
ALGORITMI E STRUTTURE DATI

ANNO ACCADEMICO 2023-2024
PROFESSORE: FABIO MOGAVERO

Indice

1	Introduzione	3
1.1	Numero di Fibonacci	3
1.2	Introduzione alla terminologia sulla complessità	4
1.3	Prima definizione di O , Ω , Θ	6
2	Problema <i>Max Subsequence Sum</i>	7
3	Collezione di dati	10
3.1	Metodi per Stack e Queue	10
3.2	Metodi delle Collezioni di Dati	10
4	Come fare la ricerca in un array	12
4.1	Array con ripetizioni	12
4.2	Array senza ripetizioni	12
4.3	Array ordinato	12
4.4	Ricerca del minimo	14
4.5	Ricerca del successore	14
5	Liste	16
6	Alberi	18
6.1	Visite di alberi	19
6.1.1	Visita in profondità (PreOrder)	19
6.1.2	Visita in profondità (PostOrder)	20
6.1.3	Visita in profondità (InOrder)	20
6.1.4	Visita in ampiezza (Breadth-First-Search)	21
7	Alberi Binari di Ricerca (Binary Search Tree)	23
7.1	Algoritmi generici	23
7.2	Inserimento	24
7.3	Cancellazione	25
7.4	Alberi Perfettamente Bilanciati	27
7.5	Alberi Bilanciati	28
8	Alberi AVL	29
8.1	Violazioni di un AVL e come procedere	31
8.1.1	Inserimento	31
8.1.2	Cancellazione	34
9	Alberi Red-Black	38
9.1	Problemi nell'inserimento	39
9.1.1	Caso 1	39
9.1.2	Caso 1.b	40
9.1.3	Caso 2	40
9.1.4	Caso 3	41
9.2	Problemi nella cancellazione	42
9.2.1	Caso 1	42
9.2.2	Caso 2	43
9.2.3	Caso 3	44
9.2.4	Caso 4	45
10	Passaggio dal ricorsivo all'iterativo	48
11	Grafi	56
11.1	Percorso, Ciclo	60
11.2	Componenti	61
11.3	Raggiungibilità	61
11.4	Grafo orientato e non orientato	62
11.5	Distanza in un grafo	62

12 Visita in ampiezza	64
12.1 Lemma(Distanze)	65
12.2 Lemma(Invarianti)	65
13 Visita in profondità	67
13.1 Teorema (struttura a parentesi della DFV)	68
13.2 Lemma (percorsi nella foresta di visita)	69
13.3 Teorema (Percorso Bianco)	69
13.4 Classificazione degli archi	70
14 Ordinamento Topologico	72
15 Componenti connesse	77
16 Grafi pesati	82
16.1 Concetto di peso di un percorso	82
16.2 Algoritmo di Dijkstra	85
16.2.1 Teorema di correttezza	86
16.3 Algoritmo di Bellman-Ford	87
17 Complessità	88
18 Algoritmi di ordinamento	89
18.1 Insertion Sort	89
18.1.1 Correttezza	91
18.2 Selection Sort	92
18.3 Heap Sort	92
18.3.1 Albero Heap	92
18.3.2 Max-Heap	93
18.3.3 Algoritmo	93
18.4 Merge Sort	96
18.5 Quick Sort	98
19 Equazioni di ricorrenza	102
20 Alberi di decisione	105

1 Introduzione

Algoritmo: sequenza finita di istruzioni elementari eseguibili da un'interprete meccanico atto a trasformare i valori in ingresso in valori in uscita.

Sequenza finita di istruzioni = programma.

Elementari eseguibili da un interprete meccanico = modello computazionale.

Un **interprete meccanico** è qualunque cosa rappresentabile come sistema meccanico che può eseguire istruzioni.
Un esempio di modello computazionale è la [Macchina di Turing](#). È definita in questo modo:

$$M = \langle Q, q_I \in Q, q_F \in Q, \Sigma, \sqcup \in \Sigma, \delta \rangle$$

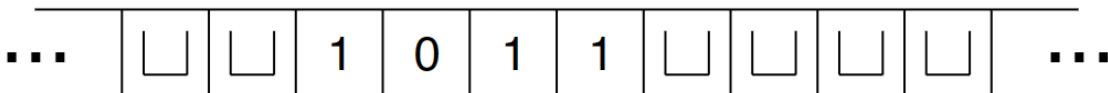
Q = insieme finito degli stati;

q_I = stato iniziale;

q_F = stato finale; δ = funzione di transizione $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{-1, 0, +1\}$

Σ = insieme finito di simboli (alfabeto);

Vediamo un esempio. Vogliamo che la macchina incrementi di 1 il valore sul nastro.



Q = inc (incrementa), halt (fermo) $\Sigma = 0, 1, \sqcup$

$(\text{inc}, 0) \mapsto (\text{halt}, 1, 0)$

$(\text{inc}, 1) \mapsto (\text{inc}, 0, -1)$

Per quanto potente sia, la Macchina di Turing presenta una limitazione: l'accesso ai dati che è per forza sequenziale. Un altro modello è il [Modello RAM](#) (Random Access Machine). La differenza sostanziale con la Macchina di Turing è la funzione di transizione δ .

1.1 Numero di Fibonacci

Problema 1: dato un numero $n \in \mathbb{N}$, calcolare l' n -esimo numero di Fibonacci.

Ricordiamo che la sequenza di Fibonacci è la seguente:

$$\begin{cases} F_0 = F_1 = 1 \\ F_{n \geq 2} = F_{n-1} + F_{n-2} \end{cases}$$

Scriviamo uno pseudo-codice di risoluzione del problema:

```
Signature : Fib : N --> N
    function Fib(n)
        if n <= 1 then
            return 1
        else
            a <-> Fib(n-1)
            b <-> Fib(n-2)
            return a + b
```

Calcoliamo quanto costa a livello di tempo. Supponiamo, per questa volta, che un'istruzione valga 1. Avremo :

```
Signature : Fib : N --> N
    function Fib(n) [1]
        if n <= 1 then ] 1
            return 1
        else
            a <-- Fib(n-1) 1 + Fib
            b <-- Fib(n-2) 1 + Fib
            return a + b 1
```

Nel caso in cui $n \leq 1$, verrà effettuato solo il primo if. Negli altri casi, invece, si attiverà l'else. Dunque avremo:

$$T_{Fib}(n) = \begin{cases} 2, & \text{se } n \leq 1 \\ 4 + T_{Fib}(n - 1) + T_{Fib}(n - 2) & \end{cases}$$

Si può osservare che $Fib(n) \leq T_{Fib}(n)$. Lo si dimostra per induzione.

Il numero di Fibonacci cresce esponenzialmente. La sua formula è:

$$Fib(n) = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{\sqrt{5}-1}{2})^n}{\sqrt{5}}$$

Dove $\frac{1+\sqrt{5}}{2}$ è la sezione aurea.

Vediamo ora un'altra versione del problema, calcoliamo la complessità temporale e valutiamo se è migliore o meno del precedente:

```
function Fib(n)
    a,b <-- 1
    While n >= 2 do
        c <-- a + b
        b <-- a
        a <-- c
        n <-- n - 1
    return a
```

La complessità temporale sarà dunque:

$$T_{Fib}(n) = \begin{cases} 2, & \text{se } n \leq 1 \\ 2 + 5(n - 1) & \end{cases}$$

1.2 Introduzione alla terminologia sulla complessità

Andiamo ad analizzare un'ennesima versione del seguente algoritmo, per vedere se è possibile ottenere un algoritmo migliore del precedente:

```

function Fib(n)
    a, b  $\leftarrow$  1
    While n  $\geq$  2 do
        a  $\leftarrow$  a + b
        b  $\leftarrow$  a - b
        n  $\leftarrow$  n - 1
    return a

```

Il penultimo algoritmo (*) e quello appena proposto (**) hanno lo stesso andamento lineare. Sicuramente (**) è leggermente più veloce, data l'assenza di una variabile. È da sottolineare quel "leggermente", in quanto la differenza tra i due algoritmi è davvero minima. In particolare, la costante moltiplicativa di (**) è poco più piccola di (*). Diremo che questi due fanno parte della stessa classe.

Con questo possiamo introdurre una nuova terminologia. Non entriamo ancora nei dettagli, ma è utile per capirne il senso generale. Vediamo prima di capire cosa si intende per *classe*. Per tutti gli algoritmi analizzati fino ad ora, il costo temporale è sempre una funzione lineare, cioè dipendente da *n*. Questa *n* è quello che intendiamo per *classe*. La nuova terminologia è costituita dai seguenti elementi:

- $O(n)$ (o grande di *n*)
- $\Omega(n)$ (omega grande di *n*)
- $\Theta(n)$ (theta grande di *n*)

Per capire la differenza tra i tre, vediamo un altro problema e i relativi algoritmi di risoluzione.

Problema 2: Calcolare, per $n \in \mathbb{N}$, la cardinalità dell'insieme $\{(i, j) \in \mathbb{N} \times \mathbb{N} / i \leq j \leq n\}$. In altre parole, si vuole calcolare il numero di coppie (i, j) tali che $i \leq j$.

```

Signature Count:  $\mathbb{N} \rightarrow \mathbb{N}$ 
function Count(n)
    c  $\leftarrow$  0
    for i from 0 to n do
        for j from 0 to n do
            if i  $\leq$  j then
                c  $\leftarrow$  c + 1
    return c

```

Calcoliamo la complessità temporale:

$$T_{Count}(n) = [(2(n+1)) + 1](n+1) + 2 = (2n+3)(n+1) + 2 = 2n^2 + 5n + 5$$

Arriviamo al dunque. Sicuramente, per il calcolo appena fatto, questo è un algoritmo la cui complessità temporale è una funzione con andamento quadratico, quindi la sua classe è n^2 . La domanda è: "Preso un qualsiasi *n*, si può ottenere un caso peggiore? Cioè, si può ottenere una complessità temporale di una classe superiore a quella appena calcolata?". Con questo specifico algoritmo, la risposta è "no", quindi possiamo dire che la funzione è di tipo $O(n^2)$.

Di contro, "preso un qualsiasi *n*, è possibile ottenere un caso migliore? Cioè, si può ottenere una complessità temporale di una classe inferiore a quella appena calcolata?". Con questo specifico algoritmo, anche adesso la risposta è "no", quindi possiamo dire che la funzione è di tipo $\Omega(n^2)$.

Ricapitolando, per un qualsiasi *n*, la complessità temporale sarà sempre una funzione con andamento quadratico. Dato che la classe è sempre la stessa, quindi abbiamo $O(n^2)$ e $\Omega(n^2)$, possiamo dire che questa funzione è di tipo $\Theta(n^2)$.

Vediamo ora un altro algoritmo che risolve sempre il Problema 2.

```

Signature Count : N → N
function Count(n)
    c ← 0
    for i from 0 to n do
        for j from i to n do
            c ← c + 1
    return c

```

1
1
1
1
1
n + 1

Questo algoritmo è sicuramente meglio del primo, data l'assenza del controllo. Calcoliamo la complessità temporale:

$$T_{Count}(n) = 2 + \sum_{i=0}^n 2(n-i+1) + 1 = 2 + 2 \sum_{i=0}^n (n-i+2) = 2 + 2 \left\{ \sum_{i=0}^n (n+2) - \sum_{i=0}^n i \right\} = 2 + 2(n+1)(n+2) - n(n+1)$$

Cosa notiamo da questo calcolo? In primo luogo, che la differenza con l'altro algoritmo è davvero minima; in secondo luogo che si tratta sempre di un'equazione quadratica. Applicando lo stesso ragionamento di prima, possiamo dire che è una funzione sia $O(n)$ sia $\Omega(n)$, dunque $\Theta(n)$

Andiamo ora ad analizzare un'ulteriore algoritmo

```

Function Count(n)
    c ← 0
    for i = 0 to n do
        c ← c + (n - i + 1)
    return c

```

1
1
1
1
1
(n + 1)

La complessità temporale sarà:

$$T_{Count}(n) = 2 + 5(n+1) = 5n + 7$$

Possiamo notare che presenta un andamento lineare e non quadratico come gli altri.

1.3 Prima definizione di O , Ω , Θ

$$f(n) = O(g(n)) \stackrel{\text{def}}{=} \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 (f(n) \leq c \cdot g(n))$$

$$f(n) = \Omega(g(n)) \stackrel{\text{def}}{=} \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 (f(n) \geq c \cdot g(n))$$

$$f(n) = \Theta(g(n)) \stackrel{\text{def}}{=} \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 (c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$$

2 Problema *Max Subsequence Sum*

Input: un array A di dimensione $n \in \mathbb{N}$ Output : la massima somma degli elementi di una sottosequenza. La indichiamo così:

$$0 \leq i \leq j < n \quad \sum_{k=i}^j A[k]$$

Con \max andiamo a prendere il massimo; con $0 \leq i \leq j < n$ prendiamo la sottosequenza; con la sommatoria andiamo a fare la somma degli elementi.

Analizziamo un primo algoritmo:

Signature : MSS : $N \times N \rightarrow N$
function MSS(A, n)

```

m ← -∞
for i = 0 to n - 1 do
    for j = i to n - 1 do
        s ← 0
        for k = i to j do
            s ← s + A[k]
        m ← max(m, s)
return m

```

$$\begin{array}{c}
 1 \\
 1 \\
 1 \\
 1 \\
 1 \\
 2 \\
 1 \\
 1
 \end{array}
 \left[\sum_{j=i}^{n-1} 3(j-i+1) + 3 \right] \quad \left[\sum_{i=0}^{n-1} \left(1 + \sum_{j=i}^{n-1} 3(j-i+1) + 3 \right) \right]$$

La complessità temporale sarà quindi:

$$T_{MSS}(n) = 2 + \sum_{i=0}^{n-1} \left(1 + \sum_{j=i}^{n-1} 3(j-i+1) + 3 \right) = 2 + \sum_{i=0}^{n-1} \left(1 + 3 \sum_{j=i}^{n-1} (j-i) + \sum_{j=i}^{n-1} 6 \right) = 2 + \sum_{i=0}^{n-1} \left(1 + 3 \sum_{j'=0}^{n-i-1} j' + 6(n-j) \right)$$

Analizziamo l'ultima parte. La sommatoria

$$3 \sum_{j'=0}^{n-i-1} j'$$

produce qualcosa di quadratico. Dato che c'è la sommatoria esterna, otteniamo la somma di qualcosa di quadratico che produce un qualcosa di cubico. Poi c'è

6($n - j$)

che è lineare e quindi, con la sommatoria esterna, abbiamo la somma di qualcosa di lineare che produce qualcosa di quadratico. Possiamo dunque affermare che la classe di questo algoritmo è cubica.

Ovviamente questo algoritmo può (e deve) essere migliorato. Vediamo:

Signature : MSS : $N \times N \rightarrow N$
function MSS(A, n)

```

function MSS(X, n)
    m ← -∞
    for i = 0 to n
        s ← 0
        for j = i to t
            s ← s + X[j]
            m ← max(m, s)
    m ← max(m, s)
return m

```

$$\text{lo } \begin{matrix} & 1 \\ & 1 \\ & 1 \\ 1 & 1 \\ 2 & 2 \\ 1 & 1 \end{matrix} \left[4(n - i) \right] \left[\sum_{i=0}^{n-1} 4(n - i) + 2 \right]$$

La complessità temporale sarà quindi:

$$T_{MSS}(n) = 2 + \sum_{i=0}^{n-1} 4(n-i) + 2$$

In conclusione, la classe di questo algoritmo è quadratico.

È possibile ottenere un algoritmo ancora migliore, quindi con andamento lineare. Prima di presentarlo, vediamo un lemma e scriviamo un algoritmo che lo risolve. Questo algoritmo è proprio quello che stiamo cercando.

Lemma:

Ipotesi: a) $\forall j \in [i, r] \ SUM[i, j] \geq 0$
b) $SUM[i, j] < 0$

Tesi: 1) $\forall p \in [i, j] \ \forall j \in [i, r] \ SUM[p, j] \leq SUM[i, j]$
2) $\forall p \in [i, r] \ \forall k \ SUM[p, r+k] < SUM[r+1, r+k]$

Dimostrazione Tesi (1):

$$SUM[i, j] \triangleq \sum_{h=1}^n A[h] = \sum_{h=i}^{p-1} A[h] + \sum_{h=p}^j A[h]$$

Analizziamolo nel dettaglio. Quello che abbiamo fatto è stato spezzare la prima sommatoria. Possiamo osservare che

$$\sum_{h=i}^{p-1} A[h] = SUM[i, p-1]$$

Questa è sicuramente ≥ 0 perché per come abbiamo preso p , questo va da i a j , ma j va da i a r . Quindi $p-1$ non arriva a r e possiamo dunque considerarlo come j nell'ipotesi (a). Osserviamo inoltre che

$$\sum_{h=p}^j A[h] = SUM[p, j]$$

Dobbiamo dimostrare che $SUM[p, j] \leq SUM[i, j]$, il che è sempre vero perché abbiamo appena visto che $SUM[i, j] = SUM[p, j]$ a cui viene aggiunta una quantità sicuramente positiva (cioè $SUM[i, p-1]$).

Dimostrazione Tesi (2):

$$SUM[p, r+k] = SUM[p, r] + SUM[r+1, r+k]$$

Ora, $SUM[p, r]$ è un sottopezzo di $SUM[i, r]$ che è una parte di $SUM[i, j]$. Quest'ultimo, per il punto (a), è sicuramente maggiore o uguale di 0. Quindi con $SUM[i, r]$, essendo un suo sottopezzo, nel caso migliore è uguale ma possiamo ottenere anche qualcosa di peggiore. Possiamo dunque dire che $SUM[p, r]$ è molto negativo (perché è negativo rispetto a qualcosa di già negativo). $SUM[p, r+k]$ è la somma di qualcosa di molto negativo e $SUM[r+1, r+k]$, dunque la tesi.

Vediamo l'algoritmo:

```
Function MSS(A, m)
    m ← -∞ 1
    s ← 0 1
    for i from 0 to n - 1 do 1
        s ← s + A[i] 2
        if s < 0 then 1
            s ← 0 1
        m ← max(m, s) 1
    return m 1
```



La sua complessità temporale sarà:

$$T_{MSS}(n) = 3 + 6n$$

Notiamo che si tratta di un algoritmo sicuramente migliore rispetto agli altri due, in quanto presenta un andamento lineare. Possiamo chiedere meglio di questo? No, perché dobbiamo per forza scorrere tutto l'array. Questo per avere la certezza di non aver dimenticato qualche elemento.

3 Collezione di dati

Una collezione di dati presenta diverse caratteristiche. Queste sono:

1. Tipo di dati da gestire;
2. Relazioni tra i dati;
3. Rappresentazione concreta in memoria;
4. Staticità/Dinamicità dei dati;
5. Operazioni di creazione/distruzione della struttura;
6. Operazioni di inserimento/rimozione dei dati;
7. Operazioni di accesso/modifica dei dati.

	ARRAY	LISTE CONCATENATE	ARRAY ORDINATI	LISTE ORDINATE	QUEUE	STACK
1	ALL	ALL	Ordinati Totalmente	Ordinati Totalmente	ALL	ALL
2	Al più un ordine totale	Al più un ordine totale	Ordine Totale	Ordine Totale	Ordine di arrivo	Ordine di arrivo
3	Rappresentazione contigua	Rappresentazione non contigua	Rappresentazione contigua	Rappresentazione non contigua	?	?
4	Statico	Dinamica	Statico	Dinamica	Dinamica	Dinamico
6	$\Theta(n)$	Inserimento: $\Theta(1)$ Cancellazione: $O(n) \Omega(1)$	$\Theta(n)$	$O(n) \Omega(n)$	$\Theta(1)$	$\Theta(1)$
7	Accesso Diretto: $\Theta(1)$ Ricerca: $O(n) \Omega(1)$	Ricerca: $O(n) \Omega(1)$	Accesso Diretto: $\Theta(1)$ Ricerca: $O(\log(n)) \Omega(1)$	$O(n) \Omega(1)$	$\Theta(1)$	$\Theta(1)$

n = quantità;

È $\Theta(1)$ grazie alla rappresentazione diretta;

Lettura e scrittura dell'unica cella accessibile. Vale anche per le lo stack.

3.1 Metodi per Stack e Queue

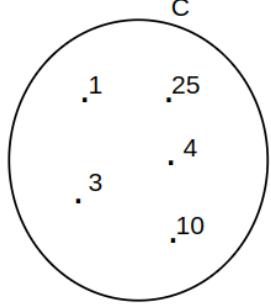
In questa sezione, vediamo quali sono i metodi principali per lo stack (o pila) e la queue (o coda).

PILE/STACK		CODE/QUEUE
EmptyStack : Stack(D)	funzioni di creazione	EmptyQueue : Queue(D)
Push : Stack(D) → Stack(D)	funzioni di inserimento	Enqueue : Queue(D)xD → Queue(D)
Top : Stack(D) → D	Funzioni di sola lettura	Head : Queue(D) → D
Pop : Stack(D) → Stack(D)	Funzione di lettura e rimozione	Head&Dequeue : Queue(D) → Queue(D)xD

3.2 Metodi delle Collezioni di Dati

Riguardo i metodi usati sulle Collezioni di Dati, quelli che andremo principalmente a studiare sono:

1. Search : Collection(D) x D → \mathbb{B} → Questa versione dice solo se il dato esiste
 $\mathbb{B} = \{0/1\} = \{F, V\} = \{\perp, \top\}$
Search : Collection(D) x D → RefD → Questa versione dice se il dato esiste e dove si trova
2. Min/Max : Collection(D) → RefD
3. Successor/Predecessor : Collection(D) x D → RefD
 $\text{Predecessor}(S, d) = \max\{d' \in S / d' < d\}$
 $\text{Successor}(S, d) = \min\{d' \in S / d' > d\}$



$\text{Predecessor}(C, 1) = \text{non esiste}$
 $\text{Successor}(C, 25) = \text{non esiste}$
Questo perché sono ordinati. In modulo n non accadrebbe e
 $\text{Successor}(C, 25) = 1$

 $\text{Successor}(C, 5) = 10$
 $\text{Successor}(C, 4) = 10$
 $\text{Predecessor}(C, 3) = 1$
 $\text{Predecessor}(C, 4) = 3$

Le funzioni appena viste, non sono distruttive (quindi non cambiano la struttura della collezione). Vediamo ora delle funzioni che lo sono:

4. Insert : Collection(D) x D → Collection(D)
5. Remove : Collection(D) x D → Collection(D)
6. EmptyX : Collection(D)
 $X = \text{nome di qualunque collezione (Es. EmptyQueue)}$

Assumendo che la struttura sia stata creata e popolata, ciò che per adesso faremo è quello di utilizzare le semplici funzioni (1), (2), (3).

4 Come fare la ricerca in un array

Gli algoritmo si differenziano in base al tipo di array che andiamo a considerare.

4.1 Array con ripetizioni

```
Signature : Array(D)xD → B RefD
function Search((A,n), d)
    for i from 0 to n-1 do
        if A[i] = d then
            return T &A[i]
    return F NULL
```

Seconda versione della Search

(A,n) è ciò che prima abbiamo chiamato C. "A" è l'indirizzo della prima cella e "n" è la dimensione dell'array. Questo algoritmo è $O(n)$ ma non $\Omega(n)$ perché, nel caso migliore, troviamo l'elemento in posizione 0. Quindi è $\Omega(1)$.

4.2 Array senza ripetizioni

Nonostante, in questo caso, abbiamo l'informazione che nell'array non sono presenti le ripetizioni, questo non ci dà la possibilità di avere un algoritmo migliore rispetto al precedente. Infatti, è identico al caso dell'array con le ripetizioni.

4.3 Array ordinato

In questo caso l'algoritmo è quello della ricerca binaria. Analizziamo sia l'algoritmo ricorsivo sia quello iterativo.

1. Ricorsivo:

```
function Search((A,n), d)
    return Search((A,n), 0, n, d)

Signature : SortedArray(D)xNxNxD → B
function Search((A,n), i, j, d)
    if j ≤ i then //caso base in cui l'array è vuoto
        return F

    else
        k ← ⌊  $\frac{i+j}{2}$  ⌋
        if A[k] = d then
            return T
        else if A[k] < d then
            return Search((A,n), k+1, j, d)
        else
            return Search((A,n), j, k, d)
```

2. Iterativo:

```

Function Search((A,n), d)
    (i, j, f)  $\leftarrow$  (0, n, F) ( $i \leftarrow 0$ ,  $j \leftarrow n$ ,  $f \leftarrow$  F (flag))
    While ( $i < j$ ) do
        k  $\leftarrow$   $\lfloor(i+j)/2\rfloor$ 
        if A[k] = d then
            f  $\leftarrow$  T
            break
        else if A[k] < d then
            i  $\leftarrow$  k+1
        else
            j  $\leftarrow$  k
    return f

```

Vediamo un'altro algoritmo, ricorsivo, simile a quello precedentemente proposto.

```

Function Search((A,n), i, j, d)
    if i < j then
        k  $\leftarrow$   $\lfloor(i+j)/2\rfloor$ 
        if A[k] > d then
            return Search((A,n), i, k, d)
        else if A[k] < d then
            return Search((A,n), k+1, j, d)
        else
            return T
    return ⊥

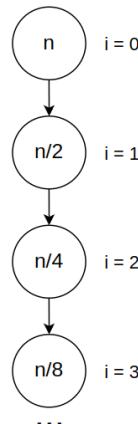
```

Calcoliamo il costo temporale:

$$T(n) = \begin{cases} 2, & \text{se } n \leq 1, \\ 4 + T(\lfloor \frac{n}{2} \rfloor), & \text{altrimenti} \end{cases}$$

$\frac{n}{2}$ viene da $\frac{i+j}{2} - i = \frac{i+j-2i}{2} = \frac{j-i}{2} = \frac{n}{2}$

Andiamo nello specifico. Dobbiamo calcolare quanti passi sono necessari per arrivare al caso base. Vediamo un piccolo schema per capire:



Da questo possiamo dedurre che arriveremo al caso base quando $\frac{n}{2^i} = 1$. Per approssimazione metteremo \leq . Dunque:

$$\frac{n}{2^i} \leq 1 \iff n \leq 2^i \iff \log_2(n) \leq \log_2(2^i) = i \log_2(2) = i$$

Quindi se $i = \log_2(n)$ significa che abbiamo raggiunto il caso base. Il costo totale è al più $5\lceil\log_2(n)\rceil = O(\log_2(n))$

4.4 Ricerca del minimo

Supponiamo di avere un array non ordinato. Vogliamo trovare l'indice dell'elemento più piccolo.

```
Signature Min:Array → N $\perp$ 
Function Min((A,n))
  if n>0 then
    m ← 0
    for i from 1 to n-1 do
      if A[m] > A[i] then
        m ← i
    return m
  return  $\perp$ 
```

La complessità di questo algoritmo è lineare e non possiamo sperare di ottenere un caso migliore. In particolare abbiamo $\Omega(n)$ e $O(n)$.

4.5 Ricerca del successore

Vogliamo trovare il successore di un determinato dato. In particolare, dobbiamo trovare il minimo tra i valori maggiori. Stiamo sempre nel caso di un array non ordinato.

```
Function Successor((A,n), d)
  if n>0 then
    s ←  $\perp$ 
    for i from 0 to n-1 do
      if A[i]>d then
        if (s= $\perp$ )  $\vee$  (A[s]>A[i])
          s ← i
    return s
  return  $\perp$ 
```

Per trovare il successore, il for deve essere per forza eseguito tutto, infatti non c'è niente che lo fa fermare all'attivazione di una particolare condizione. Questo implica che l'algoritmo sarà sempre lineare e mai costante.

Vediamo ora però il caso di un array ordinato:

```
Function Successor((A,n), i, j, d)
  if i<j then
    k ← ⌊(i+j)/2⌋
    if A[k] > d then
      s ← Successor((A,n), i, k, d)
      if s = ⊥ then
        s ← k
      return s
    else
      return Successor((A,n), k+1, j, d)
  return ⊥
```

La sua versione iterativa sarà:

```
Function Successor ((A,n), d)
  (i,j,s) ← (0,n,⊥)
  while i<j do
    k ← ⌊(i+j)/2⌋
    if A[k]>d then
      s,j ← k           //assegnamo a entrambi k
    else
      i ← k+1
  return s
```

Entrambi gli algoritmo hanno complessità logaritmica, in particolare $\Theta(\log_2(n))$.

5 Liste

Vediamo un algoritmo di ricerca di un elemento all'interno di una lista non ordinata:

```
Function Search (L, d)
    While L ≠ ⊥ do          //se la lista non è vuota
        if L.dat = d then
            return ⊤
        else
            L ← L.next
    return ⊥
```

Lista

Questo algoritmo ha complessità lineare. Con una lista ordinata potremmo trovare un algoritmo migliore, che non sia più lineare ma logaritmico. Il fulcro non sta nell'algoritmo in sé, ma nei confronti. Per esempio: un confronto tra numeri è molto più veloce e semplice di un confronto tra database. Quindi andiamo a lavorare in modo tale da ottimizzare il numero di confronti fatti usando un metodo simile alla ricerca binaria. Prendiamo una lista ordinata e usiamo un algoritmo ricorsivo:

```
Function Search (L, n, d)
    if n>0 then
        Z ← L      //facciamo una copia di L in Z così da poterci lavorare
        k ← ⌊n/2⌋

        for i from 1 to k do
            Z ← Z.next

        if Z.dat > d then
            return Search(L, k, d)
        else if Z.dat < d then
            return Search(Z.next, n-k-1, d)
        else
            return ⊤
    return ⊥
```

$n \leq |L|$

Complessivamente è lineare perché, nel for, si va da $\frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8}$ ecc, ma il confronto dei tre if viene eseguito non quanto la lunghezza della lista (come il for), ma un valore logaritmico di volte. Infatti:
Operazioni : $O(n)$ e $\Omega(1)$
Confronti : $O(\log(n))$ e $\Omega(1)$

Vediamo ora la versione iterativa:

```
Function Search (L, n, d)
  While n>0 do
    (Z,d) ← (L, ⌊n/2⌋)
    for i from 1 to k do
      Z ← Z.next

    if Z.dat > d then
      n ← k
    else if Z.dat < d then
      (L,n) ← (Z.next, n-k-1)
    else
      return ⊤
  return ⊥
```

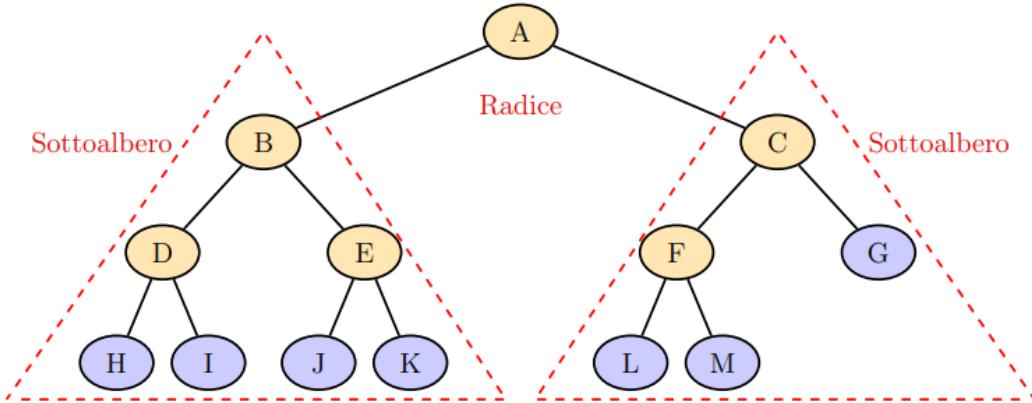
Possiamo così dare una definizione più assertiva di struttura statica: [una struttura è statica se in input abbiamo bisogno necessariamente della dimensione della struttura.](#)

6 Alberi

Un albero vuoto corrisponde a \emptyset . Un albero con un solo elemento corrisponde a $\{n_1\}$. Nel caso di $\{n_1, n_2\}$, questo insieme non basta a definire un albero perché non conosciamo le relazioni tra i due elementi. Fatta questa premessa, diamo la definizione di albero:

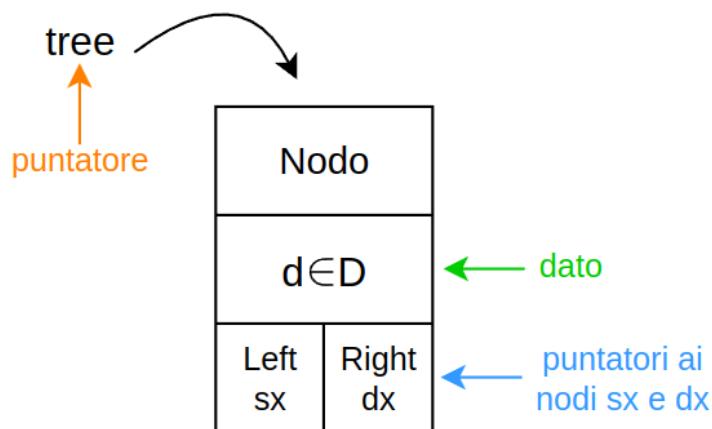
T è un albero se $T = \emptyset$ o se $\exists n \in T : \exists T_1, T_2 \subseteq T - \{n\}$ tali che $T_1 \cap T_2 = \emptyset$ (T_1 e T_2 sono alberi e sono disgiunti).

