i e j .

$P[i, j] = \text{Indice del penultimo vertice sul cammino } v_i v_j \text{ altrimenti.}$

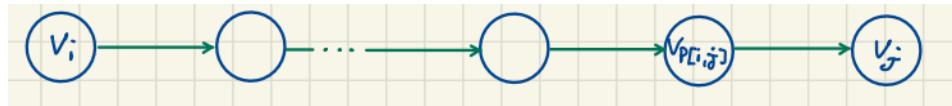
Complessità temporale: $O(n^3)$

Complessità spaziale: $O(n^2)$ (Matrice di programmazione dinamica $n * n$).

Cammino minimo tra un vertice e un altro

Modifichiamo l'algoritmo di Floyd & Warshall, per poter ricavare anche i cammini tra un vertice e un'altro (con la corrente implementazione, posso solo sapere i valori di tali cammini)

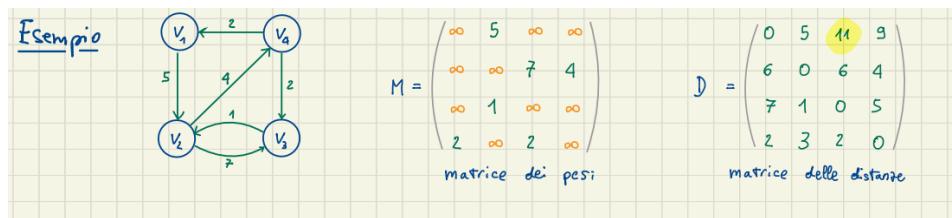
Introduciamo una matrice P , in cui $P[i, j]$ è l'indice del penultimo vertice sul cammino tra v_i e v_j .



Se $i = j$ o non ci sono cammini tra v_i e v_j , $P[i, j]$ vale 0.

$$P = \begin{pmatrix} 0 & 1 & 4 & 2 \\ 4 & 0 & 4 & 2 \\ 4 & 3 & 0 & 2 \\ 4 & 3 & 4 & 0 \end{pmatrix}$$

Osservando i valori della matrice P , posso ricostruire il percorso tra un qualsiasi vertice e un'altro.



E Data la matrice P :

Ricostruiamo il percorso tra v_1 e v_3 , di costo 11.

Nella matrice delle distanze, l'incrocio tra v_1 e v_3 , è la prima riga della terza colonna. La cella corrispondente nella matrice P , indica che il penultimo vertice tra v_1 e v_3 , è il vertice 4.

Il penultimo vertice tra v_1 e v_4 , è il vertice 2.

Il penultimo vertice tra v_1 e v_2 , è v_1 stesso.

Percorso: $v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3$.

L'algoritmo è così modificato

ALGORITMO Floyd-Warshall (Grafo G) → Matrice

Siano $D[1..n, 1..n]$ una matrice, Sia $P[1..n, 1..n]$ una matrice

```

FOR i ← 1 TO n DO
    FOR j ← 1 TO n DO
         $d_{ij}^{(0)} = \begin{cases} w(v_i, v_j) & \text{se } (v_i, v_j) \in E \text{ e } v_i \neq v_j \\ 0 & \text{se } v_i = v_j \\ \infty & \text{altrimenti.} \end{cases}$ 
         $P[i, j] \leftarrow 0$ 
        IF  $i = j$  THEN  $D[i, j] \leftarrow 0$ 
        ELSE IF  $(v_i, v_j) \in E$  THEN  $D[i, j] \leftarrow w(i, j)$ 
         $P[i, j] \leftarrow i$ 
        ELSE  $D[i, j] \leftarrow \infty$ 
    
```

```

FOR K ← 1 TO n DO
    FOR i ← 1 TO n DO
        FOR j ← 1 TO n DO
             $d_{ij}^{(k)} = \min \{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \}$ 
            IF  $D[i, k] + D[k, j] < D[i, j]$  THEN
                 $D[i, j] \leftarrow D[i, k] + D[k, j]$ 
                 $P[i, j] \leftarrow P[k, j]$ 
    
```

RETURN D

Cammini minimi tra un vertice e tutti gli altri

Algoritmo di Bellman & Ford:

Utilizza **Lista di archi (con pesi)**

Input: Grafo $G = (V, E)$ privo di cicli negativi, una funzione peso $\omega : E \rightarrow R$, un vertice di partenza $s \in V$.

Chiamiamo d il vettore delle distanze **provvisorie**, in cui $d[v]$ indica il peso minimo del cammino da s a v fin'ora trovato.

(Essenzialmente, una riga della matrice di Floyd & Warshall)

Inizialmente, $d[v]$ è uguale a 0, se $s = v$, ∞ altrimenti.

ALGORITMO Bellman & Ford (Grafo G , vertice s) \rightarrow Vettore

Sia $d[V]$ un vettore con indici in V

$d[s] \leftarrow 0$

FOR EACH $v \in V - \{s\}$ DO $d[v] \leftarrow \infty$

FOR $k \leftarrow 1$ TO $n-1$ DO

FOR EACH $(u, v) \in E$ DO

IF $d[u] + w(u, v) < d[v]$ THEN

$d[v] \leftarrow d[u] + w(u, v)$

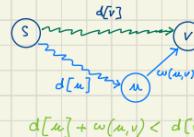
RETURN d

$d[v] = \begin{cases} 0 & \text{se } v=s \\ \infty & \text{altrimenti} \end{cases}$

Inizialmente:

$$d[v] = \begin{cases} 0 & \text{se } v=s \\ \infty & \text{altrimenti} \end{cases}$$

Aaggiornamento:



Il for da $k=1$ a $n-1$, scorre tutti i vertici v ,

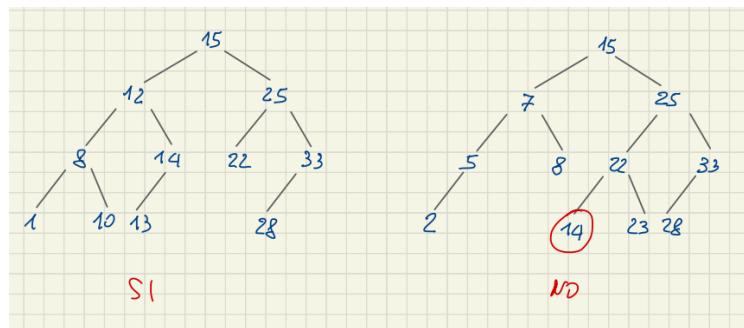
Per ogni vertice v , se il percorso da s a v passando per un nodo ad esso collegato, è minore del percorso attualmente trovato da s a v , lo aggiorno.

Tempo $T(n) = O(n * m)$.

Algoritmo di Dijkstra

Alberi binari di ricerca (ABR)

Un albero binario di ricerca, è un albero binario in cui per ogni nodo n (contenente una chiave), il sottoalbero di sinistra, contiene solo nodi la cui chiave è $<$ $n.key$, il sottoalbero destro, contiene solo nodi la cui chiave è $\geq n.key$.



L'albero a destra, non è un ABR ($14 < 15$)

Struttura di un nodo ABR:

```
type node struct{
    key int
    others..
    sx *node //accesso al sottoalbero sinistro
    dx *node //accesso al sottoalbero destro
}
```

Visitando in ordine simmetrico (In-Order) un albero binario di ricerca, ottengo gli elementi contenuti in esso in ordine crescente.

