

Algoritmi e Strutture Dati

Appunti delle lezioni dei professori
Marcello Pelillo e Alessandra Raffaetà

Daniel Andrei Bercu

Università Ca' Foscari, Venezia
Laurea Triennale in Informatica

Anno Accademico 2022-23

Indice

0 Nota al lettore	1
1 Introduzione	2
1.1 Complessità asintotica	2
1.2 Sequenza di Fibonacci	2
1.2.1 Binet - Sezione aurea	2
1.2.2 Algoritmo ricorsivo	4
1.2.3 Primo algoritmo iterativo	9
1.2.4 Secondo algoritmo iterativo	10
1.2.5 Confronto tra gli algoritmi	10
2 Classi asintotiche	11
2.1 Classe O	11
2.2 Classe Ω	12
2.3 Classe Θ - Intersezione tra O e Ω	13
2.3.1 Alcuni esempi di funzioni	14
2.3.2 Proprietà	15
2.4 Classe o	16
2.5 Classe ω	16
2.6 Osservazioni e proprietà sulle classi asintotiche	17
2.7 Polinomi	18
3 Calcolo della complessità di un algoritmo	19
3.1 Istruzioni in sequenza	19
3.2 Costrutto if-then-else	20
3.3 Cicli for	20
3.4 Cicli while	20
3.5 Esempio di algoritmo e calcolo della sua complessità	21
4 Ricorrenze	22
4.1 Metodo dell'iterazione	22
4.2 Metodo della sostituzione	24
5 Teorema Master	25
5.1 <i>Divide et impera</i>	25
5.2 Teorema Master	26
5.3 Dimostrazione	28
6 Strutture Dati	34
6.1 Tipo di dato	34
6.2 Classificazione strutture dati	34

7 Dizionari	35
7.1 Dizionari implementati con array ordinati	35
7.2 Analisi ammortizzata	41
7.3 Dizionari implementati con record e puntatori	42
8 Esercizi sul calcolo della complessità	46
8.1 Esercizio 1	46
8.2 Esercizio 2	47
8.3 Esercizio 3	47
8.4 Esercizio 4	48
8.5 Esercizio 5	49
9 Alberi	50
9.1 Alberi binari e alberi k -ari	51
9.2 Tipo Albero	53
9.3 Alberi implementati con array	53
9.3.1 Vettore padri	53
9.3.2 Vettore posizionale	55
9.4 Alberi implementati con strutture collegate	57
9.4.1 Puntatori ai figli	57
9.4.2 Lista figli	58
9.4.3 Figlio sinistro - Fratello destro	59
9.5 Algoritmi di visita di alberi	60
9.5.1 Visita generica	60
9.5.2 Visita in profondità - Depth-First Search (DFS)	60
9.5.3 Tipi di visita (relative ad alberi binari)	61
9.5.4 Visita in ampiezza - Breadth-First Search (BFS)	62
10 Esercizi d'esame sugli alberi	63
10.1 Nodi centrali	63
10.1.1 Algoritmo di decomposizione	65
10.2 Cammino terminabile	65
10.3 Stampa livello	66
10.4 Raddoppia valori	68
10.5 Discendenti dello stesso colore	69
11 Alberi binari di ricerca	70
11.1 Alberi binari bilanciati	70
11.2 Alberi binari di ricerca	70
11.3 Operazioni su alberi binari di ricerca	71
11.4 Costruzione di alberi binari di ricerca	79
12 Esercizi d'esame sugli alberi binari di ricerca	81
12.1 Conta nodi distinti	81
12.2 Elimina chiavi maggiori di k	83
12.3 Check condition	85

13 Cenni ad ulteriori alberi di ricerca	86
13.1 Alberi AVL	86
13.2 B_Alberi	86
13.3 Alberi rossi e neri	87
14 Problema dell'ordinamento	88
14.1 Insertion Sort	88
14.2 Merge Sort	92
14.3 Quick Sort	95
14.3.1 Quick Sort Randomizzato	100
14.4 Heap e Heap Sort	105
15 Code di priorità	114
15.1 Operazioni su code di massima priorità	114
15.2 Operazioni su code di minima priorità	114
15.3 Code di massima priorità implementate con max_heap	115
16 Esercizi d'esame su ordinamento e varie	118
16.1 Eliminazione nodo max_heap	118
16.2 Trovare il minimo	119
16.3 Unione di intervalli	120
16.4 Somma k	122
16.5 Somma 21	124
17 Ordinamenti lineari	126
17.1 Complessità algoritmi di ordinamento	126
17.2 Counting Sort	130
17.3 Radix Sort	133
18 Tabelle Hash	140
18.1 Tabelle ad indirizzamento diretto	140
18.2 Tabelle Hash	142
18.3 Collisioni	143
18.3.1 Concatenamento	143
18.3.2 Analisi dell'hashing con concatenamento	144
18.3.3 Funzioni hash	146
18.3.4 Indirizzamento aperto	149
18.3.5 Metodi di scansione/ispezione	152
18.3.6 Analisi dell'hashing a indirizzamento aperto	156
18.4 Concatenamento e indirizzamento aperto a confronto	158
18.4.1 Strutture dati a confronto	159
19 Programmazione dinamica	160
19.1 Problema del taglio delle aste	162
19.1.1 Approccio <i>Divide et impera</i>	163
19.1.2 Approccio di programmazione dinamica	166
19.2 Longest Common Subsequence	169

19.3 Elementi di Programmazione Dinamica	175
19.3.1 Confronti	175
20 Grafi	176
20.1 Introduzione	176
20.1.1 Terminologia	177
20.1.2 Rappresentazione di grafi	180
20.1.3 Ulteriore terminologia e approfondimenti	182
20.2 Alberi	195
20.2.1 Alberi di copertura minimi	196
20.2.2 Fatto cruciale degli <i>MST</i>	197
20.2.3 Teorema fondamentale degli <i>MST</i>	198
20.2.4 Corollario del teorema fondamentale degli <i>MST</i>	200
20.2.5 Algoritmi (propedeutici) sugli <i>MST</i>	201
20.2.6 Algoritmo di Kruskal	202
20.2.7 Algoritmo di Prim	204
20.2.8 Esercizio d'esame - Compito del 06/06/2022	206
20.3 Cammini minimi	207
20.3.1 Quattro tipi di problemi	208
20.3.2 Panoramica sugli algoritmi	210
20.3.3 Proprietà dei sottocammini minimi	212
20.3.4 Proprietà	213
20.3.5 Disuguaglianza triangolare	213
20.3.6 Funzioni ausiliarie degli algoritmi	214
20.3.7 Proprietà del limite inferiore (o superiore)	215
20.3.8 Proprietà dell'assenza di cammino	216
20.3.9 Proprietà della convergenza	216
20.3.10 Proprietà del grafo dei predecessori	216
20.3.11 Algoritmo di Dijkstra	217
20.3.12 Dijkstra - Correttezza	221
20.3.13 Dijkstra - Ottimizzazioni	222
20.3.14 Dijkstra - Ulteriori osservazioni ed esercizi	223
20.3.15 Algoritmo di Bellman-Ford	225
20.3.16 Bellman-Ford - Correttezza	227
20.3.17 Bellman-Ford - Esercizio	230
20.3.18 Dijkstra e BF per i cammini tra tutte le coppie	230
20.3.19 Algoritmo di Floyd-Warshall	231
20.3.20 Floyd-Warshall - Correttezza e spiegazione	232
20.3.21 Floyd-Warshall - Esercizio	236
21 Algoritmi <i>Greedy</i>	238
21.1 Problema della selezione delle attività	238
21.2 Struttura comune degli algoritmi <i>greedy</i>	242
21.3 Problema della <i>clique</i> massima	243
21.3.1 Trovare la <i>clique</i> massima	243

22 Teoria della NP completezza	246
22.1 Problemi	246
22.2 Classi di problemi decisionali	249
22.2.1 Classe P	249
22.2.2 Classe NP	249
22.2.3 Classe Co-NP	250
22.2.4 Classe NPC	252
22.2.5 Classe NP-hard	255
22.3 Alcuni problemi NP completi	256
22.3.1 Il problema CIRCUIT-SAT	256
22.3.2 Il problema SAT	257
22.3.3 Forma normale congiuntiva, i problemi 3-SAT e CLIQUE	258
22.4 Cosa fare davanti ad un problema intrattabile	261
A Appendici	262
A.1 <i>Worst-case scenario</i> e algoritmo del simplesso	262

0 Nota al lettore

A settembre 2022 presi la decisione di imparare ad usare quello strumento meraviglioso che è L^AT_EX, e in che modo potei affrontare questa sfida se non trascrivendo le lezioni che stavo seguendo all'università?

Ad oggi, sull'orlo dell'inizio della sessione estiva, questa dispensa è l'unica che è sopravvissuta alla mia irrimediabile indolenza per via di svariati fattori, prima fra tutti sicuramente la passione che mai avrei pensato di provare per un corso universitario.

Il corso di Algoritmi e Strutture Dati per me non è stata solo una mera introduzione all'algoritmica, perché il suo studio è riuscito a darmi qualcosa che, sono sicuro, i professori cercano sempre di trasmettere agli studenti: la consapevolezza delle analogie tra gli argomenti trattati nel corso e i problemi della vita reale, all'infuori dell'informatica.

Trovo ironico il fatto che sia stato proprio questo corso a spiegarmi che nella vita spesso e volentieri bisogna scendere a compromessi (quanto è vero che non si può avere la botte piena e la moglie ubriaca) e meditare le proprie scelte sulla base della condizione in cui ci troviamo in ogni istante, perché ciò che potremmo ritenere giusto in determinate circostanze potrebbe portare a risultati estremamente negativi in altre.

E poi, ahimé, ci sono quei problemi che non si possono risolvere. Uno può provarci, metterci tutto sé stesso, ma non potrà mai sapere se la causa dei suoi fallimenti è dovuta ad una sua mancanza o alla natura intrinsecamente irrisolvibile del problema.

Questa dispensa ha rappresentato un'importante progetto personale e un impegno che ho deciso di prendermi, innanzitutto per me stesso, per quanto riguarda lo studio della materia e l'apprendimento di un nuovo linguaggio di marcatura (spero si noti la differenza della qualità della produzione, confrontando i primi capitoli con gli ultimi), ma anche per tutte le persone che per motivi accademici o personali decidessero di cimentarsi nello studio dell'algoritmica.

In conclusione, voglio porgere un ringraziamento speciale ai professori Marcello Pelillo e Alessandra Raffaetà, che starei ad ascoltare per altre 96 ore, per avermi accompagnato in questo percorso e per aver reso le lezioni quanto più interessanti e avvolgenti possibile.

Se tu che stai leggendo queste pagine non sai cosa sia *Asd complessità + fossa*, significa che avrò finalmente trovato il coraggio di condividere pubblicamente questa dispensa. In tal caso, non ti garantisco l'assenza di errori (ma ti assicuro che, se ce ne sono, sono minimi), e se dovrà sostenere l'esame ti consiglio vivamente di seguire le lezioni, perché fondamentali per la comprensione degli argomenti e perché questo documento non è stato pensato per sostituirle.

Questo documento è stato originariamente caricato al seguente link [GitHub](#).

1 Introduzione

1.1 Complessità asintotica

Un **algoritmo** è una successione di istruzioni volte a risolvere un problema, cioè ad ottenere un preciso risultato a partire da un certo numero di dati iniziali.

Nella stesura di un algoritmo, è importante considerare il tempo che questo impiegherà per essere eseguito. Se indichiamo con n il numero di dati che il nostro algoritmo riceverà in ingresso, dobbiamo saper dire qual è la funzione di n che ne descrive il tempo d'esecuzione per n sufficientemente grande, diremo tendente a infinito. Lo studio del rapporto tra la dimensione dell'input e il tempo d'esecuzione dell'algoritmo prende il nome di analisi della **complessità asintotica**.

Appare evidente che non è possibile garantire una stima ideale tra dimensione dell'input e tempo d'esecuzione, in quanto la singola esecuzione dipenderà in modo univoco dalla natura dell'input: se volessimo trovare un elemento all'interno di un array, l'algoritmo di ricerca impiegherebbe tempi diversi sulla base della posizione dell'elemento cercato: potrebbe trovarsi all'inizio, a metà, alla fine dell'array oppure non essere presente; è utile distinguere tra tre tipi di analisi: *best*, *worst* ed *average case analysis* sono tre tipi di analisi della complessità di un algoritmo che ne riguardano lo studio del tempo d'esecuzione, rispettivamente, nel caso migliore, nel caso peggiore e nel caso medio. Il nostro studio si concentrerà sull'analisi del caso peggiore, al fine di esprimere, in maniera quanto più fedele possibile, il limite massimo per la complessità temporale dei nostri algoritmi.

Spieghiamo quanto appena espresso attraverso l'implementazione di diversi algoritmi che calcolano l' n -esimo numero di Fibonacci.

1.2 Sequenza di Fibonacci

La funzione di Fibonacci è una funzione esprimibile matematicamente nel seguente modo:

$$F(n) = \begin{cases} 1 & n = 1, 2 \\ F_{n-1} + F_{n-2} & n \geq 3 \end{cases}$$

Andiamo a vedere quattro possibili algoritmi con cui implementare questa funzione.

1.2.1 Binet - Sezione aurea

I greci cercarono di trovare un numero x tale che $x^2 = x + 1$. Questo numero si può trovare risolvendo l'equazione di secondo grado $x^2 - x - 1 = 0$, avente come soluzioni:

$$\frac{1 \pm \sqrt{5}}{2}$$

Le due soluzioni costituiscono la sezione aurea Φ e il suo complemento $\hat{\Phi}$. Binet scoprì che possiamo utilizzare queste due costanti per calcolare, con

una certa precisione, i numeri della sequenza di Fibonacci secondo la seguente formulazione:

$$F(n) = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n)$$

Procediamo a dimostrare per induzione il teorema di cui sopra.

- **Casi base**

Se $n = 1$,

$$\begin{aligned} F(1) &= \frac{1}{\sqrt{5}}(\Phi - \hat{\Phi}) = \\ &= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} - \frac{1 - \sqrt{5}}{2} \right) = \\ &= \frac{1}{\sqrt{5}} \frac{2\sqrt{5}}{2} = \frac{\sqrt{5}}{\sqrt{5}} = 1 \end{aligned}$$

Se $n = 2$,

$$\begin{aligned} F(2) &= \frac{1}{\sqrt{5}}(\Phi^2 - \hat{\Phi}^2) = \\ &= \frac{1}{\sqrt{5}} \left(\frac{(1 + \sqrt{5})^2}{4} - \frac{(1 - \sqrt{5})^2}{4} \right) = \\ &= \frac{1}{\sqrt{5}} \left(\frac{1 + 2\sqrt{5} + 5}{4} - \frac{1 - 2\sqrt{5} + 5}{4} \right) = \\ &= \frac{1}{\sqrt{5}} \left(\frac{4\sqrt{5}}{4} \right) = \frac{\sqrt{5}}{\sqrt{5}} = 1 \end{aligned}$$

- **Passo induttivo**

Supponiamo la validità della seguente ipotesi induttiva:

$$\forall k \leq n-1, F_k = \frac{1}{\sqrt{5}}(\Phi^k - \hat{\Phi}^k)$$

Vogliamo dimostrare che, per un qualsiasi $n \geq 3$,

$$F_n = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n)$$

Per definizione,

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} = \\ &= \frac{1}{\sqrt{5}}(\Phi^{n-1} - \hat{\Phi}^{n-1}) + \frac{1}{\sqrt{5}}(\Phi^{n-2} - \hat{\Phi}^{n-2}) = \\ &= \frac{1}{\sqrt{5}}[(\Phi^{n-1} + \Phi^{n-2}) - (\hat{\Phi}^{n-1} + \hat{\Phi}^{n-2})] \end{aligned}$$

La dimostrazione si conclude se riusciamo a dimostrare che:

$$\begin{cases} \Phi^n = \Phi^{n-1} + \Phi^{n-2} \\ \hat{\Phi}^n = \hat{\Phi}^{n-1} + \hat{\Phi}^{n-2} \end{cases}$$

Dividiamo la prima equazione per Φ^{n-2} e la seconda per $\hat{\Phi}^{n-2}$:

$$\begin{cases} \Phi^2 = \Phi + 1 \\ \hat{\Phi}^2 = \hat{\Phi} + 1 \end{cases}$$

Quindi, per definizione, Φ e $\hat{\Phi}$ sono le uniche soluzioni reali di $x^2 = x + 1$ e il teorema di Binet è verificato.

Algoritmo relativo

```

1 int Fib1(int n) {
2     return 1/sqrt(5) * (Phi^n - Phi-hat^n);
3 }
```

Questo algoritmo è dotato di complessità 1, poiché costituito da una sola istruzione aritmetica. Per quanto questo algoritmo sia concettualmente corretto, tuttavia, non può girare su una macchina reale a causa dei problemi di approssimazione dovuti all'utilizzo di numeri floating-point per rappresentare il numero irrazionale $\frac{1}{\sqrt{5}}$. Infatti, vengono di seguito riportati alcuni esempi di input-output del corrente algoritmo:

Input	Output
3	1,999 ≈ 2
16	986,698 ≈ 987
18	2583,1 ≈ 2583

Il risultato per il diciottesimo numero di Fibonacci è incorretto, in quanto dovrebbe essere 2584, ma per via dell'errore di approssimazione nella conversione da floating-point a interi l'algoritmo nella pratica risulta scorretto: può infatti girare solamente su una macchina ideale.

1.2.2 Algoritmo ricorsivo

Implementiamo un algoritmo che, ricorsivamente, calcoli l' n -esimo numero della sequenza di Fibonacci.

```

1 int Fib2(int n) {
2     if (n <= 2)
3         return 1;
4     else
5         return Fib2(n - 1) + Fib2(n - 2);
6 }
```

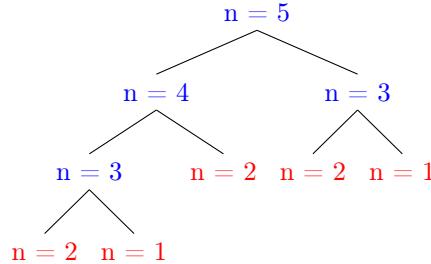
Questo algoritmo non è vulnerabile a errori di approssimazione dovuti alla conversione da floating-point a interi, dato che ha a che fare solo con numeri interi. Ma quanto è efficiente? Qual è la sua complessità?

n	T(n)
1	1
2	1
3	$2 + 1 + 1 = 4$
4	$2 + 4 + 1 = 7$

$$T(n) = \begin{cases} 1 & n = 1, 2 \\ 2 + T(n-1) + T(n-2) & n \geq 3 \end{cases}$$

Essendo l'algoritmo ricorsivo, anche la sua complessità è ricorsiva e si può esprimere attraverso un **albero delle ricorsioni**: una rappresentazione ad albero delle chiamate delle funzioni chiamate ricorsivamente. Per la seguente sezione, indicheremo con T_n l'albero delle ricorsioni relativo alla n -esima chiamata dell'algoritmo, mentre con $T(n)$ alla sua complessità asintotica.

Facciamo un esempio con la chiamata di `Fib2(5)`, rappresentando quindi T_5 :



Sono stati riportati in blu i **nodi interni** ($i(T_n)$), in rosso i **nodi foglia** ($f(T_n)$). La complessità ad ogni chiamata di un nodo foglia è pari ad 1, mentre è 2 per le altre chiamate in quanto la funzione, quando viene passato un input diverso da 1 o 2, esegue sempre 2 istruzioni: un controllo sul valore del numero e la chiamata ricorsiva.

Possiamo elaborare da qui una formulazione per la complessità di questo algoritmo:

$$T(n) = i(T_n) \cdot 2 + f(T_n)$$

dove i è il numero di nodi interni e f è il numero di nodi foglia. Nel nostro esempio, con la chiamata di `Fib2(5)`, avremo una complessità di $4 \cdot 2 + 5 \cdot 1 = 13$.