Lavoreremo sempre con alberi binari. Un **albero binario** è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come **figlio sinistro** e **figlio destro**.



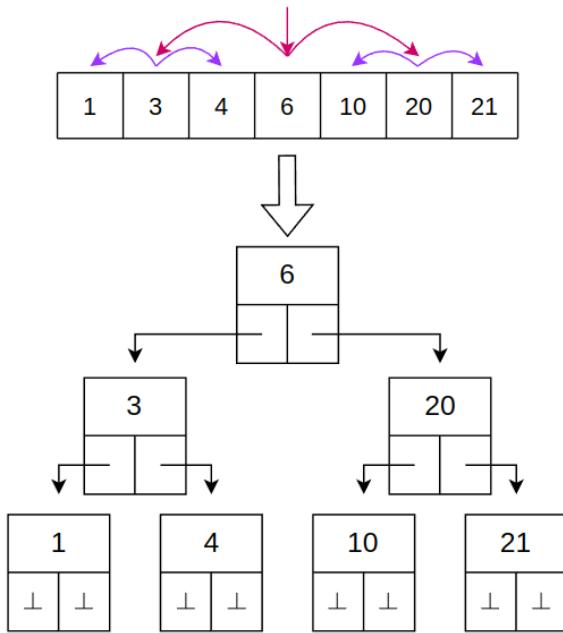
- A è la **radice**
- B, C sono radici dei **sottoalberi**
- D, E sono **fratelli**
- D, E sono **figli** di B
- B è il **padre** di D, E
- I nodi viola sono **foglie**
- Gli altri nodi sono **nodi interni**

Vediamo la struttura di un **nodo**:



6.1 Visite di alberi

Prendiamo un array e convertiamolo in un albero



Analizziamo il caso in cui vogliamo stampare i dati all'interno dell'albero. Possiamo scorrere l'albero in diversi modi:

- Visita in ampiezza;
- Visita in profondità:
 - PreOrder;
 - PostOrder;
 - InOrder.

6.1.1 Visita in profondità (PreOrder)

```
Function Print(T)
  if T ≠ ⊥ then
    printf(T.dat)
    Print(T.sx)
    Print(T.dx)
```

Output: 6 3 1 4 20 10 21

L'algoritmo ufficiale è:

F:DxA → A
 ↗ accumulatore

```
Function DFS-PreOrder(T,F,a)
    if F ≠ ⊥ then
        a ← F(T.dat,a)
        a ← DFS-PreOrder(T.sx,F,a)
        a ← DFS-PreOrder(T.dx,F,a)
    return a
```

6.1.2 Visita in profondità (PostOrder)

```
Function Print(T)
    if T ≠ ⊥ then
        Print(T.sx)
        Print(T.dx)
        printf(T.dat)
```

Output: 1 4 3 10 21 20 6

L'algoritmo ufficiale è:

```
Function DFS-PreOrder(T,F,a)
    if F ≠ ⊥ then
        a ← DFS-PreOrder(T.sx,F,a)
        a ← DFS-PreOrder(T.dx,F,a)
        a ← F(T.dat,a)
    return a
```

6.1.3 Visita in profondità (InOrder)

```
Function Print(T)
    if T ≠ ⊥ then
        Print(T.sx)
        printf(T.dat)
        Print(T.dx)
```

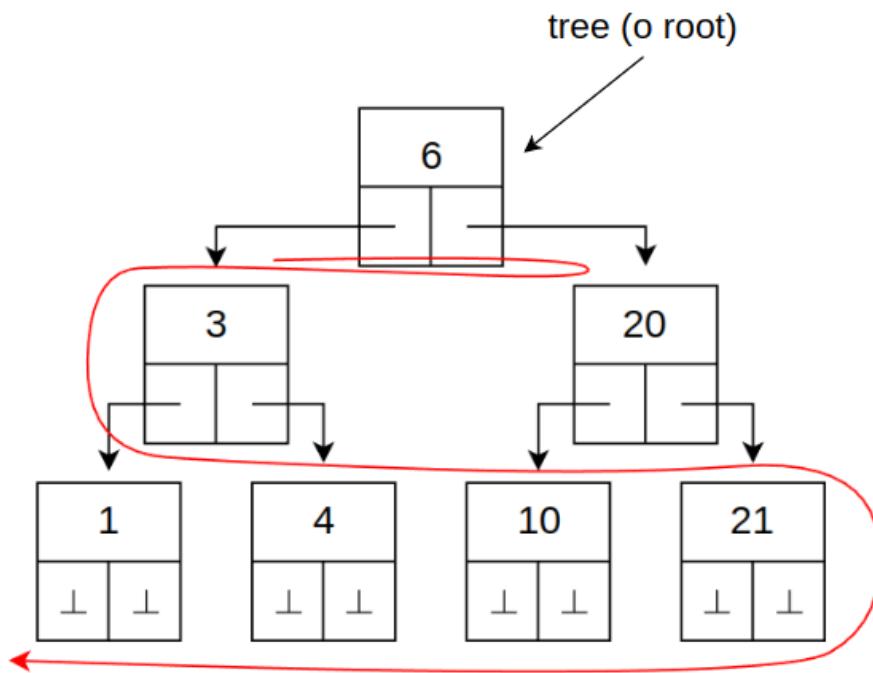
Output: 1 3 4 6 10 20 21

L'algoritmo ufficiale è:

```
Function DFS-PreOrder(T,F,a)
  if F ≠ ⊥ then
    a ← DFS-PreOrder(T.sx,F,a)
    a ← F(T.dat,a)
    a ← DFS-PreOrder(T.dx,F,a)
  return a
```

6.1.4 Visita in ampiezza (Breadth-First-Search)

In questo caso, come esempio non usiamo l'algoritmo di stampa (come negli altri casi), ma ne andiamo a vedere uno più specifico per la ricerca in generale.



```

Signature BFS : BT(D)xA → AxA → A
Function BFS(T,F,a)
  if T ≠ ⊥ then
    Q ← Singleton Queue(T)
  repeat
    (Q,T) ← Head&Dequeue(Q)

    if T.sx ≠ ⊥ then
      Q ← Enqueue(Q,T,sx)
    if T.dx ≠ ⊥ then
      Q ← Enqueue(Q,T,dx)

    a ← F(T.dat,a)
  until
    isEmpty(Q)
  return a

```

Dove BT sta per Binary Tree, A per accumulatore e Q per la coda. In particolare, notiamo l'utilizzo del ciclo *repeat-until* che ripete le istruzioni fino a quando la condizione diventa vera.

7 Alberi Binari di Ricerca (Binary Search Tree)

Definizione:

Un albero binario di ricerca è un albero binario τ tale che $\forall x \in \tau$:

- $\forall y \in x.sx (y.dat < x.dat)$
- $\forall y \in x.dx (x.dat < y.dat)$

Dove $x.sx$ è il sottoalbero sinistro di τ e $x.dx$ è il sottoalbero destro di τ .

7.1 Algoritmi generici

Vediamo quali sono gli algoritmi sui ABR:

Funzione di ricerca:

```
Function Search (T,d)
  if T ≠ ⊥ then
    if T.dat > d then
      return Search(T.sx,d)
    else if T.dat < d then
      return Search(T.dx,d)
    else //T.dat = d
      return ⊤/T.dat
  reuturn ⊥
```

Funzione di ricerca del minimo (ricorsivo):

```
Function RecMin(T)
  if T.sx = ⊥ then
    return ⊤/T.dat
  else
    return RecMin(T.sx)
```

Versione iterativa:

```
Function ItrMin(T)
  While T.sx ≠ ⊥ do
    T ← T.sx
  return ⊤/T.dat
```

Ricerca del Successore:

```
Function Successor(T,d)
    if T ≠ ⊥ then
        if T.dat > d then
            s ← Successor(T.sx,d)
            if s = ⊥ then
                s ← T
            return s
        else
            return Successor(T.dx,d)
    return ⊥
```

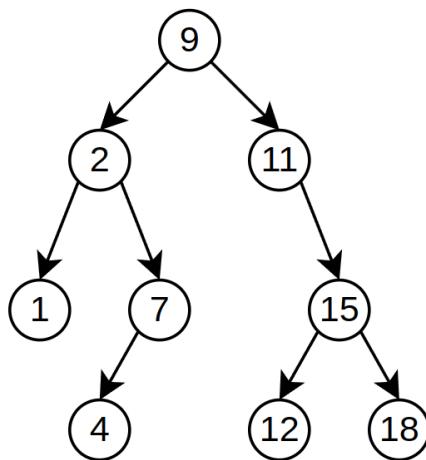
Questo algoritmo non è tale-recursivo, cioè non ha le chiamate ricorsive solo alla fine. Una versione del genere è la seguente:

```
Function Successor(T,d)
    return Successor(T,d,)

Function Successor(T,d,s)
    if T ≠ ⊥ then
        if T.dat > d then
            return Successor(T.sx,d,T)
        else
            return Successor(T.dx,d,s)
    return s
```

7.2 Inserimento

Supponiamo di avere il seguente albero:

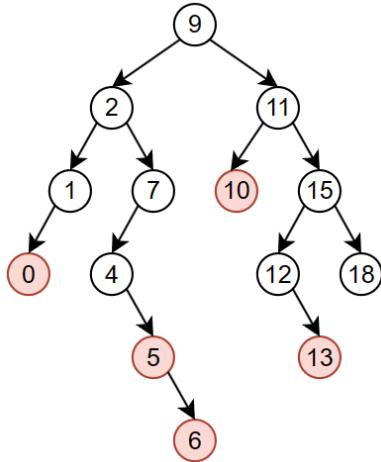


Vogliamo inserire il numero 5. I passaggi da seguire sono i seguenti:

- $5 < 9 \rightarrow$ andiamo a sinistra;
- $5 > 2 \rightarrow$ andiamo a destra;

- $5 < 7 \rightarrow$ andiamo a sinistra;
- $5 > 4 \rightarrow$ mettiamo 5 come figlio destro di 4;

Applichiamo lo stesso ragionamento per tutti i numeri che vogliamo inserire. Potremmo quindi ottenere un albero del genere:



L'algoritmo è:

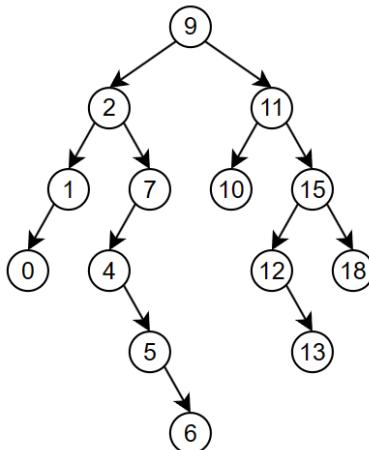
```

Function Insert(T,d)
  if T =  $\perp$  then
    T  $\leftarrow$  BuildNode(d) //tipo malloc in C
  else
    if T.dat > d then
      T.sx  $\leftarrow$  Insert(T.sx,d)
    else if T.dat < d then
      T.dx  $\leftarrow$  Insert(T.dx,d)
  return T

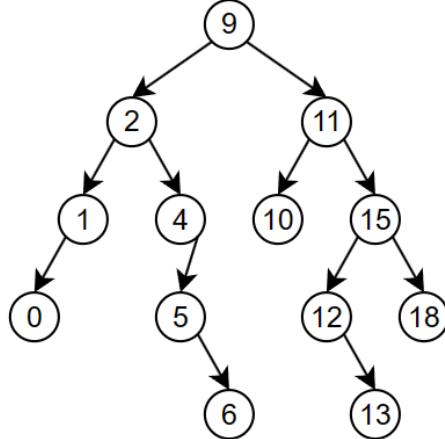
```

7.3 Cancellazione

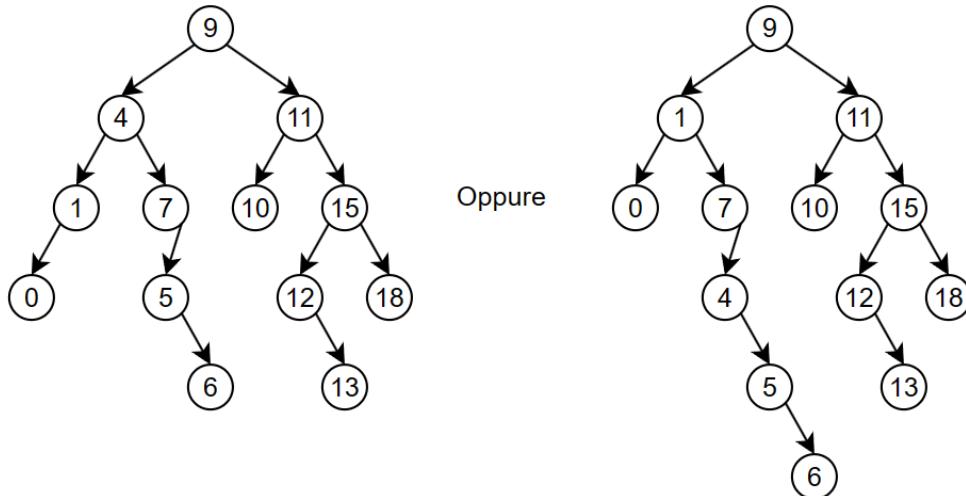
Abbiamo questo albero binario di ricerca:



Vogliamo cancellare il nodo 7. Otteniamo:



Volendo cancellare, invece, il nodo 2, le cose si complicano. Quella che facciamo non è una vera e propria cancellazione. Andiamo a sostituire il nodo in esame (il 2 in questo caso) o con il minimo del sottoalbero a destra o con il massimo del sottoalbero a sinistra. Avremo dunque:



L'algoritmo è:

```

Function Delete(T, d) //per trovare il dato da cancellare
  if T ≠ ⊥ then
    if T.dat > d then
      T.sx ← Delete(T.sx, d)
    else if T.dat < d then
      T.dx ← Delete(T.dx, d)
    else
      T ← DeleteData(T)
  return T
  
```

```

Function DeleteData(T) //per cancellare il dato effettivo
  if T.sx = ⊥ then
    T ← SkipRight(T)
  else if T.dx = ⊥ then
    T ← SkipLeft(T)
  else
    T.dat ← Get&DeleteMin(T.dx,T)
  return T

```

<pre> Function SkipLeft(T) y ← T.sx Deallocate(T) return y </pre>	<pre> Function SkipRight(T) y ← T.dx Deallocate(T) return y </pre>
---	--

```

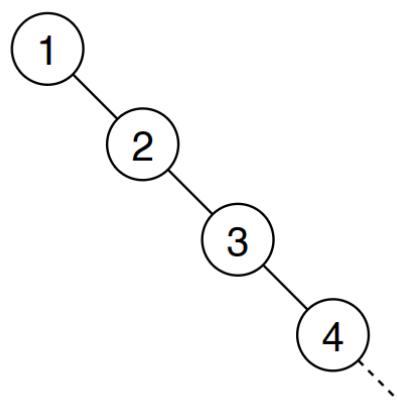
Function Get&DeleteMin(T,p)
  if T.sx = ⊥ then
    d ← T.dat
    y ← SkipRight(T)
    SwapChild(p,T,y)
  else
    d ← Get&DeleteMin(T.sx,T)
  return d

```

In particolare, *SwapChild* funziona in base a se T è figlio sinistro o destro di p . Nel primo caso y viene messo a sinistra; nel secondo caso viene messo a destra.

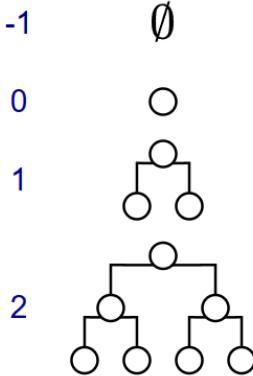
7.4 Alberi Perfettamente Bilanciati

Con le funzioni date, potremmo ottenere un albero degenere, come ad esempio:



Per ovviare al problema, dobbiamo prima introdurre la classe di alberi che ci interessa. Tecnicamente, l'albero migliore è quello in cui, a parità di nodi, è il più basso. In particolare, l'albero pieno. Vediamo degli esempi:

Altezza:



Da questo grafico possiamo notare che il numero di nodi è pari a $2^{h+1} - 1$. Gli alberi pieni però presentano un grande problema con la memorizzazione dei dati, in quanto il numero di nodi non è flessibile. Per questo motivo, gli alberi migliori sono quelli perfettamente bilanciati.

Definizione: Un albero T è perfettamente bilanciato $\iff \forall x \in T (|x.sx| - |x.dx| \leq 1)$

$\forall x \in T$ indica che la proprietà vale per ogni nodo in T .

$|x.sx|$ e $|x.dx|$ indicano la cardinalità rispettivamente del numero di nodi del sottoalbero sinistro e del sottoalbero destro.

Più semplicemente, la definizione afferma che tra il sottoalbero sinistro e il sottoalbero destro c'è al più uno scarto. Un po' come quando si divide a metà un array.

Per quanto riguarda l'altezza, in questo caso abbiamo:

$$2^{h+1} - 1 = n \Leftrightarrow 2^{h+1} = n + 1 \Leftrightarrow \log_2(2^{h+1}) = \log_2(n + 1) \Leftrightarrow h + 1 = \log_2(n + 1) \Leftrightarrow h = \log_2(n + 1) - 1$$

Mentre il numero di nodi sarà pari a:

$$N(h) \leq 1 + \text{Max}N(h-1) + \text{Max}N(h-1) \Rightarrow \text{Max}N(h) = 1 + 2\text{Max}N(h-1)$$

Se ci sono molti inserimenti e cancellazioni da effettuare, questa non è la famiglia giusta da utilizzare, in quanto comporta una o più rotazioni lineari nel numero di nodi.

7.5 Alberi Bilanciati

Quello che ci serve è una famiglia di alberi in cui l'altezza è sempre logaritmica rispetto ai nodi e flessibile rispetto all'inserimento e/o alla cancellazione. Questi sono gli alberi bilanciati, in cui $h = \Theta(\log(n))$. Potremmo scrivere anche $O(\log(n))$ perché sotto di un albero pieno non possiamo andare.

Definizione: un albero T è pieno se $h = \Theta(\log(n))$.

Gli alberi bilanciati si dividono in due gruppi:

- Alberi AVL;
- Alberi RB (Red-Black).

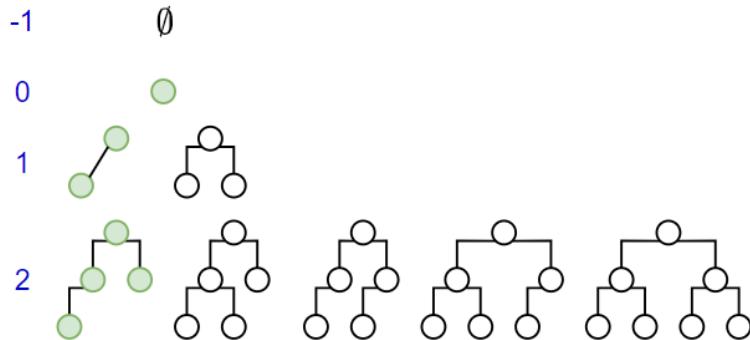
8 Alberi AVL

Definizione: un albero T è AVL $\Leftrightarrow \forall x \in T (|h(x.sx) - h(x.dx)| \leq 1)$.

Un albero perfettamente bilanciato è AVL, ma non è vero il contrario.

Un albero AVL si dice minimo se a parità di altezza ha il minimo numero di nodi. Ad esempio:

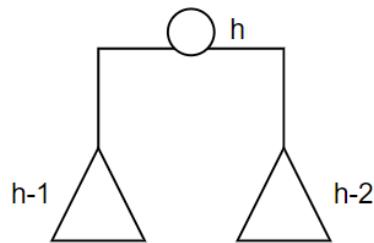
Altezza:



Gli alberi con i nodi verdi (e quello di altezza -1) sono tutti AVL minimi.

Questa famiglia di alberi ci interessa perché così non dobbiamo controllare che ogni albero sia AVL perché sappiamo che, a parità di altezza, se sono migliori del minimo allora non c'è da preoccuparsi.

Dobbiamo ora dimostrare che un AVL di altezza h sia costituito da un AVL di altezza $h-1$ a sinistra e un AVL di altezza $h-2$ a destra (vale anche il contrario). Visivamente avremo:



Perché il sottoalbero destro (o sinistro, a seconda di come lo si costruisce) deve avere altezza $h-2$? Non può avere altezza $h-1$ perché altrimenti non sarebbe minimo; non può inoltre avere altezza $h-3$ perché altrimenti ci sarebbe troppa differenza di altezza e l'albero non sarebbe più un AVL.

$$NN_{MAVL}(h) = \begin{cases} 0, & \text{se } h = -1 \\ 1, & \text{se } h = 0 \\ 1 + NN_{MAVL}(h-1) + NN_{MAVL}(h-2) & \text{altrimenti} \end{cases}$$

Come possiamo notare l'ultima parte somiglia molto alla sequenza di Fibonacci. Convinciamocene.

h	0	1	2	3	4	5	6
$NN_{MAVL}(h)$	1	2	4	7	12	20	33
Fib(h)		1	1	2	3	5	8

La riga di $NN_{MAVL}(h)$ sembra la riga della sequenza di Fibonacci riscalata di 3 e sottraendo -1. Quindi possiamo dire che:

$$NN_{MAVL}(h) = Fib(h + 3) - 1$$

Questo è solo un ragionamento intuitivo, ma pare andare bene. Ora passiamo alla vera e propria dimostrazione. La facciamo per induzione. Abbiamo due casi base, 0 e 1. Basta applicare la formula appena data e controllare nella tabella. Per il passo induttivo, invece:

$$NN(h) = 1 + NN(h - 1) + NN(h - 2) = 1 + [Fib((h - 1) + 3) - 1] + [Fib((h - 2) + 3) - 1] = Fib(h + 3) - 1$$

che è proprio quello che volevamo dimostrare.

In particolare $[Fib((h - 1) + 3) - 1] + [Fib((h - 2) + 3) - 1] = Fib(h + 3) - 1$ perché, per definizione di Fibonacci, $Fib(h + 2) + Fib(h + 1) = Fib(h)$.

Usiamo quello che abbiamo dimostrato:

$$NN(h) = Fib(h + 3) - 1 \leq \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+3} - 1$$

Poniamo $\left(\frac{1 + \sqrt{5}}{2} \right) = \varphi$ e otteniamo:

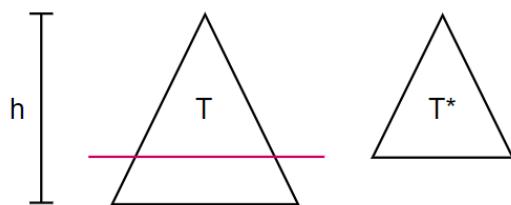
$$\sqrt{5}(NN(h) + 1) \leq \varphi^{h+3} \Leftrightarrow \log_{\varphi}(\sqrt{5}(NN(h) + 1)) \leq h + 3 \Leftrightarrow \log_{\varphi}(\sqrt{5}) + \log_{\varphi}(NN(h) + 1) \leq h + 3$$

Abbiamo così trovato che l'altezza è logaritmica sul numero di nodi. In particolare $h = O(\log(NN(h)))$.

Proposizione: $\forall T \in AVL$ con n nodi $\exists T^* \in MAVL$ con $\leq n$ nodi tale che $h(T) \leq h(T^*)$

Dimostrazione (per assurdo):

Il concetto è quello di prendere la proposizione, negarla e vedere che ci porta a una contraddizione. La sua negazione è: $\exists T \in AVL$ di dimensione n tale che $\forall T^* \in MAVL$ di dimensione n , $h(T) > h(T^*)$



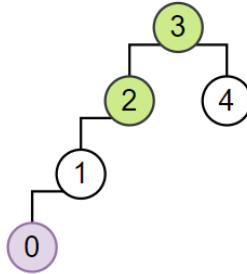
Prendiamo l'albero T e lo tagliamo togliendo le foglie di altezza h . Otteniamo un albero AVL della stessa altezza di T^* ma con meno nodi. Dato che T^* era minimo, abbiamo una contraddizione perché l'esistenza di questo nuovo albero fa sì che T^* non sia più il minimo.

Bisogna dimostrare, però, che questo taglio non viola le proprietà di AVL. Essendo T un AVL, vuol dire che lo scarto tra i sottoalberi destro e sinistro è di al più 1. Nel caso fossero inizialmente uguali, rimarrebbero tali anche dopo il taglio. Se invece, all'inizio, avessero scarto di 1, dopo il taglio diventerebbero uguali. In conclusione, il taglio non viola le proprietà di AVL.

8.1 Violazioni di un AVL e come procedere

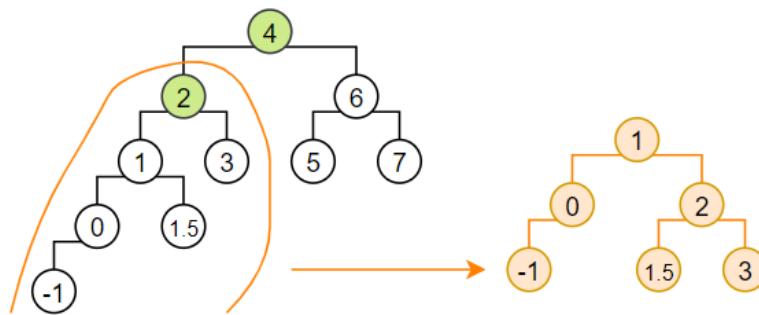
8.1.1 Inserimento

Supponiamo di avere questo albero:

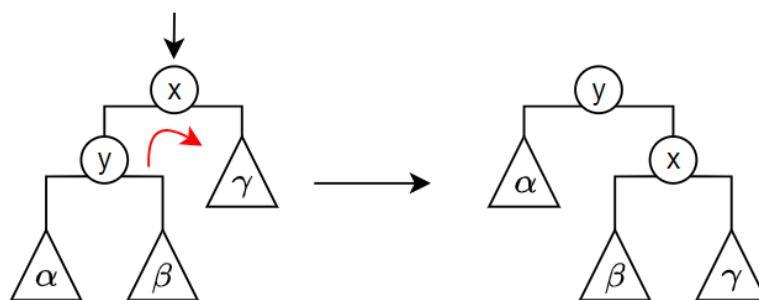


Vogliamo aggiungere lo 0. Fatto così, otterremo una violazione. Partendo dal fondo, possiamo capire che le foglie non sono un problema, ma lo sono i nodi verdi.

Una soluzione sarebbe quella di ruotare uno dei rami. Per capire meglio, vediamo un esempio più complesso:



L'albero a sinistra è quello di partenza e i nodi verdi sono quelli che dobbiamo considerare per sistemare la violazione. Quello che facciamo è prendere il sottoalbero sinistro (quello cerchiato in arancione) e applicare quella che viene chiamata **Rotazione Left To Right**. In generale, funziona così:



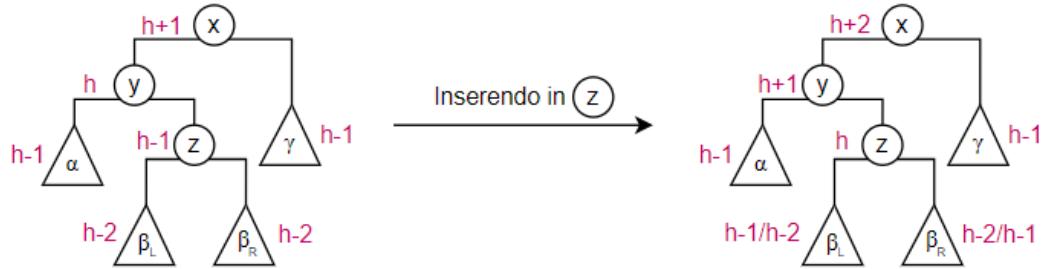
L'algoritmo è il seguente:

```

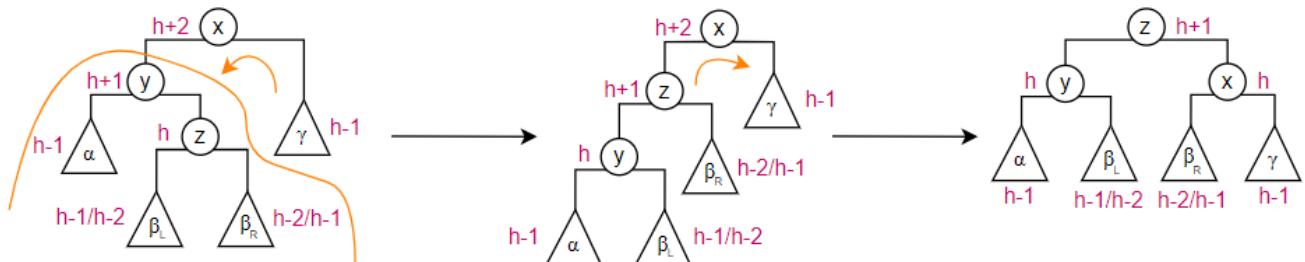
Signature BT → BT
Function L2RRot(T)
  y ← x.sx
  x.sx ← y.dx
  y.dx ← x
  return y
  
```

Importante: le rotazioni non violano mai le proprietà di un BST.

Abbiamo visto il caso in cui l'inserimento del nuovo nodo avviene nella parte più esterna dell'albero. E se invece l'inserimento viene fatto al centro? Vediamo meglio:



Possiamo notare che, a prescindere dal fatto che l'inserimento venga fatto in β_L o β_R , si crea uno sbilanciamento in y e γ in quanto c'è uno scarto di 2. Per risolvere il problema, non è sufficiente fare una singola rotazione (come abbiamo visto nel caso precedente), ma ne dobbiamo effettuare due: una interna da destra verso sinistra e una esterna da sinistra verso destra.



È importante notare l'importanza dell'altezza in questi casi, perché è proprio questa che ci indica la presenza di uno sbilanciamento. Un nodo degli AVL è infatti costruito in questo modo:

L'altezza viene aggiornata man mano che si inseriscono nuovi nodi. In particolare, l'altezza delle foglie è pari a 0. Man mano che si sale si aggiornano le altezze aggiungendo 1.

