

Algoritmi e strutture di dati

Matteo Lombardi

Indice

- 1.0 - Concetti matematici
- 2.0 - Introduzione agli algoritmi
 - 2.1 - Ingredienti di un algoritmo
 - 2.2 - Algoritmi per il calcolo dei numeri di Fibonacci
- 3.0 - Notazione asintotica
 - 3.1 - Notazioni
 - 3.2 - Complessità computazionale
 - 3.3 - Analisi ammortizzata
 - 3.4 - Equazioni di ricorrenza
- 4.0 - Algoritmi di ordinamento
 - 4.1 - Algoritmi incrementali
 - 4.2 - Algoritmi divide et impera
 - 4.3 - Algoritmi non comparativi
- 5.0 - Strutture dati elementari
 - 5.1 - Dizionario con array
 - 5.2 - Lista
 - 5.3 - Pila
 - 5.4 - Coda
 - 5.5 - Albero
- 6.0 - Alberi
 - 6.1 - Alberi binari di ricerca
 - 6.2 - Alberi AVL
 - 6.3 - Algoritmi di decisione su alberi
- 7.0 - Tabelle Hash
 - 7.1 - Tabelle ad indirizzamento diretto
 - 7.2 - Tabelle Hash
- 8.0 - Heap
 - 8.1 - Heap inari
 - 8.2 - Code con priorità
- 9.0 - Union-find
 - 9.1 - Introduzione all'union-find
 - 9.2 - QuickFind e QuickUnion

10.0 - Tecniche algoritmiche

10.1 - Divide-et-impera

10.2 - Greedy

10.3 - Programmazione dinamica

11.0 - Grafi

11.1 - Introduzione ai grafi

11.2 - Algoritmi di visita di grafi

11.3 - Minimum spanning tree

11.4 - Cammini minimi

▼ 1.0 - Concetti matematici

Serie e successioni

Serie aritmetica

La somma dei primi numeri n numeri consecutivi, detta **serie aritmetica**, ha il seguente valore:

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

Serie geometrica

La somma di numeri aventi una base costante $q \neq 1$ e un esponente variabile, detta **serie geometrica** di ragione q , ha il seguente valore:

$$\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}$$

▼ 2.0 - Introduzione agli algoritmi

Un **algoritmo** rappresenta una procedura per risolvere un problema in un numero finito di passi.

▼ 2.1 - Ingredienti di un algoritmo

Differenza tra algoritmo e programma

Un **algoritmo** è una descrizione fatta in un alto livello di una procedura. Gli algoritmi non possono essere eseguiti in memoria e possono utilizzare una quantità illimitata di memoria.

Un **programma** invece è l'implementazione di un algoritmo. Deve essere scritto in un qualche linguaggio di programmazione, può essere eseguito su un computer e deve tenere conto dei limiti di memoria di quest'ultimo.

Ingredienti di un algoritmo

Un algoritmo prende in input alcuni valori, esegue una sequenza finita di operazioni e produce un output.

Esistono infiniti set di istruzioni che risolvono lo stesso problema, ovvero forniscono lo stesso output per lo stesso input.

Cosa ci serve per poter sviluppare algoritmi?

Capire il problema che vogliamo risolvere, chiedendoci anche se esistono proprietà matematiche legate ad esso.

Apprendere in che modo stimare l'efficienza di un algoritmo, in termini di tempo e memoria.

Studiare tecniche algoritmiche e strutture dati note, in quanto problemi differenti spesso condividono la stessa struttura di base.

▼ 2.2 - Algoritmi per il calcolo dei numeri di Fibonacci

Algoritmo 1: formula chiusa

Esiste una **formula chiusa** per calcolare il valore di F_n , ovvero $\frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$, dove ϕ e $\hat{\phi}$ sono due costanti irrazionali.

L'algoritmo dunque può semplicemente restituire il valore restituito da tale formula, il problema sta però nell'implementazione informatica di tale algoritmo, in quanto un computer non può memorizzare delle costanti irrazionali, e utilizzando dei valori approssimati risulterà nell'approssimazione del risultato, cosa non accettabile.

Algoritmo 2: soluzione ricorsiva 1

Il secondo algoritmo prevede la conversione diretta della seguente **formula ricorsiva**

$$F_n = \begin{cases} 1 & n \leq 2 \\ F_{n-1} + F_{n-2} & n > 2 \end{cases}$$

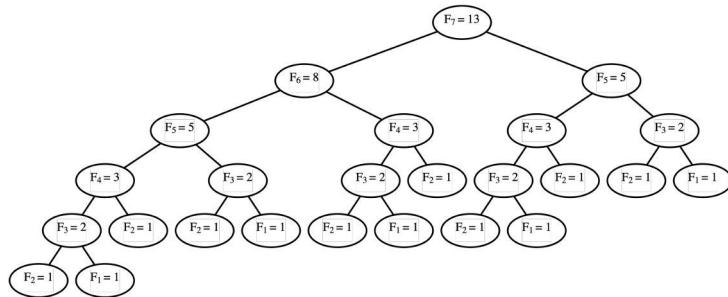
in un algoritmo ricorsivo.

Facciamo una stima della memoria e del tempo necessario al calcolo.

Stima della memoria

Ogni chiamata ricorsiva causa l'allocazione di un record di attivazione sullo stack, dunque la quantità di memoria utilizzata da tale algoritmo è causata unicamente dalle **chiamate ricorsive**.

Bisogna fare però attenzione alla quantità di chiamate ricorsive che il programma effettua in una singola esecuzione in quanto, come possiamo notare dal seguente **albero binario**, questo algoritmo non richiede solo la memoria per calcolare il numero n dato in input, ma anche per tutti i numeri inferiori ad esso.



Albero di ricorsione per $n = 13$.

Stima del tempo

Per stimare il tempo necessario all'esecuzione dell'algoritmo stimiamo innanzitutto il numero il numero $T(n)$ di nodi nell'albero di ricorsione, sapendo che l'albero $T(n)$ contiene 1 nodo + i nodi dei sottoalberi $T(n - 1)$ e $T(n - 2)$:

$$T(n) = \begin{cases} 1 & n \leq 2 \\ T_{n-1} + T_{n-2} + 1 & n > 2 \end{cases}$$

Da questa relazione possiamo inoltre stimare un limite inferiore per $T(n)$ sapendo che $T(n - 1) \geq T(n - 2)$:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq 2T(n-2) + 1 \\ &\geq 4T(n-4) + 2 + 1 \\ &\geq 8T(n-6) + 4 + 2 + 1 \\ &\geq \dots \\ &\geq 2^{\frac{n}{2}} + \frac{2^{\frac{n}{2}} - 1}{2 - 1} \\ &\geq 2^{\frac{n}{2}} \end{aligned}$$

Da ciò abbiamo ottenuto che la funzione presenta un **numero esponenziale di nodi** maggiore di $2^{\frac{n}{2}}$.

Algoritmo 3: soluzione iterativa

Visto che la soluzione precedente ricalcolava gli stessi numeri di Fibonacci più volte è facilmente pensabile una soluzione iterativa che memorizza all'interno di un **array** i numeri di Fibonacci calcolati:

```

int fib3(int n) {
    let F[1 ... n] be an array of int
    F[1] = 1
    F[2] = 2

    for (int i = 3; i <= n; i++) {
        F[i] = F[i-1] + F[i-2]
    }
}
  
```

```

        F[i] = F[i - 1] + F[i - 2]
    }

    return F[n]
}

```

Stima della memoria

Somma delle variabili utilizzate, ovvero di tutti i numeri memorizzati nell'array, **proporzionale al numero n dato in input.**

Stima del tempo

Calcoliamo il numero delle operazioni elementari del programma utilizzato: $4 + 3(n - 2)$, ovvero $3n - 2$, anch'esso **proporzionale al numero n dato in input.**

Algoritmo 4: soluzione efficiente in memoria

Visto che per calcolare un certo numero di Fibonacci occorre avere **in memoria solamente i due numeri precedenti**, non è necessario memorizzare tutti i numeri all'interno di un array, ottenendo quindi questo algoritmo:

```

int fib4(int n) {
    a = 1
    b = 1
    for (int i = 3; i <= n; i++) {
        c = a + b
        a = b
        b = c
    }
    return b
}

```

Stima della memoria

La memoria utilizzata è **costante** per qualunque numero n inserito come input, in quanto le variabili utilizzate sono sempre al massimo 5.

Stima del tempo

Calcoliamo tutte le operazioni elementari del programma, ottenendo $3 + 5(n - 2)$, ovvero $5n - 7$, **proporzionale al numero n dato in input.**

Algoritmo 5: potenza di matrici

Nonostante l'amplia ottimizzazione in tempo e memoria fatta finora possiamo sfruttare un ulteriore teorema matematico riguardante le matrici al fine di ottimizzare ulteriormente il nostro algoritmo.

Il teorema in questione è il seguente:

$$\text{Consideriamo } A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \text{ per ogni } n \geq 2 \text{ abbiamo che}$$

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F(n) & F(n-1) \\ F(n-1) & F(n-2) \end{pmatrix}$$

Sfruttando questo teorema è possibile costruire il seguente algoritmo:

```

int fib5(int n) {
    A = {{1, 1}, {1, 0}}
    for (int i = 2; i <= n; i++) {
        A = A × {{1, 1}, {1, 0}}
    }
    return A[1][1]
}

```

Utilizzando questo algoritmo in realtà non otteniamo **alcun miglioramento** in quanto le operazioni rimangono proporzionali ad n , mentre la memoria rimane costante, ma l'utilizzo di matrici ci consentirà di utilizzare una proprietà di queste in grado di velocizzare l'operazione di potenza.

Algoritmo 6: potenza di matrici velocizzata

La proprietà delle matrici che utilizziamo per velocizzare il tempo di esecuzione dal calcolo del numero di Fibonacci è la seguente:

$$A^n = \begin{cases} (A^{\frac{n}{2}})^2 & \text{n pari} \\ (A \times (A^{\frac{n-1}{2}})^2) & \text{n dispari} \end{cases}$$

L'algoritmo da utilizzare è dunque il seguente:

```

FibMat fibMatPow(int n) {
    A = {{1, 1}, {1, 0}}

    if (n > 1) {
        M = fibMatPow(n / 2)
        A = M × M
        if (n % 2 != 0)
            A = A × {{1, 1}, {1, 0}}
    }

    return A
}

int fib6(int n) {
    M = fibMatPow(n - 1)
    return M[1][1]
}

```

Stima del tempo

Il tempo di calcolo è dimostrabile essere **proporzionale a $\log_2 n$** .

Stima della memoria

La memoria è costante per ognuna delle $\log_2 n$ chiamate di procedura, dunque anch'essa è **proporzionale a $\log_2 n$** .

Sommario

Possiamo fare un resoconto in notazione asintotica riguardante gli algoritmi di calcolo dei numeri di Fibonacci appena visti:

Algoritmo	Tempo	Memoria
Fib2	$\Omega(2^{n/2})$	$O(n)$
Fib3	$O(n)$	$O(n)$
Fib4	$O(n)$	$O(1)$
Fib5	$O(n)$	$O(1)$
Fib6	$O(\log n)$	$O(\log n)$

Resoconto algoritmi di Fibonacci in termini di valore dell'input.

Solitamente però l'efficienza di un certo algoritmo viene valutata in termini di dimensione dell'input,

ovvero di numero di bit utilizzati per rappresentarlo, e non del valore effettivo che ha l'input.

Operando in questa maniera otteniamo il seguente resoconto:

Algorithm	Time	Space
Fib2	$\Omega(2^{ n })$	$O(2^{ n })$
Fib3	$O(2^{ n })$	$O(2^{ n })$
Fib4	$O(2^{ n })$	$O(1)$
Fib5	$O(2^{ n })$	$O(1)$
Fib6	$O(n)$	$O(n)$

Resoconto algoritmi di fibonacci in termini di dimensione dell'input.

▼ 3.0 - Notazione asintotica

La **notazione asintotica** permette di analizzare un algoritmo in base al suo tempo di calcolo e alla sua occupazione di memoria tenendo in considerazione solo la dimensione dell'input.

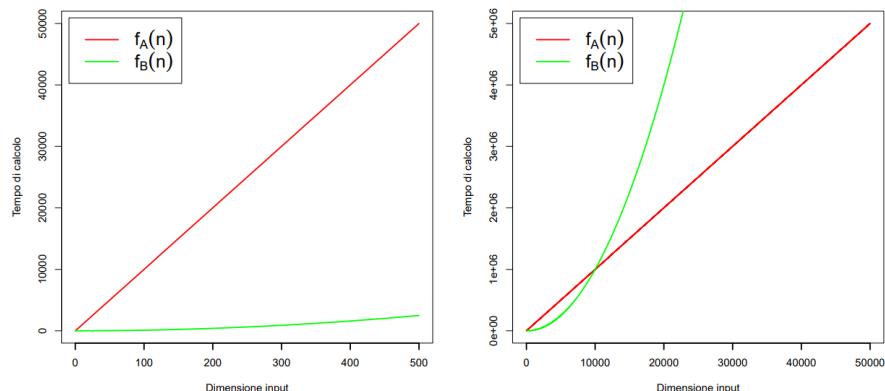
Tenendo in considerazione solamente il comportamento asintotico di un certo algoritmo si è in grado di analizzarlo in maniera corretta senza dover tenere in considerazione ad esempio i secondi o i MB utilizzati da tale algoritmo in quanto questi dipendono da fattori esterni come il linguaggio di programmazione utilizzato per implementarlo e la potenza di calcolo dell'elaboratore utilizzato per eseguirlo.

Il comportamento asintotico di un algoritmo non tiene conto di costanti additive/moltiplicative e termini di ordine inferiore all'interno della formula che mostra l'andamento dell'algoritmo.

Esempio:

- Consideriamo due algoritmi A e B per lo stesso problema.

Le funzioni che calcolano il tempo di esecuzione dei due algoritmi sono $10^2 n$ per A e $10^{-2} n^2$ per B.



I grafici di tali funzioni assumono la seguente forma.

▼ 3.1 - Notazioni

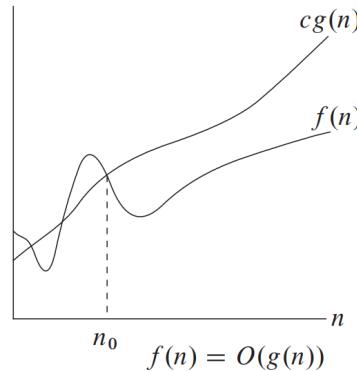
O-grande

Data una funzione $g(n)$, definiamo l'insieme di funzioni per cui $g(n)$ rappresenta un **limite asintotico superiore** come:

$$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) \leq cg(n)\}$$

Dalla definizione capiamo dunque che l'O-grande è riflessivo, in quanto $g(n) = O(g(n))$.

Inoltre, è importante sottolineare che da ora in avanti utilizzeremo l'abuso di notazione $f(n) = O(g(n))$ per indicare che $f(n) \in O(g(n))$.



Esempio grafico di O-grande.

o-piccolo

Data una funzione $g(n)$, definiamo l'insieme di funzioni **dominate asintoticamente** da $g(n)$ come:

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) < cg(n)\}$$

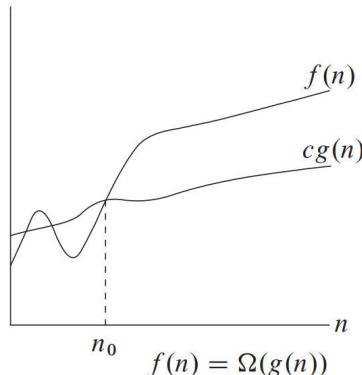
Per definizione $f(n) = o(g(n)) \implies f(n) = O(g(n))$.

Ω -grande

Data una funzione $g(n)$, definiamo l'insieme di funzioni per cui $g(n)$ rappresenta un **limite asintotico inferiore** come:

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) \geq cg(n)\}$$

Dalla definizione capiamo dunque che l' Ω -grande è riflessivo, in quanto $g(n) = \Omega(g(n))$.



Esempio grafico di Ω -grande.

ω -piccolo

Data una funzione $g(n)$, definiamo l'insieme di funzioni che **dominano asintoticamente** $g(n)$ come:

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \geq 0 \text{ tale che } \forall n \geq n_0, f(n) > cg(n)\}$$

Per definizione $f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$.

Θ

Data una funzione $g(n)$, definiamo l'insieme di funzioni **asintoticamente equivalenti** a $g(n)$ come:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 \geq 0 \text{ tale che } \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Dalla definizione capiamo dunque che l' Ω -grande è riflessivo, in quanto $g(n) = \Theta(g(n))$.

Teorema: $f(n) \in \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

Notazione asintotica e limiti

L'ordine di crescita asintotico di due funzioni può essere confrontato utilizzando i limiti:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = o(g(n)) \implies f(n) = O(g(n))$.
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$.
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0 \implies f(n) = \Theta(g(n))$.

Interpretazione intuitiva

Tabella di interpretazione del confronto tra crescita asintotica di funzioni in analogia con il confronto tra numeri reali.

Funzioni	Numeri reali
$f(n) = O(g(n))$	$f \leq g$
$f(n) = o(g(n))$	$f < g$
$f(n) = \Omega(g(n))$	$f \geq g$
$f(n) = \omega(g(n))$	$f > g$
$f(n) = \Theta(g(n))$	$f = g$

Comunque, a differenza di quanto avviene con il confronto tra numeri reali, non tutte le funzioni sono sempre confrontabili in crescita asintotica, ad esempio le due funzioni $f(n) = n$ e $g(n) = n^{\sin(n)+1}$ non sono confrontabili in quanto se $\sin(n) = -1$, $g(n)$ assume valore 1, mentre se $\sin(n) = 1$ assume valore n^2 .

Proprietà delle notazioni asintotiche

ϕ : notazione asintotica.

Transitività ($O, o, \Omega, \omega, \Theta$)

$$f(n) = \phi(g(n)) \wedge g(n) = \phi(h(n)) \implies f(n) = \phi(h(n)).$$

Riflessività (O, Ω, Θ)

$$f(n) = \phi(f(n)).$$

Simmetria (Θ)

$$f(n) = \phi(g(n)) \iff g(n) = \phi(f(n)).$$

Simmetria trasposta ($O \iff \Omega, o \iff \omega$)

$$f(n) = \phi(g(n)) \iff g(n) = \phi(f(n)).$$

Operazioni tra notazioni asintotiche

Somma

$$f_1(n) = \phi(g_1(n)) \wedge f_2(n) = \phi(g_2(n)) \implies f_1(n) + f_2(n) = \phi(f_1(n) + f_2(n)).$$

Prodotto

$$f_1(n) = \phi(g_1(n)) \wedge f_2(n) = \phi(g_2(n)) \implies f_1(n) \times f_2(n) = \phi(f_1(n) \times f_2(n)).$$

Moltiplicazione per una costante

$$f(n) = \phi(g(n)) \wedge c > 0 \implies c \times f(n) = \phi(g(n)).$$

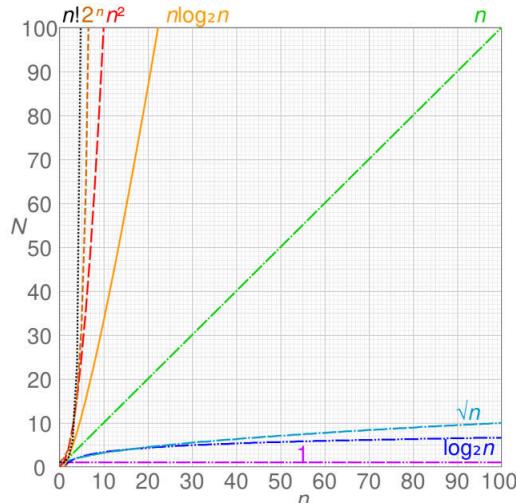
▼ 3.2 - Complessità computazionale

Ordini di crescita più comuni

Ordine di crescita	Nome
$O(1)$	Costante
$O(\log n)$	Logaritmico
$O(\log^k n)$	Polilogaritmico, $k \geq 1$
$O(n^k)$	Sublineare, $0 < k < 1$
$O(n)$	Lineare
$O(n \log n)$	Pseudolineare
$O(n^k)$	Polinomiale, $k > 1$
$O(n^2)$	Quadratico, per $k = 2$
$O(n^3)$	Cubico, per $k = 3$
$O(c^n)$	Esponenziale, base $c > 1$
$O(n!)$	Fattoriale
$O(n^n)$	Esponenziale, base n

Tabella degli ordini di crescita più comuni.

Nota: viene utilizzata la seguente notazione per i logaritmi: $\log n = \log_2 n$ e $\log^k n = (\log n)^k$.



Confronto tra gli ordini di crescita più comuni.

Complessità computazionale

Complessità computazionale di un algoritmo

Un algoritmo A ha complessità computazionale $\phi(f(n))$ rispetto ad una risorsa di calcolo se la quantità di risorse necessarie per eseguirlo su un qualsiasi input di dimensione n è $\phi(f(n))$.

Complessità computazionale di un problema

Un problema P ha complessità computazionale $\phi(f(n))$ rispetto ad una risorsa di calcolo se esiste un algoritmo che risolve P con una complessità computazionale $\phi(f(n))$ rispetto a tale risorsa di calcolo.

Analisi del caso ottimo, pessimo e medio

Spesso è necessario analizzare la complessità computazionale per quanto riguarda il caso **ottimo**, **pessimo** e **medio**.

Il caso ottimo descrive il comportamento dell'algoritmo in condizioni ottimali, ad esempio quando l'elemento cercato è il primo all'interno di una lista. Il caso pessimo descrive il comportamento in condizioni sfavorevoli, ad esempio quando l'elemento cercato è l'ultimo di una lista. Il caso medio descrive il comportamento su tutti i possibili input.

Quando si sviluppano algoritmi si è particolarmente interessati a migliorare le prestazioni nel caso pessimo e in quello medio.

Esempio:

- Analizziamo la complessità computazionale di diversi algoritmi per la ricerca di un elemento all'interno di un array.

Algoritmo 1: ricerca lineare

```
int linsearch(Array A[0 ... n], int x) {
    for (int i = 0; i < n, i++) {
```

```

        if A[i] == x then
            return i
    }
    return -1
}

```

- Caso **ottimo**: x è il primo elemento, $O(1)$.
- Caso **pessimo**: x è l'ultimo elemento, $\Theta(n)$.
- Caso **medio**:

La probabilità che x si trovi in una certa posizione i , compreso anche il caso in cui x non sia presente, è $P_i = \frac{1}{n+1}$.

Il tempo necessario per ispezionare la posizione i è $T_i = i$.

Per calcolare la complessità computazione media è dunque necessario sommare la probabilità di ispezione moltiplicata per il relativo tempo di ispezione:

$$\sum_{i=1}^n P_i T_i = \frac{1}{n+1} \sum_{i=1}^n i = \frac{1}{n+1} \frac{(n+1)(n+2)}{2} = \frac{n+2}{2} = \Theta(n).$$

Algoritmo 2: ricerca binaria

```

int binsearch(Array A[0 ... n], int x) {
    i = 1, j = n

    while (i ≤ j) {
        m = (i + j) / 2
        if A[m] == x then
            return m
        else if A[m] < x then
            i = m + 1
        else
            j = m - 1
    }

    return -1
}

```

- Caso **ottimo**: x si trova in posizione centrale, $O(1)$.
- Caso **pessimo**: x non è presente nell'array.

Dopo ogni iterazione lo spazio di ricerca viene dimezzato, dunque alla i -esima iterazione lo spazio di ricerca equivale a $\frac{n}{2^{i-1}}$.

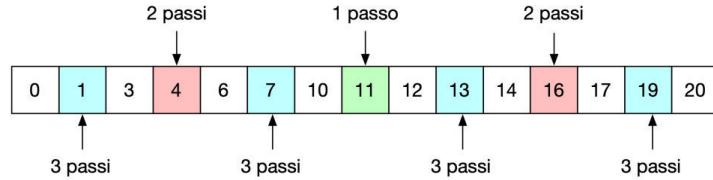
Il ciclo while termina quando lo spazio di ricerca è < 1 , dunque $\frac{n}{2^{i-1}} < 1 \implies n < 2^{i-1} \implies \log_2 n < \log_2 2^{i-1} \implies i > \log_2 n + 1$.

Il costo nel caso pessimo è dunque $\Theta(\log_2 n)$.

- Caso **medio**:

La probabilità che x si trovi in una certa posizione i , escluso il caso in cui x non sia presente per semplificare i calcoli, è $P_i = \frac{1}{n}$.

Il costo di accesso ad una certa posizione i dipende dalla sua posizione:



Siccome eseguiamo al massimo $\log_2 n$ passi, il costo medio è equivalente a:

$$\frac{1}{n} \sum_{i=1}^{\log_2 n} i 2^{i-1}$$

Tale formula è semplificabile nel seguente modo:

$$\frac{1}{n} \sum_{i=1}^{\log_2 n} i 2^{i-1} \leq \frac{\log_2 n}{n} \sum_{i=1}^{\log_2 n} 2^i = \frac{\log_2 n}{n} \times \frac{2^{\log_2 n+1} - 2}{2 - 1} = O(\log n).$$

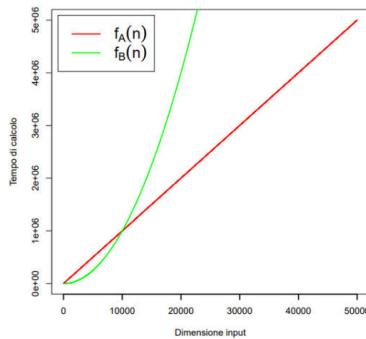
- Analizziamo la complessità computazionale di un algoritmo di ricerca del valore minimo all'interno di un array.

```
int min(Array A[0 ... n]) {
    m = 0
    for (int i = 1; i < n; i++) {
        if (A[i] < A[m]) m = i
    }
    return m
}
```

Il loop viene sempre eseguito $n - 1$ volte, dunque i casi ottimo, pessimo e medio coincidono e sono tutti $\Theta(n)$.

Scelta dell'algoritmo

Utilizziamo come esempio due algoritmi A e B che hanno complessità computazionale f_A e f_B , visibili nella seguente figura:



Rappresentazione grafica della complessità computazionale degli algoritmi A e B.

Nella **pratica** spesso viene utilizzato un algoritmo che controlla la dimensione dell'input e, in base a questa, sceglie se utilizzare il primo o il secondo al fine massimizzare le prestazioni in tutti i casi.

Nella **teoria** invece viene preso in considerazione solamente il comportamento asintotico della complessità computazionale, dunque nell'esempio appena fatto l'algoritmo A a discapito di quello B in quanto ha un costo lineare invece che esponenziale.

▼ 3.3 - Analisi ammortizzata

Molti algoritmi hanno un costo che dipende dallo stato attuale dell'input, dunque una data operazione può essere molto costosa in alcune situazioni e molto efficienti in altre, rendendo così molto difficile effettuare analisi probabilistiche per ricavarne la complessità. In questi casi occorre utilizzare l'**analisi ammortizzata** al fine di valutare il costo medio di una sequenza di operazioni.

Esistono due metodi utilizzati per l'analisi ammortizzata: il **metodo dell'aggregazione** e il **metodo degli accantonamenti**.

Vediamo come si comportano nell'analizzare il comportamento di un algoritmo di incremento che consente di incrementare un numero binario di 1 unità utilizzando un array per rappresentare il numero in input. Tale algoritmo è il seguente:

```
void increment(Array A[1 ... k])
    i = k
    while (i ≥ 1 and A[i] == 1) {
        A[i] = 0
        i = i - 1
    }
    if (i ≥ 1) // counter overflow
        A[i] = 1
```

Possiamo inoltre visualizzare in maniera grafica diverse operazioni di incremento di un numero binario il quale costo dipende dall'attuale stato del numero in input:

Valore	A[1]	A[2]	A[3]	A[4]	A[5]	Costo
0	0	0	0	0	0	0
1	0	0	0	0	1	1
2	0	0	0	1	0	2
3	0	0	0	1	1	1
4	0	0	1	0	0	3
5	0	0	1	0	1	1
6	0	0	1	1	0	2
7	0	0	1	1	1	1
8	0	1	0	0	0	4
9	0	1	0	0	1	1
10	0	1	0	1	0	2

Operazioni di incremento di numeri binari e relativo costo.

Metodo dell'aggregazione

Il **metodo dell'aggregazione** consiste nel determinare un limite superiore al costo totale di una sequenza di n operazioni per poi dividere tutto per n ed ottenere il costo medio di una singola operazione.

Notiamo dalla tabella sopra che il k -esimo bit viene cambiato ad ogni incremento, il $k-1$ -esimo bit ogni due incrementi, $k-2$ -esimo bit ogni 4 incrementi e così via. Tramite questa osservazione possiamo dunque costruire una formula per calcolare il costo totale di n operazioni: $n + \frac{n}{2} + \dots +$

$$\frac{n}{2^{k-1}} = \sum_{i=0}^{k-1} \frac{n}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = n \frac{1}{1 - \frac{1}{2}} = 2n.$$

Abbiamo dunque trovato che il costo totale di n operazioni è $O(n)$, e da questo possiamo trovare il costo ammortizzato per operazione dividendo per n : $\frac{O(n)}{n} = O(1)$.

Metodo degli accantonamenti

Il **metodo degli accantonamenti** è un metodo basato sulla contabilità economica. Consiste nel fare una previsione assegnando un **costo “ammortizzato”** per svolgere una singola operazione. In questo modo addebitiamo ogni operazione con il suo costo ammortizzato, salvando in un credito la differenza tra il suo costo ammortizzato e il costo reale. Il credito servirà poi in futuro per pagare operazioni che hanno un costo reale maggiore di quello ammortizzato. Capiamo che il costo ammortizzato scelto inizialmente è quello corretto se il credito non è mai negativo.

Utilizziamo tale metodo per calcolare il costo di una singola operazione della funzione increment. Assegniamo un costo ammortizzato di 2€ per cambiare ad 1 un bit con valore 0. Scegliamo proprio 2 per via del fatto che qualunque bit che viene trasformato da 0 a 1 verrà poi ritrasformato da 1 a 0 prima o poi, dunque ci servirà 1€ di credito per farlo. In questo modo, ogni volta che un bit viene trasformato da 1 a 0 viene accumulato un credito di 1€, dunque il credito residuo per una certa operazione è equivalente al numero di 1 presenti nel numero in input. Siccome il numero di 1 presenti in input non è mai negativo, allora neanche il credito sarà mai negativo, dunque abbiamo dimostrato che il costo ammortizzato di 2€ è esatto.

Siccome a ogni operazione abbiamo assegnato un costo di 2€, allora il costo totale è equivalente a $2n\text{€}$, e il costo ammortizzato è uguale a $\frac{2n}{n} = O(1)$.

Possiamo visualizzare in maniera grafica il metodo appena utilizzato:

Valore	A[1]	A[2]	A[3]	A[4]	A[5]	Credito residuo	Costo totale
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	2
2	0	0	0	1	0	1	4
3	0	0	0	1	1	2	6
4	0	0	1	0	0	1	8
5	0	0	1	0	1	2	10
6	0	0	1	1	0	2	12
7	0	0	1	1	1	3	14
8	0	1	0	0	0	1	16
9	0	1	0	0	1	2	18
10	0	1	0	1	0	2	20

Utilizzo del metodo degli accantonamenti nel costo ammortizzato dell'algoritmo increment.

▼ 3.4 - Equazioni di ricorrenza

Le **equazioni di ricorrenza** descrivono ogni elemento in una sequenza in termini degli elementi precedenti. Per questo motivo è possibile utilizzarle per determinare la crescita asintotica degli algoritmi ricorsivi.

Vedremo **3 modi** per risolvere equazioni di ricorrenza:

- Metodo dell'**iterazione**
- Metodo della **sostituzione**
- **Master Theorem**

Le equazioni di ricorrenza di algoritmi ricorsivi sono solitamente del tipo:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/3) + O(n) & n > 1 \end{cases}$$

Tipicamente vengono sostituite le notazioni asintotiche con espressioni positive:

$$T(n) = \begin{cases} d & n = 1 \\ 2T(n/3) + cn & n > 1 \end{cases}$$

Inoltre viene tipicamente utilizzata la costante 1 al posto delle costanti simboliche:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/3) + n & n > 1 \end{cases}$$

Metodo dell'iterazione

Il **metodo dell'iterazione** consiste nel sostituire in modo iterativo la parte ricorsiva dell'equazione finchè non appare uno schema sintetizzabile tramite una formula. A questo punto occorre chiedersi quando la ricorrenza termina e in questo modo si riesce a determinare la crescita asintotica dell'algoritmo ricorsivo.

Esempio:

- Utilizziamo il metodo dell'iterazione nell'equazione di ricorrenza:

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

Sostituendo la parte ricorsiva in modo iterativo otteniamo:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= T(n/8) + c + c \\ &\dots \\ &= T(n/2^i) + c \cdot i \end{aligned}$$

La ricorrenza termina quando $n/2^i = 1 \implies i = \log_2 n$.

Quindi $T(n) = T(1) + c \cdot \log_2 n = 1 + c \cdot \log_2 n = \Theta(\log n)$.

Come abbiamo visto in precedenza possiamo sostituire la costante c con 1, il risultato non cambierebbe.

Metodo della sostituzione

Il **metodo della sostituzione** consiste nell'ipotizzare una soluzione e nel cercare di validare tale ipotesi effettuando una dimostrazione tramite induzione.

Esempio:

- Utilizziamo il metodo della sostituzione nell'equazione di ricorrenza:

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

Ipotizziamo che $T(n) = O(n)$, ovvero che $\exists c > 0, n_0 \geq 0$ tale che $\forall n \geq n_0. T(n) \leq cn$.

◦ $n = 1$

$$T(1) \leq c \cdot 1 \implies 1 \leq c, \text{ vero } \forall c \geq 1.$$

- $n > 1$

Assumiamo per ipotesi induttiva che l'ipotesi fatta sia vera per $T(n/2)$, quindi $T(n/2) \leq c \cdot (n/2)$.

Dobbiamo provare che $T(n) \leq c \cdot n \implies T(n/2) + n \leq c \cdot n$.

Per ipotesi induttiva abbiamo quindi che $c \cdot (n/2) + n \leq c \cdot n \implies (c/2 + 1) \cdot n \leq c \cdot n \implies c/2 + 1 \leq c$, vero $\forall c \geq 2$.

L'ipotesi fatta è dunque corretta, quindi $T(n) = O(n)$.

- Utilizziamo il metodo della sostituzione nell'equazione di ricorrenza di Fibonacci:

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

Ipotizziamo che $T(n) = O(2^n)$, ovvero che $\exists c > 0, n_0 \geq 0$ tale che $\forall n \geq n_0. T(n) \leq c \cdot 2^n$.

- $n \leq 2$

$T(1) \leq c \cdot 2^1 \implies 1 \leq c \cdot 2$, vero $\forall c \geq \frac{1}{2}$.

$T(2) \leq c \cdot 2^2 \implies 1 \leq c \cdot 4$, vero $\forall c \geq \frac{1}{4}$.

- $n > 2$

Assumiamo per ipotesi induttiva che l'ipotesi fatta sia vera per $T(n-1)$ e $T(n-2)$, quindi $T(n-1) \leq c \cdot 2^{n-1}$ e $T(n-2) \leq c \cdot 2^{n-2}$.