Trovare il nodo con chiave max/min

Dalla radice, voglio spingermi il più a destra/sinistra possibile, per identificare il nodo di chiave maggiore/minore

```
FUNZIONE massimo (AlberoDiRicerca radice) -> Nodo
  IF radice=null THEN
    return NULL
  ELSE
    nodoDaIterare <- radice
    WHILE nodoDaIterare.dx != null DO
      nodoDaIterare = nodoDaIterare.dx
    return nodoDaIterare
```

```
FUNZIONE minimo (AlberoDiRicerca radice) -> Nodo
  IF radice=null THEN
    return NULL
  ELSE
    nodoDaIterare <- radice
    WHILE nodoDaIterare.sx!= null DO
      nodoDaIterare = nodoDaIterare.sx
    return nodoDaIterare
```

Ricerca dicotomica ricorsiva

```
FUNZIONE ricerca (AlberoDiRicerca radice, valore intero) -> Nodo
  IF radice=null THEN
    return NULL
  ELSE IF valore=radice.key THEN
    return radice
  ELSE IF valore<radice.key THEN
    ricerca(radice.sx,valore)
  ELSE
    ricerca(radice.dx,valore)
```

Inserimento ricorsivo

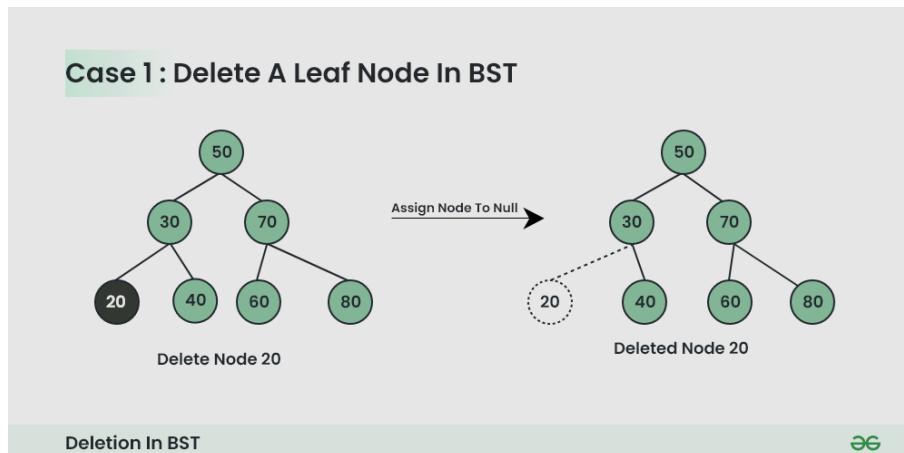
```
FUNZIONE inserisci(AlberoRicerca r, elemento d) -> AlberoRicerca
  k<-d.key
  IF r=null THEN //se l'albero è vuoto, d è la nuova radice
    r <- riferimento ad un nuovo nodo.
    r.key <- k
    r.sx <- null
    r.dx <- null
  ELSE IF k<r.key THEN
    r.sx <- inserisci(r.sx,d)
  ELSE
    r.dx <- inserisci(r.dx,d)
  RETURN r
```

Cancellazione di un nodo

Tre casi:

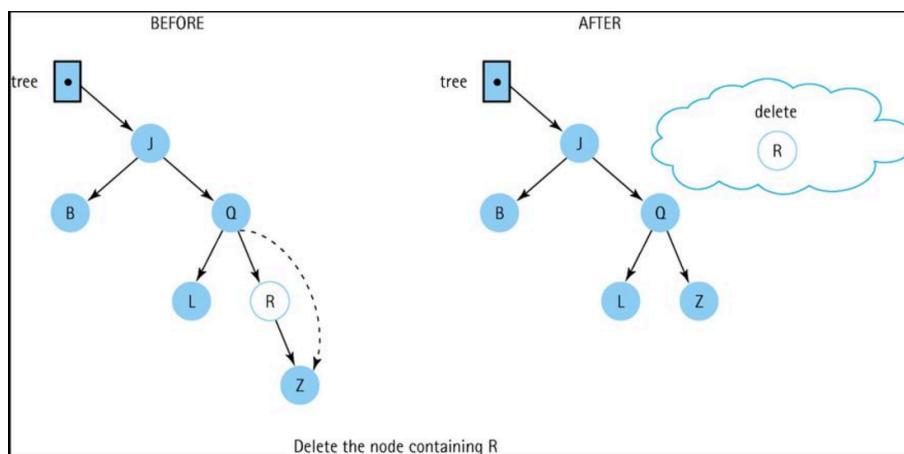
- Il nodo da cancellare, è un nodo foglia.

Aggiorno il puntatore del padre, per puntare a null.



- Il nodo da cancellare, ha un figlio.

Aggiorno il puntatore del padre, per farlo puntare all'unico figlio del nodo eliminato.



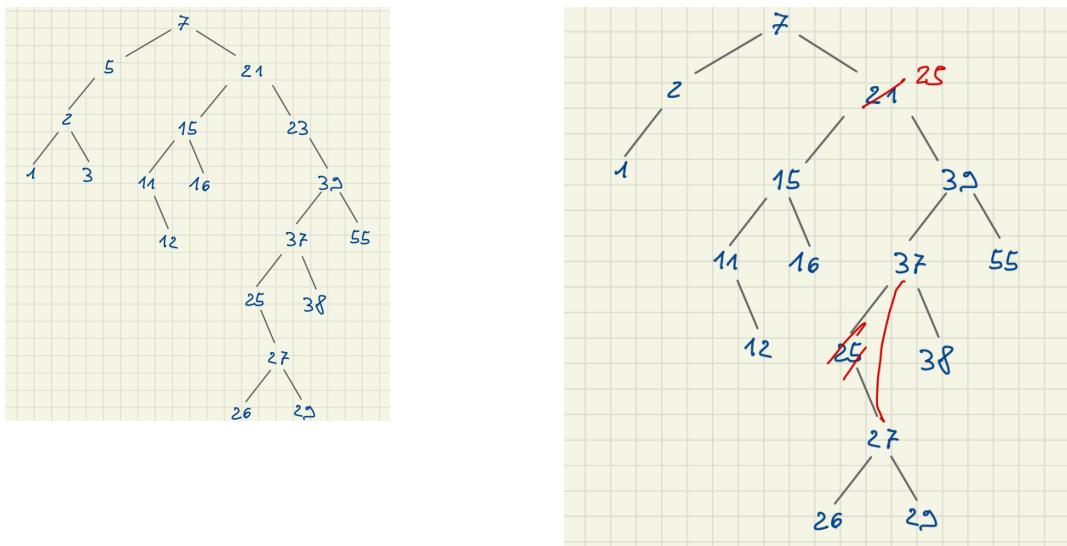
- Il nodo da cancellare, ha entrambi i figli

Sostituisco il contenuto del nodo da cancellare, con il contenuto del nodo di chiave minima, nel sottoalbero destro del nodo da cancellare.

Elimino tale nodo, seguendo il caso 1 o 2.

(Essendo il nodo più a sinistra del sottoalbero di destra, esso avrà o un figlio a destra, oppure 0 figli).

Esempio: eliminazione del nodo 21:

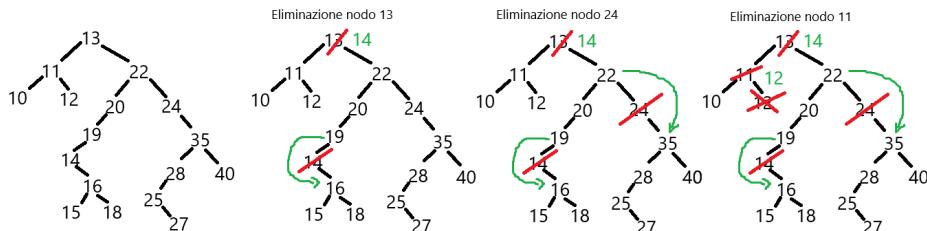


Esercizio: Cancellazione nodi

Esercizio:

Disegnare l'albero di ricerca che si ottiene a partire da un albero vuoto inserendo uno dopo l'altro, nell'ordine indicato, i seguenti numeri:
13,22,20,11,24,35,28,40,19,14,10,12,16,25,15,27,18

Eliminare i nodi 13,24,11



Complessità temporale

La complessità temporale, misurata in numero di "passi", delle operazioni di inserimento, ricerca e cancellazione, è $T(n) = O(h)$, dove h è l'altezza dell'albero.

Ricordando che negli alberi binari con n nodi, vale la relazione $\log_2 n \leq h < n$, le operazioni nel caso peggiore vengono eseguite in tempo $O(n)$.

Relazione tra altezza e numero di nodi:

In un albero binario di altezza h , ho un numero di nodi compreso tra $h + 1$ (caso degenere, in cui l'albero binario diventa una lista) e $2^{h+1} - 1$ (per ogni livello di profondità i , ho 2^i nodi).

La sommatoria per i che va da 0 ad h di 2^i è $2^{h+1} - 1$.