Vediamo ora i vari algoritmi per la risoluzione dello sbilanciamento. La prima è la funzione che restituisce l'altezza:

D	ht
sx	dx

→ Altezza

```
Function Height(x)
    return (if x = ⊥ then -1 else x.ht)
```

Questa è la funzione di aggiornamento dell'altezza a seguito di una modifica o cancellazione.

```
Procedure UpdateHeight(x)
    hL ← Height(x.sx)
    hR ← Height(x.dx)
    x.ht ← 1 + max(hL, hR)
```

Questa è la funzione di rotazione.

```
Function L2RRotAVL(x)
    y ← L2RRot(x)
    UpdateHeight(x)
    UpdateHeight(y)
    return y
```

Questa è la funzione di doppia rotazione.

```
Function LDRotAVL(x) //Left Double Rotation AVL
    x.sx ← R2LRotAVL(x.sx)
    return L2RRotAVL(x)
```

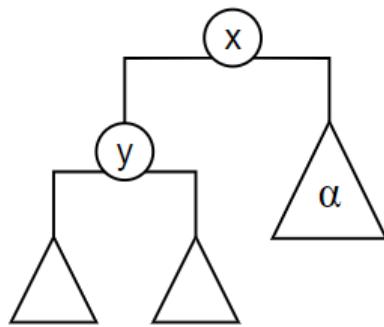
Ora vediamo l'algoritmo generico.

```
Function InsertAVL(x, d)
    if x = ⊥ then
        x ← BuildAVLNode(d)
    else
        if x.dat > d then
            x.sx ← InsertAVL(x.sx, d)
            x ← LBalance(x) //Left Balance, controlla se c'è un problema e lo sistema
        else if x.dat < d then
            x.dx ← InsertAVL(x.dx, d)
            x ← RBalance(x) //Right Balance, stesso discorso di Left Balance ma a destra
    return x
```

Questa è la funzione di bilanciamento.

```
Function LBalance(x)
    if Height(x.sx) - Height(x.dx) > 1 then
        if Height(x.sx.sx) > Height(x.sx.dx) then
            x ← L2RRotAVL(x)
        else
            x ← LDRotAVL(x)
    else
        UpdateHeight(x)
    return x
```

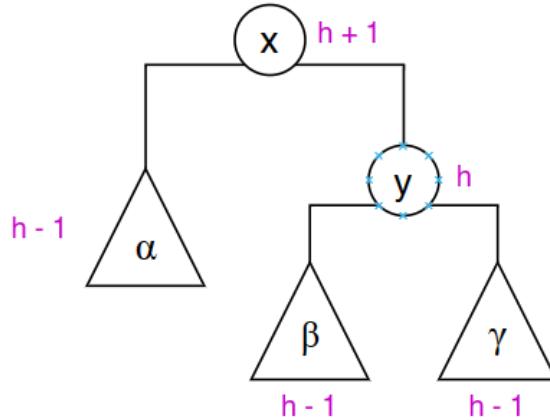
Vediamo bene perché il richiamo di *UpdateHeight()* nell'ultimo else. Supponiamo di avere questo albero:



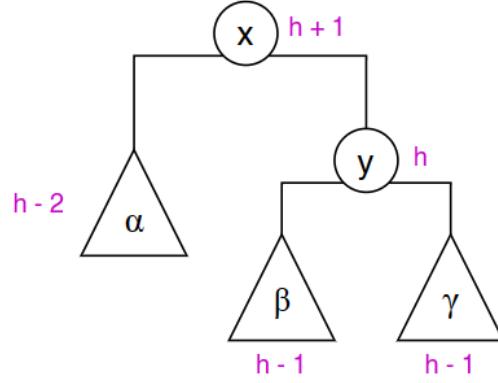
Aggiungendo a sinistra (quindi in *y*), *y* diventa più profondo di 1. Supponendo che, prima dell'inserimento, *y* e *α* avevano la stessa altezza, adesso hanno uno scarto di 1 e questo va ancora bene. Non bisogna però dimenticare che è comunque cambiata l'altezza di tutto l'albero, quindi bisogna aggiornare l'altezza anche nella radice.

8.1.2 Cancellazione

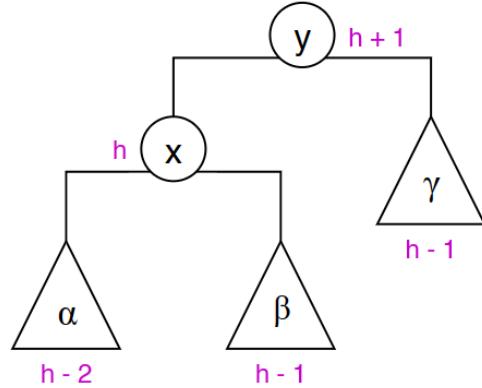
Supponiamo di avere:



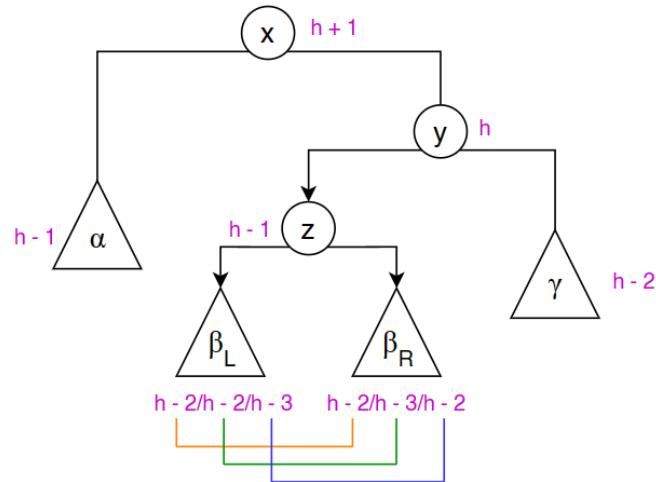
Cancelliamo un nodo nel sottoalbero sinistro. Otterremo:



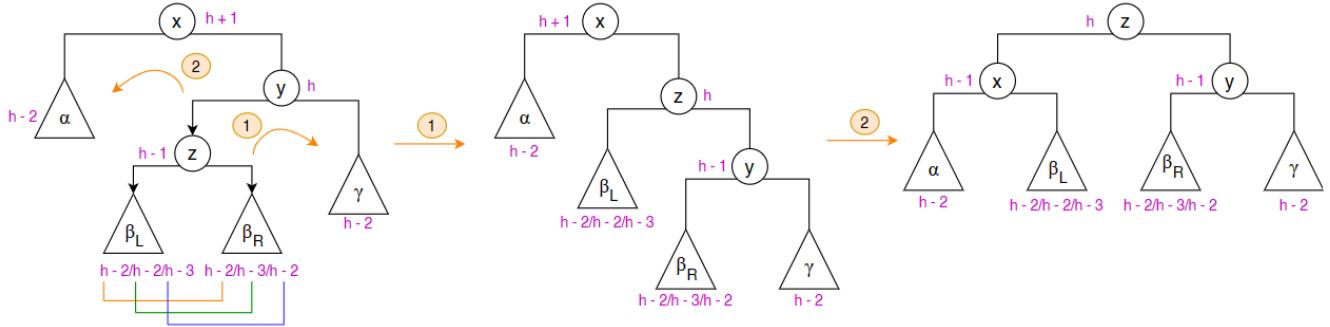
Come possiamo notare, tra il figlio sinistro della radice e quello destro c'è uno scarto di 2, il che viola la condizione di AVL. Per correggere ci basta fare una semplice rotazione da destra verso sinistra. Da notare però che questo procedimento lo possiamo effettuare solo se β e γ hanno lo stesso peso o se γ è più pesante di al più 1. Applicando la rotazione, otteniamo:



Ma cosa succede se invece il peso maggiore non lo ha γ ma β ? Avremo:



Cancellando sempre nel sottoalbero sinistro (quindi in α) vediamo che c'è uno scarto di 2. Per risolverlo una singola rotazione non basta e quindi ne dobbiamo effettuare due:



Il problema è stato risolto ma, come possiamo notare, l'altezza dell'albero è diminuita, in quanto alla radice non abbiamo più $h+1$ ma h .

La situazione peggiore possibile la possiamo riscontrare quando andiamo a cancellare un nodo in un AVL minimo, in particolare quando andiamo a cancellare nell'albero più corto. In questo caso bisogna effettuare diverse rotazioni fino alla radice.

Vediamo ora l'algoritmo generale della cancellazione:

```
Function DeleteAVL(x,d)
  if x ≠ ⊥ then
    if x.dat > d then
      x.sx ← DeleteAVL(x.sx,d)
      x ← R.Balance(x)
    else if x.dat < d then
      x.dx ← DelteAVL(x.dx,d)
      x ← L.Balance(x)
    else
      x ← DeleteDataAVL(x) //funzione accessoria che cancella il dato
  return x
```

```
Function DeleteDataAVL(x)
  if x.sx = ⊥ then
    x ← SkipRight(x)
  else if x.dx = ⊥ then
    x ← SkipLeft(x)
  else
    x.dat ← Get&DeleteMinAVL(x.dx,x)
    x ← L.Balance(x)
  return x
```

→ padre

```
Function Get&DeleteMinAVL(x,p)
    if x.sx = ⊥ then
        d ← x.dat
        y ← SkipRight(x)
    else
        d ← Get&DeleteMinAVL(x.sx,x)
        y ← R.Balance(x)
    SwapChild(p,x,y)
    return d
```

9 Alberi Red-Black

L'altezza di un albero AVL è di circa:

$$h(T) \approx 1.44 \log(n + 2) - 0.328$$

mentre l'altezza di un albero pieno è:

$$h(T) = \lceil \log_2(n + 1) - 1 \rceil$$

Il fattore moltiplicativo *1.44* ci dice che gli AVL, a parità di nodi con un albero pieno, è più basso. Questo perché l'obiettivo principale di un albero AVL è mantenere il bilanciamento per garantire un tempo di ricerca efficiente, mentre un albero pieno non ha questo requisito di bilanciamento e può crescere in altezza più velocemente. Questa caratteristica degli AVL viene "pagata" molto, in quanto capita spesso di dover bilanciare.

Gli alberi Red-Black cercano di "rilassare" questo vincolo degli AVL, mettendo un *2* al posto di *1.44*. Succede quindi che "pagniamo" di più la costante moltiplicativa, ma abbiamo una situazione più tranquilla. Possiamo quindi dire che un albero Red-Black con *n* nodi (interni) ha altezza:

$$h \leq 2 \log_2(n + 1)$$

Dunque, gli alberi Red-Black sono alberi bilanciati, e le operazioni fondamentali di ricerca, min, max, predecessore,successore sono eseguite in tempo $O(\log_2 n)$.

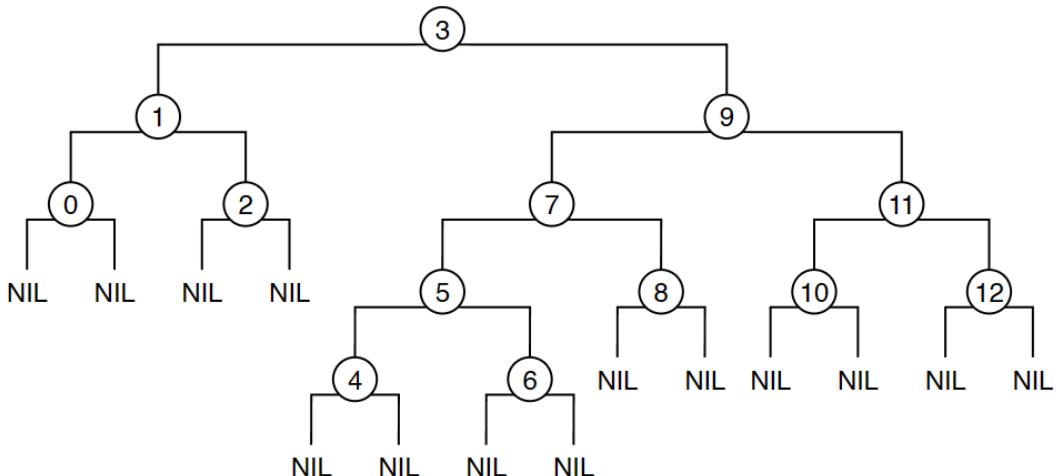
Definizione:

- 1 È un ABR/BST (albero binario di ricerca/binary search tree);
- 2.a I dati sono solo sui nodi interni;
- 2.b Le foglie sono identiche e NIL (non contengono i dati);
- 3.a Tutti i nodi sono colorati con rosso o nero;
- 3.b Tutte le foglie sono nere;
- 3.c Nodi rossi non possono avere figli rossi;
- 3.d Partendo da un qualsiasi nodo interno, ogni percorso che raggiunge le foglie ha lo stesso numero di nodi neri.

In particolare il "numero di nodi neri" prende il nome di lunghezza nera. Per il calcolo di questa non si conta mai la radice, che si presuppone nera.

Il nodo di un albero red-black è costituito da: il dato, il colore, il puntatore al figlio sinistro e il puntatore al figlio destro.

Vediamo un esempio e cerchiamo di capire se e come farlo diventare red-black.

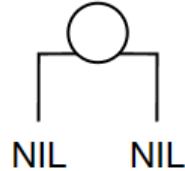


In questo caso abbiamo che $1 \leq h_N(T) \leq 3$, con h_N = altezza nera. Dimostriamo (per induzione) che l'altezza dell'albero red-black non supera il doppio dell'albero pieno. Per farlo abbiamo bisogno di una proprietà:

$$\#Int(T) \geq 2^{h_N(T)} - 1$$

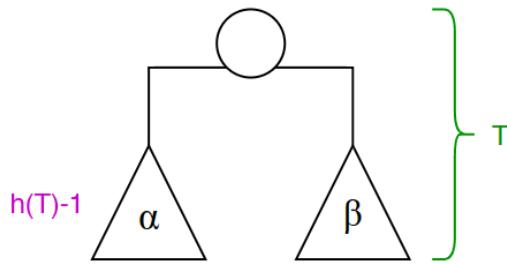
Cioè, il numero di nodi interni è maggiore o uguale a 2 elevato all'altezza nera, meno 1 .

Caso base ($h(T) = 0$)



$$h_N(T) = 1 \Rightarrow \#Int(T) = 1 \geq 2^1 - 1$$

Caso induttivo ($h(T) > 0$)



$$\#Int(T) = 1 + \#Int(\alpha) + \#Int(\beta) \geq 1 + (2^{h_N(\alpha)} - 1) + (2^{h_N(\beta)} - 1) = 2 \cdot 2^{h_N(\alpha)} - 1 = 2^{h_N(\alpha)+1} - 1 = 2^{h_N(T)} - 1$$

In particolare, $(2^{h_N(\alpha)} - 1)$ e $(2^{h_N(\beta)} - 1)$ sono uguali perché siamo in un red-black.

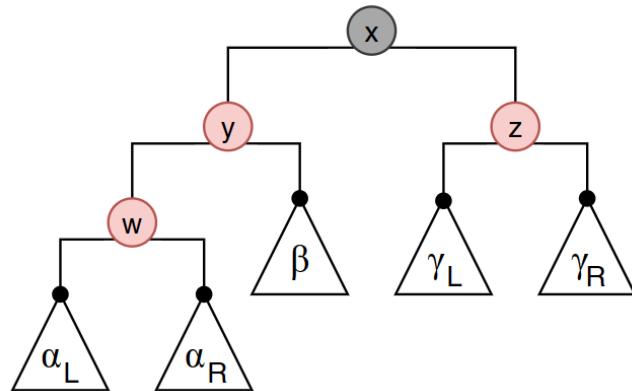
$$\#Int(T) \geq 2^{h_N(T)} - 1 \Rightarrow \#Int(T) + 1 \geq 2^{h_N(T)} \Rightarrow \log_2(\#Int(T)) \geq h_N(T) \Rightarrow 2 \log_2(\#Int(T)) \geq 2h_N(T) \geq h(T)$$

Con $2h_N(T)$ massimo limite di altezza di un percorso.

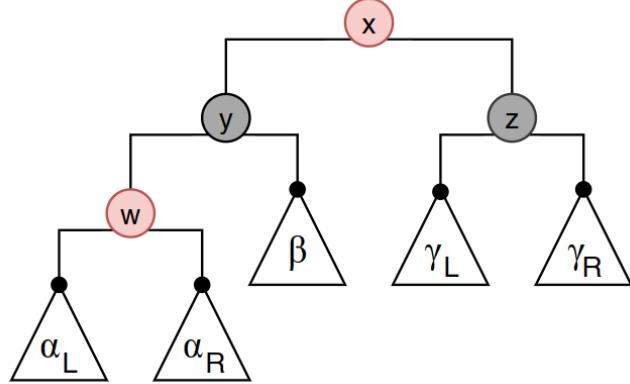
9.1 Problemi nell'inserimento

Quando si effettua l'inserimento all'interno di un albero red-black possono succedere diversi casi. Molti di questi si risolvono facendo un'operazione che ci permette di arrivare a un altro caso, e così via fino a quando non si arriva a quello che porta direttamente alla correzione del problema.

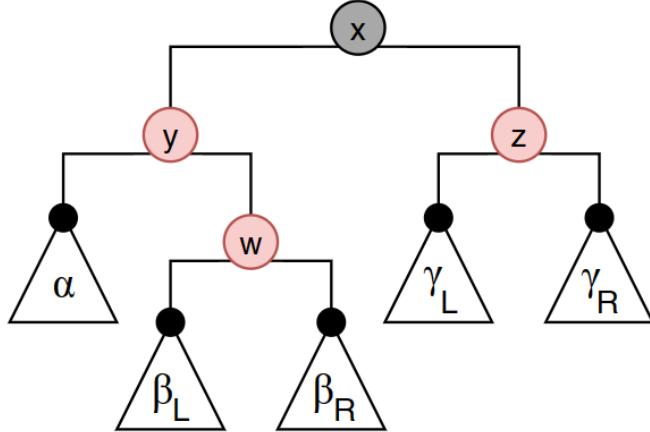
9.1.1 Caso 1



Vediamo che il nodo rosso y ha figlio w anch'esso rosso. Questo viola la proprietà 3.c degli alberi red-black. Dunque y è proprio il nodo che presenta il problema. Per risolverlo facciamo una semplice ricolorazione, cercando di portare così il lavoro più in alto. Avremo:

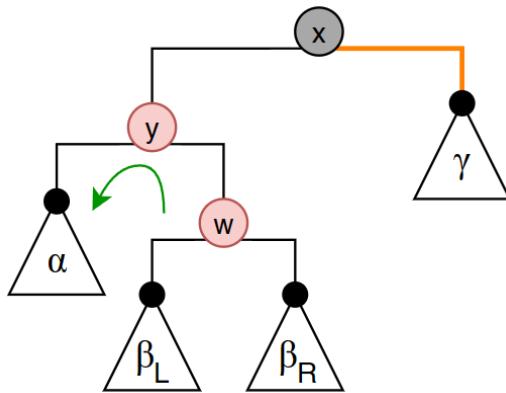


9.1.2 Caso 1.b

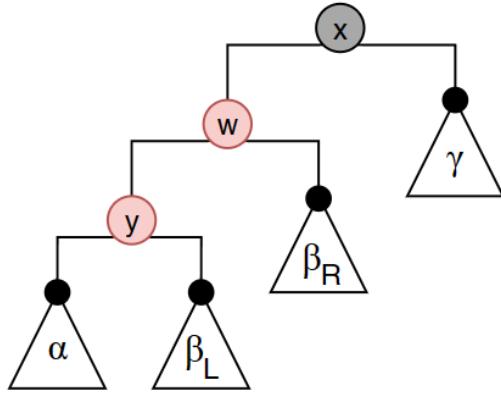


In questo caso abbiamo il nodo rosso a destra. Si risolve come nel caso 1.

9.1.3 Caso 2



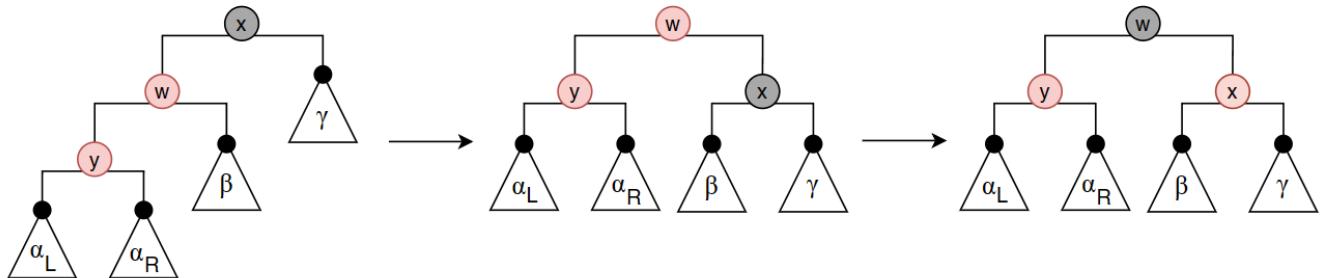
Nella zona arancione, c'è dello spazio che possiamo occupare aggiungendo un nodo rosso. Facciamo quindi una rotazione da destra verso sinistra. Non stiamo facendo altro che spostare il problema nella zona esterna così da poter effettuare un'ulteriore rotazione (caso 3). Otteniamo:



Notiamo che la lunghezza nera è rimasta invariata.

9.1.4 Caso 3

Bisogna fare una rotazione da sinistra a destra. Per correggere definitivamente il problema facciamo infine una ricolorazione. Quindi:



Vediamo ora gli algoritmi:

```

Function InsertRB(x,d)
  if x = NILRB then //equivale a x = ⊥
    x ← BuildNodeRB(d)
  else
    if x.dat > d then
      x.sx ← InsertRB(x.sx,d)
      x ← LInsBalance(x) //Left Insert Balance
    else if x.dat < d then
      x.dx ← InsertRB(x.dx,d)
      x ← RInsBalance(x)
  return x

```

```

Function LInsBalance(x)
  if x.sx ≠ NILRB then
    Switch LInsViolation(x) do
      case 1 do x ← LInsBalanceRB1(x)
      case 2 do x ← LInsBalanceRB2(x)
      case 3 do x ← LInsBalanceRB3(x)
    return x
  
```

dobbiamo scriverle noi

```

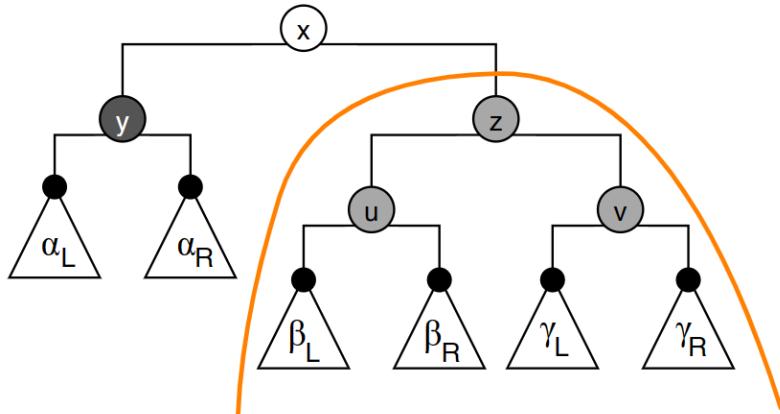
Function LInsViolation(x)
  (v,l,r) ← (0,x.sx,x,dx)
  if l.cl = R then
    if r.cl = R then
      if l.sx.cl = R OR l.dx.cl = R then
        v ← 1
      else
        if l.dx.cl = R then
          v ← 2
        else if l.sx.cl = R then
          v ← 3
  return v
  
```

9.2 Problemi nella cancellazione

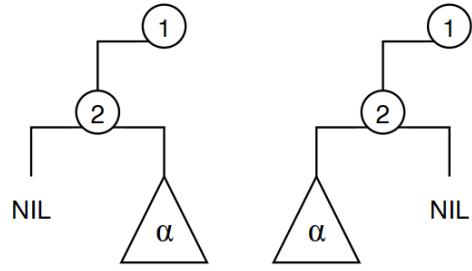
Anche in questo caso si presentano diversi casi.

9.2.1 Caso 1

Ci troviamo con questo albero:

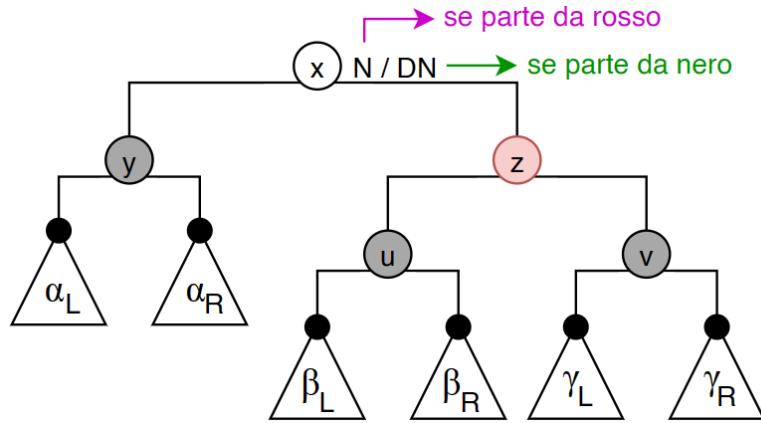


La radice può essere o rossa o nera. Andiamo a cancellare nel sottoalbero a sinistra (quindi nell'albero in y). Il sottoalbero destro (quindi quello in arancione), dato che non è stato toccato, rimanere red-black. Possiamo notare come il nodo y sia più scuro degli altri. Il suo colore, infatti, è doppio nero. Perché questa cosa? Vediamo un esempio per capire meglio:



In entrambi i casi vogliamo cancellare il nodo 2. Se 2 è rosso, allora non ci sono problemi perché 1 e α sono neri; se 2 è nero allora portiamo il "nero" in 1. Se 1 era rosso, ora diventa nero; se era nero, diventa doppio nero. Ecco perché il nodo y , dopo la cancellazione, assume quel colore.

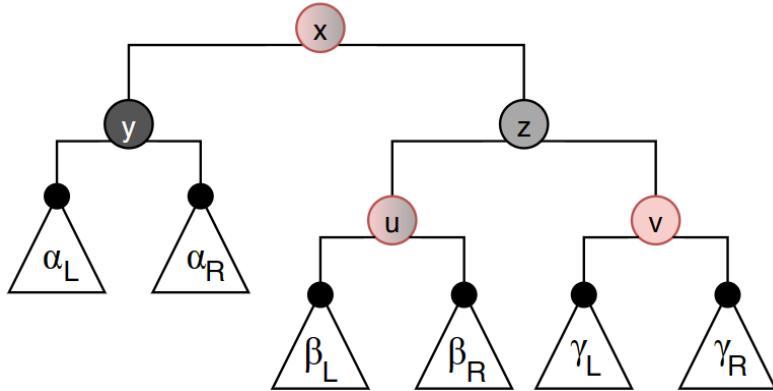
L'obiettivo adesso è quello di portare il "doppio nero" più in alto. Otteniamo:



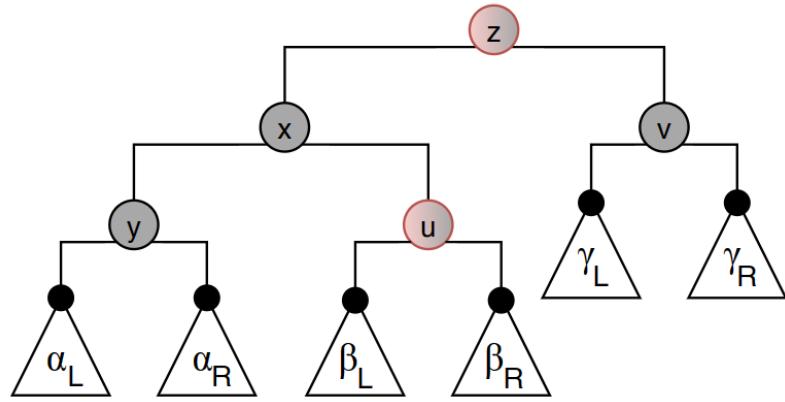
Può quindi succedere che il doppio nero arriva alla radice. Per sistemarlo, facciamo una semplice ricolorazione della radice in nero.

9.2.2 Caso 2

In questo caso abbiamo una situazione analoga al caso 1 ma con colori diversi.

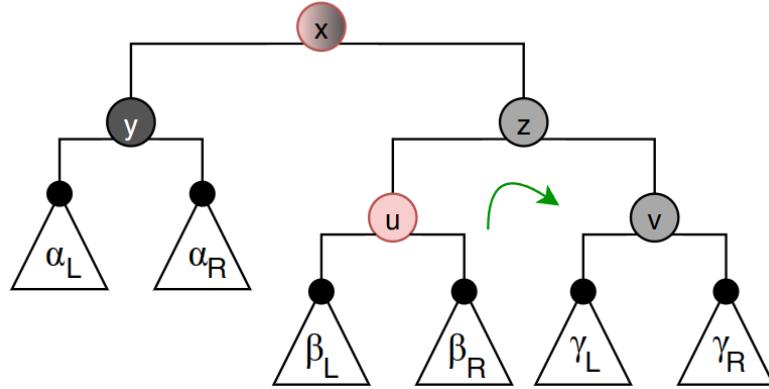


Il doppio nero è sempre in y . I nodi x e u possono essere o rossi o neri indifferentemente. Per risolvere andiamo ad effettuare una semplice rotazione da destra verso sinistra e un successivo controllo dei colori.

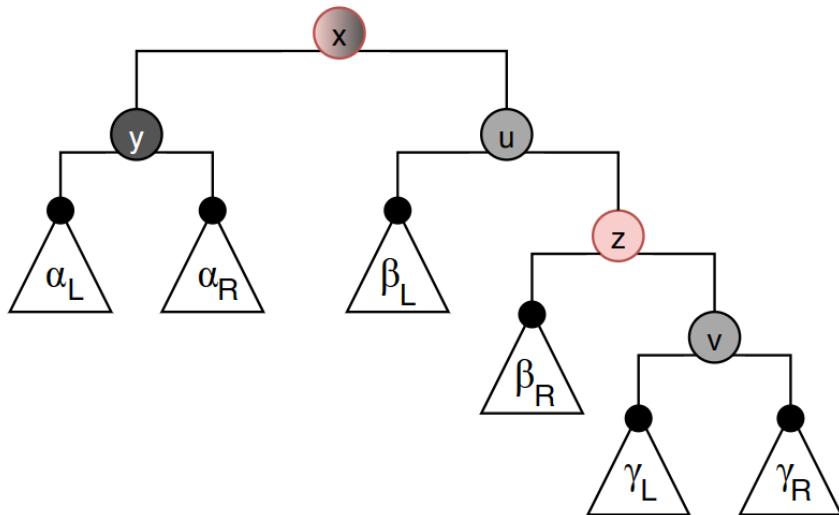


I nodi z e u possono essere o rossi o neri.

9.2.3 Caso 3

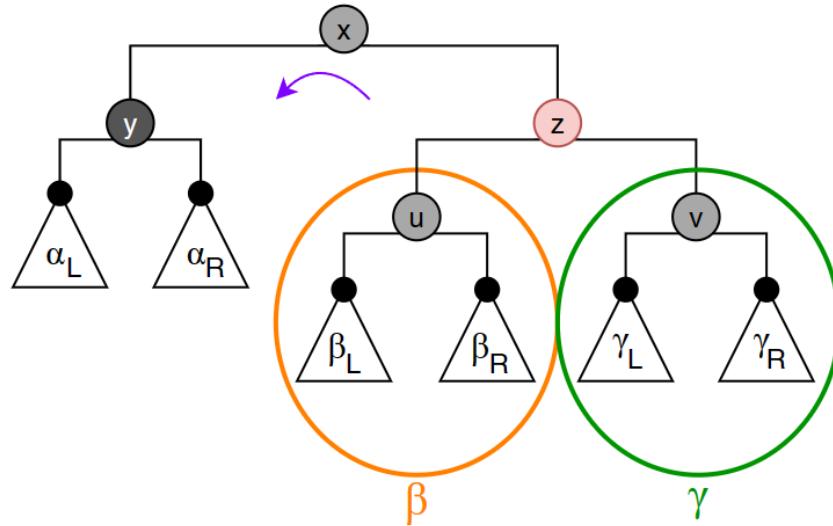


In questo caso facciamo prima una rotazione da sinistra verso destra in z .

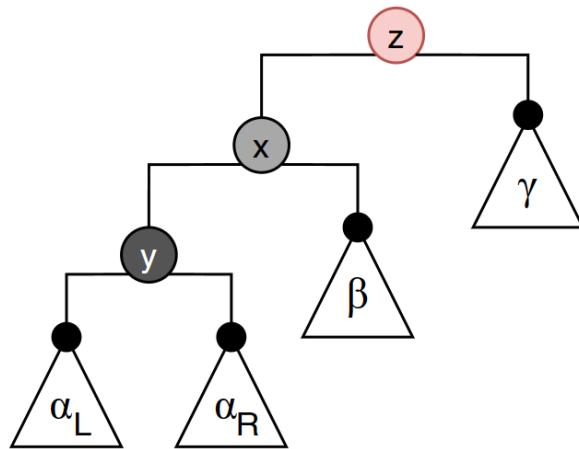


A questo punto ci basta provare ad arrivare al caso 2 e risolvere il problema.