Dobbiamo provare che $T(n) \leq c \cdot 2^n \implies T(n-1) + T(n-2) + 1 \leq c \cdot 2^n$.

Per ipotesi induttiva abbiamo quindi che $c \cdot 2^{n-1} + c \cdot 2^{n-2} + 1 \leq c \cdot 2^n \implies c \cdot 2 \cdot 2^{n-2} + c \cdot 2^{n-2} + 1 \leq c \cdot 2^n \implies c \cdot 2^{n-2}(2+1) + 1 \leq c \cdot 2^n$.

$\forall n \geq 2 - \log_2 c$ possiamo arrivare alla forma $c \cdot 2^{n-2}(2+2) \leq c \cdot 2^n \implies c \cdot 2^n \leq c \cdot 2^n$, vero $\forall c \in \mathbb{R}$.

L'ipotesi fatta è quindi corretta, dunque $T(n) = O(2^n)$.

È possibile inoltre dimostrare tramite lo stesso metodo che la ricorrenza di Fibonacci è limitata inferiormente da $\Omega(\sqrt{2}^n)$, ma è difficile trovare dei limiti più stretti andando a tentativi con il metodo della sostituzione.

Master Theorem

Il Master Theorem è un approccio per risolvere ricorrenze della forma

■

con $a \geq 1$ e $b > 1$ costanti e $f(n)$ asintoticamente positiva. Il costo di ogni chiamata ricorsiva è dato da $f(n)$.

Si consideri la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} d & n = 1 \\ aT(n/b) + cn^\beta & n > 1 \end{cases}$$

dove $a \geq 1, b > 1$ e c, d costanti.

Sia $\alpha = \log_b a = \frac{\log a}{\log b}$. Allora

- Se $\alpha > \beta$ allora $T(n) = \Theta(n^\alpha)$
- Se $\alpha = \beta$ allora $T(n) = \Theta(n^\alpha \log n)$
- Se $\alpha < \beta$ allora $T(n) = \Theta(n^\beta)$

Esempio:

- Utilizziamo il Master Theorem per individuare la crescita asintotica del seguente algoritmo di ricerca binaria:

```
int binsearch(Array A[1 ... n], int x, int i, int j)
    if (i > j) return -1
    else {
        m = (i + j) / 2
        if (A[m] == x)
            return m
        else if (A[m] > x)
            return search(A, x, i, m - 1)
        else
            return search(A, x, m + 1, j)
    }
```

Dallo pseudocodice ricaviamo che la funzione di ricorrenza è la seguente:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n/2) + 1 & n > 0 \text{ (ricerca su } 1/2 \text{ dello spazio)} \end{cases}$$

$\alpha = \log_2 1 = 0$ e $\beta = 0 \implies \alpha = \beta \implies T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

▼ 4.0 - Algoritmi di ordinamento

Il problema dell'ordinamento consiste nei seguenti input e output.

- **Input:** una sequenza di n numeri $[a_1, \dots, a_n]$
- **Output:** una permutazione $p : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ degli indici degli elementi della sequenza in input al fine di ottenere un'altra sequenza $[a_{p(1)}, \dots, a_{p(n)}]$ tale che $a_{p(1)} \leq \dots, \leq a_{p(n)}$.

Discuteremo dei seguenti algoritmi di ordinamenti e dei loro andamenti asintotici:

- **Incrementali:** SelectionSort, InsertionSort
- **Divide et impera:** MergeSort, QuickSort
- **Non-comparativi:** CountingSort, RadixSort

Nozioni preliminari

Assumiamo di utilizzare algoritmi di ordinamenti per array in cui ogni elemento è composto da:

- Una **chiave**, confrontabili tra loro rispetto a $\leq, =, \geq$.
- Un **valore**, il quale rappresenta il contenuto associato alla chiave.

Tali algoritmi ordineranno quindi gli array rispetto alla chiave e non al valore, e in molti casi l'array in input conterrà solamente la chiave.

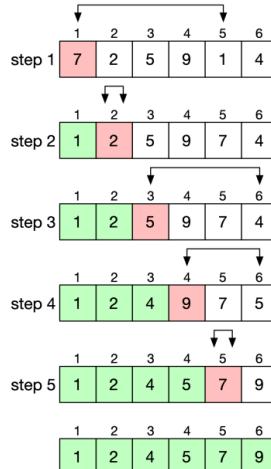
Inoltre gli algoritmi di ordinamento possono presentare due **proprietà**:

- **In place:** l'algoritmo riordina gli array utilizzando solamente l'array in input, senza bisogno di array di sostegno.
- **Stabile:** valori che hanno la stessa chiave appaiono nell'array in output nello stesso ordine in cui si trovavano nell'array in input.

▼ 4.1 - Algoritmi incrementali

SelectionSort

Dato un array A di dimensione n , ad ogni passo i da 1 a $n - 1$, viene cercata la posizione j corrispondente alla minima chiave nel sottoarray $A[i, \dots, n]$ e viene scambiato l'elemento $A[j]$ con $A[i]$.



Esempio grafico di SelectionSort.

Lo pseudocodice dell'algoritmo di SelectionSort è il seguente:

```

void selectionsort(Array A[1, ..., n]) {
    for (int i = 0; i < n - 1; i++) {
        m = i

        for (int j = i + 1; j < n; j++) {
            if (A[j] < A[m]) m = j
        }

        if (m != i) swap(A, i, m)
    }

    void swap(Array A[1, ..., n], int i, int j) {
        tmp = A[i]
        A[i] = A[j]
        A[j] = tmp
    }
}

```

Complessità computazionale di SelectionSort

Notiamo che l'implementazione del SelectionSort utilizza due cicli, dei quali il primo viene eseguito esattamente $n - 1$ volte, mentre il secondo viene eseguito $n - i$ volte per ogni passo i . La funzione swap che viene richiamata all'interno dei due cicli invece ha costo costante $O(1)$.

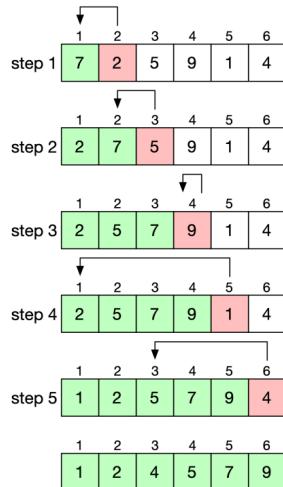
Siccome sappiamo che i due cicli vengono eseguiti per intero ogni volta che l'algoritmo viene eseguito, allora il caso ottimo, medio e pessimo possono essere condensati in un unico:

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

Possiamo dunque concludere che l'algoritmo di ordinamento **SelectionSort** ha **complessità quadratica** sulla lunghezza dell'array.

InsertionSort

Dato un array A di dimensione n , ad ogni passo i da 2 a n , il sottoarray $A[1, \dots, i-1]$ risulta ordinato e viene inserito l'elemento $A[i]$ nella corretta posizione in $A[1, \dots, i]$.



Esempio grafico di InsertionSort.

Lo pseudocodice dell'algoritmo InsertionSort è il seguente:

```

void insertionsort(Array A[1, ... , n]) {
    for (int i = 1; i < n; i++) {
        j = i
        while (j > 0 && A[j] < A[j - 1]) {
            swap(A, j, j - 1)
            j = j - 1
        }
    }
}

```

Complessità computazionale di InsertionSort

Notiamo che anche in questo caso l'algoritmo è formato da due cicli annidati, dei quali il primo viene sempre eseguito interamente $n - 1$ volte, mentre il secondo potrebbe anche non venire eseguito. Dividiamo dunque l'analisi nel caso ottimo, pessimo e medio.

Nel **caso ottimo** le chiavi nell'array sono già ordinate dalla più piccola alla più grande, dunque il secondo ciclo non viene mai eseguito. Abbiamo dunque che la complessità è $\Theta(n)$, costo **lineare**.

Il caso ottimo si verifica anche per array in input **quasi ordinati**, in quanto se A è ordinato per tutti tranne k elementi, allora, visto che il secondo ciclo viene eseguito solo su questi k elementi, la complessità totale sarà $\Theta(n)$ (ciclo for) + $O(nk)$ (ciclo while) = $O(nk)$. Quindi abbiamo che se il numero k di elementi non ordinati è costante rispetto ad n , allora il costo di InsertionSort rimane **lineare**: $\Theta(n)$.

Nel **caso pessimo** le chiavi nell'array sono ordinate dalla più grande alla più piccola, dunque il secondo ciclo viene eseguito ogni volta $i - 1$ volte. Calcoliamo dunque la complessità tramite la

$$\text{sommatoria } \sum_{i=2}^n i - 1 = \sum_{i=2}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2), \text{ costo quadratico.}$$

Per il **caso medio** occorre fare delle assunzioni probabilistiche. Sappiamo infatti che il ciclo while viene eseguito al minimo 0 volte e al massimo $i - 1$, dunque possiamo assumere che questo viene eseguito in media $\frac{i-1}{2}$ volte. Il costo totale nel caso medio è dunque calcolabile tramite la

$$\text{sommatoria } \sum_{i=2}^n \frac{i-1}{2} = \frac{1}{2} \sum_{i=2}^{n-1} i = \frac{n(n-1)}{4} = \Theta(n^2), \text{ costo quadratico.}$$

▼ 4.2 - Algoritmi divide et impera

Quella del **divide et impera** è una strategia per ottenere controllo militare e politico: fare in modo che gli avversari siano divisi e si combattano tra di loro.

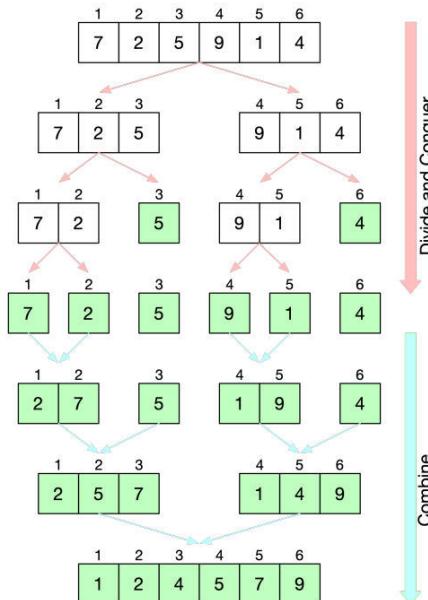
Negli algoritmi la strategia del divide et impera consiste nei seguenti step:

- **Divide**: dividere il problema in sotto-problemi più piccoli.
- **Conquer**: risolvere i sotto-problemi in maniera ricorsiva.
- **Combine**: fondere le soluzioni dei sotto-problemi per ottenere quella del problema originario.

MergeSort

Gli step divide et impera del **MergeSort** sono i seguenti:

1. **Divide**: dividere l'array in input $A[1, \dots, n]$ in due metà $A_1 = A[1, \dots, \frac{1+n}{2}]$ e $A_2 = A[\frac{1+n}{2} + 1, \dots, n]$.
2. **Conquer**: richiamare ricorsivamente l'algoritmo su A_1 e A_2 se hanno lunghezza > 1 , in quanto array con lunghezza $= 1$ sono già ordinati.
3. **Combine**: combinare i due array ordinati A_1, A_2 in un unico array ordinato.



Esempio grafico di MergeSort.

Lo pseudocodice dell'algoritmo di MergeSort è il seguente:

```

void mergesort(Array A[1, ... , n], int p, int r) {
    if (p < r) {
        q = (p + r) / 2
        mergesort(A, p, q)
        mergesort(A, q + 1, r)
        merge(A, p, q, r)
    }
}

void merge(Array A[1, ... , n], int p, int q, int r) {
    B = A[1, ... , r - p + 1]
    i = p
    j = q + 1
    k = 1
}
  
```

```

while (i <= q && j <= r) {
    if (A[i] <= A[j]) {
        B[k] = A[i]
        i = i + 1
    } else {
        B[k] = A[j]
        j = j + 1
    }
    k = k + 1
}

while (i <= q) {
    B[k] = A[i]
    k = k + 1
    i = i + 1
}

while (j <= r) {
    B[k] = A[j]
    k = k + 1
    j = j + 1
}

for (int k = 0; k++; k < r - p + 1) {
    A[p + k - 1] = B[k]
}
}

```

Complessità computazionale di MergeSort

Non è necessario comprendere alla perfezione l'algoritmo di **merge**, ci basta sapere che esso ha un **costo lineare** sulla lunghezza $r - p + 1$. Questo costo viene dal fatto che in ognuno dei tre cicli while viene incrementato il valore di i oppure di j , mai insieme, e i viene incrementato da 1 a q , mentre j va da $q + 1$ a r , dunque vengono effettuate $r - p + 1$ iterazioni. Notiamo che anche il ciclo for viene eseguito $r - p + 1$ volte.

Dopo aver analizzato il costo dell'algoritmo merge ($\Theta(n)$) possiamo ricavare l'**equazione di ricorrenza di mergesort**:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

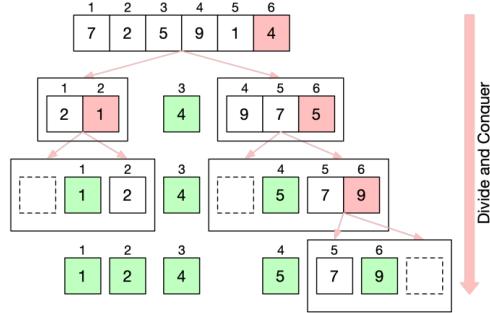
Possiamo dunque ricavare il costo tramite l'utilizzo del Master Theorem: $\Theta(n \log n)$, ovvero **pseudologaritmico**.

Inoltre notiamo che il costo di mergesort non dipende da come i numeri sono inizialmente organizzati all'interno dell'array, dunque il caso ottimo, medio e pessimo coincidono.

QuickSort

Gli step divide et impera del **QuickSort** sono i seguenti:

1. **Divide**: partizionare l'array in input $A[1, \dots, n]$ in due sottoarray $A_1 = A[1, \dots, q - 1]$ e $A_2 = A[q + 1, \dots, n]$ tali che tutte le chiavi in A_1 e A_2 siano rispettivamente \leq e $>$ della chiave $A[q]$ scelta durante il partizionamento.
2. **Conquer**: richiamare in modo ricorsivo l'algoritmo su A_1 e A_2 .
3. **Combine**: non è necessario ricombinare A_1 e A_2 in quanto già ordinati.



Esempio grafico di QuickSort.

Lo pseudocodice dell'algoritmo QuickSort è il seguente:

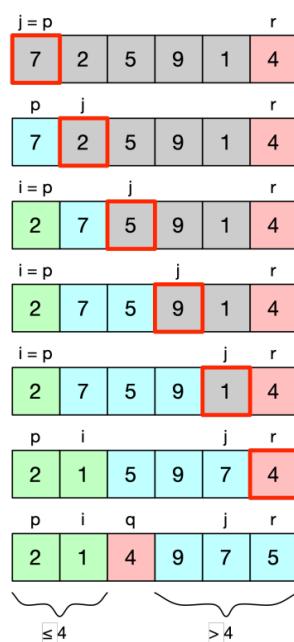
```

void quicksort(Array A[1, ... , n], int p, int r) {
    if (p < r) {
        q = partition(A, p, r)
        quicksort(A, p, q - 1)
        quicksort(A, q + 1, r)
    }
}

int partition(Array A[1, ... , n], int p, int r) {
    x = A[r] // scelta deterministica del pivot
    i = p - 1
    for (j = p, ... , r - 1) {
        if (A[j] ≤ x) {
            swap(A, i + 1, j)
            i = i + 1
        }
    }
    swap(A, i + 1, r) // move the pivot in A[i + 1]
    return i + 1
}

```

Osserviamo che la funzione partition contiene un unico ciclo che viene sempre eseguito da p a $r-1$, dunque essa ha un costo nel caso ottimo, medio e pessimo equivalente a $\Theta(r - p + 1)$.



Esempio grafico di partition.

Il costo computazionale della funzione QuickSort dipende invece dalla scelta del pivot, la quale causa il bilanciamento/sbilanciamento dei sottoarray, basati sulla loro lunghezza. Distinguiamo dunque il costo di QuickSort in caso ottimo, pessimo e medio:

- **Caso ottimo:** i due sottoarray sono entrambi lunghi $n/2$.

L'equazione di ricorrenza è dunque la seguente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

Possiamo utilizzare il Master Theorem per calcolare che il costo nel caso ottimo è **pseudologaritmico** $\Theta(n \log n)$.

- **Caso pessimo:** un array di lunghezza 0 e l'altro $n - 1$.

L'equazione di ricorrenza è dunque la seguente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n - 1) + T(0) + n & n > 1 \end{cases}$$

Possiamo risolverla tramite il metodo iterativo:

$$\begin{aligned} T(n) &= T(n - 1) + T(0) + n \\ &= T(n - 1) + 1 + n \\ &= T(n - 2) + 2 + (n - 1) + n \\ &= T(n - 3) + 3 + (n - 2) + (n - 1) + n \\ &\dots \\ &= T(n - i) + i + \sum_{k=0}^{i-1} n - k \end{aligned}$$

La ricorsione termina per $n - i = 0$, ovvero $i = n$. Sostituiamo dunque n a i per trovare il costo del caso pessimo: $1 + n + \sum_{k=0}^{n-1} n - k = 1 + n + \sum_{k=1}^n k = 1 + n + \frac{n(n+1)}{2} = \Theta(n^2)$.

- **Caso medio:** l'equazione di ricorrenza è la seguente:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(i) + T(n - i - 1) + n & n > 1 \end{cases}$$

Osservando che i e $n - i - 1$ possono cambiare ad ogni chiamata ricorsiva allora dobbiamo fare un'assunzione probabilistica pensando al fatto che tutte le partizioni sono equiprobabili e costruendo dunque la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \sum_{i=0}^{n-1} \frac{T(i) + T(n - i - 1)}{n} + n & n > 1 \end{cases}$$

Notando che $\sum_{i=0}^{n-1} T(n - i - 1) = \sum_{i=0}^{n-1} T(i)$, in quanto una sommatoria è equivalente all'altra al contrario, possiamo semplificare l'equazione di ricorrenza ottenendo:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n & n > 1 \end{cases}$$

Infine utilizziamo il metodo di sostituzione per dimostrare (slide 34 del pacco di slide "Algoritmi di ordinamento") che $T(n) \leq cn \ln n$, e potendo dunque concludere che QuickSort nel caso medio ha un costo **pseudologaritmico**: $O(n \log n)$.

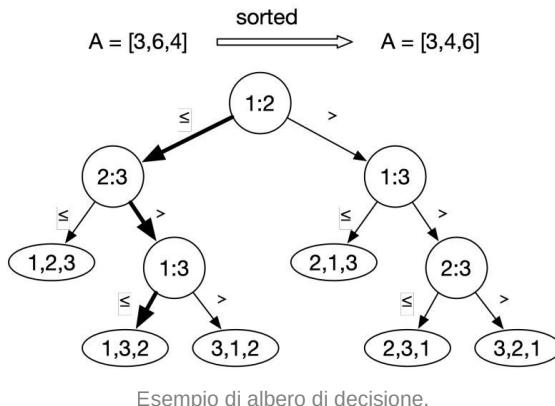
Nell'algoritmo di QuickSort che abbiamo scritto e analizzato viene scelto come pivot sempre l'ultimo elemento dell'array. Questo, nel caso in cui l'array di partenza è quasi ordinato, comporta spesso il caso pessimo, e nella realtà array quasi ordinati sono più probabili di array totalmente disordinati. Lasciando questa scelta del pivot dunque si ricadrebbe più spesso nel caso pessimo. Per risolvere questo problema è possibile scegliere il pivot in maniera **randomica**, causando più spesso il caso medio.

▼ 4.3 - Algoritmi non comparativi

Lower bound per ordinamento comparativo

Tutti gli algoritmi che abbiamo descritto finora sono considerabili **algoritmi comparativi**, ovvero che si basano sul confronto tra valori.

Ciascun algoritmo comparativo può essere descritto tramite un **albero di decisione**, ossia un albero binario in cui ogni nodo corrisponde al confronto tra due argomenti dell'array.



Esempio di albero di decisione.

Osservazioni sull'albero di decisione:

- L'esecuzione dell'algoritmo dato un determinato array in input corrisponde dunque ad un percorso radice-foglia.
- L'altezza dell'albero corrisponde numero di confronti nel caso pessimo, mentre l'altezza media corrisponde al numero di confronti nel caso medio.
- Il numero di foglie è almeno uguale a $n!$, ossia il numero di permutazioni dell'array di partenza.

Teorema altezza albero di decisione

Sia T_k un albero binario con k foglie in cui ogni nodo ha esattamente due figli e sia $h(T_k)$ l'altezza di T_k . Allora $h(T_k) \geq \log k$.

Dimostrazione slide 41 del pacco di slide "Algoritmi di ordinamento".

Teorema lower bound per ordinamento comparativo

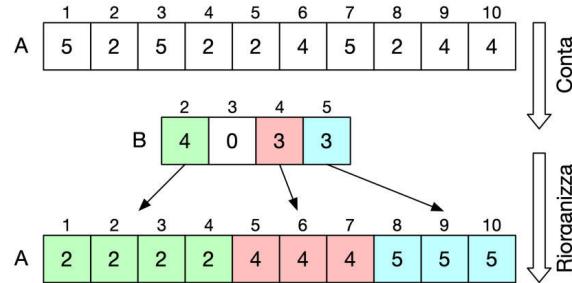
Ogni algoritmo di ordinamento comparativo richiede $\Omega(n \log n)$ confronti nel caso pessimo.

Dimostrazione: siccome abbiamo visto che ogni albero di decisione ha almeno $n!$ foglie, utilizzando il teorema precedente otteniamo che ogni albero di decisione ha un'altezza $\Omega(\log n!) = \Omega(n \log n)$, e tale altezza corrisponde ai confronti da effettuare nel caso pessimo.

Tale lower bound appena dimostrato vale però solo per algoritmi di tipo comparativo. Vediamo dunque due algoritmi che utilizzano una struttura di tipo non comparativa.

CountingSort

L'algoritmo di **CountingSort** deriva dall'idea che nell'array A dato in input sono contenuti interi contenuti in un certo intervallo $[a, b]$. È possibile dunque creare un secondo array B di lunghezza $b - a + 1$ in cui per ogni intero nell'intervallo $[a, b]$ viene inserito il numero di volte che compare all'interno dell'array A . Fatto ciò vengono riorganizzati tali valori all'interno dell'array A .



Esempio grafico di CountingSort.

Lo pseudocodice dell'algoritmo di CountingSort è il seguente:

```

void countingsort(Array A[1, ... , n]) {
    a = min(A)
    b = max(A)
    k = b - a + 1
    B[1, ... , k]
    // initialize B
    for (i = 1, ... , k)
        B[i] = 0
    // insert values into B
    for (i = 1, ... , n)
        B[A[i] - a + 1] = B[A[i] - a + 1] + 1
    // insert ordered values into A
    j = 1
    for (i = 1, ... , k) {
        while (B[i] > 0) {
            A[j] = i + a - 1
            B[i] = B[i] - 1
            j = j + 1
        }
    }
}

```

Complessità computazionale di CountingSort

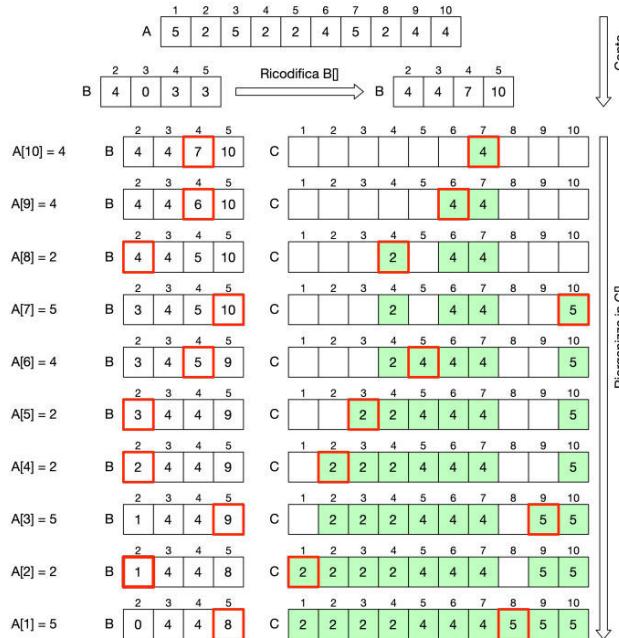
Osserviamo che le funzioni min e max costano entrambe $\Theta(n)$, il primo ciclo for viene eseguito k volte, il secondo ciclo for viene eseguito n volte, infine gli ultimi cicli annidati for e while vengono

eseguiti n volte in quanto devono inserire in A tutti gli n numeri. Concludiamo dunque che il caso ottimo, medio e pessimo coincidono e hanno un costo equivalente a $\Theta(n + k)$.

Notiamo dunque che il costo dell'algoritmo dipende sia dal numero di elementi nell'array in input ma anche da k , ovvero la dimensione del range $[a, b]$.

CountingSort stabile

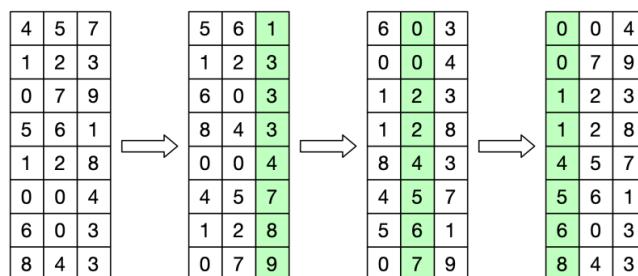
L'algoritmo di CountingSort che abbiamo appena analizzato non è stabile, ma è comunque possibile effettuare delle piccole modifiche all'algoritmo per renderlo stabile e mantenere i costi asintotici.



Esempio grafico di CountingSort stabile.

RadixSort

L'algoritmo di **RadixSort** consiste nell'ordinare i valori dell'array dato in input prima rispetto alla cifra meno significativa, poi rispetto alla penultima cifra meno significativa e così via.



Esempio grafico di RadixSort.

È importante notare che l'algoritmo di ordinamento utilizzato per ordinare ogni cifra dei numeri dati input deve essere **stabile**, altrimenti al termine dell'esecuzione i numeri potrebbero non risultare ordinati:

Esempio di errore causato dall'utilizzo di un algoritmo non stabile.

Lo pseudocodice dell'algoritmo di RadixSort è il seguente:

```
void radixsort(Array A[1, ... , n])
    d = max key length
    for (i = 1, ... , d)
        // ordinamento stabile della i-esima cifra delle chiavi
```

Complessità computazionale di RadixSort

La complessità computazionale di RadixSort dipende dall'algoritmo di ordinamento utilizzato per ordinare le cifre delle chiavi.

Possiamo ad esempio pensare di utilizzare l'algoritmo di CountingSort stabile in quanto la sua complessità dipende dal range delle chiavi, dunque se le chiavi dell'array in input sono interi allora il range massimo sarà 10, mentre se le chiavi sono delle stringhe il range massimo sarà equivalente al numero di caratteri ammessi nella stringa. Il costo dell'algoritmo di RadixSort con l'utilizzo del CountingSort equivale dunque a $\Theta(d(n + k))$, dove d rappresenta il numero di cifre massime delle chiavi in input.

Osserviamo inoltre che se $k = O(n)$ e d è un valore costante, allora il costo è $\Theta(n)$, quindi lineare sul numero degli elementi dati in input.

Riassunto

Il riassunto dei costi e delle proprietà degli algoritmi di ordinamento che abbiamo analizzato è dunque rappresentato nella seguente tabella:

	Caso ottimo	Caso medio	Caso pessimo
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
QuickSort	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n^2)$
CountingSort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$
RadixSort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$

	In place	Stabile
SelectionSort	Si	Si
InsertionSort	Si	Si
MergeSort	No	Si
QuickSort	Si	No
CountingSort	No	Si
RadixSort	No	Si

Riassunto dei costi e delle proprietà degli algoritmi di ordinamento.

- n : numero di elementi da ordinare.
- k : ampiezza del range di valori chiave.
- d : numero massimo di cifre in una chiave.

▼ 5.0 - Strutture dati elementari

Le **strutture dati** descrivono come i dati sono logicamente organizzati e le operazioni per accedervi e modificarli. Nessuna struttura dati descrive quali dati è in grado di memorizzare, ma solo la maniera in cui tali dati vengono memorizzati, in quanto ogni struttura dati è in grado di memorizzare qualunque tipologia di dato.

All'interno di questa sezione verranno analizzate le seguenti quattro **tipologie** di strutture dati elementari:

- **Dizionario** (Dictionary)
- **Liste concatenate** (Linked list)
- **Pile** (Stack)
- **Coda** (Queue)
- **Alberi** (Tree)

Prototipo vs implementazione

È importante distinguere il **prototipo** dall'**implementazione** di una struttura dati in quanto il primo descrive solo la struttura e le operazioni della struttura dati, permettendo al programmatore di implementarla e all'utente di capire come usarla. L'implementazione consiste invece nella realizzazione della struttura dati in un certo linguaggio di programmazione, ed è da quest'ultima che solitamente dipendono i tempi di esecuzione.

Classi di strutture dati

È possibile individuare diverse classi nelle quali suddividere le strutture dati in base alle loro proprietà:

- **Lineari**: i dati vengono memorizzati in ordine sequenziale (primo, secondo, terzo ecc.).
- **Non lineari**: non esiste un ordine sequenziale in cui i dati vengono memorizzati.
- **Statiche**: il numero degli elementi memorizzati al suo interno rimane costante.
- **Dinamiche**: il numero degli elementi memorizzati al suo interno può variare in maniera dinamica.
- **Omogenee**: un solo tipo di dato può essere memorizzato al suo interno.
- **Eterogenee**: differenti tipi di dato possono essere memorizzati al suo interno.

▼ 5.1 - Dizionario con array

Un **dizionario** è una struttura dati dinamica che consente di memorizzare oggetti che presentano una chiave e un valore. Le chiavi all'interno di un dizionario devono essere univoche, mentre i valori possono anche essere duplicati.

Operazioni basilari di un dizionario:

- **search(Key k)**: ritorna l'oggetto associato alla chiave k, se la chiave non è contenuta ritorna null.
- **insert(Key k, Data d)**: se la chiave non è già contenuta aggiunge la coppia (k, d) al dizionario, mentre se è già contenuta sostituisce solo il vecchio dato con d.
- **delete(Key k)**: rimuove l'oggetto con chiave k dal dizionario.

Implementazione con array non ordinato

L'idea è quella di implementare il dizionario tramite un array che contiene le coppie (Key, Data) non tenendo conto dell'ordine delle chiavi.

Search

Viene svolta una ricerca della chiave tramite un algoritmo di ricerca lineare e viene ritornato l'oggetto corrispondente oppure null.

```
Data search(Dict D, Key k) {
    i = linsearch(D.A, D.size, k)

    if (i != 1) return D.A[i].data
    else return NIL
}

int linsearch(Array A[1, ..., m], int n, Key k) {
    for (i = 1, ..., n) {
        if (A[i].key == k) return i
    }
    return -1
}
```

Costo **ottimo** (chiave nell'ultima posizione dell'array o non trovata): $O(1)$.

Costo **medio** (costo medio della ricerca lineare): $\Theta(n)$.

Costo **pessimo** (chiave nella prima posizione dell'array): $\Theta(n)$.

Insert

Controlla tramite ricerca lineare se la chiave è già presente nell'array. In caso affermativo sostituisce i dati, altrimenti inserisce la coppia (Key, Data) nella prima posizione libera.

```
void insert(Dict D, Key k, Data d) {
    i = linsearch(D.A, D.size, k)
    if (i == -1) {
        D.size = D.size + 1
        i = D.size
    }
    D.A[i].key = k
    D.A[i].data = d
}
```

Costo **ottimo**: $O(1)$.

Costo **medio e pessimo**: $\Theta(n)$.

Delete

Cerca tramite ricerca lineare se la chiave è presente nell'array. In caso affermativo rimuove la coppia (Key, Data) e sposta di una posizione a sinistra tutte le coppie dopo Key.

```
void delete(Dict D, Key k) {
    i = linsearch(D.A, D.size, k)
    if (i != -1)
        leftshift(D.A, D.size, i)
    D.size = D.size - 1
}

void leftshift(Array A[1, ..., m], int n, int i) {
    for (j = i, ..., n - 1)
        A[j] = A[j + 1]
}
```

Notiamo che il costo di linsearch + leftshift è equivalente a $\Theta(i) + \Theta(n - i) = \Theta(n)$, dunque il caso **ottimo**, **medio** e **pessimo** coincidono e sono uguali a: $\Theta(n)$.

Implementazione con array ordinato

L'idea è quella di implementare il dizionario sempre tramite un array ma di mantenere gli oggetti al suo interno ordinati in base al valore delle chiavi.

Tutti gli algoritmi per le operazioni search, insert e delete rimangono dunque uguali tranne per il fatto che viene utilizzata la ricerca binaria al posto di quella lineare e che nell'insert vengono spostati parte degli elementi a destra se il nuovo elemento viene inserito in posizione centrale.

A seguito di questa modifica i costi diventano dunque i seguenti:

- **search**

Costo ottimo: $O(1)$.

Costo medio e pessimo: $O(\log n)$.

- **insert**

Costo **ottimo**: $O(\log n)$. Questo perchè la posizione ottimale di inserimento è quella in fondo a destra, in quanto poi non sarà necessario spostare nessun elemento a destra. Per trovare però l'ultima posizione a destra tramite ricerca binaria occorrono $\log n$ passi.

Costo **medio**: $O(n)$. Assumiamo infatti che ogni posizione sia equiprobabile, notiamo che la ricerca binaria è asintoticamente logaritmica, mentre in media vengono shiftati $n/2$ elementi.

Costo **pessimo**: $\Theta(n)$. Questo perchè il caso pessimo è quello in cui l'elemento deve essere inserito nella prima posizione dell'array. In questo caso la ricerca binaria svolge $\log n$ passi e lo shift n passi.

- **delete**

Costo **ottimo**: $O(\log n)$.

Costo **medio**: $O(n)$.

Costo **pessimo**: $\Theta(n)$.

Notiamo dunque che in generali i costi delle operazioni elementari di un dizionario utilizzando un array ordinato sono migliorate. Tali costi sono riassunti nella seguente tabella:

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Tabella dei costi asintotici delle operazioni elementari dei dizionari implementati tramite array ordinati e non.

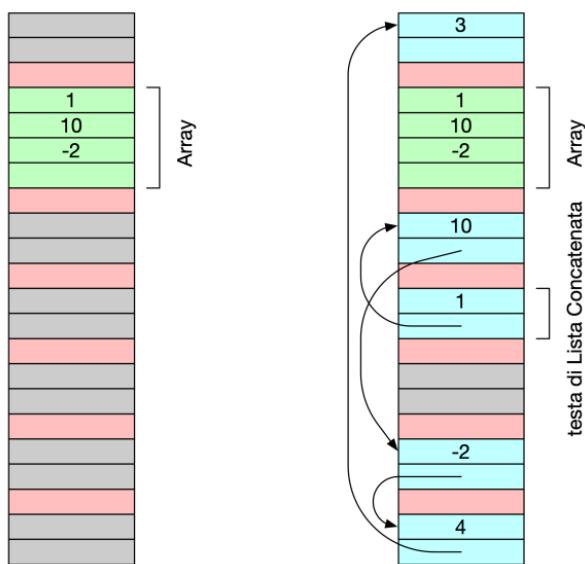
▼ 5.2 - Lista

Una **lista** è una struttura dati in cui tutti gli elementi sono organizzati in ordine sequenziale.