Utilizza **Lista di adiacenza con pesi**

Strategia Greedy

Input: Grafo $G = (V, E)$ privo di archi con peso negativo, una funzione peso $\omega : E \rightarrow R$, un vertice di partenza $s \in V$

Vettore d delle distanze **provvisorie**, in cui $d[v]$ indica il peso minimo del cammino da s a v fin'ora trovato.

Inizialmente, $d[v]$ è uguale a 0, se $s = v$, ∞ altrimenti.

Con strategia greedy, si considera un insieme C di elementi candidati a formare una soluzione (inizialmente $C = V$). Procedo per ogni passo a scegliere un vertice u a distanza minima da s .

Per ogni nodo v collegato ad u , se il percorso da s a v passando per u , è minore del percorso attualmente trovato da s a v , lo aggiorno.

ALGORITMO Dijkstra (G , vertice s) \rightarrow Vettore

Sia $d[V]$ un vettore con indici in V

$d[s] \leftarrow 0$

FOR EACH $v \in V - \{s\}$ DO $d[v] \leftarrow \infty$

$C \leftarrow V$

WHILE $C \neq \emptyset$ DO

$u \leftarrow$ elemento di C con $d[u]$ minima

$C \leftarrow C - \{u\}$

FOR EACH $(u, v) \in E$ DO

IF $d[u] + \omega(u, v) < d[v]$ THEN

$d[v] \leftarrow d[u] + \omega(u, v)$

RETURN d

Questa versione, restituisce un vettore d , contenente le distanze minime tra il nodo di partenza e gli altri. Come ricaviamo i cammini minimi?

Come prima, introduciamo un vettore dei predecessori $prev[v]$

```

ALGORITMO Dijkstra (Grafo G, vertice s) → Vettore
Sia d[V] un vettore con indici in V
Sia pred[V] un vettore con indici in V
d[s] ← 0
FOR EACH v ∈ V \ {s} DO d[v] ← ∞
Sia C una coda con priorità uoFa
FOR EACH v ∈ V DO C.insert(v, d[v])
WHILE C ≠ ∅ DO
    u ← C.deleteMin()
    FOR EACH (u, v) ∈ E DO
        IF d[u] + w(u, v) < d[v] THEN
            d[v] ← d[u] + w(u, v)
            C.changeKey(v, d[v])
            pred[v] ← u
RETURN d

```

L'insieme dei candidati C , è implementato utilizzando una coda con priorità. Ogni elemento della coda C , è un vertice v con chiavi pari a $d[v]$.

Essendo le code con priorità implementate tramite un min heap, estrarre l'elemento a distanza minima da s , significa semplicemente estrarre la radice dal min heap, operazione che avviene in tempo logaritmico.

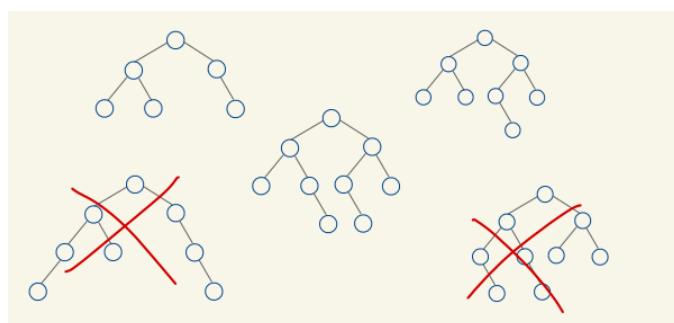
Complessità temporale:

Il costo maggiore, è dato dal ciclo for interno, che scorre per tutti gli m archi, e per ognuno di essi, esegue eventualmente un changeKey, a costo logaritmico.

Costo totale: $O(m \log n)$, tenendo conto che se G è connesso, allora $m \geq n - 1$.

Alberi perfettamente bilanciati

Un albero binario è perfettamente bilanciato, se per ogni nodo, la differenza (in valore assoluto) tra il numero di nodi nel sottoalbero sinistro, e il numero di nodi nel sottoalbero destro, è al più 1.



L'albero in basso a sinistra non è perfettamente bilanciato, se considero il primo nodo sulla destra, il suo sottoalbero destro ha due nodi, il suo sottoalbero sinistro ne ha 0.

Analizziamo l'altezza di un albero perfettamente bilanciato:

- Numero massimo di nodi: $2^{h+1} - 1$ (Albero completo)
- Numero minimo di nodi: 2^h

Lo si dimostra per induzione:

- Se $h=0$, ho solo la radice, $n=1$, che è uguale a 2^0 .
- Per $h > 0$,

Sia T un albero perfettamente bilanciato di altezza h , esso è composto da una radice, e due sottoalberi T' , T'' di altezza $h - 1$.

Per ipotesi induttiva, T' di altezza $h - 1$, ha almeno 2^{h-1} nodi.

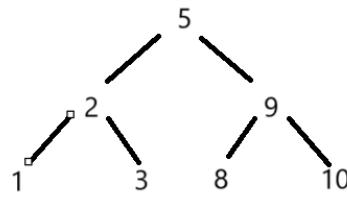
Sempre per ipotesi, T è perfettamente bilanciato, quindi T'' può differire al massimo di 1 rispetto al numero di nodi di T' . T'' ha allora $2^{h-1} - 1$ nodi.

In totale, T è composto da $1 + 2^{h-1} + 2^{h-1} - 1 = 2^h$ nodi.

Conseguenza: $h = \lfloor \log_2(n) \rfloor$

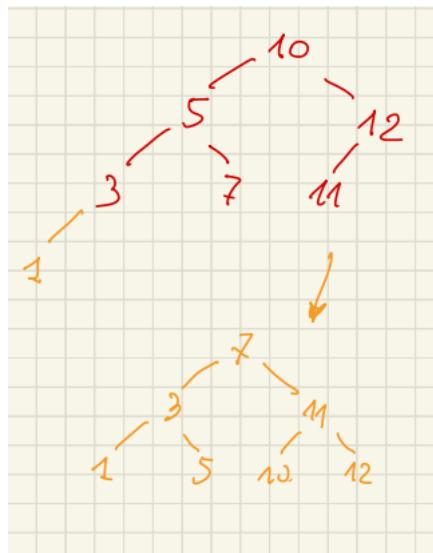
Da $h = \lfloor \log_2(n) \rfloor$, le operazioni di ricerca (in un albero binario di ricerca perfettamente bilanciato) avvengono in tempo logaritmico.

Se considerassi la sequenza di numeri: 1,2,3,5,8,9,10 da inserire in un albero binario di ricerca, otterrei un albero estremamente inefficiente (caso degenere)



Supponendo di conoscere a priori la sequenza di nodi da dover inserire, posso costruire un albero perfettamente bilanciato, considerando come radice la mediana dei numeri da inserire.

Gli alberi perfettamente bilanciati, risultano estremamente inefficienti nei casi di inserimento:



Volendo inserire il nodo 1 nell'albero sopra, la struttura non sarà più perfettamente bilanciata (dalla radice, il sottoalbero sinistro ha 4 nodi, quello destro ne ha 2).

Bisogna spostare e ricostruire tutto l'albero, operazione che avviene in tempo $O(n)$.

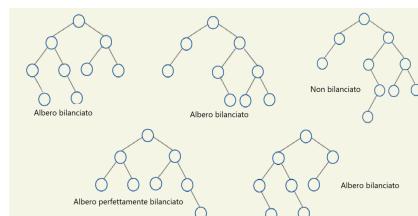
Gli alberi perfettamente bilanciati, vanno utilizzati esclusivamente in casi in cui sono previsti inserimenti minimi.

Alberi bilanciati in altezza (AVL)

Un albero binario è bilanciato (in altezza), quando per ogni nodo, la differenza (in valore assoluto) tra le altezze dei suoi sottoalberi è al più 1.

E' una condizione meno forte rispetto a quella degli alberi perfettamente bilanciati.

Se un albero è perfettamente bilanciato, allora è anche bilanciato in altezza. Non vale il contrario.



Dato un albero perfettamente bilanciato di altezza h :

- Numero massimo di nodi: $2^{h+1} - 1$ (Albero binario completo)
- Numero minimo di nodi = Numero di nodi del corrispondente albero di Fibonacci di altezza h

Un albero di Fibonacci di altezza h , è un AVL di altezza h , con minor numero possibile di nodi.