9.2.4 Caso 4



In questo caso facciamo una rotazione da destra verso sinistra nella radice. Otteniamo:



Siamo così arrivati al caso 2 ma con radice rossa.

Vediamo ora gli algoritmi:

```

Function DeleteRB(x,d)
  if x ≠ NILRB then
    if x.dat > d then
      x.sx ← DeleteRB(x.sx,d)
      x ← LDelBalanceRB(x)
    else if x.dat < d then
      x.dx ← DeleteRB(x.dx,d)
      x ← RDelBalanceRB(x)
    else
      x ← DeleteNodeRB(x)
  return x

```

```
Function DeleteNodeRB(x)
    if x.sx = NILRB then
        x ← SkipRightRB(x)
    else if x.dx = NILRB then
        x ← SkipLeftRB(x)
    else
        x.dat ← Get&DeleteMinRB(x.dx, x)
        x ← RDelBalanceRB(x)
    return x
```

```
Function Get&DeleteMinRB(x, p)
    if x.sx = NILRB then
        d ← x.dat
        y ← SkipRightRB(x)
    else
        d ← Get&DeleteMinRB(x.sx, x)
        y ← LDelBalanceRB(x)
    SwapChild(p, x, y)
    return d
```

```
Function PropagateBlack(x)
    x.cl ← if x.cl = R then N else DN
```

```
Function SkipLeftRB(x)
    if x.cl = N then PropagateBlack(x.sx)
    return SkipLeft(x)
```

```
Function LDelBalanceRB(x)
    if x.sx ≠ NILRB ∧ x.dx ≠ NILRB then
        Switch LDelViolation(x) do
            case 1 do x ← LDelBalanceRB(1)
            case 2 do x ← LDelBalanceRB(2)
            case 3 do x ← LDelBalanceRB(3)
            case 4 do x ← LDelBalanceRB(4)
    return x
```

```
Function LDelViolationRB(x)
  (v,l,r) ← (0,x.sx,x.dx)
  if x.cl = DN then
    if v.cl = R then
      v ← 4
    else
      if r.dx.cl = R then
        v ← 2
      else if r.sx.cl = R then
        v ← 3
      else
        v ← 1
  return v
```

10 Passaggio dal ricorsivo all'iterativo

Prendiamo l'algoritmo ricorsivo di ricerca per gli alberi binari di ricerca.

```
Function Search(x,d)
    if x = ⊥ then
        return ⊥ // 1° caso base
    else
        if x.dat > d then
            return Search(x.sx,d)
        else if x.dat < d then
            return Search(x.dx,d)
        else
            return ⊤ // 2° caso base
```

Vediamo come passare alla versione iterativa. Questo algoritmo è tale-recursivo e, per questo, il dato non subisce cambiamenti dopo le chiamate ricorsive. Quindi, al ritorno delle chiamate, andiamo semplicemente a restituire il valore. Il processo è: eseguire il codice (tranne le chiamate ricorsive) in un ciclo while; quando non si arriva a un caso base, simuliamo le chiamate ricorsive cambiando i valori. È molto importante ricordare che questo tipo di ragionamento lo possiamo effettuare solo perché l'algoritmo è tale-recursivo.

```
Function ItrSearch(x,d)
    While (true) do
        if x = ⊥ then
            return ⊥
        else
            if x.dat > d then
                x ← x.sx
            else if x.dat < d then
                x ← x.dx
            else
                return ⊤
```

L'esempio considerato però è un caso troppo specifico. Infatti non è molto flessibile se viene fatta anche solo una minima modifica. Tipo:

```
Function F(x,d)
    if x = ⊥ then
        return 0
    else
        if x.dat > d then
            return (2*F(x.sx,d))
        else if x.dat < d then
            return (3*F(x.dx,d))
        else
            return 1
```

In questo caso, l'algoritmo è sempre tale-recursivo.

Per convertirlo in iterativo, dobbiamo dividere la simulazione della chiamata e quella del ritorno. Prima questa divisione non era necessaria perché restituivamo o \top o \perp . Adesso la chiamata viene moltiplicata per 2 o per 3, quindi il valore di ritorno cambierà ogni volta a seconda di dove si trova il dato.

A causa di questo cambiamento, abbiamo bisogno di uno stack per tenere traccia di x , dato che dobbiamo sapere da dove sta tornando.

Importante notare che se le chiamate fosse state moltiplicate per lo stesso numero (quindi, ad esempio, entrambe per 2), non sarebbe nato il problema perché a prescindere di dove si trovi il dato, si moltiplica sempre per lo stesso valore.

```

Function ItrF( $x, d$ )
  Stack  $S_x$  //  $S_x$  sta per Stack di  $x$ 
   $call \leftarrow \top$ 

  While ( $call \vee \#S_x \neq \emptyset$ ) do
    if  $call$  then
      if  $x = \perp$  then
         $ret \leftarrow 0$  // valore di ritorno
         $call = \perp$ 
      else
        if  $x.dat > d$  then
           $S_x \leftarrow Push(S_x, x)$ 
           $x \leftarrow x.sx$ 
        else if  $x.dat < d$  then
           $S_x \leftarrow Push(S_x, x)$ 
           $x \leftarrow x.dx$ 
        else
           $ret \leftarrow 1$ 
           $call \leftarrow \perp$ 
      else
         $x \leftarrow Top(S_x)$ 

        if  $x.dat > d$  then
           $ret \leftarrow 2*ret$ 
        else
           $ret \leftarrow 3*ret$ 

     $S_x \leftarrow Pop(S_x)$ 
  return  $ret$ 

```

Questo era un esempio specifico, per capire il senso generale. Ora vediamo la versione astratta:

```
Function RecF(x,i,j)
    a ←  $F_{ini}(i, j)$ 
    if  $x = \perp$  then
        m ←  $F_{\perp}(a)$ 
    else
        if  $C_L(x)$  then
            ( $i'$ ,  $j'$ ,  $h$ ) ←  $F_{pre}^L(x, a)$ 
            z ← RecF( $x.sx, i', j'$ )
            m ←  $F_{post}^L(x, h, z)$ 
        else if  $C_R(x)$  then
            ( $i'$ ,  $j'$ ,  $h$ ) ←  $F_{pre}^R(x, a)$ 
            z ← RecF( $x.dx, i', j'$ )
            m ←  $F_{post}^R(x, h, z)$ 
        else
            m ←  $F_{loc}(x, a)$ 
    return  $F_{fin}(m)$ 
```

Vogliamo convertire $RecF$ in iterativo. La prima cosa da fare è analizzare tutte le variabili e vedere se hanno bisogno di uno Stack:

x = viene usato per fare dei test iniziali e per andare a destra o a sinistra. Viene usato però anche dopo le chiamate ricorsive. Per questo motivo ha bisogno di uno Stack. Quindi *se una variabile viene creata prima di una chiamata ricorsiva e viene usata dopo la stessa, necessita dello Stack*;

i = non necessita dello Stack perché viene prodotto e usato prima della chiamata ricorsiva;

j = non necessita dello Stack perché viene prodotto e usato prima della chiamata ricorsiva;

a = è prodotta prima della chiamata ricorsiva. Usata per fare lavoro in pre-ordine, come parametro della chiamata ricorsiva e per fare controlli. Dunque non necessita dello Stack;

m = non necessita dello Stack;

i', j' = non hanno bisogno dello Stack;

h = ha bisogno dello Stack. In particolare andiamo a distinguere lo Stack destro e quello sinistro per semplificare il lavoro;

z = viene prodotto dalla chiamata ricorsiva, quindi non ha bisogno di uno Stack.

L'algoritmo iterativo sarà:

```

Function ItrF(x,i,j)
  Stack Sx,Shl,Shr
  call  $\leftarrow \top$ 

  While (call  $\vee \neg \text{isEmpty}(Sx)$ )
    if call then
      a  $\leftarrow F_{ini}(i, j)$ 
      if x =  $\perp$  then
        ret  $\leftarrow F_{fin}(F_{\perp}(a))$   $\leftarrow$  m  $\leftarrow F_{\perp}(a)$ 
        call  $\leftarrow \perp$ 
      else
        if  $C_L(x, a)$  then
          (i',j',h)  $\leftarrow F_{pre}^L(x, a)$ 
          Sx  $\leftarrow \text{Push}(Sx, x)$ 
          Shl  $\leftarrow \text{Push}(Shl, h)$ 
          (x,i,j)  $\leftarrow (x.\text{sx}, i', j')$ 
        else if  $C_R(x, a)$  then
          (i',j',h)  $\leftarrow F_{pre}^R(x, a)$ 
          Sx  $\leftarrow \text{Push}(Sx, x)$ 
          Shr  $\leftarrow \text{Push}(Shr, h)$ 
          (x,i,j)  $\leftarrow (x.\text{dx}, i', j')$ 
        else
          ret  $\leftarrow F_{fin}(F_{loc}(x, a))$ 
          call  $\leftarrow \perp$ 
    else
      x  $\leftarrow \text{Top}(Sx)$ 
      z  $\leftarrow \text{ret}$ 

      if  $C_L(x)$  then
        (Shl,h)  $\leftarrow \text{Top\&Pop}(Shl)$ 
        ret  $\leftarrow F_{fin}(F_{post}^L(x, h, z))$ 
      else
        (Shr,h)  $\leftarrow \text{Top\&Pop}(Shr)$ 
        ret  $\leftarrow F_{fin}(F_{post}^R(x, h, z))$ 

      Sx  $\leftarrow \text{Pop}(Sx)$ 
  return ret

```

Nell'else esterno, l'istruzione $(Shr, h) \leftarrow \text{Top\&Pop}(Shr)$ è necessaria perché dobbiamo ripristinare il valore di h prendendo quello quello dallo Stack. Se non lo facessimo, h avrebbe un valore casuale.

Facciamo una piccola modifica al codice dell'algoritmo ricorsivo.

```
Function RecF(x,i,j)
    a ←  $F_{ini}(i, j)$ 
    if  $x = \perp$  then
        m ←  $F_{\perp}(a)$ 
    else
        if  $C_L(x, a)$  then
            ( $i'$ ,  $j'$ , h) ←  $F_{pre}^L(x, a)$ 
            z ← RecF( $x.sx, i', j'$ )
            m ←  $F_{post}^L(x, h, z)$ 
        else if  $C_R(x, a)$  then
            ( $i'$ ,  $j'$ , h) ←  $F_{pre}^R(x, a)$ 
            z ← RecF( $x.dx, i', j'$ )
            m ←  $F_{post}^R(x, h, z)$ 
        else
            m ←  $F_{loc}(x, a)$ 
    return  $F_{fin}(m)$ 
```

La versione iterativa sarebbe:

```

Function ItrF(x,i,j)
  Stack Sx,Shl,Shr
  call  $\leftarrow \perp$ 

  While (call  $\vee \neg \text{isEmpty}(\text{Sx})$ )
    if call then
      a  $\leftarrow F_{\text{ini}}(i, j)$ 
      if x  $= \perp$  then
        ret  $\leftarrow F_{\text{fin}}(F_{\perp}(a))$ 
        call  $\leftarrow \perp$ 
      else
        if  $C_L(x, a)$  then
          (i',j',h)  $\leftarrow F_{\text{pre}}^L(x, a)$ 
          Sx  $\leftarrow \text{Push}(\text{Sx}, x)$ 
          Shl  $\leftarrow \text{Push}(\text{Shl}, h)$ 
          (x,i,j)  $\leftarrow (\text{x.sx}, i', j')$ 
        else if  $C_R(x, a)$  then
          (i',j',h)  $\leftarrow F_{\text{pre}}^R(x, a)$ 
          Sx  $\leftarrow \text{Push}(\text{Sx}, x)$ 
          Shr  $\leftarrow \text{Push}(\text{Shr}, h)$ 
          (x,i,j)  $\leftarrow (\text{x.dx}, i', j')$ 
        else
          ret  $\leftarrow F_{\text{fin}}(F_{\text{loc}}(x, a))$ 
          call  $\leftarrow \perp$ 
    else
      x  $\leftarrow \text{Top}(\text{Sx})$ 
      a  $\leftarrow \text{Top}(\text{Sa})$  // Sa andrebbe dichiarato sopra
      z  $\leftarrow \text{ret}$ 

      if  $C_L(x)$  then
        (Shl,h)  $\leftarrow \text{Top\&Pop}(\text{Shl})$ 
        ret  $\leftarrow F_{\text{fin}}(F_{\text{post}}^L(x, h, z))$ 
      else
        (Shr,h)  $\leftarrow \text{Top\&Pop}(\text{Shr})$ 
        ret  $\leftarrow F_{\text{fin}}(F_{\text{post}}^R(x, h, z))$ 

      Sx  $\leftarrow \text{Pop}(\text{Sx})$ 
      Sa  $\leftarrow \text{Pop}(\text{Sa})$ 
  return ret

```

Abbiamo detto che se una variabile non viene usata dopo le chiamate ricorsive, non ha bisogno di uno Stack. Se però, questa stessa variabile, è cruciale per capire se stiamo andando a destra o a sinistra, allora dobbiamo comunque memorizzarla su uno Stack. Bisogna comunque porsi una domanda: questa variabile, è ricalcolabile (in modo semplice e non dispendioso)? Se la risposta è sì, allora lo Stack rimane non necessario; lo è altrimenti. Per capire, se avessimo una cosa del tipo *a ← rand()* allora lo Stack è d'obbligo perché ogni volta *a* avrebbe un valore diverso.

Vediamo ora l'ultima cosa che ci interessa per la conversione, cioè il caso in cui nell'algoritmo sono presenti più chiamate ricorsive. Vediamo una cosa simile al Merge Sort.

```

Function MergeSort(A,i,j)
  if i < j then
    m ← ⌊ $\frac{i+j}{2}$ ⌋
    MergeSort(A,i,m)
    MergeSort(A,m+1,j)
    Merge(A,i,m,j)

```

La sua versione iterativa sarà:

```

Function ItrMergeSort(A,i,j)
  Stack Si,Sj
  call ← ⊥

  While (call ∨ ¬isEmpty(Si)) do
    if call then
      if i < j then
        m ← ⌊ $\frac{i+j}{2}$ ⌋
        (Si,Sj) ← (Push(Si,i),Push(Sj,j))
        j ← m
      else
        call ← ⊥
    else
      (i,j) ← (Top(Si),Top(Sj))
      m ← ⌊ $\frac{i+j}{2}$ ⌋
      if last = i then
        i ← m+1
        call ← ⊤
      else
        Merge(A,i,j,m)
        (Si,Sj) ← (Pop(Si),Pop(Sj))
        last ← i

```

Vediamo ora un altro algoritmo che presenta sempre due chiamate ricorsive una dopo l'altra:

```

Function DFS(x,F,a)
  if x ≠ ⊥ then
    a ← F(x.dat,a)
    a ← DFS(x.sx,F,a)
    a ← DFS(x.dx,F,a)
    return a
  else
    a ← G(a) // funzione generica
    return a

```

Per simularlo dovremmo fare la stessa cosa fatta prima. Il problema sta nel fatto che non sappiamo sempre da dove torniamo dopo le chiamate ricorsive. In particolare quando torniamo dalle foglie perché non sappiamo se stiamo tornando da destra o da sinistra in quando sono entrambe \perp . Per risolvere, mettiamo come condizione che a sinistra ci andiamo sempre, mentre a destra ci andiamo solo quando non è \perp .

Per la versione iterativa di questo algoritmo abbiamo bisogno dello Stack solo per x e non anche per a perché non viene mai usata dopo le chiamate ricorsive.

```
Function ItrDFS(x,F,a)
  Stack Sx
  call ← ⊤

  While (call ∨ ¬isEmpty(Sx)) do
    if call then
      if x ≠ ⊥ then
        a ← F(x.dat,a)
        Sx ← Push(Sx,x)
        x ← x.sx
      else
        a ← G(a)
        ret ← a
        (last,call) ← (x,⊥)
    else
      x ← Top(Sx)
      if x.sx = last then
        if x.dx ≠ ⊥ then
          a ← ret
          x ← x.dx
          call ← ⊤
        else
          a ← G(a)
          ret ← a
          last ← x
      else
        last ← x
      Sx ← Pop(Sx)
  return ret
```

In base alle casistiche può anche essere messo qui

dato che stiamo al ritorno delle chiamate
call è \perp e dobbiamo cambiarlo

Nell'ultimo else avremmo dovuto scrivere anche $a \leftarrow ret$ e $ret \leftarrow a$ ma, come si può ben notare, è inutile; per quanto riguarda $call$ è già \perp , quindi non ci serve lavorare su di lei. È per questo che eseguiamo solo l'istruzione $last \leftarrow x$.

11 Grafi

Definizione: un grafo $G = \langle V, E \rangle$ consiste in:

- Un insieme di vertici V (o nodi);
- Un insieme E di coppie di vertici, detti archi o spigoli: ogni arco connette due vertici. Infatti E è una relazione binaaria su V , ossia $E \subseteq V \times V$.

Supponiamo di avere:

$$R = \{(a, b) \in [1, 6] \times [1, 6] / (a \text{ divide } b)\}$$

Le coppie possibili sono quindi:

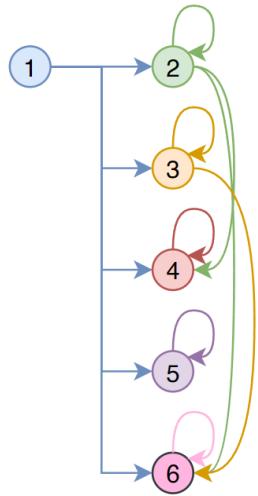
	1	2	3	4	5	6
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
2		(2,2)			(2,4)	
3			(3,3)			(3,6)
4				(4,4)		
5					(5,5)	
6						(6,6)

Tabellare i dati in questo modo non aggiunge niente a quello in cui quando c'è una relazione viene messo 1 e quando non c'è viene messo 0. La tabella diventerebbe:

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	0	1	0	1	0	0
3	0	0	1	0	0	1
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	1

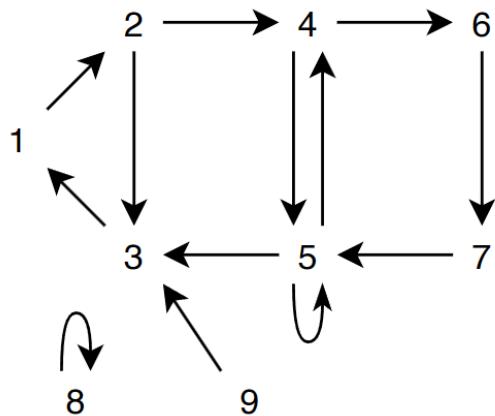
Questa prende il nome di rappresentazione tramite matrice di adiacenza.

Graficamente sarebbe:



Riallocare un grafico (aggiungere vertici) è quadratico nel numero di vertici; aggiungere solo archi è lineare.

Prendiamo un ulteriore esempio:



Costruiamo la sua matrice di adiacenza:

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0
5	0	0	1	1	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0
9	0	0	1	0	0	0	0	0	0

Notiamo che si tratta di una matrice non simmetrica, quindi $\text{Valore}^{i,j} \neq \text{Valore}^{j,i}$.

Le operazioni sugli archi sono a tempo costante, dato che c'è accesso diretto; mentre le operazioni sui vertici sono a tempo quadratico ($\Theta(|v|^2)$), perché ogni modifica comporta la creazione di una nuova matrice con dimensioni modificate e il trasferimento dei dati dalla vecchia matrice alla nuova.

Questo in particolare prende il nome di **grafo orientato**.

Un grafo $G = (V, E)$ è un grafo orientato (grafo diretto o digrafo) se E è formato da coppie ordinate di nodi

Possiamo così introdurre la stella uscente da un nodo w_i , indicata con $\text{adj}(w_i)$, che è l'insieme $\{w_j \in V | (w_i, w_j) \in E\}$. Esiste ovviamente anche la stella entrante di un nodo w_i , indicata con $\text{inc}(w_i)$, che è l'insieme $\{w_j \in V | (w_j, w_i) \in E\}$. Il costo del calcolo degli adiacenti è lineare in n , dato che andiamo a controllare tutta la riga.

Ogni grafo presenta un **grado**.

Il grado di un grafo è il massimo numero di archi uscenti da un nodo

Nell'esempio precedente, il grado è 3.

È possibile riscontrare grafi senza archi. Questi prendono il nome di **grafi isolati** o **grafi vuoti**. Un esempio è il grafo $G = (\{A, B, C\}, \{\})$. Dato che l'insieme E è vuoto, questo vuol dire che i vertici A, B, C non sono connessi tra loro. Anche nei grafi vuoti il calcolo di $\text{adj}(w)$ è lineare.

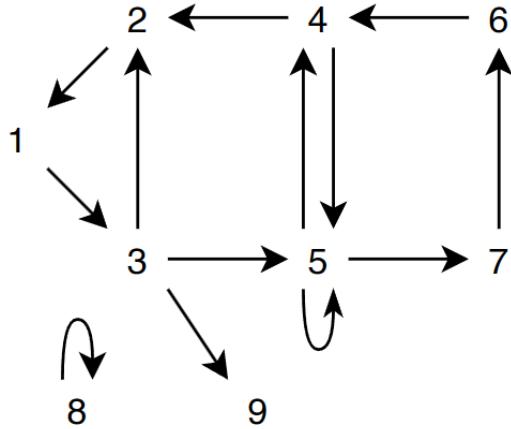
Un grafo G si dice denso se $|E| = \Theta(|V|^2)$

quindi, un grafo è considerato "denso" quando il numero di archi è vicino al numero massimo possibile di archi. Nella matrice di adiacenza, il grafo è denso quando c'è una maggioranza di 1. Non a caso, infatti, questa è la miglior rappresentazione per questo tipo di grafo.

Un grafo G si dice sparso se $|E| = O(|V|)$

quindi ci si riferisce a "grafi sparsi" quando il numero di archi è significativamente inferiore al numero massimo possibile di archi. In questi casi, l'uso di una rappresentazione standard (matrice di adiacenza o lista di adiacenza, che vedremo dopo) può comportare uno spreco di memoria, poiché molte delle posizioni nella matrice o nella lista saranno vuote.

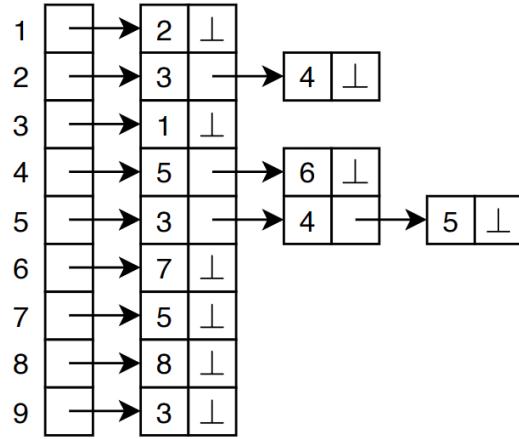
Per affrontare questo problema, si può considerare l'uso del "complemento del grafo". Il complemento di un grafo G , spesso indicato come \overline{G} , è un grafo che ha gli stessi vertici di G , ma gli archi che non sono presenti in G sono presenti in \overline{G} e viceversa. Con il complemento, il grafo completo è sparso. Il grafo trasposto dell'esempio precedente sarebbe:



La sua matrice di adiacenza non sarà altro che la trasposta di quella precedente, cioè con righe e colonne invertite.

La matrice di adiacenza ci permette di avere un facile accesso ai nodi adiacenti tramite le righe e ai nodi incidenti tramite le colonne. In particolare, i nodi adiacenti ($adj(w)$) a un dato nodo possono essere trovati esaminando gli archi uscenti dalla corrispondente riga della matrice, e i nodi incidenti ($inc(w)$) a un dato nodo possono essere trovati esaminando gli archi entranti dalla corrispondente colonna della matrice.

Come abbiamo già accennato prima, esiste un altro tipo di rappresentazione: la **lista di adiacenza**. In questo caso andiamo a considerare le righe della matrice come lista e non come array. Per ogni nodo viene memorizzata la lista dei nodi ad esso adiacenti.



La complessità nelle liste di adiacenza è pari a $\Theta(|V| + |E|)$. Perché? Ogni vertice nel grafo è rappresentato da un nodo nella lista di adiacenza. Pertanto, la componente $\Theta(|V|)$ rappresenta la complessità associata alla memorizzazione dei vertici. Ogni arco nel grafo è rappresentato dalla presenza di un elemento nella lista di adiacenza del vertice di partenza. Poiché ogni arco è rappresentato una volta, la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(|E|)$. La complessità totale di spazio per le liste di adiacenza è quindi la somma di queste due componenti.

Questa rappresentazione è efficiente nei casi in cui il grafo è sparsamente connesso, poiché non è necessario memorizzare ogni possibile arco, ma solo quelli effettivamente presenti. La complessità $\Theta(|V| + |E|)$ riflette il fatto che la quantità di spazio occupato dipende sia dal numero di vertici che dal numero di archi presenti nel grafo.

L'inserimento e la cancellazione di archi tendono ad essere costanti (se per esempio si effettua un inserimento in testa), mentre l'inserimento e la cancellazione di nodi tendono ad essere lineari perché bisogna andare a riallocare tutto il vettore di vertici. Supponendo si voglia fare la ricerca di un determinante arco, questo è lineare al grado del grafo. Se il grado è molto basso, allora è quasi costante.

Con questa rappresentazione si va a perdere l'accesso in colonna, quindi non abbiamo l'accesso agli incidenti. Per ovviare a questa perdita, ci basta creare il grafo trasposto. Quindi, ad esempio, se abbiamo $1 \rightarrow 2$, scriveremo $2 \rightarrow 1$. In generale quindi otterremo una rappresentazione non per le righe ma per le colonne. Se le unissimo, otterremo un'unica lista per entrambi.

11.1 Percorso, Ciclo

Cos'è un percorso in un grafo? Diamo la definizione:

$\pi \in Pth(G)$, è una sequenza di vertici, ovvero $\pi \in V^*$ tale che $\forall i \in [0, |\pi| - 1] \circ ((\pi)_i, (\pi)_{i+1}) \in E$

Analizziamola nel dettaglio:

- $\pi \in Pth(G)$: $Pth(G)$ indica, in generale, l'insieme di tutti i percorsi nel grafo G ;
- $\pi \in V^*$: questo significa che π è una sequenza di vertici, con V^* l'insieme di tutte le sequenze possibili di vertici (compreso l'insieme vuoto);
- $\forall i \in [0, |\pi| - 1]$: Questa parte sta affermando che la seguente condizione deve essere vera per ogni i compreso tra 0 e $|\pi| - 1$ (il numero totale di vertici meno uno);
- $((\pi)_i, (\pi)_{i+1}) \in E$: Questa parte specifica che ogni coppia di vertici consecutiva nel percorso π deve essere un lato del grafo. Quindi $(\pi)_i, (\pi)_{i+1}$ sono vertici consecutivi nel percorso π , e la coppia $(\pi)_i, (\pi)_{i+1}$ deve appartenere all'insieme E .

In sintesi, stiamo dicendo che un percorso π in un grafo G è una sequenza di vertici dove ogni coppia consecutiva di vertici nel percorso forma un lato del grafo. Non è detto che i percorsi siano unici.

Per continuare il discorso sui percorsi, abbiamo bisogno della definizione di ciclo:

$\pi \in Cye(G)$, $\pi \in Pth(G)$ tale che $first(\pi) = (\pi)_0 = last(\pi) = (\pi)_{|\pi|-1}$

Analizziamola del dettaglio:

- $\pi \in Cye(G)$: questa parte indica che π è un ciclo nel grafo G ;
- $\pi \in Pth(G)$: questo afferma che il ciclo π è anche un percorso in G . Ogni ciclo è un percorso, ma non ogni percorso è un ciclo. Un percorso può essere aperto, mentre un ciclo è per definizione chiuso;
- $first(\pi) = (\pi)_0$: Questa parte afferma che il primo vertice del ciclo è lo stesso del vertice in posizione 0 nella sequenza di vertici π ;
- $last(\pi) = (\pi)_{|\pi|-1}$: Questa parte afferma che l'ultimo vertice del ciclo è lo stesso del vertice in posizione $(|\pi| - 1)$, cioè l'ultima posizione della sequenza di vertici π .

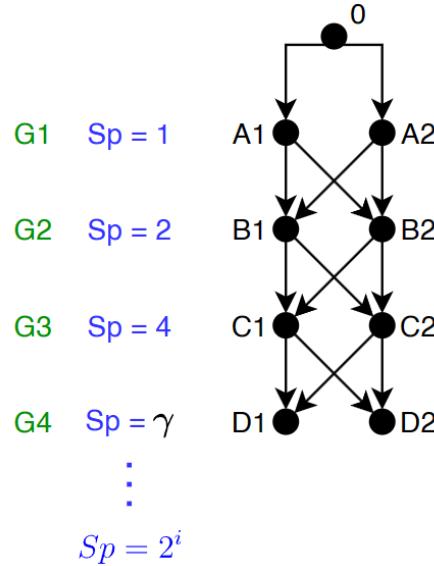
In sintesi, la definizione afferma che un ciclo π è un percorso chiuso in cui il vertice di partenza e il vertice di arrivo coincidono, e il primo e l'ultimo vertice nella sequenza π sono gli stessi.

Possiamo così dare la definizione di percorso semplice:

$\pi \in SPth(G) \iff \pi \in Pth(G)$ e non contiene un ciclo, $\neg \exists i, j \in [0, |\pi|] \circ (\pi)_i = (\pi)_j$

In sintesi, la definizione afferma che un percorso π è un percorso semplice se è un percorso che attraversa vertici distinti e non contiene cicli (ripetizioni di vertici), ossia non ci sono indici distinti i e j tali che $\pi_i = \pi_j$.

Vediamo un esempio:



Numero di vertici: $|G_i| = |V_i| = 2i + 1$

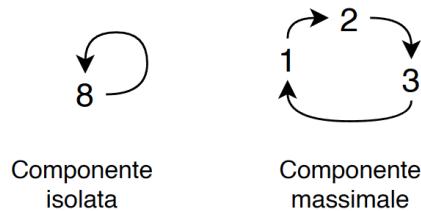
Numero di archi: $|E_i| = 2 + 4(i - 1)$

Numero Spath: $|SPth(G_i)| = |Pth(4_i)| = 2^i$

Lineare nella dimensione del grafo \rightarrow non polinomiale. Non bisogna esplorare tutti i percorsi ma attraversarlo una volta. Quindi bisogna capire il concetto di raggiungibilità.

11.2 Componenti

- **Componente isolata:** Una componente isolata è una sottografo in cui ciascun vertice è isolato, cioè non è connesso ad alcun altro vertice. In altre parole, non ci sono lati all'interno di una componente isolata. In un grafo, una componente isolata può consistere di uno o più vertici che non sono connessi ad alcun altro vertice del grafo principale;
- **Componente massimale:** Una componente massimale è una sottografo con la proprietà che, se viene aggiunto qualsiasi vertice o lato esterno alla componente, la sua connettività è interrotta. In altre parole, non è possibile estendere la componente massimale aggiungendo altri vertici o lati senza interrompere la sua connettività. È la componente più grande con questa proprietà.



11.3 Raggiungibilità

Definizione:

$$Reach(G) \subseteq V \times V : Reach(G) = \{(v, w) \in V \times V / \exists \pi \in Pth(G), first(\pi) = v \wedge last(\pi) = w\}$$

Quindi esiste un percorso che, prendendo un punto di partenza e un punto di arrivo, questa coppia appartiene ai percorsi esistenti. La "raggiungibilità" è una relazione con se stessa.

È sempre possibile modificare la definizione di Reach mettendo al posto di $Pth(G)$, $SPth(G)$. Questa modifica però deve essere tale che il numero di nodi nel percorso semplice non superi il numero totale di nodi del grafo.

Ora diamo altre definizioni:

$$E^0 = Id_E \text{ ovvero } = (v, v) / v \in V$$

In altre parole, E^0 contiene tutti gli archi che collegano un vertice a se stesso.

$$E^1 = E, \text{ archi semplici}$$

ovvero percorsi di lunghezza 1.