Operazioni basilari di una lista:

- **search(Key k)**: ritorna l'oggetto associato alla chiave k, se la chiave non è contenuta ritorna null.
- **insert(Key k, Data d)**: se la chiave non è già contenuta aggiunge la coppia (k, d) al dizionario, mentre se è già contenuta sostituisce solo il vecchio dato con d.
- **delete(Key k)**: rimuove l'oggetto con chiave k dal dizionario.

È possibile implementare una lista sia utilizzando **array** che **liste concatenate**. Nel primo caso si ha una dimensione allocata in maniera statica, ma l'accesso agli elementi della lista è più veloce in quanto avviene tramite l'utilizzo degli indici. Nel secondo caso invece la lista viene implementata tramite un insieme di puntatori, dunque la dimensione è dinamica e ne viene allocata di nuova su richiesta ma il costo dell'accesso agli elementi della lista dipende dalla loro posizione. Notiamo inoltre dalla seguente figura che per memorizzare una lista tramite un array occorre avere uno spazio libero contiguo in memoria, mentre ciò non è necessario se si utilizza una lista concatenata.



Visualizzazione grafica della memoria per una lista implementata tramite array e lista concatenata.

Lista concatenata semplice

Analizziamo ora il costo delle operazioni basilari di una lista implementata tramite l'utilizzo di una lista concatenata semplice.

Search

```
Node search(SLLList L, Key k) {
    tmp = L.head
    while (tmp != null) {
        if (tmp.key == k)
            return tmp
        tmp = tmp.next
    }
    return null
}
```

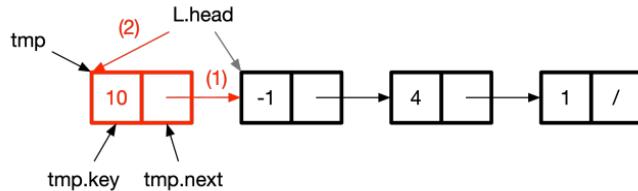
Costo **ottimo**: $O(1)$.

Costo **medio e pessimo**: $\Theta(n)$.

Insert

Possono essere utilizzati due metodi di inserimento in una lista concatenata, `headInsert` e `tailInsert`. Come suggeriscono i loro nomi il primo inserisce il nuovo elemento in testa alla lista, mentre il secondo in coda.

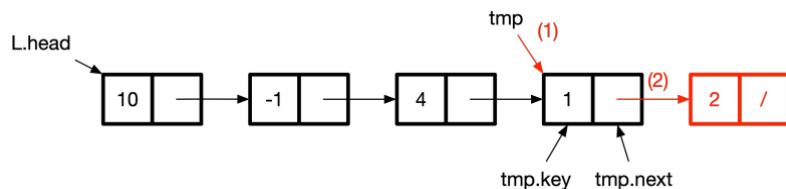
```
void headInsert(SLLList L, Key k) {
    tmp = new Node(k)
    tmp.next = L.head
    L.head = tmp
}
```



Visualizzazione grafica di `headInsert`.

Costo ottimo, medio e pessimo: $O(1)$.

```
void tailInsert(SLLList L, Key k) {
    if (L.head == null)
        L.head = new Node(k)
    else
        tmp = L.head
        while (tmp.next != null)
            tmp = tmp.next
        tmp.next = new Node(k)
}
```



Visualizzazione grafica di `tailInsert`.

Costo ottimo, medio e pessimo: $\Theta(n)$.

Delete

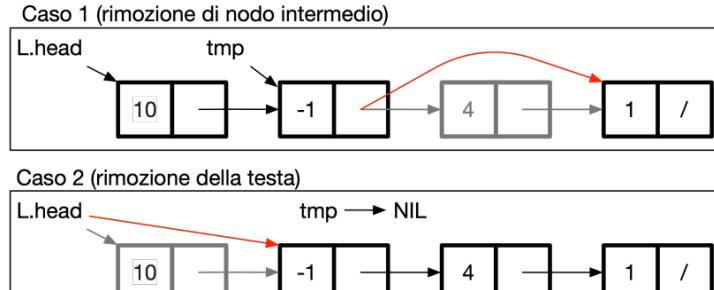
```
void delete(SLLList L, Key k) {
    tmp = searchPrev(L, k)
    if (tmp != null)
        tmp.next = tmp.next.next
    else if (L.head != null && L.head.key == k)
        L.head = L.head.next
}

void searchPrev(SLLList L, Key k) {
    prev = null
    curr = L.head
    while (curr != null) {
        if (curr.key == key)
            return prev
        prev = curr
    }
}
```

```

        curr = curr.next
    }
    return null
}

```



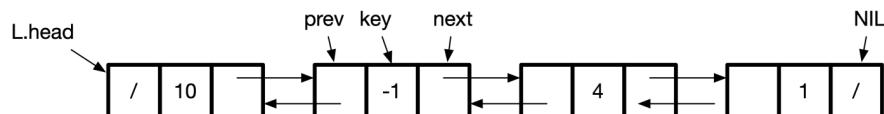
Visualizzazione grafica di delete.

Caso ottimo: $O(1)$.

Caso medio e pessimo: $\Theta(n)$.

Lista doppiamente concatenata

Una **lista doppiamente concatenata** è una lista concatenata semplice in cui ogni nodo contiene anche un puntatore al nodo precedente della lista. Il nodo in testa alla lista ha valore null nel puntatore al nodo precedente.

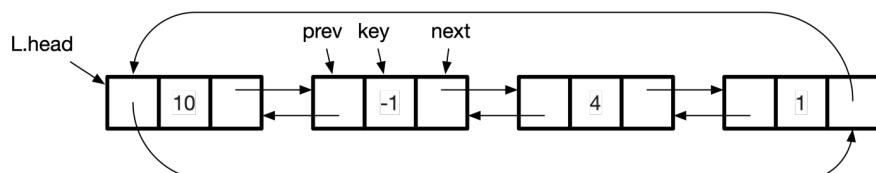


Esempio grafico di lista doppiamente concatenata.

Notiamo dunque che può essere visitata in entrambe le direzioni. I costi computazionali corrispondono a quelli di una lista concatenata semplice e l'unico pseudocodice che viene modificato è quella della funzione delete, la quale non necessita della ricerca del nodo precedente per via della presenza del puntatore ad esso.

Lista concatenata circolare

Una **lista concatenata circolare** è una lista doppiamente concatenata in cui il puntatore a next dell'ultimo nodo punta al primo nodo e il puntatore a prev del primo nodo punta all'ultimo nodo.



Esempio grafico di lista concatenata circolare.

Notiamo che anch'essa può essere visitata in entrambe le direzioni e l'accesso alla coda della lista è più veloce, ma diventa più complesso visitare la lista in quanto il nodo in coda non ha valore null

come puntatore al nodo successivo. I costi computazioni corrispondono dunque a quelli di una lista concatenata semplice tranne per la funzione di tailInsert, la quale assume un costo costante $O(1)$.

Lista con puntatori a testa e coda

Una lista con puntatori a testa e coda è una lista concatenata semplice o doppiamente concatenata in cui vengono mantenuti i puntatori alla testa e alla coda della lista.

In questo modo vengono velocizzate le operazioni di accesso alla testa della lista senza rendere più complessa la visita della lista, come avveniva per le liste concatenate circolari.

Riassunto dei costi

Type	SEARCH	INSERT (testa)	INSERT (coda)	DELETE
Concatenata semplice	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Doppiamente concatenata	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Circolare	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Puntatori testa e coda	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse varianti di liste concatenate.

Dizionario con lista concatenata

È possibile implementare un dizionario utilizzando anche una lista concatenata, in modo da evitare la shift che domina i costi del dizionario implementato con array. Purtroppo però utilizzando le liste c'è necessità di svolgere la ricerca del nodo per ogni operazione, dunque i costi vengono dominati da quest'ultima e non si ottengono miglioramenti rispetto agli array ordinati.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 5.3 - Pila

Una **pila** è una struttura dati di tipo **LIFO (Last In First Out)** che supporta due operazioni basilari:

- **push**: aggiunge un nuovo elemento alla pila.
- **pop**: elimina l'elemento aggiunto più di recente e lo restituisce.

Una pila consente l'accesso solo all'ultimo elemento, e può avere applicazioni in diversi ambiti, come nella gestione dei record di attivazione, nei linguaggi stack oriented e nell'undo/redo degli editor di testo.

Implementazione di una pila

Una pila può essere implementata in diversi modi, ad esempio.

- **Lista concatenata**

Pro: dimensione illimitata.

Contro: overhead di memoria (occorre memorizzare anche i puntatori ai nodi della lista).

- **Array**

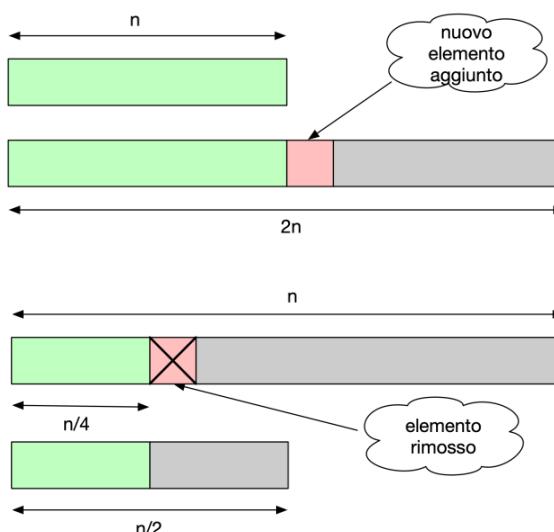
Pro: nessun overhead di memoria (non occorre memorizzare anche i puntatori ai nodi della lista).

Contro: dimensione limitata

In entrambe le implementazioni i **costi** delle funzioni push e pop rimangono $O(1)$.

Implementazione con array dinamico

È possibile implementare una pila tramite un array e mantenere il pro di avere una dimensione illimitata tramite l'utilizzo di un array dinamico, il quale adatta la sua dimensione al numero degli elementi contenuti in esso. In genere viene adottata la seguente strategia: viene raddoppiata la dimensione dell'array quando non c'è più spazio libero e viene dimezzata quando l'occupazione è di $1/4$.



Aggiunta e rimozione di un elemento in un array dinamico.

L'implementazione di un array dinamico prevede la copia di tutti gli elementi presenti all'interno di esso nel nuovo array quando questo viene creato per aumentare o diminuire la dimensione dell'array iniziale.

Lo pseudocodice per le operazioni di aggiunta e di rimozione di un elemento da un array dinamico è dunque il seguente:

```
void push(Stack S, Int x) {
    if (S.top == S.length) {
        // ridimensionamento dell'array
        n = S.length
        T = new array[1, ... , 2n]
        for (i = 1, ... , n)
            T[i] = S.stack[i]
        S.stack = T
        S.length = 2n
    }
    // push
    S.top = S.top + 1
    S.stack[S.top] = x
}
```

```

int pop(Stack S) {
    if (S.top == 0)
        error "underflow"
    else
        // pop
        e = S.stack[S.top]
        S.top = S.top - 1
        if (S.top <= [S.length / 4]) {
            // ridimensionamento dell'array
            n = S.length
            T = new array[1, ..., n / 2]
            for (i = 1, ..., n / 4)
                T[i] = S.stack[i]
            S.stack = T
            S.length = [n / 2]
        }
    return e
}

```

Notiamo che il costo computazionale di entrambe le funzioni è equivalente e distinguiamo il **caso ottimo**, ovvero quando l'array non è ancora pieno per l'operazione di push e quando l'array non è utilizzato per più di $1/4$ nell'operazione di pop, in cui il costo è costante, ovvero $O(1)$, e il caso pessimo, ovvero l'opposto del caso precedente, nel quale i costi vengono dominati dall'operazione di copia di tutti gli elementi all'interno del nuovo array, dunque si ottiene un costo lineare, ovvero $O(n)$.

Possiamo inoltre approfondire l'analisi del costo di tali funzioni effettuando un **analisi ammortizzata** al fine di comprendere quanto sia il costo di n push/pop partendo da una pila vuota.

Calcoliamo dunque il costo di una sequenza di n push utilizzando il metodo dell'aggregazione e otteniamo che il costo ammortizzato è costante, ovvero $O(1)$.

È possibile effettuare la stessa analisi per l'operazione di pop, e utilizzando il metodo degli accantonamenti otteniamo anche qui che il costo ammortizzato di n operazioni di pop è costante, ovvero $O(1)$.

▼ 5.4 - Coda

Una **coda** è una struttura dati di tipo **FIFO (First In First Out)** che supporta due operazioni basilari:

- **enqueue**: aggiunge un elemento in fondo alla coda.
- **dequeue**: rimuove l'elemento in testa alla coda.

Una coda non consente l'accesso agli elementi interni ad essa, e intuitivamente rappresenta una fila di elementi, ad esempio una file di persone in attesa di un servizio.

Implementazione di una coda

Una coda può essere implementata in diversi modi, ad esempio:

- **Lista concatenata circolare**

Pro: dimensione illimitata.

Contro: overhead di memoria (2 puntatori per ogni nodo).

- **Lista con puntatori a testa e coda**:

Pro: dimensione illimitata.

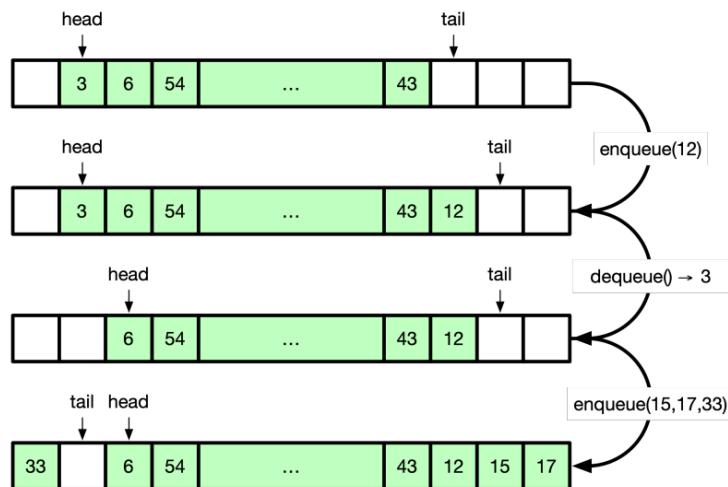
Contro: overhead di memoria.

- **Array circolari**

Gli array circolari permettono, nel caso in cui la coda sia giunta all'ultima posizione dell'array, di sfruttare le posizioni iniziali dell'array nel caso in cui queste siano libere.

Pro: nessun overhead di memoria.

Contro: dimensione limitata.



Implementazione con array circolari

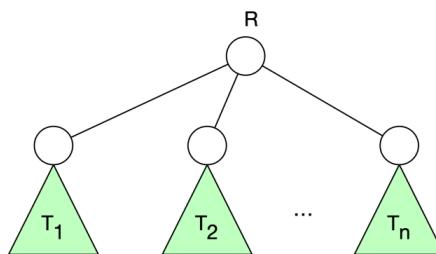
In tutti e 3 le implementazioni i **costi** delle operazioni di enqueue e dequeue rimangono $O(1)$.

▼ 5.5 - Albero

Un **albero** è una struttura dati non lineare ad albero gerarchico. Esso contiene un insieme di nodi e un insieme di archi che connettono i nodi, ed esiste un solo percorso per andare da un nodo all'altro.

Un albero si dice ordinato se i figli di ogni nodo sono ordinati, ovvero se è possibile identificare un ordine di tale figli (es. primo figlio, secondo figlio ecc.)

Un albero si dice **radicato** se uno dei suoi nodi è identificato come radice. È possibile dare una definizione ricorsiva di albero radicato, dicendo che esso è un insieme vuoto di nodi oppure una radice R e zero o più alberi disgiunti le cui radici sono connesse ad R.



Esempio grafico di albero radicato.

Alcune definizioni

La **profondità** di un nodo è la lunghezza del percorso che va dalla radice a tale nodo.

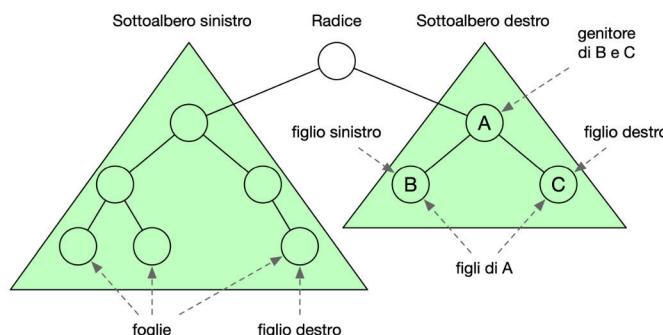
Un **livello** è l'insieme di tutti i nodi alla stessa profondità.

L'**altezza** di un albero è la sua massima profondità.

Il **grado** di un nodo è il numero dei suoi figli.

Albero binario

Un **albero binario** è un albero ordinato in cui ogni nodo ha al massimo due figli, e tali figli vengono identificati come destro e sinistro.



Esempio grafico di albero binario.

Un albero binario è **completo** se ogni nodo intermedio ha due figli.

Un albero binario è **perfetto** se è completo e tutte le foglie hanno la stessa profondità.

Proprietà fondamentale di un albero

Ogni albero non vuoto con n nodi ha esattamente $n - 1$ archi.

Algoritmi di visita su alberi

Un algoritmo di visita su albero consente di visitare tutti i nodi di una struttura dati albero.

Esistono principalmente due tipologie di visite su albero:

- **Visita in profondità (DFS - Depth First Search)**

La ricerca va in profondità il più possibile prima di visitare il nodo successivo sullo stesso livello.

Esistono tre tipologie di visite in profondità, ovvero pre-ordine, post-ordine e in-ordine.

- **Visita in ampiezza (BFS - Breadth First Search)**

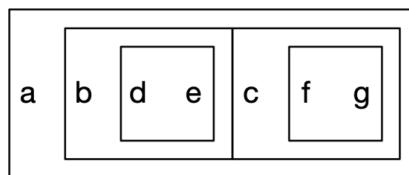
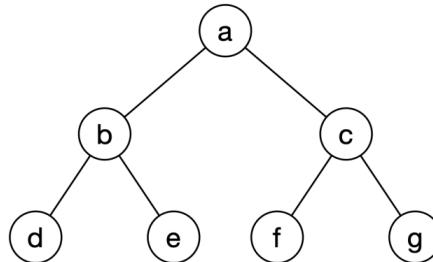
Vengono visitati tutti i nodi appartenenti ad un livello prima di passare a quello successivo.

Visita in profondità: pre-ordine.

```

void preorder(Node T) {
    if (T != null) {
        visit(T)
        preorder(T.left)
        preorder(T.right)
    }
}

```



Esempio di visita pre-ordine.

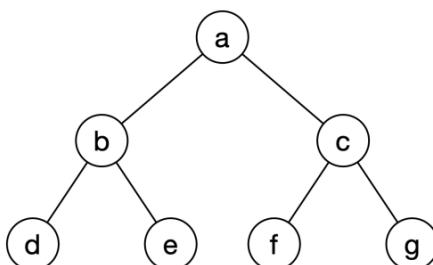
Assumendo che visit abbia un costo costante, la funzione preorder ha un costo computazionale equivalente a $\Theta(n)$ (n = numero di nodi).

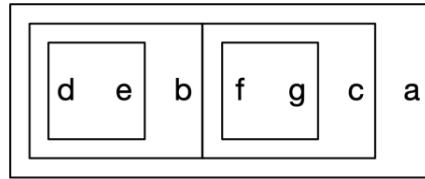
Visita in profondità: post-ordine

```

void postorder(Node T) {
    if (T != null) {
        postorder(T.left)
        postorder(T.right)
        visit(T)
    }
}

```

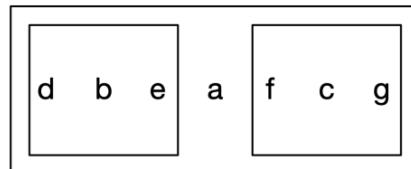
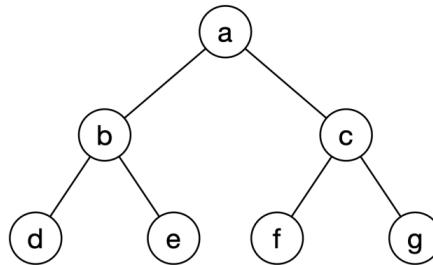




Esempio di visita post-ordine.

Visita in profondità: in-ordine

```
void inorder(Node T) {
    if (T != null) {
        postorder(T.left)
        visit(T)
        postorder(T.right)
    }
}
```



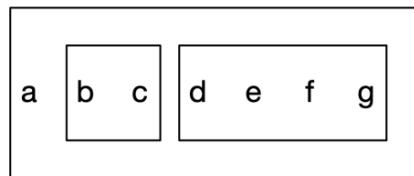
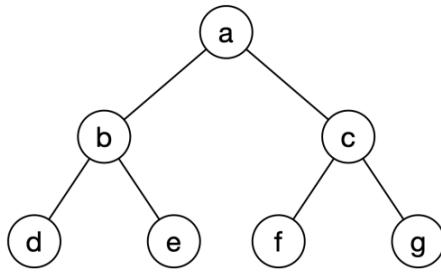
Esempio di visita in-ordine.

Assumendo che visit abbia un costo costante, la funzione inorder ha un costo computazionale equivalente a $\Theta(n)$ ($n = \text{numero di nodi}$).

Visita in ampiezza

È possibile utilizzare una coda per imporre un ordine di visita per livello.

```
void BFS(Tree T) {
    Q = new Queue
    if (T.root != null)
        enqueue(Q, T.root)
    while (Q.size != 0) {
        x = dequeue(Q)
        visit(x)
        if (x.left != null)
            enqueue(Q, x.left)
        if (x.right != null)
            enqueue(Q, x.right)
    }
}
```



Esempio di visita in ampiezza.

Assumendo che visit abbia un costo costante, la funzione BFS ha un costo computazionale equivalente a $\Theta(n)$ (n = numero di nodi).

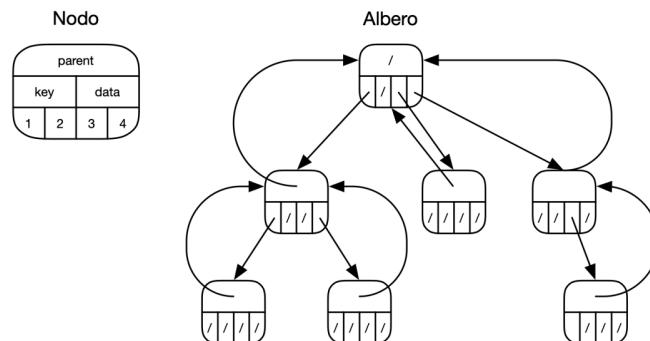
Visita su alberi non binari

Le tipologie di visite che abbiamo appena visto possono essere generalizzate ad alberi non binari. Solo per la visita in-ordine sono richieste specifiche aggiuntive, infatti occorre specificare su quanti nodi figli richiamare la funzione prima di visitare il nodo corrente.

Implementazione di un albero non binario

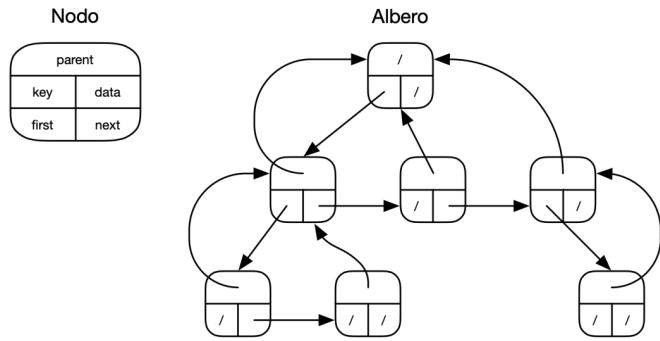
È possibile implementare un albero non binario in diversi modi.

Un esempio è quello di utilizzare per ogni nodo un array di puntatori a k figli. Il lato negativo però risiede nel fatto che si rischia di sprecare spazio se molti nodi hanno meno di k figli.



Implementazione con array di puntatori.

Per risolvere questo problema si può pensare di utilizzare per ogni nodo un puntatore al primo nodo figlio e al fratello successivo.



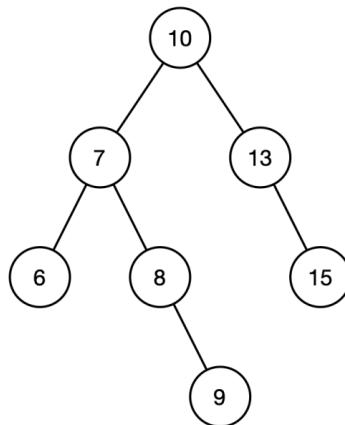
Implementazione con puntatore al fratello successivo.

Utilizzando queste implementazioni i costi delle funzioni presentate in precedenza rimangono equivalenti a $\Theta(n)$.

▼ 6.0 - Alberi

▼ 6.1 - Alberi binari di ricerca

Gli **alberi binari di ricerca (BST - Binary Search Tree)** sono alberi binari radicati in cui ogni nodo presenta una chiave confrontabile e dati associati alla chiave. La peculiarità di tale albero è il fatto di avere, per ogni nodo v , nel sottoalbero sinistro chiavi $\leq v.key$ e nel sottoalbero destro chiavi $\geq v.key$. Tale proprietà consente di effettuare una **ricerca binaria** sull'albero, in modo da diminuire i costi computazionali delle operazioni basilari.



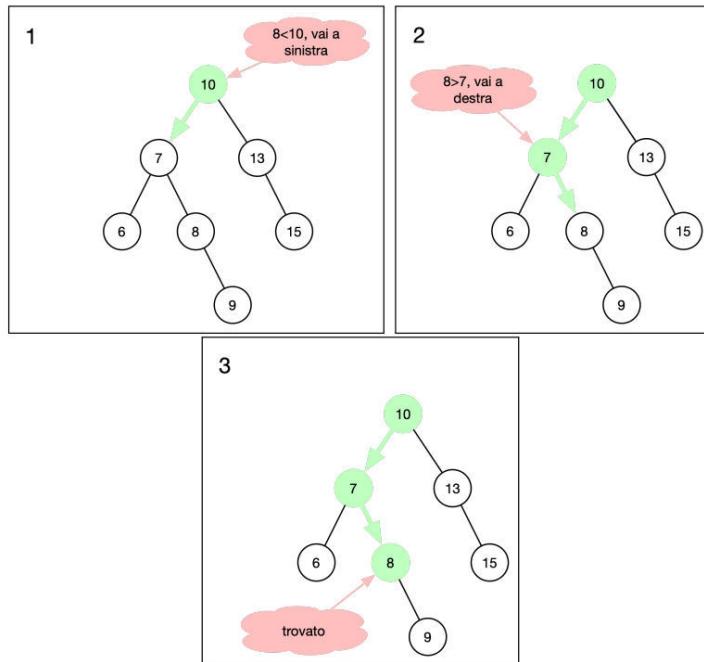
Esempio di albero binario di ricerca.

Le operazioni basilari di un BST sono le seguenti:

- **search(BST T, Key k)**: ritorna il nodo con chiave k in T, null se k non appare in T.
- **max(BST T)**: ritorna il nodo con chiave massima in T.
- **min(BST T)**: ritorna il nodo con chiave minima in T.
- **predecessor(BST T)**: ritorna il nodo avente la più grande chiave $\leq T.key$ se $T.key$ non è la chiave minima in T, altrimenti ritorna null.
- **successor(BST T)**: ritorna il nodo avente la più piccola chiave $\geq T.key$ se $T.key$ non è la chiave massima in T, altrimenti ritorna null.
- **insert(BST T, Key k, Data d)**: inserisce un nodo con chiave k e dati d in T.
- **delete(BST T, Key k)**: rimuove il nodo con chiave k in T.

Search

L'idea è quella di utilizzare la ricerca binaria.



Esempio di search in BST.

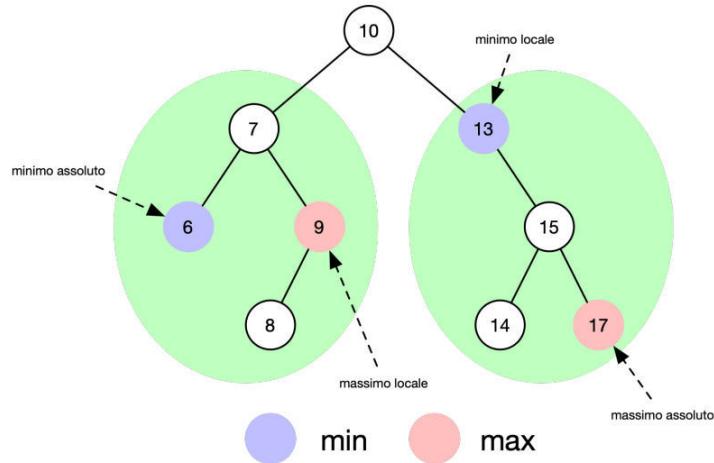
Lo **pseudocodice** di search è il seguente:

```
Node search(BST T, Key k) {
    tmp = T.root
    while (tmp != NIL) {
        if (k == tmp.key)
            return tmp
        else if (k < tmp.key)
            tmp = tmp.left
        else
            tmp = tmp.right
    }
    return NIL
}
```

Il **costo computazionale** di search è il seguente:

- Caso **ottimo**, quando $k = T.key$: $O(1)$.
- Caso **peggiore**, quando k si trova nel livello h (h : altezza dell'albero): $O(h)$.

Max e min



Esempio di massimo/minimo locale/assoluto.

Dato un albero binario T , il nodo massimo è il nodo **più a destra** in T , mentre il nodo minimo è quello **più a sinistra** in T .

Lo **pseudocodice** di max e min è il seguente:

```

Node max(BST T) {
    while (T != null && T.right != null)
        T = T.right
    return T
}

Node min(BST T) {
    while (T != null && T.left != null)
        T = T.left
    return T
}

```

Il **costo computazionale** di max e min è il seguente:

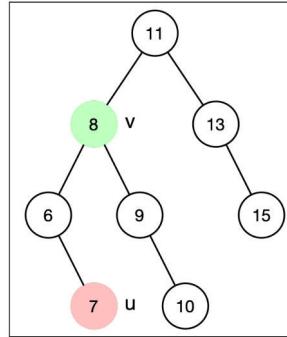
- Caso **ottimo**, quando, nel caso di max, T non ha il figlio destro, oppure, nel caso di min, T non ha il figlio sinistro: $O(1)$.
- Caso **pesimistico**, quando il massimo/minimo si trova nel livello h (h : altezza dell'albero): $O(h)$.

Predecessor

Per calcolare il predecessore di un nodo all'interno di un albero binario bisogna distinguere due casi:

- **Caso 1: il nodo ha un figlio sinistro.**

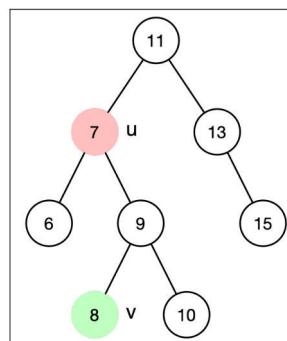
Il predecessore è il nodo con chiave massima nel sottoalbero sinistro del nodo dato in input.



Esempio di predecessore nel caso 1 (v: nodo dato in input, u: predecessore).

- **Caso 2: il nodo non ha un figlio sinistro.**

Il predecessore è il primo antenato tale che il nodo dato in input stia nel suo sottoalbero destro.



Esempio di predecessore nel caso 2 (v: nodo dato in input, u: predecessore).

Lo **pseudocodice** di predecessor è il seguente:

```

Node predecessor(Node T) {
    if (T == null)
        return NIL
    else if (T.left != null)
        // case 1
        return max(T.left)
    else {
        // case 2
        P = T.parent
        while (P != null && T == P.left) {
            T = P
            P = P.parent
        }
        return P
    }
}

```

Il **costo computazionale** di predecessor è il seguente:

- Caso **ottimo**, quando nel caso 1 il figlio sinistro è il massimo del sottoalbero che ha tale nodo come radice, oppure nel caso 2 quando il nodo corrente è un figlio destro: $O(1)$.
- Caso **pessimo**, quando nel caso 1 il nodo corrente è la radice e il predecessore si trova nel livello h (h : altezza dell'albero), oppure nel caso 2 quando il nodo corrente si trova nel livello h e l'unico sottoalbero destro a cui appartiene è quello della radice: $O(h)$.

Successor

L'idea per trovare il successore di un nodo è simmetrica a quella del predecessore, dunque lo **pseudocodice** è il seguente:

```

Node successor(Node T) {
    if (T == null)
        return NIL
    else if (T.right != null)
        // case 1
        return min(T.right)
    else {
        // case 2
        P = T.parent
        while (P != null && T == P.right) {
            T = P
            P = P.parent
        }
        return P
    }
}

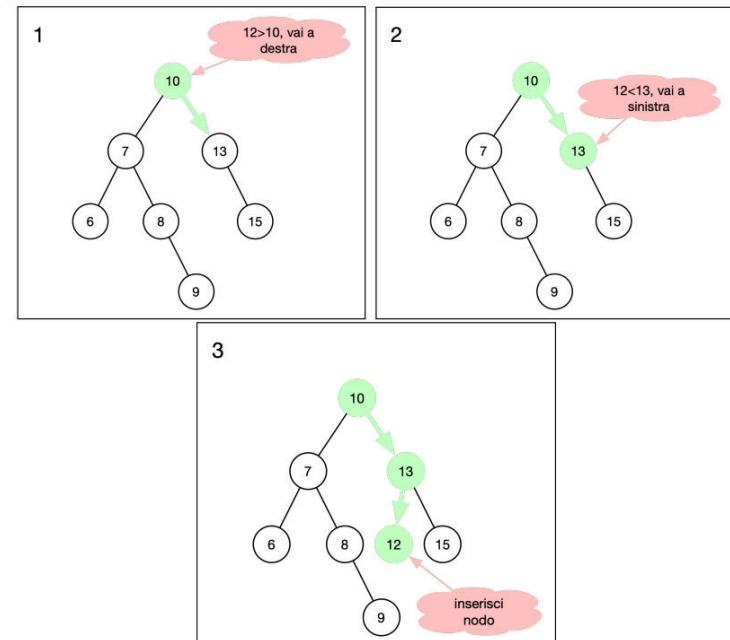
```

Il **costo computazionale** di successor è il seguente:

- Caso **ottimo**, quando nel caso 1 il figlio destro è il minimo del sottoalbero che ha tale nodo come radice, oppure nel caso 2 quando il nodo corrente è un figlio sinistro: $O(1)$.
- Caso **pessimo**, quando nel caso 1 il nodo corrente è la radice e il successore si trova nel livello h (h : altezza dell'albero), oppure nel caso 2 quando il nodo corrente si trova nel livello h e l'unico sottoalbero sinistro a cui appartiene è quello della radice: $O(h)$.

Insert

L'idea è quella di cercare la posizione in cui inserire il nuovo nodo tramite ricerca binaria.