In esso, $n_h = F_{h-3} - 1$, dove F_{h-3} è il numero di Fibonacci corrispondente.

Si definisce il numero di Fibonacci come $F_1 = F_2 = 1, F_k = F_{k-1} + F_{k-2}$

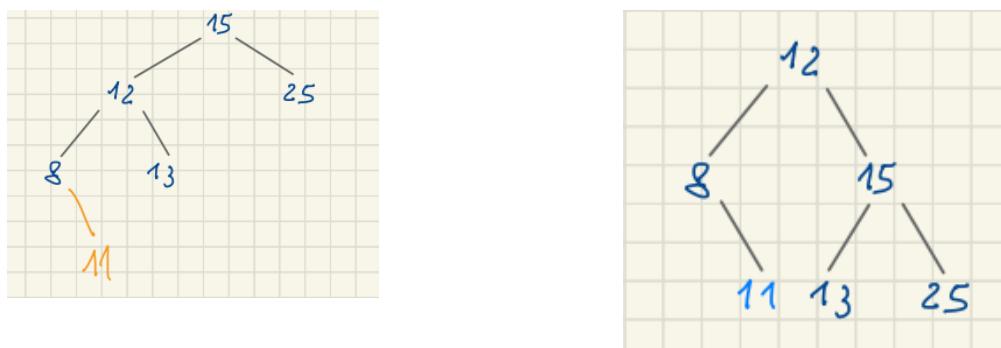
Si dimostra, che l'altezza dell'albero è $O(\log n)$, quindi le operazioni di ricerca avvengono in tempo logaritmico.

Inserimento e bilanciamento

L'inserimento in un AVL, segue le stesse regole di un normale albero binario di ricerca, cioè confronto il valore da inserire con la radice, e procedo ricorsivamente a dx o sx fino a trovare la posizione corretta.

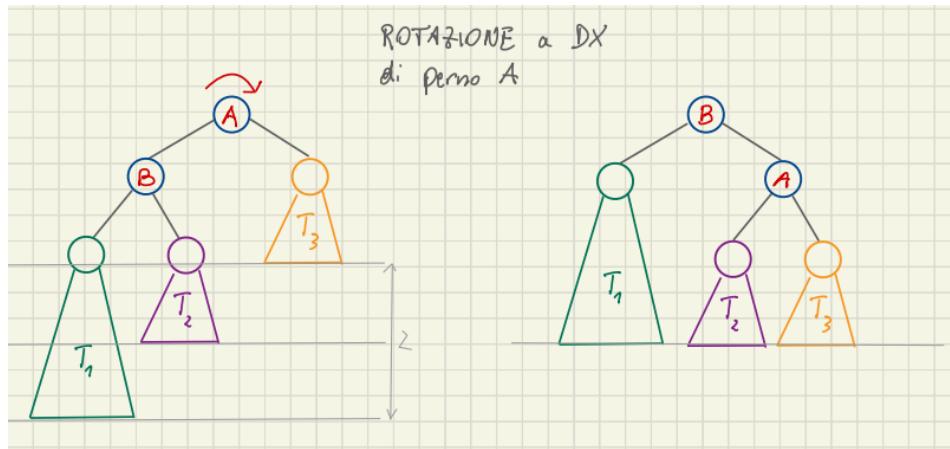
Anche in questa struttura, un'operazione di inserimento potrebbe portare alla necessità di ribilanciare l'albero, per continuare a garantire la proprietà di bilanciamento.

Sbilanciamento dell'albero a sinistra, nel sottoalbero sinistro



Volendo aggiungere il nodo 11 all'albero, la radice non è più bilanciata.

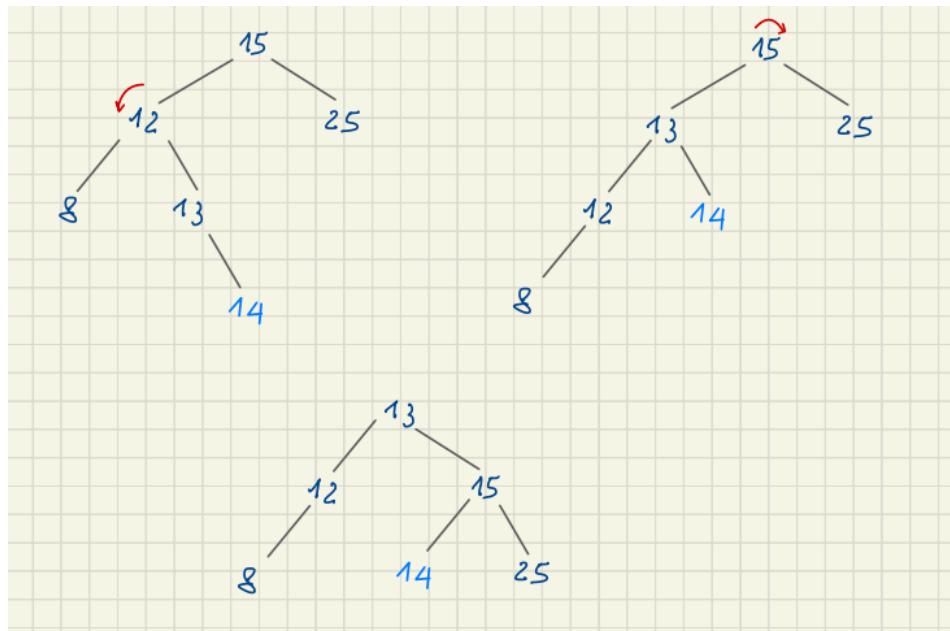
Attuo una "rotazione dell'albero" a Destra



"Faccio scendere" la radice A a destra, muovo il puntatore del sottoalbero destro di B, al sottoalbero sinistro di A.

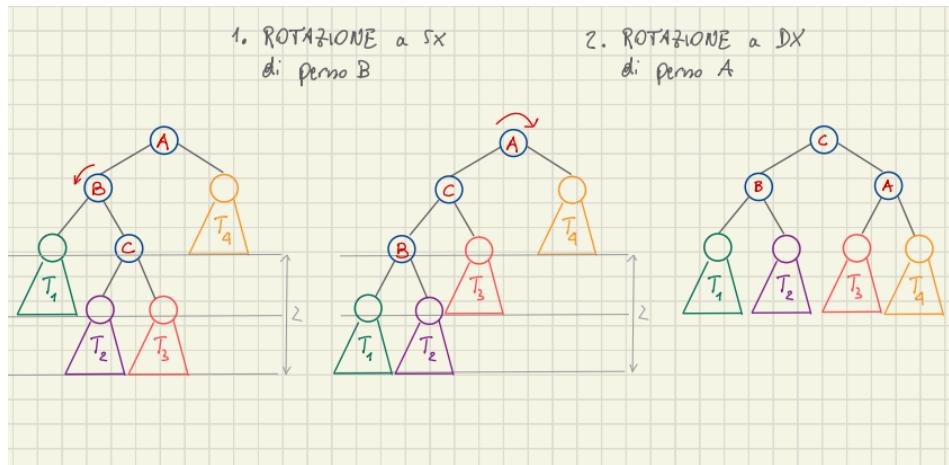
Devo semplicemente modificare il valore di un puntatore.

Sbilanciamento dell'albero a destra, nel sottoalbero sinistro



Attuo una rotazione a sinistra di perno B, e una successiva rotazione a destra di perno A (come esempio precedente)

Devo semplicemente modificare il valore di due puntatori



Riepilogo costo operazioni

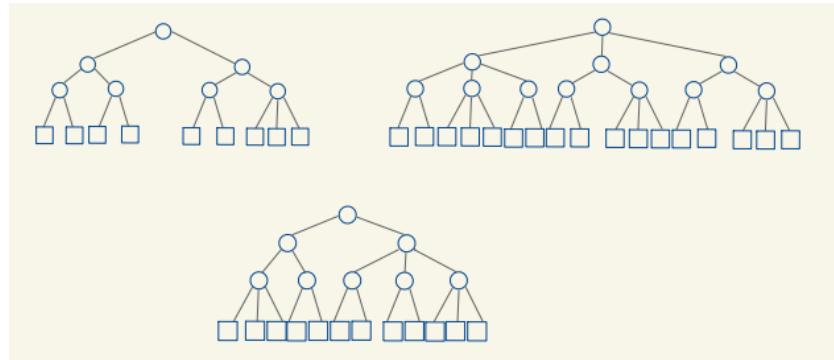
In un AVL, le tre operazioni di ricerca, inserimento e cancellazione, avvengono in tempo $O(\log n)$.