Più in generale:

$$E^i = \begin{cases} Id, & \text{se } i = 0 \\ E^{i-1} \circ E, & \text{altrimenti} \end{cases}$$

Possiamo calcolare fino a E^n (n è il numero di nodi) poiché tutto ciò che verrebbe dopo sarebbe un loop.

Si può osservare che:

$$\bigcup_{k \in [0, n)} E^k$$

Questa notazione rappresenta l'unione di tutti i percorsi di lunghezza k in un grafo G , per k che varia da 0 a $n-1$, dove n è il numero di nodi del grafo. La notazione E^k indica l'insieme di tutti i percorsi di lunghezza k nel grafo. L'unione di questi insiemi, quindi, rappresenta l'insieme di tutti i percorsi possibili nel grafo, indipendentemente dalla loro lunghezza, fino a una lunghezza massima di $n-1$.

È possibile fare un esempio con il prodotto di matrici. Infatti, applicandolo n volte otteniamo Reach. Prendiamo un esempio di matrice con 4 nodi:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Moltiplicando le matrici di E^1 abbiamo ottenuto tutti i percorsi di lunghezza 2, cioè E^2 . In questo caso, abbiamo tenuto solo 0 e 1 perché vogliamo semplicemente sapere se esiste o no un arco tra due vertici. Per E^3 faremo lo stesso:

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

11.4 Grafo orientato e non orientato

Un grafo *non orientato* $G = (V, E)$ si dice **connesso** se esiste un cammino tra ogni coppia di vertici in G .

Un grafo *orientato* $G = (V, E)$ si dice **fortemente connesso** se esiste un cammino (orientato) tra ogni coppia di vertici in G .

Andiamo nel formale e aggiungiamo altre definizioni:

- G non orientato è **connesso** se $\forall v, w \in V, (v, w) \in \text{Reach}$;
- G orientato è **fortemente connesso** se $\forall v, w \in V, (v, w) \in \text{Reach}$;
- G' è **sottografo** di G se $V' \subseteq V$ e $E' \subseteq E$;
- G' è un **sottografo indotto** di G se $V' \subseteq V$ e $E' \subseteq E \cap V' \times V'$;

11.5 Distanza in un grafo

Definizione:

$$\delta(v, w) = \min_{\pi \in Pth(G, v, w)} |\pi|$$

Con $|\pi| =$ numero di archi. Vale se il numero di nodi è pari a $|\pi| - 1$.

In altre parole, la **distanza** tra due vertici v e w in un grafo, è definita come il minimo numero di archi in un percorso tra v e w , se esiste almeno un percorso tra i due vertici. Ovviamente per il minimo numero di archi dobbiamo considerare solo i percorsi semplici, così da evitare loop.

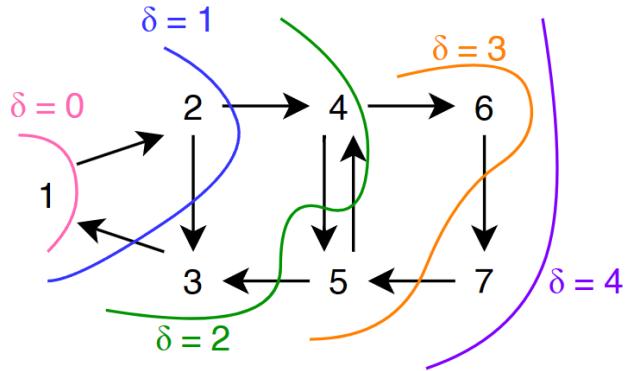
Vediamo la matrice di adiacenza per $\delta(v, w)$. Seguiamo l'esempio presentato prima:

	1	2	3	4	5	6	7	8	9	10
1	0	1	2	2	3	3	4	∞	∞	∞
2	2	0	1	1	2	2	3	∞	∞	∞
3	1	2	0	3	4	4	5	∞	∞	∞
4	3	4	2	0	1	1	2	∞	∞	∞
5							0			
6							0			
7							0			
8							0			
9							0			
10							0			

È da completare. Si mette il simbolo ∞ quando non è possibile raggiungerlo. Importante specificare che questa matrice è simmetrica per i grafi non orientati.

12 Visita in ampiezza

La visita o ricerca in ampiezza (breadth-first search o BFS) è uno degli algoritmi di base per la ricerca su grafi. Partendo da un certo vertice, esploro i contorni ampliando man mano i suoi orizzonti. Facciamo un esempio in cui ampliamo gli orizzonti di 1:



Così andiamo a calcolare tutti in modo lineare.

In particolare, partiamo da un nodo iniziale (nel nostro esempio 1) e andiamo ad esplorare prima tutti i nodi direttamente raggiungibili da esso (quindi quelli a distanza 1); poi tutti i nodi raggiungibili in due passi (quindi quelli a distanza 2) e così via.

Durante la ricerca in ampiezza (e in generale per le ricerche nei grafi) dobbiamo evitare di visitare un nodo più volte. Per questo assegnamo ad ogni nodo un colore:

- Bianco: il nodo non è stato ancora visitato;
- Grigio: il nodo è stato visitato ma potrebbe avere dei vicini non visitati;
- Nero: il nodo è stato visitato e anche tutti i suoi vicini.

Utilizziamo una coda per tenere traccia dei nodi che abbiamo già visitato.

```

Function BFS(G,v)
    (c,d,p)  $\leftarrow$  Init(G) // p = insieme dei predecessori
    (Q,c[v],d[v])  $\leftarrow$  (Singleton(v),Gr,0)

    repeat
        (Q,v)  $\leftarrow$  Head&Dequeue(Q)
        for each w  $\in$  Adj[v] do
            if c[w] = Bn then
                (Q,c[w],d[w],p[w])  $\leftarrow$  (Enqueue(Q,w),Gr,d[v]+1,v)
            c[v]  $\leftarrow$  Nr
        until isEmpty(Q)
    return (c,d,p)

```

```

Function Init(G)
    for each v  $\in$  V do
        (c[v],d[v],p[v])  $\leftarrow$  (Bn, $\infty$ , $\perp$ )

```

Utilizziamo una coda per tenere traccia dei nodi che abbiamo già visitato. Alla fine del programma avremo solo nodi bianchi e neri.

Ora vediamo un algoritmo che restituisce, tramite uno Stack, il percorso minimo tra il nodo v e il nodo w .

```
Function MinPath(G, v, w)
    S  $\leftarrow$  EmptyStack
    (c, _, p)  $\leftarrow$  BFS(G, v) // _ indica qualcosa che poi non useremo più
    if c[w] = Nr then // controlliamo che w sia raggiunto da v
        return BuildMinPath(S, p, v, w)
    return S
```

```
Function BuildMinPath(S, p, v, w)
    S  $\leftarrow$  Push(S, w)
    if v  $\neq$  w then // se non abbiamo raggiunto la fine del percorso
        S  $\leftarrow$  BuildMinPath(S, p, v, p[w])
    return S
```

Dobbiamo dimostrare la correttezza del BFS, o meglio, che i valori restituiti dallo stesso abbiano senso. Dobbiamo quindi dimostrare il Theorem(BFS(G,v)):

1. $\forall w \in V, (v, w) \in \text{Reach}(G) \iff c[w] = Nr$
2. $\forall w \in V, d[w] = \delta(v, w)$
3. $\forall w \in V, \text{se } p[w] \neq \perp \text{ allora } p[w] \text{ è il predecessore di } w \text{ lungo un percorso minimo da } v \text{ a } w. \text{ Formalmente: } \exists \pi \in \text{MinPath}(G, v, w) : (\pi)_{|\pi|-2} = p(w)$

Prima però, abbiamo bisogno di alcuni lemmi.

12.1 Lemma(Distanze)

$$\forall v, w, u \in V : \text{se } (u, w) \in E \text{ allora } \delta(v, w) \leq \delta(v, u) + 1$$

Dimostrazione (per assurdo):

Se u è raggiungibile da v , allora lo è anche w . In questo caso, il percorso minimo da v a w non può essere maggiore del percorso minimo da v a u più l'arco u, w , e quindi la disegualanza vale. Se u non è raggiungibile da v , allora $\delta(v, u) = \infty$, e la disegualanza vale.

12.2 Lemma(Invarianti)

1. $\forall w \in V, d[w] \geq \delta(v, w)$
2. Sia $Q = [v_o, v_1, \dots, v_k]$ una istantanea della coda durante l'esecuzione di $\text{BFS}(G, v)$. Allora $\forall i \in [0, k), d[v_i] \leq d[v_{i+1}]$ (i) e $d[v_k] \leq 1 + d[v_0]$ (ii)

Dimostrazione (punto a):

La dimostrazione viene fatta per induzione. L'ipotesi induttiva è che $d[w] \geq \delta(v, w), \forall w \in V$.

Il caso base è la situazione immediatamente successiva a quando inseriamo v in Q al rigo 2 del codice della BFS. L'ipotesi induttiva rimane in quanto $d[v] = 0 = \delta(v, v)$ e $d[w] = \infty \geq \delta(v, w), \forall w \in V - \{v\}$.

Per il passo induttivo, consideriamo un vertice bianco w che viene scoperto durante la ricerca da un vertice u . L'ipotesi induttiva implica che $d[u] \geq \delta(v, u)$. Per l'assegnamento $d[w] \leftarrow d[v] + 1$ e il lemma delle distanze,

otteniamo:

$$d[w] = d[u] + 1 \geq \delta(v, u) + 1 \geq \delta(v, w)$$

Abbiamo così trovato che $d[w] \geq \delta(v, w)$. Per concludere dobbiamo fare questa osservazione: il nodo w viene messo in coda, e non succederà mai più perché è già grigio e il blocco *if-then* viene eseguito solo per i nodi bianchi. Quindi il valore di $d[w]$ non cambierà mai, e l'ipotesi induttiva resta.

Dimostrazione (punto b):

La dimostrazione viene fatta per induzione. Inizialmente quando la coda contiene solo v , il lemma è sicuramente verificato. Basta analizzare cosa succede prima del ciclo.

Per il passo induttivo, dobbiamo dimostrare che il lemma regge dopo sia l'eliminazione sia l'inserimento di un nodo dalla coda. Se la testa della coda v_0 viene eliminata, v_1 diventa la nuova testa (se la coda diventa vuota, il lemma è valido). Per l'ipotesi induttiva, $d[v_0] \leq d[v_1]$. Ma poi abbiamo che $d[v_k] \leq d[v_0] + 1 \leq d[v_1] + 1$, e le restanti disuguaglianze non vengono modificate. Quindi il lemma segue con v_1 come testa.

Al fine di comprendere cosa succede quando inseriamo un nodo nella coda, dobbiamo analizzare il codice più approfonditamente. Quando inseriamo in coda un nodo w nel blocco *if-then* del BFS, questo diventa v_{k+1} . In questo istante, abbiamo già rimosso il vertice v dalla coda Q , la cui stella uscente è in corso di scansione, e per l'ipotesi induttiva la nuova testa v_1 ha $d[v_1] \geq d[v]$. Quindi $d[v_{k+1}] = d[w] = d[v] + 1 \leq d[v_1] + 1$. Dall'ipotesi induttiva abbiamo inoltre che $d[v_k] \leq d[v] + 1$ e così $d[v_k] \leq d[v] + 1 = d[w] = d[v_{k+1}]$ e le restanti disuguaglianze rimangono invariate. Dunque il lemma segue che w viene inserito.

Dimostrazione (Theorem(BFS(G,v))):

Arriviamo così all'effettiva dimostrazione del teorema sulla correttezza del BFS.

Supponiamo, per lo scopo della contraddizione, che un qualche nodo riceva un valore d non uguale alla distanza del percorso minimo. Supponiamo esista un vertice w con il minimo $\delta(v, w)$ che presenta questo valore scorretto di d ; chiaramente $w \neq v$. Per il lemma degli invarianti, $d[w] \geq \delta(v, w)$ e quindi abbiamo che $d[w] > \delta(v, w)$. Il nodo w deve essere raggiungibile da v , perché se non lo è, allora $\delta(v, w) = \infty \geq d[w]$. Sia u il nodo immediatamente precedente a w nel percorso minimo da v a w , così che $\delta(v, w) = \delta(v, u) + 1$. Poiché $\delta(v, u) < \delta(v, w)$, e per come abbiamo scelto w , abbiamo $d[u] = \delta(v, u)$. Mettendo insieme queste proprietà otteniamo:

$$d[w] > \delta(v, w) = \delta(v, u) + 1 = d[u] + 1.$$

Ora consideriamo il momento in cui la BFS decide di togliere dalla coda il nodo v tramite il comando $(Q, v) \leftarrow Head\&Dequeue(Q)$. In questo istante, il vertice w è bianco, grigio o nero. Dobbiamo mostrare che in tutti e tre i casi troviamo una contraddizione nella disuguaglianza appena trovata. Se w è bianco, allora viene settato $d[w] \leftarrow d[v] + 1$, contraddicendo la disuguaglianza. Se w è nero, allora è stato già rimosso dalla coda, dunque $d[w] \leq d[v]$, contraddicendo la disuguaglianza. Se w è grigio, allora è stato colorato al momento della rimozione dalla coda di un certo vertice s , che è stato rimosso da Q prima di v e per il quale $d[w] = d[s] + 1$. Sappiamo che $d[s] \leq d[v]$, quindi $d[w] = d[s] + 1 \leq d[v] + 1$, che contraddice ancora la disuguaglianza.

Dunque concludiamo che $d[w] = \delta(v, w), \forall w \in V$. Tutti i nodi w raggiungibili da v devono essere scoperti, perché altrimenti avremo $\infty = d[w] > \delta(v, w)$. Per concludere la dimostrazione, osserviamo che se $p[w] = u$, allora $d[w] = d[u] + 1$. Quindi, possiamo ottenere un percorso minimo da v a w prendendo un percorso minimo da v a $p[w]$ e poi percorrendo l'arco $(p[w], w)$.

Facciamo un'analisi sulla complessità di questo algoritmo. Il ciclo while (o repeat-until) costa $|E|$, poiché copre al più tutti gli archi. A prescindere di tutto quello che succede dopo, all'inizio viene richiamata la funzione *Init()* che ha un costo $|V|$, dunque il BFS paga sempre l'inizializzazione. La complessità totale dell'algoritmo sarà quindi $|E| + |V|$.

13 Visita in profondità

Con il BFS abbiamo svolto una visita in ampiezza. Come con gli alberi, anche con i grafi è possibile eseguire una visita in profondità. La differenza con gli alberi sta nel fatto che non esiste il concetto di *InOrder*.

In generale, quello che andiamo a fare è esplorare tutto ciò che è esplorabile da qualsiasi nodo. Importante specificare che qui non esiste il concetto di distanza, presente invece nella visita in ampiezza perché dovevamo andare a prendere il percorso più breve. Inoltre c'è la possibilità di identificare la presenza di cicli. L'algoritmo è il seguente:

```
Function DFV(G)
  ((c,p,d,f),t) ← (Init(G), 0)

  for each v ∈ V do
    if c[v] = Bn then
      (c,p,d,f,t) ← DFV(G,v,c,p,d,f,t)

  return (c,p,d,f)
```

Il vettore p non ci fa più trovare il percorso minimo perché non c'è il concetto di distanza, ma ci permette di identificare la foresta, cioè un insieme di alberi. d è il vettore dei tempi di inizio visita; f è il vettore dei tempi di fine visita, mentre t è il contatore degli istanti di tempo.

```
Function Init(G)
  for each v ∈ V do
    (c[v], p[v]) ← (Bn, ⊥)
  return (c,p,d,f)
```

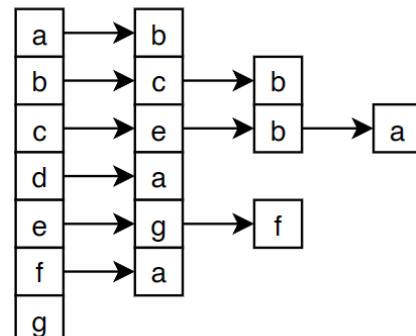
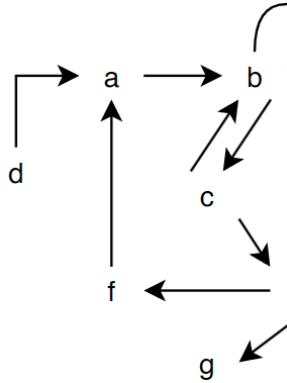
↗ // la funzione li dichiara solo

```
Function DFV(G,v,c,p,d,f,t)
  (c[v],d[v],t) ← (Gr,t,t+1)

  for each w ∈ Adj[v] do
    if c[w] = Bn then
      p[w] ← v
      (c,p,d,f,t) ← DFV(G,w,c,p,d,f,t)

  (c[v],f[v],t) ← (Nr,t,t+1)
  return (c,p,d,f,t)
```

Vediamo un esempio:



Applichiamo l'algoritmo e vediamo cosa otteniamo.

t	0	1	2	3	4/5	6/7	8	9	10	11	12
v	a	b	c	e	g	f	e	c	b	a	d
d	0	1	2	3	4	6	-	-	-	-	12
f						5	7	8	9	10	11
p	⊥	a	b	c	e	e	-	-	-	-	⊥

A prescindere da qualsiasi cosa, andiamo sempre a visitare tutti i nodi e tutti gli archi una sola volta, quindi non possiamo fare ne più ne meno di questo. Per ciò la complessità è pari a $\Theta(|E| + |V|)$.

13.1 Teorema (struttura a parentesi della DFV)

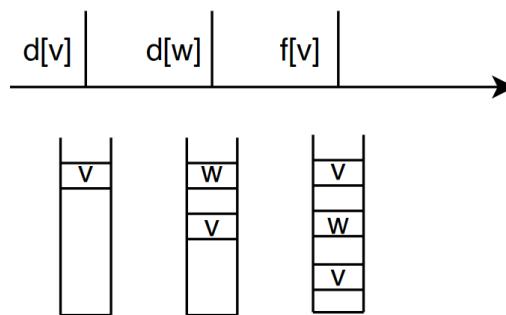
$\forall v, w \in V$, con $d[v] < d[w]$. Una delle seguenti è vera:

- a) $d[v] < d[w] < f[w] < f[v]$
- b) $d[v] < f[v] < d[w] < f[w]$

Ossia è impossibile che $d[v] < d[w] < f[v] < [fw]$.

Dimostrazione:

Supponiamo per assurdo che il teorema sia falso e che quindi valga $d[v] < d[w] < f[v] < [fw]$.



All'istante $d[v]$, v viene messo sullo stack a grigio perché è appena stato scoperto. Poi arriva l'istante $d[w]$ dove anche w viene scoperto e quindi messo sullo stack a grigio. In questo momento, v non ha ancora terminato, quindi

si trova ancora sullo stack. Arriva poi l'istante $f[v]$ in cui tecnicamente v dovrebbe terminare. Per settare la sua chiamata di fine visita, l'unica cosa possibile è che ci sia stata un'altra chiamata su di esso che permetta di riaverlo in cima. Questo però non può accadere perché per fare un'altra chiamata il nodo v deve essere bianco (per come è stato costruito l'algoritmo) ma è grigio. Dunque è un assurdo.

13.2 Lemma (percorsi nella foresta di visita)

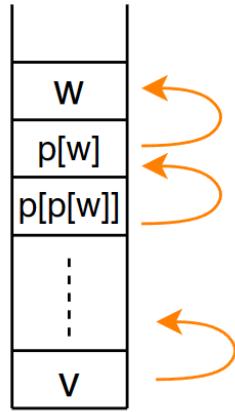
Prima di presentare il lemma, diamo la definizione di foresta di visita:

Sia $G = \langle V, E \rangle$ e p = vettore dei predecessori di $DFV(G)$. Allora $F_p = \langle v, \{(p[v], v) / v \in V, p[v] \neq \perp\} \rangle$. F_p indica la foresta dei predecessori.

Il lemma afferma:

$$\forall v, w \in V, d[v] < d[w] < f[w] < f[v] \iff \exists \pi \in Path(F_p, v, w)$$

Dimostrazione (\Rightarrow):



Sappiamo che $d[v] < d[w]$ quindi v viene scoperto prima di w . Tra i due potrebbero esserci altri nodi, predecessori di w . Per definizione di foresta, esistono quegli archi, quindi sicuramente esiste un percorso da v a w .

(\Leftarrow):

Sappiamo che $\exists \pi \in Path(F_p, v, w)$. Quindi possiamo affermare che w è discendente di v . Questo implica che $d[v] < d[w]$ perché v viene scoperto prima e che $f[w] < f[v]$ perché il processo di esplorazione di w all'interno del sottoalbero di v deve essere completato prima di poter dichiarare che tutto il sottoalbero radicato in v è esplorato. Data l'esistenza di π , e per il teorema precedente, concludiamo che $d[v] < d[w] < f[w] < f[v]$.

13.3 Teorema (Percorso Bianco)

$$\forall v, w \in V, \exists \pi_F \in Path(F_p, v, w) \iff \exists \pi_G \in Path(G, v, w) \text{ t.c. } c[\pi_i] = Bn, \forall i \in [0, |\pi|]$$

Cioè, per ogni v, w in V , w è discendente di v in F_p se e solo se al tempo $d[v]$ esiste un percorso bianco da v a w in G .

Dimostrazione (\Rightarrow)

Supponiamo che w è un discendente proprio di v nella foresta di visita. Per il lemma dei percorsi nella foresta di visita, $d[v] < d[w]$, e quindi w è bianco al tempo $d[v]$. Poiché w può essere un qualunque discendente di v , tutti i nodi dell'unico percorso semplice da v a w nella foresta di visita sono bianchi al tempo $d[v]$.

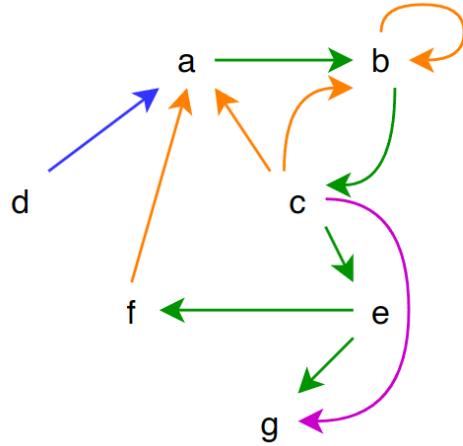
(\Leftarrow)

Supponiamo esista un percorso di nodi bianchi da v a w al tempo $d[v]$, ma w non diventa un discendente di v nella foresta di visita. Senza perdita di generalità, assumiamo che tutti i nodi, tranne w , diventano discendenti di v lungo il percorso. (Altrimenti, facciamo essere w il più vicino nodo a v lungo il percorso che non diventa un suo discendente). Sia z il predecessore di w nel percorso, così che z sia un discendente di v (perché abbiamo detto che w è il primo non discendente, quindi i nodi precedenti lo sono). v e z potrebbero anche essere lo stesso nodo. Per il lemma dei percorsi lungo la foresta di visita, $f[z] \leq f[v]$. Poiché w deve essere scoperto dopo che lo è stato v ,

ma prima che sia settato il tempo di fine visita di z , abbiamo che $d[v] < d[w] < f[z] \leq f[v]$. Per il teorema della struttura a parentesi della DFV, dobbiamo avere $d[v] < d[w] < f[w] < f[z] \leq f[v]$ e dunque, per il lemma dei percorsi nella foresta di visita, possiamo dire che w deve essere un discendente di v .

13.4 Classificazione degli archi

Un'altra importante caratteristica della DFV è che la ricerca può essere utilizzata per classificare gli archi del grafo in input $G = \langle V, E \rangle$. Il tipo di ogni arco provvedere a dare importanti informazioni riguardo un grafo. Per esempio, vedremo che un grafo orientato è aciclico se e solo se non presenta archi all'indietro. Vediamo un disegno di esempio e diamo la formalizzazione:



$$\forall (v, w) \in E$$

- **Arco di foresta** se $(v, w) \in F_p$, al tempo $d[v], c[w]$ è bianco;
- **Arco all'indietro** se $(v, w) \in F_p$, al tempo $d[v], c[w]$ è grigio;
- **Arco in avanti** se $(v, w) \in F_p$, al tempo $d[v], c[w]$ è nero e w è discendente di v in F_p ;
- **Arco di attraversamento** se $(v, w) \in F_p$, al tempo $d[v], c[w]$ è nero e w è non discendente di v in F_p ;

Vediamo l'algoritmo che sfrutta questa differenza e ci permette di identificare l'aciclicità.

```

Function Acyclic (G)
  c  $\leftarrow$  Init(G)

  for each v  $\in$  V do
    if c[v] = Bn then
      if  $\neg$ Acyclic(G,v) then
        return  $\perp$ 

  return T

```

Per dimostrare la correttezza di questo algoritmo, dobbiamo assumere che $Acyclic(G, v)$ sia anch'esso corretto. Per l'effettiva dimostrazione, basta spiegare cosa fa.

```

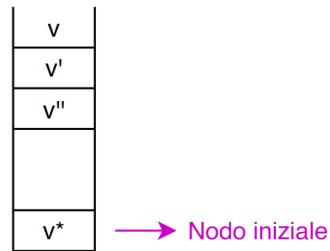
Function Acyclic(G, v)
  c[v] ← Gr

  for each w ∈ Adj[v] do
    if c[w] = Bn then
      if ¬Acyclic(G, w) then
        return ⊥
    else if c[w] = Gr then
      return ⊥

  c[v] = Nr
  return T

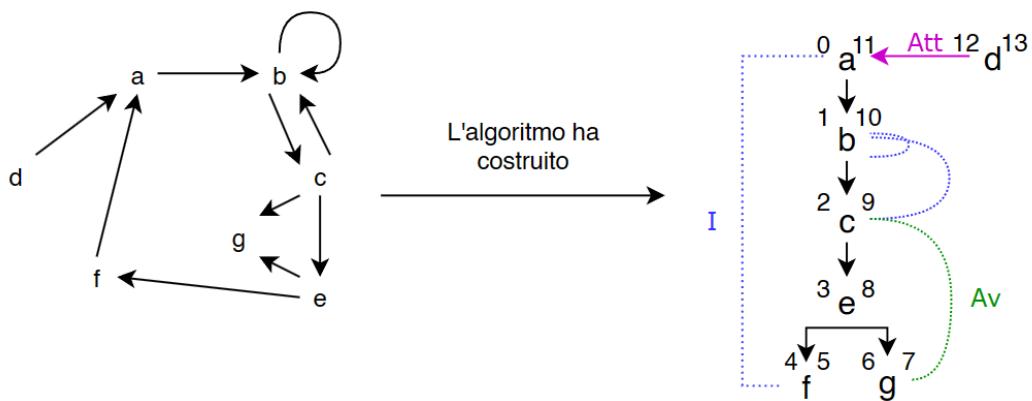
```

Per la correttezza di questo algoritmo notiamo che durante l'esecuzione lo stack sarà così:



Per il lemma del percorso in una foresta, sappiamo che questi formano un percorso. Osserviamo inoltre che i nodi sullo stack sono solo grigi, quindi quando troviamo w grigio, sicuro sarà un vertice tra v, v', v'', \dots, v^* . Due specifiche: v^* è il nodo iniziale, cioè quello che viene passato alla funzione; inoltre questo succede quando la funzione restituisce \perp nel secondo blocco *if-then*.

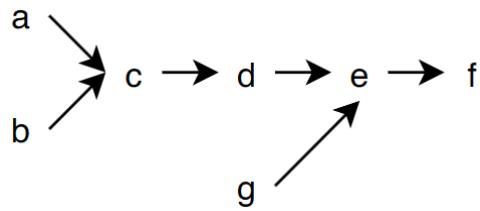
Ora vediamo il caso in cui la funzione restituisce \perp nel secondo blocco *if-then*. Supponiamo di avere:



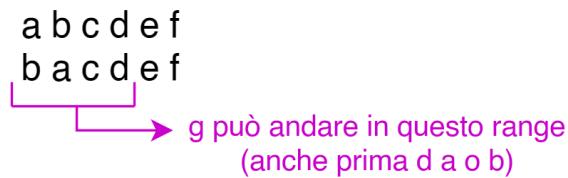
A prescindere dalla foresta costruita, l'algoritmo analizzandola troverà sempre un ciclo (quindi un arco all'indietro). La complessità di questo algoritmo è uguale a quella della BFS. In particolare, può essere addirittura più veloce in quanto, se viene trovato un ciclo nel primo nodo, si ferma. Un algoritmo del genere è importante perché può essere usato in diverse applicazioni. Ad esempio: in java quando proviamo a far ereditare alla classe a, b e viceversa, il compilatore non lo fa fare perché costruisce una struttura a grafo dell'ereditarietà e applica un algoritmo del genere. Altra applicazione è lo scheduling a livello di sistemi operativi.

14 Ordinamento Topologico

Prendiamo in considerazione questo esempio:



Vogliamo trovare una linearizzazione, o meglio, un ordinamento topologico.



Non esiste dunque un unico ordinamento topologico. I casi particolari sono:

- Se il grafo è una sequenza, allora l'ordinamento topologico è pari a 1;
- Se il grafo ha n nodi ma non ha archi, allora l'ordinamento topologico è pari a $n!$.

Andiamo quindi ora a formalizzare il tutto:

Un ordinamento topologico di un grafo G è una permutazione dei vertici di G tale che $\forall(v, w) \in E, (v <_{\pi} w)$

Lemma:

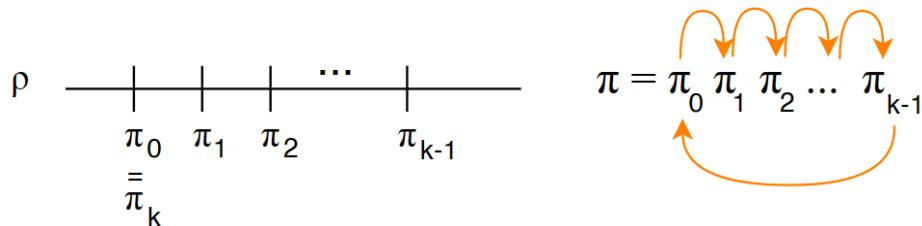
Se G è ciclico, allora non esiste un ordinamento topologico per G .

Dimostrazione:

Ipotesi: G ciclico. Quindi $\exists \pi \in Path(G), last(\pi) = first(\pi)$

Assurdo: $\exists \rho$ ordinamento topologico in G .

Se ρ è un ordinamento topologico in $G \Rightarrow$ è una permutazione \Rightarrow contiene tutti i vertici.



π_0 dovrebbe star dopo dato che è π_k , ma sta prima, quindi è un assurdo.

```

Function TopologicalOrdering(G)
  (c, S) ← (Init(G), EmptyStack(S))

  for each v ∈ V do
    if c[v] = Bn then
      S ← TopologicalOrdering(G, S, v)

  return S

```

```

Function TopologicalOrdering(G, S, v)
  c[v] ← Gr

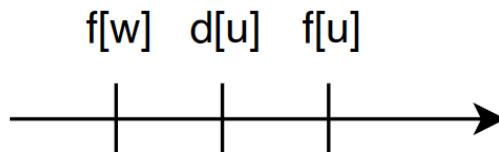
  for each w ∈ Adj[v] do
    if c[w] = Bn then
      S ← TopologicalOrdering(G, S, w)

  (c[v], S) ← (Nr, Push(S, v))
  return S

```

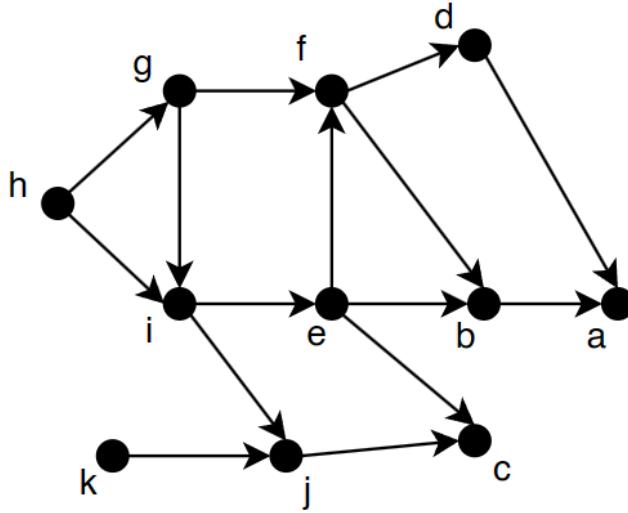
Sia $S = v_0, v_1, \dots, v_k$ il contenuto di S alla fine dell'esecuzione. Se è un ordinamento topologico allora $\forall(u, w) \in E(\exists i, j \in [0, k] \text{ t.c. } v_i = u, v_j = w, i < j)$. Dire questo ci permette di dire che, dato il fatto che $f[v_0] > f[v_1] > \dots > f[v_k]$, allora $f[u] > f[w]$ poiché u viene prima di w .