Esempio di insert in BST.

Lo **pseudocodice** di insert è il seguente:

```

void insert(BST T, Key k, Data d) {
    N = new Node(k, d)

```

```

P = null
S = T.root
// search parent node
while (S != null) {
    P = S
    if (k < S.key) S = S.left
    else S = S.right
}
// insert node
if (P == null)
    T.root = N
else {
    N.parent = P
    if (k < P.key) P.left = N
    else P.right = N
}
}

```

Il **costo computazionale** di insert è il seguente:

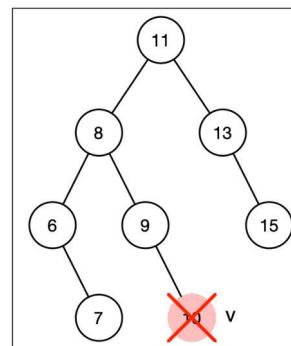
- Caso **ottimo**, quando il nuovo nodo deve essere inserito come figlio della radice o l'albero era vuoto: $O(1)$.
- Caso **pessimo**, quando il nuovo nodo deve essere inserito nel livello h (h : altezza dell'albero): $O(h)$.

Delete

Per rimuovere un nodo all'interno di un BST occorre distinguere tre casi:

- **Caso 1, il nodo da rimuovere è una foglia.**

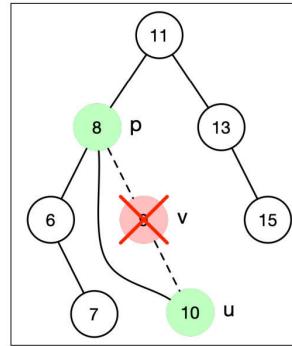
Viene semplicemente rimosso il nodo da rimuovere.



Esempio di delete in un BST nel caso 1.

- **Caso 2, il nodo da rimuovere ha un solo figlio.**

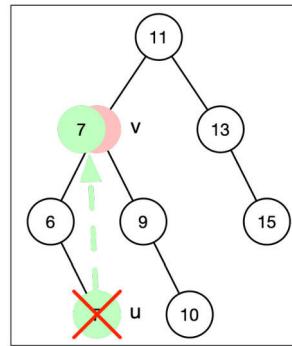
Viene rimosso il nodo da rimuovere e l'unico figlio di tale nodo diventa figlio del padre del nodo da rimuovere.



Esempio di delete in un BST nel caso 2.

- Caso 3, il nodo da rimuovere ha due figli.**

Viene copiato il predecessore/successore del nodo da rimuovere al posto di quest'ultimo e viene rimosso tale predecessore/successore.



Esempio di delete in un BST nel caso 3.

Lo **pseudocodice** di delete è il seguente:

```

void delete(BST T, Key k) {
    v = search(T, k)
    if (v != null) {
        if (v.left == null || v.right == null)
            // case 1 or 2
            deleteNode(T, v)
        else {
            // case 3
            u = predecessor(v)
            v.key = u.key
            v.data = u.data
            deleteNode(T, u)
        }
    }
}

void deleteNode(BST T, Node v) {
    p = v.parent
    if (p != null) {
        if (v.left == null && v.right == null) {
            // case 1
            if (p.left == v)
                p.left = null
            else p.right = null
        } else if (v.right != null) {
            // case 2
            if (p.left == v)
                p.left = v.right
            else p.right = v.right
        }
    }
}

```

```

} else if (v.left != null) {
    // case 2
    if (p.left == v)
        p.left = v.left
    else p.right = v.left
}
} else if (v.right != null)
    // case 2
    T.root = v.right
else
    // case 1 or 2
    T.root = v.left
}

```

Il **costo computazionale** di delete è il seguente:

Notiamo che la funzione `deleteNode` ha sempre un costo $O(1)$.

- Caso **ottimo**, quando la ricerca e il predecessore si trovano nel caso ottimo: $O(1)$.
- Caso **pessimo**, quando la ricerca e il predecessore si trovano nel caso pessimo: $O(h)$ (h : altezza dell'albero).

Caso medio

Notiamo che per tutte le funzioni basilari degli alberi BST il costo computazionale dipende dell'altezza h dell'albero. Tale altezza, per un albero BST con n nodi, può variare da un caso pessimo in cui $h = \Theta(n)$, quando l'albero binario è una lista, e un caso ottimo in cui $h = \Theta(\log n)$, quando l'albero binario è perfetto. È possibile però dimostrare che un BST costruito da n inserimenti casuali ha un altezza media $\bar{h} = O(\log n)$. Quindi il costo di tutte le funzioni basilari nel caso medio è $O(\log n)$.

Dizionario con albero binario di ricerca

Visto che un dizionario ha come operazioni basilari quelle di search, insert e delete, le quali sono già state analizzate per il caso degli alberi BST, possiamo pensare di poter implementare un dizionario tramite quest'ultima struttura dati. In questo modo otteniamo i seguenti costi computazionali confrontati agli altri tipi di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 6.2 - Alberi AVL

Abbiamo visto che l'utilizzo di alberi binari di ricerca consentono di effettuare operazioni di ricerca, inserimento e rimozione con costi $O(h)$, dove h è l'altezza dell'albero. Abbiamo inoltre visto che l'altezza di un albero è $\Omega(\log n)$ e $O(n)$. Il nostro obiettivo, per mantenere le operazioni basilari di un albero con un costo $O(\log n)$, è quello di creare una struttura dati ad albero bilanciato che non si sbilanci a seguito di operazioni di inserimento e rimozione di nodi.

Una struttura dati di questo tipo è un **albero AVL**, ovvero un albero binario di ricerca **perfettamente bilanciato**. Tale albero mantiene il suo bilanciamento in quanto viene automaticamente bilanciato in seguito ad inserimenti e rimozioni che causano sbilanciamento.

Nozioni preliminari

Fattore di bilanciamento

Il **fattore di bilanciamento** $\beta(v)$ di un nodo v è dato dalla differenza tra l'altezza del suo sottoalbero sinistro e destro:

$$\beta(v) = \text{altezza}(v.left) - \text{altezza}(v.right)$$

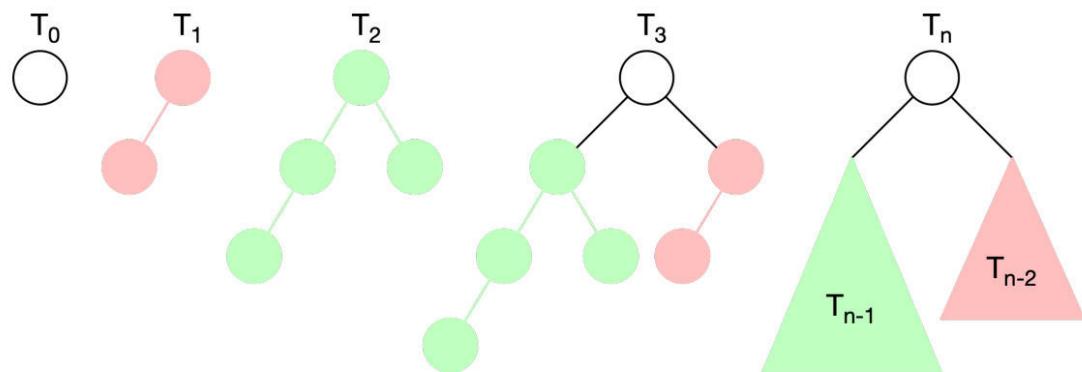
Bilanciamento in altezza

Un albero si dice **bilanciato in altezza** se per ogni suo nodo v le altezze dei suoi sottoalberi sinistro e destro differiscono al più di 1:

$$|\beta(v)| = 1$$

Altezza di un albero AVL

Siccome un albero AVL è un albero bilanciato in altezza, possiamo calcolare la sua altezza massima calcolando l'altezza dell'albero bilanciato con sbilanciamento massimo. È possibile costruire un tale albero utilizzando l'algoritmo di Fibonacci, e ottenendo così un albero, detto albero di Fibonacci, il quale per ogni nodo v non foglia $|\beta(v)| = 1$.



Costruzione dell'albero di Fibonacci.

Per calcolare il numero di nodi di un albero di Fibonacci utilizziamo il seguente teorema:

Il numero di nodi di un albero di Fibonacci con altezza h è equivalente a:

$$n_h = F_{h+3} - 1$$

(Dimostrazione a slide 9 del pacco di slide “Alberi AVL”).

Ricordando che $F_n = \Theta(\phi^n)$, dove $\phi \approx 1.618$, otteniamo dunque che $n_h = F_{h+3} - 1 = \Theta(\phi^{h+3}) = \Theta(\phi^h)$ e possiamo quindi concludere che $h = \Theta(\log n_h)$.

Poichè l'albero di Fibonacci è l'albero bilanciato con altezza massima possiamo concludere che l' altezza di un albero AVL è $\Theta(\log n)$.

Mantenere il bilanciamento

Per mantenere il bilanciamento su un albero AVL occorre modificare il codice delle funzioni insert e delete degli alberi binari, mentre la funzione search rimane invariata.

Tale obiettivo si raggiunge innanzitutto tenendo traccia delle altezze dei sottoalberi sinistro e destro di ogni nodo, al fine di calcolare poi il fattore di bilanciamento β . Tali operazioni possono essere effettuate utilizzando i seguenti pseudocodici, entrambi con costo $O(1)$:

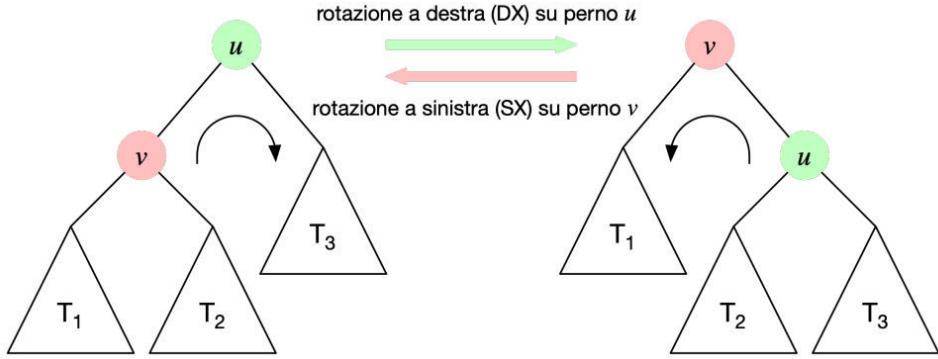
```
void updateHeight(Node T) {
    if (T != null) {
        lh = -1
        rh = -1
        if (T.left != null)
            lh = T.left.height
        if (T.right != null)
            rh = T.right.height
        T.height = max(lh, rh) + 1
    }
}

int β(Node T) {
    lh = -1
    rh = -1
    if (T.left != null)
        lh = T.left.height
    if (T.right != null)
        rh = T.right.height
    return lh - rh
}
```

Per mantenere il bilanciamento degli alberi AVL occorre inoltre, a seguito di uno sbilanciamento causato da un'aggiunta/rimozione di un nodo dall'albero, effettuare delle rotazioni, introduciamo dunque tale concetto.

Rotazioni

Una **rotazione semplice**, la quale può essere effettuata sia a destra che a sinistra, è un'operazione fondamentale per ribilanciare un albero. Visualizziamo tale operazione nella seguente immagine:



Con tali rotazioni la proprietà di ordine di un BST viene preservata in quanto nelle rotazioni a destra $v.key \leq u.key \leq T_3$ e nelle rotazioni a sinistra $u.key \geq v.key \geq T_1$.

È possibile effettuare una rotazione semplice utilizzando una funzione con costo $O(1)$.

Tipologie di sbilanciamenti

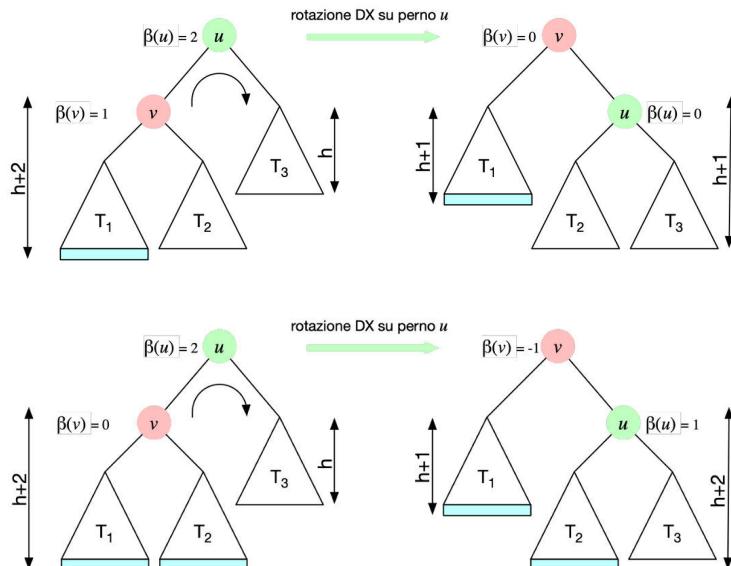
Assumendo di avere un albero AVL bilanciato, in cui un suo sottoalbero radicato in un nodo u diventa sbilanciato a seguito di una operazione di inserimento o rimozione, è possibile ricadere in 4 casi di sbilanciamento:

- **Sbilanciamento SS:** $\beta(u) = 2$ e $\beta(u.left) \geq 0$.
- **Sbilanciamento DD:** $\beta(u) = -2$ e $\beta(u.right) \leq 0$.
- **Sbilanciamento SD:** $\beta(u) = 2$ e $\beta(u.left) = -1$.
- **Sbilanciamento DS:** $\beta(u) = -2$ e $\beta(u.right) = 1$.

Ribilanciamenti

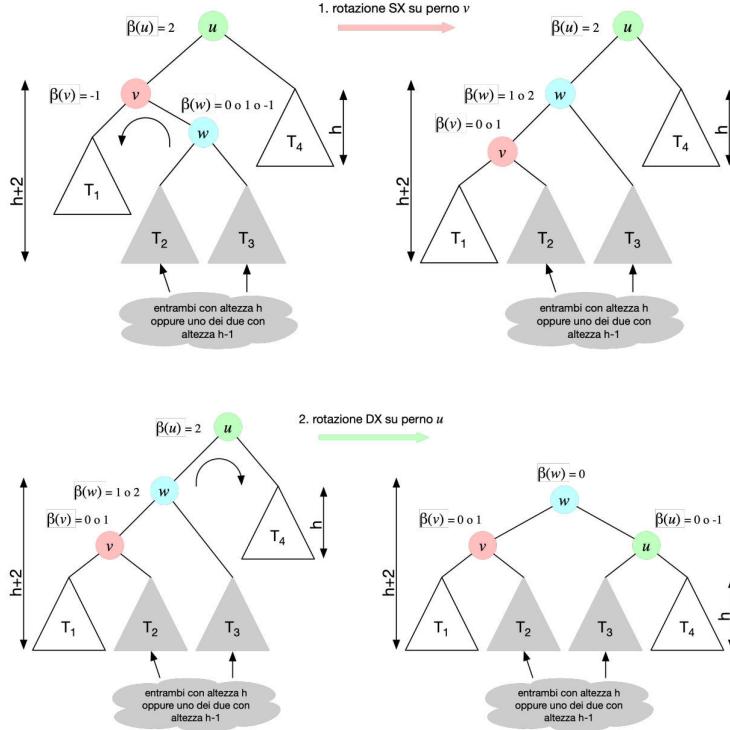
Per ciascuno dei 4 casi di sbilanciamento occorre effettuare una particolare rotazione al fine di riottenere un albero bilanciato.

- **Sbilanciamento SS** \implies Rotazione DX su u .



- **Sbilanciamento DD** \implies Rotazione SX su u .

- **Sbilanciamento SD** \implies Rotazione SX su $u.left$ e rotazione DX su u .



- **Sbilanciamento DS** \implies Rotazione DX su $u.right$ e rotazione SX su u .

Visto che in tutti i casi, per ribilanciare l'albero, vengono utilizzate delle rotazioni le quali funzioni hanno costo costante $O(1)$, anche il ribilanciamento di un albero AVL ha un costo costante $O(1)$.

Inserimento e rimozione in un albero AVL

A questo punto è possibile analizzare il costo complessivo dell'inserimento e della rimozione di un nodo all'interno di un albero AVL.

In generale, il costo dell'inserimento e della rimozione di un nodo all'interno di un albero è $O(h)$, e visto che l'altezza di un albero AVL è $\log n$ concludiamo che tali operazioni in un albero AVL hanno un costo pessimo $O(\log n)$. A questa operazione dobbiamo però aggiungere i costi dell'aggiornamento delle altezze dei sottoalberi, nel caso pessimo $O(\log n)$, e il costo della procedura di ribilanciamento nel caso in cui si presentano dei nodi critici (il fattore di bilanciamento del loro sottoalbero diventa ± 2), che nel caso della rimozione possono essere molteplici in un unico percorso radice-foglio, dunque il costo pessimo di ribilanciamento è $O(\log n)$.

Concludiamo dunque che il **costo computazionale** dell'inserimento e della rimozione di un nodo in un albero AVL è:

- Caso pessimo: $O(\log n)$.

Dizionario con albero binario di ricerca

Riassumiamo il costo delle operazioni basilari di un dizionario implementato tramite albero AVL confrontati con le altre tipologie di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$
Albero AVL	$O(\log n)$					

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 6.3 - Algoritmi di decisione su alberi

Game tree

Un **game tree** è un albero che rappresenta tutte le possibili partite in un gioco a turni. In tale albero i nodi rappresentano una situazione di gioco, gli archi tutte le mosse giocabili a partire da un nodo e le foglie gli stati finali di gioco (vittoria, sconfitta e patta).

I game tree vengono utilizzati per realizzare algoritmi di decisione per giochi a turni, i quali riescono a scegliere le scelte migliori da effettuare in ogni possibile situazione di gioco.

Il problema solitamente, in algoritmi di tale genere, è quello di capire come effettuare le scelte di gioco senza dover generare e visitare l'intero albero, il quale è tipicamente troppo ampio.

Algoritmo MiniMax

L'algoritmo **MiniMax** è un algoritmo di decisione che consiste nel cercare di minimizzare la massima perdita possibile.

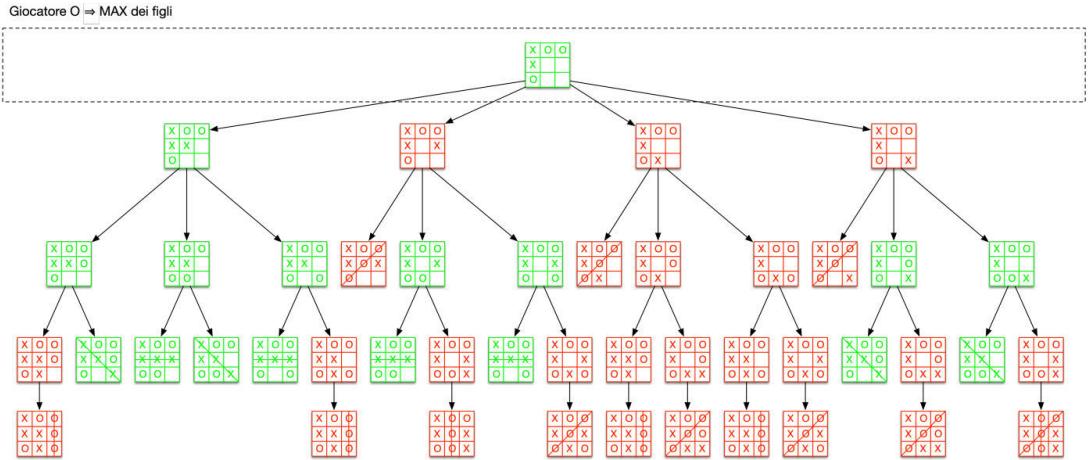
Funzionamento

Per attuare tale algoritmo occorre etichettare tramite un valore le diverse configurazioni finali di gioco, presenti nelle foglie dell'albero:

- +1: vittoria.
- 0: patta.
- -1: sconfitta.

A questo punto occorre propagare tale valore dalle foglie alla radice al fine di comprendere quale scelta sia la migliore da intraprendere. Per propagare i valori si utilizza dunque la tecnica della massimizzazione per i nodi scelti dal proprio giocatore, ovvero gli viene assegnato il valore massimo tra quelli assegnati ai figli, e della minimizzazione per i nodi scelti dal giocatore avversario, ovvero gli viene assegnato il valore minimo tra quelli assegnati ai figli.

Capiamo il tutto con un esempio nel gioco tic tac toe nel quale il proprio giocatore utilizza il simbolo x e viene utilizzato il colore rosso per il valore -1 e il colore verde per il valore +1:



Notiamo che alle foglie sono stati assegnati i valori +1/-1 in base alla situazione di vittoria/sconfitta. A questo punto, livello dopo livello, occorre massimizzare quando l'arco è dato dal turno del proprio giocatore e minimizzare quando l'arco è dato dal turno del giocatore avversario, fino ad arrivare alla radice. Tramite questo algoritmo dunque si suppone che il giocatore avversario faccia la migliore scelta possibile per minimizzare la sua perdita. Una volta arrivato alla radice il giocatore ha la possibilità di fare la scelta migliore al fine di minimizzare la massima perdita possibile, dunque nel gioco del tic tac toe di vincere o pareggiare, nei casi in cui la vittoria non è possibile.

Algoritmo

Lo **pseudocodice** di MiniMax è il seguente:

```

int miniMax(Node T, bool playerA) {
    if (isLeaf(T)) eval = evaluate(T)
    else if (playerA == true) {
        for (c ∈ children(T))
            eval = max(eval, miniMax(c, false))
    } else {
        for (c ∈ children(T))
            eval = min(eval, miniMax(c, true))
    }
    return eval
}

```

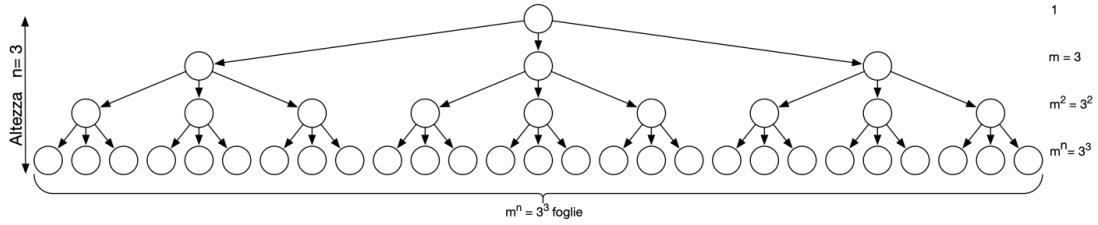
Il **costo computazione** di MiniMax è il seguente:

- **Tempo** (viene visitato tutto l'albero): $\Theta(n)$.
- **Memoria**: $O(h)$.

Grandezza di un game tree

Per stimare in maniera esatta il costo computazionale di MiniMax occorre calcolare il numero esatto di nodi di un game tree. Purtroppo però tale operazione è molto complessa, dunque possiamo solamente trovare un upper bound al game tree effettivo ammettendo configurazioni di gioco illegali, ovvero configurazioni che non sono possibili in una partita reale.

Per fare ciò occorre supporre che un giocatore ha a disposizione al massimo m mosse possibili per turno e il gioco termina in massimo n turni. Ad esempio nel gioco forza 4, $m = 7$, ossia le colonne della matrice nelle quali è possibile effettuare una mossa, e $n = 6 \times 7$, ossia il prodotto tra le righe e le colonne della matrice, il numero dei buchi della matrice.



Il numero $P(m, n)$ di **partite possibili** è dato dal numero di foglie, quindi contando che alcune di queste sono illegali:

$$P(m, n) \leq m^n = O(m^n)$$

Il numero totale $T(m, n)$ di **nodi del game tree** è limitato da:

$$T(m, n) \leq 1 + m + m^2 + \dots + m^n = \sum_{k=0}^n m^k = \frac{m^{n+1} - 1}{m - 1} = O(m^n)$$

Grandezza di un game tree su un gioco con n oggetti

Se un gioco, come tic tac toe, ha n oggetti a disposizione tra i quali scegliere e i quali, dopo la scelta, non sono più disponibili, è possibile migliorare l'accuratezza degli upper bound scelti in precedenza.

In questo caso il numero $P(n)$ di partite disponibili partendo da una scelta di n oggetti diventa:

$$P(n) \leq n! = O(n!)$$

Il numero totale $T(n)$ di **nodi del game tree** diventa:

$$T(n) \leq 1 + n + n(n-1) + n(n-1)(n-2) + \dots + n! = \dots = O(n!)$$

Costo computazionale di MiniMax

A seguito dell'analisi della grandezza di un game tree possiamo elencare con maggiore esattezza i **costi computazionali** della funzione MiniMax.

Per un gioco con m mosse ed un massimo di n turni abbiamo il seguente costo computazionale:

- Costo **pessimo** in termini di **tempo**: $O(m^n)$.
- Costo **pessimo** in termini di **memoria**: $O(h)$.

Per un gioco con n oggetti consumabili abbiamo il seguente costo computazionale:

- Costo **pessimo** in termini di **tempo**: $O(n!)$.
- Costo **pessimo** in termini di **memoria**: $O(n)$.

Algoritmo AlphaBeta

Ottimizzazione per MiniMax

È possibile pensare ad una **piccola ottimizzazione** di MiniMax per aumentarne le prestazioni.

Notiamo infatti che nel caso in cui un nodo ha almeno un figlio con il valore **massimo/minimo assoluto**, tale valore viene trasmesso al padre nel caso in cui occorre massimizzare/minimizzare. In

base a tale concetto possiamo dunque interrompere la visita sui sottoalberi del nodo attuale nel momento in cui uno dei figli abbia come valore il massimo/minimo assoluto.

Aggiungiamo dunque tale controllo nello pseudocodice:

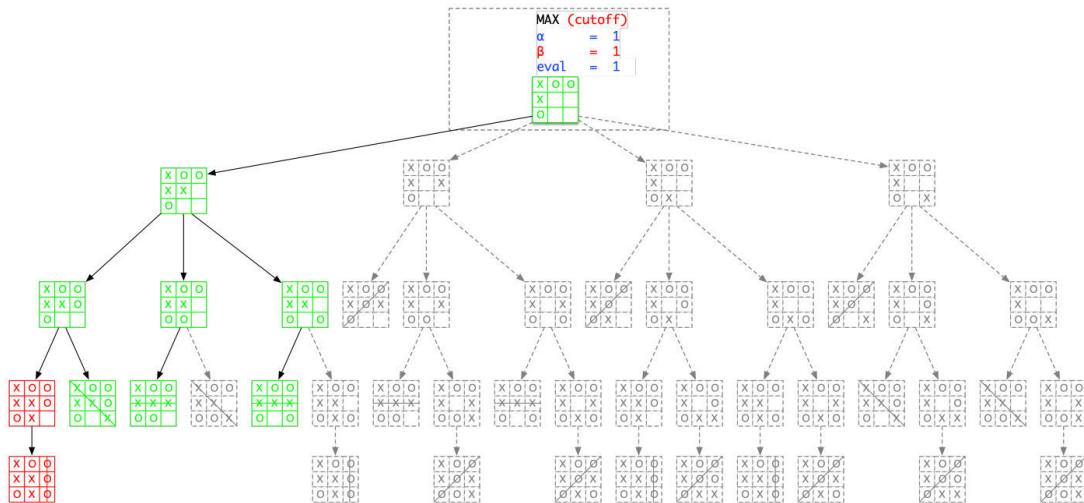
```
int MiniMax(Node T, bool playerA) {
    if (isLeaf(T)) eval = evaluate(T)
    else if (playerA == true) {
        for (c ∈ children(T)) {
            eval = max(eval, MiniMax(c, false))
            if (eval == maxval) break
        }
    } else {
        for (c ∈ children(T)) {
            eval = min(eval, MiniMax(c, true))
            if (eval == minval) break
        }
    }
    return eval
}
```

Generalizzazione dell'ottimizzazione

Tale ottimizzazione può essere **generalizzata** costruendo un nuovo algoritmo di decisione, chiamato **AlphaBeta**, il quale partendo dall'algoritmo MiniMax utilizza due valori, α e β , al fine di visitare meno nodi.

In tale algoritmo α rappresenta il **punteggio minimo** ottenibile dal nostro giocatore, β invece rappresenta il **punteggio massimo** ottenibile dal giocatore avversario. Tali valori permettono di rappresentare il range $[\alpha, \beta]$ di possibile risultati per i figli di tale nodo. Se ad un certo punto, per un certo nodo, $\beta \leq \alpha$ allora non ha più senso continuare ad analizzare gli altri suoi figli e questi vengono saltati.

Visualizziamo tale algoritmo in esecuzione sul game tree di una partita di tic tac toe:



Visualizzazione grafica di ottimizzazione di MiniMax tramite l'algoritmo AlphaBeta.

Lo **pseudocodice** è il seguente:

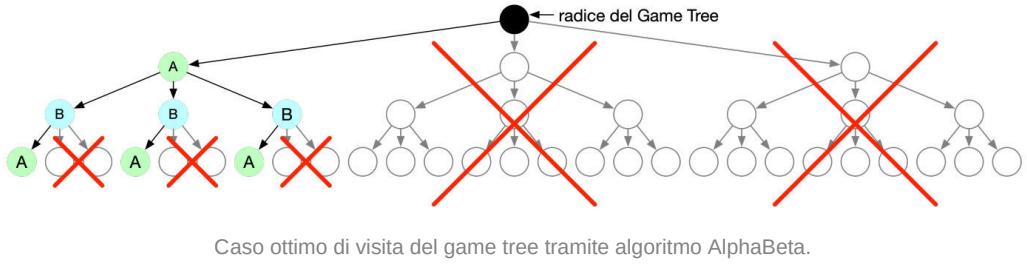
```
int AlphaBeta(Node T, bool playerA, int α, int β) {
    if (isLeaf(T)) eval = evaluate(T)
    else if (playerA == true) {
        for (c ∈ children(T)) {
            eval = max(eval, AlphaBeta(c, false, α, β))
            if (eval == maxval) break
        }
    } else {
        for (c ∈ children(T)) {
            eval = min(eval, AlphaBeta(c, true, α, β))
            if (eval == minval) break
        }
    }
    return eval
}
```

```

        eval = max(eval, AlphaBeta(c, false, α, β))
        α = max(eval, α)
        if (β <= α) break
    }
} else {
    for (c ∈ children(T)) {
        eval = min(eval, AlphaBeta(c, true, α, β))
        β = min(eval, β)
        if (β <= α) break
    }
}
return eval
}

```

Notiamo che nel caso in cui ad ogni turno del proprio giocatore viene analizzata come prima mossa una mossa che lo porta alla vittoria, si evita di visitare la restanti mosse.



Caso ottimo di visita del game tree tramite algoritmo AlphaBeta.

In tal caso il numero $T(m, n)$ dei nodi dell'albero che vengono visitati è limitato da:

$$\begin{aligned}
 T(m, n) &\leq 1 + 1 + m + m + m^2 + m^2 + \cdots + m^{n/2} + m^{n/2} \\
 &= \sum_{k=0}^{n/2} 2m^k = 2 \frac{m^{(n/2)+1} - 1}{m - 1} = O(m^{n/2}) = O(\sqrt{m^n})
 \end{aligned}$$

Otteniamo dunque uno **speed-up quadratico** rispetto all'algoritmo MiniMax.

Il **costo computazionale** è il seguente:

- Caso **ottimo**: $O(\sqrt{m^n})$.
- Caso **pessimo**, quando viene visitato tutto l'albero: $O(m^n)$.

Il caso ottimo si riesce ad ottenere sempre solo nel caso in cui si trova una **strategia ottima** che permette sempre di valutare come prima mossa una mossa vincente, tale strategia renderebbe però inutile l'algoritmo AlphaBeta. Il costo computazionale medio invece è difficile da calcolare, comunque sappiamo che il numero di nodi valutati in tal caso è minore dell'algoritmo MiniMax.

Ricerca limitata in profondità

Visto che anche nel caso ottimo dell'algoritmo AlphaBeta l'albero potrebbe avere una dimensione molto grande per una visita in tempi ragionevoli, spesso è utile limitare la ricerca ad un **livello massimo di profondità** e valutare le configurazioni non-finali tramite euristiche nel seguente modo (X : vittoria, 0: patta, $-X$: sconfitta):

- $0 < eval \leq X$: configurazione favorevole.
- $-X \leq eval < 0$: configurazione sfavorevole.
- 0: totale incertezza o patta.

In questo modo non si riesce ad avere la certezza di minimizzare la perdita ma vengono scelte le configurazioni più favorevoli al fine di ottenere la **maggior possibilità di vittoria possibile**.

Lo **pseudocodice** è il seguente:

```

int AlphaBeta(Node T, bool playerA, int α, int β, int depth) {
    if (depth == 0 || isLeaf(T))
        eval = evaluate(T, depth)
    else if (playerA == true) {
        for (c ∈ children(T)) {
            eval = max(eval, AlphaBeta(c, false, α, β, depth - 1))
            α = max(eval, α)
            if (β ≤ α) break
        }
    } else {
        for (c ∈ children(T)) {
            eval = min(eval, AlphaBeta(c, true, α, β, depth - 1))
            β = min(eval, β)
            if (β ≤ α) break
        }
    }
    T.label = eval
    return eval
}

```

Algoritmo IterativeDeepening

Nel caso in cui si hanno dei **limiti di tempo per la scelta** di una mossa è molto difficile stimare il livello massimo di profondità visitabile in modo completo tramite algoritmo AlphaBeta con limite di profondità, in quanto quest'ultimo effettua una visita in profondità e non in ampiezza. Una stima troppo ottimistica del livello di profondità massima causerebbe una visita completa solo per poche mosse, mentre una stima troppo pessimista causerebbe una valutazione poco accurata delle varie mosse a disposizione.

Una soluzione a tale problema è quella di effettuare una **visita in ampiezza** la quale va a tentativi visitando ogni volta l'albero partendo dalla radice e aggiungendo di volta in volta un nuovo livello alla profondità che si va a visitare. Nel momento in cui si raggiunge il limite di tempo prestabilito viene utilizzato l'eval ottenuto nell'ultima visita completa effettuata.

Lo **pseudocodice** è il seguente:

```

int IterativeDeepening(Node T, bool playerA, int depth) {
    α = MinAlpha
    β = MaxBeta
    eval = 0
    for (d = 0, ... , depth) {
        if (timeIsRunningOut()) break
        eval = AlphaBeta(T, playerA, α, β, d)
    }
    return eval
}

```

Il **costo computazionale** in termini di **memoria** è il seguente:

- Caso **pessimo**: $O(d)$.