Possiamo fare delle ulteriori osservazioni sull'algoritmo appena visto:

Proposizione 1 Sia T_n l'albero di ricorsione relativo a `Fib2(n)`. Allora

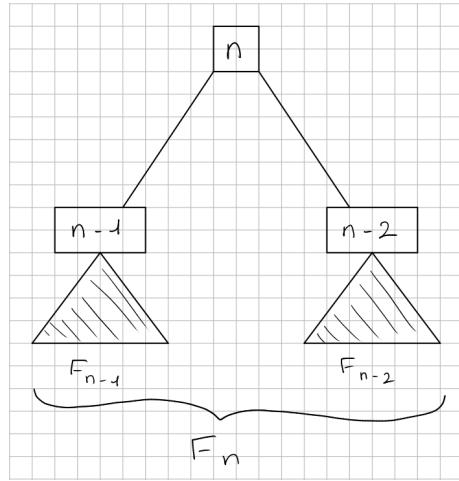
$$f(T_n) = F_n$$

cioè il numero di nodi foglia relativo all'albero T_n è pari all' n -esimo numero di Fibonacci.

Dimostrazione Avviene per induzione su n .

Caso base Prendendo l'albero relativo a $n = 1$ o $n = 2$ ci accorgiamo che questi sono costituiti da un solo nodo che non ha figli; di conseguenza il caso base è verificato.

Passo induttivo Supponiamo $n \geq 3$ e che il principio valga fino alla $(n-1)$ -esima chiamata.



Essendo l'albero relativo alla n -esima chiamata costituito dai due sottoalberi relativi alle chiamate $(n-1)$ -esima e $(n-2)$ -esima, allora il numero totale dei nodi foglia sarà pari alla somma delle foglie dei due sottoalberi, che sappiamo essere pari a $F_{n-1} + F_{n-2}$ per l'ipotesi induttiva, quindi la proposizione è verificata.

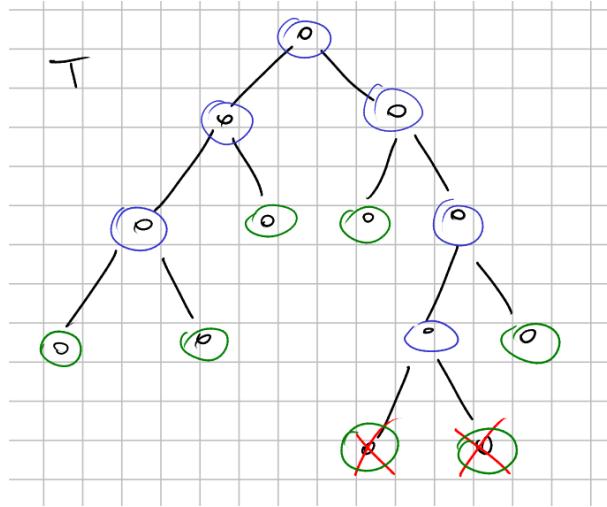
Proposizione 2 Sia T un qualsiasi albero dove i nodi interni hanno esattamente 2 figli (come nel nostro caso). Allora

$$i(T) = f(T) - 1$$

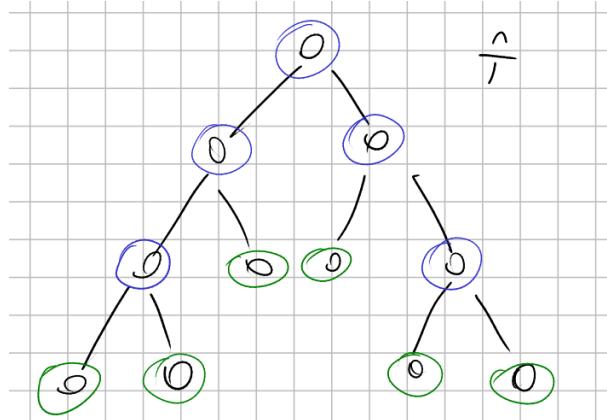
Dimostrazione Avviene per induzione sul numero di nodi di T .

Caso base Nel caso di un albero con solo 1 nodo interno, per ipotesi avremo 2 nodi foglia e quindi il caso base è verificato.

Passo induttivo Poniamo $n \geq 2$ e disegniamo un generico albero T .



Scegliamo arbitrariamente due nodi foglia di T da rimuovere, e otteniamo il nuovo albero \hat{T} :



Applicando l'ipotesi induttiva che $i(\hat{T}) = f(\hat{T}) - 1$, se riusciamo a dimostrare che $i(T) = f(T) - 1$ abbiamo finito la dimostrazione.

Infatti, per ogni coppia di nodi foglia esiste un nodo interno che è il loro padre. \hat{T} è stato costruito rimuovendo da T due nodi foglia, e il padre di questi nodi è diventato a sua volta una foglia. Per ogni coppia di nodi foglia rimossi da un albero, l'albero risultante avrà un nodo foglia e un nodo interno in meno:

$$i(T) = i(\hat{T}) + 1 = (f(\hat{T}) - 1) + 1 = f(\hat{T}) = f(T) - 1$$

Complessità del nostro algoritmo

Avvalendoci delle proposizioni precedenti, possiamo affermare che:

$$\begin{aligned} T(n) &= 2 \cdot i(T_n) + f(T_n) \\ &= 2 \cdot (F_n - 1) + F_n = 3 \cdot F_n - 2 \approx F_n \end{aligned}$$

Ciò significa che partendo da un algoritmo con formulazione ricorsiva, siamo giunti ad esprimere la complessità temporale del suddetto algoritmo attraverso una formula non ricorsiva, elaborando l'approssimazione

$$T(n) \approx F_n$$

Quanto più è alto l'input, tanto più tempo sarà richiesto al nostro algoritmo per elaborare il risultato. Infatti,

- se $n = 8$, $T(8) = 3 \cdot F_8 - 2 = 3 \cdot 21 - 2 = 61$
- se $n = 45$, $T(45) = 3 \cdot F_{45} - 2 = 3.404.709.508$

Proposizione 3

$$\forall n \geq 6, \quad F_n \geq 2^{\frac{n}{2}}$$

Dimostrazione Avviene per induzione su n

Caso base

- Se $n = 6$, abbiamo che $F_6 = 8 \geq 2^{\frac{6}{2}} = 2^3 = 8$.
- Se $n = 7$, abbiamo che $F_7 = 13 \geq 2^{\frac{7}{2}} = \sqrt{128} \approx 11.3$.

Quindi il caso base è verificato.

Passo induttivo Assumiamo l'ipotesi induttiva che $n \geq 8$ e che la proposizione valga per ogni n fino a $n - 1$; avremo:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}} \\ &\dots \\ &\geq 2^{\frac{n}{2}} \left(\frac{1}{\sqrt{2}} + \frac{1}{2} \right) \geq 2^{\frac{n}{2}} \end{aligned}$$

Le tre proposizioni appena dimostrate ci sono servite per affermare che la complessità asintotica di tale algoritmo ricorsivo è esponenziale, cioè il suo tempo d'esecuzione cresce esponenzialmente rispetto alla dimensione dell'input, che è, nel nostro caso, l' n -esimo numero di Fibonacci.

Nei capitoli successivi spiegheremo più dettagliatamente i rapporti tra le diverse funzioni di complessità, stabilendo quali funzioni sono preferibili e quali sono da evitare per un'esecuzione ragionevole dei nostri algoritmi.

1.2.3 Primo algoritmo iterativo

```

1 int Fib3(int n){
2
3     //Istruzione 1
4     int* F = malloc(sizeof(int) * n); //Allocò spazio per un array
5     F di n interi
6
7     //Istruzione 2
8     F[0] = 1; F[1] = 1;
9
10    //Istruzione 3
11    for (int i = 2; i < n; i++){
12
13        //Istruzione 4: corpo del ciclo
14        F[i] = F[i - 1] + F[i - 2];
15    }
16
17    //Istruzione 5
18    return F[n - 1];
19 }
```

Come l'algoritmo precedente, anche questo è formalmente corretto, ma essendo iterativo invece che ricorsivo la sua complessità si può esprimere in questo modo:

$$T(n) = 3 + (n - 1) + (n - 2) = 2n,$$

dove:

- 3: istruzioni 1, 2 e 5, eseguite una volta sola
- $(n - 1)$: istruzione 3, eseguita appunto $(n - 1)$ volte
- $(n - 2)$: istruzione 4, eseguita appunto $(n - 2)$ volte

Questa sarà una delle poche volte in cui calcoleremo esattamente la complessità temporale di un algoritmo in base al numero di istruzioni: d'ora in poi infatti ci concentreremo ad esprimere a grandi linee il rapporto della complessità come funzione del numero di istruzioni (*i.e.* costante, lineare, esponenziale...); infatti, anche se in questo caso la complessità è $2n$, ci limiteremo a dire che la complessità è lineare e possiamo approssimare $2n$ a n , visto che in tempi della CPU la differenza è irrilevante.

Ci basti sapere che, a differenza di quello precedente, questo algoritmo iterativo offre una complessità temporale molto più efficiente:

$$T(45) = 2 \cdot 45 = 90 \ll 3 \cdot F_{45} - 2$$

Vediamo come possiamo migliorare ulteriormente questo algoritmo.

1.2.4 Secondo algoritmo iterativo

```

1 int Fib4(int n){
2
3     //Istruzione 1
4     int a = 1, b = 1, c;
5
6     //Istruzione 2
7     for (int i = 2; i < n; i++) {
8
9         //Istruzione 3
10        c = a + b;
11
12        //Istruzione 4
13        a = b;
14
15        //Istruzione 5
16        b = c;
17    }
18
19    //Istruzione 6
20    return b;
21 }
```

Invece di utilizzare un array contenente n interi, possiamo ridurre la complessità spaziale ad un numero costante di 3 variabili su cui eseguire interamente le operazioni di somma. Tra tutti gli algoritmi che abbiamo visto, questo è sicuramente quello migliore considerando correttezza e complessità temporale e spaziale.

$$T(n) \approx n$$

1.2.5 Confronto tra gli algoritmi

	Corretto?	Complessità temporale	Complessità spaziale
Fib1	NO	Costante	Costante
Fib2	SI	Esponenziale	Lineare
Fib3	SI	Lineare	Lineare
Fib4	SI	Lineare	Costante

2 Classi asintotiche

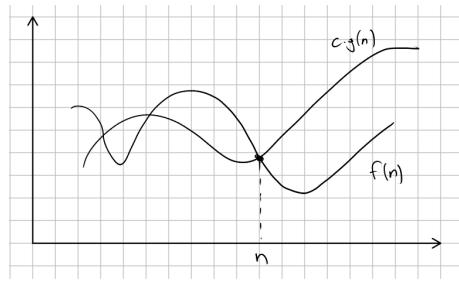
2.1 Classe O

Definizione

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n)\}$$

Notazione: $f(n) = O(g(n))$ (più formalmente: $f(n) \in O(g(n))$).

Significa che, per n sufficientemente grande, $f(n)$ sarà sempre sotto a una costante positiva c moltiplicata per un'altra funzione $g(n)$.



Esempio: $\frac{1}{2}n^2 - 3n = O(n^2)$

$$\begin{aligned} \exists c > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow \frac{1}{2}n^2 - 3n = O(n^2) &\Leftrightarrow \\ \Leftrightarrow \frac{1}{2}n^2 - 3n &\leq c \cdot n^2 \\ \Leftrightarrow \frac{1}{2}n - 3 &\leq c \cdot n \\ \Leftrightarrow \frac{1}{2}n - c \cdot n &\leq 3 \\ \Leftrightarrow \left(\frac{1}{2} - c\right) \cdot n &\leq 3 \end{aligned}$$

Possiamo porre la condizione che $\left(\frac{1}{2} - c\right) \leq 0 \Rightarrow c \geq \frac{1}{2} > 0 \forall n \geq 1$ (posto che $0 \notin \mathbb{N}$), in modo da trovare le seguenti soluzioni:

$$c = \frac{1}{2} \quad n_0 = 1$$

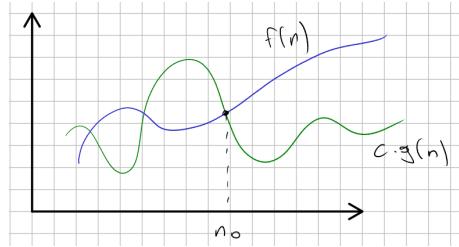
2.2 Classe Ω

Definizione

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow c \cdot g(n) \leq f(n)\}$$

Notazione: $f(n) = \Omega(g(n))$

Significa che, per n sufficientemente grande, $f(n)$ sarà sempre sopra a una costante positiva c moltiplicata per un'altra funzione $g(n)$.



Esempio: $\frac{1}{2}n^2 - 3n = \Omega(n^2)$

$$\begin{aligned} \exists c > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow \frac{1}{2}n^2 - 3n = \Omega(n^2) &\Leftrightarrow \\ \Leftrightarrow c \cdot n^2 &\leq \frac{1}{2}n^2 - 3n \\ \Leftrightarrow c \cdot n &\leq \frac{1}{2}n - 3 \\ \Leftrightarrow n \left(\frac{1}{2} - c \right) &\geq 3 \end{aligned}$$

Supponendo che $\frac{1}{2} - c > 0 \Rightarrow c < \frac{1}{2}$, si ha:

$$n \geq \frac{3}{\frac{1}{2} - c} = \frac{6}{1 - 2c}$$

Scegliamo arbitrariamente $c = \frac{1}{14}$ e otteniamo:

$$n \geq \frac{6}{1 - \frac{2}{14}} = 7$$

Quindi

$$c = \frac{1}{14} \quad n_0 = 7$$

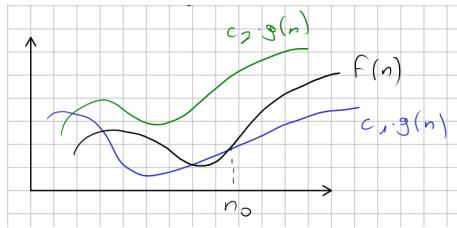
2.3 Classe Θ - Intersezione tra O e Ω

Definizione

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Notazione: $f(n) = \Theta(g(n))$

Significa che, per n sufficientemente grande, $f(n)$ sarà sempre sopra a una costante positiva c_1 moltiplicata per un'altra funzione $g(n)$ ma contemporaneamente sarà sempre sotto a un'altra costante positiva c_2 moltiplicata per $g(n)$.



Proprietà

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

Quindi $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Esempio: $\sqrt{n+10} = \Theta(\sqrt{n})$

$$\begin{aligned} &\exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N} \text{ s.t. } \forall n \geq n_0 \Rightarrow \\ &c_1\sqrt{n} \leq \sqrt{n+10} \leq c_2\sqrt{n} \Leftrightarrow \\ &\Leftrightarrow c_1^2 n \leq n + 10 \leq c_2^2 n \end{aligned}$$

1.

$$c_1^2 n \leq n + 10 \Leftrightarrow (c_1^2 - 1)n \leq 10$$

Poniamo $c_1^2 - 1 \leq 0 \Rightarrow c_1^2 \leq 1 \Rightarrow c_1 \leq 1 \Rightarrow$ la condizione è verificata $\forall n \geq 1$. Scegliamo $c_1 = 1$.

2.

$$n + 10 \leq c_2^2 n \Leftrightarrow n(c_2^2 - 1) \geq 10$$

Se $c_2 > 1$, abbiamo

$$n \geq \frac{10}{c_2^2 - 1}$$

Poniamo arbitrariamente $c_2 = \sqrt{2} > 1$; si ha $n \geq 10$.

Quindi,

$$c_1 = 1 \quad c_2 = \sqrt{2} \quad n_0 = 10$$

2.3.1 Alcuni esempi di funzioni

- $\log n = O(n)$ $\because \log n \leq n - 1 \leq n \Rightarrow c = 1, n_0 = 1$
- $n \log n = O(n^2)$ $\because \forall n \geq 1 : \log n \leq n \Leftrightarrow n \log n \leq n^2 \Rightarrow c = 1, n_0 = 1$
- $n! = O(n^n)$, infatti:

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdots \cdots n \\ &\leq n \cdot n \cdot n \cdots \cdots n \\ &= n^n \end{aligned}$$

- $n! = \Omega(2^n)$, infatti:

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdots \cdots n \\ &\geq 1 \cdot 2 \cdot 2 \cdots \cdots 2 \\ &= 2^{n-1} \end{aligned}$$

Quindi $2^{n-1} \leq n!$ $\forall n \geq 1$, da cui $\frac{1}{2}2^n \leq n!$ $\forall n \geq 1$.
Poniamo $c = \frac{1}{2}, n_0 = 1$.

- $\log n! = O(n \log n)$, infatti:

$$\begin{aligned} \log n! &= \log \prod_{i=1}^n i \\ &= \sum_{i=1}^n \log i \\ &\leq n \log n \end{aligned}$$

Infatti, se $i < n$, $\log i < \log n$ e dato che la sommatoria è di n termini la proprietà è verificata.

- $\sqrt{n} = O(n)$, infatti (banalmente):

$$n < n^2 \quad \forall n \geq 1 \Rightarrow \sqrt{n} \leq n$$

2.3.2 Proprietà

Prima proprietà

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Dimostrazione

Ipotesi: $\exists c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : f(n) \leq c \cdot g(n) \Rightarrow \frac{1}{c} \cdot f(n) \leq g(n)$

Tesi: $\exists c' > 0, \exists n'_0 \in \mathbb{N} \ni \forall n \geq n'_0 : c' \cdot f(n) \leq g(n)$

Basta porre $c' = \frac{1}{c} > 0, n'_0 = n_0 \Rightarrow$ Proprietà verificata

Proprietà transitiva

$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Nota: vale anche per Ω e Θ .

Dimostrazione

Ipotesi 1: $\exists c_1 > 0, \exists n_1 \in \mathbb{N} \ni \forall n \geq n_1 : f(n) \leq c_1 \cdot g(n)$

Ipotesi 2: $\exists c_2 > 0, \exists n_2 \in \mathbb{N} \ni \forall n \geq n_2 : g(n) \leq c_2 \cdot h(n)$

Tesi: $\exists c_3 > 0, \exists n_3 \in \mathbb{N} \ni \forall n \geq n_3 : f(n) \leq c_3 \cdot h(n) ?$

Procediamo nel seguente modo:

$$\begin{aligned} n_3 &= \max(n_1, n_2) \quad \forall n \geq n_3 \\ f(n) &\leq c_1 \cdot g(n) \quad \wedge \quad g(n) \leq c_2 \cdot h(n) \\ f(n) &\leq c_1 \cdot c_2 \cdot h(n) \Rightarrow f(n) \leq c_3 \cdot h(n) \end{aligned}$$

Scegliendo $c_3 = c_1 \cdot c_2$ e $n_3 = \max(n_1, n_2)$ abbiamo verificato la proprietà.

2.4 Classe o

Definizione

$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 \Rightarrow f(n) < c \cdot g(n)\}$$

Osservazione

$$f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$$

Infatti,

$$o(g(n)) \subset O(g(n))$$

Proprietà

$$f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

Questa proprietà è molto utile per la verifica delle classi asintotiche. Infatti, possiamo verificare i rapporti tra due funzioni non solo tramite la definizione di classe, ma anche attraverso l'applicazione di questo limite. Vediamo un esempio.

Esempio: $\log n = O(\sqrt{n})$

$$\lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2}n^{-\frac{1}{2}}} = \lim_{n \rightarrow \infty} \frac{2}{n}\sqrt{n} = \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = 0$$

Quindi, $\log n = o(\sqrt{n}) \Rightarrow \log n = O(\sqrt{n})$.

2.5 Classe ω

Definizione

$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 \Rightarrow c \cdot g(n) < f(n)\}$$

Osservazione

$$f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$$

Infatti,

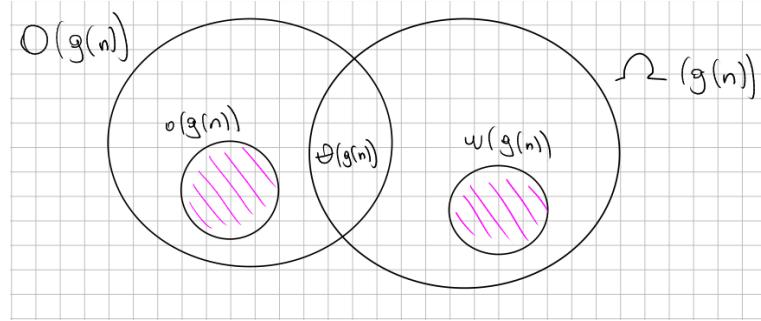
$$\omega(g(n)) \subset \Omega(g(n))$$

Proprietà

$$f(n) = \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

2.6 Osservazioni e proprietà sulle classi asintotiche

1. $o(g(n)) \cap \Omega(g(n)) = \emptyset$
2. $\omega(g(n)) \cap O(g(n)) = \emptyset$



Dimostrazione Avviene per assurdo.