Le operazioni di ribilanciamento, avvengono in tempo costante $O(1)$.

Negli array ordinati invece, la ricerca avviene in $O(\log n)$ (ricerca dicotomica), ma l'inserimento avviene in costo $O(n)$.

Alberi 2-3

Albero in cui ogni nodo intermedio ha 2 o 3 figli, e tutte le foglie si trovano allo stesso livello (condizione di bilanciamento)



Dato un albero 2-3 di altezza h :

- Numero massimo di nodi lo si ottiene quando ogni nodo intermedio dell'albero ha 3 figli: ottengo un albero con $\frac{3^{h+1}-1}{2}$ nodi, e 3^h foglie.
- Numero minimo di nodi lo si ottiene quando ogni nodo intermedio dell'albero ha 2 figli, quindi ottengo un albero binario completo di altezza h , con $2^{h+1} - 1$ nodi, e 2^h foglie.

Il numero di foglie è allora compreso tra $2^h \leq f \leq 3^h$, passando ai logaritmi, otteniamo che l'altezza è logaritmica in base al numero di foglie $h = O(\log f)$.

Alberi 2-3 di ricerca

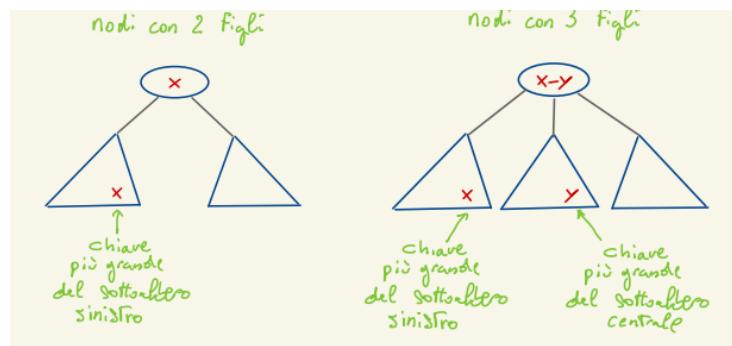
I Dati sono memorizzati esclusivamente nelle foglie, in ordine crescente da sinistra a destra.

Utilizzati per implementare i dizionari in memoria centrale

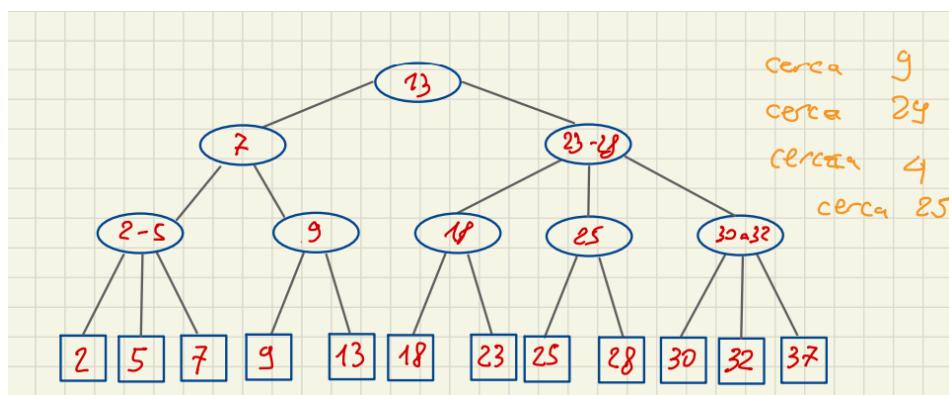
```
type foglia struct{
    key int
    others...
}
```

I nodi interni, servono a memorizzare alcune informazioni di instradamento verso le foglie.

```
type nodoInterno struct{
    chiaveMaggioreSottoalberoSinistro int
    chiaveMaggioreSottoalberoCentrale int
    puntatoreAlberoSX
    puntatoreAlberoCentro
    puntatoreAlberoDX
    puntatoreAlPadre (Utile per inserimenti e cancellazioni)
    others...
}
```



Ricerca di un nodo:



```
IF numeroDaCercare <= nodo.ChiaveMaggioreAlberoSX THEN
    Cerca albero SX
ELSE IF numeroDaCercare <= nodo.ChiaveMaggioreAlberoCentrale (e il nodo centrale esiste) THEN
    Cerca albero centrale
```

ELSE

Cerca albero DX

La ricerca termina quando arrivo ad un nodo foglia: se la chiave della foglia è uguale a quella che sto cercando, si restituisce il puntatore alla foglia, altrimenti restituisco null.

L'Operazione di ricerca ha costo logaritmico $O(h) = O(\log f)$, così come quelle di inserimento e cancellazione del nodo.

Inserimento

Durante l'inserimento, è importante tener bilanciato l'albero.

Supponiamo di inserire un nuovo elemento con chiave k : creo un nuovo nodo u con chiave k , e procedo ad effettuare una ricerca partendo dalla radice, per trovare la posizione giusta in cui esso va collocato.

Trovo così un nodo v al penultimo livello, a cui va attaccato u :

- Se v ha due figli, posso attaccare u direttamente ad esso, come figlio sx, dx o centrale.
- Se v ha già tre figli, non posso attaccarne un altro. Eseguo un'operazione di split, cioè divido v in due nodi, creando un nuovo nodo w . Assegno a w la chiave più grande del sottoalbero centrale.

I Figli originari di v , vengono ripartiti, i minori a v , i maggiori a w .

A quel punto, attacco w al padre di v .

Se il padre di v aveva due figli, l'operazione termina. Altrimenti, effettuo nuovamente split sul padre.

Ciascuno split ha la conseguenza di far crescere verso l'alto l'altezza dell'albero. Ogni split avviene in tempo costante. Nel caso peggiore, posso eseguire fino ad $O(h) = O(\log f)$ split.

Cancellazione

Quando cancellando un nodo, rimane un nodo interno con un solo figlio, devo effettuare un'operazione inversa allo split.

Questo perché se consentissi la presenza di nodi con un solo figlio, potrebbe capitare (come negli ABR), che l'albero degeneri in una lista.

Riepilogo costo operazioni

In un Albero 2-3, le tre operazioni di ricerca, inserimento e cancellazione, avvengono in tempo $O(\log f)$.

Memorizzazione dei dati su memoria secondaria

DBMS (Data Base Management System), Filesystem, Utilizzano indici per garantire accesso veloce e diretto ai dati.

Gli indici possono essere molto grandi (milioni di chiavi), e non possono essere memorizzati in memoria centrale. Possono essere realizzate delle strutture ad albero, per rappresentare dizionari in memoria secondaria, ed implementare gli indici.

Memorizzare i dati in memoria secondaria (di massa) e non in memoria centrale, comporta due principali svantaggi:

- Accesso lento ai dati, dovuto alla presenza di parti meccaniche (I dischi sono **100.000 volte più lenti** della RAM.)
- Accesso in memoria di massa viene effettuato solo per blocchi (o pagine), e non per singolo dato, per cui ad ogni accesso si ottengono più dati.