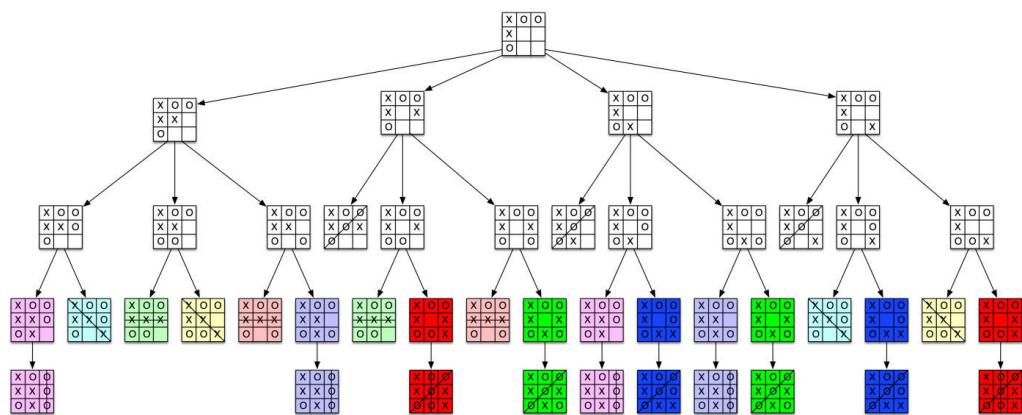
Per calcolare il **costo computazionale** in termini di **tempo** osserviamo che la radice viene visitata $d + 1$ volte, i suoi figli d volte ecc, dunque effettuando alcune equivalenze matematiche otteniamo che:

- Caso **pessimo**: $O(m^d)$.

Notiamo dunque che a parità di profondità la ricerca in ampiezza IterativeDeepening ha lo **stesso costo asintotico** della ricerca in profondità AlphaBeta, in quanto il numero di nodi al livello d domina l'intero costo. In termini pratici è facile però notare che le costanti moltiplicative nascoste dalla notazione asintotica di IterativeDeepening lo rendono **concretamente più lento** di AlphaBeta, in quanto gli stessi nodi vengono visitati più volte. Tale svantaggio in termini di tempo viene ripagato dalla possibilità di avere un maggiore controllo sulla visita del Game Tree quando questa deve avvenire entro certi limiti di tempo.

Gestione delle ripetizioni

Infine analizziamo un'ulteriore ottimizzazione notando che molti stati di gioco possono comparire più volte nell'albero in quanto uno stesso stato può essere ottenuto con differenti sequenze di mosse. Per velocizzare la ricerca è dunque utile riconoscere le configurazioni già valutate al fine di non ripetere il lavoro.



▼ 7.0 - Tabelle Hash

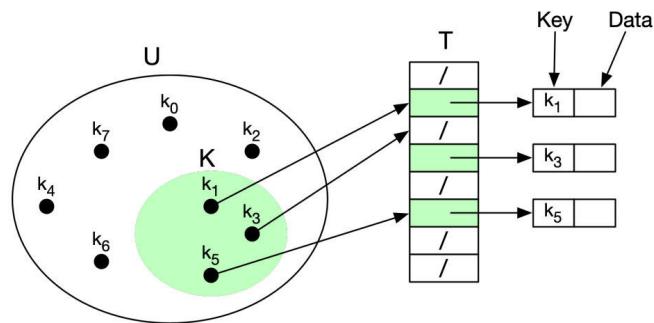
Le **tabelle Hash** sono strutture dati che consentono di implementare dizionari con operazioni basilari (Search, Insert e Delete) estremamente efficienti. Tali operazioni effettuate su un dizionario implementato tramite tabella Hash possono avere un costo pessimo lineare, ma in media le prestazioni computazionali sono efficienti, addirittura sotto ragionevoli assunzioni probabilistiche le operazioni basilari hanno un costo costante.

Tale tipologia di implementazione non viene scelta universalmente, ma in base al dominio di applicazione. Analizzeremo tale scelta mettendola a confronto con l'implementazione tramite tabelle ad indirizzamento diretto, tenendo a mente che indicheremo con:

- U : universo di tutte le chiavi possibili.
- K : insieme di tutte le chiavi effettivamente utilizzate.

▼ 7.1 - Tabelle ad indirizzamento diretto

Tale implementazione è basata sull'utilizzo di un array di dimensione U in cui ogni chiave k è memorizzata all'interno di esso in posizione k .



Le operazioni basilari per un dizionario implementato tramite tabella ad indirizzamento diretto hanno il seguente **pseudocodice**:

```
Data search(HashTab T, Key k) {
    if (T[k] == null) return null
    else return T[k].data
}

void insert(HashTab T, Key k, Data d) {
    T[k].key = new node(k, d)
}

void delete(HashTab T, Key k) {
    T[k] = null
}
```

Il **costo computazionale** di una tale tabella è dunque il seguente:

- Costo in termini di **tempo** (tutte le operazioni hanno lo stesso costo): $O(1)$.
- Costo in termini di **memoria**: $O(U)$.

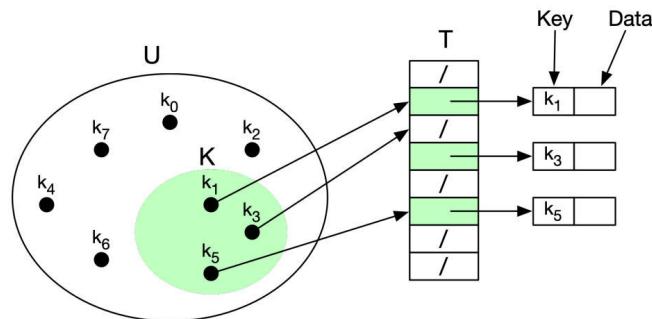
Notiamo dunque che il costo in termini di tempo è molto conveniente per tutte le operazioni basilari, ma la scelta di una tale implementazione risiede nella dimensione di K rispetto a U ; se $K \approx U$, allora tale soluzione è accettabile, altrimenti, se $K \ll U$, c'è un grande spreco di memoria e tale soluzione risulta non accettabile.

Inoltre, nel caso in cui U è molto grande di per sé, può risultare impossibile memorizzare un tale array all'interno di un elaboratore. Ad esempio, se occorre utilizzare un dizionario in cui si utilizzano degli identificatori che contengono lettere e numeri e possono avere una lunghezza massima di 20 caratteri, l'universo U risulta grande $36^{20} \approx 10^{31}$, e nel caso in cui l'array utilizzato per implementare la struttura dati contiene dei puntatori con 4 bytes ognuno, allora la memoria totale utilizzata da un tale array è di circa $10^{31} \cdot 4$ bytes $> 10^{19}$ terabytes, impossibile da memorizzare all'interno degli elaboratori oggi in circolazione.

▼ 7.2 - Tabelle Hash

La tipologia di implementazione a tabella Hash viene dal problema che molte volte la dimensione dell'insieme K è molto minore rispetto a quello dell'insieme U . L'idea è quella di utilizzare un array T di dimensione m , con $m = \Theta(K)$, e una funzione Hash $h : U \rightarrow [0, \dots, m - 1]$ che fornito un elemento da memorizzare restituisca il valore Hash $h(k)$ di tale elemento, il quale corrisponde all'indice da utilizzare per memorizzare l'elemento in posizione $T[h(k)]$.

Il problema di tale implementazione è il fatto che, siccome $U > m$, allora due o più elementi dell'insieme U possono avere lo stesso valore Hash, ottenendo così una **collisione**. Idealmente si vorrebbe utilizzare funzioni Hash che evitino collisioni, ma visto che queste sono inevitabili, occorre almeno minimizzarle.



Funzione Hash

Una buona funzione Hash deve soddisfare la proprietà di **uniformità semplice**, ovvero deve distribuire in maniera equiprobabile le chiavi negli m indici dell'array T . Infatti nel caso in cui certi indici vengono scelti con maggiore probabilità allora si otterebbe un numero maggiore di collisioni.

Per costruire una funzione Hash che soddisfi tale proprietà occorre conoscere la **distribuzione di probabilità** con cui le chiavi sono estratte da U , purtroppo però conoscere tale distribuzione probabilistica è spesso irrealistico.

Assunzioni

Per discutere dell'implementazione di una tabella Hash assumiamo che tutte le chiavi hanno la stessa probabilità di essere estratte da U , in quanto è l'unico modo per proporre un meccanismo generale. Assumiamo inoltre, per analizzare i costi computazionali, che la funzione Hash possa essere calcolata in un tempo $O(1)$. Nel caso in cui invece una funzione Hash non sia $O(1)$, allora tale costo dominerebbe quello di tutte le operazioni basilari.

Inoltre sappiamo che è sempre possibile trasformare in un qualche modo una chiave k in un intero positivo al fine di ottenere un indice da utilizzare per indirizzare nell'array T . Un esempio banale può essere quello di utilizzare il numero decimale ottenuto dalla rappresentazione binaria di k .

Funzione Hash: metodo della divisione

$$h(k) = k \bmod m$$

Esempi:

- $m = 12, k = 100 \implies h(k) = 4$.
- $m = 10, k = 101 \implies h(k) = 1$.

Il **vantaggio** di tale approccio è quello di utilizzare una funzione molto efficiente, in quanto composta da una singola operazione.

Lo **svantaggio** invece è quella di essere suscettibile a certi valori di m . Ad esempio infatti se $m = 10$, allora $h(k) = \text{ultima cifra di } k$, oppure se $m = 2^p$, allora $h(k)$ dipende unicamente dai p bit meno significativi di k e non da tutti i bit. La soluzione a questo problema è quella di preferire un numero m diverso da una potenza di 2 e di 10, dunque sarebbe meglio un numero primo non troppo vicino a una potenza esatta di 2 e di 10.

Funzione Hash: metodo della moltiplicazione

$$h(k) = \lfloor m(kC - \lfloor kC \rfloor) \rfloor$$

Il metodo della moltiplicazione consiste nello scegliere una costante $0 < C < 1$, moltiplicare tale costante per la chiave e prendere la parte frazionaria, a questo punto moltiplicare quest'ultima per m e tenere la parte intera.

Il **vantaggio** è quello di poter scegliere un qualunque valore di m in quanto questo non è critico.

Lo **svantaggio** invece è il fatto che la costante C influenza la proprietà di uniformità della funzione. Per ottenere una buona proprietà di uniformità in tutti i casi viene suggerita una costante $C = (\sqrt{5} - 1)/2$.

Funzione hash: metodo della codifica algebrica

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0) \bmod m$$

dove:

- k_i è l' i -esimo bit della rappresentazione binaria di k , oppure l' i -esima cifra della rappresentazione decimale di k , o anche il codice ascii dell' i -esimo carattere.
- x è un valore costante.

Esempio:

- $k = 234, x = 3, m = 12 \implies h(k) = (2 \cdot 3^2 + 3 \cdot 3^1 + 4) \bmod 12 = 7$.

Il **vantaggio** di tale approccio è quello di far dipendere il risultato da tutti i bit/caratteri della chiave.

Lo **svantaggio** è invece il fatto di risultare in una funzione abbastanza costosa da calcolare, in quanto richiedere n addizioni e $n \cdot (n + 1)/2$ prodotti, dove $n = \Theta(\log k)$.

È possibile comunque ovviare a questo problema utilizzando una regola algebrica, la regola di Horner, la quale afferma che un polinomio di grado n del tipo $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots +$

$a_1x + a_0$ può essere riscritto nel seguente modo $p(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_{n-1} + a_nx))))$.

Utilizzando tale regola è possibile abbassare il costo computazionale del calcolo della funzione Hash da quadratico a lineare sul numero di cifre della chiave k . Dato inoltre che il numero di cifre di una chiave è tipicamente un numero molto piccolo, possiamo assumere un costo **costante** per il calcolo della funzione Hash tramite il metodo della codifica algebrica.

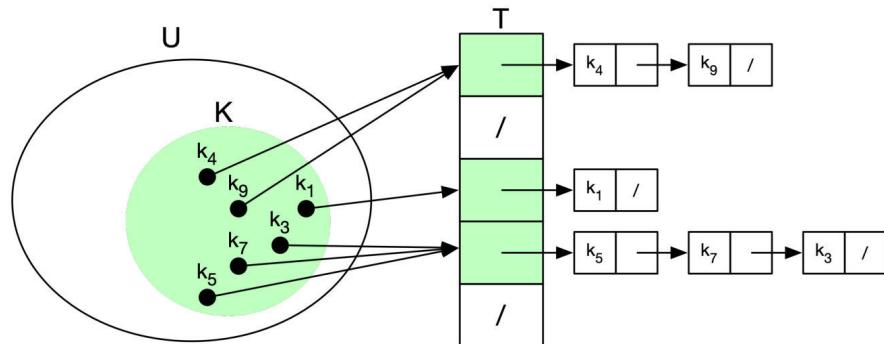
Problema delle collisioni

Abbiamo visto delle funzioni Hash che rispettano l'uniformità semplice, la quale riduce le collisioni ma non le elimina. È possibile inoltre notare che anche assumendo hashing uniforme semplice la probabilità che ci sia collisione tra due chiavi è molto alta (vedi teorema del compleanno).

Occorre dunque trovare delle tecniche di gestione di tali collisioni, noi vedremo quella del concatenamento e quella dell'indirizzamento aperto.

Concatenamento

La tecnica del concatenamento consiste nell'utilizzare la tabella Hash come un array di puntatori che puntano alla testa di liste contenenti tutte chiavi con lo stesso valore hash.



I **costi computazionali** delle principali operazioni su una lista Hash con metodo del concatenamento sono i seguenti (L = lunghezza della lista di collisione più lunga, $\alpha = n/m$):

- **search** (ricerca lineare su lista concatenata)
 - Caso **ottimo**: $O(1)$.
 - Caso **pessimo**: $O(L)$.
 - Caso **medio**: $\Theta(1 + \alpha)$.

Dimostrazione

Sotto l'assunzione di hashing uniforme semplice ogni slot della tabella ha mediamente α chiavi.

Una ricerca senza successo in una tabella hash con concatenamento ha dunque costo medio $\Theta(1 + \alpha)$, in quanto se k non compare nella tabella la ricerca visita tutte le chiavi nella lista concatenata $T[h(k)]$, che ha in media α chiavi. Quindi il costo computazionale è uguale al costo del calcolo della funzione hash, che possiamo assumere costante, sommato al tempo di visita della lista $T[h(k)]$, ovvero α : $\Theta(1 + \alpha)$.

Una ricerca con successo in una tabella hash con concatenamento ha dunque costo medio $\Theta(1 + \alpha)$, in quanto se k compare nella tabella la ricerca visita in media la metà delle chiavi nella lista concatenata $T[h(k)]$, che ha in media α chiavi. Quindi il costo computazionale è

uguale al costo del calcolo della funzione hash, che possiamo assumere costante, sommato al tempo di visita della lista $T[h(k)]$, ovvero $\alpha/2$: $\Theta(1 + \alpha/2) = \Theta(1 + \alpha)$.

- **insert** (inserimento in coda su lista concatenata)
 - Caso **ottimo**: $O(1)$.
 - Caso **pessimo**: $O(L)$.
 - Caso **medio**: $\Theta(1 + \alpha)$.
- **delete** (rimozione su lista concatenata)
 - Caso **ottimo**: $O(1)$.
 - Caso **pessimo**: $O(L)$.
 - Caso **medio**: $\Theta(1 + \alpha)$.

Indirizzamento aperto

L'idea nell'utilizzo dell'**indirizzamento aperto** è quella di, data una chiave k , se uno slot $T[h(k)]$ è già occupato, ispezionare altre celle della tabella alla ricerca di uno slot libero. Per decidere quali slot ispezionare viene estesa la funzione hash in modo che prenda in input anche il parametro "passo di ispezione":

$$h : U \times [0, \dots, m - 1] \rightarrow [0, \dots, m - 1]$$

Siccome tramite la sequenza di ispezione vogliamo visitare ogni slot della tabella solo una volta, tale sequenza deve fornire una permutazione degli indici della tabella.

Per questo motivo possiamo inoltre capire come il costo computazionale nel caso **pessimo** delle funzioni search, insert e delete è $O(m)$, mentre ciò che cambia a seconda della strategia di ispezione adottata è il costo medio.

Analizziamo 3 strategie di ispezione:

- Ispezione lineare
- Ispezione quadratica
- Doppio hashing

Ispezione lineare

$$h(k, i) = (h'(k) + i) \bmod m$$

dove $h'(k)$ è una funzione hash ausiliaria.

Il problema di questa strategia di ispezione è il **clustering primario**, ovvero il fatto di avere lunghe sotto-sequenze occupate, che diventano sempre più lunghe. I tempi medi di inserimento e cancellazione infatti crescono con il riempimento della tabella, e assumendo hashing uniforme semplice, uno slot vuoto preceduto da i slot pieni viene riempito con probabilità $(i + 1)/m$, quindi la probabilità di creare sotto-sequenze è molto più alta rispetto a quella di occupato uno slot vuoto preceduto da un altro slot vuoto.

Ispezione quadratica

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

dove $h'(k)$ è una funzione hash ausiliaria e $c_1 \neq c_2$.

Questa strategia di ispezione richiede che vengano scelte le costanti c_1 e c_2 in modo che venga garantita una permutazione degli indici della tabella.

Il problema di questa strategia è il **clustering secondario**, ovvero se due chiavi hanno la stessa ispezione iniziale, allora le loro sequenze di ispezione saranno identiche.

Doppio hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

dove $h_1(k)$ e $h_2(k)$ sono la funzione hash primaria e secondaria. Occorre scegliere la funzione hash h_2 in modo da non dare mai in output il valore 0 e in modo da permettere di iterare su tutta la tabella.

Quando avviene una collisione si usa anche la funzione secondaria per determinare il successivo slot da ispezionare. Notiamo che se $h_1 \neq h_2$ è meno probabile che per una coppia di chiavi $a \neq b$, $h_1(a) = h_1(b)$ e $h_2(a) = h_2(b)$, dunque tramite questa strategia di ispezione viene evitato il clustering primario e secondario.

Indirizzamento aperto: analisi del numero di ispezioni della funzione search

I numeri di ispezioni della funzione search in una tabella hash con indirizzamento aperto nei diversi casi sono i seguenti ($\alpha = n/m$):

- Sequenze di ispezione **equiprobabili**

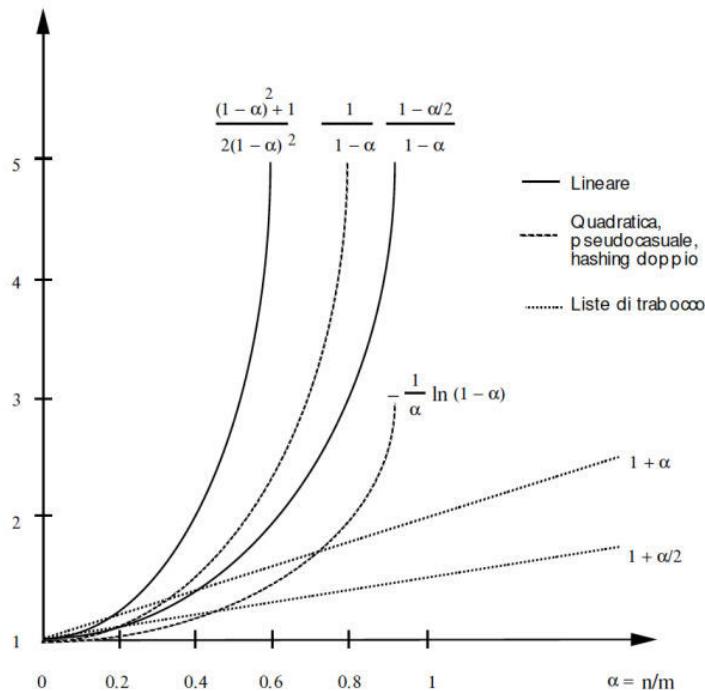
Ricadono in questa categoria l'ispezione quadratica e il doppio hashing.

- Caso **pessimo** di una ricerca **senza successo**: $O(\frac{1}{1-\alpha})$
- Caso **pessimo** di una ricerca **con successo**: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

- Sequenze di ispezione **non equiprobabili**

Ricade in questa categoria l'ispezione lineare.

- Caso **medio** di una ricerca **senza successo**: $O(\frac{(1-\alpha)^2+1}{2(1-\alpha)^2})$
- Caso **medio** di una ricerca **con successo**: $O(\frac{1-\alpha/2}{2(1-\alpha)})$



Riepilogo numero di ispezioni della funzione search in una tabella hash con indirizzamento aperto.

Notiamo inoltre che le prestazioni delle tabelle hash con indirizzamento aperto dipendono dal fattore di carico α . La strategia per migliorare le prestazioni è dunque quella di mantenere un fattore di carico basso (un fattore di carico $\alpha < 0.75$ è considerato ottimale), dunque è utile ridimensionare la tabella quando il fattore di carico supera una certa soglia critica.

Dizionario con tabella hash

Riassumiamo il costo delle operazioni basilari di un dizionario implementato tramite tabella hash confrontati con le altre tipologie di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$
Albero AVL	$O(\log n)$					
Tabelle Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

Nonostante i dizionari implementati tramite tabelle hash abbiano costi pessimi lineari, sotto ragionevoli assunzioni probabilistiche hanno un costo costante e nella pratica sono molto efficienti, infatti le implementazioni di strutture dati di tipo dizionario fanno tipicamente uso di tabelle hash.

Ad esempio in java la struttura dati **HashMap** fa completo utilizzo di tabelle hash con liste concatenate che vengono convertite in alberi AVL quando le liste concatenate diventano troppo grandi.

▼ 8.0 - Heap

▼ 8.1 - Heap binari

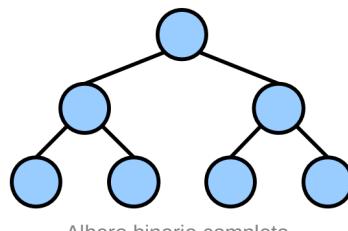
Alberi binari heap e array heap

Albero binario completo

Un **albero binario completo** è un albero binario che rispetta le seguenti proprietà:

- Tutte le foglie si trovano allo stesso livello h
- Tutti i nodi interni hanno grado (numero di figli diretti) 2

Osservazione: un albero binario completo con N nodi ha altezza $h \approx \log N$ e $N = 2^{h+1} - 1$.



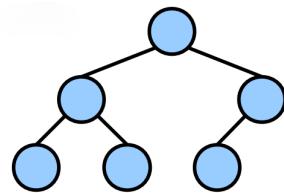
Albero binario completo.

Albero binario quasi completo

Un albero binario quasi completo è un albero binario che rispetta le seguenti proprietà:

- Albero completo almeno fino al livello $h - 1$.
- Tutti i nodi del livello h sono disposti a sinistra il più possibile.

Osservazione: tutti nodi interni di un albero quasi completo hanno grado 2, meno al più uno.



Albero binario quasi completo.

Albero binario heap

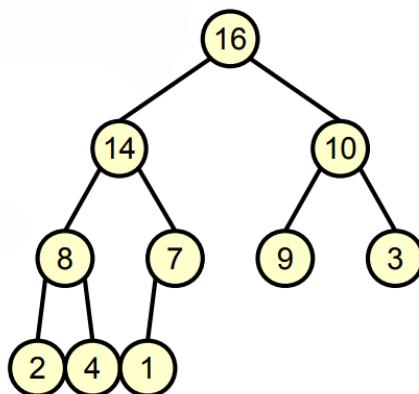
Un albero binario quasi completo è un **albero max-heap** se e solo se:

- Ad ogni nodo i viene associato un valore $A[i]$.
- $A[\text{parent}(i)] \geq A[i]$.

Un albero binario quasi completo è un **albero min-heap** se e solo se:

- Ad ogni nodo i viene associato un valore $A[i]$.
- $A[\text{parent}(i)] \leq A[i]$.

Nota: le operazioni che vedremo su un albero max-heap sono simmetriche a quelle di un albero min-heap.

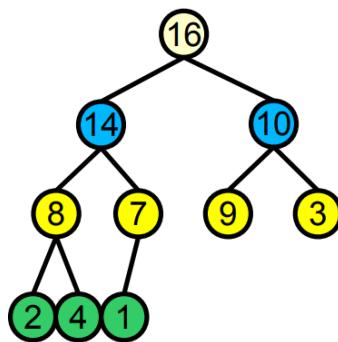


Albero max-heap.

Array heap

È possibile rappresentare un qualunque albero binario heap utilizzando un array A strutturato in questo modo:

- $A[1] =$ radice dell'albero.
- $\text{left}(i) = 2 \cdot i$.
- $\text{right}(i) = 2 \cdot i + 1$.
- $\text{parent}(i) = \text{Math.floor}(i/2)$.





Array heap.

Operazioni su array heap

Le **operazioni basilari** per array heap che vedremo sono le seguenti:

- **findMax**: ritorna il valore massimo contenuto in un max-heap.
- **heapify**: costruisce un max-heap a partire da un array privo di alcun ordine.
- **fixHeap**: risristina la proprietà di max-heap in un array con solo una radice di indice i fuori posto.
- **deleteMax**: rimuove l'elemento con il valore massimo da un max-heap, mantenendo le proprietà di max-heap.

FindMax

Il valore massimo di un max-heap è sempre la **radice**, dunque il valore da ritornare è quello dell'elemento $A[1]$.

Heapify

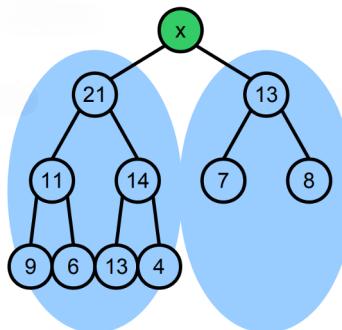
Per costruire un max-heap a partire da un array privo di alcun ordine è possibile utilizzare una funzione di tipo ricorsivo, la quale prima effettua l'heapify sul figlio sinistro della radice e poi su quello destro, per poi richiamare la funzione fixHeap una volta che l'unico elemento che ha la possibilità di essere fuori posto è la radice. Per creare una funzione di questo tipo occorre utilizzare come parametri l'array heap A , l'indice dell'ultimo elemento dell'array n e l'indice della radice i .

Lo **pseudocodice** di tale funzione è il seguente:

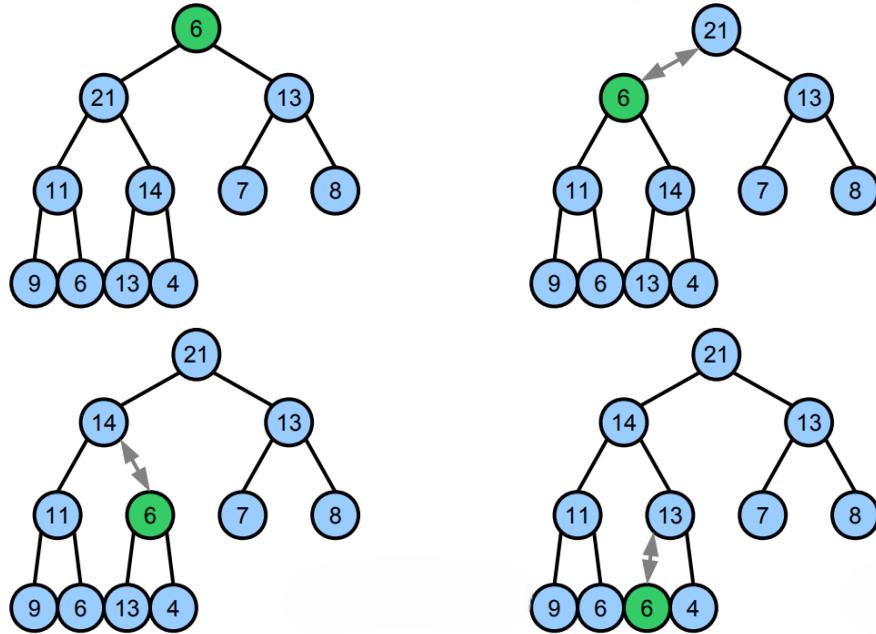
```
void heapify(Comparable A[], int n, int i) {
    if (i > n) return
    heapify(A, n, 2 * i) // heapify left child
    heapify(A, n, 2 * i + 1) // heapify right child
    fixHeap(A, n, i)
}
```

FixHeap

L'idea è quella di partire dalla radice di indice i e scambiarla con il figlio di valore massimo in maniera ricorsiva fino a quando le proprietà di max-heap non vengono ristabilite.



Albero con figlio destro e sinistro che rispettano il max-heap.



Esempio di fixHeap su albero con radice fuori posto.

Lo **pseudocodice** di tale funzione è il seguente:

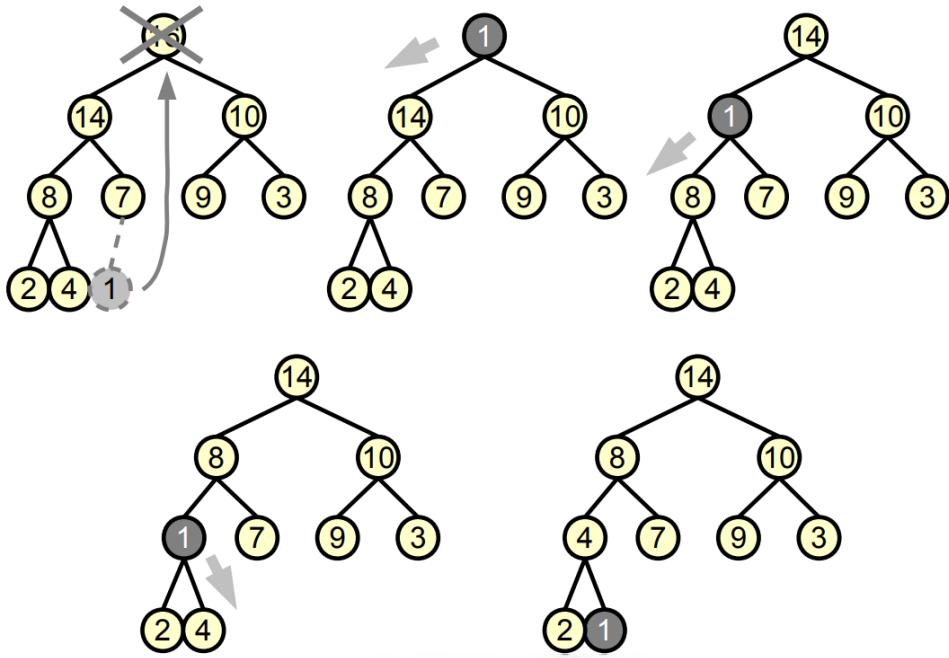
```

void fixHeap(Comparable S[], int c, int i) {
    int max = 2 * i
    if (2 * i > c) return
    if (2 * i + 1 <= c && S[2 * i].compareTo(S[2 * i + 1]) < 0) {
        // left child < right child
        max = 2 * i + 1
    }
    if (S[i].compareTo(S[max]) < 0) {
        // switch S[i] and S[max]
        Comparable temp = S[max]
        S[max] = S[i]
        S[i] = temp
        fixHeap(S, c, max)
    }
}

```

DeleteMax

L'idea è quella di sostituire il valore presente nella radice dell'albero con quello dell'elemento in ultima posizione nell'array heap, per poi richiamare la funzione fixHeap per ripristinare la proprietà di heap.



Esempio di deleteMax.

Costi computazionali

- **FindMax**: $O(1)$.
- **Heapify** (Master Theorem): $O(n)$.
- **FixHeap** (nel caso pessimo, il numero di scambi è uguale alla profondità dell'heap): $O(\log n)$.
- **DeleteMax** (nel caso pessimo, il numero di scambi è uguale alla profondità dell'heap): $O(\log n)$

HeapSort

È possibile utilizzare la struttura dati heap e le sue operazioni basilari al fine di **ordinare un array**. Per fare ciò occorre seguire i seguenti passaggi:

1. Richiamare la funzione heapify sull'array da ordinare al fine di farlo diventare un array heap.
2. Estrarre il massimo dall'array heap tramite la funzione deleteMax ed inserirlo in ultima posizione dell'array.
3. Ripetere il punto 2 finché l'heap non diventa vuoto.

Lo **pseudocodice** della funzione heapSort è il seguente:

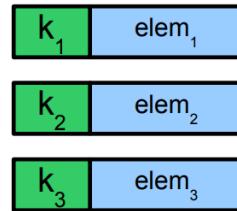
```
void heapSort(Comparable S[]) {
    heapify(S, S.length - 1, 1)
    for (int c = (S.length - 1); c > 0; c--) {
        Comparable k = findMax(S)
        deleteMax(S, c)
        S[c] = k
    }
}
```

Il costo computazionale di heapSort è dettato dalla funzione heapify, la quale ha un costo computazionale $O(n)$, e dal for, ciascuna delle cui iterazioni ha costo $O(\log c)$. Troviamo dunque che il costo computazionale è:

$$O(n) + O\left(\sum_{c=n}^1 \log c\right) = O(n \log n)$$

▼ 8.2 - Code con priorità

Una **coda con priorità** è una struttura dati che contiene un insieme di elementi ai quali sono associate delle chiavi che presentano una relazione d'ordine, ovvero sono ordinabili.



Coda con priorità.

Una tale struttura dati può essere utile ad esempio nella **gestione della banda di trasmissione**, in quanto nel routing dei pacchetti in rete è importante processare per primi i pacchetti con priorità più alta, dunque tali pacchetti possono essere inseriti in una coda con priorità.

Una coda con priorità inoltre può essere implementata in due modi: la prima implementazione corrisponde ad una semplice modifica della struttura dati heap, mentre la seconda consiste nell'utilizzare gli heap binomiali e gli heap di Fibonacci, i quali però non verranno trattati.

Operazioni basilari

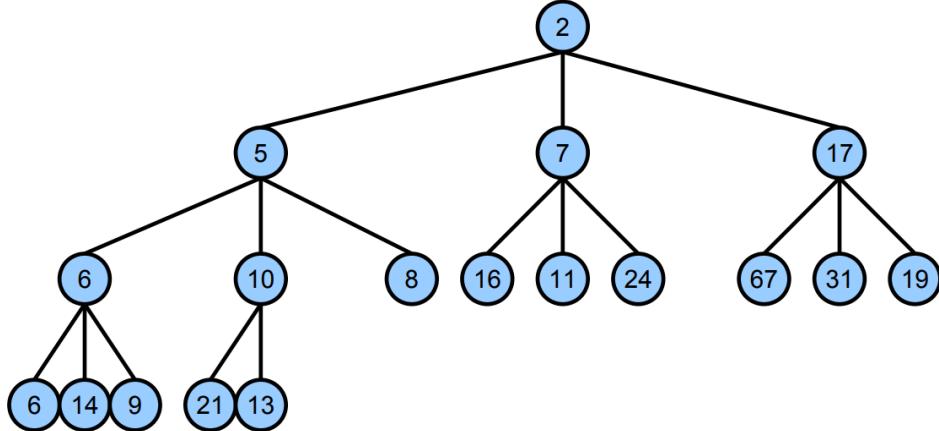
Le **operazioni basilari** per una coda con priorità sono le seguenti:

- **findMin()**: restituisce l'elemento associato alla chiave minima.
- **insert(e, k)**: inserisce un nuovo elemento e con associata la chiave k .
- **delete(e)**: rimuove l'elemento e .
- **deleteMin()**: rimuove l'elemento associato alla chiave minima.
- **increaseKey(e, c)**: incrementa la chiave dell'elemento e di un valore c .
- **decreaseKey(e, c)**: decrementa la chiave dell'elemento e di un valore c .

D-heap

Un **d-heap** corrisponde ad un heap basato su un albero d-ario, dunque non solo binario, in cui ogni nodo diverso dalla radice ha chiave maggiore o uguale a quella del padre.

Osservazione: un d-heap con N nodi ha altezza $O(\log_d N)$ e $N > \sum_{i=0}^{h-1} d^i = \frac{d^h - 1}{d - 1}$ = numero nodi di un albero completo con altezza $h - 1$.