Supponiamo che $\exists f(n) \in o(g(n)) \cap \Omega(g(n))$. Allora,

$$\begin{aligned} \forall c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : f(n) < c \cdot g(n) \\ \exists c' > 0, \exists n'_0 \in \mathbb{N} \ni \forall n \geq n'_0 : c' \cdot g(n) \leq f(n) \end{aligned}$$

Se $n \geq \max\{n_o, n'_o\}$, in modo tale da soddisfare entrambe le ipotesi, si avrebbe:

$$f(n) < c' \cdot g(n) \leq f(n)$$

che è assurdo.

Proposizione

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \ell, \quad \ell \in]0, \infty[\quad \Rightarrow \quad f(n) = \Theta(g(n))$$

Ma non è detto che valga il contrario.

Vediamo una generalizzazione di questa proprietà per le altre classi asintotiche:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Leftrightarrow f(n) = o(g(n)) \Rightarrow f(n) = O(g(n)) \\ \ell, \quad \ell \in]0, \infty[& \Rightarrow f(n) = \Theta(g(n)) \\ \infty & \Leftrightarrow f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n)) \end{cases}$$

Esempio di un caso in cui la proprietà di cui sopra non vale

Prendiamo per esempio l'equazione $(1 + \sin n)n = O(n)$. Poiché $-1 \leq \sin n \leq 1$,

$$\begin{aligned} \forall n \geq 1 : 0 \leq 1 + \sin n \leq 2 &\Leftrightarrow \\ &\Leftrightarrow (1 + \sin n)n \leq 2n \end{aligned}$$

Perciò attraverso la definizione di classe asintotica, abbiamo dimostrato che $(1 + \sin n)n = O(n)$; ma proviamo ora a verificarlo attraverso la proprietà dei limiti:

$$\lim_{n \rightarrow \infty} \frac{(1 + \sin n)n}{n} = \lim_{n \rightarrow \infty} 1 + \sin n,$$

che non esiste.

2.7 Polinomi

Risulta molto facile capire i rapporti di asintoticità quando le nostre funzioni $f(n)$ e $g(n)$ sono due funzioni polinomiali attraverso l'utilizzo dei limiti: posta $f(n)$ una qualsiasi funzione polinomiale, è banale dimostrare che essa sia $\Theta(g(n))$, dove $g(n)$ è un'altra funzione polinomiale dello stesso grado di $f(n)$:

$$f(n) = 3n^3 + 2n^2 + 6n + 5 = \Theta(n^3)$$

$$\lim_{n \rightarrow \infty} \frac{3n^3 + 2n^2 + 6n + 5}{n^3} = \lim_{n \rightarrow \infty} \frac{n^3(3 + \frac{2}{n} + \frac{6}{n^2} + \frac{5}{n^3})}{n^3} = 3$$

Regola dei polinomi

Se $P(n)$ è un polinomio di grado k , allora $P(n) = \Theta(n^k)$.

In notazione matematica,

$$P(n) = \Theta(n^{\deg[P(n)]})$$

Esempi

- $3n^2 + 7n = \Theta(4n^2)$
- $27n^2 + n^2 \log n + \sqrt{n} + \log n^2 = \Theta(n^2 \log n)$

Il secondo di questi esempi può essere dimostrato attraverso una generalizzazione della regola dei polinomi.

Proposizione

$$f(n) + o(f(n)) = \Theta(f(n))$$

Spiegazione e dimostrazione

$$\left. \begin{array}{l} g(n) = f(n) + h(n) \\ h(n) = o(f(n)) \end{array} \right\} g(n) = \Theta(f(n))$$

$h(n) = o(f(n))$ significa che $h(n)$ cresce molto più lentamente di $f(n)$, o, in altre parole, è "dominata" da $f(n)$. Infatti,

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{f(n) + h(n)}{f(n)} = \lim_{n \rightarrow \infty} 1 + \frac{h(n)}{f(n)} = 1$$

Il limite è finito perché il limite del rapporto $\frac{h(n)}{f(n)}$ tende a 0 per definizione di $o(f(n))$.

3 Calcolo della complessità di un algoritmo

Come abbiamo detto all'inizio del documento, dato un algoritmo, dobbiamo saperne determinare la complessità asintotica, $T(n)$, cioè la funzione che determina il tempo d'esecuzione dell'algoritmo in base al numero di istruzioni elementari compiute dallo stesso durante la sua esecuzione, ricevendo in ingresso una grande istanza di input n .

Nell'esempio che abbiamo citato all'inizio di questo documento, cioè la ricerca di un elemento in un array, si sono presentati diversi casi in cui la complessità non è sempre la stessa. Ripetiamoli.

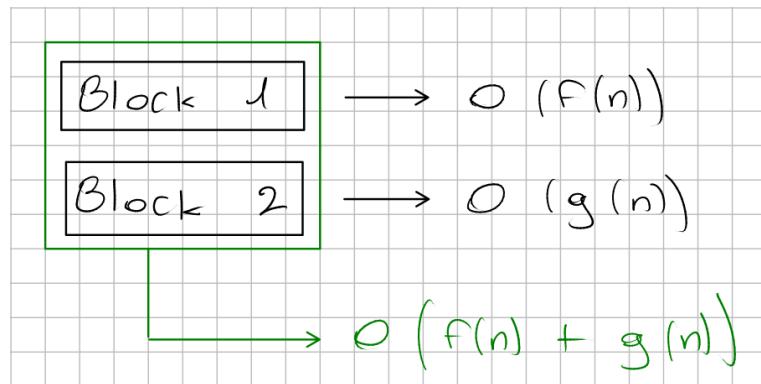
- $T_{best}(n) = 1$ (o comunque un valore costante): questa è la complessità dell'algoritmo nel caso **migliore**, ovvero in cui l'elemento cercato si trova al primo posto nell'array.
- $T_{worst}(n) = n$: questa è la complessità dell'algoritmo nel caso **peggiore**, ovvero in cui l'elemento cercato si trova nell'ultimo posto nell'array.
- $T_{avg}(n) = \frac{n}{2}$: questa è la complessità dell'algoritmo nel caso **medio**, ovvero in cui non sappiamo con esattezza la posizione dell'elemento cercato, ma supponiamo si trovi più o meno a metà; è un caso di studio poco considerato.

Nel nostro studio andremo ad affrontare sempre la ricerca della complessità nel caso peggior.

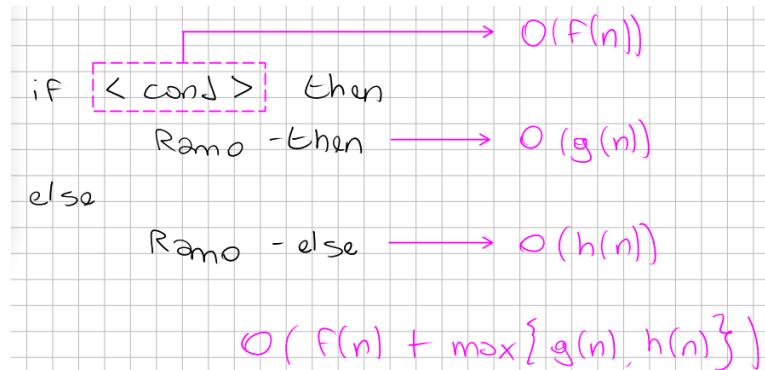
Vediamo come calcolare la complessità di alcuni costrutti molto frequenti nella programmazione.

3.1 Istruzioni in sequenza

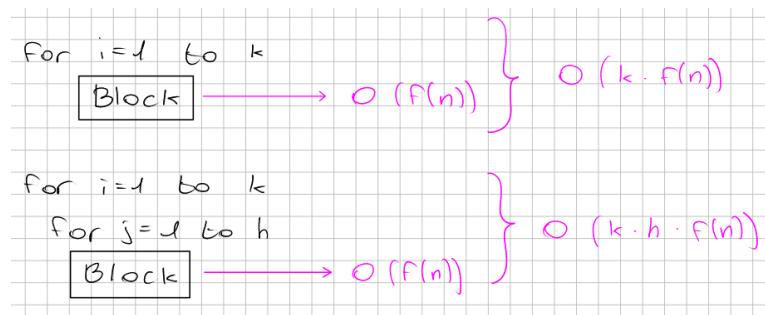
Se ho due blocchi di istruzioni in sequenza, basterà sommare le loro complessità.



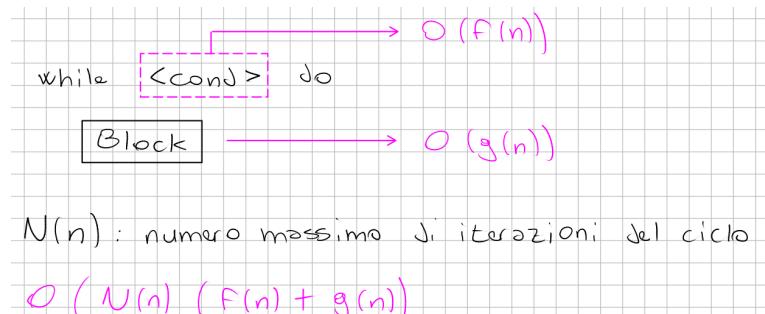
3.2 Costrutto if-then-else



3.3 Cicli for



3.4 Cicli while



3.5 Esempio di algoritmo e calcolo della sua complessità

```

1 int MyAlgorithm (int n) {
2     int a, i, j;
3     if (n > 1) then {
4         a = 0;
5         for (i = 1; i < n; i++) {
6             for (j = 1; j < n; j++) {
7                 a = (a + i) * j + a / 2;
8             }
9         }
10        for (i = 1; i < 16; i++) {
11            a = a + MyAlgorithm(n/4);
12        }
13        return a;
14    }
15    else {
16        return n;
17    }
18 }
```

Per semplificare la nostra indagine e perché irrilevanti rispetto al calcolo della complessità dell'algoritmo, ignoreremo le istruzioni di definizione, dichiarazione, inizializzazione e assegnamento di variabili e le istruzioni di ritorno.

Andiamo a vedere nel dettaglio l'esempio riportato:

- Alla riga 3 troviamo un **if-statement**. Quando ci troviamo in questo contesto, dobbiamo sempre analizzare il caso peggiore, cioè studiare la complessità del ramo contenente il maggior numero di istruzioni. In questo caso, si tratta del ramo **then**, che va dalla riga 3 alla riga 13.
- All'interno di questo ramo, il grosso dei calcoli viene eseguito all'interno di tre cicli **for**, di cui uno innestato in un altro. Analizziamo prima i due cicli innestati (righe 5 - 9), e poi l'ultimo (righe 10 - 12).
- I due cicli da riga 5 a riga 9 prevedono ciascuno n iterazioni, per un totale di n^2 iterazioni; di conseguenza, possiamo dire che la complessità di questo blocco di cicli innestati è pari a n^2 .
- Il ciclo che va da riga 10 a riga 12 presenta una peculiarità: al suo interno fa una chiamata ricorsiva all'algoritmo stesso. Sappiamo per certo che questo ciclo compie 16 iterazioni, ma essendo presente una chiamata ricorsiva, non possiamo dare una stima precisa della complessità di questa chiamata a `MyAlgorithm(n/4)`.
È quindi importante capire che, quando un algoritmo è ricorsivo, allora anche la sua complessità sarà ricorsiva: di conseguenza, possiamo dire che la complessità totale di questo ciclo è pari a $16 \cdot T(n/4)$.

Questo conclude lo studio sulla complessità dell'algoritmo riportato, limitatamente ai costrutti e alle istruzioni che noi riteniamo rilevanti, che possiamo definire come segue:

$$T(n) = O\left(n^2 + 16 \cdot T\left(\frac{n}{4}\right)\right)$$

In realtà, facendo gli opportuni accorgimenti, possiamo arrivare alla più "pulita" conclusione che la complessità totale dell'algoritmo di cui sopra è pari a

$$T(n) = O(n^2 \log n),$$

e ne scopriremo il motivo nella sezione successiva.

4 Ricorrenze

4.1 Metodo dell'iterazione

Vediamo come stimare la complessità di un algoritmo ricorsivo ed esprimerla attraverso una formulazione non ricorsiva attraverso il **metodo dell'iterazione**, una strategia che prevede di "srotolare" le chiamate ricorsive fino a ricondurci al caso base dell'algoritmo.

Prendiamo come esempio la complessità di un algoritmo di ricerca binaria:

Esempio: Caso Ricerca Binaria

$$T(n) = \begin{cases} c + T\left(\frac{n}{2}\right) & \text{se } n \geq 2 \\ 1 & \text{se } n = 1 \end{cases}$$

Sviluppiamo tale ricorrenza col metodo dell'iterazione.

$$\begin{aligned} T(n) &= c + T\left(\frac{n}{2}\right) \\ &= c + \left[c + T\left(\frac{n}{2^2}\right)\right] = 2c + T\left(\frac{n}{2^2}\right) \\ &= 2c + \left[c + T\left(\frac{n}{2^3}\right)\right] = 3c + T\left(\frac{n}{2^3}\right) \\ &= \dots \\ &= kc + T\left(\frac{n}{2^k}\right) \end{aligned}$$

Al k -esimo passo dell'iterazione di "srotolamento" della complessità, la formulazione che otterremo sarà quella di cui sopra.

Cerchiamo ora di convertire la condizione terminale per ricondurci al caso base (per $n = 1$):

$$\frac{n}{2^k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow k = \log_2 n$$

Potremo quindi affermare che:

$$\begin{aligned} T(n) &= c \cdot \log_2 n + T(1) \Rightarrow \\ &\Rightarrow T(n) = c \cdot \log_2 n + 1 \Rightarrow \\ &\Rightarrow T(n) = \Theta(\log n) \end{aligned}$$

La formulazione data vale con il logaritmo di qualsiasi base, non per forza la base 2 utilizzata nell'esempio, per definizione di $\Theta(g(n))$.

Esempio

$$T(n) = \begin{cases} 9T\left(\frac{n}{3}\right) + n & \text{se } n \geq 2 \\ 1 & \text{se } n = 1 \end{cases}$$

Per n sufficientemente grande, infatti,

$$\begin{aligned} T(n) &= 9T\left(\frac{n}{3}\right) + n \\ &= 9\left[9T\left(\frac{n}{3^2}\right) + \frac{n}{3}\right] + n = 9^2T\left(\frac{n}{3^2}\right) + 3n + n \\ &= 9^2\left[9T\left(\frac{n}{3^3}\right) + \frac{n}{3^2}\right] + 3n + 3 = 9^3T\left(\frac{n}{3^3}\right) + 9n + 3n + n \\ &= 9^3T\left(\frac{n}{3^3}\right) + n(3^2 + 3^1 + 3^0) \\ &= \dots \\ &= 9^kT\left(\frac{n}{3^k}\right) + n \sum_{i=0}^{k-1} 3^i \end{aligned}$$

Ricordandoci che

$$\sum_{i=0}^{k-1} q^i = \frac{q^{(k-1)+1} - 1}{q - 1} = \frac{q^k - 1}{q - 1}, \quad q \neq 1$$

possiamo esprimere la seguente formulazione della complessità:

$$T(n) = 9^kT\left(\frac{n}{3^k}\right) + n \frac{3^k - 1}{2}$$

Per le stesse motivazioni fatte per l'esempio precedente, poniamo

$$\frac{n}{3^k} = 1 \Leftrightarrow n = 3^k \Leftrightarrow k = \log_3 n$$

e otteniamo:

$$\begin{aligned} T(n) &= 9^{\log_3 n} \cdot T(1) + n \frac{3^{\log_3 n} - 1}{2} \Rightarrow \\ &\Rightarrow T(n) = (3^2)^{\log_3 n} \cdot T(1) + n \frac{3^{\log_3 n} - 1}{2} \Rightarrow \\ &\Rightarrow T(n) = 3^{\log_3 n^2} \cdot 1 + n \frac{n - 1}{2} \Rightarrow \\ &\Rightarrow T(n) = n^2 + \frac{n(n - 1)}{2} \Rightarrow \\ &\Rightarrow T(n) = \Theta(n^2) \end{aligned}$$

4.2 Metodo della sostituzione

Il **metodo della sostituzione** prevede di supporre la reale complessità di un algoritmo, e di dimostrare la correttezza di tale supposizione attraverso l'induzione. Vediamo in che modo possiamo applicare il metodo della sostituzione.

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

Ricordiamo la definizione di funzione pavimento:

$$\forall x \in \mathbb{R}, \lfloor x \rfloor \leq x \leq \lceil x \rceil$$

E facciamo la seguente ipotesi induttiva: $T(n) = O(n)$? In altre parole,

$$\exists c > 0, \exists n_0 \in \mathbb{N} \ni \forall n \geq n_0 : T(n) \leq c \cdot n ?$$

Dimostriamolo attraverso l'induzione.

Caso base

$$n = 1 \Rightarrow T(1) = 1 \leq c \cdot 1 \quad \text{vero } \forall c \geq 1$$

Passo induttivo

Supponiamo che la proprietà valga fino a $n - 1$:

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq c \cdot \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq c \cdot \frac{n}{2} + n \\ &= \left(\frac{c}{2} + 1\right) \cdot n \end{aligned}$$

che risulta vero se poniamo c tale che $\frac{c}{2} + 1 \leq c$:

$$\frac{c}{2} + 1 \leq c \Leftrightarrow \frac{c}{2} \geq 1 \Leftrightarrow c \geq 2$$

Ponendo $c = 2, n_0 = 1$ la dimostrazione è verificata.

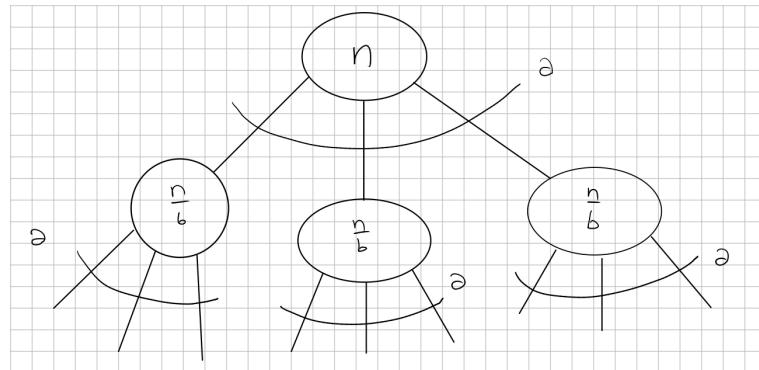
Questo metodo tuttavia può risultare fallace: se nel caso base avessimo ottenuto $c < 2$ e nel passo induttivo avessimo ottenuto $c \geq 2$, non saremmo giunti ad una soluzione. In questo caso potremmo trovare una soluzione cambiando il valore di n_0 (e.g. 2) e rifacendo i calcoli.

5 Teorema Master

5.1 Divide et impera

Esiste una classe di algoritmi ricorsivi che prende il nome di *Divide et impera*, i cui algoritmi sono accomunati da alcuni passi fondamentali che ne caratterizzano la struttura.

1. Divisione del problema iniziale in un numero costante di sottoproblemi; tale operazione prende il nome di *Split*.
2. Risoluzione dei sottoproblemi.
3. Unione delle soluzioni dei singoli sottoproblemi; tale operazione prende il nome di *Merge*.