Vediamo ora al tempo $d[u]$ cosa fa l'algoritmo. A questo punto u o è bianco o nero (grigio no perché altrimenti ci sarebbe un ciclo). Se è bianco, per il lemma dei percorsi nella foresta di visita otteniamo $d[u] < d[w] < f[w] < f[u]$, in particolare che $f[w] < f[u]$ che era proprio ciò che volevamo. Se è nero:



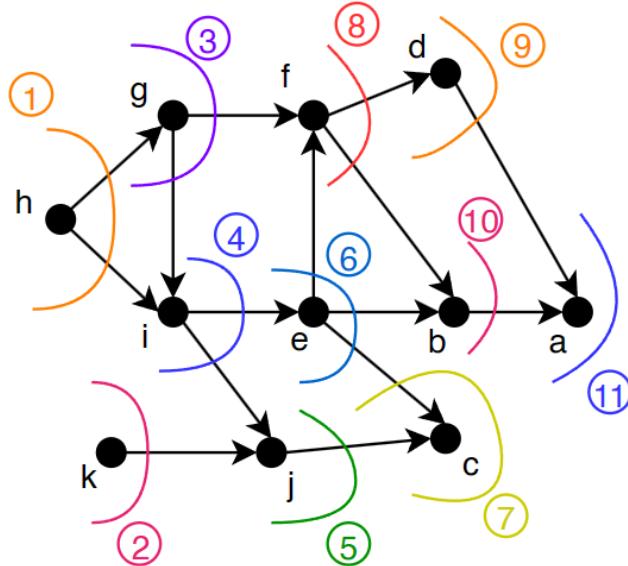
Per il teorema della struttura a parentesi, $f[u]$ deve stare per forza lì. Abbiamo così trovato nuovamente che $f[w] < f[u]$.

Vediamo, come risoluzione dello stesso problema, un altro modo. Consideriamo il seguente grafo aciclico:



Osserviamo che in un grafo aciclico finito deve esistere almeno un nodo con grado entrante 0.

L'algoritmo parte da un nodo con grado entrante 0. Questo nodo, non avendo vincoli, possiamo toglierlo, così da ottenere un grafo indotto in cui permuta l'aciclicità. Questo sottografo, essendo aciclico, deve avere almeno un nodo con grado entrante 0. Si effettua lo stesso ragionamento e così via. Questi nodi che andiamo man mano ad eliminare, li possiamo mettere nella permutazione fino a quando non terminano.



Partendo da h , l'algoritmo lo elimina e si crea un sottografo con adesso k e g con grado entrante 0. Si sceglie poi k o g arbitrariamente. Si continua così fino alla fine.

L'algoritmo terrà il grado entrante di ogni nodo in un vettore. Quelli con grado entrante 0 verranno messi in una coda (va bene una qualsiasi struttura dati, usiamo la coda perché è più semplice e meno costosa). Quando ne andiamo a prendere uno, per simulare la creazione del sottografo, esploriamo la sua stella uscente e decrementiamo di 1 il grado entrante di questi. Facendo così sappiamo che l'algoritmo restituirà sicuramente una permutazione (perché lavoriamo e prendiamo nodi che non presentano vincoli).

Lemma:

Ipotesi: G aciclico e finito

Tesi: $\exists v \in V, t.c. ge(v) = 0$

In particolare, $ge = |\{(w, v) \in E / w \in V\}|$. ($|$ sta per cardinalità).

Dimostrazione (contraddizione)

Supponiamo che tutti i nodi hanno $ge > 0$. Consideriamo $V = \{v_0, v_1, \dots, v_{n-1}\}$. Prendiamo ad esempio v_5 . Questo ha grado entrante, quindi esiste almeno un v_i tale che $v_i \rightarrow v_5$. Supponiamo sia v_8 . Possiamo fare lo stesso ragionamento anche con questo nodo, quindi potremmo ottenere una coda del tipo $v_0 \rightarrow v_8 \rightarrow v_5$. Dopo un numero di passi pari al numero di nodi, arriviamo all'ultimo nodo disponibile. Anche questo avrà almeno un arco entrante, quindi dovrà esistere un v_i che raggiunge questo nodo. Ma i nodi a disposizione sono finiti, quindi questo v_i dovrà essere per forza uno dei nodi già usati. Si va così a creare un ciclo, ma è una contraddizione dato che nell'ipotesi avevamo detto che G è aciclico.

Lemma:

Ipotesi: G aciclico, $G' \sqsubseteq G$ (G' è sottografo di G)

Tesi: G' aciclico

Dimostrazione (contraddizione)

Neghiamo la tesi e supponiamo che $\exists \pi \in Cycle(G')$. Dato che $G' \sqsubseteq G$ (gli archi di G' sono anche in G), allora $\pi \in Path(G) \Rightarrow \pi \in Cycle(G)$ ma non è possibile dato che nell'ipotesi abbiamo detto che G è aciclico.

Vediamo l'algoritmo:

Function EnteringDegree(G)

```
for each  $v \in V$  do  
     $ge[v] \leftarrow 0$   
  
for each  $v \in V$  do  
    for each  $w \in Adj[v]$  do  
         $ge[w]++$   
  
return  $ge$ 
```

Function InitQueue(G, ge)

```
 $Q \leftarrow EmptyQueue$   
  
for each  $v \in V$  do  
    if  $ge[v] = 0$  then  
         $Q \leftarrow Enqueue(Q, v)$   
  
return  $Q$ 
```

```
Function TopologicalOrdering(G)
    ge ← EnteringDegree(G)
    Q ← InitQueue(G,ge)

    While ¬isEmpty(Q) do
        (Q,v) ← Head&Dequeue(Q)

        for each w ∈ Adj[v] do
            ge[w]--
            if ge[w] = 0 then
                Q ← Enqueue(Q,w)

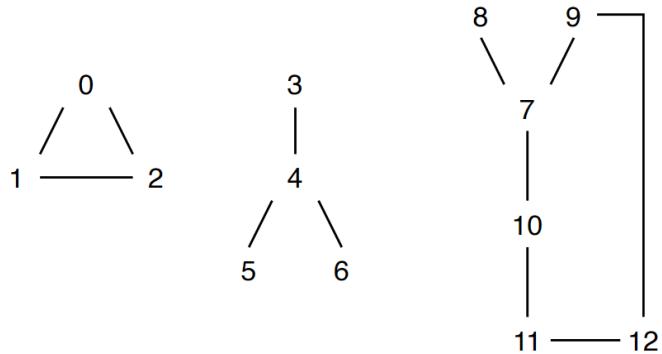
    L ← Append(L,v)

    return L
```

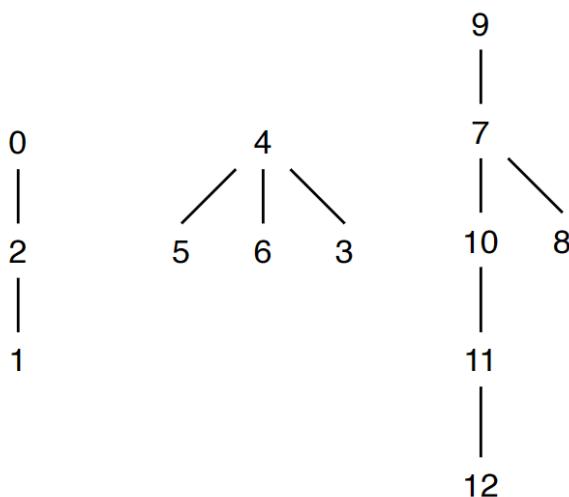
Il ciclo *While* viene eseguito n volte. Lo usiamo perché potremmo incontrare un grafo vuoto, quindi, in caso, il ciclo non si attiverebbe.

15 Componenti connesse

Supponiamo di avere il seguente grafo.

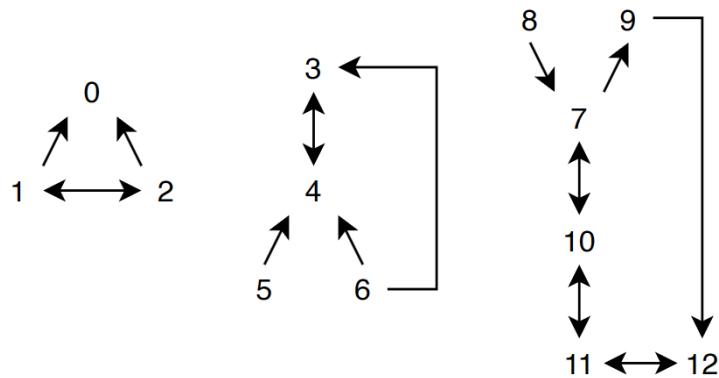


Il nostro obiettivo è quello di trovare un algoritmo che restituisca le componenti connesse. Quello che facciamo è considerare il grafo non orientato come orientato da ambo i versi. Facciamo un a visita in profondità e dal vettore dei predecessori estraiamo la foresta di visita. Ogni albero nella foresta è una componente连通的, addirittura massimale. Una possibile foresta può essere:

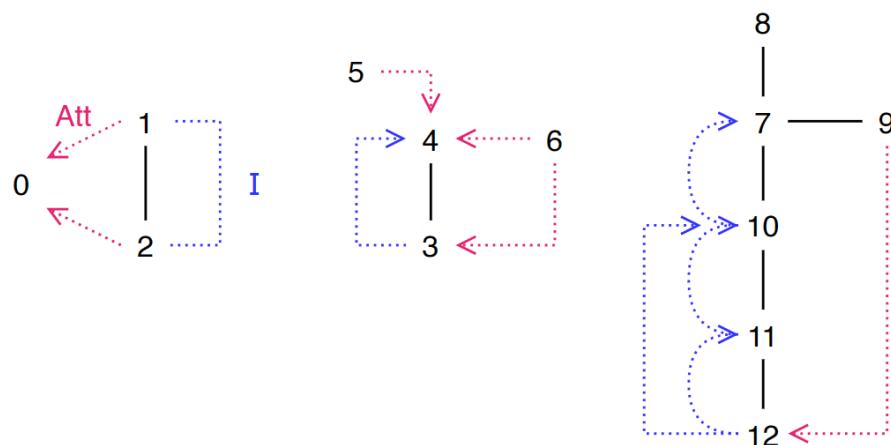


Osservazione: nei grafi non orientati non ci sono archi di attraversamento. Per dimostrarlo, supponiamo che esista un arco che va dal nodo 6 al nodo 2. Supponiamo inoltre che 2 lo abbia trovato nero. Dato che questo arco deve valere anche al contrario, allora dovremmo avere anche $2 \rightarrow 6$. Ma 2 è nero e 6 è bianco, quindi come è possibile che 6 non sia stato preso quando è stata esplorata la stella uscente di 2? Per il teorema del percorso bianco, possiamo dunque dire che è un assurdo.

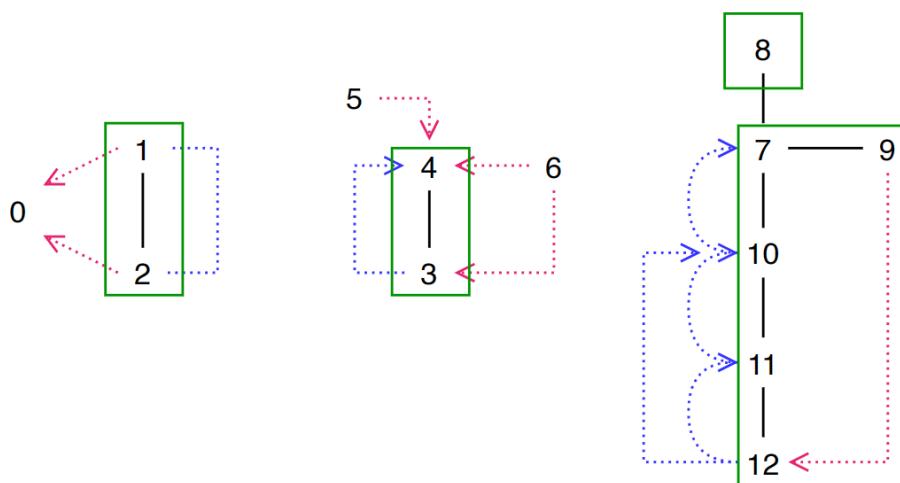
Supponiamo ora di avere:



Avremo:

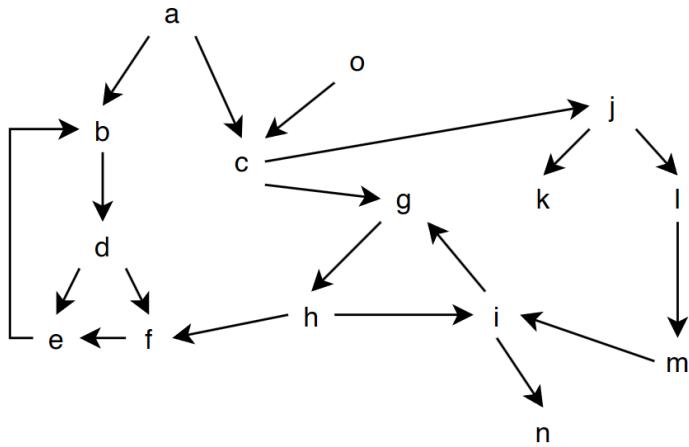


Le componenti sono:

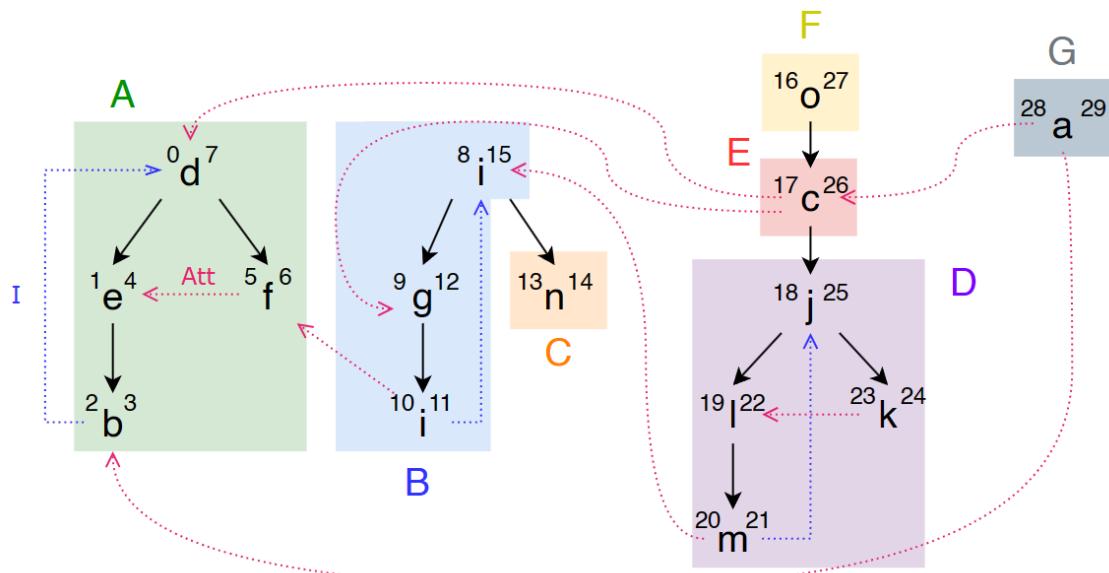


Inoltre, 0,5,6 prendono il nome di componenti isolate.

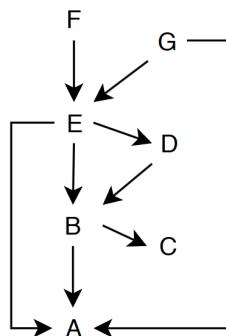
Vediamo un altro esempio più complesso.



La foresta di visita sarà:



Con le componenti otterremo:



Che è un grafo aciclico. Se c'è una componente connessa, allora questa è contenuta all'interno del grafo.

Vogliamo capire se in un grafo c'è una componente fortemente connessa. Per farlo, supponiamo di prendere il grafo trasposto. Facciamo una visita partendo dall'ordine di fine visita della ricerca precedente (quindi partendo dal nodo con tempo di fine visita maggiore e andiamo a ritroso). Se vediamo che da un certo nodo raggiungo nodi che nel grafo originale raggiungevano lo stesso, allora siamo sicuri che l'insieme di questi nodi forma una componente fortemente connessa. Importante specificare che non ci curiamo degli archi che vanno in nodi già esplorati. Vediamo l'algoritmo:

```
Function Transpose(G)
    V' ← V

    for each v ∈ V do
        for each w ∈ Adj[v] do
            Adj'[w] ← Insert(Adj'[w], v)

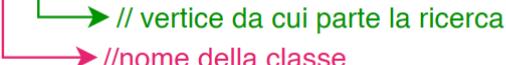
    return(V', Adj')
```

Con $V' \leftarrow V$ facciamo una copia dell'insieme V in V' che possiamo considerare come una struttura dati semplice. $Adj'[w]$ è una struttura dati degli adiacenti di w . La usiamo perché vogliamo gli archi al contrario. Questo algoritmo ha complessità $|V| + |E|$. $|V|$ per $V' \leftarrow V$; mentre $|E|$ per la chiamata a $Insert()$ che è costante. In particolare è costante se usiamo, ad esempio, una lista ma non se usiamo un albero.

```
Function SCC(G)
    S ← DFS1(G) // stesso algoritmo dell'ordinamento topologico via DFS
    GT ← Transpose(G)
    scc ← DFS2(GT, S) // scc è un vettore
    return scc
```

```
Function DFS2(G, s)
    c ← Init(G) // variante della Init() per inizializzare il vettore dei colori

    While ¬isEmpty(S) do
        (S, v) ← Top&Pop(S)
        if c[v] = Bn then
            DFS2(G, scc, c, v, v)
    return scc
```



```
Function DFS2(G, scc, c, v, s)
    c[v] ← Gr
    scc[v] ← s

    for each w ∈ Adj[v] do
        if c[w] = Bn then
            scc[w] ← s
            DFS2(G, scc, c, w, s)

    scc[v] ← s
    c[v] ← Nr
```

L'istruzione $scc[w] \leftarrow s$ può essere fatta in questo modo, quindi in InOrder, oppure può essere fatta in InOrder o addirittura in PostOrder. La scelta è arbitraria.

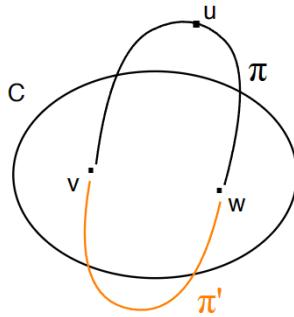
Vediamo ora i teoremi per verificarne la correttezza.

Lemma:

Ipotesi: sia C , scc massimale di G , $v, w \in C$

Tesi: $\forall \pi \in Path(G, v, w), \pi \subseteq C$

Dimostrazione (assurdo):



Per assurdo neghiamo la tesi. Da v arriviamo a u e da u torniamo a v tramite il restante percorso di π più π' . Quest'ultimo esiste perché $v, w \in C$ e può stare o dentro o fuori C senza distinzioni. Dunque u fa parte della componente连通. Quindi come è possibile che non stia in C dato che è massimale? Assurdo.

Lemma:

Ipotesi: Sia C , scc di G e p vettore dei predecessori calcolato da una DFS

Tesi: $C \subseteq T$ per un qualche albero T in F_p

Dimostrazione:

La componente massimale non è mai vuota, quindi c'è almeno un elemento. Sia $v \in C$ e t il tempo di inizio visita di v , con $t \leq d[w], \forall w \in C$ (v è il primo nodo).

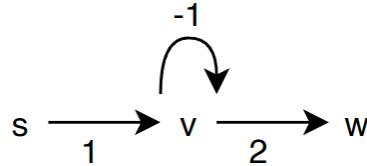
Apparte v , tutti gli altri nodi sono bianchi. Per il teorema del percorso bianco, v raggiunge tutti i nodi in C all'interno di F_p . Ipoteticamente possiamo dire che un elemento di C che viene raggiunto da v si trova in un altro albero. Però questo non è possibile perché se questo nodo fa parte della stessa componente di v , per il lemma precedente, il percorso che li unisce deve stare nella componente.

Dimostrati questi due lemmi, adesso non basta altro che vedere le varie casistiche: la scc prende tutti i nodi? Può prendere nodi erronni? Per la prima domanda la risposta è sì. Questo perché nel grafo trasposto viene fatta una visita al contrario. Per la seconda domanda, la scc non prende nodi erronni perché a sinistra e in basso non va perché gli archi sono invertiti e a destra ci sono nodi già esplorati.

16 Grafi pesati

Un grafo pesato è definito come $G = \langle V, E, wg \rangle$, dove $wg : E \rightarrow \mathbb{R}$, ed è la funzione che assegna ad ogni arco un peso. Di solito al posto di \mathbb{R} viene usato \mathbb{Q} perché \mathbb{R} non è rappresentabile.

In questi grafi, il concetto di percorso minimo può perdere di significato. Ad esempio:

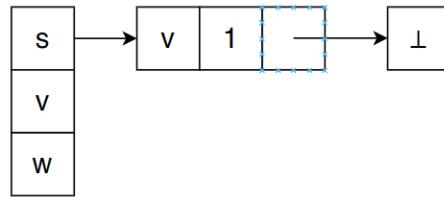


In questo caso possiamo avere un percorso semplice di lunghezza 2 di peso 3. Ma possiamo avere anche percorsi di peso 2 (1-1+2) o di peso 1 (1-1-1+2) e così via. Quindi è possibile trovarsi nella situazione in cui i percorsi hanno pesi infinitamente piccoli, e quindi non c'è il minimo.

La rappresentazione del grafo in questo caso sarà:

	s	v	w
s	⊥	1	⊥
v	⊥	-1	2
w	⊥	⊥	⊥

Un'altra possibilità è:



16.1 Concetto di peso di un percorso

Si possono usare due notazioni: wg o δ_{wg} .

$$\delta_{wg} = \begin{cases} \sum_{i=0}^{|\pi|-2} wg((\pi)_i, (\pi)_{i+1}), & \text{se } |\pi| \geq 2 \\ 0, & \text{altrimenti} \end{cases}$$

Andiamo semplicemente a fare la somma dei pesi degli archi.

Sapendo che:

$$\delta(S, v) = \min_{\pi \in \text{Path}(G, s, v)} (|\pi| - 1) = \min_{\pi \in \text{Path}(G, s, v)} (\delta(\pi))$$

Se al posto di $\delta(\pi)$ mettiamo $\delta_{wg}(\pi)$ otteniamo la distanza pesata da s a v . Quindi:

$$\delta_{wg}(s, v) = \min_{\pi \in \text{Path}(G, s, v) (\delta_{wg}(\pi))} \delta_{wg}(\pi)$$

Prima di presentare il primo lemma, diamo la definizione di $\text{MinWPath}(G, s, v)$:

$$\begin{aligned} \text{MinWPath}(G, s, v) &\subseteq \text{Path}(G, s, v) \\ \pi \in \text{MinWPath}(G, s, v) &\text{ se } \delta_{wg}(\pi) = \delta(s, v) \end{aligned}$$

Ora passiamo al primo lemma.

Lemma1:

Ipotesi: G grafo pesato con pesi non negativi, $\pi \in \text{MinWPath}(G)$

Tesi: $\forall i, j \in [0, |\pi|], i \leq j. \pi[i, j] \in \text{MinWPath}(G, (\pi)_i, (\pi)_j)$

Dimostrazione (assurdo):

Supponiamo $\exists i, j (i \leq j)$ t.c $\pi[i, j] \notin \text{MinWPath}(G, \pi_i, \pi_j)$.

Allora $\exists \pi' \in \text{MinWPath}(G, (\pi)_i, (\pi)_j)$ t.c. $\delta_{wg}(\pi') < \delta_{wg}(\pi[i, j])$.

Ora scomponiamo π in: $\pi[0, i - 1] \pi[i, j] \pi[j + 1, |\pi| - 1]$

Dato che π e π' partono e terminano nello stesso punto possiamo scrivere: $\pi = \pi[0, i - 1] \pi' \pi[j + 1, |\pi| - 1]$.

Dunque esisterà $\pi'' = \pi[0, i) \pi' \pi(j, |\pi| - 1)$.

Però π' ha un peso minore rispetto a $\pi[i, j]$, quindi $\delta_{wg}(\pi'') < \delta_{wg}(\pi)$, il che è impossibile perché π avevamo supposto fosse il minimo.

Corollario:

Ipotesi: G grafo pesato e $\pi = \pi' \cdot vw \in \text{MinWPath}(G, s, w)$

Tesi: $\delta_{wg}(\pi) = \delta_{wg}(s, v) + wg(v, w)$

(vw è un singolo arco ed è l'ultimo)

Dimostrazione:

Sapendo per ipotesi che $\pi = \pi' \cdot vw$, possiamo scrivere $\delta_{wg}(\pi) = \delta_{wg}(\pi' \cdot v) + wg(v, w)$. Ora, $\pi' \cdot v$ è un sottopercorso di π ed è minimo per il lemma precedente. Dunque $\delta_{wg}(\pi' \cdot v) = \delta_{wg}(s, v)$. In conclusione, $\delta_{wg}(\pi) = \delta_{wg}(s, v) + wg(v, w)$, che è proprio la nostra tesi.

Lemma2:

Ipotesi: G grafo pesato

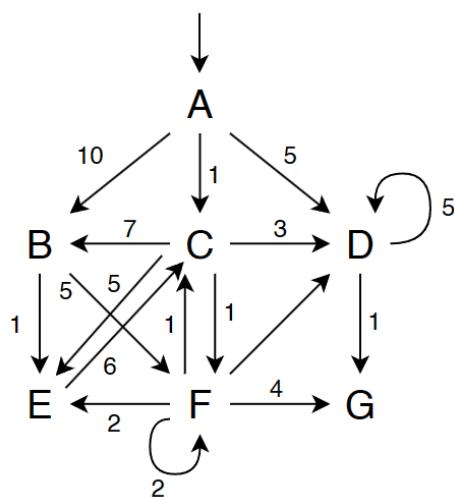
Tesi: $\forall s, v, w \in V, (v, w) \in E. \delta_{wg}(s, w) \leq \delta_{wg}(s, v) + wg(v, w)$

Dimostrazione (contraddizione):

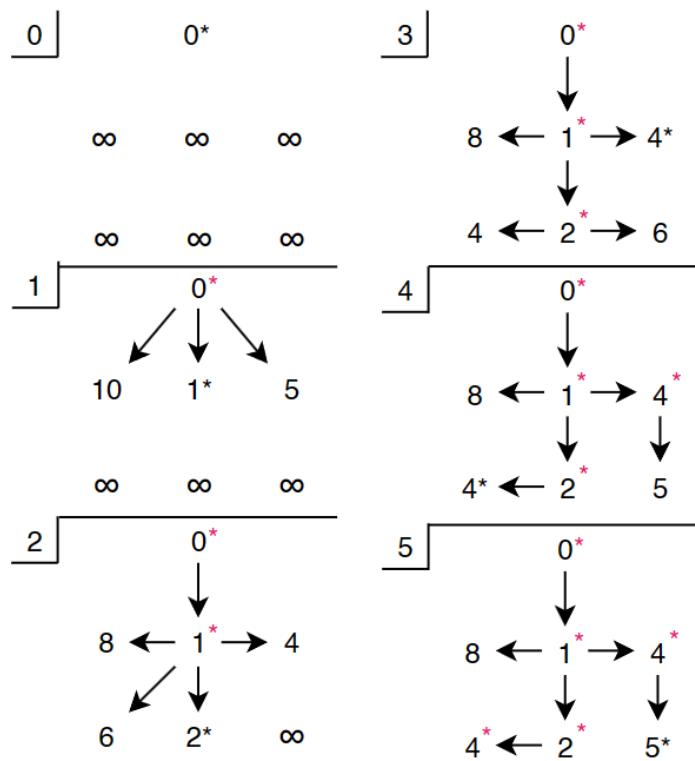
Supponiamo che $\exists s, v, w \in V$ con $(v, w) \in E$ t.c. $\delta_{wg}(s, w) > \delta_{wg}(s, v) + wg(v, w)$.

Esistono quindi $\pi \in \text{MinWPath}(G, s, w)$ e $\pi' \in \text{MinWPath}(G, s, v)$ t.c. $\delta_{wg}(\pi) = \delta_{wg}(s, w)$ e $\delta_{wg}(\pi') = \delta_{wg}(s, v)$. $\delta_{wg}(\pi) > \delta_{wg}(\pi') + wg(v, w) = \delta_{wg}(\pi' \cdot w)$, ma $\pi' \cdot w \in \text{Path}(G, s, w)$ che è più piccolo di un percorso minimo, il che è un assurdo.

Vediamo ora qual è l'approccio per il calcolo di un percorso minimo a partire da una certa sorgente. Si tratta di una generalizzazione della visita in ampiezza. Vediamo un esempio:



Durante il procedimento avviene quello che viene chiamato rilassamento dell'arco, che va a migliorare la stima. Vediamo graficamente tutti i passaggi:



E così via fino alla fine. L'asterisco indica che si sta lavorando su quel nodo. Se è colorato, allora è deciso.

Vediamo ora il codice di due funzioni. La prima è la $Init(G,s)$ che va a settare il tutto come nel caso 0.

```
Function Init( $G, s$ )
  for each  $v \in V$  do
     $d[v] = \infty$ 
     $p[v] = \perp$ 
   $d[s] = 0$ 
```

L'altra è la funzione $Relax(G, (v,w))$, che effettua il rilassamento dell'arco.

```
Function Relax( $G, (v,w)$ )
  if  $d[w] > d[v] + wg[v,w]$  then
     $d[w] \leftarrow d[v] + wg[v,w]$ 
     $p[w] \leftarrow v$ 
```

16.2 Algoritmo di Dijkstra

```

Function Dijkstra(G, s)
    (d, p)  $\leftarrow$  Init(G, s)
    Q  $\leftarrow$  V

    repeat
        (Q, V)  $\leftarrow$  RemoveMin(Q, d)
        for each w  $\in$  Adj[v] do
            Relax(G, (v, w))
    until Q =  $\emptyset$ 
    return (d, p)

```

Facciamo delle specifiche. Con questo algoritmo si prende in considerazione il caso in cui i pesi di tutti i percorsi siano non negativi. Con la riga $Q \leftarrow V$ andiamo a fare una copia degli elementi di V in Q . Quest'ultimo può essere un qualsiasi insieme in cui si può fare l'inserimento e la ricerca del minimo (non gli ABR perché non ammettono i duplicati).

Per quanto riguarda il costo, possiamo dire che:

- $(d, p) \leftarrow \text{Init}(G, s)$ ha un costo pari a $\Theta(|V|)$;
- $Q \leftarrow V$ ha un costo pari a $\Theta(|V|)$;
- Il blocco *repeat-until* ha un costo pari a $\Theta(|V| \cdot |V| + |E|)$. Con una coda a priorità invece sarà $\Theta((|V| + |E|) \cdot \log |V|)$ dato che ci sarà un'istruzione in più in *Relax()*.

Per un grafo denso, conviene non usare una coda.

Dimostriamo ora la correttezza dell'algoritmo. Per farlo dobbiamo prima enunciare alcuni lemmi.