Esempio di d-heap con $d = 3$.

D-heap array

È possibile rappresentare un qualunque albero d-heap utilizzando un array A strutturato in questo modo:

- $A[1]$ = radice dell'albero.
- Il livello h inizia in $1 + \sum_{i=0}^{h-1} d^i = 1 + \frac{d^h - 1}{d - 1}$.
- Il livello h termina in $d^h + \sum_{i=0}^{h-1} d^i = d^h + \frac{d^h - 1}{d - 1}$.
- $firstChild(i) = ((i - 1) \cdot d) + 2$.
- $lastChild(i) = (i \cdot d) + 1$.
- $parent(i) = Math.floor(\frac{i-1}{d})$.

Proprietà fondamentale dei d-heap

La radice di un d-heap contiene l'elemento con chiave minima.

Dimostrazione

Effettuiamo una dimostrazione per induzione sul numero di nodi n di un d-heap:

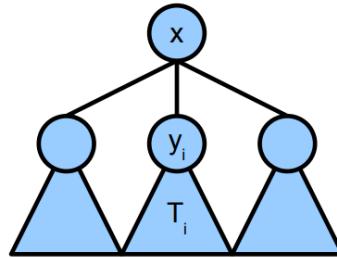
- Caso $n = 0$ o $n = 1$

La proprietà vale.

- Caso $n > 1$

Supponiamo per ipotesi induttiva che la proprietà vale per qualunque d-heap con al massimo $n - 1$ nodi.

Consideriamo la seguente figura:



Sappiamo per ipotesi induttiva che y_i ha la chiave minima nell'albero T_i , il quale ha $n - 1$ nodi.

Per la definizione di d-heap sappiamo inoltre che la chiave in $x \leq$ della chiave di ciascun figlio, dunque la chiave in x ha il valore minimo dell'intero heap.

Operazioni

Operazioni ausiliarie

- **MuoviAlto**: muove un elemento in alto nella coda con priorità fino a ristabilire l'heap.

Lo pseudocodice di tale funzione è il seguente:

```
void muoviAlto(v) {
    while(v != root(T) && chiave(v) < chiave(padre(v))) {
        scambia di posto v e padre(v) in T
        v = padre(v)
    }
}
```

Costo computazionale nel caso **peggiore** (v viene spostato da una foglia alla radice): $\log_d n$.

- **MuoviBasso**: muove un elemento in basso nella coda con priorità fino a ristabilire l'heap.

Lo pseudocodice di tale funzione è il seguente:

```
void muoviBasso(v) {
    while (true) {
        if (v non ha figli) return
        else {
            sia u il figlio di v con la minima chiave
            if (chiave(u) < chiave(v)) {
                scambia di posto u e v
                v := u;
            } else return
        }
    }
}
```

Costo computazionale nel caso **peggiore** (v viene spostato dalla radice a una foglia e per ogni spostamento vengono controllati tutti i d figli per trovare quella con la chiave minima): $d \cdot \log_d n$

FindMin

In base alla proprietà fondamentale dei d-heap, la radice di una coda con priorità è l'elemento con chiave minima, dunque findMin deve ritornare la radice.

Il costo computazionale: $O(1)$.

Insert

Si inserisce il nuovo nodo come ultima foglia a destra della coda. In questo modo viene rispettata la proprietà di struttura della coda. Per rispettare anche la proprietà di ordine, occorre eseguire muoviAlto sul nuovo nodo.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto): $O(\log_d n)$.

Delete

Viene sostituito il nodo da eliminare con il nodo presente nell'ultima foglia a destra della coda con priorità. A questo punto di esegue su tale nodo le operazioni muoviAlto e muoviBasso, una delle quali terminerà immediatamente.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto o muoviBasso, a seconda di quale viene eseguita): $O(d \cdot \log_d n)$.

DeleteMin

Viene richiamata la funzione delete sulla radice dell'albero, la quale ha chiave minima per la proprietà fondamentale dei d-heap.

Il **costo computazione** nel caso **peggiore**: $O(d \cdot \log_d n)$.

IncreaseKey

Viene incrementata la chiave del nodo passato in input e successivamente viene richiamata la funzione muoviBasso su di esso.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviBasso): $O(d \cdot \log_d n)$.

DecreaseKey

Viene decrementata la chiave del nodo passato in input e successivamente viene richiamata la funzione muoviAlto su di esso.

Il **costo computazionale** nel caso **peggiore** (dettato dalla funzione muoviAlto): $O(\log_d n)$.

▼ 9.0 - Union-find

▼ 9.1 - Introduzione all'union-find

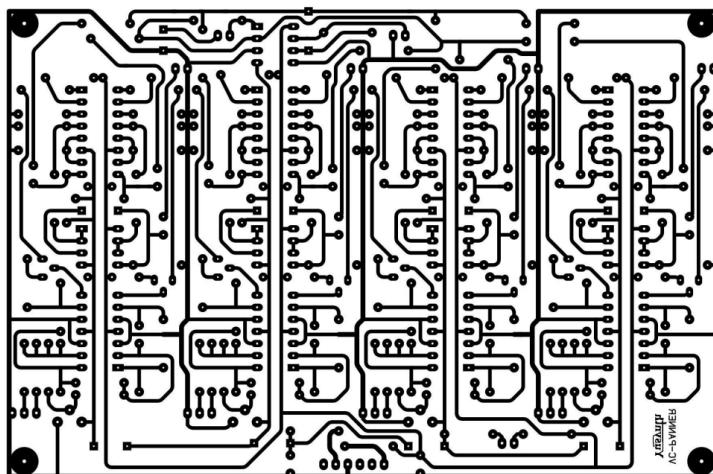
Definizione

Per arrivare al concetto di struttura dati union-find occorre precisare che tutto, anche un singolo elemento, viene considerato un insieme.

Una struttura dati **union-find** è una collezione $S = \{S_1, S_2, \dots, S_k\}$ di insiemi dinamici disgiunti in cui ogni insieme è identificato da un rappresentante univoco.

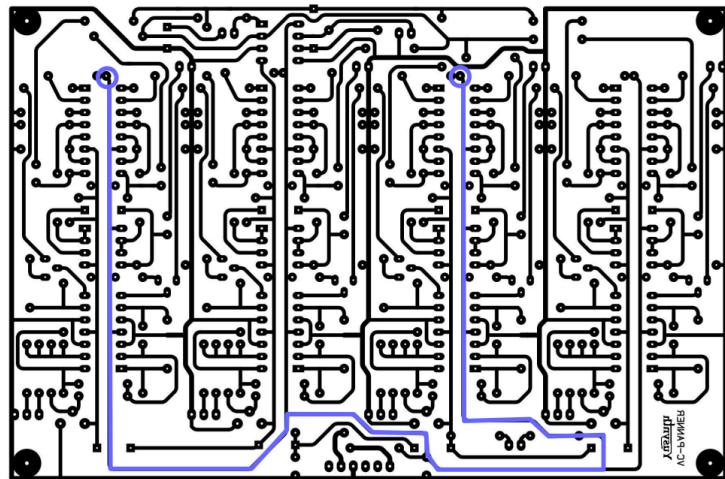
Il **rappresentante** di un insieme può essere un qualsiasi membro dell'insieme, ed essendo univoco qualsiasi operazione di ricerca del rappresentante in un certo insieme deve restituire sempre lo stesso risultato. Tale rappresentante inoltre può cambiare solo nel caso di unione con un altro insieme.

Possibili applicazioni



Circuito elettronico.

Una possibile applicazione della struttura dati union-find può essere trovata nell'ambito dei **circuiti elettronici**, nei quali è possibile creare degli insiemi che contengono i pin collegati da segmenti conduttori, e in questo modo poter controllare tramite le operazioni elementari per union-find se due di questi pin sono contenuti nello stesso insieme, ovvero se esiste una strada percorribile che li connette.



Operazioni elementari

- **makeSet(x)**: crea un nuovo insieme il cui unico elemento è rappresentato è x (x non deve appartenere a nessun altro insieme).
- **find(x)**: restituisce il rappresentante dell'insieme che contiene x.
- **union(x, y)**: unisce i due insiemi rappresentati da x e da y, eliminando i due insiemi x e y e scegliendo un rappresentante univoco per il nuovo insieme.

▼ 9.2 - QuickFind e QuickUnion

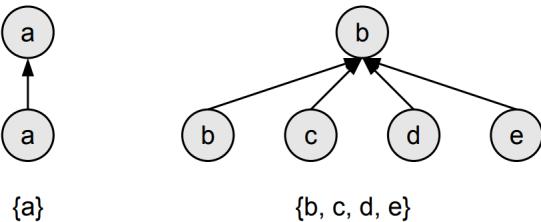
È possibile rappresentare una struttura dati union-find tramite diverse tipologie di implementazione. Noi analizzeremo le due implementazioni **QuickFind** e **QuickUnion**, che, come suggerito dal nome, rendono asintoticamente veloci rispettivamente le operazioni find e union:

- **QuickFind**: alberi di altezza 1
 - makeSet: $O(1)$.
 - find: $O(1)$.
 - union: $O(n)$.
- **QuickUnion**: alberi generali
 - makeSet: $O(1)$.
 - find: $O(n)$.
 - union: $O(1)$.

Il consiglio è dunque quello di utilizzare l'implementazione QuickFind quando le find sono frequenti e le union sono rare, altrimenti utilizzare QuickUnion.

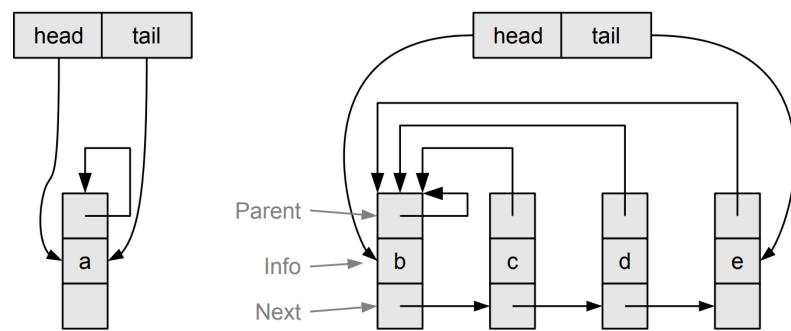
QuickFind

L'implementazione **QuickFind** utilizza **alberi di altezza 1** nelle quali foglie sono contenuti gli elementi dell'insieme, mentre nella radice è contenuto il rappresentante dell'insieme.



Nota implementativa

È possibile rappresentare gli alberi di altezza 1 tramite liste in cui ogni elemento ha un puntatore che punta al rappresentante dell'insieme.



Costi computazionali

Le operazioni **makeSet(x)** e **find(x)** hanno un costo computazionale $O(1)$ in quanto makeSet crea un albero in cui l'unica foglia è x e la radice è il rappresentante di x, ovvero x stesso, e find restituisce il puntatore al padre di x.

L'operazione **union(x, y)** richiede invece che tutte le foglie dell'albero y vengano spostate nell'albero x, quindi, avendo y nel caso peggiore $n - 1$ foglie, con n il numero degli elementi nel nuovo insieme, il costo computazionale è $O(n)$. È comunque possibile abbassare il costo computazionale della funzione union utilizzando un'euristica sul peso.

Euristica sul peso

Una **strategia** per diminuire il costo computazionale dell'operazione union è quella di memorizzare negli alberi QuickFind la dimensione di tali alberi (tale operazione di mantenimento della dimensione può avvenire con un costo $O(1)$) e ogni volta che avviene un union appendere l'insieme con meno elementi all'insieme con più elementi.

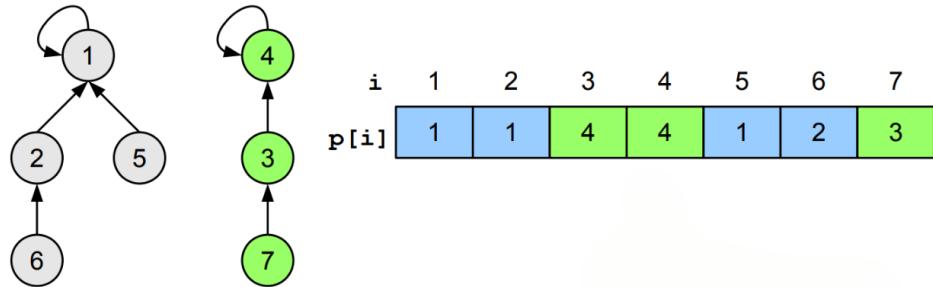
In questo modo **osserviamo** che ogni foglia che acquista un nuovo padre farà parte di un insieme almeno il doppio più grande dell'insieme di partenza, dunque una foglia facente parte di un insieme di n elementi ha cambiato padre al più $\log n$ volte. In base a questa osservazione, calcoliamo il **costo ammortizzato** dell'operazione union, ricordando che è dato dal costo complessivo di k esecuzioni diviso k . Considerando un insieme di dimensione n creato con $n - 1$ unioni, ovvero il numero massimo di unioni possibili, e basandoci sull'osservazione di prima notiamo che per costruirlo tramite operazioni di union tutte le n foglie hanno avuto al massimo $\log n$ cambi di padre. Il costo ammortizzato risulta quindi essere $\frac{O(n \cdot \log n)}{n-1} = O(\log n)$.

QuickUnion

L'implementazione **QuickUnion** utilizza alberi generici che nella radice contengono il rappresentante e i quali figli sono alberi che corrispondono ai sottoinsiemi contenuti nell'insieme generale.

Nota implementativa

Un modo per rappresentare alberi generici tramite un array è quello di utilizzare un cosiddetto **vettore dei padri**, il quale per ogni indice dell'array viene rappresentato un elemento dell'albero, e il contenuto di tale nodo indica l'indice del nodo padre.



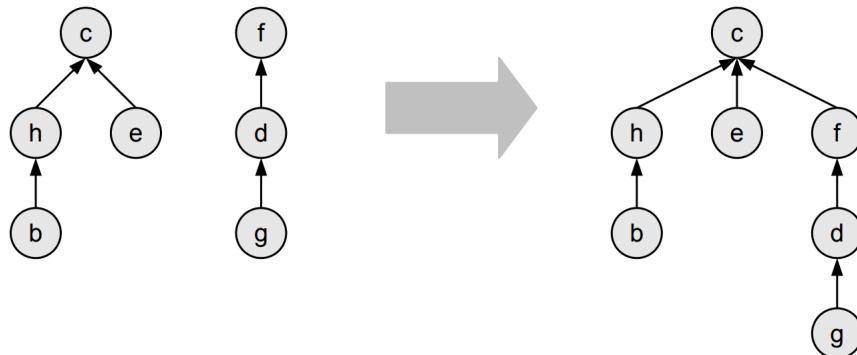
Vettore dei padri di due alberi generici.

Costi computazionali

La funzione **makeSet(x)** crea un albero che ha un unico nodo x , dunque il costo è $O(1)$.

La funzione **find(x)** invece deve ripercorrere l'albero partendo da x e arrivando alla radice, dunque il costo è $O(n)$.

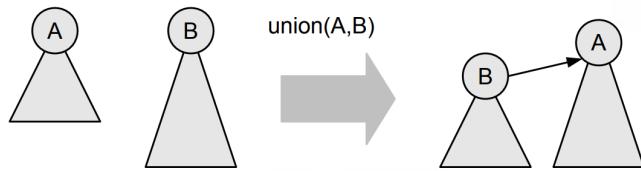
La funzione **union(x, y)** deve inserire l'albero rappresentato da y come figlio della radice x , quindi ha costo $O(1)$.



Esecuzione di $\text{union}(c, f)$.

Euristica sul rango

Visto che il costo computazionale dell'operazione **find** è dato dall'altezza dell'albero che rappresenta l'insieme, allora è possibile pensare ad una **strategia** che abbassi tale costo evitando di creare alberi troppo alti. È possibile fare ciò memorizzando il **rango** del nodo radice, ovvero il numero di archi del cammino più lungo tra il nodo in input e una foglia sua discendente (tale operazione di memorizzazione del rango può avvenire con un costo $O(1)$) e quando avviene un'operazione **union** appendiamo l'albero con la radice con rango minore a quello con la radice con rango maggiore.



Denotando il rango di un nodo x tramite la funzione $rank(x)$ e sia n il numero di nodi contenuti in un albero costruito tramite euristica sul rango, è possibile dimostrare che $n \geq 2^{rank(x)}$, dove x è il nodo radice dell'albero.

Siccome il costo della funzione find dipende dall'altezza dell'albero, e tale altezza h è uguale a $rank(x)$, allora per l'affermazione precedente abbiamo che $h \leq \log n$, e quindi la funzione find ha un costo computazionale $\leq \log n$, quindi $O(\log n)$.

Riassunto costi computazionali

	QuickFind	QuickUnion	QuickFind eur. peso	QuickUnion eur. rank
makeSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$
union	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
find	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di una struttura dati union-find.

▼ 10.0 - Tecniche algoritmiche

Esistono diverse **tecniche algoritmiche** ognuna delle quali permette di risolvere problemi di simile fattura. Noi vedremo le seguenti 3 tecniche algoritmiche:

- **Divide-et-impera**

Un problema viene suddiviso in sotto-problemi che vengono risolti ricorsivamente, approccio top-down.

- **Algoritmi greedy**

Ad ogni passo si fa sempre la scelta che al momento sembra ottima.

- **Programmazione dinamica**

La soluzione viene costruita a partire dalle soluzioni di un insieme di sotto problemi, approccio bottom-up.

▼ 10.1 - Divide-et-impera

Struttura di un algoritmo divide-et-impera

Gli algoritmi **divide-et-impera** sono in genere formati dalle seguenti **3 fasi**:

- **Divide**: viene suddiviso il problema di partenza in sotto-problemi di dimensione minore.
- **Impera**: vengono risolti i sotto-problemi in maniera ricorsiva.
- **Combina**: vengono unite le soluzioni dei sottoproblemi per costruire la soluzione del problema di partenza.

Esempio: torre di Hanoi

Il problema della **torre di Hanoi** è un gioco matematico che consiste nell'avere 3 pioli sistemati da sinistra verso destra. Nel piolo di destra sono presenti n dischi di dimensione diversi impilati in ordine decrescente con il grande in basso e il più piccolo in alto. Lo scopo del gioco è quello di impilare tutti gli n dischi in maniera decrescente nel piolo di sinistra potendo spostare un solo disco alla volta e senza mai impilare un disco più grande sopra un disco più piccolo, utilizzando se serve anche il piolo centrale.



Lo **pseudocodice** dell'algoritmo divide-et-impera di tale problema è il seguente:

```
void Hanoi(Stack p1, Stack p2, Stack p3, integer n) {
    if (n = 1) p3.push(p1.pop())
    else {
        hanoi(p1, p3, p2, n - 1)
        p3.push(p1.pop())
        hanoi(p2, p1, p3, n - 1)
```

}

Le **fasi del divide-et-impera** di questo algoritmo sono le seguenti:

- **Divide**

1. $n - 1$ dischi da p1 a p2.
2. 1 disco da p1 a p3.
3. $n - 1$ dischi da p2 a p3.

- **Impera**

Esecuzione ricorsiva degli spostamenti.

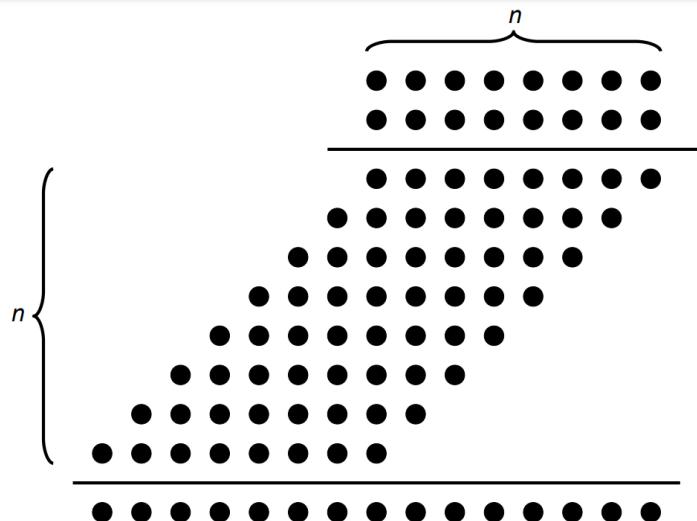
Per calcolare il **costo computazionale** occorre risolvere l'equazione di ricorrenza:

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

È possibile dimostrare tale soluzione al problema della torre di Hanoi ha un costo computazionale **esponenziale**, dunque se venisse effettuata un operazione al secondo, per trasferire 64 dischi dal piolo di sinistra a quello di destra servirebbe un tempo pari a circa 127 volte l'età del nostro sole.

Esempio: moltiplicazione di interi

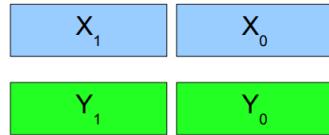
Consideriamo due interi ad n cifre X e Y . Notiamo che l'algoritmo di "moltiplicazione in colonna" ha costo $O(n^2)$ in quanto ogni cifra di Y deve essere moltiplicata per ogni cifra di X .



Metodo della "moltiplicazione in colonna".

Ci chiediamo dunque se è possibile fare di meglio con un algoritmo più efficiente.

Dividiamo i due numeri X e Y in due parti uguali, ottenendo dunque $X = X_1 \cdot 10^{n/2} + X_0$ e $Y = Y_1 \cdot 10^{n/2} + Y_0$.



Calcoliamo a questo punto il prodotto di X e Y :

$$\begin{aligned} X \cdot Y &= (X_1 \cdot 10^{n/2} + X_0) \cdot (Y_1 \cdot 10^{n/2} + Y_0) \\ &= (X_1 Y_1) \cdot 10^n + (X_1 Y_0 + X_0 Y_1) \cdot 10^{n/2} + (X_0 Y_0) \cdot 10^n \end{aligned}$$

Sapendo che la moltiplicazione per 10^n ha un costo computazionale $O(n)$ in quanto equivale ad uno shift a sinistra di n posizioni e notando che l'operazione contiene 4 prodotti di numeri a $n/2$ cifre, otteniamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 4 \cdot T(n/2) + n & n > 1 \end{cases}$$

Tale equazione è risolvibile utilizzando il Master Theorem, con il quale si ottiene un costo computazionale $O(n^2)$, equivalente a quello del metodo della "moltiplicazione in colonna".

Notiamo però che se poniamo P_1, P_2 e P_3 come segue:

$$\begin{aligned} P_1 &= (X_1 + X_0) \cdot (Y_1 + Y_0) = X_1 Y_0 + Y_1 X_0 + X_1 Y_1 + X_0 Y_0 \\ P_2 &= X_1 Y_1 \\ P_3 &= X_0 Y_0 \end{aligned}$$

possiamo porre $X \cdot Y$ nel seguente modo:

$$X \cdot Y = P_2 \cdot 10^n + (P_1 - P_2 - P_3) \cdot 10^{n/2} + P_3 \cdot 10^n$$

Osserviamo quindi che il calcolo di $X \cdot Y$ richiede in tutto solo 3 prodotti (P_1, P_2 e P_3). Riscriviamo quindi l'equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 \cdot T(n/2) + n & n > 1 \end{cases}$$

Otteniamo dunque, utilizzando il Master Theorem, un costo computazionale equivalente a $O(n^{\log_2 3}) \approx O(n^{1.59})$.

Esempio: sottovettore di valore massimo

Il problema consiste nel, dato in input un vettore v di lunghezza n contenente n valori arbitrari, individuare il sottovettore non vuoto di v la **somma dei cui elementi sia massima** e restituire tale somma.

3	-5	10	2	-3	1	4	-8	7	-6	-1
---	----	----	---	----	---	---	----	---	----	----

Una prima soluzione di tale problema può essere quella di utilizzare un sistema di **forza bruta**, il quale analizza tutti i possibili sotto vettori del vettore di partenza ed individua quello con valore massimo. Lo **pseudocodice** di un tale algoritmo è il seguente:

```

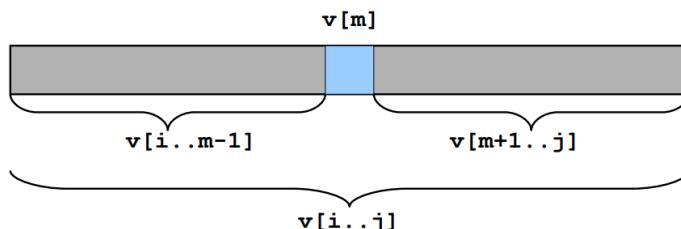
double SommaMax(double v[1 ... n]) {
    double smax = v[1]
    for (int i = 1; i < n; i++) {
        double s = 0
        for integer (int j = i; j < n; j++) {
            s = s + v[j]
            if (s > smax) smax = s
        }
    }
    return smax
}

```

Il **costo computazionale** di questo algoritmo è $O(n^2)$.

È possibile però migliorare questo costo utilizzando una tecnica divide-et-impera, la quale consiste nel dividere l'array in due parti, separate dall'elemento centrale $v[m]$, e ricadendo in 3 casistiche:

- Il sottoarray con valore massimo si trova nell'array di sinistra $v[i \dots m - 1]$.
- Il sottoarray con valore massimo si trova nell'array di destra $v[m + 1 \dots j]$.
- Il sottoarray con valore massimo si trova a cavallo tra i due array.



Occorre dunque calcolare il valore della somma degli elementi di questi 3 sottoarray e confrontarli. Per i primi 2 è possibile farlo richiamando lo stesso algoritmo in maniera ricorsiva, mentre per il terzo il calcolo è più complicato. Notiamo che tale sotto-array può contenere una parte prima di $v[m]$ e una parte dopo di $v[m]$. Occorre dunque calcolare il sottoarray sa con valore massimo, incluso quello vuoto, che abbia come ultimo elemento $v[m - 1]$ e quello sb con valore massimo, incluso quello vuoto, che abbia come primo elemento $v[m + 1]$. Otteniamo dunque che il sottoarray con valore massimo a cavallo tra i due sottoarray è equivalente a $sa + v[m] + sb$.

Lo **pseudocodice** che sfrutta tale algoritmo è dunque il seguente:

```

double sommaMax(double v[1 ... n], int i, int j) {
    if (i > j) return 0
    else if (i == j) return v[i]
    else {
        m = Math.floor((i + j) / 2)

        double l = sommaMax(v, i, m - 1)
        double r = sommaMax(v, m + 1, j)

        double sa = 0, sb = 0, s = 0
        for (int k = m - 1; k >= i; k--) {
            s = s + v[k]
            if (s > sa) sa = s
        }
        s = 0;
        for (int k = m + 1; k <= j; k++) {
            s = s + v[k]
            if (s > sb) sb = s
        }

        return max(l, r, sa + v[m] + sb)
    }
}

```

```
    }  
}
```

Il **costo computazionale** di tale algoritmo, dimostrabile tramite l'utilizzo del Master Theorem, è $O(n \log n)$.

▼ 10.2 - Greedy

La tecnica algoritmica **greedy** consiste nel fare ad ogni passo la scelta che sembra ottima.

Tale tecnica può essere utilizzata nel caso in cui è possibile dimostrare che tra le scelte disponibili ne esiste una semplice che porta alla soluzione ottima, e che fatta tale scelta resta un sottoproblema della stessa struttura del problema principale, risolvibile a sua volta tramite una scelta greedy.

Esempio: problema del resto

Dato un intero R che rappresenta un importo in centesimi da erogare, scegliere un numero minimo di monete necessarie per erogare tale importo utilizzando solo i seguenti tagli: 50c, 20c, 10c, 5c, 2c, 1c.

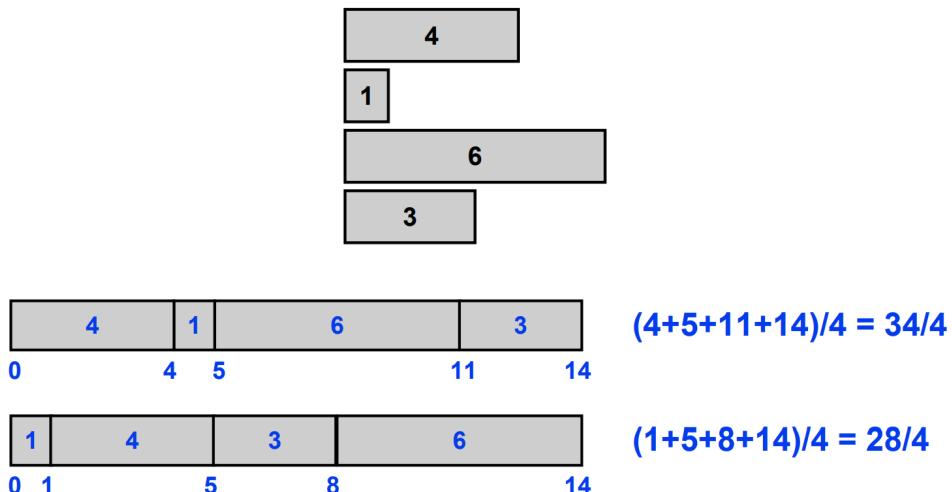
Per un sistema monetario di questo tipo è possibile adottare un algoritmo di tipo greedy, il quale consiste nel prendere di volta in volta la moneta più grande contenuta nella cifra da erogare. I sistemi monetari che consentono un approccio di tipo greedy sono detti **sistemi monetari canonici**, ma esistono anche altri tipo di sistemi monetari per i quali l'approccio greedy non è quello ottimale, ad esempio per erogare 6 con monete da 4, 3 e 1 (greedy: $4 + 1 + 1$, ottimo: $3 + 3$).

Lo **pseudocodice** dell'algoritmo di tipo greedy per questa tipologia di problemi è il seguente:

```
// R = resto da erogare  
// T[1 ... n] = gli n tagli di monete a disposizione  
// output = numero totale di monete da erogare  
int restoGreedy(int R, int T[1 ... n]) {  
    int coinNum = 0  
    int i = 1  
  
    while (R > 0 && i <= n) {  
        if (R >= T[i]) {  
            R -= T[i]  
            coinNum++  
        } else i++  
    }  
  
    if (R > 0) errore: resto non erogabile // es. il taglio minore è > 1c  
    else return coinNum  
}
```

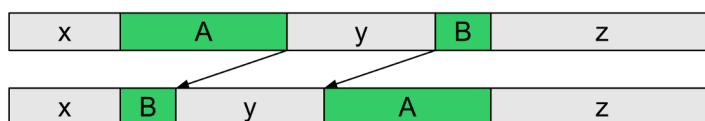
Esempio: problema di scheduling

Un problema risolvibile tramite un algoritmo di tipo greedy consiste nel **minimizzare il tempo medio di completamento** di un insieme di operazioni. Ad esempio questo problema è rilevante nello scheduling del job di un processore. Molto spesso infatti un singolo processore si ritrova a dover gestire un numero n di job ognuno dei quali ha un proprio tempo di esecuzione. Per massimizzare l'efficienza generale dell'elaboratore e sfruttare al massimo le capacità del processore è utile minimizzare il tempo medio di completamento, il quale si calcola tramite la somma dei tempi di completamento dei singoli job diviso il numero di job completati.



L'algoritmo greedy per questo tipo di problemi consiste nell'eseguire n passi, in ognuno dei quali si manda in esecuzione il job, tra quelli che rimangono, che richiede meno tempo.

Possiamo dimostrare la correttezza di questo ragionamento tramite un esempio:



Osserviamo che i tempi di completamento dei job x e z rimangono uguali, e il tempo di completamento di A nella seconda soluzione è uguale al tempo di completamento di B nella prima. Notiamo però che il tempo di completamento di B nella seconda soluzione è minore rispetto al tempo di completamento di A nella prima soluzione, e che il tempo di completamento di y è diminuito. Possiamo quindi concludere che il tempo medio di completamento è diminuito.

Esempio: problema della compressione

Il **problema della compressione** consiste nel dover rappresentare un insieme di caratteri tramite una certa codifica (ad ogni carattere deve essere assegnato un certo codice binario) in modo tale da minimizzare la lunghezza di un insieme di caratteri codificati.

Nel fare ciò dobbiamo dunque tenere conto della probabilità con cui viene utilizzato un certo caratteri, in modo tale da assegnare ad un carattere molto frequente una codifica con meno bit.

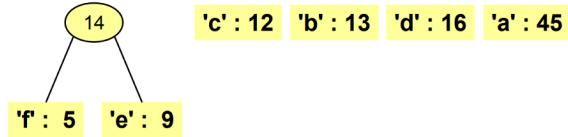
Inoltre, visto che utilizziamo una **codifica a lunghezza variabile** per il motivo appena accennato, dobbiamo tenere in mente che nessun codice può essere un **prefisso** di un altro codice. Ad esempio possiamo notare che la codifica $f(a) = 1$, $f(b)=10$, $f(c)=101$ risulta ambigua in quanto il codice 101 può essere decodificato sia come "c" che come "ba", in quanto la codifica di "b" è un prefisso della codifica di "c".

I **codici di Huffman** sono delle tipologie di codifiche realizzabili utilizzando un algoritmo greedy. Tali codifiche si realizzano seguendo i seguenti step:

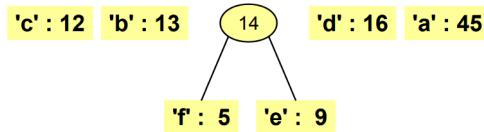
1. Costruire una lista di nodi, in cui ogni nodo contiene un carattere e la sua frequenza (es. numero di volte in cui compare all'interno di un pdf), ordinata in base alla frequenza.

'f' : 5 'e' : 9 'c' : 12 'b' : 13 'd' : 16 'a' : 45

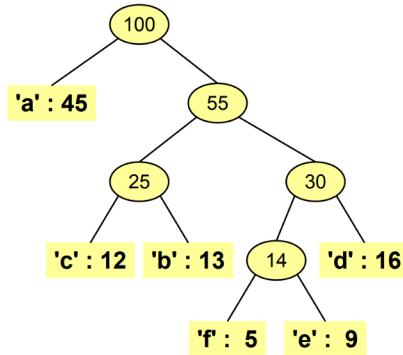
2. Rimuovere dalla lista i due nodi con la frequenza minore e collegarli ad un nodo padre etichettato con la somma delle frequenze dei suoi due figli.



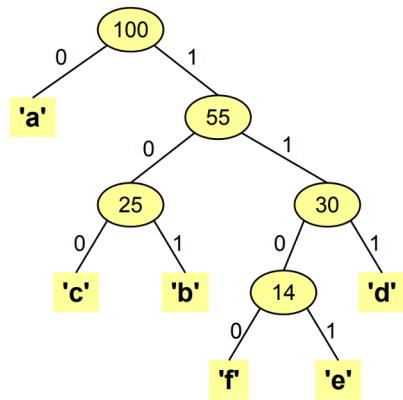
3. Aggiungere l'albero appena creato alla lista in posizione adatta in modo da mantenere una lista ordinata.



4. Ripetere i passi 2 e 3 fino a quando non rimane un solo albero.



5. Etichettiamo gli archi sinistri dell'albero con uno 0 e gli alberi destri con un 1.



Una volta costruito l'albero possiamo associare ogni carattere al suo codice unendo i numeri presenti negli archi necessari per passare dalla radice al nodo contenente il carattere. Ad esempio nell'albero appena costruito il carattere 'f' viene codificato con 1100.