Dato un algoritmo appartenente alla classe *Divide et impera* che prende inizialmente n dati in ingresso, possiamo riportare di seguito il suo tempo d'esecuzione.

$$T(n) = T_{\text{Split}}(n) + T_{\text{Sub}}(n) + T_{\text{Merge}}(n)$$

Poiché $T_{\text{Split}}(n)$ e $T_{\text{Merge}}(n)$ non sono ricorsive e dipendono unicamente dalla dimensione del problema corrente, e non dei sottoproblemi, possiamo considerarli come un'unico risultato di una generica funzione f della dimensione dell'input, n :

$$T_{\text{Split}}(n) + T_{\text{Merge}}(n) = f(n)$$

Consideriamo anche il fatto che il numero di sottoproblemi generati da ogni iterazione dell'algoritmo è pari ad un numero costante, a , e che se il problema iniziale ha dimensione n e i sottoproblemi hanno dimensione $\frac{n}{b}$, possiamo riscrivere il tempo d'esecuzione dei sottoproblemi come segue:

$$T_{\text{Sub}}(n) = a \cdot T\left(\frac{n}{b}\right)$$

Possiamo ora riscrivere la complessità dell'algoritmo sulla base delle considerazioni precedenti:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad n > 1$$

Le condizioni necessarie per la correttezza dell'algoritmo sono le seguenti:

1. $f(n) \geq 0$, cioè è sempre positiva per n sufficientemente grande.
2. a può essere un numero reale, ma deve essere sempre maggiore o uguale a 1: $a \geq 1$.
3. b può essere un numero reale, ma deve essere sempre maggiore di 1: $b > 1$.

5.2 Teorema Master

Siano date le seguenti definizioni:

$$d = \log_b a, \quad g(n) = n^d$$

Prima di esporre formalmente i tre casi del Teorema Master, per comprendere a pieno il funzionamento di questo metodo, occorre anticipare il fatto che quello che fa il Teorema Master è confrontare due funzioni: una è la $f(n)$ che abbiamo visto appena sopra, cioè l'insieme del tempo di *Split* e di *Merge* dei problemi *Divide et impera*, e l'altra è la $g(n)$ che abbiamo appena introdotto.

Sulla base del rapporto asintotico tra queste due funzioni, il Teorema Master è in grado di determinare un limite per la complessità degli algoritmi di tipo *Divide et impera*, cioè di tutti quei problemi che si presentano nella seguente forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

in cui a , b e $f(n)$ rispettano le condizioni sopracitate.

Vedremo nel dettaglio il significato di $g(n)$ nella dimostrazione del Teorema Master: per ora, ci basti sapere che essa rappresenta il tempo d'esecuzione dei sottoproblemi, e alla luce di ciò dovrebbe sembrare più chiaro che il confronto tra il tempo di risoluzione dei sottoproblemi e il tempo delle operazioni di *Split* e *Merge* ci permette di capire se una delle due funzioni domina sull'altra, e in tal caso, quella che domina determinerà il grosso della complessità dell'algoritmo. Procediamo ad esaminare i tre casi del Teorema Master.

Caso 1

$$\exists \varepsilon > 0 : f(n) = O(n^{d-\varepsilon}) \Rightarrow T(n) = \Theta(n^d)$$

Questo è il caso che vede $f(n)$ crescere meno velocemente di $g(n)$, cioè in cui il tempo di *Split* e *Merge* è asintoticamente minore del tempo di risoluzione dei sottoproblemi.

Questo significa che il tempo d'esecuzione totale dell'algoritmo è approssimativamente dello stesso ordine di grandezza di $g(n)$.

La costante ε che compare in questo caso è un numero reale positivo, che può anche assumere valori infinitamente piccoli.

Caso 2

$$f(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$$

Nel secondo caso, le operazioni di *Split* e *Merge* sono asintoticamente equivalenti alla risoluzione dei sottoproblemi. Il motivo della presenza del logaritmo verrà affrontato nella dimostrazione del teorema.

Caso 3

$$\begin{aligned} \exists \varepsilon > 0 : f(n) = \Omega(n^{d+\varepsilon}) \wedge \exists 0 < c < 1 : \text{per } n \text{ suff. grande } a \cdot f\left(\frac{n}{b}\right) &\leq c \cdot f(n) \\ \Rightarrow T(n) &= \Theta(f(n)) \end{aligned}$$

In modo analogo a quanto avviene nei casi precedenti, qui accade che $f(n)$ è asintoticamente maggiore di $g(n)$, a sostegno del fatto che il tempo d'esecuzione totale dell'algoritmo è caratterizzato, appunto, da $f(n)$.

Valgono anche in questo caso le considerazioni fatte sulla costante ε nella discussione del primo caso del Teorema Master.

Esempio: Caso Ricerca Binaria

$$T(n) = T\left(\frac{n}{2}\right) + c$$

In questo caso,

- $a = 1$
- $b = 2$
- $d = \log_2 1 = 0 \Rightarrow d = 0$
- $f(n) = c$

Siamo nel secondo caso, perché $f(n) = \Theta(n^0) \Rightarrow T(n) = \Theta(\log n)$.

Nota I casi del Teorema Master non sono esclusivi (potremmo non finire in nessuno di questi 3 casi) o il teorema potrebbe non essere sempre applicabile.

Esempio

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

In questo caso non possiamo applicare il Teorema Master, perché il numero di sottoproblemi creati ad ogni chiamata ricorsiva (2^n) non è costante.

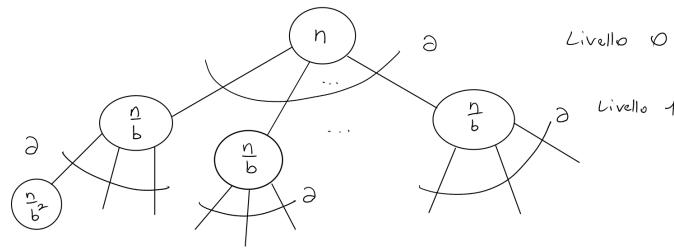
5.3 Dimostrazione

La dimostrazione del Teorema Master si divide in due macro-passaggi: il primo è la trasformazione della formulazione della complessità da ricorsiva a esplicita (non ricorsiva), il secondo è la dimostrazione caso per caso.

Primo step Prendiamo la formulazione ricorsiva della complessità di un algoritmo dello schema *Divide et impera*:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Consideriamo l'albero delle ricorsioni della chiamata iniziale ad un algoritmo di tipo *Divide et impera*, e traiamo alcune considerazioni.



Ogni livello presenta un certo numero di nodi, che rappresentano delle chiamate ricorsive (o, all'ultimo livello, la risoluzione dei casi base) volte a risolvere dei sottoproblemi.

Prendiamo come riferimento il livello 0: qui troviamo un solo nodo, che rappresenta la chiamata iniziale e riceve un problema di dimensione n . Questo nodo creerà un certo numero di sottoproblemi: in particolare, genererà a nodi, ciascuno dei quali risolve un problema di dimensione $\frac{n}{b}$.

Questa operazione si ripete per ogni nodo dell'albero fino al raggiungimento delle foglie, che rappresentano appunto i casi base. Possiamo notare che:

1. Ad un qualsiasi livello $i \geq 0$, il numero di nodi presenti in quel livello è pari ad a^i .
2. Ad un qualsiasi livello $i \geq 0$, la dimensione del problema risolto da ogni nodo è pari a $\frac{n}{b^i}$.
3. Di conseguenza, ad un qualsiasi livello $i \geq 0$, il tempo che ogni nodo impiegherà a risolvere il relativo sottoproblema non è altro che una funzione della dimensione del sottoproblema: $f\left(\frac{n}{b^i}\right)$.

Dunque, il tempo totale sarà pari alla somma delle complessità di ogni livello, data dal prodotto del numero di nodi per ogni livello e il contributo temporale di ogni nodo:

$$T(n) = \sum_{i=0}^{n_{livelli}} a^i \cdot f\left(\frac{n}{b^i}\right)$$

Cerchiamo ora di capire il numero totale di livello dell'albero delle ricorsioni. Occorre sottolineare che la risoluzione effettiva del sottoproblema che indica il caso base, cioè quando $n = 1$, è costante: possiamo infatti scrivere la relazione di ricorrenza che già conosciamo nel seguente modo più esplicito.

$$T(n) = \begin{cases} a \cdot T\left(\frac{n}{b}\right) + f(n) & n \geq 2 \\ c & n = 1 \end{cases}$$

Per capire quanti livelli occuperà l'albero delle ricorsioni del nostro algoritmo, poniamo la seguente condizione per ricondurci al caso base:

$$\frac{n}{b^i} = 1 \Leftrightarrow n = b^i \Leftrightarrow i = \log_b n$$

Quindi, $n_{livelli} = \log_b n$, e possiamo ultimare la trasformazione della relazione di ricorrenza in una formulazione non ricorsiva.

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right)$$

Calcoliamo ora il numero di nodi foglia e successivamente il numero di nodi totali (sono state evidenziate in blu le costanti irrilevanti per quanto riguarda il comportamento asintotico):

$$n_{foglie} = a^{\log_b n} = a^{\log_b a \cdot \log_a n} = (a^{\log_a n})^{\log_b a} = n^{\log_b a} = n^d$$

$$n_{nodi} = \sum_{i=0}^{\log_b n} a^i = \frac{a^{\log_b n+1} - 1}{a - 1} = \frac{a \cdot a^{\log_b n} - 1}{a - 1} = \frac{\textcolor{blue}{a} \cdot n^d - 1}{\textcolor{blue}{a} - 1} = \Theta(n^d)$$

Questi ultimi conti spiegano l'importanza della costante d che viene calcolata nell'applicazione del Teorema Master: n^d rappresenta il numero di nodi foglia, e più in generale è una funzione che determina la dimensione dell'intero albero delle ricorsioni.

Secondo step Procediamo alla dimostrazione del teorema caso per caso.

Caso 1 Per ipotesi, sappiamo che $\exists \varepsilon > 0, f(n) = O(n^{d-\varepsilon})$. Applichiamo tale ipotesi alla complessità totale del livello i -esimo:

$$\begin{aligned} a^i \cdot f\left(\frac{n}{b^i}\right) &= a^i \cdot O\left(\left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \text{ per ipotesi} \\ &= O\left(a^i \cdot \left(\frac{n}{b^i}\right)^{d-\varepsilon}\right) \\ &= O\left(a^i \cdot \frac{n^{d-\varepsilon}}{(b^i)^{d-\varepsilon}}\right) \\ &= O\left(a^i \cdot \frac{n^{d-\varepsilon}}{(b^i)^d (b^i)^{-\varepsilon}}\right) \\ &= O\left(\cancel{a^i} \cdot \frac{n^{d-\varepsilon}}{(\cancel{b^d})^{\cancel{i}} (b^i)^{-\varepsilon}}\right) \\ &= O\left((b^i)^{\varepsilon} \cdot n^{d-\varepsilon}\right) \\ &= O\left((b^\varepsilon)^i \cdot n^{d-\varepsilon}\right) \end{aligned}$$

Calcoliamo ora la sommatoria dei livelli per trovare la complessità totale, servendoci del risultato appena ottenuto:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right) \\ &= \sum_{i=0}^{\log_b n} O\left((b^\varepsilon)^i \cdot n^{d-\varepsilon}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \sum_{i=0}^{\log_b n} (b^\varepsilon)^i\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{(b^\varepsilon)^{\log_b n+1} - 1}{b^\varepsilon - 1}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{b^\varepsilon \cdot (b^\varepsilon)^{\log_b n} - 1}{b^\varepsilon - 1}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{b^\varepsilon \cdot (b^{\log_b n})^\varepsilon - 1}{b^\varepsilon - 1}\right) \\ &= O\left(n^{d-\varepsilon} \cdot \frac{\cancel{b^\varepsilon} \cdot n^\varepsilon - 1}{\cancel{b^\varepsilon} - 1}\right) = O(n^{d-\varepsilon} \cdot n^\varepsilon) = O(n^d) \end{aligned}$$

E per quanto riguarda $T(n) = \Omega(n^d)$? Questo risulta vero perché l'algoritmo deve passare per tutti i n^d nodi foglia, quindi ci vorrà un tempo almeno uguale a n^d .

Di conseguenza,

$$T(n) = \Theta(n^d)$$

Caso 2

$$f(n) = \Theta(n^d) \Rightarrow T(n) = \Theta(n^d \log n)$$

Ci avvaliamo del fatto che

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right)$$

e procediamo con la dimostrazione, prendendo un termine qualsiasi per un generico i appartenente alla sommatoria e vedendo in che modo possiamo sfruttare l'ipotesi per elaborare la sommatoria.

$$a^i \cdot f\left(\frac{n}{b^i}\right) = a^i \cdot \Theta\left(\left(\frac{n}{b^i}\right)^d\right) = \Theta\left(a^i \cdot \frac{n^d}{(b^i)^d}\right) = \Theta\left(a^i \cdot \frac{n^d}{(b^d)^i}\right) = \Theta(n^d)$$

$$T(n) = \sum_{i=0}^{\log_b n} \Theta(n^d) = \Theta\left(\sum_{i=0}^{\log_b n} n^d\right) = \Theta\left(n^d (\log_b n + 1)\right) = \Theta(n^d \log_b n)$$

Caso 3

$$\exists \varepsilon > 0 : f(n) = \Omega(n^{d+\varepsilon}) \wedge \exists 0 < c < 1 : \text{per } n \text{ suff. grande } a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$$

In questo terzo caso, la complessità del primo livello delle nostre chiamate sarà più dispendioso delle complessità dei livelli successivi (ovverosia, la complessità dei sottoproblemi diminuisce man mano che l'algoritmo procede per i livelli inferiori), e questo avviene perché $f(n)$, cioè il tempo necessario alla divisione del problema in sottoproblemi e all'unione delle soluzioni domina sulla risoluzione effettiva dei sottoproblemi.

Vediamo cosa comporta questa condizione nel passaggio da un livello all'altro:

La complessità del livello 1 è minore del livello 0: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$

La complessità del livello 2 è minore del livello 1: $a \cdot f\left(\frac{n}{b^2}\right) \leq c \cdot f\left(\frac{n}{b}\right)$

Moltiplichiamo per a entrambe le parti della disequazione del passaggio dal livello 1 al livello 2:

$$\begin{aligned} a^2 \cdot f\left(\frac{n}{b^2}\right) &\leq c \cdot a \cdot f\left(\frac{n}{b}\right) \\ &\leq c \cdot c \cdot f(n) \\ &= c^2 \cdot f(n) \end{aligned}$$

Possiamo dire ciò perché abbiamo la sicurezza che $c^2 \ll c$, dal momento che $0 < c < 1$.

Si può dimostrare per induzione (e lo verificheremo in seguito) che

$$\forall i \geq 0 : a^i \cdot f\left(\frac{n}{b^i}\right) \leq c^i \cdot f(n)$$

e possiamo utilizzare questa condizione per procedere con la dimostrazione del terzo caso; non faremo infatti uso della prima condizione per dimostrare la correttezza di questo caso, essendo essa diretta conseguenza della proprietà di cui sopra, tuttavia, nell'applicazione del Teorema Master, è buona norma verificare anche la prima per essere sicuri di rientrare nel terzo caso.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right) \\ &\leq \sum_{i=0}^{\log_b n} c^i \cdot f(n) \\ &= f(n) \cdot \sum_{i=0}^{\log_b n} c^i \\ &\leq f(n) \cdot \sum_{i=0}^{\infty} c^i \quad \text{perché } 0 < c < 1 \\ &= f(n) \cdot \frac{1}{1 - c} \end{aligned}$$

Abbiamo potuto eliminare la sommatoria maggiorandola ad una serie geometrica perché sappiamo con certezza che una somma finita di elementi positivi sarà sempre minore ad una somma infinita di elementi positivi.

Grazie a questa osservazione, abbiamo ottenuto il coefficiente $\frac{1}{1-c}$, che sappiamo essere un valore costante e che possiamo quindi ignorare: siamo arrivati alla conclusione che $T(n) = O(f(n))$.

E per quanto riguarda $T(n) = \Omega(f(n))$? Sappiamo che è sicuramente vero, perché

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

dove $a \cdot T\left(\frac{n}{b}\right) \geq 0$, quindi sicuramente $T(n) \geq f(n)$.

Concludiamo confermando che $T(n) = \Theta(f(n))$.

Dimostrazione della condizione di regolarità Abbiamo visto che per il terzo caso del Master Theorem, la condizione di regolarità ci garantisce che la dimensione dei sottoproblemi è inferiore alla dimensione della loro divisione e all'unione delle soluzioni, e quindi che il problema del primo livello domina sui sottoproblemi. Formalmente, vogliamo dimostrare che

$$\forall i \geq 0, a^i \cdot f\left(\frac{n}{b^i}\right) \leq c^i \cdot f(n)$$

Ragioniamo per induzione. Per il caso base, cioè per $i = 0$, avremo che:

$$a^0 \cdot f\left(\frac{n}{b^0}\right) \leq c^0 \cdot f(n) \Rightarrow f(n) \leq f(n)$$

La dimostrazione del caso base per $i = 1$ corrisponde alla seconda ipotesi del terzo caso del Master Theorem: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$.

Passiamo ora al passo induttivo: l'ipotesi induttiva ci garantisce la seguente diseguaglianza.

$$a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \leq c^{i-1} \cdot f(n)$$

Eseguiamo i dovuti conti:

$$\begin{aligned} a^i \cdot f\left(\frac{n}{b^i}\right) &= a \cdot a^{i-1} \cdot f\left(\frac{\frac{n}{b^{i-1}}}{b}\right) \\ &= a^{i-1} \cdot a \cdot f\left(\frac{\frac{n}{b^{i-1}}}{b}\right) \\ &\leq a^{i-1} \cdot c \cdot f\left(\frac{n}{b^{i-1}}\right) \text{ per la condizione 2 di questo caso} \\ &= c \cdot a^{i-1} \cdot f\left(\frac{n}{b^{i-1}}\right) \\ &\leq c \cdot c^{i-1} \cdot f(n) \quad \text{per ipotesi induttiva} \\ &= c^i \cdot f(n) \end{aligned}$$

Come volevasi dimostrare.

6 Strutture Dati

La prossima parte del documento riguarderà le lezioni tenute nel Modulo 2 del corso di Algoritmi e Strutture Dati. Più precisamente, andremo a parlare di tipi di dati, strutture dati e relativi algoritmi, operazioni e implementazioni.

6.1 Tipo di dato

Un **tipo di dato** è un modello matematico che consiste in una collezione di valori sui quali sono ammesse certe operazioni. Possiamo vedere un tipo di dato come una coppia (Valori, Operazioni)

Un tipo di dato specifica **cosa** un'operazione deve fare, ma **non come** questa operazione deve essere realizzata, **né come** gli oggetti della collezione possono essere organizzati in modo che le operazioni siano efficienti e la collezione stessa occupi poco spazio in memoria.

Non è inusuale infatti parlare di tipo di dato **astratto**: astratto perché indipendente dalla rappresentazione che vogliamo dare al tipo di dato in discussione.

Una **struttura dati** è una particolare organizzazione delle informazioni che permette di supportare in modo efficiente le operazioni di un tipo di dato.

Una struttura dati specifica **come** organizzare i dati e **come** realizzare le operazioni definite per il tipo di dato.

6.2 Classificazione strutture dati

Le strutture dati possono essere classificate sulla base di tre criteri.

1. Disposizione dei dati

- **Lineare:** I dati sono disposti in sequenze (è possibile identificare il primo elemento, il secondo, ecc...) (*e.g.* array, liste, pile, code...)
- **Non lineare:** Non è individuata una sequenza (*e.g.* alberi, grafi...)

2. Numero

- **Statiche:** Il numero di elementi della struttura dati è costante nel tempo (*e.g.* array...)
- **Dinamiche:** Il numero di elementi della struttura dati può variare nel tempo (*e.g.* liste, alberi, code, pile...)

3. Tipo

- **Omogenee:** I dati sono tutti dello stesso tipo
- **Non omogenee:** I dati non sono tutti dello stesso tipo

7 Dizionari

Un **dizionario** rappresenta il concetto matematico di relazione univoca, definita da un certo dominio D ad un certo codominio C ($R : D \rightarrow C$). Gli elementi appartenenti al dominio prendono il nome di **chiavi**, quelli appartenenti al codominio prendono il nome di **valori**.

Possiamo quindi vedere il dizionario come un insieme di associazioni di tipo (CHIAVE, VALORE).

Dati Un insieme S di coppie (CHIAVE, VALORE).