Lemma3:

Ipotesi: G grafo pesato

Tesi: dopo un'applicazione di *Relax*($G, (v, w)$) si ha $d[w] \leq d[v] + wg(v, w)$

Dimostrazione:

Se, poco prima di rilassare l'arco (v, w) , abbiamo $d[w] > d[v] + wg(v, w)$, allora avremo $d[w] = d[v] + wg(v, w)$ in seguito (per come è stato costruito l'algoritmo). Se, invece, abbiamo $d[w] \leq d[v] + wg(v, w)$ poco prima del rilassamento, allora nè $d[v]$ nè $d[w]$ cambiano, e quindi avremo comunque $d[w] \leq d[v] + wg(v, w)$ successivamente.

Lemma4:

Ipotesi: G grafo pesato

Tesi: assumendo l'esecuzione di *Init*(G, s), ogni sequenza di *Relax*($G(v_0, w_0)$)... *Relax*($G(v_k, w_k)$) ha come invarianti $d[v] \geq \delta_{wg}(s, v)$. Inoltre, appena si ottiene $d[v] = \delta_{wg}(s, v)$, sia $d[v]$ sia $p[v]$ non cambieranno più.

Dimostrazione (induzione):

Caso base ($i = 0$):

Se $w \neq s \Rightarrow d[w] = \infty \geq \delta(s, w)$;

Se $w = s \Rightarrow d[w] = 0 = \delta(s, s)$

Caso induttivo:

$$\begin{aligned}
 d[w_{i+1}] &= d[v_{i+1}] + wg(v_{i+1}, w_{i+1}) \\
 &\geq \delta_{wg}(s, v_{i+1}) + wg(v_{i+1}, w_{i+1}) \quad (\text{per l'ipotesi induttiva, quindi } d[v] \geq \delta_{wg}(s, v)) \\
 &\geq \delta_{wg}(s, w_{i+1}) \quad (\text{per il Lemma2})
 \end{aligned}$$

Corollario2:

Ipotesi: G grafo pesato, $\text{Path}(s, w) = \emptyset$, assumendo $\text{Init}(G, s)$

Tesi: per ogni sequenza di $\text{Relax}(G, (v_0, w_0)) \dots \text{Relax}(G, (v_k, w_k))$ si avrà $d[w] = \delta(s, w) = \infty$

Dimostrazione:

Per il Lemma4, abbiamo sempre che $\infty = \delta(s, w) \leq d[w]$, e pertanto $d[w] = \infty = \delta(s, w)$.

Lemma5:

Ipotesi: G grafo pesato, $\pi = \pi' \cdot vw \in \text{MinWPath}(G, s, w)$. Si assume $\text{Relax}(G, (v_0, w_0)) \dots \text{Relax}(G, (v_k, w_k))$ e che $d[v] = \delta_{wg}(s, v)$ subito prima di $\text{Relax}(G, (v, w))$.

Tesi: dopo $\text{Relax}(G, (v, w))$ si ha $d[w] = \delta_{wg}(s, w)$

Dimostrazione:

Per il Lemma4, se $d[v] = \delta_{wg}(s, v)$ in qualche punto prima del rilassamento dell'arco (v, w) , allora l'uguaglianza vale anche dopo la chiamata. In particolare, dopo il rilassamento dell'arco (v, w) , abbiamo:

$$\begin{aligned} d[w] &\leq d[v] + wg(v, w) && (\text{Per il Lemma3}) \\ &= \delta_{wg}(s, v) + wg(v, w) \\ &= \delta_{wg}(s, w) && (\text{Per il Lemma1}) \end{aligned}$$

16.2.1 Teorema di correttezza

Quando v viene eliminato da Q si ha $d[v] = \delta(s, v)$

Dimostrazione (contraddizione):

Supponiamo esista v (primo nodo d'errore) tale che $d[v] \neq \delta(s, v)$. Se non è uguale, sarà sicuramente maggiore per il Lemma4. Si consideri $\pi \in \text{MinWPath}(G, s, v)$. Per il Corollario2, $\pi \neq \emptyset$. Sappiamo che v non può essere s , quindi π non è banale.

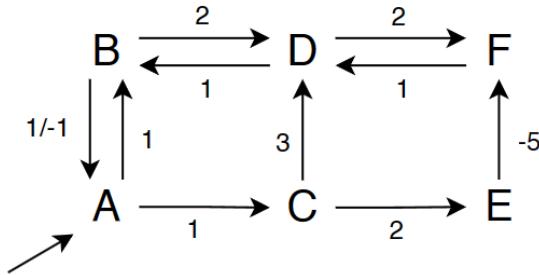
Sia y un nodo in π tale che y ancora appartiene a Q dopo la rimozione di v . Quindi deve esistere un $x \in \pi$ predecessore di y , tale che $x \notin Q$ dopo la rimozione di v . Quindi $\pi = s \dots xy \dots v$ (s e x possono essere potenzialmente uguali). Abbiamo quindi:

$$\begin{array}{c} \pi = \overbrace{s \dots xy \dots v}^{\pi'} \\ \sum \\ \delta_{wg}(\pi) = \sum \end{array}$$

Dunque, $\delta_{wg}(\pi) \geq \delta_{wg}(\pi') = \delta_{wg}(s, y) = d[y]$ che deve essere maggiore o uguale di $d[v]$. Ma $\delta_{wg}(\pi') = \delta_{wg}(s, v)$. Abbiamo così trovato che $\delta_{wg}(s, v) \geq d[v]$ ma avevamo detto che $d[v] > \delta_{wg}(s, v)$, dunque è un assurdo.

16.3 Algoritmo di Bellman-Ford

Partiamo da un esempio, supponiamo che la sorgente sia A. Con questo algoritmo andiamo a considerare anche pesi negativi.



Il procedimento sarà:

<u>0</u>	∞	∞	∞	<u>3</u>	1	3	-2
0	∞	∞	∞	0	1	3	-2
<u>1</u>	1	∞	∞	<u>4</u>	1	-1	-2
0	1	∞	∞	0	1	∞	-2
<u>2</u>	1	4	∞	<u>5</u>	0	-1	-2
0	1*	4	∞	0	1	3	-2
<u>2</u>	1*	3	∞	<u>6</u>	0	-1	-2
0	1	3	∞	-1	1	3	∞

Quando tenta il sesto passaggio ($6 \in n-1$) si accorge che può mettere -1, questo però significa che c'è un ciclo e quindi si ferma. L'algoritmo è il seguente:

```

Function BellmanFord(G, s)
    (d, p) ← Init(G, s)

    loop |V| - 1 times
        for each v ∈ V do
            for each w ∈ Adj[v] do
                Relax(G, (v, w))

    for each v ∈ V do
        for each w ∈ Adj[v] do
            if d[w] > d[v] + wg(v, w) then
                return ⊥

    return (d, p)

```

Facciamo una specifica sulla complessità. Il primo blocco *for each v ∈ V do* costa $\Theta(|E| + |V|)$. Considerando tutto il blocco *loop*, questo costa $\Theta(|V||E| + |V|^2)$. Il secondo blocco *for each* costa $O(|E|)$.

17 Complessità

Vediamo bene come capire se $f(n)$ è $O(g(n))$ o $\Omega(g(n))$ o $\Theta(g(n))$. Fino ad ora abbiamo sempre considerato funzioni positive ($\exists n_0, \forall n \geq n_0$ t.c. $f(n), g(n) > 0$) e monotone crescenti ($\forall n_1, n_2 \geq n_1, n_1 < n_2 \Rightarrow f(n_1) \leq f(n_2), g(n_1) \leq g(n_2)$).

Ricordiamo che:

$$f(n) = \begin{cases} O(g(n)) & \text{se } \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \ f(n) \leq c \cdot g(n) \\ \Omega(g(n)) & \text{se } \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \ c \cdot g(n) \leq f(n) \\ \Theta(g(n)) & \text{se } \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0 \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \end{cases}$$

Andiamo a formalizzare questo concetto tramite un teorema.

Teorema:

Ipotesi: sia $h(n) \triangleq \frac{f(n)}{g(n)}$

Tesi: Se $\lim_{n \rightarrow \infty} (h(n)) \in \mathbb{R}^+$ allora $f(n) = \Theta(g(n))$

Se $\lim_{n \rightarrow \infty} (h(n)) = \infty$ allora $f(n) = \Omega(g(n))$

Se $\lim_{n \rightarrow \infty} (h(n)) = 0$ allora $f(n) = O(g(n))$

Dimostrazione:

Se abbiamo $\lim_{n \rightarrow \infty} (h(n)) = L \in \mathbb{R} \stackrel{\text{def}}{\iff} \forall \epsilon > 0, \exists n_0 \in \mathbb{N} \text{ t.c. } \forall n \geq n_0, |h(n) - L| < \epsilon$.

Ricordiamo che: $|h(n) - L| < \epsilon = -\epsilon \leq h(n) - L \leq \epsilon = L - \epsilon \leq h(n) \leq L + \epsilon$.

Ora, se questo è vero per tutti gli ϵ , sarà vero anche per un particolare ϵ .

$$L - \epsilon \leq \frac{f(n)}{g(n)} \leq L + \epsilon = (L - \epsilon)g(n) \leq f(n) \leq (L + \epsilon)g(n)$$

Fissato ϵ otteniamo i nostri c_1 e c_2 .

Se $L = 0 \Rightarrow (L - \epsilon)g(n)$ è inutilizzabile perché è negativo, mentre $(L + \epsilon)g(n)$ lo consideriamo come il nostro c . Otteniamo così il caso di $O(g(n))$.

Vediamo ora l'ultimo caso rimasto:

$$\lim_{n \rightarrow \infty} h(n) = \infty \stackrel{\text{def}}{\iff} \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N}, \forall n \geq n_0, h(n) > c$$

$$h(n) > c \Rightarrow \frac{f(n)}{g(n)} > c \Rightarrow f(n) > g(n)$$

18 Algoritmi di ordinamento

18.1 Insertion Sort

L'idea di base è simile a quando si organizzano le carte da gioco in mano: si prende una carta alla volta e la si inserisce nella posizione corretta rispetto alle carte già ordinate. Esso è un algoritmo *in place*, cioè ordina l'array senza doverne creare una copia, risparmiando memoria. Pur essendo molto meno efficiente di algoritmi più avanzati, può avere alcuni vantaggi: ad esempio, è semplice da implementare ed è efficiente per insiemi di partenza che sono quasi ordinati.

Input: sequenza \vec{A} di lunghezza n , di elementi di un insieme D totalmente ordinato.
Output: permutazione \vec{A}' di \vec{A} tale che $\forall i, j \in [0, n), i < j \text{ t.c. } \vec{A}'[i] \leq \vec{A}'[j]$.

Si parte dall'elemento alla posizione 1 (considerando che l'elemento alla posizione 0 è già "ordinato" in quanto non ha elementi precedenti con cui confrontarsi). Si confronta l'elemento corrente con gli elementi precedenti nella sequenza ordinata. Se l'elemento è più piccolo, lo si sposta verso la sinistra finché non trova la sua corretta posizione. Si ripete questo processo per ogni elemento della sequenza, espandendo gradualmente la parte ordinata dell'array fino a che tutti gli elementi sono ordinati. L'algoritmo è il seguente:

```
Function InsertionSort(A, n)
  if n>1 then
    InsertionSort(A, n-1)
    Insert(A, n-1)
```

```
Function Insert(A, i)
  x ← A[i]
  j ← i-1

  While (j ≥ 0) ∧ (x < A[j]) do
    A[j+1] ← A[j]
    j ← j-1

  A[j+1] ← x
```

In base a come è fatto l'array A , questo algoritmo ha diverse complessità: se è già ordinato allora è lineare sul numero di elementi di A ; se, invece, è ordinato al contrario, allora la complessità è $O(n^2)$.

Vediamo la versione iterativa:

```
Function InsertionSort(A, n)
  for i = 1 to n-1 do
    Insert(A, i)
```

Calcoliamo il tempo $T_{IS}(A, n)$ di Insertion Sort. Diciamo che il ciclo *for* è $O(n)$ e le esecuzioni di $Insert(A, i)$, $T_{INS}(A, i)$. Dunque:

$$T_{IS}(A, n) = \sum_{i=1}^{n-1} [T_{INS}(A, i) + O(1)] = \sum_{i=1}^{n-1} T_{INS}(A, i) + \sum_{i=1}^{n-1} O(1) = \sum_{i=1}^{n-1} (T_{INS}(A, i)) + \Theta(n)$$

Se A è ordinato:

$$\sum_{i=1}^{n-1} T_{INS}(A, i) = \Theta(1)$$

Se A non è ordinato:

$$\sum_{i=1}^{n-1} T_{INS}(A, i) = \Theta(n)$$

Il caso migliore è quando l'array è già ordinato. Quindi avremo:

$$\sum_{i=1}^{n-1} \Theta(i) = \Theta(n) \Rightarrow T_{INS}^{BEST}(A, n) = \Theta(n)$$

Il caso peggiore è quando l'array è ordinato al contrario. Avremo:

$$\sum_{i=1}^{n-1} \Theta(i) = \Theta\left(\sum_{i=1}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2) \Rightarrow T_{INS}^{WORST}(A, n) = \Theta(n^2)$$

Possiamo concludere dicendo:

$$T_{ISI}^B(A, n) \leq T_{ISI}(A, n) \leq T_{ISI}^W(A, n)$$

Dove $T_{ISI}^B(A, n) = \Theta(n)$

$T_{ISI}(A, n) = \Omega(n) \vee O(n^2)$

$T_{ISI}^W(A, n) = \Theta(n^2)$.

In questo caso specifico, il caso medio si comporta principalmente come il caso peggiore.

Per quanto riguarda l'algoritmo ricorsivo, il tempo è:

$$T_{ISR}(A, n) = \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ \Theta(1) + I_{ISR}(A, n-1) + T_{INS}(A, n-1), & \text{altrimenti} \end{cases}$$

Il caso migliore è:

$$T^B = \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ \Theta(1) + T^B(n-1), & \text{altrimenti} \end{cases}$$

Per il caso peggiore invece:

$$T^B(n) = \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ \Theta(n) + T^W(n-1), & \text{altrimenti} \end{cases}$$

Facendo un'analisi più approfondita del caso migliore, scriviamo $\Theta(1) = c$. Ora, sapendo che:

$$\begin{aligned} T^B(n) &= T^B(n-1) + c \\ T^B(n-1) &= T^B(n-2) + c \\ \vdots &\quad \vdots \\ T^B(2) &= T^B(1) + c \\ T^B(1) &= c \end{aligned}$$

Possiamo dire:

$$\sum_{i=1}^n T^B(i) = \sum_{i=1}^{n-1} [T^B(i) + c] + c = \left[\sum_{i=1}^{n-1} T^B(i) \right] + \left[\sum_{i=1}^n c \right] \rightarrow [\dots] + nc.$$

Quindi:

$$\left[\sum_{i=1}^n T^B(i) \right] - \left[\sum_{i=1}^{n-1} T^B(i) \right] = nc = \Theta(n)$$

Sapendo che $\left[\sum_{i=1}^n T^B(i) \right] - \left[\sum_{i=1}^{n-1} T^B(i) \right] = T(n) \Rightarrow T(n) = \Theta(n)$

Per l'analisi approfondita del caso peggiore, sapendo che:

$$\begin{aligned} T^W(n) &= T(n-1) + cn \\ T^W(n-1) &= T(n-2) + c(n-1) \\ \vdots &\quad \vdots \end{aligned}$$

Possiamo dire:

$$\sum_{i=1}^n T^W(i) = \sum_{i=1}^{n-1} [T^W(i) + c(i+1)] + c = \left[\sum_{i=1}^{n-1} T^W(i) \right] - \left[\sum_{i=1}^{n-1} T^W(i) \right] = \left[\sum_{i=1}^n (c(i+1)) \right]$$

Poichè: $\sum_{i=1}^n (c(i+1)) = c \left[\frac{n(n+1)}{2} + n \right] = \Theta(n^2)$, allora $\left[\sum_{i=1}^{n-1} T^W(i) \right] - \left[\sum_{i=1}^{n-1} T^W(i) \right] = T^W(n) = \Theta(n^2)$

18.1.1 Correttezza

Facciamo la dimostrazione per induzione sul numero di iterazioni. Partiamo dall'idea informale: sia A il nostro array di dimensione n . La nostra ipotesi induttiva è che dopo la i -esima iterazione, $A[i+1]$ è ordinato. È banalmente vero dopo l'iterazione 0 (quindi prima che l'algoritmo parta) perché il primo elemento dell'array $A[1]$ è già ordinato. Poi vedremo che per un qualsiasi k con $0 < k < n$, se l'ipotesi induttiva resta valida per $i = k - 1$, allora vale anche per $i = k$. Quindi se è vero che $A[k]$ è ordinato dopo la $k - 1$ iterazione, allora è vero che $A[k+1]$ è ordinato dopo la k -esima iterazione. Dunque, concluderemo che $A[n]$ (tutto l'array) è completamente ordinato dopo $n - 1$ iterazioni.

Formalmente diciamo: sia A l'array di input di dimensione n .

Ipotesi induttiva: dopo i iterazioni, $A[i+1]$ è ordinato.

Caso base: dopo l'iterazione 0 (quindi prima che l'algoritmo parta), l'array $A[1]$ contiene un solo elemento ed è ordinato.

Passo induttivo: sia k un intero t.c. $0 < k < n$. Supponiamo che l'ipotesi induttiva valga per $k - 1$, così che $A[k]$ sia ordinato dopo la $k - 1$ iterazione. Vogliamo dimostrare che dopo $A[k+1]$ è ordinato dopo la k -esima iterazione.

Supponiamo che $j*$ sia il più grande intero in $\{0, \dots, k-1\}$ t.c. $A[j*] < A[k]$. L'array passerà, durante l'esecuzione, da

$$[A[0], A[1], \dots, A[j*], \dots, A[k-1], A[k]]$$

a

$$[A[0], A[1], \dots, A[j*], A[k], A[j*+1], \dots, A[k-1]]$$

Questo array è ordinato. Perché? Perché $A[k] > A[j*]$ e, per l'ipotesi induttiva, abbiamo $A[j*] \geq A[j]$ per tutte le $j \leq j*$, quindi $A[k]$ è più grande di tutto ciò che si suppone stia prima. Similmente, per la scelta di $j*$ abbiamo che $A[k] \leq A[j*+1] \leq A[j]$ per tutte le $j \geq j*+1$, quindi $A[k]$ è più piccolo di tutto ciò che si suppone stia dopo. Dunque, $A[k]$ è nel posto giusto. Anche tutti gli altri elementi sono nella giusta posizione, quindi la dimostrazione è conclusa.

Conclusione: per induzione, concludiamo che l'ipotesi induttiva regge per tutti $0 \leq i \leq n - 1$. In particolare, questo implica che dopo la fine dell'a $n-1$ -esima iterazione (dopo che l'algoritmo termina) $A[n]$ è ordinato. Poiché $A[n]$ è tutto l'array, questo significa che l'intero array è ordinato quando l'algoritmo termina, che è proprio ciò che volevamo dimostrare.

18.2 Selection Sort

Il *Selection Sort* divide l'array in due sotto-array, uno ordinato e l'altro no, e ad ogni iterazione cerca il massimo elemento nel sotto-array non ordinato e lo mette in ultima posizione. Il codice è il seguente:

```
Function SelectionSort(A,n)
    for i from n-1 down to 1 do
        m ← Max(A,i)
        Swap(A,i,m)
```

```
Function Max(A,i)
    m ← 0
    for j from 1 to i do
        if A[m] < A[j] then
            m ← j

    return m
```

L'operazione di $\text{Swap}(A,i,m)$ costa $\Theta(1)$, mentre la funzione $\text{Max}(A,i)$ costa $\Theta(i)$. Sapendo questo, possiamo dunque dire:

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

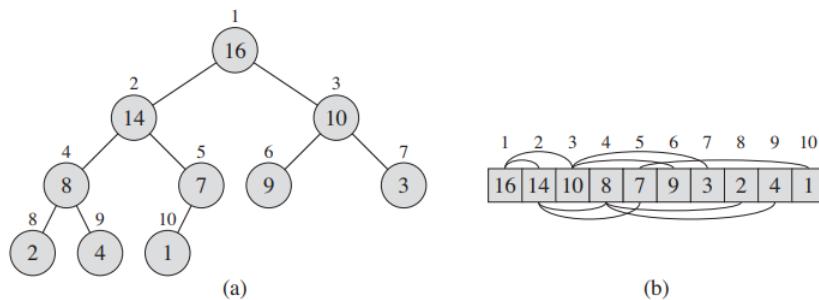
L'andamento dell'algoritmo è sempre lo stesso, indipendentemente dal caso. Questo perché il *Selection Sort* deve sempre trovare il massimo all'interno dell'array, e questo richiede un numero di confronti quadratico nel caso migliore, medio o peggiore.

18.3 Heap Sort

18.3.1 Albero Heap

La struttura dati **heap** è un array di oggetti che possiamo vedere come un quasi completo albero binario. Ogni nodo dell'albero corrisponde a un elemento dell'array. L'albero è completamente pieno in tutti i livelli tranne eventualmente l'ultimo, che viene riempito da sinistra fino a un punto. Per riassumere:

- È un albero binario;
- Tutti i nodi interni tranne al più uno hanno 2 figli;
- Tutte le foglie sono a profondità h o $h - 1$, dove $h = h(T)$



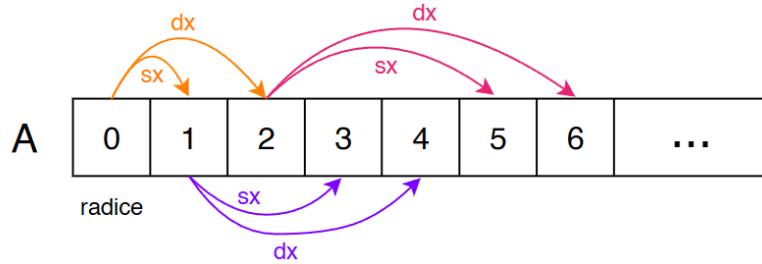
Ci sono due tipi di alberi heap: i max-heap e i min-heap.

18.3.2 Max-Heap

Sia T un max-heap. Allora presenta queste proprietà:

1. È un albero completo;
2. $\forall x \in T, (x.dat \geq x.sx.dat) \wedge (x.dat \geq x.dx.dat)$

Prendiamo questo esempio:

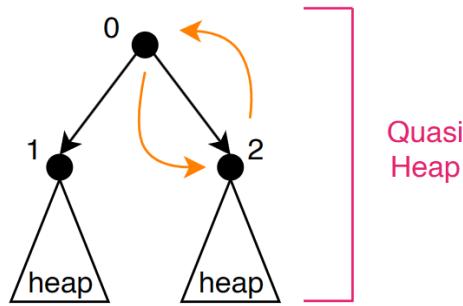


In un vettore, consideriamo un nodo p . Possiamo ottenere gli indici dei figli con:
 $left(p) = 2p + 1$
 $right(p) = 2p + 2$.

Dunque per trovare l'ultimo padre di foglia:

$$\begin{cases} 2p + 1 < n \\ 2p + 2 \geq n \end{cases} \Rightarrow 2p + 1 < n \leq 2p + 2 \Rightarrow p = \lfloor \frac{n}{2} \rfloor - 1$$

Avremo quindi questa situazione:



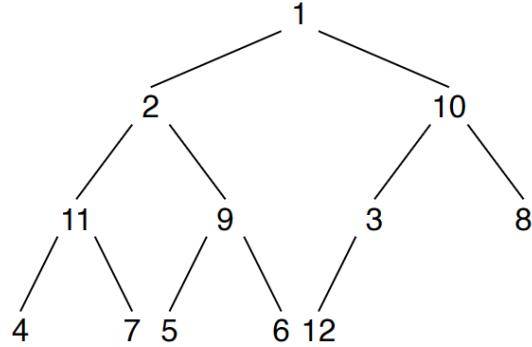
Un quasi-heap è un albero che rispetta la proprietà di heap su tutti i nodi, tranne alla radice. La funzione *Heapify()* ci permette di passare da un quasi-heap ad un heap.

18.3.3 Algoritmo

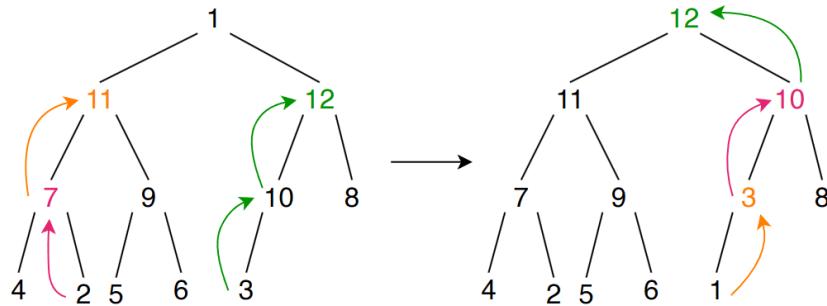
Date queste premesse, possiamo ora vedere effettivamente come funziona l'algoritmo di *Heap Sort()*. Facciamo un esempio. Consideriamo il seguente array:

1	2	10	11	9	3	8	4	7	5	6	12
---	---	----	----	---	---	---	---	---	---	---	----

Come prima cosa andiamo a costruire l'heap:



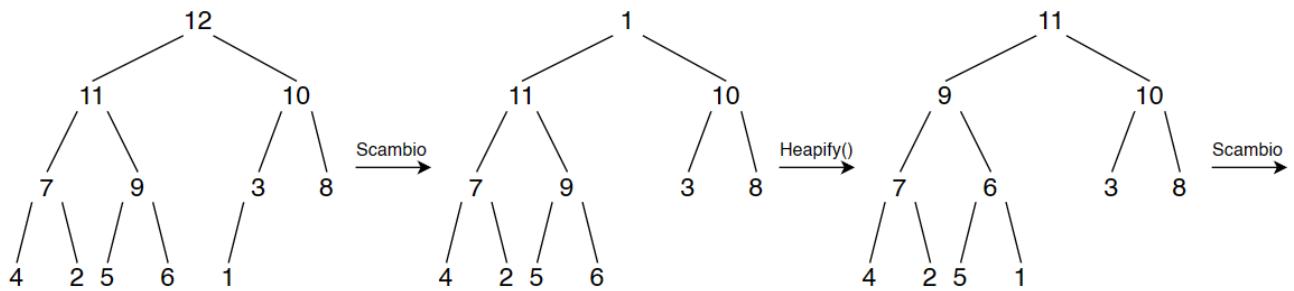
Si tratta di un albero completo, in cui il nodo 3 viene detto *padre di foglia*. Questo però, in particolare, è un quasi-heap. Tramite *Heapify()* andiamo a creare il max-heap facendo i dovuti spostamenti. Quindi:

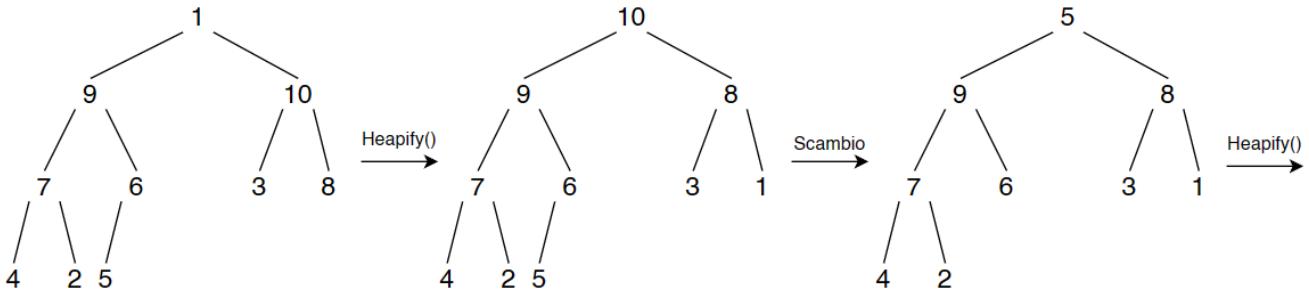


L'array corrispondente sarà:

12	11	10	7	9	3	8	4	2	5	6	1
----	----	----	---	---	---	---	---	---	---	---	---

Dobbiamo ora andare a ordinare in modo crescente. Quindi andiamo a scambiare il primo e l'ultimo elemento (quindi 1 e 12) ed eliminiamo l'ultimo dall'albero. Vediamo se si tratta ancora di un quasi-heap e, in caso affermativo, applichiamo *Heapify()* come prima.





Andando avanti così, l'array diventerà:

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

Vediamo ora com'è fatto l'algoritmo:

```
Function HeapSort(A,n)
    BuildHeap(A,n)
    for i from n-1 down to 1 do
        Swap(A,0,i)
        Heapify(A,i,0)
```

```
Function BuildHeap(A,n)
    for i from ⌊n/2⌋-1 down to 0 do
        Heapify(A,n,i)
```

↑ padre di foglia

```
Function Heapify(A,n,i)
    (m,l,r) ← (i,2i+1,2i+2)

    if (l < n) ∧ (A[m] < A[l]) then
        m ← l

    if (r < n) ∧ (A[m] < A[r]) then
        m ← r

    if m ≠ i then // se il più grande non è la radice
        Swap(A,i,m)
        Heapify(A,n,m)
```

Facciamo ora un'analisi sulla complessità. Partiamo da *BuildHeap()*:

$$\begin{aligned}
\sum_{i=1}^h \frac{n}{2^{i+1}} \cdot i &= \frac{n}{4} \sum_{i=1}^h i \cdot \frac{1}{2^{i-1}} = \frac{n}{4} \sum_{i=1}^h i \cdot \left(\frac{1}{2}\right)^{i-1} = \frac{n}{4} \sum_{i=i}^h [i \cdot x^{i-1}]_{x=\frac{1}{2}} = \frac{n}{4} \left[\sum_{i=1}^h i \cdot x^{i-1} \right]_{x=\frac{1}{2}} = \\
&= \frac{n}{4} \left[\sum_{i=1}^h \frac{d}{dx} x^i \right]_{x=\frac{1}{2}} = \frac{n}{4} \left[\frac{d}{dx} \sum_{i=1}^h x^i \right]_{x=\frac{1}{2}} \leq \frac{n}{4} \left[\frac{d}{dx} \sum_{i=0}^{\infty} x^i \right]_{x=\frac{1}{2}} = \frac{n}{4} \left[\frac{d}{dx} \frac{1}{1-x} \right]_{x=\frac{1}{2}} = \\
&= \frac{n}{4} \left[\frac{d}{dx} (1-x)^{-1} \right]_{x=\frac{1}{2}} = \frac{n}{4} \left[(-1)(1-x)^{-2}(-1) \right]_{x=\frac{1}{2}} = \frac{n}{4} \left[\frac{1}{(1-x)^{-2}} \right]_{\frac{1}{2}} = \\
&= \frac{n}{4} \cdot \frac{1}{(1-\frac{1}{2})^2} = n \Rightarrow O(n)
\end{aligned}$$

Per quanto riguarda *Heapify()*: la lunghezza del percorso dalla radice a qualsiasi foglia in un heap binario completo è logaritmica rispetto al numero di elementi n . Poiché *Heapify()* coinvolge solo scambi tra nodi lungo il percorso dalla radice al nodo interessato, la complessità è $O(\log(n))$.

In conclusione, *HeapSort()* costa $O(n \log(n))$, dato che la chiamata a *BuildHeap()* costa $O(n)$ e ognuna delle $n-1$ chiamate a *Heapify()* costa $O(\log(n))$.

Supponiamo ora di avere un certo max-heap con un nodo i con valore $A[i] = 10$. In questo nodo vogliamo metterci un nuovo valore, cioè 14. Così facendo stiamo andando a cambiare la priorità. Poichè si tratta di un max-heap, dopo questa modifica il problema non starà sotto ma sopra, quindi scambiamo il nodo i con il padre, facciamo ulteriori controlli e vediamo se si può fare altro. Possiamo così considerare questa struttura dati come una sorta di coda a priorità vista nell'algoritmo di Dijkstra. In entrambi i casi, la struttura dati sottostante è spesso un heap (solitamente un min heap per Dijkstra e un max heap per *HeapSort*), poiché permette un'efficiente gestione delle priorità e consente di ottenere le operazioni richieste in tempi logaritmici rispetto alla dimensione della coda. Ovviamente il contesto e le operazioni specifiche eseguite su tali code a priorità sono differenti date le diverse esigenze degli algoritmi.