In un albero binario classico:

- Ogni nodo ha 1 chiave → altezza alta → **molti accessi a disco**
- Ogni accesso a un figlio può richiedere **una nuova lettura del blocco**

Soluzione:

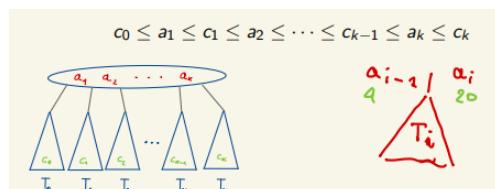
- **B-albero**: ogni nodo contiene **molte chiavi** (decine o centinaia)
- Altezza dell'albero **drasticamente ridotta** → meno accessi al disco
- Le chiavi di un nodo sono memorizzate **tutte nello stesso blocco**
- Quando accedi a un blocco, **sfrutti tutta la sua capacità**

B-Alberi

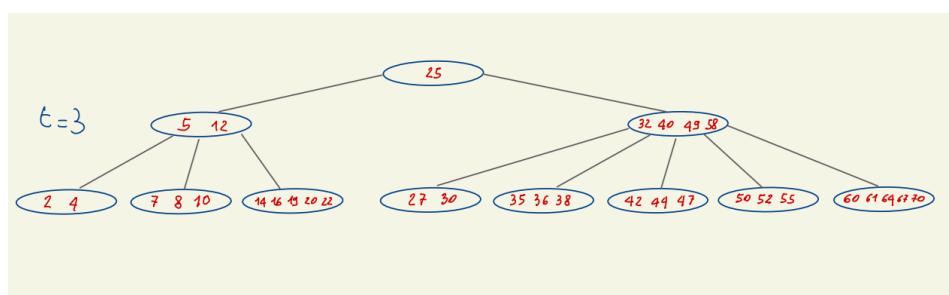
Un **B-albero** è una struttura dati ad **albero bilanciato**, progettata per **funzionare efficientemente in memoria secondaria** (come dischi o SSD). È un'estensione dell'albero di ricerca binario, ma ogni **nodo può contenere molte chiavi** e avere molti figli.

Un B-Albero di ordine t è un albero tale che:

- Ogni nodo interno ha al massimo $2t$ figli
- Ogni nodo interno diverso dalla radice ha almeno t figli.
- La radice ha almeno 2 figli
- Tutte le foglie si trovano allo stesso livello
- Ogni foglia contiene k chiavi ordinate in modo crescente (k deve essere compreso tra $t - 1$ e $2t - 1$, o tra 1 e $2t - 1$ se la foglia è la radice)
- Ogni nodo interno, con $k + 1$ figli e sotto alberi $T_0, T_1 \dots T_k$, contiene k chiavi ordinate $a_1 \dots a_k$, tali per cui:
 - Tutte le chiavi in T_0 sono $< a_1$
 - Tutte le chiavi in T_1 sono comprese tra a_1 e a_2
 - \vdots
 - Tutte le chiavi in T_{k-1} sono comprese tra a_{k-1} e a_k
 - Tutte le chiavi in T_k sono $\geq a_k$



L'albero vuoto è sempre un B-Albero di ordine t .



Nell'esempio, il 5 (nel secondo nodo a sinistra), è maggiore di 2 e 4, e minore di 7,8,10.

Il 12, è maggiore di 7,8,10, e minore di 14,16,19,20,22.

A differenza degli alberi 2-3, nei B alberi le informazioni stanno sia nei nodi interni che nelle foglie.

Ricerca di un nodo

Durante la ricerca, si alternano due operazioni:

- Ricerca in un singolo nodo: essendo ogni nodo un array ordinato con al più $2t - 1$ elementi, la ricerca dicotomica viene effettuata in $O(\log t)$.
- Spostamento nel sottoalbero indicato dalle guide.

Al più, eseguirò quindi h ricerche dicotomiche.

Richiede $O(h \log t)$ passi.

Dalla relazione $h \leq \log_t \frac{n+1}{2}$, dove n è il numero di chiavi, si ottiene che la ricerca avviene in $O(\log n)$, indipendentemente da t (n è il numero di chiavi dell'albero, non il numero di nodi).

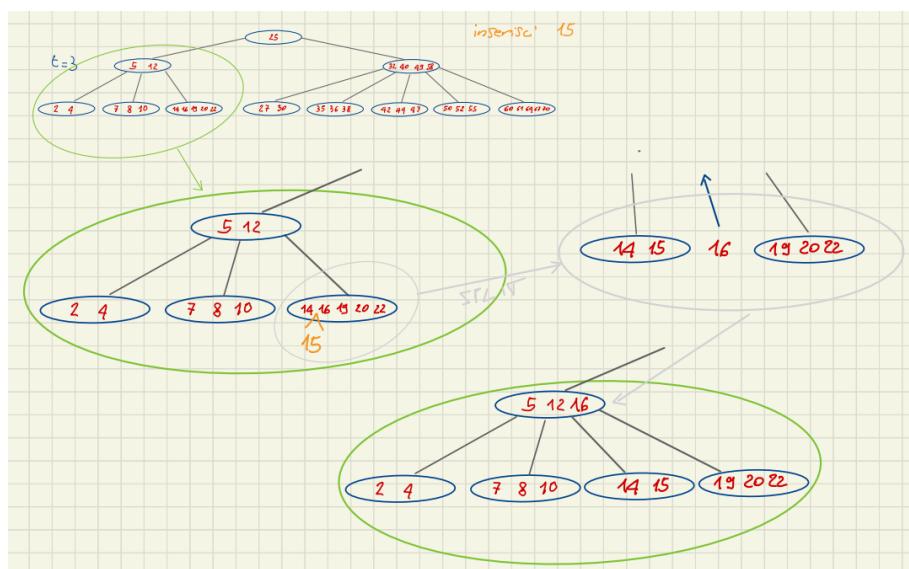
Inserimento di un nodo

Ricordando che ogni foglia (gli elementi sono inseriti solo nei nodi foglia) può avere massimo $2t - 1$ nodi, al momento dell'inserimento distinguiamo due casi:

Ricercò la foglia in cui il nuovo nodo (chiave) andrebbe inserito:

- Nella foglia c'è posto, inserisco
- In quella foglia ci sono già $2t - 1$ chiavi

Se inserissi un nuovo nodo, avrei $2t$ chiavi: posso allora dividere la foglia da $2t$ chiavi, in due foglie da $t - 1$ e t chiavi (una chiave (quella "in mezzo" tra il blocco $t - 1$ e t) viene messa nel padre di queste due nuove foglie)



Il costo dell'inserimento è $O(t \log n)$ ($O(t)$ per l'inserimento in un array ordinato, $O(\log n)$ per la ricerca), perché nel caso peggiore, lo split può propagarsi fino alla radice.

Cancellazione

Strategia inversa allo split, ossia il merge. Il costo è lo stesso dell'inserimento.

Accessi a memoria secondaria

Per minimizzare il numero di accessi alla memoria secondaria, vogliamo dimensionare adeguatamente t , per far sì che tutte le chiavi in un nodo siano contenute in uno stesso blocco su disco.

A questo punto, io per ogni nodo effettuo un solo accesso a memoria.

Il numero di accessi a memoria durante una ricerca, diventa allora pari all'altezza dell'albero.

$$O(h) = O(\log_t n)$$

Per le operazioni di inserimento/cancellazione invece, il numero di accessi a memoria è circa pari a $C * \log_t n$, dove C è una costante piccola, indicativamente circa 4.

Dizionari

Collezione di elementi ciascuno dei quali identificato da una chiave.

Su di essi, identifichiamo le operazioni principali:

- Ricerca
- Inserimento
- Cancellazione

I cui costi variano in base a come il dizionario è implementato:

Implementazione tramite:	Array ordinati Svantaggio: capacità fissa	ABR	Alberi AVL o 2-3
Ricerca	$O(\log n)$	$O(n)$	$O(\log n)$
Inserimento	$O(n)$	$O(n)$	$O(\log n)$
Cancellazione	$O(n)$	$O(n)$	$O(\log n)$

L'implementazione tramite alberi AVL o 2-3, risulta la più efficiente, a patto di avere abbastanza memoria per la rappresentazione dei puntatori.

Tabelle hash

E' una struttura dati che permette di implementare un dizionario tramite un'array.

L'idea è associare ad esso una funzione hash ($h : U \rightarrow \{0, 1, 2 \dots m - 1\}$, dove m è il numero di elementi del dizionario/array, U è l'universo delle chiavi), che data una chiave, mi restituisce l'indice di tale elemento nell'array (dizionario).

Fattore di carico:

Definito come $\frac{n}{m}$, dove n è il numero di elementi memorizzati nella tabella, m il numero di posizioni disponibili.

E' un valore compreso tra 0 e 1: 0 indica che la tabella è vuota, 1 indica che la tabella è piena.

Tabella hash ad accesso diretto

La più semplice implementazione di una tabella hash, è quella ad accesso diretto.

Ipotizzando di dover memorizzare record con chiavi intere univoche, l'insieme delle chiavi $U = \{0, \dots, m - 1\}$, utilizziamo come funzione hash $h(i) = i$, cioè ogni record di chiave i , viene memorizzato nella posizione i dell'array.