Lo **pseudocodice** per costruire un tale albero è il seguente:

```

Tree huffman(float f[1 ... n], char c[1 ... n]) {
    Q = new MinPriorityQueue()

    // insert single nodes into the queue
    for (int i = 1; i < n; i++) {
        z = new TreeNode(f[i], c[i])
        Q.insert(f[i], z)
    }

    // create the tree
    for (int i = 1; i < n - 1; i++) {
        z1 = Q.findMin()
        Q.deleteMin()
        z2 = Q.findMin()
        Q.deleteMin()

        z = new TreeNode(z1.f + z2.f, '')
        z.left = z1
        z.right = z2

        Q.insert(z1.f + z2.f, z)
    }

    // return the tree
    return Q.findMin()
}

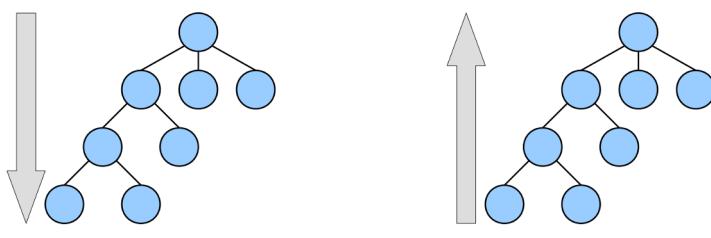
```

▼ 10.3 - Programmazione dinamica

Programmazione dinamica vs divide-et-impera

La **programmazione dinamica** consiste nel risolvere un problema combinando in maniera appropriata le soluzioni dei suoi sottoproblemi.

A differenza degli algoritmi divide-et-impera però la programmazione dinamica è una **tecnica iterativa** invece che ricorsiva e utilizza un approccio **bottom-up** piuttosto che top-down. Questa è vantaggiosa quando ci sono **sottoproblemi ripetuti**, mentre il divide-et-impera è utile quando i sottoproblemi sono indipendenti.



Approccio divide-et-impera (destra) vs programmazione dinamica (sinistra).

Esempio: algoritmo di Fibonacci

Divide-et-impera

Un primo approccio per calcolare un numero di Fibonacci è quello di utilizzare la tecnica del **divide-et-impera**, utilizzando la definizione di numero di Fibonacci e quindi calcolando in maniera ricorsiva i due numeri di Fibonacci precedenti.

Lo **pseudocodice** è il seguente:

```

int fibRic(int n) {
    if ((n = 1) || (n = 2)) return 1
    else return fibRic(n - 1) + fibRic(n - 2)
}

```

Il **costo computazionale** è $O(2^n)$.

È possibile migliorare questo risultato notando che alcuni numeri vengono calcolati più volte, dunque utilizzando una tecnica di **memorizzazione** si evita di ricalcolarsi.

Lo **pseudocodice** è il seguente:

```

int cache[n]

int fibMemorization(int n) {
    for (i = 0; i < n; i++) cache[i] = -1
    fibMem(n)
}

int fibMem(int n) {
    if (cache[n] != -1) return cache[n]

    if ((n = 1) || (n = 2)) cache[n] = 1
    else cache[n] = fibMem(n - 1) + fibMem(n - 2)

    return cache[n]
}

```

Il **costo computazionale** è $O(n)$.

Programmazione dinamica

Un secondo approccio consiste nell'utilizzare la **programmazione dinamica**, risolvendo tramite un approccio bottom-up prima tutti i numeri di Fibonacci precedenti per poi calcolare il numero dato in input seguendo la definizione.

Lo **pseudocodice** è il seguente:

```

int fibIter(integer n) {
    if (n <= 2) return 1
    else {
        int f[1 ... n]
        f[1] = 1;
        f[2] = 1;

        for (int i = 3; i < n; i++)
            f[i] = f[i - 1] + f[i - 2]

        return f[n]
    }
}

```

Il **costo computazionale** è:

- **Tempo**: $O(n)$.
- **Spazio**: $O(n)$.

Visto che per calcolare ogni numero di Fibonacci ci occorrono solamente i due numeri di Fibonacci precedenti possiamo ottimizzare l'utilizzo della memoria facendo diventare il costo computazionale in termini di spazio costante.

Lo **pseudocodice** è il seguente:

```

int fib(int n) {
    if (n < 2) return 1
    else {
        int f[0 ... 2]
        f[1] = 1
        f[2] = 1

        for (int i = 2; i < n; i++) {
            f[0] = f[1]
            f[1] = f[2]
            f[2] = f[1] + f[0]
        }

        return f[2]
    }
}

```

Il costo computazionale è:

- **Tempo:** $O(n)$.
- **Spazio:** $O(1)$.

Esempio: sottovettore di valore massimo

Abbiamo già analizzato questo problema nella sezione degli algoritmi divide-et-impera. Gli algoritmi che abbiamo utilizzato sono stati 2, uno che utilizzava la tecnica della **forza bruta** e aveva un costo computazionale $O(n^2)$, e l'altro che utilizzava la tecnica del **divide-et-impera** il quale costo computazionale era $O(n \log n)$.

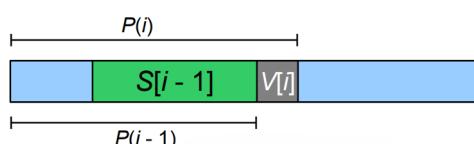
Possiamo trovare un'ulteriore soluzione a questo problema utilizzando la **programmazione dinamica**. Sia $V[1 \dots n]$ il vettore in input, consideriamo il sottoproblema $P(i)$ il quale calcola il valore massimo tra i sottovettori di V che abbiano come ultimo elemento $V[i]$. L'idea sarebbe quella di costruire un vettore $S[1 \dots n]$ che in posizione i presenta la soluzione del sottoproblema $P(i)$. La soluzione al problema originario è dunque il valore massimo contenuto nel vettore S .

A questo punto dobbiamo risolvere il sottoproblema $P(i)$:

- Caso $i = 1$
 $S[1] = V[1]$, in quanto è l'unico sottovettore vuoto possibile.
- Caso $i > 1$

Supponiamo di aver già risolto il sottoproblema $P(i - 1)$. Siccome nel risultato dobbiamo per forza includere $V[i]$, ci resta solo da valutare se aggiungere a $V[i]$ anche $S[i - 1]$ o meno.

Quindi $S[i] = \max(V[i], V[i] + S[i - 1])$.



A questo punto abbiamo costruito un vettore del seguente tipo:

V[]	3 -5 10 2 -3 1 4 -8 7 -6 -1
-----	---

S[]	3 -2 10 12 9 10 14 6 13 7 6
-----	---

Lo **pseudocodice** per fare costruire tale vettore e restituire il valore massimo è il seguente:

```
double sottovettoreMax(double V[1 ... n]) {
    double S[1 ... n]
    S[1] = V[1]

    int imax = 1
    for (int i = 2; i < n; i++) {
        if (S[i - 1] + V[i] >= V[i])
            S[i] = S[i - 1] + V[i]
        else S[i] = V[i]

        if (S[i] > S[imax]) imax = i
    }

    return S[imax]
}
```

Nel caso in cui volessimo conoscere quale sia il sottovettore con somma massima possiamo farlo osservando il vettore S . L'indice contenente l'elemento con valore massimo è l'indice di fine del sottoarray, mentre l'indice di inizio corrisponde a quello del primo elemento, partendo dall'indice di fine e andando verso sinistra, il quale valore corrisponde al valore presente nel vettore V .

V[]	3 -5 10 2 -3 1 4 -8 7 -6 -1
-----	---

S[]	3 -2 10 12 9 10 14 6 13 7 6
-----	---



Esempio: problema dello zaino

Il problema dello zaino consiste nell'avere un insieme $X = \{1, \dots, n\}$ di n oggetti, ognuno dei quali ha un peso $p[i]$ e $v[i]$. Disponiamo di un contenitore/zaino in grado di trasportare fino a un peso massimo P e dobbiamo determinare un sottoinsieme di X tale che il peso complessivo dei suoi oggetti sia minore di P , mentre il valore complessivo di essi sia il massimo possibile.

Greedy 1

La prima soluzione consiste nello scegliere ad ogni passo l'oggetto con il **valore massimo** tra quelli rimanenti nell'insieme X e tali per cui il loro peso sia minore o uguale alla capacità residua nello contenitore.

Questo algoritmo però non fornisce sempre la soluzione ottima.

Greedy 2

La seconda soluzione consiste nello scegliere ad ogni passo l'oggetto con il **valore specifico massimo** tra quelli rimanenti nell'insieme X e tali per cui il loro peso sia minore o uguale alla capacità residua nello contenitore.

Il valore specifico di un oggetto consiste nel rapporto tra il suo valore e il suo peso.

Anche questo algoritmo, pur essendo più preciso rispetto al primo, non fornisce sempre la soluzione ottima.

Programmazione dinamica

È possibile trovare una soluzione ottima per questo problema utilizzando la programmazione dinamica. Questo approccio consiste nel creare un vettore V di due dimensioni in cui per ogni elemento $V[i, j]$ inserire il valore del problema risolto per i primi i elementi di X e per la capacità j del contenitore.

Per riempire tale array ci ritroviamo i seguenti casi:

- $i = 1$

Se $p[1] > j$, allora $V[1, j] = 0$.

Se $p[1] \leq j$, allora $V[1, j] = v[1]$.

- $i > 1$

Supponiamo di avere già il valore $V[i - 1, j]$, dobbiamo scegliere se inserire il nuovo elemento i oppure no.

Se $p[i] > j$, allora non possiamo inserirlo, quindi $V[i, j] = V[i - 1, j]$.

Se $p[i] \leq j$ allora possiamo decidere se inserirlo, e in quel caso $V[i, j] = V[i - 1, j - p[i]] + v[i]$, oppure non inserirlo, quindi $V[i, j] = V[i - 1, j]$. Dobbiamo dunque scegliere il valore massimo tra le due opzioni.

Riassumendo abbiamo che:

$$V[i, j] = \begin{cases} V[i - 1, j] & p[i] > j \\ \max(V[i - 1, j - p[i]] + v[i], V[i - 1, j]) & p[i] \leq j \end{cases}$$

$$\begin{aligned} p &= [2, 7, 6, 4] \\ v &= [12.7, 6.4, 1.7, 0.3] \end{aligned}$$

		j										
		0	1	2	3	4	5	6	7	8	9	10
i	1	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	12.7	12.7	12.7
	2	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	12.7	19.1	19.1
	3	0.0	0.0	12.7	12.7	12.7	12.7	12.7	12.7	14.4	19.1	19.1
	4	0.0	0.0	12.7	12.7	12.7	12.7	13.0	13.0	14.4	19.1	19.1

$$V[3, 8] = \max(V[2, 6] + 1.7, V[2, 8]).$$

Il valore della soluzione del problema di partenza è dunque $V[n, P]$.

Per conoscere anche l'indice degli oggetti che vengono inseriti nel contenitore utilizziamo una tabella ausiliaria K della stessa grandezza della tabella V che riempiamo durante il riempimento di quest'ultima tramite un true se l'oggetto i viene inserito nel contenitore quando la capienza è j , altrimenti false.

	0	1	2	3	4	5	6	7	8	9	10
1	F	F	T	T	T	T	T	T	T	T	T
2	F	F	F	F	F	F	F	F	F	T	T
3	F	F	F	F	F	F	F	F	T	F	F
4	F	F	F	F	F	F	T	T	F	F	F

Successivamente utilizziamo il seguente algoritmo che fa uso della tabella K :

```

int j = P
int i = n
while (i > 0) {
    if (K[i, j]) {
        print("Seleziona oggetto ", i)
        j -= p[i]
    }
    i--
}
    
```

Esempio: Seam Carving

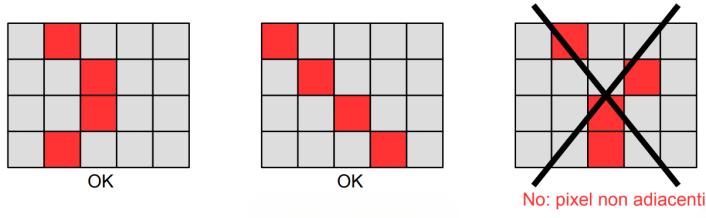
Il **Seam Carving** è un algoritmo di ridimensionamento delle immagini che utilizza l'importanza dei pixel e la programmazione dinamica per eliminare le parti meno rilevanti della figura.



Differenza tra le usuali tecniche di ridimensionamento di un'immagine e il Seam Carving.

Questa tecnica, per ogni immagine con M righe e N colonne da ridimensionare, utilizza un vettore E di due dimensioni $M \times N$ che per ogni pixel presenta un valore compreso tra 0 e 1 che ne rappresenta l'energia, ossia l'importanza calcolata solitamente in base al suo colore e al contrasto con i pixel adiacenti (0: poco importante, 1: molto importante).

L'obiettivo è quello di scegliere, per ogni riga/colonna da eliminare, la cucitura la cui somma delle energie dei pixel in essa contenuti sia minore, dove per cucitura si intende un cammino composto da pixel adiacenti.



Cuciture verticali.

Per risolvere il problema è dunque necessario creare un vettore W di due dimensioni, grande quanto il vettore E , in cui per ogni elemento $W[i, j]$ viene memorizzato il valore della cucitura con valore minimo che termini nel pixel $[i, j]$.

Per riempire tale vettore, prendendo in considerazione cuciture verticali, ci ritroviamo i seguenti casi:

- $i = 1$

L'ultimo pixel della cucitura si trova nella prima riga, dunque $W[1, j] = E[1, j]$.

- $i > 1$

- $j = 1$

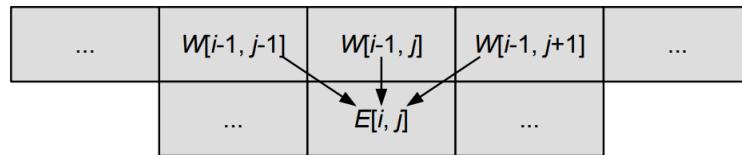
$$W[i, j] = \min(W[i - 1, j], W[i - 1, j + 1]).$$

- $1 < j < N$

$$W[i, j] = \min(W[i - 1, j - 1], W[i - 1, j], W[i - 1, j + 1]).$$

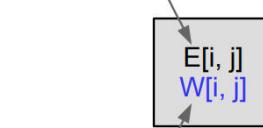
- $j = N$

$$W[i, j] = \min(W[i - 1, j - 1], W[i - 1, j]).$$



Ci ritroveremo con un vettore di questo tipo:

Energia del pixel (i, j)



Minimo peso tra tutte le cuciture che iniziano sulla prima riga e terminano nel pixel (i, j)

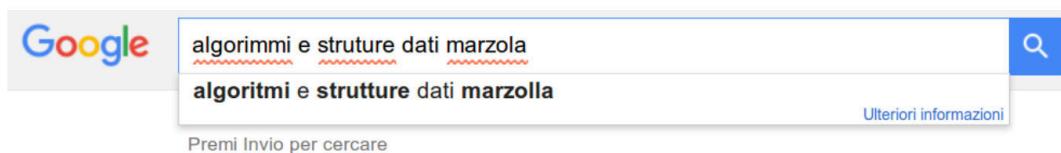
0.1 0.1	0.0 0.0	0.2 0.2	0.9 0.9	0.8 0.8
0.9 0.9	0.2 0.2	0.8 0.8	0.4 0.6	0.7 1.5
0.8 1.0	0.8 1.0	0.1 0.3	0.7 1.3	0.8 1.4
0.1 1.1	0.0 0.3	0.6 0.9	0.5 0.8	0.7 2.0

A questo punto per eliminare una cucitura dall'immagine scegliamo quella con il valore minimo tra tutte le cuciture che terminano nella riga M .

0.1 0.1	0.0 0.0	0.2 0.2	0.9 0.9	0.8 0.8
0.9 0.9	0.2 0.2	0.8 0.8	0.4 0.6	0.7 1.5
0.8 1.0	0.8 1.0	0.1 0.3	0.7 1.3	0.8 1.4
0.1 1.1	0.0 0.3	0.6 0.9	0.5 0.8	0.7 2.0

Esempio: distanza di Levenshtein

I **correttori ortografici** sono strumenti in grado di suggerire le parole corrette più simili rispetto a quelle che abbiamo digitato.



Alcuni di questi strumenti fanno utilizzo del concetto di “**edit distance**” al fine di comprendere la distanza tra due stringhe e suggerire le stringhe più simili a quella che sta venendo digitata. L’edit distance consiste nel numero di operazioni di editing necessarie per trasformare una stringa in un’altra, e le **operazioni di editing** disponibili sono le seguenti:

- Lasciare immutato un carattere, costo 0.
- Cancellare un carattere, costo 1.
- Inserire un carattere, costo 1.
- Sostituire un carattere con uno diverso, costo 1.

Si inizia dal primo carattere della stringa e ad ogni operazione ci si sposta sul carattere successivo. Possiamo notare tramite un esempio che è possibile avere più di una sequenza di operazioni possibili per trasformare una stringa in un’altra.

ALBERO	→	LBERO	cancellazione A
LBERO	→	BERO	cancellazione L
BERO	→	ERO	cancellazione B
ERO	→	RO	cancellazione E
RO	→	O	cancellazione R
O	→	-	cancellazione O
-	→	L-	inserimento L
L-	→	LI-	inserimento I
LI-	→	LIB-	inserimento B
LIB-	→	LIBR-	inserimento R
LIBR-	→	LIBRO-	inserimento O

<u>ALBERO</u>	→	<u>L</u> BERO	cancello A
<u>L</u> BERO	→	LBERO	lascio immutato
<u>L</u> BERO	→	LIBERO	inserisco I
LIBERO	→	LIBERO	lascio immutato
LIBERO	→	LIBRO	cancello E

Due sequenze di trasformazione della stringa "albero" in "libro".

Osserviamo infatti che abbiamo utilizzato due sequenze differenti per trasformare la stringa "albero" in "libro", le quali presentano rispettivamente costo 11 e 3.

La **distanza di Levenshtein** tra due stringhe $S[1 \dots n]$ e $T[1 \dots m]$ è il costo minimo tra tutte le sequenze di editing che trasformano la stringa S nella stringa T .

È possibile ricavare la distanza di Levenshtein tramite programmazione dinamica, costruendo un vettore L di due dimensioni in cui per ogni elementi $L[i, j]$ viene inserita la distanza di Levenshtein tra $S[1 \dots i]$ e $T[1 \dots j]$. Possiamo quindi riempire tale vettore nel seguente modo:

- $i = 0$ oppure $j = 0$

In questo caso una delle due stringhe è vuota, dunque la distanza di Levenshtein corrisponde alla lunghezza della stringa non vuota.

$$L[i, j] = \max(i, j).$$

- Altrimenti
 - $S[i] \neq T[j]$

La distanza di Levenshtein è il minimo tra il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j]$ + rimozione del carattere $S[i]$, il costo per trasformare $S[1 \dots i]$ in $T[1 \dots j - 1]$ + aggiunta del carattere $T[j]$ e il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j - 1]$ + modifica del carattere $S[i]$ in $T[j]$.

$$L[i, j] = \max(L[i - 1, j] + 1, L[i, j - 1] + 1, L[i - 1, j - 1] + 1).$$

- $S[i] = T[j]$

La distanza di Levenshtein è il minimo tra il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j]$ + rimozione del carattere $S[i]$, il costo per trasformare $S[1 \dots i]$ in $T[1 \dots j - 1]$ + aggiunta del carattere $T[j]$ e il costo per trasformare $S[1 \dots i - 1]$ in $T[1 \dots j - 1]$ + lasciare l'ultimo carattere invariato.

$$L[i, j] = \min(L[i - 1, j] + 1, L[i, j - 1] + 1, L[i - 1, j - 1]).$$

	“”	L	I	B	R	O
“”	0	1	2	3	4	5
A	1	1	2	3	4	5
L	2	1	2	3	4	5
B	3	2	2	2	3	4
E	4	3	3	3	3	4
R	5	4	4	4	3	4
O	6	5	5	5	4	3

Minimo numero di operazioni di editing necessarie per trasformare ALBE in LIB

Distanza di Levenshtein tra ALBERO e LIBRO

Dopo aver costruito tale vettore troviamo il valore della distanza di Levenshtein tra $S[1 \dots n]$ e $T[1 \dots m]$ in $L[n, m]$.

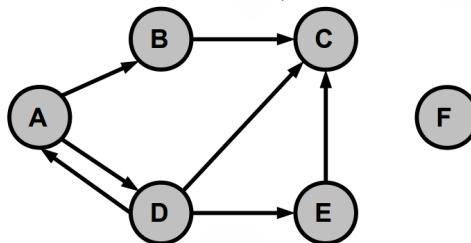
▼ 11.0 - Grafi

▼ 11.1 - Introduzione ai grafi

Grafi orientati e non orientati

Un **grafo orientato** è una coppia (V, E) tale che:

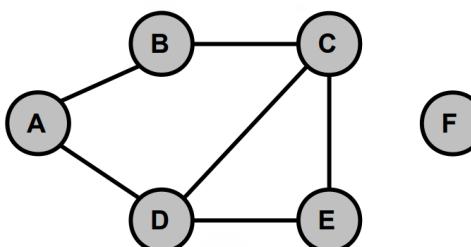
- V è un insieme di **vertici**.
- E è un insieme di **archi**, ovvero coppie non ordinate di vertici.
- Esistono archi (X, X) e vengono chiamati **cappi**.



Grafo orientato.

Un **grafo non orientato** è una coppia (V, E) tale che:

- V è un insieme di **vertici**.
- E è un insieme di **archi**, ovvero coppie ordinate di vertici.
- Non esistono archi (X, X) .



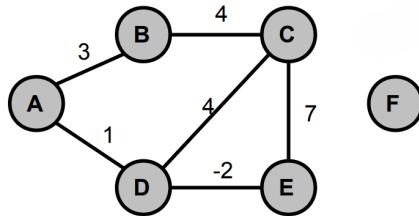
Grafo non orientato.

Grafi pesati

Un **grafo pesato** è un grafo in cui ogni arco ha un valore numerico che corrisponde al suo peso.

In alcune implementazioni, come ad esempio nella matrice di adiacenza, viene utilizzato il peso ∞ per due vertici tra i quali non esiste un arco.

Solitamente esiste una **funzione** $w : E \rightarrow \mathbb{R}$ che dato un arco restituisce il suo peso.



Grafo pesato.

Incidenza e adiacenza

In un grafo orientato l'arco (v, w) è **incidente** da v in w .

In un grafo (orientato/non orientato) w è **adiacente** a v se e solo se $(v, w) \in E$.

Notiamo che l'adiacenza è una relazione simmetrica se il grafo non è orientato, mentre se il grafo è orientato è importante l'ordine dei vertici nella coppia (v, w) (es. se $(v, w) \in E$ allora w è adiacente a v , ma non è detto che anche v sia adiacente a w).

Grado di un vertice

Grafo non orientato

In un grafo non orientato, il **grado** (δ) di un vertice corrisponde al numero di archi che partono da esso.

Grafo orientato

In un grafo orientato, il **grado entrante (uscente)** di un vertice corrisponde al numero di archi incidenti in (da) da esso.

In un grafo orientato, il **grado** (δ) di un vertice corrisponde alla somma del suo grado entrante e del suo grado uscente.

Grafi completi

Un **grafo non orientato completo** è un grafo non orientato che ha un arco tra ogni coppia di vertici.

Il numero di archi in un grafo non orientato completo con n vertici è equivalente a $\binom{n}{2} = \frac{n(n-1)}{2}$.

Cammini

Un **cammino** in un grafo è una sequenza di vertici $\langle w_0, w_1, \dots, w_k \rangle$ tali che w_{i+1} è adiacente a w_i per $0 \leq i \leq k - 1$.

La **lunghezza di un cammino** corrisponde al numero di archi traversati, equivalente al numero di vertici meno 1.

Un cammino si dice **semplice** se tutti i suoi vertici sono distinti, ovvero compaiono una sola volta nella sequenza.

Si dice che w è **raggiungibile** da v se esiste almeno un cammino che partendo da v arriva a w .

In un grafo non orientato, si dice che tale grafo è **connesso** se esiste un cammino da ogni vertice verso ogni altro vertice.

In un grafo orientato, si dice che tale grafo è **fortemente connesso** se esiste un cammino da ogni vertice verso ogni altro vertice.

Versione non orientata

Se un grafo è orientato, il grafo ottenuto ignorando la direzione degli archi e i cappi è detto grafo non orientato sottostante oppure **versione non orientata** del grafo.

Se un grafo orientato non è fortemente connesso, ma la sua versione non orientata è connessa, allora diciamo che tale grafo è **debolmente connesso**.

Cicli

Un **ciclo** è un cammino $\langle w_0, w_1, \dots, w_n \rangle$ tale che $w_0 = w_n$ e con una lunghezza di almeno 1 nei grafi orientati e di almeno 3 nei grafi non orientati.

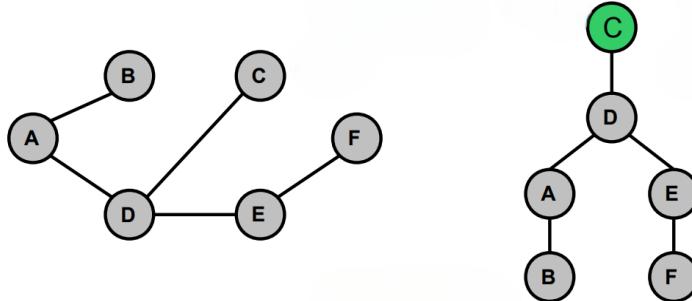
Un ciclo è **semplice** se tutti i vertici che compongono il suo cammino sono distinti.

Un grafo non orientato è **aciclico** se è privo di cicli semplici.

Un grafo orientato è **aciclico** se è privo di cicli.

Alberi liberi

Un **albero libero** è un grafo non orientato connesso e aciclico. Se un vertice di tale albero è detto radice, otteniamo un **albero radicato**.



Albero libero e albero radicato.

Operazioni basilari

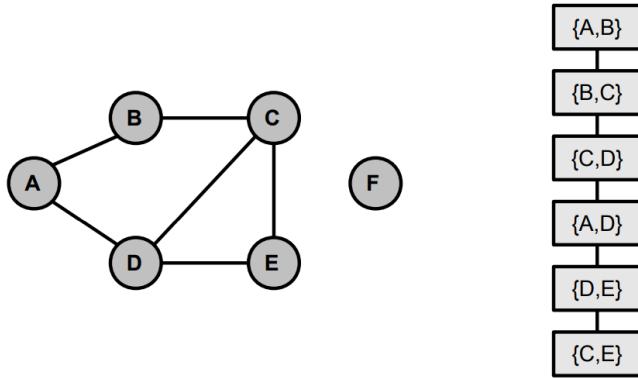
Le **operazioni basilari** che ogni grafo deve supportare sono le seguenti:

- `int numVertici()`
- `int numArchi()`
- `int grado(vertice v)`
- `(arco, arco, ..., arco) archiIncidenti(vertice v)`
- `(vertice, vertice) estremi(arco e)`
- `vertice opposto(vertice x, arco e)`
- `bool sonoAdiacenti(vertice x, vertice y)`
- `void aggiungiVertice(vertice v)`
- `void aggiungiArco(vertice x, vertice y)`
- `void rimuoviVertice(vertice v)`
- `void rimuoviArco(arco e)`

Implementazioni dei grafi

Liste di archi

Un primo modo per implementare dei grafi è quello di utilizzare una semplice **lista** per elencare i suoi archi.



Lista di archi.

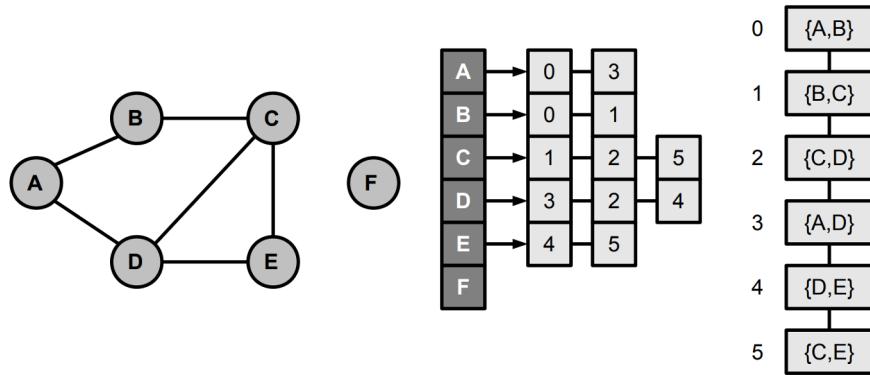
Il **costo computazionale** in termini di spazio è $\Theta(|E|)$.

I **costi computazionali** in termini di tempo delle operazioni basilari risultano però poco efficienti in quanto spesso richiedono la scansione dell'intera lista di archi:

- grado(vertice v): $O(m)$.
- archiIncidenti(vertice v): $O(m)$.
- sonoAdiacenti(vertice x, vertice y): $O(m)$.
- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(1)$.

Liste di incidenza

Per migliorare i costi computazionali è utile mantenere insieme alla lista di archi anche una lista detta **lista di incidenza** nella quale per ogni vertice si ha una lista degli indici dei nodi a cui esso appartiene.



Lista di incidenza.

I **costi computazionali** diventano i seguenti:

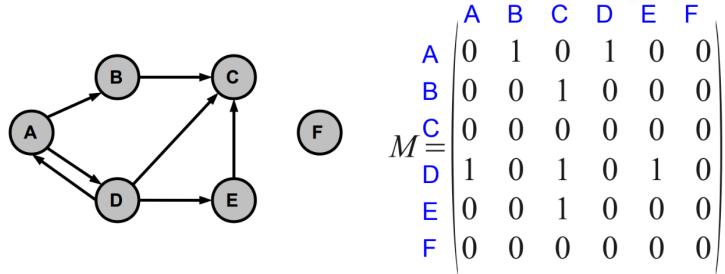
- grado(vertice v): $O(\delta(v))$.
- archiIncidenti(vertice v): $O(\delta(v))$.
- sonoAdiacenti(vertice x, vertice y): $O(\min(\delta(x), \delta(y)))$.

- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(\delta(x) + \delta(y))$.

Matrici di adiacenza

Un'altra possibile implementazione utilizza una matrice, detta **matrice di adiacenza**, i quali valori vengono decisi in questo modo:

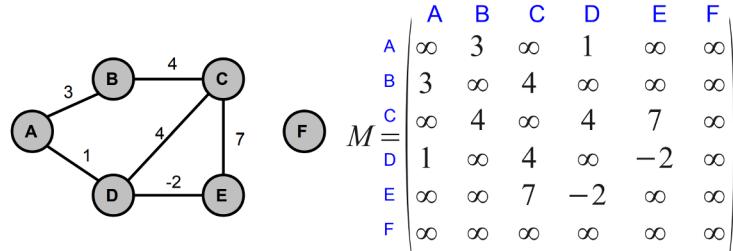
$$M(u, v) = \begin{cases} 1 & (u, v) \in E \\ 0 & (u, v) \notin E \end{cases}$$



Matrice di adiacenza per grafo non pesato.

Oppure per un grafo pesato:

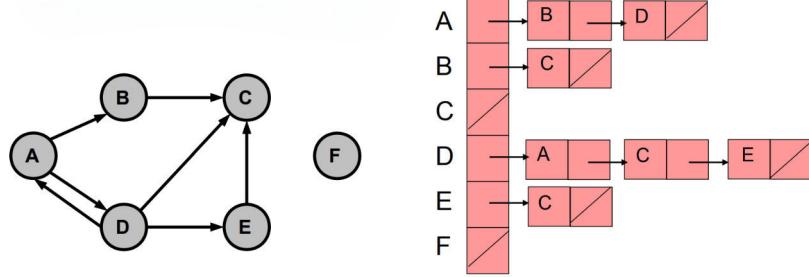
$$M(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ \infty & (u, v) \notin E \end{cases}$$



Matrice di adiacenza per grafo pesato.

Liste di adiacenza

Un'ulteriore implementazione comprende l'utilizzo di una lista detta **lista di incidenza** nella quale per ogni vertice si ha una lista dei nodi adiacenti ad esso.



Lista di adiacenza.

Il **costo computazionale** in termini di spazio è $\Theta(|V| + |E|)$.

I **costi computazionali** in termini di tempo sono i seguenti:

- grado(vertice v): $O(\delta(v))$.
- archiIncidenti(vertice v): $O(\delta(v))$.
- sonoAdiacenti(vertice x, vertice y): $O(\min(\delta(x), \delta(y)))$.
- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(\delta(x) + \delta(y))$.

▼ 11.2 - Algoritmi di visita di grafi

Il problema della **visita di grafi** consiste nel, dato un vertice del grafo, visitare una sola volta tutti i vertici raggiungibili dal vertice dato.

L'idea è quella di costruire un algoritmo che, dato un vertice s , costruisce un **albero** T radicato in s che al termine della visita conterrà una sola volta tutti i vertici raggiungibili partendo da s . Durante la visita, al fine di evitare di visitare dei vertici più di una volta, ogni vertice può trovarsi in uno dei seguenti 3 stati:

- Inesplorato**: il vertice non è ancora stato incontrato.
- Aperto**: l'algoritmo ha incontrato il vertice la prima volta.
- Chiuso**: il vertice è stato visitato completamente, ovvero tutti i suoi archi incidenti sono stati esplorati.

Durante l'esecuzione è utile mantenere un sottoinsieme F di T nel quale inserire tutti i nodi che stanno venendo visitati. In questo modo in base alla presenza o meno di un vertice nell'insieme F sapremo in quale dei 3 stati si trova:

- v non è in T : inesplorato.
- v è in F : aperto.
- v è in $T - F$: chiuso.

L'algoritmo generico per la visita di un grafo è il seguente:

```

albero visita(G, s) {
    for (each v in V)
        v.mark = false

```

```

    T = s

```

```

F = {s}
s.mark = true

while (F != Ø) {
    u = F.extract()
    // visita il vertice u
    for (each v adiacente a u) {
        if (!v.mark) {
            v.mark = true
            T = T ∪ v
            F.insert(v)
            v.parent = u
        }
    }
}
return T
}

```

Il **costo computazionale** di questo algoritmo è (n : numero di vertici, m : numero di archi):

- **Liste di adiacenza:** $O(n + m)$.
- **Matrice di adiacenza:** $O(n^2)$.

Analizzeremo due tipologie di visite di un grafo, la visita in ampiezza e quella in profondità.

Visita in ampiezza

Tramite la **visita in ampiezza (BFS - Breadth First Search)** di un grafo vogliamo creare un albero in cui inserire tutti i vertici del grafo visitati a distanza crescente dal nodo sorgente dato in input, ovvero prima di visitare i nodi a distanza $k + 1$ dal vertice dato in input devono essere prima visitati tutti i nodi a distanza k .

È possibile utilizzare tale tipologia di visita anche per creare algoritmi che calcolino la minima distanza tra la sorgente e un altro nodo del grafo raggiungibile da essa.