Operazioni

- **search(Dizionario S, Chiave K) → Elem U {NIL}**
 - Nessuna precondizione: posso sempre fare l'operazione di **search**
 - Post-condizione: restituisce il valore associato alla chiave K se presente in S , NIL altrimenti (indica che la chiave non è presente nel dizionario)
- **insert(Dizionario S, Elem v, Chiave K)**
 - Post-condizione: associa il valore v alla chiave K aggiungendo la coppia al dizionario se non presente, aggiornando il valore relativo alla chiave K se presente
- **delete(Dizionario S, Chiave K)**
 - Pre-condizione: K deve essere presente in S
 - Post-condizione: cancella da S la coppia avente chiave K

7.1 Dizionari implementati con array ordinati

I dati di questa realizzazione sono contenuti in un array, che chiameremo A , di dimensione n contenente un record con due campi (**Key**, **Info**) ordinati in base a **Key**.

Scegliamo di utilizzare, per tale rappresentazione, un array ordinato per avere la possibilità di implementare l'operazione di **search()** attraverso il concetto di ricerca binaria.

$$A \quad [4, -7 \mid 6,9 \mid 8, -4 \mid 12,11 \mid 24, 6 \mid 30, 7]$$

Abbiamo a nostra disposizione l'attributo **A.length** che contiene la dimensione dell'array.

Il costo in termini di spazio di questa implementazione sarà la seguente:

$$S(n) = \Theta(n),$$

cioè sarà linearmente proporzionale al numero di elementi del dizionario.

Operazioni

Nota D'ora in poi i codici riportati saranno scritti in pseudocodice, per garantire una maggiore leggibilità senza essere vincolati alle limitazioni sintattiche del linguaggio C utilizzato finora.

1. search

```

1  search(Dizionario A, Key K)
2      i = search_index(A, K, 1, A.length)
3      if i == -1
4          return NIL
5      else
6          return A[i].info

1  search_index(Dizionario A, Key K, int p, int r) -> int
2      if r < p      //In questo modo si specifica quando un
                   //vettore e' vuoto.
3          return -1
4      else
5          med = (p+r)/2
6          if A[med].Key == K
7              return med
8          else
9              if A[med].Key > K
10                 return search_index(A, K, p, med - 1)
11             else
12                 return search_index(A, K, med + 1, r)

```

Spiegazione e osservazioni

- L'algoritmo si divide in due funzioni: una chiamante, `search()`, e una funzione ausiliaria ricorsiva, `search_index()`.
La `search()` serve funge da wrapper per la `search_index()`, che viene chiamata nella prima istruzione e sulla base del risultato di questa chiamata la `search()` ritornerà o il valore relativo alla chiave data (se presente) o il valore `NIL`.
- Post-condizione di questo algoritmo: se K è presente in A , restituisce l'indice della cella dove K si trova, altrimenti restituisce -1.
Si tratta di un algoritmo che segue i principi del concetto di *Divide et Impera*, che abbiamo già incontrato.
 - 1) Operazioni *Divide*: riga 5
 - 2) Operazioni *Impera*: righe 3, 6-7, 9-12
 - 3) Operazioni *Combina*: qui non c'è una vera e propria fase *Combina*: il nostro algoritmo sceglie solo una delle due vie possibili
Concretamente, questo algoritmo di ricerca binaria riceve 4 parametri: il dizionario su cui effettuare la ricerca, la chiave da cercare, l'indice da cui inizia la ricerca (p) e l'indice in cui termina (r). Alla prima chiamata di `search_index()`, quella che avviene nella `search()`, gli

indici saranno quello iniziale e quello finale dell'array, mentre nelle chiamate ricorsive uno dei due indici verrà calcolato sulla base della metà tra `p` ed `r`, `med`: se la chiave che abbiamo nella posizione `med` è maggiore di quella che stiamo cercando, sappiamo che dovremo cercarla nella parte sinistra dell'array, quindi la chiamata verrà effettuata indice di partenza `p` e indice di termine `med - 1`, perché (se esiste) si trova sicuramente in quella metà dell'array; al contrario, se la chiave alla posizione `med` è minore di quella che stiamo cercando, la ricerca alla chiamata successiva avverrà nella parte destra dell'array, e verranno passati come parametri `med + 1` e lo stesso `r`.

- Attenzione a non calcolare la media dei due indici come $\frac{(r-p)}{2}$: facendo in questo modo finiremo per cercare la chiave sempre nella parte sinistra dell'array, anche quando non si dovesse trovare qui.

Complessità e ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T\left(\frac{n}{2}\right) + \Theta(1) & \text{se } n > 0 \end{cases}$$

$\Theta(1)$ è dovuto alle operazioni costanti che facciamo attraverso istruzioni aritmetiche, controlli delle condizioni degli `if-statement`, ecc...

$T\left(\frac{n}{2}\right)$ si deve invece alle chiamate ricorsive effettuate nel caso peggiore della ricerca (*worst case scenario*), in cui viene chiamata la funzione su metà della lunghezza della chiamata precedente.

Volendo applicare il Master Theorem, avremo le seguenti condizioni:

$$\begin{aligned} a &= 1, \quad b = 2 \quad \Rightarrow \quad \log_b a = \log_2 1 = 0 \quad \Rightarrow \quad n^{\log_b a} = n^0 = 1 \\ f(n) &= 1 = \Theta(1) \\ n^{\log_b a} &= \Theta(f(n)) \quad \Rightarrow \quad n^0 = 1 \end{aligned}$$

Siamo nel secondo caso, quindi possiamo stabilire con certezza la complessità della funzione `search_index()` nel seguente modo:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Per calcolare la complessità della funzione `search()`, invece, basta sommare alla complessità di `search_index()` quella delle istruzioni rimanenti, che risultano avere complessità $\Theta(1)$:

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

2. insert

```

1  insert(Dizionario A, Info v, Key K)
2      i = 1
3      while i <= A.length and A[i].Key < K
4          i = i + 1
5      if i <= A.length and A[i].Key == K
6          A[i].Info = v
7      else
8          reallocate(A, A.length + 1)
9          A.length = A.length + 1
10         for j = A.length down to i + 1
11             A[j] = A[j - 1]
12         A[i].Key = K
13         A[i].Info = v

```

Spiegazione ed osservazioni

- La `insert()` è un'operazione che inserisce in `A` la coppia `(K, v)` o aggiorna il valore della chiave `K` se già presente.
- L'algoritmo procede nel seguente modo: un intero scorre la lunghezza dell'array per trovare l'indice del record avente chiave `K`. Se trovato, imposta il relativo valore dell'informazione a `v`. Se non trovato, l'algoritmo si preoccupa di riallocare l'array in un'altra area di memoria, assegnandogli la lunghezza dell'array precedente + 1 (dovuta all'aggiunta dell'elemento).
- Il ciclo `for` delle righe 10-11 si occupa dello *shifting* a destra degli elementi dell'array a partire dal primo elemento avente chiave maggiore di quella da inserire (dal momento che quando viene trovata questa chiave il flusso del programma esce dal ciclo `while` delle righe 3-4 e non entra nell'`if-statement` delle righe 5-6).
- L'istruzione alla riga 9 aggiorna il campo contenente la lunghezza dell'array, qualora la funzione `reallocate()` non lo faccia già di suo; la `reallocate()` è una funzione che prende come parametri un array (o un puntatore) e la nuova lunghezza che gli si vuole assegnare, crea un nuovo array con la dimensione voluta e copia il contenuto delle celle di interesse dal vecchio al nuovo spazio.
Con "celle di interesse" intendiamo il numero di celle pari al valore minimo tra la vecchia dimensione e la nuova dimensione: infatti, una `reallocate()` può non solo aumentare la dimensione dell'array, ma anche ridurla.
Un approfondimento sulla `reallocate()` verrà fatto dopo aver discusso anche della funzione `delete()`.

Complessità e ricorrenze

L'algoritmo non è ricorsivo, quindi la sua complessità si può esprimere algebricamente come somma delle complessità dei singoli blocchi di istruzioni.

- Il ciclo **while** delle righe 3-4 viene eseguito i volte, e il suo corpo ha un costo temporale pari a d : in tutto il corpo del **while** avrà un costo di $i \cdot d$.
- La **reallocare()** ha un costo di $O(n)$.
- Il ciclo **for** delle righe 10-11 viene eseguito $(n+1)-(i+1)+1 = n-i+1$ volte, e il suo corpo ha un costo temporale pari a d : in tutto il corpo del **for** avrà un costo di $(n-i+1) \cdot d$.
- Le restanti istruzioni hanno complessità costante, che possiamo raggiungere come $O(1)$

Tirando le somme, possiamo esprimere la complessità dell'algoritmo come segue:

$$\begin{aligned} T(n) &= O(1) + i \cdot d + O(n) + (n - i + 1) \cdot d \\ &= O(1) + n \cdot d + d + O(n) \\ &= O(n) \end{aligned}$$

Chiaramente il tempo di esecuzione varia di caso in caso sulla base degli input, ma dal momento che l'algoritmo non può interagire con un numero di elementi maggiore degli n input, possiamo affermare con certezza che n rappresenta il limite superiore della complessità, e da qui $T(n) = O(n)$.

3. **delete**

```

1 delete(Dizionario A, Key K)
2     i = search_index(A, K, 1, A.length)
3     for j = i to A.length - 1
4         A[j] = A[j + 1]
5     reallocate(A, A.length - 1)
6     A.length = A.length - 1

```

Spiegazione ed osservazioni

- L'algoritmo di eliminazione di un record dall'array suppone la precondizione che la chiave K sia presente nell'array.
- Facciamo uso della funzione **search_index()** utilizzata per l'algoritmo di ricerca binaria per trovare la chiave da eliminare dall'array. Successivamente, il ciclo **for** delle righe 3-4 si occupa dello *shifting* a sinistra degli elementi dell'array avente chiave maggiore di quella da eliminare, e infine riallochiamo la memoria dell'array togliendogli una posizione e riassegnando manualmente la nuova lunghezza come nel caso dell'algoritmo precedente.

Complessità e ricorrenze

Come nel caso precedente, anche qui la complessità si può calcolare come somma delle singole complessità dei blocchi di istruzioni.

- Abbiamo già calcolato la complessità di `search_index()`: $\Theta(\log n)$.
- Il ciclo `for` delle righe 3-4 viene eseguito $(n - 1) - i + 1 = n - i$ volte, e il corpo delle istruzioni ha un costo temporale arbitrario che chiamiamo d ; l'intero ciclo costerà quindi $(n - i) \cdot d$.
- Come abbiamo già visto, la funzione `realloc()` ha un costo di $O(n)$.

Possiamo quindi esprimere la complessità dell'algoritmo nel modo seguente:

$$T(n) = \Theta(\log n) + (n - i) \cdot d + O(n) = O(n)$$

Per i motivi di cui sopra, anche qui sceglieremo di adottare la casistica peggiore, cioè quella in cui l'elemento da eliminare si trova nella prima posizione: in questo modo, la funzione `realloc()` impiegherà il massimo del tempo possibile, appunto $O(n)$.

L'algoritmo potrebbe metterci anche meno tempo ad essere eseguito, ad esempio nel caso in cui il record con chiave K si trovi in ultima posizione, cosicché la complessità risulterebbe $\Theta(\log n)$ dovuta alla completa esecuzione della `search_index()`.

Anche qui n rappresenta il limite superiore del tempo di esecuzione.

Bonus. `realloc`

La riallocazione della memoria di un array avviene secondo la **tecnica del raddoppiamento e del dimezzamento**: si mantiene un array di dimensione h , dove per ogni $n > 0$, h soddisfa il seguente invarianto:

$$n \leq h < 4n$$

Le prime n celle dell'array contengono gli elementi della collezione, mentre il contenuto delle altre celle è indefinito.

- Inizialmente, quando $n = 0$, poniamo $h = 1$.
- Ogni qualvolta n supera h , l'array viene riallocato raddoppiando la dimensione ($h = 2h$).
- Ogni qualvolta n scende a $\frac{h}{4}$, l'array viene riallocato dimezzandone la dimensione ($h = \frac{h}{2}$)

Spazialmente, l'algoritmo avrà questa complessità:

$$S(n) = \Theta(h) = \Theta(n),$$

poiché lo spazio è limitato superiormente e inferiormente da n .

Conviene utilizzare questo metodo perché, effettuando molteplici operazioni di `insert` e/o `delete` andando a manipolare la dimensione dell'array, avere più memoria a disposizione ammortizza il costo totale delle operazioni.

7.2 Analisi ammortizzata

L'analisi ammortizzata ci permette di calcolare il costo medio dell'esecuzione di m operazioni su una struttura dati.

Assumiamo di partire da un vettore inizialmente di dimensione 1, e vogliamo eseguire n `insert`. Sia C_i il costo dell' i -esimo inserimento:

$$C_i = \begin{cases} i & \text{se } \exists k : i = 2^k + 1 \\ 1 & \text{altrimenti (maggior parte dei casi)} \end{cases}$$

C_i sarà uguale ad i quando avrà effettuato $2^k + 1$ operazioni di inserimento, riempiendo quindi lo spazio precedentemente allocato e rendendo necessaria una riallocazione degli i elementi. Poniamo $C(n)$ il costo di tutti gli inserimenti e calcoliamolo:

$$\begin{aligned} C(n) &= \sum_{i=1}^k C_i = n \text{ (inserimenti puri)} + \sum_{k=0}^{\log_2 n} 2^k \text{ (copia)} \\ &= n + \frac{2^{\log_2 n+1} - 1}{2 - 1} = n + 2n - 1 \leq 3n \end{aligned}$$

Quindi,

$$C(n) \leq 3n \Rightarrow \frac{C(n)}{n} \leq \frac{3n}{n} = 3 \text{ che è costante} \Rightarrow T_{amm}(n) = O(1)$$

Possiamo concludere che, grazie a questo metodo, il costo medio di un'operazione $\left(\frac{C(n)}{n}\right)$ è di complessità costante.

7.3 Dizionari implementati con record e puntatori

I dizionari si possono implementare anche attraverso una collezione L di n record contenenti le seguenti informazioni:

- Una chiave **Key**
- L'informazione **Info**
- Un puntatore al record successivo **next**
- Un puntatore al record precedente **prev**

Un attributo **L.head** punta al primo elemento della lista. La lista vuota viene codificata con la seguente condizione: **L.head == NIL**.

In termini di spazio, la complessità spaziale sarà la seguente: $S(n) = \Theta(n)$.

Operazioni

1. **insert**

```

1 insert(Dizionario L, Info v, Key K)
2     creo un nuovo record p con valore v e chiave K
3     p.next = L.head
4     if L.head != NIL
5         L.head.prev = p
6     p.prev = NIL
7     L.head = p

```

Spiegazione ed osservazioni

L'algoritmo consiste in una semplice aggiunta di un elemento in testa ad una lista doppiamente concatenata, con la sistemazione dei puntatori che tale operazione richiede.

A differenza della **insert()** dell'implementazione con gli array, questa operazione non si preoccupa di aggiornare il valore del record contenente la chiave K qualora questa sia già presente nella lista.

Complessità e ricorrenze

L'algoritmo, non ricorsivo, consiste in un insieme di istruzioni di assegnamento: possiamo affermare con sicurezza che la complessità di questo algoritmo è costante.

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

2. search

```

1  search(Dizionario L, Key K)
2      x = L.head
3      while x != NIL and x.Key != K
4          x = x.next
5      if x != NIL
6          return x.Info
7      else
8          return NIL

```

Spiegazione e osservazioni

L'algoritmo consiste in uno semplice scorriamento della lista elemento per elemento. Ritorna l'informazione portata dal record amente la chiave cercata, se presente, NIL altrimenti.

Complessità e ricorrenze

Nel caso peggiore possibile, l'algoritmo deve scorrere l'intera lista:

$$T(n) = O(n)$$

Analisi della correttezza dell'algoritmo

Affinché questo algoritmo sia corretto, ci deve garantire di ritornare la prima occorrenza della chiave K. Possiamo provarlo dimostrando la persistenza di un invariante per l'intera iterazione del ciclo **while** dell'algoritmo. Un **invariante** è un'asserzione (formula) che deve essere vera prima, dopo e ad ogni iterazione del ciclo. Per un invariante si deve dimostrare:

- **Inizializzazione:** l'invariante è vero prima della prima iterazione del ciclo.
- **Conservazione:** se l'invariante è vero prima di un'iterazione del ciclo, rimane vero prima della successiva iterazione.

$INV \wedge \text{Guardia} \text{ (Condizione del ciclo)} \Rightarrow INV \text{ [dopo esecuzione corpo ciclo]}$

- **Conclusione:** quando il ciclo termina, l'invariante fornisce un'utile proprietà che aiuta a dimostrare la correttezza dell'algoritmo.

$INV \wedge \neg \text{Guardia} \Rightarrow \text{Asserzione finale}$

- **Funzione di terminazione:** è una funzione a valori naturali che decresce strettamente ad ogni iterazione del ciclo. Ad un certo punto arriva allo 0: il ciclo necessariamente termina.

Nel nostro caso (ciclo `while` della `search`):

- La funzione è il numero di elementi della lista non ancora visitati.
- $INV \equiv$ Gli elementi da `L.head` ad `x` escluso hanno chiave diversa da `K`.

Vediamo nel dettaglio la dimostrazione della correttezza della `search`.

- (a) **Inizializzazione:** all'inizio poniamo `x = L.head`: l'asserzione è vera perché ci sono 0 elementi tra `L.head` e `x` non compreso, quindi non possiamo aver già trovato l'elemento con chiave `K`. Diciamo che l'asserzione è vacuamente vera.
- (b) **Conservazione:**

$$INV \wedge \text{Guardia} \Rightarrow INV \left[\frac{x.\text{next}}{x} \right]$$

Per ipotesi, sappiamo che:

- i. Invariante vero: gli elementi da `L.head` ad `x` non compreso hanno chiave diversa da `K`.
- ii. Guardia vera: `x != NIL && x.Key != K`

Dobbiamo verificare la tesi:

- iii. $INV \left[\frac{x.\text{next}}{x} \right] \equiv$ Gli elementi da `L.head` ad `x.next` non compreso hanno chiave diversa da `K`.

Per l'ipotesi (i.) gli elementi da `L.head` a `x` non compreso hanno chiave diversa da `K`, e per l'ipotesi (ii.) l'elemento puntato da `x` ha chiave diversa da `K`. Quindi ho dimostrato che tutti gli elementi da `L.head` ad `x` compreso hanno chiave diversa da `K`.

- (c) **Conclusione:**

$$INV \wedge \neg \text{Guardia} \Rightarrow \text{Asserzione finale}$$

Il ciclo può terminare per due ragioni:

- i. `x == NIL`: in questo caso abbiamo esaurito gli elementi della lista, l'abbiamo scorsa tutta; ma alla fine del ciclo l'invariante resta vera: tutti gli elementi da `L.head` ad `x` non compreso (ma in questo caso `x == NIL`, la fine della lista) hanno chiavi diverse da `K`. Quindi in questo caso l'elemento cercato non appartiene alla mia lista.
- ii. `x != NIL && x.Key == K`: l'invariante mi assicura che `K` non è presente prima di `x`, e dunque `x` è la prima occorrenza della chiave `K`.

In questo modo abbiamo dimostrato la correttezza del nostro algoritmo di ricerca.

3. `delete`

Dato che l'`insert` non aggiorna il valore delle chiavi qualora queste siano già presenti, bisogna scorrere tutta la lista per rimuovere tutte le occorrenze della chiave K.

```

1 delete(Dizionario L, Key K)
2     x = L.head
3     while x != NIL
4         if x.Key == K
5             if x.next != NIL
6                 x.next.prev = x.prev
7                 if x.prev != NIL
8                     x.prev.next = x.next
9                 else
10                    L.head = x.next
11                    temp = x
12                    x = x.next
13                    rimuovi temp
14                else
15                    x = x.next

```

Spiegazione ed osservazioni

L'algoritmo consiste in un ciclo `while` che scorre tutta la lunghezza della lista. Se il puntatore iteratore trova un elemento avente la chiave cercata, sistema i puntatori degli elementi precedente e successivo (qualora presenti) e infine elimina l'elemento della lista.

Complessità e ricorrenze

Il codice consiste in un'insieme di istruzioni di assegnamento. Tuttavia, poiché l'algoritmo deve scorrere l'array per trovare tutte le entry avente la chiave cercata, la complessità dell'algoritmo è lineare rispetto al numero di elementi.

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

8 Esercizi sul calcolo della complessità

8.1 Esercizio 1

```

1 A(int n)
2     s = 0
3     for i = 1 to n
4         s = s + B(i)
5     return s
6
7 B(int n)
8     s = 0
9     for j = 1 to n
10        s = s + 1
11    return s

```

Dal momento che la funzione B viene chiamata in A, conviene prima calcolare il costo di B.