Questo implica che avrò bisogno di un array di m elementi.

Le operazioni di ricerca, inserimento, cancellazione, avvengono in tempo costante, in quanto dato un record, so subito in che posizione è memorizzato.

In questo approccio, le collisioni sono impossibili, ma difficilmente è praticabile.

Quando le chiavi assumono valori molto grandi, io devo prevedere un array molto grande, per potenzialmente utilizzare poche posizioni, portando ad uno spreco di memoria elevato (fattore di carico vicino allo 0)

Esempio: Numeri di telefono.

Un numero di telefono inizia con 3 ed ha 10 cifre.

Per memorizzare tutti i numeri di telefono da 10 cifre, dovrei prevedere tutti i valori delle chiavi che vanno da 3000000000 a 3999999999, cioè dovrei avere un array di $m = 1000000000$ (un milione) elementi, per poi utilizzarne una porzione decisamente minore.

Funzione hash perfetta

Una funzione hash è perfetta, se è iniettiva, cioè date due chiavi $u \neq v$, allora $f(u) \neq f(v)$: ciò significa che la tabella deve poter contenere tanti elementi quanti sono gli elementi dell'universo delle chiavi. (non si verificano mai collisioni)

Esempio: se volessi definire una funzione hash perfetta, per associare gli studenti di UniMi, dovrei implementare una tabella da più di un milione di studenti.

Nella realtà, il numero di chiavi possibili, è molto più grande del numero di chiavi attese.

La dimensione della tabella (m), viene scelta paragonabilmente al numero di chiavi attese.

Esempio: se volessi rappresentare le persone in base al cognome, in cui ogni cognome ha lunghezza massima 10 caratteri, allora dovrei rappresentare 26^{10} stringhe, tabella troppo grande! In questi casi è indispensabile dover ricorrere a funzioni hash non perfette, anche perché sarebbe inutile rappresentare tutte le stringhe possibili di lunghezza 10, molte di queste sarebbero insensate (Esempio: ahsdodjoia è insensata, non può essere un cognome!)

Collisioni

Si parla di collisione, quando dovendo due chiavi hanno stessa posizione all'interno della tabella.

Vogliamo evitarne il più possibile, cioè far in modo che la funzione hash "sparpagli" il più possibile le chiavi nella tabella, per far sì che non ci siano "accumuli" di chiavi in posizioni vicine, che renderebbero più probabili le collisioni.

Funzioni hash

Una buona funzione di hash deve:

- Sparagliare
- Essere uniforme, cioè dato un indice i , la probabilità che esso venga scelto è $1/m$ (m è il numero di elementi della tabella)

Esempio: devo rappresentare 25 studenti, utilizzando come chiave la prima lettera del proprio cognome.

La funzione di hash, prende la prima lettera del cognome, e associa la posizione nell'alfabeto di quella lettera

Non è uniforme, la probabilità che ci siano cognomi con una lettera è maggiore di cognomi con altre (Esempio: cognomi con la I vs cognomi con la Z, la probabilità che escano cognomi con la Z è nettamente minore).

- Veloce da calcolare
- Dipendere dall'intera chiave

Esempi di funzione hash:

- Trasformo la chiave in un intero
- Moltiplico il valore ASCII corrispondente ad ogni carattere, per 31^n , dove n è la posizione in cui compare tale carattere

- Divido modulo $m = 1024$

U = parole della lingua italiana

$m = 1024$

Data la parola $x = \text{GATTO} \rightarrow$ Ne associo i valori ASCII $\rightarrow 71(\text{G})\ 65(\text{A})\ 84(\text{T})\ 84(\text{T})\ 79(\text{O})$

$$h(\text{"GATTO"}) = 71 * 31^4 + 65 * 31^3 + 84 * 31^2 + 84 * 31^1 + 79 * 31^0 = 67589813 \ MOD\ 1024 = 693$$

Buona uniformità, ma lenta da calcolare.

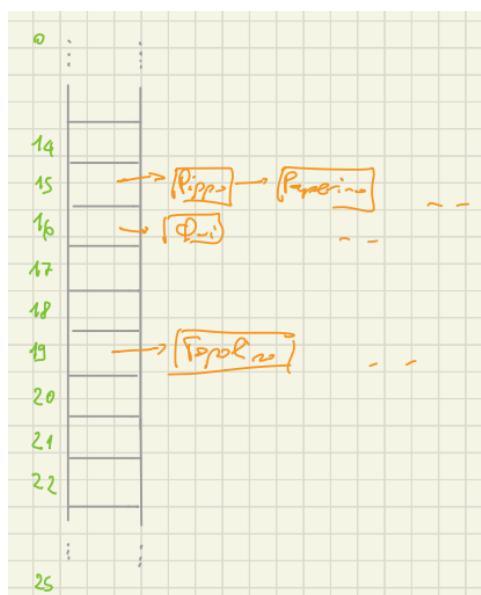
Funziona bene perché utilizzo un numero primo per moltiplicare (31).

Gestione delle collisioni

La posizione che dovrebbe essere utilizzata per una chiave x , è già occupata per un'altra chiave y

Gestione esterna (utilizza strutture dati esterne): Liste di collisione

Ogni elemento i dell'array con cui è implementata la tabella hash, è un realtà un puntatore ad una lista collegata, contenente i record la cui chiave ha indice comune pari a i



Nell'esempio, pippo e paperino "collidono", hanno indice comune pari a 15.

Se la funzione hash "sparpaglia male", potrebbero formarsi liste molto lunghe, e altre molto corte, ci sarebbero zone in cui i dati si concentrano più di altre.

Utilizzando questa tecnica:

- Inserimento - Inserendo in testa, l'inserimento è costante $O(1)$.
- Ricerca - Caso peggiore, tutti gli elementi sono in una lista sola, $O(n)$

Caso medio, se la funzione hash "sparpaglia bene", in media la lunghezza delle lista è $\frac{n}{m}$.

La ricerca viene effettuata in $O(1 + \frac{n}{m})$ passi.

Dato un elemento di chiave k da cercare, consideriamo $O(1)$ per il calcolo dell'hash $h(k)$, poi $\frac{n}{m}$ passi per cercare la chiave nella lista $h(k)$

- Cancellazione - Stesse considerazioni della ricerca.

Gestione interna: Indirizzamento aperto

Se la funzione hash applicata ad una chiave restituisce un indice già occupato, si applica una strategia predefinita per cercare un'altra posizione libera.

Utilizziamo funzioni ausiliarie $c(K, i)$, dove K è una chiave, i è un intero ≥ 0 , tale che $c(K, 0) = h(K)$, e $\{c(K, 0), c(K, 1), c(K, 2) \dots\} = \{0, 1, 2 \dots m - 1\}$, cioè la funzione non esca dai limiti della tabella

- Scansione lineare: $c(K, i) = (h(K) + i) \text{ MOD } m$ (cioè se la posizione $h(k)$ è occupata, vado avanti finché non trovo una posizione libera. Il MOD serve per far sì che non possa uscire dai limiti della tabella)

$f : \{a, \dots, z\} \rightarrow \{0, \dots, 15\}$		
x	f(x)	
a	0	
b	1	
c	2	
d	3	
e	4	
f	5	
g	6	
h	6	
i	7	
j	7	
k	7	
l	8	
m	9	
n	10	
o	10	
p	11	
q	12	
r	12	
s	13	
t	13	
u	14	
v	14	
w	15	
x	15	
y	15	
z	15	

funzione hash: $h(k) = f(\text{prima lettera di } k)$ scansione lineare inserire nell'ordine: salmone tonno rombo aringa cernia aragosta totano ostrica nasello palombo	0 ARINGA 1 ARAGOSTA 2 CERNIA 3 PACOMO 4 5 6 7 8 9 10 OSTRICA 11 MASULLO 12 ROMBO 13 SALMONE 14 TONNO 15 PUFANO
---	---

- Scansione quadratica: $c(K, i) = \lfloor h(K) + c_1 i + c_2 i^2 \rfloor \text{ MOD } m$, con c_1 e c_2 opportuni
- Hashing doppio: $c(K, i) = (h(K) + i h'(K)) \text{ MOD } m$, dove h' è una seconda funzione hash (dove preferibilmente, non ci sono collisioni, cioè se nella prima funzione hash, $h(k1) = h(k2)$, per la seconda $h'(k1) \neq h'(k2)$)

Inserimento di un elemento (Hash Table)

```

FUNZIONE Inserimento (Elemento e, chiave K)
    i<-0
    While i<m AND V[c(K,i)] è occupata DO //dove v è la tabella
        i<-i+1
    IF i<m THEN v[c(K,i)] <- (e,k) //inserisco elemento
    ELSE errore //la tabella è piena

```

Ricerca (Hash Table)

```

FUNZIONE Ricerca(chiave K) -> elemento
    i<-0
    While i<m AND V[c(K,i)] è occupata AND V[c(K,i)].key != chiave DO //dove v è la tabella
        i<-i+1
    IF i=m OR V[c(K,i)] è libera THEN
        RETURN null //elemento non trovato

```

```

    ELSE
        RETURN V[c(K,i)] //found

```

Il costo delle operazioni, dipende dal costo di scansione (posizioni da visitare), caso peggiore $O(n)$ (scandisco tutte le posizioni).