Lo **pseudocodice** di un'algoritmo di visita in ampiezza è il seguente:

```

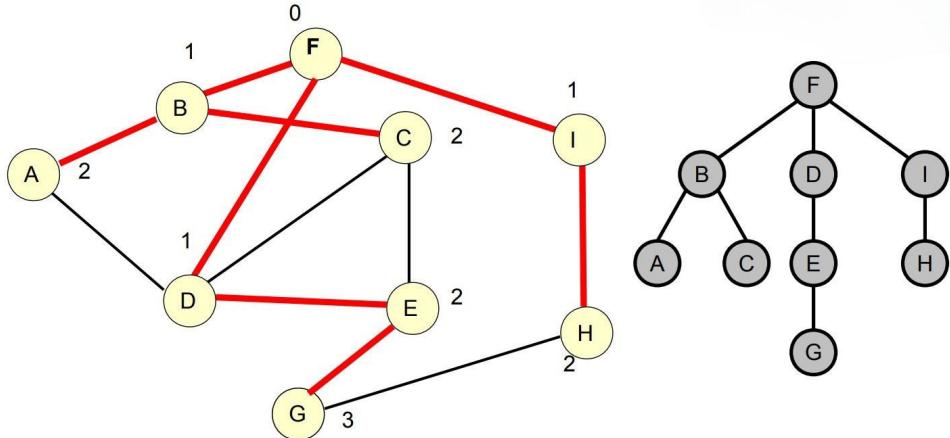
albero BFS(Grafo G, vertice s) {
    for (each v in V)
        v.mark = false

    T = s
    F = new Queue()
    F.enqueue(s)
    s.mark = true

    s.dist = 0
    while (F != Ø) {
        u = F.dequeue()
        // visita il vertice u
        for (each v adiacente a u) {
            if (!v.mark) {
                v.mark = true
                T = T ∪ v
                F.enqueue(v)
                v.parent = u
                v.dist = u.dist + 1
            }
        }
    }

    return T
}

```



Albero creato da una visita in ampiezza.

Visita in profondità

La **visita in profondità (DFS - Depth First Search)**, a differenza di quella in ampiezza, visita tutti i nodi presenti in un grafo, dunque non solo quelli raggiungibili dal vertice dato in input. Per questo motivo la visita in profondità restituisce una foresta, ovvero un insieme di alberi nei quali vengono visitati i nodi andando il più lontano da livello attuale prima di passare al nodo sul successivo in quel livello.

Lo **pseudocodice** di una visita in profondità in versione ricorsiva è il seguente (white: inesplorato, grey: aperto, black: chiuso, dt : discovery time, ft : finish time):

```

int time = 0

void DFS(Grafo G) {
    for (each u in V) {
        u.mark = white
        u.parent = null
    }

    for (each u in V) {
        if (u.mark == white)
            DFS-visit(u)
    }
}

void DFS-visit(vertice u) {
    u.mark = gray
    time++
    u.dt = time

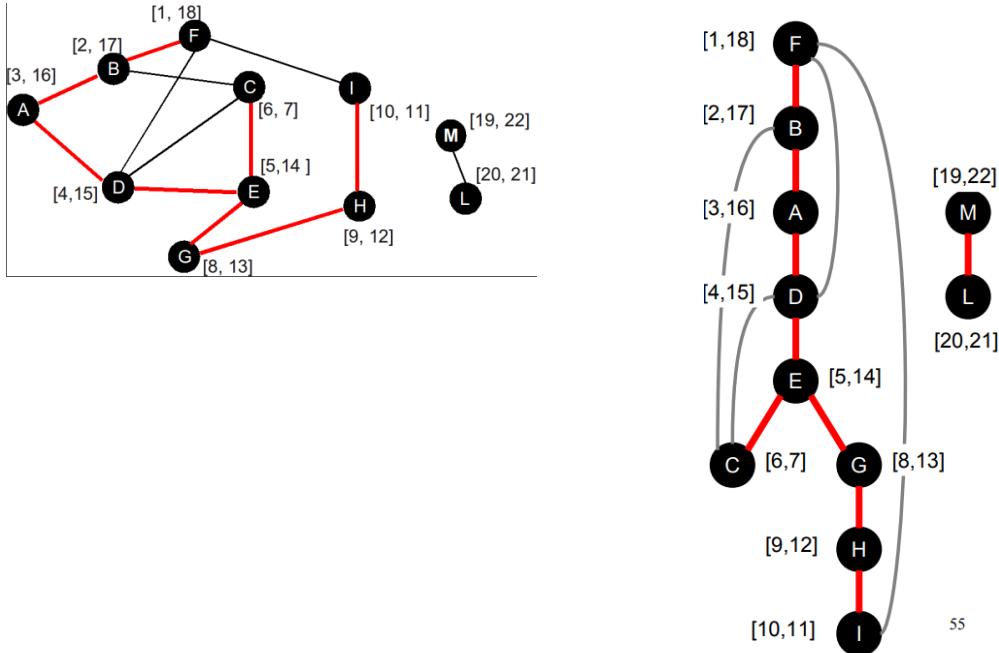
    for (each v adiacente a u) {
        if (v.mark == white) {
            v.parent = u
            DFS-visit(v)
        }
    }

    time++
    u.ft = time
    u.mark = black
}

```

Poniamo ora $n.dt$ (discovery time) come l'istante in cui un vertice è stato aperto, e $n.ft$ (finish time) come l'istante in cui lo stesso vettore è stato chiuso, possiamo concludere che:

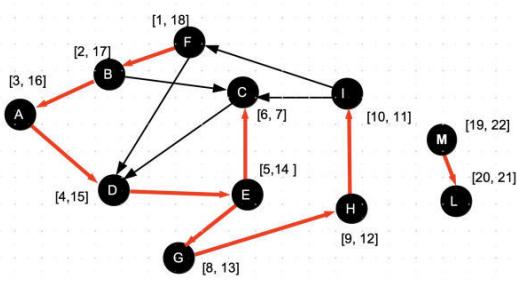
v è discendente di v nella foresta $\iff u.dt < v.dt < v.ft < u.ft$

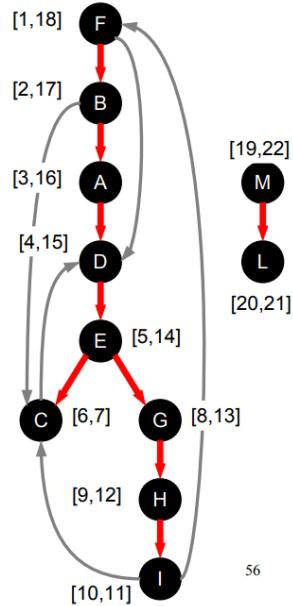


Nelle figure soprastanti $[x, y]$ corrispondono a $[n.dt, n.ft]$.

È inoltre possibile sfruttare il discovery time e il finish time per verificare la direzione di un arco (u, v) :

- Se $u.dt < v.dt$ e $v.ft < u.ft$ allora l'arco è in **avanti**, ovvero va da un nodo ascendente a un nodo suo discendente.
- Se $v.dt < u.dt$ e $u.ft < v.ft$ allora l'arco è all'**indietro**, ovvero va da un nodo discendente a un nodo suo ascendente.
- Se $v.ft < u.dt$ allora l'arco è di **attraversamento a sinistra**, ovvero i due nodi hanno un padre in comune, ma nessuno è ascendente/descendente dell'altro.





56

Applicazioni delle visite di grafi

Alcune **possibili applicazioni** degli algoritmi di visita di un grafo è il seguente:

- Identificare il **cammino più breve** tra due vertici di un grafo.
- Verificare che un grafo sia **aciclico**.

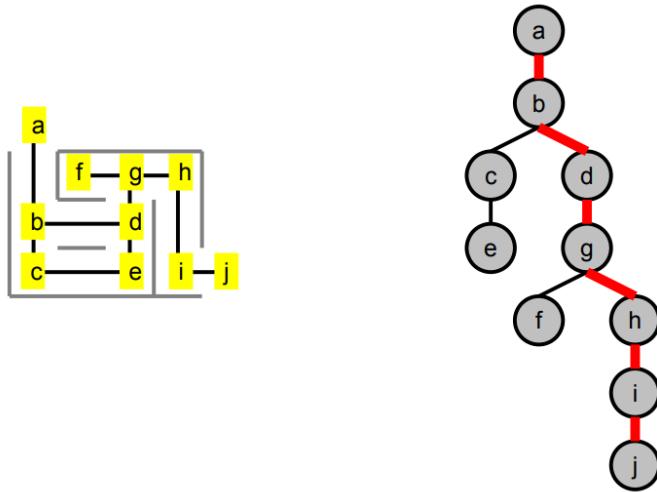
Realizzabile eseguendo una visita in profondità e verificando che non ci siano archi all'indietro.

- **Ordinamento topologico**.
- Individuare le componenti **connesse** di un grafo non orientato.
- Individuare le componenti **fortemente connesse** di un grafo orientato.

Cammino più breve

È possibile utilizzare l'albero creato dopo la visita in ampiezza di un grafo al fine di ottenere il **percorso più breve** tra due vertici:

```
void printPath(G, s, v) {
    if (v == s) print(s)
    else if (v.parent == null)
        print("no path from s to v")
    else {
        print-path(G, s, v.parent)
        print(v)
    }
}
```

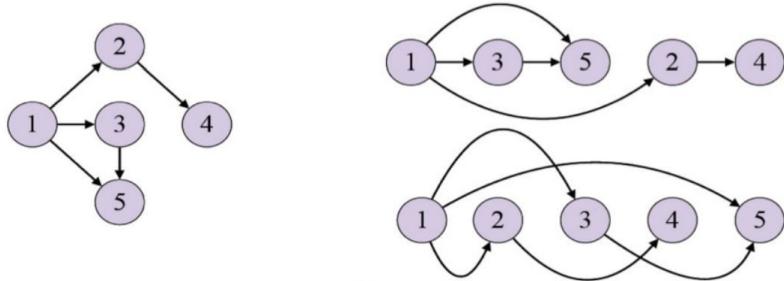


Utilizzo della visita in ampiezza per calcolare il percorso più veloce per uscire da un labirinto.

Ordinamento topologico

Dato un grafo ordinato privo di cicli, un **ordinamento topologico** consiste in un ordinamento lineare dei suoi nodi tale che, se v è raggiungibile da u , allora u compare prima di v nell'ordinamento.

Possono esistere più di un ordinamento topologico per lo stesso grafo.



È possibile realizzare ordinamenti topologici facendo uso di una visita in profondità aggiungendo ogni nodo alla testa di una lista nel momento del suo finish time. In questo modo si otterrà una lista in cui i nodi sono ordinati in base decrescente di finish time, quindi tale lista rispetta la definizione di ordinamento topologico in quanto come abbiamo visto se un nodo v è raggiungibile da un nodo u , allora il finish time di v sarà minore di u , quindi nella lista v si troverà dopo u .

Componenti connesse

In un grafo non orientato, due vertici appartengono alla stessa **componente连通子图** se da uno è possibile raggiungere l'altro. Si può utilizzare la visita in profondità per identificare tutte le componenti connesse tra loro.

Nel seguente pseudocodice eticheremo tutti i nodi di un grafo con un intero, e al termine dell'esecuzione se due nodi sono etichettati con lo stesso intero allora sono connessi:

```

void CC(G) {
    for (each u in G) {
        u.cc = -1
        u.parent = null
    }

    k = 0
    for (each u in G) {

```

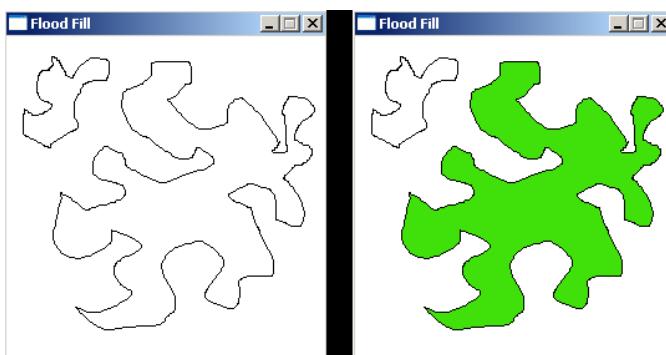
```

        if (u.cc < 0) {
            CC-visit(u, k)
            k++
        }
    }

void CC-visit(u, k) {
    u.cc = k
    for (each v adiacente a u) {
        if (v.cc < 0) {
            v.parent = u
            CC-visit(v, k)
        }
    }
}

```

Una possibile applicazione della ricerca delle componenti fortemente connesse è il riempimento a tinta unita di alcuni software come paint.



Floodfill, riempimento a tinta unita.

Componenti fortemente connesse

In un grafo orientato, due nodi appartengono alla stessa **componente fortemente connessi** se c'è un cammino che connette un vertice con l'altro e viceversa.

Notiamo che è possibile determinare tutti i nodi fortemente connessi a un certo vertice x calcolando prima l'insieme $D(x)$ dei nodi raggiungibili da x e l'insieme $A(x)$ dei nodi da cui si può raggiungere x per poi effettuare l'intersezione $D(x) \cap A(x)$ tra i due insiemi.

Per costruire tali insiemi è possibile, per quanto riguarda $D(x)$, richiamare una funzione di visita in ampiezza usando x come sorgente, mentre per quanto riguarda $A(x)$ occorre invertire la direzione degli archi nel grafo e richiamare ancora una funzione di visita in ampiezza usando x come sorgente.

Lo **pseudocodice** è il seguente:

```

lista SCC(Grafo G, nodo x) {
    L = lista vuota di nodi
    (1) Eseguire BFS(G, x) marcando i nodi visitati
    (2) Calcolare il grafo trasposto GT
        (invertire la direzione degli archi di G)
    (3) Eseguire BFS(GT, x), mettendo in L i nodi
        visitati che sono stati marcati durante (1)
    return L
}

```

Il **costo computazionale**, visto che tutte e 3 i passaggi hanno costo $O(n + m)$, è $O(n + m)$.

Per calcolare tutte le componenti fortemente connesse di un grafo occorre richiamare la funzione appena discussa per ogni nodo del grafo. Il costo computazionale in questo caso è $n \cdot O(n^2 + mn) = O(n^2 + mn)$. Esiste un algoritmo più sofisticato che ha costo $O(n + m)$ ma che non tratteremo.

▼ 11.3 - Minimum spanning tree

Definizione

Albero di copertura

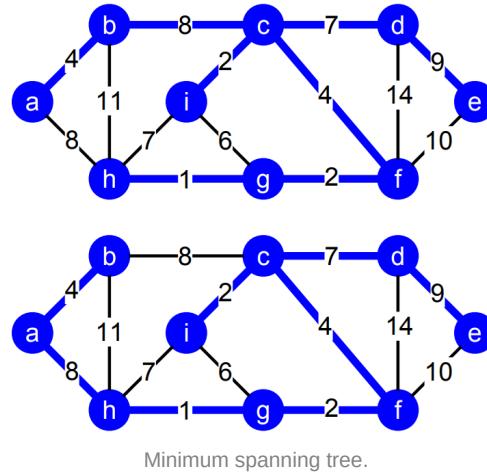
Un **albero di copertura** di un grafo $G = (V, E)$ è un sottografo $T = (V, E_T)$ tale che:

- T è un albero.
- T ha gli stessi nodi di G .
- $E_T \subseteq E$.

Minimum spanning tree

Un **minimum spanning tree (MST)**, o albero di copertura di peso minimo, è un albero di copertura il cui peso totale, ovvero la somma dei pesi dei singoli archi, sia minimo tra tutti i possibili alberi di copertura.

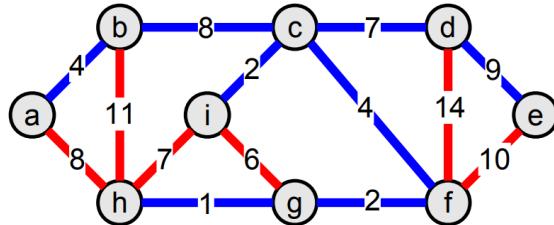
Nota: il minimum spanning tree non è necessariamente unico.



Minimum spanning tree.

Vedremo 2 algoritmi di tipo greedy che consentono di calcolare il minimum spanning tree.

Per convenzione ci riferiamo agli archi di colore **blu** per gli archi che fanno parte del MST, e agli archi di colore **rosso** per quelli che non ne fanno parte.



Minimum spanning tree.

L'idea nella costruzione di un MST tramite metodo greedy è quella di creare un albero T aggiungendo di volta in volta archi sicuri.

Un arco (u, v) è detto **sicuro** per T se $T \cup (u, v)$ è ancora un sottoinsieme di un MST.

Lo **pseudocodice** di un tale algoritmo è il seguente:

```
Tree genericMST(Grafo G = (V, E, w)) {
    Tree T = Albero vuoto
    while (T non forma un albero di copertura) {
        trova un arco sicuro {u, v}
        T = T U {u, v}
    }
    return T
}
```

Al fine di poter individuare archi sicuri occorre introdurre alcune definizioni.

Un **taglio** $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V in due sottoinsiemi disgiunti.

Un arco (u, v) **attraversa il taglio** se $u \in S$ e $v \in V - S$.

Un taglio **rispetta** un insieme di archi chiamato T se nessun arco appartenente a T attraversa il taglio.

Un arco che attraversa il taglio è **leggero** se il suo peso è il minimo tra i pesi di tutti gli archi che attraversano il taglio.

Un metodo greedy per la costruzione di MST consiste nell'utilizzo in successione di una qualunque, purchè si possa usare, di queste due regole:

- **Regola del taglio**

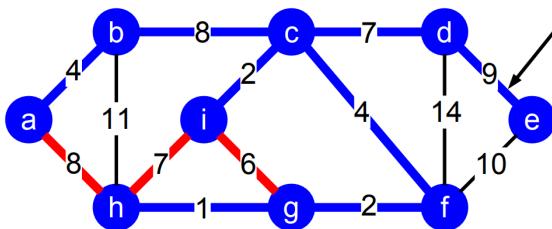
Viene scelto un taglio che rispetta l'insieme degli archi già colorati di blu. Tra tutti gli archi che attraversano il taglio ne viene scelto uno leggero e viene colorato di blu.

- **Regola del ciclo**

Viene scelto un ciclo che non contenga archi di colore rosso e tra tutti gli archi non colorati di tale ciclo ne viene scelto uno di peso massimo e viene colorato di rosso.

Algoritmo di Kruskal

L'idea dell'**algoritmo di Kruskal** consiste nell'avere inizialmente n insiemi, uno per ogni vertice del grafo, e ognuno dei quali formato da un singolo nodo. Successivamente vengono uniti tra loro questi insiemi colorando di blu gli archi che faranno parte del MST, e di rosso quelli che non ne faranno parte. Per deciderlo analizziamo gli archi del grafo in ordine non decrescente, e se l'arco che si sta analizzando connette due insiemi distinti, allora lo si colora di blu, altrimenti di rosso.



Lo **pseudocodice** di un algoritmo di Kruskal è il seguente:

```

Tree Kruskal-MST(Grafo G = (V, E, w)) {
    UnionFind UF
    Tree T = albero vuoto
    for (int i = 1; i < G.numNodi()) UF.makeSet(i)

    // ordina gli archi di E per peso w crescente
    sort(E, w)

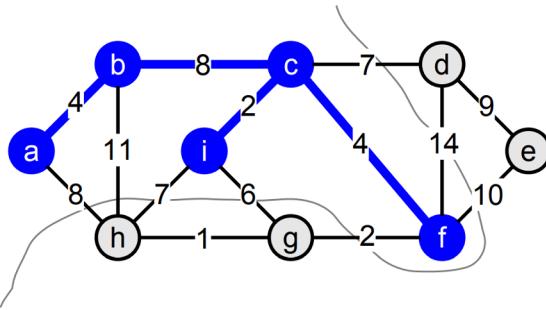
    for (each {u, v} in E) {
        Tu = UF.find(u)
        Tv = UF.find(v)
        if (Tu != Tv) { // non fanno parte dell'insieme
            T = T u {u, v} // aggiungi arco
            UF.union(Tu, Tv) // unisci componenti
        }
    }
    return T
}

```

Il **costo computazionale** nel caso in cui venga utilizzata la struttura dati QuickUnion è $O(m \log n)$.

Algoritmo di Prim

L'**algoritmo di Prim** si basa sull'idea di iniziare a costruire l'albero di copertura minima da un certo nodo arbitrario chiamato radice, e ad ogni passo creare un taglio che divida il grafo in 2 insiemi di nodi, uno è quello dei vertici già facenti parte dell'MST e l'altro quello dei nodi non ancora raggiunti dall'MST. Tra gli archi che attraversano tale taglio ne viene scelto uno leggero e viene colorato di blu.



Lo **pseudocodice** di un algoritmo di Prim è il seguente:

```

integer[] primMST(Grafo G = (V, E, w), nodo s) {
    /* per ogni nodo viene indicato il peso
       minimo per essere collegato all'albero
       se non esiste un arco che lo collega -> infinito */
    double d[1 ... n]
    /* per ogni nodo viene indicato l'indice del padre nel MST */
    int p[1 ... n]
    /* per ogni nodo viene indicato con true se già
       presente nel MST, false altrimenti */
    boolean b[1 ... n]

    for (int v = 1; v < n; v++) {
        d[v] = ∞
        p[v] = -1
        b[v] = false
    }

    d[s] = 0
    /* coda con priorità in cui vengono inseriti
       i nodi non ancora nel MST ordinati in base
       al valore presente nell'array d */
    CodaPriorita<int, double> Q
    Q.insert(s, d[s])

    while (!Q.isEmpty()) {
        /* viene preso il nodo con il valore minimo
           nella coda e messo nel MST */
        u = Q.findMin()
        Q.deleteMin()
        b[u] = true

        /* vengono inseriti nella coda tutti i nodi
           adiacenti all'ultimo nodo aggiunto nell'albero
           con il relativo valore */
        for (each v adiacente a u && !b[v]) {
            if (d[v] == ∞) {
                Q.insert(v, w(u,v))
                d[v] = w(u,v)
                p[v] = u
            } else if (w(u,v) < d[v]) {
                Q.decreaseKey(v, d[v] - w(u, v))
                d[v] = w(u,v)
                p[v] = u
            }
        }
    }

    return p
}

```

Il **costo computazionale** nel caso in cui la coda con priorità venga implementata tramite min-heap è $O(m \log n)$.

▼ 11.4 - Cammini minimi

Definizione

Costo di un cammino

Dato un grafo G , il **costo di un cammino** $\pi = (v_0, \dots, v_k)$ che collega il vertice v_0 con il vertice v_k è definito come:

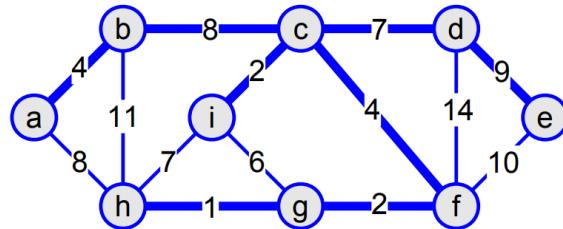
$$w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Cammino minimo

Dato un grafo G e due nodi v_0 e v_k il **cammino minimo** consiste nel cammino con costo minimo tra tutti i cammini che vanno da v_0 e v_k .

È importante notare che il problema della costruzione del MST e della ricerca del cammino minimo sono due problemi completamente differenti, in quanto non è detto che il cammino minimo tra due nodi si trovi all'interno dell'MST.

Ad esempio il cammino di costo minimo tra i nodi h e i nel seguente grafo non fa parte del MST.



Diverse formulazioni del problema

Esistono **diverse tipologie di problemi** che utilizzano il concetto di ricerca del cammino minimo:

1. Cammino minimo fra una coppia di nodi

Determinare, se esiste, un cammino minimo tra due nodi appartenenti a un grafo.

2. Single-source shortest path

Determinare i cammini di costo minimo da un nodo sorgente s fino a tutti gli altri nodi raggiungibili da s .

3. All-pairs shortest path

Determinare i cammini di costo minimo tra ogni coppia di nodi in un grafo.

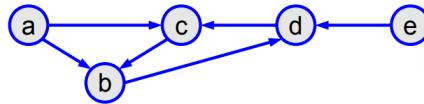
È importante tenere a mente che ancora non è stato trovato un algoritmo che consenta di risolvere il problema 1 senza risolvere anche il problema 2.

Single-source shortest path

Quando non esiste un cammino minimo?

Non esiste un cammino minimo tra due nodi se:

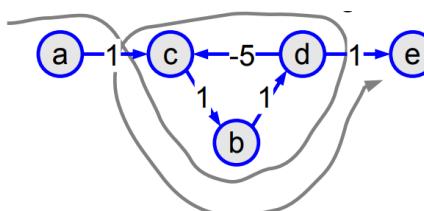
- La destinazione **non è raggiungibile** dalla sorgente.



Nodo *e* non raggiungibile partendo da *a*.

- Esiste un ciclo tra i due nodi di **costo negativo**.

In questo caso è infatti sempre possibile trovare un cammino di costo minore facendo un ulteriore giro del ciclo.



Ciclo negativo per raggiungere *e* partendo da *a*.

Algoritmo di Bellman e Ford

L'**algoritmo di Bellman e Ford** consente di risolvere il problema single-source shortest path in un grafo.

L'idea è quella di assegnare ad ogni nodo un valore corrispondente al costo del cammino minimo trovato fino a quel momento per arrivarci, per comodità ci riferiamo a tale valore con $D[v]$, dove v è il nodo destinazione. L'algoritmo imposta quindi inizialmente il valore 0 alla sorgente e valore $+\infty$ a tutti gli altri nodi, in quanto ancora non è stato trovato un cammino minimo per nessun nodo. Ad ogni passo si controlla, per ogni arco (u, v) presente nel grafo, se il costo del cammino $D[u] + w(u, v) < D[v]$, e in quel caso $D[v]$ diventa $D[u] + w(u, v)$, in quanto è stato trovato un cammino di costo minore per arrivare a quel nodo. Dopo $n - 1$ passi, tante quante sono il numero dei possibili vertici raggiungibili da s , siamo sicuri di aver calcolato tutti i valori $D[v]$ minimi.

Lo **pseudocodice** dell'algoritmo di Bellman e Ford è il seguente:

```

double[1 ... n] bellmanFord(Grafo G = (V, E, w), int s) {
    int n = G.numNodi()
    int pred[1 ... n] // array che per ogni nodo tiene conto
    // del suo nodo precedente nel cammino minimo
    double D[1 ... n]

    for (int v = 1; v <= n; v++) {
        D[v] = +∞
        pred[v] = -1
    }

    D[s] = 0
    for (int i = 1; i <= n - 1; i++) {
        for (each (u, v) in E) {
            if (D[u] + w(u, v) < D[v]) {
                D[v] = D[u] + w(u, v)
                pred[v] = u
            }
        }
    }
}
  
```

```

// eventuale controllo per cicli negativi
for (each (u,v) in E) {
    if (D[u] + w(u,v) < D[v])
        error "Il grafo contiene cicli negativi"
}

return D
}

```

Il **costo computazionale** è $O(nm)$.

Algoritmo di Dijkstra

L'**algoritmo di Dijkstra** consente di risolvere il problema single-source shortest path in un grafo con archi non negativi.

Tale algoritmo si basa sul seguente **lemma**:

Sia G un grafo i quali archi sono ≥ 0 e sia T una parte dell'albero dei cammini di costo minimo che partono da una sorgente s , allora l'arco (u, v) che minimizza la quantità $d_{su} + w(u, v)$ appartiene al cammino minimo da s a v .

Lo **pseudocodice** dell'algoritmo è dunque il seguente:

```

double[1 ... n] dijkstra(Grafo G = (V, E, w), int s) {
    int n = G.numNodi()
    int pred[1 ... n] // array che per ogni nodo tiene conto
    // del suo nodo precedente nel cammino minimo
    double D[1 ... n]

    for (int v = 1; v <= n, v++) {
        D[v] = +∞
        pred[v] = -1
    }

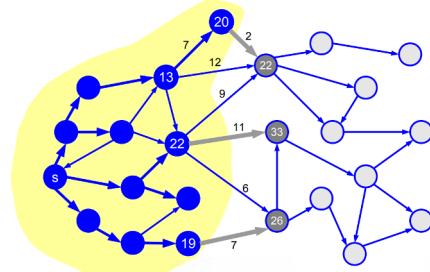
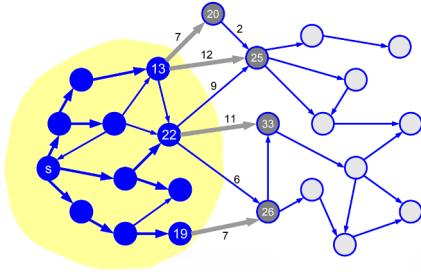
    D[s] = 0
    CodaPriorita<int, double> Q
    Q.insert(s, D[s])

    while (!Q.isEmpty()) {
        u = Q.findMin()
        Q.deleteMin()

        for (each v adiacente a u) {
            if (D[v] == +∞) {
                D[v] = D[u] + w(u, v)
                Q.insert(v, D[v])
                pred[v] = u
            } else if (D[u] + w(u,v) < D[v]) {
                Q.decreaseKey(v, D[v] - D[u] - w(u, v))
                D[v] = D[u] + w(u,v)
                pred[v] = u
            }
        }
    }

    return D
}

```



Il costo computazionale è $O(m \log n)$

All-pairs shortest path

Algoritmo di Floyd e Warshall

L'**algoritmo di Floyd e Warshall** consente di risolvere il problema all-pairs shortest path in un grafo con archi anche negativi utilizzando la programmazione dinamica.

Tale algoritmo utilizza il concetto di D_{xy}^k , ovvero la distanza minima tra x e y , con l'ipotesi che gli eventuali nodi intermedi possano appartenere solamente all'interno di nodi $\{1, \dots, k\}$.

L'idea è quella di utilizzare un ciclo che modifichi la variabile k partendo da 1 e arrivando a n , ovvero tutti gli archi presenti nel grafo. Ad ogni passo vengono aggiornate le D_{xy}^k di ogni coppia di nodi del grafo al fine di arrivare, per ogni coppia di nodi x e y il valore di D_{xy}^n , ovvero il cammino minimo tra di essi.

All'inizio dell'algoritmo c'è un'**inizializzazione** del cammino minimo tra ogni coppia di nodi senza l'utilizzo di nodi intermedi:

$$D_{xy}^0 = \begin{cases} 0 & \text{if } x = y \\ w(x, y) & \text{if } (x, y) \in E \\ \infty & \text{if } (x, y) \notin E \end{cases}$$

Successivamente viene eseguito il ciclo di cui prima, e per calcolare il valore D_{xy}^k consideriamo i due casi seguenti:

- Il nodo k **non fa parte** del nuovo cammino minimo.

In questo caso $D_{xy}^k = D_{xy}^{k-1}$.

- Il nodo k **fa parte** del nuovo cammino minimo.

In questo caso $D_{xy}^k = D_{xk}^{k-1} + D_{ky}^{k-1}$.

Per scegliere se inserire o meno k nel nuovo cammino minimo occorre dunque calcolare il **minimo tra i due cammini**:

$$D_{xy}^k = \min(D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1})$$

Lo **pseudocodice** è il seguente:

```
double[1 ... n, 1 ... n] floydWarshall(G = (V, E, w)) {
    int n = G.numNodi()
    double D[1 ... n, 1 ... n, 0 ... n] // array tridimensionale in cui
    // gli indici sono [x, y, k]

    // inizializzazione
    for (int x = 1; x <= n; x++) {
```

```

        for (int y = 1; y <= n; y++) {
            if (x == y) D[x, y, 0] = 0
            else if ((x, y) ∈ E) D[x, y, 0] = w(x, y)
            else D[x, y, 0] = +∞
        }

        // ciclo generale
        for (int k = 1; k <= n; k++) {
            for (int x = 1; x <= n; x++) {
                for (int y = 1; y <= n; y++) {
                    D[x, y, k] = D[x, y, k - 1]
                    if (D[x, k, k - 1] + D[k, y, k - 1] < D[x, y, k])
                        D[x, y, k] = D[x, k, k - 1] + D[k, y, k - 1]
                }
            }
        }

        // eventuale controllo per cicli negativi
        for (int x = 1; x <= n; x++) {
            if (D[x, x, n] < 0)
                error "Il grafo contiene cicli negativi"
        }

        return D[1 ... n, 1 ... n, n]
    }
}

```

Il **costo computazionale** in termini di tempo e di spazio è $O(n^3)$.

È possibile ottimizzare il costo in termini di spazio di questo algoritmo notando che una volta caricato in memoria il valore $D[x, y, k]$ per ogni coppia di nodi x e y il valore $D[x, y, k - 1]$ non servirà più, dunque possiamo utilizzare una matrice bidimensionale $D[x, y]$ al posto di quella tridimensionale e calcolare $D[x, y]$ al passo k -esimo nel seguente modo:

$$D[x, y] = \max(D[x, y], D[x, k] + D[k, y])$$

Lo pseudocodice è il seguente:

```

double[1 ... n, 1 ... n] floydWarshall(G = (V, E, w)) {
    int n = G.numNodi()
    double D[1 ... n, 1 ... n] // array bidimensionale in cui
    // gli indici sono [x, y]

    // inizializzazione
    for (int x = 1; x <= n; x++) {
        for (int y = 1; y <= n; y++) {
            if (x == y) D[x, y] = 0
            else if ((x, y) ∈ E) D[x, y] = w(x, y)
            else D[x, y] = +∞
        }
    }

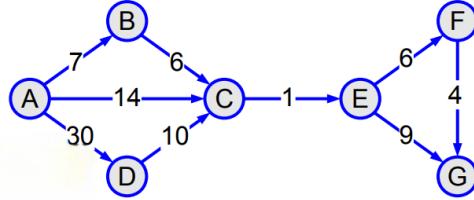
    // ciclo generale
    for (int k = 1; k <= n; k++) {
        for (int x = 1; x <= n; x++) {
            for (int y = 1; y <= n; y++) {
                if (D[x, k] + D[k, y] < D[x, y])
                    D[x, y] = D[x, k] + D[k, y]
            }
        }
    }

    // eventuale controllo per cicli negativi
    for (int x = 1; x <= n; x++) {
        if (D[x, x] < 0)
            error "Il grafo contiene cicli negativi"
    }

    return D[1 ... n, 1 ... n, n]
}

```

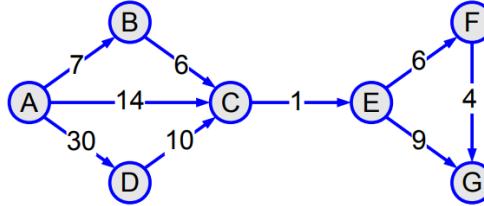
Il **costo computazionale** in termini di tempo è $O(n^2)$ e in termini di spazio è $O(n^3)$.



D =	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

All-pairs shortest path tramite algoritmo di Floyd e Warshall.

Per individuare i nodi che compongono i cammini di costo minimo generati dall'algoritmo di Floyd e Warshall è possibile utilizzare una matrice $next[x, y]$ in cui, per ogni coppia di nodi x e y , viene memorizzato il nodo dopo x nel cammino minimo, in modo poi da richiamare $next[next[x, y], y]$ per trovare il nodo ancora successivo.



next =	A	B	C	D	E	F	G
A	-1	B	B	D	B	B	B
B	-1	-1	C	-1	C	C	C
C	-1	-1	-1	-1	E	E	E
D	-1	-1	C	-1	C	C	C
E	-1	-1	-1	-1	-1	F	G
F	-1	-1	-1	-1	-1	-1	G
G	-1	-1	-1	-1	-1	-1	-1