La complessità di B dipende dalla lunghezza del suo ciclo `for`:

$$T_B(n) = \Theta(n)$$

Dall'alta parte, anche la complessità di A dipende dalla lunghezza del suo ciclo `for`, ma a differenza di B, nel suo corpo ha anche una chiamata a una funzione esterna (B, appunto); ergo, per calcolarne la complessità, bisogna sommare ad ogni iterazione la complessità di B a cui viene passato l'indice del ciclo di A:

$$\begin{aligned} T_A(n) &= \sum_{i=1}^n \Theta(i) \\ &= \sum_{i=1}^n k \cdot i + d \\ &= k \sum_{i=1}^n i + d \\ &= k \frac{n(n+1)}{2} + d = \Theta(n^2) \end{aligned}$$

Notiamo che $B(n) = n$: B è la funzione identità, quindi A è la funzione che fa la somma dei primi n numeri naturali. Esiste un modo migliore, con complessità minore, che fa lo stesso lavoro?

```

1 A'(int n)
2     return n(n+1)/2

```

$$T_{A'}(n) = O(1) \Rightarrow \text{Complessità costante}$$

8.2 Esercizio 2

```

1 foo(int n)
2     if n <= 2
3         return 1
4     else
5         if n > 321
6             i = n/2
7             return 2 * foo(i) + n * n * n * i
8         else
9             return foo(n - 1) + foo(n - 2)

```

Dal momento che bisogna studiare il comportamento asintotico della funzione (cioè per quando $n \rightarrow \infty$), la porzione di codice di nostro interesse è il ramo `if` delle righe 5-7.

Poiché questo blocco di istruzioni è costituito da operazioni aritmetiche e di assegnamento di complessità costante e da una (e una sola) chiamata ricorsiva con parametro $\frac{n}{2}$, la complessità di `foo` sarà la seguente:

$$T(n) = 1 \cdot T\left(\frac{n}{2}\right) + c$$

Applichiamo il Teorema Master:

- 1) $a = 1$, $b = 2 \Rightarrow d = \log_b a = \log_2 1 = 0$
- 2) $f(n) = c$ (funzione costante)
- 3) $n^0 = \Theta(f(n)) \Rightarrow 1 = \Theta(c)$

Siamo nel secondo caso del Teorema Master:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$$

8.3 Esercizio 3

Nell'ipotesi che `Proc(n)=Θ(√n)`, determinare la complessità asintotica della procedura `Fun(A, n)`.

```

1 Fun(A, n)
2     if n < 1
3         return 1
4     else
5         t = Fun(A, n/2)
6         if t > A[n]
7             t = t + Fun(A, n/2)
8         for j = 1 to n
9             t = t + A[j] + Proc(n)
10        return t

```

Per gli stessi motivi degli esercizi precedenti, andremo a studiare il blocco di codice delle righe 4-10 (ramo `else`): notiamo una prima chiamata ricorsiva a `Fun` nella riga 5, una seconda a riga 7 e una chiamata a `Proc` a riga 9, di cui conosciamo la complessità; bisogna notare anche che la chiamata di riga 9 viene

eseguita tante volte quante sono le iterazioni del ciclo **for** di riga 8 (quindi n). Quindi,

$$\begin{aligned} T_{Fun}(n) &= T_{Fun}\left(\frac{n}{2}\right) + T_{Fun}\left(\frac{n}{2}\right) + \Theta(n\sqrt{n}) + c \\ &= 2T_{Fun}\left(\frac{n}{2}\right) + \Theta(n\sqrt{n}) \end{aligned}$$

Applichiamo il Teorema Master:

- 1) $a = 2, b = 2 \Rightarrow d = \log_b a = \log_2 2 = 1$
- 2) $f(n) = n\sqrt{n} = n^{\frac{3}{2}}$
- 3) Verifichiamo di essere nel terzo caso: $f(n) = \Omega(n^{d+\epsilon})$

Devo cercare un $\epsilon > 0$: poniamo $\epsilon = \frac{1}{2}$ e la condizione è verificata.

Verifichiamo la condizione di regolarità:

$af\left(\frac{n}{b}\right) \leq cf(n)$, cioè quando vado in ricorsione la funzione decresce.

Troviamo un $c < 1$:

$$2\frac{n}{2}\sqrt{\frac{n}{2}} \leq cn\sqrt{n} \Rightarrow c \geq \frac{1}{\sqrt{2}} \Rightarrow \text{Pongo } c = \frac{1}{\sqrt{2}} \Rightarrow \text{Verificata.}$$

Quindi siamo nel terzo caso del Teorema Master:

$$T(n) = \Theta(f(n)) = \Theta(n\sqrt{n})$$

8.4 Esercizio 4

```

1 calcola(int n)
2     if n < 1
3         return 5
4     else
5         k = 1
6         for i = 1 to n
7             k = k * 2
8         return calcola(n - 1) * k

```

$$\begin{aligned} T(n) &= T(n - 1) + \Theta(n) + d \\ &= T(n - 1) + cn \\ T(0) &= a \end{aligned}$$

Per calcolare la complessità possiamo procedere a srotolare la formulazione delle ricorrenze fino al caso base $T(0)$: sappiamo infatti che la complessità al livello n è cn :

$$\begin{aligned} T(n) &= T(n - 1) + cn = T(n - 2) + c(n - 1) + cn = \dots \\ &= \sum_{i=1}^n ci + T(0) = c \sum_{i=1}^n i + T(0) = c \frac{n(n + 1)}{2} + T(0) = \Theta(n^2) \end{aligned}$$

8.5 Esercizio 5

```

1 computa(int n)
2     if n >= 10
3         for i = n down to n - 3
4             j = floor(n/2)
5             return computa(floor(n/2))*computa(floor(n/2))+5n
6     else
7         if n >= 20
8             return computa(n + 1)
9     else
10        return n

```

Il ciclo **for** delle righe 3-4 viene eseguito esattamente 3 volte, e il suo corpo è costituito da una sola istruzione aritmetica: avrà quindi complessità costante $O(1)$.

Bisogna sottolineare il fatto che le righe 7-8 costituiscono codice morto: non si verificherà mai la condizione tale per cui quelle righe di codice verranno eseguite. La complessità dell'algoritmo si può derivare sostanzialmente dalle due chiamate ricorsive nella riga 5:

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

Applichiamo il Teorema Master:

- 1) $a = 2, b = 2 \Rightarrow d = \log_b a = \log_2 2 = 1$
- 2) $f(n) = c$ (funzione costante)
- 3) $f(n) = c = O(n^{d-\varepsilon}) = O(n^{1-\varepsilon})$

Ponendo $\varepsilon = \frac{1}{2}$, abbiamo verificato di essere nel primo caso del Teorema Master:

$$T(n) = \Theta(n^d) = \Theta(n)$$

9 Alberi

Un **albero radicato** è una coppia $T = (N, A)$ dove:

- N è un insieme finito di **nodi** fra cui si distingue un nodo r detto radice
- $A \subseteq N \times N$ è un insieme di coppie di nodi, detti **archi**

In un albero, ogni nodo v (eccetto la radice) ha esattamente un **genitore** (o **padre**) u tale che esiste un arco $(u, v) \in A$.

Definizioni

- Un nodo n può avere zero o più figli v tali che esiste un arco $(u, v) \in A$.
- Il numero dei figli di un nodo è detto **grado** del nodo.
- Un nodo senza figli (grado zero) è detto **foglia** (o nodo esterno, ma non useremo questo termine).
- Un nodo non foglia è detto **nodo interno**.
- Se due nodi hanno lo stesso padre sono detti **fratelli**.
- Il **cammino** da un nodo u a un nodo u' in un albero T è una sequenza di nodi $< n_0, n_1, \dots, n_k >$ tali che:
 - $u = n_0$
 - $u' = n_k$
 - $< n_{i-1}, n_i > \in A \forall i = 1, \dots, k$
- La **lunghezza** di un cammino è il numero degli archi del cammino, o il numero di nodi che formano il cammino meno uno.
 - Esiste sempre un cammino di lunghezza zero che va da u a u .
- Sia x un nodo in un albero radicato T con radice r . Un nodo qualsiasi y in un cammino che parte da r e arriva a x è detto **antenato** di x .
 - Ogni nodo è antenato di sé stesso.
- Se y è antenato di x , allora x è **descendente** di y .
 - Ogni nodo è quindi antenato e descendente di sé stesso.
- Se y è un antenato di x e x è diverso da y , allora y è un **antenato proprio** di x , e x è un **descendente proprio** di y .
- Il **sottoalbero** con radice in x è l'albero indotto dai discendenti di x .
- La **profondità** di un nodo x è la lunghezza del cammino dalla radice a x .

- Un **livello** di un albero è costituito da tutti i nodi che stanno alla stessa profondità.
- L'**altezza** di un nodo è la lunghezza del più lungo cammino che scende dal nodo x a una foglia.
 - L'altezza di un albero è l'altezza della sua radice, ed è anche uguale alla profondità massima di un qualsiasi nodo dell'albero.

9.1 Alberi binari e alberi k -ari

Un albero binario è un albero definito in modo ricorsivo:

- Un albero vuoto è un albero binario.
- Un albero costituito da un nodo radice, da un albero binario (detto sottoalbero sinistro della radice) e da un albero binario (detto sottoalbero destro della radice) è un albero binario.

Un albero k -ario è un albero in cui i figli di un nodo sono etichettati con interi positivi distinti e le etichette maggiori di k sono assenti.

Un albero binario è un caso particolare di albero k -ario, avente $k = 2$.

Un albero k -ario **completo** è un albero k -ario in cui valgono le seguenti caratteristiche:

- Tutte le foglie hanno la stessa profondità
- Tutti i nodi interni hanno grado k (hanno esattamente k figli)

Esercizio Trovare il numero di foglie e il numero di nodi interni di un albero k -ario completo, la cui altezza risulta essere h .

Idea Numero di foglie = k^h

Possiamo dimostrarlo per induzione sull'altezza dell'albero.

Caso base $h = 0 \Rightarrow$ L'albero è costituito dalla sola radice. Al livello 0, $k^0 = 1$: ho infatti un'unica foglia che è la radice. Vero perché assumiamo l'ipotesi induttiva per un albero di altezza h sia vero che $n_{foglie}(h) = k^h$ e dimostro che $n_{foglie}(h+1) = k^{h+1}$ per alberi di altezza $h+1$.

Passo induttivo Il numero di nodi alla profondità h sono k^h per l'ipotesi induttiva in quanto l'albero di altezza h è un albero completo.

Ognuno dei nodi al livello h ha esattamente k figli perché l'albero è completo.
Dunque,

$$k^h \cdot k = k^{h+1}$$

Quindi,

$$n_{foglie}(h+1) = k^{h+1}$$

come volevasi dimostrare.

Per calcolare il numero di nodi interni,

$$\sum_{i=0}^{h-1} k^i = \frac{k^{(h-1)+1} - 1}{k - 1} = \frac{k^h - 1}{k - 1}, \quad k \neq 1$$

Esercizio Troviamo l'altezza di un albero k -ario completo, sapendo n_{foglie} .
Abbiamo dimostrato che $n_{foglie} = k^h \Rightarrow h = \log_k n_{foglie}$.

Esercizio Dimostriamo che il numero di foglie in un albero binario completo non nullo con n nodi è $\frac{n+1}{2}$.

Abbiamo dimostrato che, in un albero k -ario completo di altezza h ,

$$n_{foglie} = k^h, \quad n_{interni} = \frac{k^h - 1}{k - 1}$$

Nel caso di un albero binario (cioè con $k = 2$), avremo:

$$n_{foglie} = k^h = 2^h, \quad n_{interni} = \frac{2^h - 1}{2 - 1} = \frac{2^h - 1}{1} = 2^h - 1$$

Sommando, otteniamo:

$$n_{nodi} = n_{foglie} + n_{interni} = 2^h + 2^h - 1 = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

Andando a sostituire nella nostra ipotesi, otteniamo:

$$\begin{aligned} n_{foglie} &= \frac{n_{nodi} + 1}{2} \\ &= \frac{(2^{h+1} - 1) + 1}{2} \\ &= \frac{2^{h+1} - 1 + 1}{2} \\ &= \frac{2 \cdot 2^h}{2} = 2^h \end{aligned}$$

che è l'equivalenza da cui siamo partiti e che abbiamo precedentemente dimostrato.

9.2 Tipo Albero

Dati Un insieme di nodi (di tipo Node) e un insieme di archi.

Operazioni

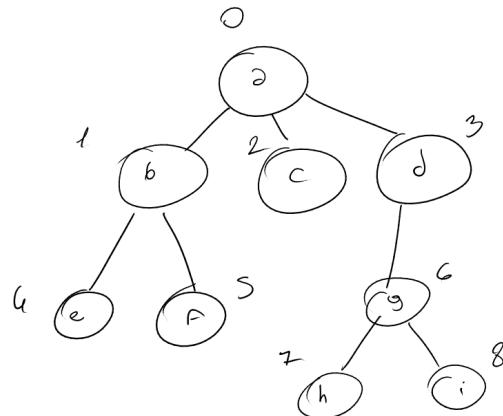
1. `newtree() → Tree`
 - Post-condizione: restituisce un albero vuoto.
2. `treeEmpty(Tree P) → bool`
 - Post-condizione: restituisce `true` se l'albero è vuoto, `false` altrimenti.
3. `padre(Tree P, Node v) → Node U {NIL}`
 - Pre-condizione: $v \in P$
 - Post-condizione: restituisce il padre di v se v è diverso dalla radice, `NIL` altrimenti.
4. `figli(Tree P, Node v) → List of Node`
 - Pre-condizione: $v \in P$
 - Post-condizione: restituisce una lista contenente i figli del nodo v .

9.3 Alberi implementati con array

9.3.1 Vettore padri

L'implementazione con gli array prevede l'utilizzo di vettore di coppie, che rappresentano i singoli nodi, rappresentati da delle coppie (`info`, `parent`), contenenti rispettivamente l'informazione del nodo e l'indice del suo nodo padre.

Sia $T = (N, A)$ un albero con n nodi numerati da 0 a $n - 1$ rappresentabile come segue.



Il relativo vettore P di dimensione n sarà il seguente:

P.info	a	b	c	d	e	f	g	h	i
P.parent	-1	0	0	0	1	1	3	6	6
Indice	0	1	2	3	4	5	6	7	8

Osservazioni

- $\forall v \in [0, n - 1]$,
 - $P[v].info$ è il contenuto informativo
 - $P[v].parent == u$ se e solo se vi è un arco $(u, v) \in A$
- Se v è la radice, $P[v].parent == -1$
- $P.length$ contiene il numero di nodi dell'albero
- Spazio richiesto: $\Theta(n)$ per un albero di n nodi

Operazioni

1. padre

```

1 padre(Tree P, Node v)
2     if P[v].parent == -1
3         return NIL
4     else
5         return P[v].parent

```

Complessità $T(n) = \Theta(1)$ perché eseguiamo solo operazioni di complessità costante.

2. figli

```

1 figli(Tree P, Node v)
2     l = creaLista()
3     for i = 0 to P.length - 1
4         if P[i].parent == v
5             inserisci i in l
6     return l

```

Complessità $T(n) = \Theta(n)$ perché dobbiamo scorrere tutta la lunghezza dell'array.

9.3.2 Vettore posizionale

Per questa rappresentazione, supponiamo di avere un albero k -ario con $k \geq 2$, ogni nodo del quale ha una posizione prestabilita nella struttura.

Sia $T = (N, A)$ un albero k -ario completo, P è un vettore di dimensione n tale che $P[v]$ contiene l'informazione associata al nodo v . Inoltre,

- 0 è la posizione della radice
- L' i -esimo figlio di v è in posizione $k \cdot v + 1 + i$, $i \in \{0, \dots, k - 1\}$
- Il padre del nodo f , $f \neq 0$, è in posizione

$$k \cdot v + 1 \text{ (primo figlio)} \leq f \leq k \cdot v + 1 + (k - 1) \text{ (ultimo figlio)}$$

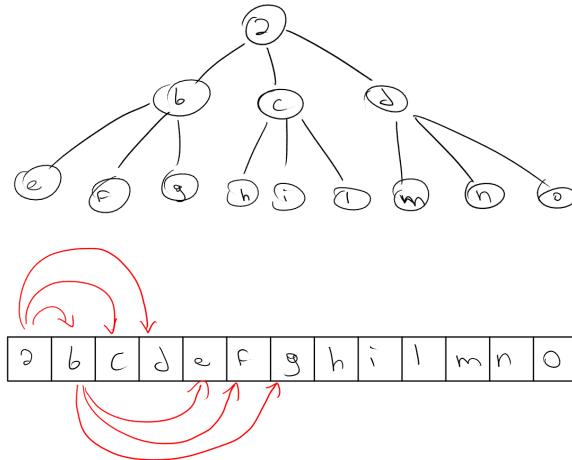
$$k \cdot v \leq f - 1 \leq k \cdot v + k + 1$$

$$v \leq \frac{f - 1}{k} \leq v + \frac{k - 1}{k} < v + 1$$

Poiché v è un indice, un numero intero, si ha:

$$v = \left\lfloor \frac{f - 1}{k} \right\rfloor$$

Vediamo un esempio.



Volendo cercare il padre del nodo 1, avente indice 9 nel nostro array, possiamo scoprirne l'indice facendo la seguente operazione: $\left\lfloor \frac{9-1}{3} \right\rfloor = 2 \Rightarrow$ nodo c.

Lo spazio per un albero con n nodi è $\Theta(n)$, e la struttura dati contiene i seguenti campi:

- $P.length$: contiene il numero di nodi
- $P.K$: contiene il grado di tutti i nodi

Operazioni

1. padre

```

1  padre(Tree P, Node v)    //v è un indice
2      if v == 0
3          return NIL
4      else
5          return floor((v-1)/P.K)

```

Complessità: $O(1)$.

2. figli

```

1  figli(Tree P, Node v)
2      l = creaLista()
3      if v * P.K + 1 >= P.length
4          return l
5      else
6          for i = 0 to P.K - 1
7              inserisci v * P.K + 1 + i in l
8      return l

```

Complessità: $\Theta(\text{grado}(v)) = O(K)$.

Le rappresentazioni con i vettori hanno lo svantaggio di avere delle operazioni di inserimento e cancellazione costose: vediamo ora un'altra rappresentazione, questa volta però basata su strutture collegate.

9.4 Alberi implementati con strutture collegate

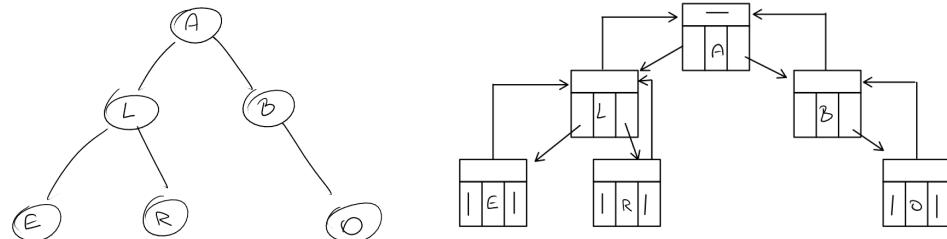
Possiamo rappresentare un albero facilmente attraverso la definizione di un record che contiene:

- L'informazione associata al nodo
- Un puntatore al padre
- Altri puntatori (per accedere ai figli)

9.4.1 Puntatori ai figli

Assunzione: se ogni nodo ha grado al più k , possiamo mantenere in ogni nodo un puntatore a ciascuno dei possibili k figli. Supponendo un albero binario ($k = 2$), ogni nodo x dell'albero avrà:

- $x.p$: puntatore al padre
- $x.left$: puntatore al figlio sinistro
- $x.right$: puntatore al figlio destro
- $x.Key$: contiene l'informazione del nodo



Definiamo un albero T avente come campo $T.root$, un puntatore alla radice dell'albero.

Lo spazio richiesto per questa implementazione è $\Theta(n \cdot k)$ e, se k è costante, $\Theta(n)$.

Operazioni

1. padre

```
1  padre(Tree P, Node v)           //v è un puntatore
2      return v.p
```

Complessità: $O(1)$.