In media, se la funzione hash "sparpaglia bene", possiamo dire:

<i>n° di passi</i>	scansione lineare	scansione quadratica hashing doppio	$n = \# \text{chiavi presenti}$ $m = \dim \text{tabella}$
chiave trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$	$-\frac{1}{\alpha} \lg_e (1-\alpha)$	
chiave non trovata	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{1-\alpha}$	$\alpha = \frac{n}{m}$ FATTORE DI CARICO

Esempio nel caso la chiave non fosse trovata, utilizzando scansione quadratica o hashing doppio:

- Se la tabella è piena per metà, $\alpha = 1/2$, riesco a determinare che l'elemento non c'è dopo due passi.
- Se la tabella è piena per tre quarti, $\alpha = 3/4$, riesco a determinare che l'elemento non c'è dopo 4 passi.
- Se la tabella è piena, $\alpha = 1$, diventa infinito (1/0).

Re-hashing

All'aumentare del riempimento della tabella quindi, le prestazioni dell'hash table diminuiscono, e aumenta il rischio di collisioni.

Quando la tabella viene riempita oltre la soglia stabilita (viene specificato un limite in termini di fattore di carico, generalmente 0.5), per mantenere alte le performance della struttura dati, viene creata una nuova tabella (di solito grande il doppio), in cui vengono copiati tutti gli elementi.

- Devo riadattare la funzione hash alla nuova dimensione della tabella
- Costoso in termini di tempo

Analisi ammortizzata nel tempo:

Ipotizziamo una tabella T_0 di dimensione iniziale m , massimo fattore di carico consentito $\alpha = 1/2$, e di effettuare una serie di inserimenti che richiedano rehashing

tabella	T_0	\rightarrow	T_1	\rightarrow	T_2	\rightarrow	\dots	\rightarrow	T_k
capacità	m		$2m$		2^2m		\dots		$2^k m$
numero max elementi	$m/2$		m		$2m$		\dots		$2^{k-1}m$

Partendo da T_0 inizialmente vuota, ed effettuando in totale N inserimenti (effettuando quando necessario re-hashing), otteniamo che oltre agli N inserimenti, ne effettuo un numero ulteriore $< 2n$ dovuto al rehashing.

$$N + 2N = 3N = O(N).$$

Se effettuiamo allora N operazioni di inserimento, a causa del rehashing effettuiamo allora in totale $3N = O(N)$ inserimenti, dividendo per N inserimenti effettivi, ottengo sembra $O(N)/N = O(1)$, pertanto anche effettuando rehashing, l'inserimento rimane di costo costante.

Risorse necessarie e sufficienti

Siano:

- r una risorsa computazionale (tempo, spazio...)
- π un problema risolvibile algoritmamente (esistono problemi non risolvibili tramite algoritmi, esempio: stabilire se un programma termina, o stabilire se un programma è corretto).

Ci chiediamo, qual'è la quantità di risorsa r necessaria (limite inferiore), e quant'è quella sufficiente (limite superiore), per risolvere π ?

Limitazione superiore (Upper bound)

Definiamo una funzione $f : N \rightarrow N$, dove il dominio è la lunghezza di un input, il codominio rappresenta il numero di risorse.

La quantità $f(n)$ di una risorsa r è sufficiente per risolvere π , se esiste un algoritmo A che risolve π , utilizzando al più $f(n)$ risorse r per ogni input di lunghezza n .

Limitazione inferiore (Lower bound)

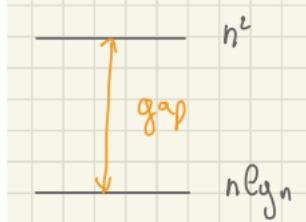
Definiamo una funzione $g : N \rightarrow N$, dove il dominio è la lunghezza di un input, il codominio rappresenta il numero di risorse.

La quantità $g(n)$ di una risorsa r è necessaria per risolvere π , se per ogni algoritmo A che risolve π , esiste un input di lunghezza n , su cui A utilizza ALMENO $g(n)$ risorse r .

Esempio:

- π Problema - sorting
- r Risorsa - Numero di confronti

Analizziamo l'algoritmo InsertionSort.



Upper bound: $O(n^2)$ (O , worst case)

Lower bound: $\Omega(n \log n)$, si dimostra che ogni algoritmo di ordinamento basato su confronti, utilizza ALMENO (Ω , best case) $n \log n$ per ordinare n chiavi.

Complessità computazionale

Sistema di classificazione dei problemi, in base alla quantità di risorse utilizzate per la loro soluzione.

Classi di complessità

Insieme di problemi, che possono essere risolti utilizzando la stessa quantità di una determinata risorsa r .

Classe P

$$P = \bigcup_{c=0}^{\infty} TIME(n^c)$$

Classe dei problemi risolubili (ottimamente) in tempo polinomiale, in relazione alla lunghezza dell'input.

Alcuni esempi di problemi appartenenti a questa classe, sono:

- Prodotto di matrici
- Ordinamento di un vettore
- Minimum spanning tree

Il problema dello zaino per esempio, non è risolvibile ottimamente in tempo polinomiale (l'algoritmo greedy, non da sempre soluzione ottima)

Tipologie di problemi

Consideriamo un problema $\pi \subseteq I \times S$, dove I è l'universo dei possibili input, S è l'universo delle soluzioni

- Ricerca - Dato un input $x \in I$, trovare $s \in S$ t.c il problema π è risolto da s , (utilizziamo questa notazione $(x, s) \in \pi$)
- Ottimizzazione - Dato un input $x \in I$, trovare $s \in S$ t.c $(x, s) \in \pi$ soddisfi un criterio di ottimalità fissato (es: minimo/massimo)
- Decisione - Restringiamo l'universo delle soluzioni a $\{0, 1\}$, risposta binaria.

Classe TIME e SPACE

Date le funzioni $s : N \rightarrow N$ e $t : N \rightarrow N$, definiamo le classi:

- $TIME(t(n))$ - Classe dei problemi di decisione, risolubili da algoritmi in tempo $O(t(n))$
- $SPACE(s(n))$ - Classe dei problemi di decisione, risolubili da algoritmi in spazio $O(s(n))$

E le classi derivate:

$$\begin{aligned} PSPACE &= \bigcup_{c=0}^{\infty} SPACE(n^c) && \text{spazio polinomiale} \\ EXPTIME &= \bigcup_{c=0}^{\infty} TIME(2^{n^c}) && \text{tempo esponenziale} \end{aligned}$$

Relazione tempo-spatio

Se fisso un tempo $t(n)$, e ipotizziamo che ogni istruzione macchina visiti al max k celle di memoria, io alla fine del tempo avrò visitato un numero $\leq k * t(n)$ celle.

Pertanto lo spazio utilizzato dal programma sarà $s(n) \leq k * t(n)$.

Se $t(n)$ è polinomiale, anche lo spazio lo è.

Relazione spazio-tempo

Tutto ciò che faccio in PSPACE, lo faccio in tempo EXPTIME esponenziale.

Se lo spazio è polinomiale, allora il tempo è esponenziale.