18.4 Merge Sort

È un algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola. Concettualmente, l'algoritmo funziona nel seguente modo:

- Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:
- La sequenza viene divisa (*divide*) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)
- Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo (*impera*)
- Le due sottosequenze ordinate vengono fuse (*combina*). Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata.

Il codice è il seguente:

```

Function MergeSort(A,n)
    MergeSort(A,0,n-1)

Function MergeSort(A,p,r)
    if p<r then
        q ← ⌊(p+r)/2⌋ //costante
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r) //lineare sulla dimensione su cui lavora

```

```

Function Merge(A,p,q,r)
    (L,R) ← (Copy(A,p,q),Copy(A,q+1,r))
    i,j ← 0

    for k from p to r do //costa lineare sulla dimensione dell'array
        if ((i ≤ q-p) ∧ ((j ≥ r-q) ∨ (L[i] ≤ R[j]))) then
            A[k] ← L[i]
            i ← i+1
        else
            A[k] ← R[j]
            j ← j+1

```

La complessità di questo algoritmo è:

$$T_{MS}(A, n) = \begin{cases} \Theta(1), & \text{se } n \leq 1 \\ 2 \cdot T_{MS}(A, \frac{n}{2}) + \Theta(n), & \text{altrimenti} \end{cases}$$

Possiamo anche omettere Θ in quanto il nostro ragionamento è indipendente da questo o da Ω . Quindi possiamo scrivere anche:

$$T_{MS}(A, n) = \begin{cases} 1, & \text{se } n \leq 1 \\ 2 \cdot T_{MS}(\frac{n}{2}) + n, & \text{altrimenti} \end{cases}$$

$$\begin{array}{lll} \left\{ \begin{array}{ll} T(n) & = 2T(\frac{n}{2}) + n \\ T(\frac{n}{2}) & = 2T(\frac{n}{4}) + \frac{n}{2} \\ T(\frac{n}{4}) & = 2T(\frac{n}{8}) + \frac{n}{4} \\ \vdots & \vdots \\ T(2) & = 2T(1) + 2 \\ T(1) & = 1 \end{array} \right. & \left\{ \begin{array}{ll} T(n) & = 2T(\frac{n}{2}) + n \\ 2T(\frac{n}{2}) & = 4T(\frac{n}{4}) + n \\ 4T(\frac{n}{4}) & = 8T(\frac{n}{8}) + n \\ \vdots & \vdots \\ T(2) & = 2T(1) + 2 \\ T(1) & = 1 \end{array} \right. \end{array}$$

Così facendo i calcoli risultano più semplici.

$$\sum_{i=0}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) = \sum_{i=1}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) + h \cdot n + 1$$

Portiamo la seconda sommatoria al primo membro:

$$\sum_{i=0}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) - \sum_{i=1}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) = h \cdot n + 1$$

Sapendo che $\sum_{i=0}^h (2^i \cdot T(\frac{n}{2^i})) = (2^0 \cdot T(n)) + \sum_{i=1}^h (2^i \cdot T(\frac{n}{2^i}))$, otteniamo:

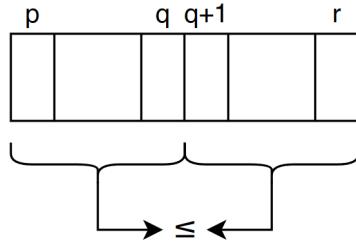
$$(2^0 \cdot T(n)) + \sum_{i=1}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) - \sum_{i=1}^h \left(2^i \cdot T\left(\frac{n}{2^i}\right) \right) = h \cdot n + 1 \implies T(n) = h \cdot n + 1$$

Quando, quindi, la somma raggiunge il caso base? Quando $\frac{n}{2^h} \leq 1 \implies n \leq 2^h \implies h = \log_2(n)$. In conclusione:

$$T(n) = h \cdot n + 1 = n \log_2(n) + 1 = \Theta(n \log(n))$$

18.5 Quick Sort

Si tratta di un algoritmo ricorsivo *in place*. Intuitivamente, viene scelto un cosiddetto "pivot" in modo tale che l'array venga diviso in due parti tali che gli elementi del sotto-array $A[p, q]$ siano minori o uguali degli elementi nel sotto-array $A[q+1, r]$.



Formalizzando, deve valere la seguente proprietà:

$$\forall i, j \in [p, r] \text{ con } i \leq q, q < j, A[i] \leq A[j]$$

Vediamo l'algoritmo:

```

Function QuickSort(A, n)
    QuickSort(A, 0, n-1)

Function QuickSort(A, p, r)
    if p < r then
        q ← Partition(A, p, r)
        QuickSort(A, p, q)
        QuickSort(A, q+1, r)
    
```

La domanda è: perché $p \leq q < r$? Perché, se $q = r$ allora la chiamata $QuickSort(A, q+1, r)$ va bene perché l'array è vuoto, ma non vale lo stesso per la chiamata $QuickSort(A, p, q)$ in quanto andiamo in ricorsione infinita. Se, invece, $q = p - 1$ o inferiore, vale il contrario, quindi la chiamata $QuickSort(A, p, q)$ va bene perché l'array è vuoto ma non vale anche per $QuickSort(A, q+1, r)$ poiché andiamo in ricorsione infinita. Dunque, per la scelta di q siamo sicuri che gli intervalli (p, q) e $(q + 1, r)$ sono sempre strettamente minori di p, r .

Per quanto riguarda il costo dell'algoritmo, possiamo dire che è:

$$T_{QS} = \begin{cases} 1, & \text{se } n \leq 1 \\ T_{QS}(A, k) + T_{QS}(A, n - k) + n, & \text{altrimenti} \end{cases}$$

Il caso peggiore è quando k è una costante. Se dopo $Partition()$ l'array è diviso in parti proporzionali, allora il costo totale è $\Theta(n \log(n))$; se le parti, invece, sono molto diverse, allora il costo è $\Theta(n^2)$.

Da questo possiamo dedurre che è proprio $Partition()$ il cuore di tutto l'algoritmo. Vediamolo:

```

Function Partition(A,p,r)
  x  $\leftarrow$  A[p]
  (i,j)  $\leftarrow$  (p-1,r+1)

  repeat
    repeat
      j  $\leftarrow$  j-1
    until A[j]  $\leq$  x

    repeat
      i  $\leftarrow$  i+1
    until A[i]  $\geq$  x

    if i < j then
      Swap(A,i,j) //scambiamo A[i] con A[j]
  until j  $\leq$  i

return j

```

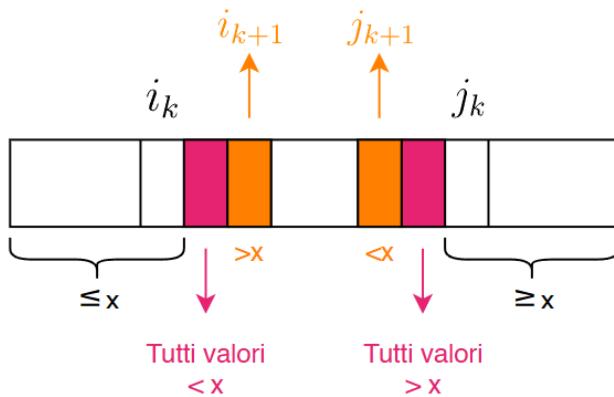
Dimostriamo che $p \leq q < r$. Partiamo dimostrando che $q < r$. All'inizio q parte male perché vale $r + 1$. Però il *repeat-until* vale almeno una volta, quindi ci sarà sicuramente almeno un decremento. Non è che detto che ne faccia un altro. Per vedere che accade basta analizzare cosa fa l'algoritmo, in particolare notiamo che, a questo punto, la condizione dell'*if* è vera, dunque il *repeat-until* esterno non è finito e quindi ricomincia. Ora però dobbiamo dimostrare che non decrementi troppo. Questo *repeat-until* termina quando trova un valore $A[j] \leq x$. Può, la prima volta che viene eseguito, decrementare tante volte da sfornare p ? No. Perché nel momento in cui dovesse fare anche una sequenza di decrementi fino a raggiungere la posizione di p , si troverà poi lì esattamente il valore di x , che fa parte del controllo. Dunque la prima esecuzione non può portare allo sfornamento. Può capitare dopo? Se l'algoritmo dopo ritorna a ripetere questo *repeat-until*, allora significa che è stato eseguito $Swap(A,i,j)$, quindi adesso a sinistra c'è un valore che è o proprio x o più piccolo, e questa è proprio la condizione del ciclo.

Ora dobbiamo dimostrare che $\forall i, j \in [p, r]$ con $i \leq q, q < j$, $A[i] \leq A[j]$. Prendiamo delle istantanee poco prima che venga eseguito il *repeat-until* esterno. Vogliamo trovare che in quel momento vale questa proprietà. Usiamo k come indice di ripetizione (ci indica cioè quante volte è stato eseguito il *repeat-until*). Quello che vogliamo dimostrare è che:

$$\forall k, \forall p \leq i \leq i_k, \forall r \leq j \leq j_k, A[i] \leq x \leq A[j]. \text{ proprietà invariante dell'algoritmo}$$

Per $k = 0$, è vero perché non viene fatto nulla. In particolare, $i = p - 1$ e $j = r + 1$, quindi non abbiamo nulla su cui chiedere qualcosa.

Per $k + 1$ ci troviamo nella situazione in cui:



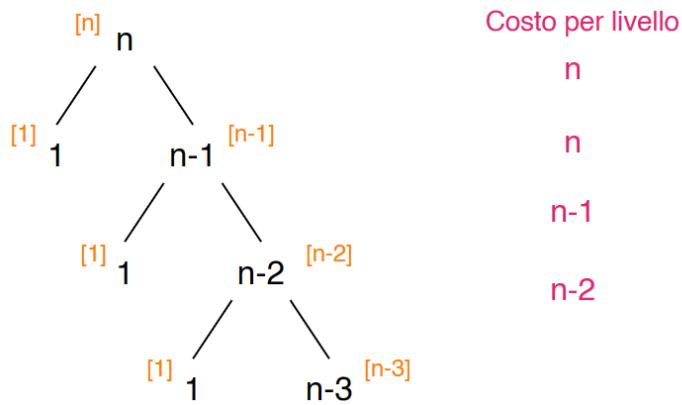
L'algoritmo comincia a decrementare j fino ad arrivare a valori che sono strettamente maggiori di x . Incrementa poi i fino ad arrivare a valori che sono strettamente maggiori di x . Una volta arrivati a questo punto, bisogna fare lo swap, quindi andremo a scambiare i_{k+1} e j_{k+1} . Successivamente la proprietà è banalmente verificata.

Dopo aver dimostrato la correttezza di $\text{Partition}()$, dobbiamo calcolare la complessità: per come è definita la terminazione del ciclo esterno, i e j incrementano e decrementano senza mai superarsi. I due *repeat-until* verranno globalmente ripetuti $n+1$ volte al massimo, quindi sono lineari in n . Inoltre, anche il *repeat-until* esterno si fa tutto l'array. Dunque tutto il ciclo (compreso lo swap) costa $O(n)$.

Dobbiamo ora vedere quanto costa *Quick Sort*. Torniamo a:

$$T_{QS} = \begin{cases} 1 & \text{se } n \leq 1 \\ T_{QS}(A, k) + T_{QS}(A, n - k) + n, & \text{altrimenti} \end{cases}$$

Analizziamo il caso peggiore. Abbiamo detto che avviene quando k è una costante. Per semplicità supponiamo che $k = 1$. L'albero di ricorrenza (vedi sezione successiva) è:



Abbiamo un albero la cui profondità è n (contiamo anche la radice). A sinistra abbiamo sempre 1, mentre a destra scendiamo di 1 fino ad arrivare a 2 (quindi fa $n - 1, n - 2, n - 3, \dots, 2$). Stiamo, a meno della ripetizione di n nel costo totale per livello (che possiamo considerare $n + 1$ per semplicità) e a meno del fatto che alla fine raggiungiamo 2 e non 1, se aggiungiamo 1 sotto e quella ripetizione la consideriamo $n + 1$, allora avremo:

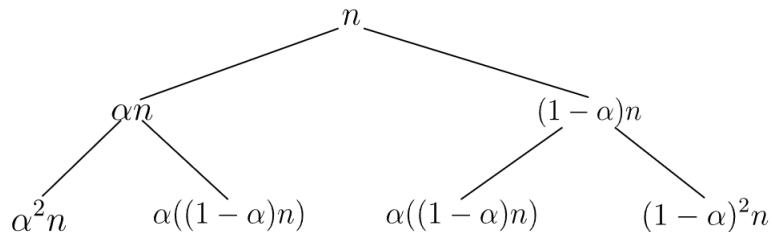
$$\sum_{i=1}^{n+1}(i) = \frac{(n+1)(n+2)}{2} = \Theta(n^2)$$

Non è molto diverso da quello che fa *Insertion Sort()*. Il caso in cui k è costante è quando la sequenza è ordinata o antiordinata.

Il caso medio, invece, abbiamo detto essere il caso in cui la divisione è proporzionale. Avremo quindi:

$$T_{QS} = \begin{cases} 1, & \text{se } n \leq 1 \\ T_{QS}(A, \alpha n) + T_{QS}(A, (1 - \alpha)n) + n, & \text{altrimenti} \end{cases}$$

Supponiamo, per semplicità, che $\alpha \leq \frac{1}{2}$:



Il contributo totale per livello è sempre n . Dobbiamo fare ora il calcolo di h_{min} e h_{max} . Per quanto riguarda il primo:

$$\alpha^{h_{min}} n \leq 1 \implies n \leq \left(\frac{1}{\alpha}\right)^{h_{min}} \implies \log_{\frac{1}{\alpha}}(n) \leq h_{min}$$

I termini sono sempre costanti, quindi basta fare $n \cdot h = n \cdot \log_{\frac{1}{\alpha}}(n)$. Non importa quale sia la base del logaritmo, quello che ci interessa è chè è venuto $n \log(n)$. Facendo lo stesso ragionamento per h_{max} , otterremo:

$$n \log_{\frac{1}{\alpha}}(n) \leq T(n) \leq n \log_{\frac{1}{1-\alpha}}(n)$$

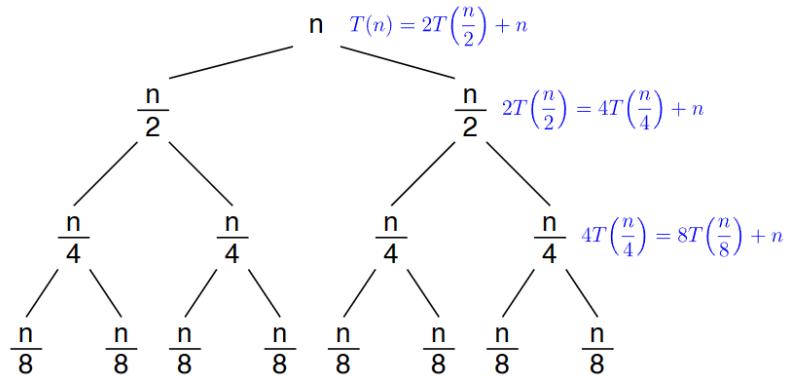
Dunque, in conclusione, $T(n) = \Theta(n \log(n))$.

19 Equazioni di ricorrenza

Lo svolgimento del calcolo della complessità dell'algoritmo *Merge Sort()* è un tipo di calcolo di un'equazione di ricorrenza. Vediamone ora un'altro che sfrutta l'albero di ricorrenza. L'equazione su cui andiamo a lavorare è la seguente:

$$T_{MS}(A, n) = \begin{cases} 1, & \text{se } n \leq 1 \\ 2 \cdot T_{MS}\left(\frac{n}{2}\right) + n, & \text{altrimenti} \end{cases}$$

L'albero di ricorrenza equivalente sarà:



Il prossimo passo è quello di inserire in una tabella tutti i dati necessari.

Livello	#Nodi per livello	Dimensione input	Contributo per nodo	Contributo totale per livello
0	1	n	n	n (per $1 \cdot n$)
1	2	$\frac{n}{2}$	$\frac{n}{2}$	n (per $2 \cdot \frac{n}{2}$)
2	4	$\frac{n}{4}$	$\frac{n}{4}$	n
3	8	$\frac{n}{8}$	$\frac{n}{8}$	n
i	2^i	$\frac{n}{2^i}$	$\frac{n}{2^i}$	n (per $2^i \cdot \frac{n}{2^i}$)

Vediamo ora un altro esempio:

$$T_{MS}(n) = \begin{cases} 1, & \text{se } n \leq 1 \\ 2 \cdot T_{MS}\left(\frac{n}{2}\right) + n^2, & \text{altrimenti} \end{cases}$$

In questo caso, l'albero di ricorrenza è uguale al precedente, in quanto è stato cambiato solo il contributo. Cambieranno però i dati nella tabella, infatti:

Livello	#Nodi per livello	Dimensione input	Contributo per nodo	Contributo totale per livello
0	1	n	n^2	n^2
1	2	$\frac{n}{2}$	$\frac{n^2}{4}$	$\frac{n^2}{2}$
2	4	$\frac{n}{4}$	$\frac{n^2}{16}$	$\frac{n}{4}$
3	8	$\frac{n}{8}$	$\frac{n^2}{64}$	$\frac{2}{8}$
i	2^i	$\frac{n}{2^i}$	$\frac{n^2}{2^{2i}}$	$\frac{n^2}{2^i}$

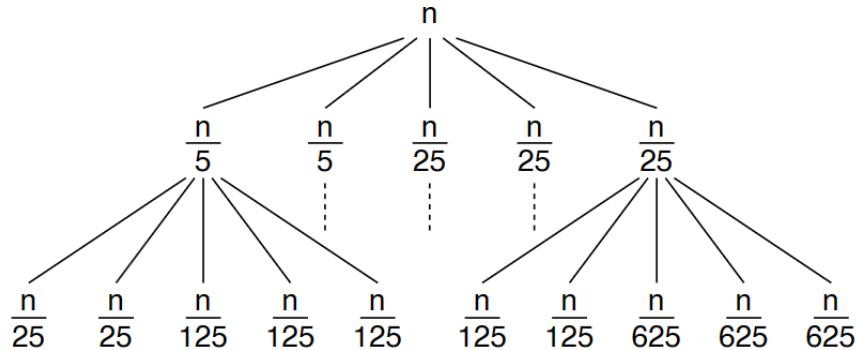
In questo caso, per il calcolo della complessità faremo:

$$\sum_{i=0}^h \frac{n^2}{i} = n^2 \sum_{i=0}^h \left(\frac{1}{2}\right)^i = \text{per somma geometrica} = n^2 \cdot \frac{r^{h+1} - 1}{r - 1} = n^2 \cdot 2 \left(1 - \left(\frac{1}{2}\right)^{h+1}\right) = \\ = n^2 \cdot 2 \left(1 - \frac{1}{2 \cdot 2^h}\right) = n^2 \cdot 2 \left(1 - \frac{1}{2 \cdot 2^{\log_2(n)}}\right) = n^2 \cdot 2 \left(1 - \frac{1}{2 \cdot n}\right)$$

Diventa molto più semplice fare il calcolo se al posto della sommatoria consideriamo la serie. Questa approssimazione, però, si può fare solo quando la ragione è minore di 1.

Vediamo ora un esempio leggermente diverso.

$$T(n) = \begin{cases} 1, & \text{se } n \leq 1 \\ 2T\left(\frac{n}{5}\right) + 3T\left(\frac{n}{25}\right) + 4, & \text{altrimenti} \end{cases}$$



Livello	#Nodi per livello	Dimensione input	Contributo per nodo	Contributo totale per livello
0	1	n	4	4
1	5	$\frac{n}{5} \circ \frac{n}{25}$	4	20
2	25	$\frac{n}{25} \circ \frac{n}{125} \circ \frac{n}{625}$	4	4·25
i	5^i	/	4	$4 \cdot 5^i$

In questo caso l'albero non è pieno, in quanto raggiungiamo il caso base prima a destra (perché scaliamo di 25, mentre a sinistra di 5). Quindi per calcolare il costo totale dobbiamo fare un'approssimazione:

$$\begin{aligned}
\sum_{i=0}^{h_{min}} 4 \cdot 5^i &\leq T(n) \leq \sum_{i=0}^{h_{max}} 4 \cdot 5^i \\
4 \cdot \frac{5^{h_{min}+1} - 1}{5 - 1} &\leq T(n) \leq 4 \cdot \frac{5^{h_{max}+1} - 1}{5 - 1} \\
5^{h_{min}+1} - 1 &\leq T(n) \leq 5^{h_{max}+1} - 1 \\
5 \cdot 5^{\frac{\log_5(n)}{2}} - 1 &\leq T(n) \leq 5 \cdot 5^{\log_5(n)} - 1 \\
5 \cdot \sqrt{n} - 1 &\leq T(n) \leq 5n - 1
\end{aligned}$$

Se applichiamo al risultato ottenuto i teoremi sulla complessità, otteniamo che $T(n) = O(n)$ e $\Omega(\sqrt{n})$.

Vediamo un altro esempio:

$$T(n) = \begin{cases} 1, & \text{se } n \leq 2 \\ n \cdot T(\sqrt{n}) + n^2 & \text{altrimenti} \end{cases}$$

Livello	#Nodi per livello	Dimensione input	Contributo per nodo	Contributo totale per livello
0	1	n	n^2	n^2
1	n	$\frac{1}{n^2}$	n	n^2
2	$n^{\frac{3}{2}}$	$\frac{1}{n^4}$	$\frac{1}{n^2}$	n^2
i	$\frac{2^i - 1}{n^{2^{i-1}}}$	$\frac{1}{n^{2^i}}$	$\frac{1}{n^{2^{i-1}}}$	n^2

Quindi $T(n) = h \cdot n^2$. Dobbiamo calcolare h :

$$\frac{1}{n^{2^h}} \leq 2 \implies \log_2(n^{\frac{1}{2^h}}) \leq 1 \implies \frac{1}{2^h} \log_2(n) \leq 1 \implies \log_2(n) \leq 2^h \implies h \geq \log_2(\log_2(n))$$

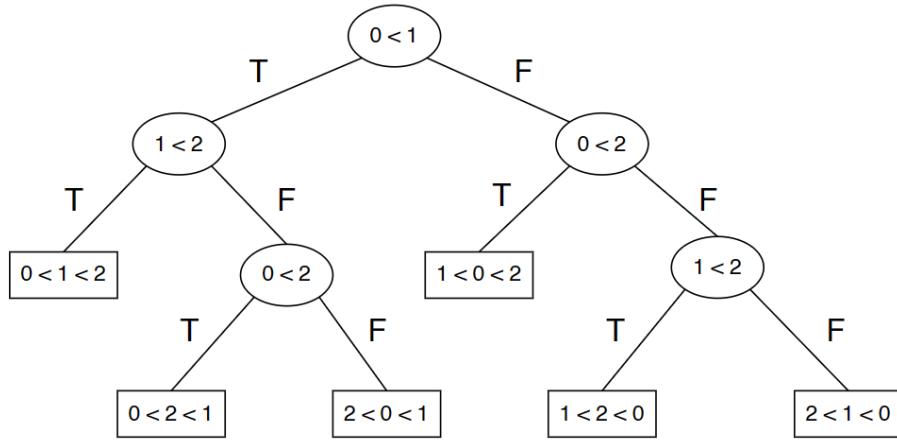
In conclusione, $T(n) = n^2 \log_2(\log_2(n)) \implies \Theta(n^2 \log_2(\log_2(n)))$.

20 Alberi di decisione

Consideriamo un algoritmo di ordinamento basato solo su confronti. Supponiamo di avere il seguente array:

k_0	k_1	k_2
-------	-------	-------

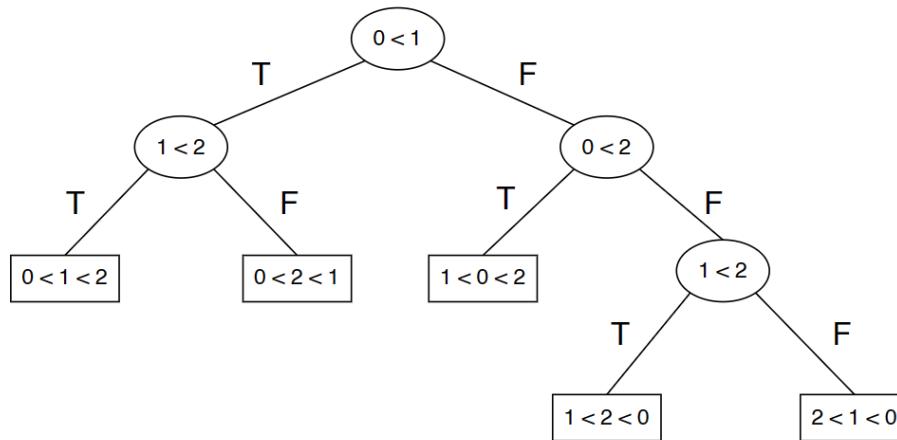
Costruiamo il cosiddetto albero di decisione:



Nei cerchi ci facciamo una domanda, ad esempio alla radice ci chiediamo se $A[0] < A[1]$. Se è vero, andiamo a sinistra, se è falso a destra. Nei quadrati, invece, abbiamo l'output finale. Per semplicità di analisi supponiamo che gli scambi vengano effettuati solo alla fine. All'estrema sinistra c'è il caso migliore, cioè quando l'array è già ordinato; all'estrema destra invece c'è il caso peggiore, cioè quando l'array è ordinato al contrario.

Questo tipo di ragionamento è proprio quello che fa *Insertion Sort()*. È possibile costruire un albero di decisione di un qualsiasi algoritmo di ordinamento basato su confronti.

Ora supponiamo che non ci sia il confronto $A[0] < A[2]$.



Prendiamo in considerazione la permutazione

1	2	0
---	---	---

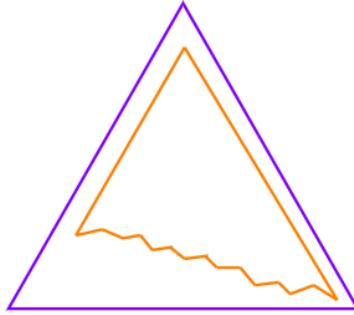
Applicando il ragionamento precedente, otterremo:

1	0	2
---	---	---

ma non va bene. Con questo abbiamo voluto dimostrare che un albero di decisione è binario e con un numero di foglie pari a $n!$. Ci permette di fare anche una stima delle operazioni da effettuare in base all'altezza in quando il

programma segue un solo percorso (è un po' come un algoritmo con soli if, che va solo da una parte e non dall'altra). L'altezza dell'albero non può essere minore di n perché altrimenti non staremo facendo dei confronti.

Come facciamo a dimostrare il caso peggiore nella migliore della possibilità? Vediamo. Quello che dobbiamo fare è stimare l'altezza dell'albero pieno più piccolo che può contenere il nostro albero di decisione.



Sapendo che l'albero di decisione ha $n!$ nodi e l'albero pieno ha 2^h foglie, per come abbiamo costruito l'albero possiamo dire che $n! \leq 2^h$. Ci basterà calcolarla per h e abbiamo fatto.

$$n! \leq 2^h \implies \log_2(n!) \leq h$$

In particolare:

$$\log_2(n!) = \text{per definizione di fattoriale} = \log_2\left(\prod_{i=1}^n i\right) = \text{per proprietà dei logaritmi} = \sum_{i=1}^n \log_2(i)$$

Dunque quello che vogliamo calcolare è:

$$\sum_{i=1}^n \log_2(i) \leq h \leq \sum_{i=1}^n \log_2(i)$$

Lavoriamo prima a destra:

$$h \leq \sum_{i=1}^n \log_2(i) = \log_2(n) \sum_{i=1}^n (1) = n \log(n)$$

Per il lato sinistro invece (leggilo in questo verso \leftarrow):

$$(\log_2(n) - \log_2(2)) \frac{n}{2} = \log_2\left(\lfloor \frac{n}{2} \rfloor\right) \sum_{i=\lceil \frac{n}{2} \rceil}^n 1 = \sum_{i=\lceil \frac{n}{2} \rceil}^n \log_2\left(\lfloor \frac{n}{2} \rfloor\right) \leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \log_2(i) \leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} \log_2(i) + \sum_{i=\lfloor \frac{n}{2} \rfloor}^n \log_2(i) = \sum_{i=1}^n \log_2(i)$$

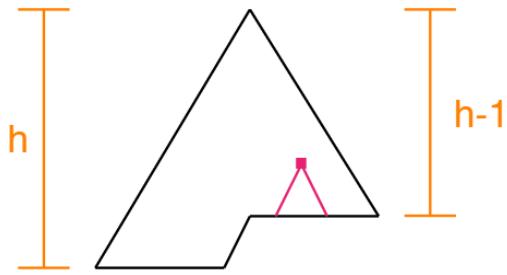
In conclusione, $h = \Theta(n \log(n))$.

Per l'analisi del caso medio dobbiamo prendere la lunghezza di tutti i percorsi e fare la media. La formula è:

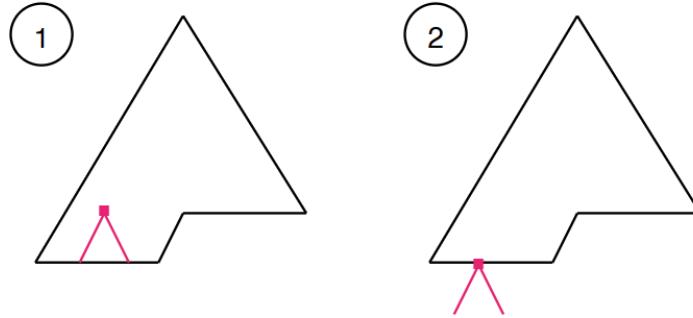
$$T_M(T, n) = \frac{PE(T, n)}{n!}$$

Dove T è la struttura e $PE(T, n)$ è la somma dei percorsi.

L'albero con minimo valore del percorso esterno è quello completo. Quindi, supponiamo di prendere un albero completo con un certo valore di $PE(T, n)$. Se, andando a fare un qualsiasi spostamento, andiamo solo a peggiorare il percorso esterno, allora abbiamo dimostrato quello che volevamo.



Il nodo in figura può essere spostato in due punti:



Nel primo caso, non succede nulla. Per il secondo invece:

$$PE(T, n) - 2(h - 1) + (h - 2) - h + 2(h + 1) = PE(T, n) + 2 \implies \text{è peggiorato}$$

In particolare: $-2(h - 1) + (h - 2)$ è solo per l'eliminazione, mentre $-h + 2(h + 1)$ è per lo spostamento a sinistra, in quanto il nodo ad altezza h viene sostituito dalle due nuove foglie di altezza $h + 1$.

L'unica informazioni che abbiamo dell'albero completo è che abbiamo potuto separare le foglie in due classi: quelle di altezza $h - 1$ e quelle di altezza h , ma non sappiamo quando ce ne sono. Fortunatamente il fatto di essere completo, e quindi di avere un albero pieno incluso (basta togliere le foglie di altezza h) e l'albero pieno che lo include (di cui sappiamo calcolare l'altezza) e sapendo che il numero totale di foglie deve essere $n!$, ci permette di determinare quello che vogliamo. Dobbiamo calcolare il seguente sistema:

$$\begin{cases} N_h + 2(N_{h-1}) = 2^h \\ N_h + N_{h-1} = n! \end{cases}$$

Se andiamo a sottrarre a quella di sopra quella di sotto, otteniamo $N_{h-1} = 2^h - n!$. Se ora, invece, andiamo a sottrarre e moltiplicare per 2 la seconda, otteniamo $N_h = 2n! - 2^h$. Quindi, sapendo l'altezza dell'albero, siamo in grado di determinare quante foglie ci sono a sinistra e quante ce ne sono a destra. Dunque:

$$PE(n) = (h - 1)(2^h - n!) + h(2n! - 2^h) = (h + 1)n! - 2^h$$

In conclusione:

$$T_M(T, n) = \frac{PE(T, n)}{n!} = \frac{(h + 1)n! - 2^h}{n!} = (h + 1) - \frac{2^h}{n!} \approx h = \Theta(n \log(n))$$

Nel caso migliore, $\frac{2^h}{n!}$ è 1.