2. figli

```

1 figli(Tree P, Node v)           //Condizione: k = 2
2   l = creaLista()
3   if v.left != NIL
4     inserisci v.left in l
5   if v.right != NIL
6     inserisci v.right in l
7   return l

```

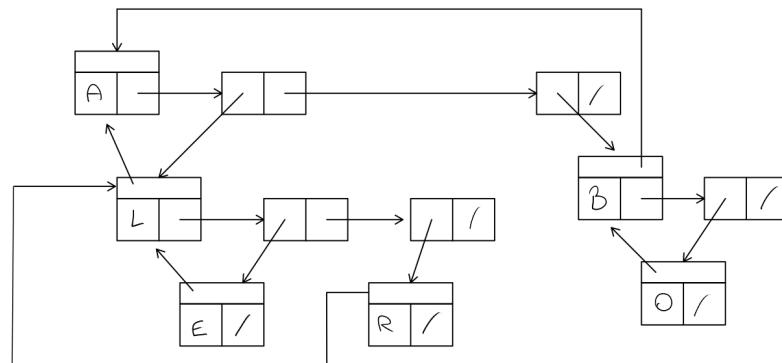
Complessità: $\Theta(\text{grado}(v)) = O(k)$ perché accedo a tanti puntatori quanto è il grado di P.

Anche questa rappresentazione comporta uno svantaggio: in presenza di `child1`, `child2`, ..., `childk`, se k è limitato da una grande costante ma la maggior parte dei figli sono NIL c'è uno spreco di memoria.

9.4.2 Lista figli

Se il numero di figli non è noto a priori, è possibile associare ad ogni nodo la lista dei puntatori ai suoi figli. Si tratta di una rappresentazione relativamente complessa, che per ogni nodo avente dei nodi figli prevede di memorizzare anche una lista a cui bisogna accedere prima di accedere ai figli stessi. Molto probabilmente non verrà chiesta in esame e non è una struttura altamente usata, ma male non fa saperla.

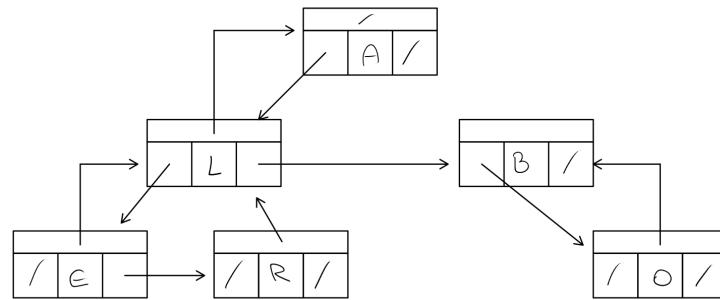
Con questa rappresentazione, l'albero di cui sopra può essere rappresentato come segue:



9.4.3 Figlio sinistro - Fratello destro

Ogni nodo x di questa rappresentazione ha:

- $x.Key$: contiene l'informazione del nodo.
- $x.p$: puntatore al padre.
- $x.left_child$: puntatore al figlio più a sinistra di x ; se vale NIL, x è un nodo foglia.
- $x.right_sib$: puntatore al fratello di x immediatamente alla sua destra; se vale NIL, è il figlio più a destra di suo padre.



Operazioni

1. padre

```

1 padre(Tree P, Node v)
2     return v.p
  
```

Complessità: $O(1)$.

2. figli

```

1 figli(Tree P, Node v)
2     l = creaLista()
3     iter = v.left_child
4     while iter != NIL
5         inserisci iter in l
6         iter = iter->right_sib
7     return l
  
```

Complessità: $\Theta(k)$.

9.5 Algoritmi di visita di alberi

9.5.1 Visita generica

```

1 visitaGenerica(Node r)
2     S = {r}
3     while S != emptySet
4         estrai un nodo n da S
5         visita il nodo n
6         S = S U {figli di n}

```

Teorema L'algoritmo di visita applicato alla radice di un albero con n nodi termina in $O(n)$ iterazioni. Lo spazio utilizzato è $O(n)$.

Dimostrazione Assumiamo l'ipotesi che la cancellazione e l'inserimento da S siano costanti. Ogni nodo verrà inserito ed estratto da S una volta sola, perché in un albero non si può tornare ad un nodo a partire dai suoi figli se lo stiamo scorrendo di figlio in figlio. Quindi le iterazioni del ciclo `while` saranno al più $O(n)$.

Poiché ogni nodo compare al più una volta in S , lo spazio richiesto è $O(n)$.

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

Per questo algoritmo faremo uso di una struttura dati a pila/stack S , dotata di politica LIFO.

```

1 visitaDFS(Node r)
2     Stack S
3     push(S, r)
4     while !stackEmpty(S)
5         u = pop(S)
6         if u != NIL
7             visita il nodo u
8             push(S, u.right)
9             push(S, u.left)

```

Complessità: lineare.

Versione ricorsiva

```

1 visidaDFS_rec(Node r)
2     if r != NIL
3         visita il nodo r
4         visidaDFS_rec(r.left)
5         visidaDFS_rec(r.right)

```

Teorema Se x è la radice di un sottoalbero di n nodi, la chiamata di `visitaDFS(x)` richiede il tempo $\Theta(n)$.

Dimostrazione $T(n)$ è il tempo di esecuzione della procedura quando è chiamata per la radice di un sottoalbero di n nodi. Poiché `visitaDFS` visita tutti gli n nodi dell'albero, $T(n) = \Omega(n)$.

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

dove $T(k)$ è il numero di nodi del sottoalbero sinistro, $T(n - k - 1)$ è il numero di nodi del sottoalbero destro.

Risolviamo col metodo di sostituzione, e diamo per vera l'intuizione che $T(n)$ abbia complessità lineare, supponiamo $T(n) = an + b$. Procediamo alla dimostrazione per induzione su n .

Caso base $n = 0 \Rightarrow T(0) = c$ per definizione. Quindi $T(0) = an + b = a \cdot 0 + b = b \Rightarrow b = c$.

Passo induttivo Assumiamo che per ogni $m < n$ vale che $T(m) = am + b$ e lo dimostro per n .

$$\begin{aligned} T(n) &= T(k) + T(n - k - 1) + d \\ &= ak + b + a(n - k - 1) + b + d \quad (\text{per l'ipotesi induttiva}) \\ &= ak + b + an - ak - a + b + d \\ &= an - a + 2b + d \end{aligned}$$

Vogliamo dimostrare che $an - a + 2b + d = an + b$:

$$\begin{aligned} an - a + 2b + d &= an + b \\ -a + b + d &= 0 \Rightarrow a = b + d = c + d \end{aligned}$$

Abbiamo dimostrato che $T(n) = (d + c)n + c$, che è lineare. La procedura `visitaDFS` ha tempo di esecuzione $T(n) = \Theta(n)$.

9.5.3 Tipi di visita (relative ad alberi binari)

1. **Visita in preordine (anticipata)**: si visita prima la radice, poi si fa una chiamata ricorsiva sul figlio sinistro e poi sul figlio destro. Nel caso dell'albero di cui sopra, l'ordine di visita sarebbe **A, L, E, R, B, O**.
2. **Visita simmetrica (in order)**: prima si effettua la chiamata ricorsiva sul figlio sinistro, poi si visita la radice, infine si fa la chiamata ricorsiva sul figlio destro. Nel caso dell'albero di cui sopra, l'ordine di visita sarebbe **E, L, R, A, B, O**.

3. **Visita in postordine (posticipata)**: si effettuano prima le chiamate ricorsive sul figlio sinistro e sul figlio destro, poi si visita la radice. Nel caso dell'albero di cui sopra, l'ordine di visita sarebbe E, R, L, O, B, A.

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

La visita in ampiezza, altrimenti detta *Breadth-First Search*, è un algoritmo che permette di visitare tutti i nodi presenti alla corrente profondità di un albero prima di scendere a visitare i nodi delle profondità inferiori.

Per questo algoritmo faremo uso di una struttura dati a coda/queue C , dotata di politica FIFO.

```

1 visitaBFS(Node r)
2     Queue C
3     C = newQueue()
4     enqueue(C, r)
5     while !queueEmpty(C)
6         u = dequeue(C)
7         if u != NIL
8             visita il nodo u
9             enqueue(C, u.left)
10            enqueue(C, u.right)

```

Complessità: $O(n)$.

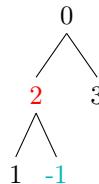
10 Esercizi d'esame sugli alberi

10.1 Nodi centrali

Un nodo di un albero binario è detto centrale se il numero di foglie del sottoalbero di cui è radice è pari alla somma delle chiavi dei nodi appartenenti al cammino dalla radice al nodo stesso.

- a) Scrivere un algoritmo efficiente che restituisce il numero di nodi centrali.
- b) Discuterne la complessità.
- c) Se vogliamo modificare l'algoritmo in modo che restituisca l'insieme dei nodi centrali, che tipo di struttura dati si può utilizzare per rappresentare un insieme? La complessità deve rimanere la stessa del caso a).

Forniamo un esempio di albero dotato di nodi centrali.



$0 + 2 = 2$, e il nodo 2 ha due foglie; $0 + 2 - 1 = 1$, e il nodo -1 è foglia di sé stesso.

Quindi in questo albero i nodi 2 e -1 sono nodi centrali.

Definiamo ora un albero binario e l'algoritmo richiesto in C++:

```

1 struct Node{
2     int Key;
3     Node* left;
4     Node* right;
5     Node* p;
6     Node(int K, Node* dad, Node* sx = nullptr, Node* dx = nullptr)
7         : Key{K}, p{dad}, left{sx}, right{dx} {}
8 };
9
10 typedef Node* Pnode;
11
12 int contaCentrali(Pnode u){
13     int numf;
14     return contaCentraliAux(u, 0, numf);
15 }
16
17 int contaCentraliAux(PNode u, int sum, int& numf){
18     int nodic, nodisx, numfsx, nodidx, numfdx;
19
20     //Caso base: albero vuoto
21     if (u == nullptr){
22         numf = 0;
23         return 0;
24     }
  
```

```

25
26     //Caso base: caso radice/foglia
27     if (u->left == nullptr && u->right == nullptr){
28         numf = 1;
29         nodic = 0;
30     }
31     else{
32         nodisx = contaCentraliAux(u->left, sum + u->Key, numfsx);
33         nodidx = contaCentraliAux(u->right, sum + u->Key, numfdx);
34         numf = numfsx + numfdx;
35         nodic = nodisx + nodidx;
36     }
37     if (numf == sum + u->Key){
38         return nodic + 1;
39     }
40     return nodic;
41 }
```

Complessità

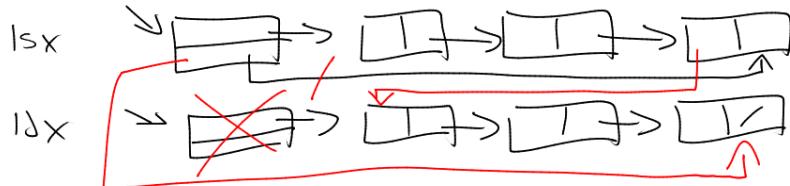
$$T(n) = \begin{cases} c & n \leq 1 \\ T(k) + T(n - k - 1) + d & n > 1 \end{cases}$$

Questa ricorrenza l'abbiamo già vista negli algoritmi di visita, quindi $T(n) = \Theta(n)$. Possiamo anche affermarlo perché stiamo visitando i nodi dell'albero una volta sola, dunque il tempo di esecuzione risulta essere $\Theta(n)$. Avendo n nodi e una complessità lineare rispetto a n , il nostro algoritmo è ottimale, non potremmo fare un algoritmo più efficiente.

Per quanto riguarda il punto c), useremo sicuramente una lista con un puntatore alla coda.

```

1 lsx = listaCentraliAux(u->left, sum + u->Key, numfsx);
2 ldx = listaCentraliAux(u->right, sum + u->Key, numfdx);
```



10.1.1 Algoritmo di decomposizione

Presentiamo un generico algoritmo ricorsivo per risolvere un problema decomponibile su alberi binari.

```

1 dekomponibile(Node u)    //Insieme ad eventuali altri parametri
2   if n == NIL
3     <risolvo direttamente>
4   else
5     //Potrebbero esserci altri casi base
6     //dipende dal problema
7     risultatosx = dekomponibile(u.left)
8     risultatodx = dekomponibile(u.right)
9     return ricombina(risultatosx, risultatodx)

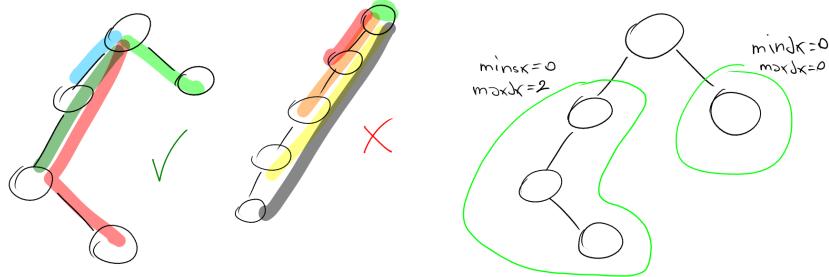
```

Risalendo a questo schema, sappiamo già quale sarà la complessità del nostro algoritmo: ogni nodo viene attraversato un numero costante di volte, per cui se il caso base e la regola `ricombina()` richiedono tempo costante, l'esecuzione richiede tempo totale $O(n)$, dove n è il numero di nodi dell'albero.

10.2 Cammino terminabile

Un cammino $< x_0, x_1, \dots, x_n >$ è detto terminabile se $x_n.left == NIL$ oppure se $x_n.right == NIL$. Diciamo che un albero è β -bilanciato se tutti i cammini terminabili che partono dalla radice hanno lunghezze che differiscono al più di β .

Scrivere un algoritmo efficiente che verifica se l'albero è β -bilanciato.



Una soluzione di questo esercizio può essere quella di cercare il cammino più lungo e il cammino più corto e calcolarne la differenza: se minore o uguale di β ritorna `true`, altrimenti `false`.

```

1 bool 3_bil(PNode u){
2   int min, max;
3   3_bilAux(u, min, max);
4   return (max - min <= 3);
5 }
6
7 void 3_bilAux(PNode u, int& min, int& max){
8   int minsx, maxsx, mindx, maxdx;
9   if (u == nullptr){
10     min = -1;
11     max = -1;

```

```

12     }
13     else{
14         3_bilAux(u->left, minsx, maxsx);
15         3_bilAux(u->right, mindx, maxdx);
16         min = (minsx <= mindx ? minsx : mindx) + 1;
17         max = (maxsx <= maxdx ? maxdx : maxsx) + 1;
18     }
19 }
```

Poiché caso base e funzione di ricombinazione sono costanti, $T(n) = \Theta(n)$. Alternativamente, si può dimostrare attraverso l'esplicitazione della ricorrenza:

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

che risulta essere la solita ricorrenza degli algoritmi di visita; quindi $T(n) = \Theta(n)$.

10.3 Stampa livello

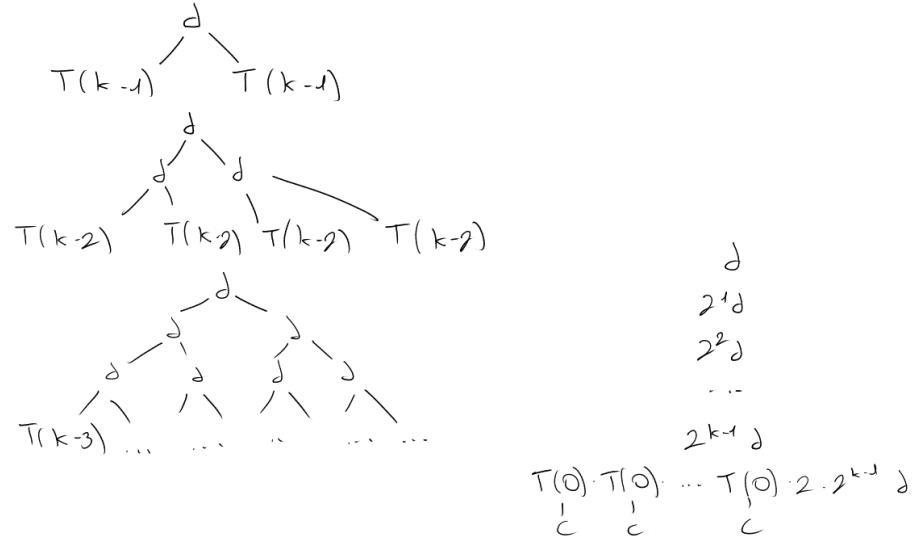
Sia T un albero binario. Progettare un algoritmo che, presi in ingresso la radice di T e un intero k che risulti essere maggiore di 0, stampi le chiavi contenuti nei nodi di T al livello k , procedendo da sinistra verso destra.

```

1 void stampaLivello(PNode u, int k){
2     if (u != nullptr){
3         if (k == 0){
4             std::cout << u->Key;
5         }
6         else{
7             stampaLivello(u->left, k - 1);
8             stampaLivello(u->right, k - 1);
9         }
10    }
11 }
```

Complessità in funzione di k

$$T(k) = \begin{cases} c & k = 0 \\ 2T(k - 1) + d & k > 0 \end{cases}$$

Albero delle ricorsioni

$$\begin{aligned}
 T(k) &= \sum_{i=0}^{k-1} d \cdot 2^i + c \cdot 2^k \\
 &= d \cdot \frac{2^{k-1+1} - 1}{2 - 1} + c \cdot 2^k \\
 &= d \cdot 2^k - d + c \cdot 2^k \\
 &= 2^k(d + c) - d = \Theta(2^k)
 \end{aligned}$$

Usando come dimensione del problema la profondità k , la complessità è $\Theta(2^k)$, esponenziale.

Complessità in funzione di n

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

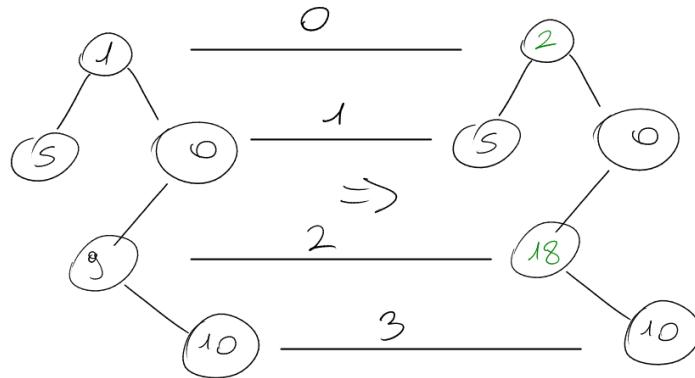
che è la solita ricorrenza che abbiamo trovato negli algoritmi di visita: $T(n) = O(n)$.

In questo caso la complessità non è $\Theta(n)$ perché posso anche non andare attraverso tutti i nodi.

10.4 Raddoppia valori

Sia T un albero generale i cui nodi hanno chiavi intere e campi: `Key`, `left_child`, `right_sib`.

Scrivere una procedura ricorsiva efficiente che trasformi T raddoppiando i valori di tutte le chiavi sui livelli pari dell'albero.



```

1 struct NodeG{
2     int Key;
3     NodeG* left_child;
4     NodeG* right_sib;
5 };
6
7 typedef NodeG* PNodeG;
8
9 void trasforma(PNodeG u){
10     PNodeG iter;
11     if (u != nullptr){
12         u->Key = u->Key * 2; //Per ipotesi, sto chiamando la fun-
13                                //zione sulla radice (che sta al li-
14                                //vello 0, che e' pari)
15         trasforma(u->right_sib); //Siccome questo nodo si trova
16                                //su un livello pari, trasformiamo
17                                //tutti i nodi su quel livello
18         iter = u->left_child;
19         while(iter != nullptr){
20             trasforma(iter->left_child);
21             iter = iter->right_sib;
22         }
23     }
24 }

```

Complessità Il tempo di esecuzione è $\Theta(n)$ perché ogni nodo è visitato una volta sola.

10.5 Discendenti dello stesso colore

Dato un albero binario i cui nodi sono colorati di bianco e nero, scrivere una funzione efficiente che calcoli il numero di nodi aventi lo stesso numero di discendenti bianchi e neri. Ricordiamo che un nodo è discendente di sé stesso. Analizzare la complessità della funzione.

Il tipo `Node` utilizzato per rappresentare l'albero binario è il seguente:

```

1  struct Node{
2      char c;
3      Node* left;
4      Node* right;
5      Node(char col, Node* l = nullptr, Node* r = nullptr)
6          : c{col}, left{l}, right{r} {}
7  };
8
9  typedef Node* PNode;
10
11 struct Tree{
12     PNode root;
13     Tree(PNode r = nullptr) : root{r} {}
14 };
15
16 typedef Tree* PTree;
17
18 int discSameColor(PTree t){
19     int count = 0, bianchi = 0, neri = 0;
20     sameColorAux(t->root, count, bianchi, neri);
21     return count;
22 }
23
24 void sameColorAux(PNode u, int& count, int& bianchi, int& neri){
25     if (u){
26         int bsx = 0, nsx = 0, bdx = 0, ndx = 0;
27         sameColorAux(u->left, count, bsx, nsx);
28         sameColorAux(u->right, count, bdx, ndx);
29         bianchi = bsx + bdx + (u->c == 'b');
30         neri = nsx + ndx + (u->c == 'n');
31         if (bianchi == neri){
32             count++;
33         }
34     }
35 }
```

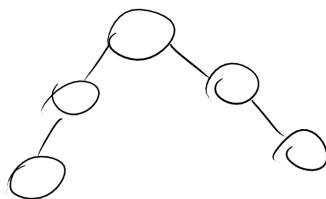
Complessità Il tempo d'esecuzione è $\Theta(n)$ perché ogni nodo è visitato una volta sola.

11 Alberi binari di ricerca

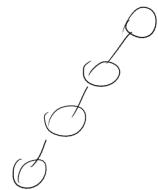
11.1 Alberi binari bilanciati

Definizione Un albero binario è **bilanciato** se $h = O(\log n)$.

Un albero completo è un albero bilanciato, ma un albero bilanciato non è necessariamente completo.



Esempio di albero bilanciato ma non
completo



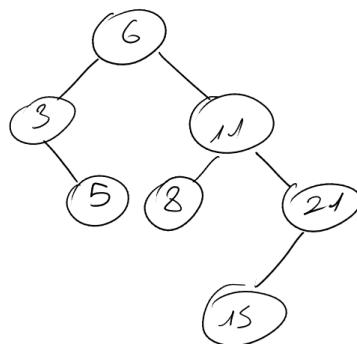
Albero fortemente sbilanciato in cui
 $h = n - 1$

11.2 Alberi binari di ricerca

Definizione Un **albero binario di ricerca** è un albero binario che soddisfa la seguente **proprietà di ricerca**.

Proprietà di ricerca Sia x un nodo di un albero binario di ricerca. Se y è un nodo nel sottoalbero sinistro di x , allora $y.Key \leq x.Key$. Se y è un nodo nel sottoalbero destro di x , allora $y.Key \geq x.Key$.

Vediamo un esempio.



La proprietà di ricerca consente di elencare in ordine non decrescente le chiavi di un albero binario di ricerca visitando l'albero in ordine simmetrico.

```

1 if u != NIL
2   visita(u->left)
3   print(u->Key)
4   visita(u->right)
  
```

(Da dimostrare per induzione sul numero di nodi dell'albero)

11.3 Operazioni su alberi binari di ricerca

1. Ricerca

```
1 Tree_search(Node x, Elem k) -> Node U NIL
```

Post-condizione: restituisce un nodo con chiave k se esiste, NIL altrimenti.

Versione ricorsiva

```
1 Tree_search(Node x, Elem k)
2     if x == NIL or x.Key == k
3         return x
4     else
5         if x.Key > k
6             return Tree_search(x.left, k)
7         else
8             return Tree_search(x.right, k)
```

Versione iterativa

```
1 Tree_search_iter(Node x, Elem k)
2     while x != NIL AND x.Key != k
3         if k < x.Key
4             x = x.left
5         else
6             x = x.right
7     return x
```

Correttezza Possiamo tagliare lo spazio di ricerca perché la proprietà degli alberi binari di ricerca assicura che nel sottoalbero destro non ci sono nodi con chiavi minori di $x.Key$. In maniera analoga si taglia il sottoalbero sinistro perché la proprietà assicura che non ci sono nodi con chiavi maggiori di $x.Key$ in tale albero.

Complessità I nodi incontrati durante la ricorsione formano un cammino verso il basso dalla radice dell'albero, quindi il tempo di esecuzione è $O(h)$ dove h è l'altezza dell'albero.

Se l'albero è bilanciato allora $T(n) = O(\log n)$.

Se l'albero è fortemente sbilanciato (caso di una catena come nella pagina precedente), $T(n) = O(n)$.

2. Massimo e minimo

Massimo

```
1 Tree_maximum(Node x) -> Node
```

Pre-condizione: $x \in T$.

Post-condizione: ritorna il nodo con chiave più grande nel sottoalbero radicato in x .

```
1 Tree_maximum(Node x)
2     while x.right != NIL
3         x = x.right
4     return x
```

Minimo

```
1 Tree_minimum(Node x) -> Node
```

Pre-condizione: $x \in T$.

Post-condizione: ritorna il nodo con chiave più piccola nel sottoalbero radicato in x .

```
1 Tree_minimum(Node x)
2     while x.left != NIL
3         x = x.left
4     return x
```

Correttezza Se un nodo x non ha sottoalbero sinistro allora poiché ogni chiave nel sottoalbero destro è almeno grande come $x.Key$, la chiave minima è proprio $x.Key$. Se x ha sottoalbero sinistro allora nessuna chiave nel sottoalbero destro può essere più piccola di $x.Key$ e ogni chiave nel sottoalbero sinistro non è maggiore di $x.Key$. Dunque il minimo lo trovo nel sottoalbero sinistro.

Complessità Il tempo di esecuzione è $O(h)$ perché, come in `Tree_search()`, la sequenza dei nodi visitati forma un cammino che scende dalla radice, e che può avere lunghezza massima h .

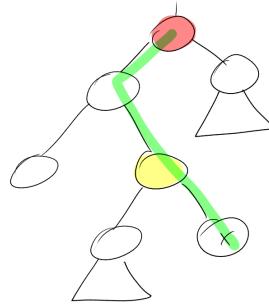
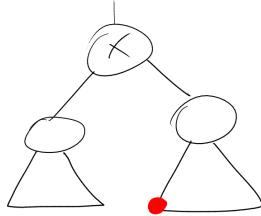
3. Predecessore e successore

Dato un albero binario di ricerca, voglio determinare dato un nodo il suo predecessore e successore nell'ordine stabilito in una vista simmetrica.

Se tutte le chiavi sono distinte, il successore di un nodo x è il nodo con la più piccola chiave che è maggiore di $x.Key$.

Successore

Distinguiamo due casi.



- a) x ha un figlio destro. In questo caso, il successore di x è il minimo del sottosalbero destro di x .
- b) x non ha un figlio destro. In questo caso, se esiste (perché potrebbe anche non esistere se x è il massimo), è l'antenato più prossimo di x di cui il figlio sinistro è anch'esso antenato di x . Per trovarlo, si risale da x verso la radice fino a incontrare la prima svolta a destra.

```
1 Tree_successor(Node x) -> Node
```

Pre-condizione: $x \in T$.

```
1 Tree_successor(Node x)
2     if x.right != NIL
3         return Tree_minimum(x.right)
4     else
5         y = x.p
6         while y != NIL AND x == y.right
7             x = y
8             y = y.p
9         return y
```

Complessità Se $x.right \neq NIL$ viene chiamata `Tree_minimum()`, che ha complessità temporale $O(h)$, altrimenti viene fatto un cammino partendo dal nodo x alla radice, che può essere anche questo lungo al più h ; quindi anche in questo caso il tempo di esecuzione è $O(h)$.

Si segue un cammino che scende nel caso di ricerca del minimo o che sale altrimenti, ma siamo sempre nella condizione di seguire un cammino lungo al massimo h .

Predecessore

Le casistiche e gli algoritmi relativi alla ricerca del predecessore di un nodo x appartenente ad un albero binario di ricerca sono analoghi a quelli di ricerca del successore, ma invertiti.

Se x ha un figlio sinistro, allora il predecessore sarà il massimo del sottosalbero sinistro di x , altrimenti sarà l'antenato più prossimo di x il cui figlio destro è anch'esso antenato di x . Per trovarlo, si risale da x verso la radice fino a incontrare la prima svolta a sinistra.

```
1 Tree_predecessor(Node x) -> Node
```

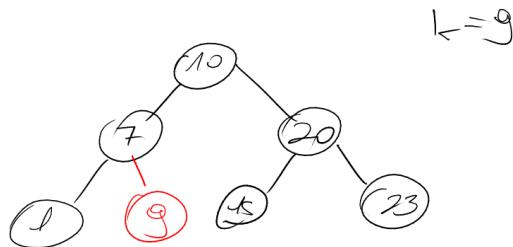
Pre-condizione: $x \in T$.

```
1 Tree_predecessor(Node x)
2     if x.left != NIL
3         return Tree_maximum(x.left)
4     else
5         y = x.p
6         while y != NIL AND x == y.left
7             x = y
8             y = y.p
9         return y
```

Complessità Analogamente allo studio del successore, questo algoritmo ha complessità $O(h)$.

4. Inserimento e cancellazione

Inserimento



Il nostro albero ha un campo `T.root`, che contiene l'indirizzo della radice.

```
1 Tree_insert(Tree T, Node z)
```

Post-condizione: inserisce il nodo `z` nell'albero `T` mantenendo la proprietà di ricerca.

`z` è un nodo avente le seguenti caratteristiche: `z.Key = v`, `z.left = NIL`, `z.right = NIL`.

```

1 Tree_insert(Tree T, Node z)
2     y = NIL      //Variabile che usiamo per trovare il padre di z
3     x = T.root    //Variabile che usiamo per scorrere l'albero
4     while x != NIL
5         y = x
6         if z.Key < x.Key
7             x = x.left
8         else
9             x = x.right
10        z.p = y
11        if y == NIL    //Albero vuoto: la radice diventa il nodo z
12            T.root = z
13        else
14            if z.Key < y.Key
15                y.left = z
16            else
17                y.right = z
  
```

Complessità Il ciclo `while` costa $O(h)$, e le altre istruzioni sono semplici assegnamenti: il tempo di esecuzione è quindi $O(h)$ perché si segue un cammino dalla radice verso il basso.

Cancellazione

Prima di fornire un'implementazione dell'algoritmo di cancellazione di un nodo da un albero binario di ricerca, è utile tenere a mente la seguente proprietà degli alberi binari di ricerca.

Proprietà

Se un nodo x in un albero binario di ricerca ha due figli, allora il suo successore non ha un figlio sinistro, e, in maniera analoga, il suo predecessore non ha un figlio destro.

Dimostrazione

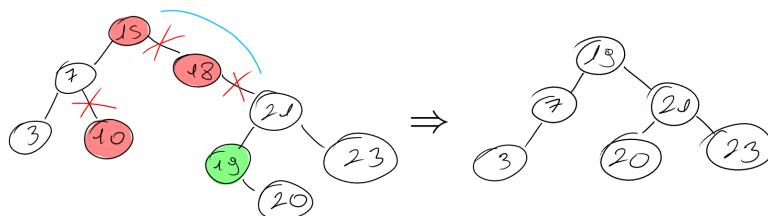
Sia x un nodo con due figli. In una visita simmetrica, i nodi del sottoalbero sinistro precedono x e quelli del sottoalbero destro seguono x .

Di conseguenza, il predecessore di x si troverà nel sottoalbero sinistro di x , e il suo successore nel suo sottoalbero destro.

Se s è il successore di x , assumiamo per assurdo che s abbia un figlio sinistro, che chiamiamo y . Allora, y segue x perché si trova nel suo sottoalbero destro, ma precede s perché si trova nel sottoalbero sinistro di s . Questo è assurdo, perché s non sarebbe più il successore di x , ma lo sarebbe y .

In modo simmetrico si può dimostrare che il predecessore non ha figlio destro.

Ritornando alla cancellazione...



Supponiamo di avere l'albero di cui sopra, da cui vogliamo rimuovere il nodo z . Distinguiamo tre casi:

- 1) Se z non ha figli, modifichiamo suo padre ($z.p$) per sostituire z con NIL. Un esempio di questo caso si ottiene ponendo $z = 10$.
- 2) Se z ha un unico figlio, stacchiamo z creando un collegamento tra suo figlio e suo padre. Un esempio di questo caso si ottiene ponendo $z = 18$.
- 3) Se z ha due figli, troviamo il successore y di z che deve trovarsi nel sottoalbero destro di z e facciamo in modo che y assuma la posizione di z nell'albero. Un esempio di questo caso si ottiene ponendo $z = 15$.

Per spostare dei sottoalberi all'interno di un albero si usa la seguente procedura.

```
1 Transplant(Tree T, Node u, Node v)
```

Pre-condizione: $u \in T$.

Post-condizione: sostituisce il sottoalbero con radice nel nodo u con i sottoalbero con radice nel nodo v .

```
1 Transplant(Tree T, Node u, Node v)
2     if u.p == NIL
3         T.root = v
4     else
5         if u == u.p.left //u e' figlio sinistro di suo padre?
6             u.p.left = v //v diventa figlio sx del padre di u
7         else //u e' figlio destro di suo padre
8             u.p.right = v //v diventa figlio dx del padre di u
9         if v != NIL
10            v.p = u.p
```

Complessità $O(1)$ perché costituita da semplici assegnamenti.

Possiamo ora stabilire una formulazione per l'operazione di eliminazione di un nodo da un albero binario di ricerca.

```
1 Tree_delete(Tree T, Node z)
```

Pre-condizione: $z \in T$.

```
1 Tree_delete(Tree T, Node z)
2     if z.left == NIL
3         Transplant(T, z, z.right)
4     else
5         if z.right == NIL
6             Transplant(T, z, z.left)
7         else
8             y = Tree_minimum(z.right)
9             if y.p != z
10                Transplant(T, y, y.right)
11                y.right = z.right
12                z.right.p = y
13            Transplant(T, z, y)
14            y.left = z.left
15            y.left.p = y
```

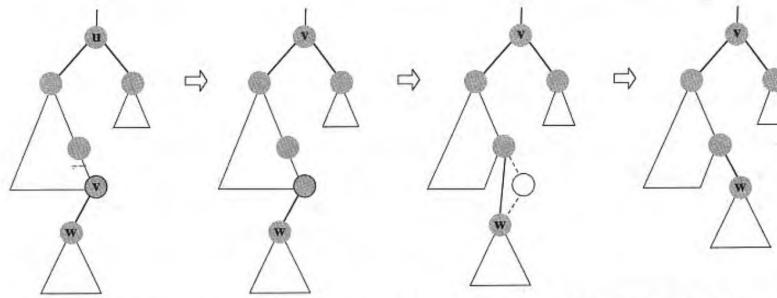
Complessità $O(h)$ per via della chiamata a `Tree_minimum()`.

Spiegazione ed osservazioni

Le due casistiche in cui il nodo da rimuovere non abbia nodi foglia o ne abbia al più uno sono coperte dall'algoritmo nelle righe 2-6, in cui viene effettuata una `Transplant()` del nodo da rimuovere con il suo sottoalbero

destro (qualora non ne abbia uno sinistro) o con il suo sottoalbero sinistro (qualora non ne abbia uno destro).

A partire dalla riga 7 viene affrontato il caso in cui il nodo abbia entrambi i figli. Per prima cosa viene trovato il suo successore y attraverso un calcolo del minimo del suo sottoalbero destro. Se questo non dovesse essere il figlio di z , verrebbe prima fatta una `Transplant()` del figlio destro di y nella posizione di y : affinché il nodo che originariamente è in posizione y possa prendere il posto del nodo in posizione z , una volta fatta la `Transplant()` tra questi due nodi, in y ci sarà un buco e non saranno corretti i puntatori del padre e dei figli di y se prima non lo sostituiamo col suo figlio destro. A prescindere che il successore sia il figlio o no, una volta eseguite le operazioni di cui sopra, l'algoritmo terminerebbe con una `Transplant()` del nodo y nella posizione del nodo z , il puntatore al figlio sinistro di y punterà al figlio sinistro di z e il puntatore di questo figlio a suo padre punterà a y .



11.4 Costruzione di alberi binari di ricerca

Teorema Le operazioni di ricerca di minimo, massimo, successore, predecessore, inserimento e cancellazione possono essere realizzate nel tempo $O(h)$ in un albero binario di ricerca di altezza h .

Se l'albero è bilanciato, $O(h) = O(\log n)$. Quanto più un albero è sbilanciato, tanto più il tempo d'esecuzione dei suddetti algoritmi su di esso tenderà ad essere $O(n)$.

Ma qual è la condizione che ci porta ad avere albero fortemente sbilanciati? Supponiamo di avere un vettore di elementi strettamente crescenti o decrescenti che vogliamo trasformare in un albero binario di ricerca attraverso il seguente algoritmo.

```

1 Arr A = [1, 3, 5, 7, 9]
2
3 buildBST(Arr A) -> Tree
4     t = newTree()
5     for i = 1 to A.length
6         u = creaNodo(A[i])    //u.Key = A[i], u.left = u.right = NIL
7         Tree_insert(t, u)
8     return t

```

Complessità Supponendo che la procedura `creaNodo()` abbia complessità costante, il tempo d'esecuzione di questo algoritmo dipende dalla complessità della funzione `Tree_insert()` e dal numero di volte che questa viene chiamata. Essendo chiamata in un ciclo che impiega tante iterazioni quanto è il numero di elementi del vettore, poiché la chiamata impiega un tempo pari a $O(h)$, e poiché l'albero risultante è fortemente sbilanciato dato il continuo inserimento di nodi nel sottoalbero sinistro o destro (se gli elementi sono ordinati in modo crescente o decrescente) ma non in entrambi, l'altezza dell'albero sarà pari al numero di nodi e quindi anche al numero di elementi del vettore passato come parametro. Se il numero di questi elementi è pari a n , il tempo d'esecuzione di `buildBST()` sarà $\Theta(n^2)$.

Lo possiamo anche dimostrare algebricamente:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-1} (c + d \cdot i) \\
&= \sum_{i=0}^{n-1} c + \sum_{i=0}^{n-1} d \cdot i \\
&= c \cdot n + d \cdot \sum_{i=0}^{n-1} i \\
&= c \cdot n + d \left(\frac{(n-1)(n-1+1)}{2} \right) \\
&= c \cdot n + d \left(\frac{n(n-1)}{2} \right) = \Theta(n^2)
\end{aligned}$$

Tuttavia questo algoritmo si può migliorare. Supponiamo di avere comunque un vettore ordinato di elementi, e di partire dal centro di questo array.

```

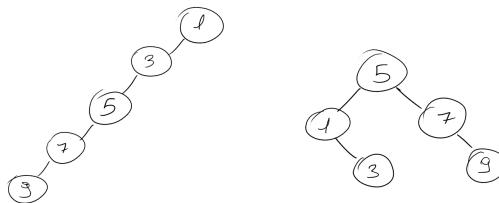
1 Arr A = [1, 3, 5, 7, 9]
2
3 buildBSTott(Arr A)
4     t = newTree()
5     t.root = buildBSTottAux(A, 1, A.length, NIL)
6     return t
7
8 buildBSTottAux(Arr A, int inf, int sup, Node padre)
9     if inf > sup
10        return NIL
11    else
12        med = (inf + sup) / 2
13        r = creaNodo(A[med])
14        r.p = padre
15        r.left = buildBSTottAux(A, inf, med - 1, r)
16        r.right = buildBSTottAux(A, med + 1, sup, r)
17        return r

```

Pre-condizione: L'array in input deve essere ordinato.

Nel nostro esempio, entrambe le funzioni hanno ricevuto come parametro un array ordinato, ma le due procedure ritornano degli alberi sostanzialmente differenti: l'algoritmo ottimizzato ritorna un albero binario di ricerca più bilanciato del primo, avente altezza $h = \Theta(\log n)$.

Nella figura sottostante è riportato a sinistra il risultato della prima procedura, a destra quello della seconda.



Calcoliamo il tempo d'esecuzione dell'algoritmo ottimizzato.

$$T(n) = \begin{cases} c & n = 0 \\ 2T\left(\frac{n}{2}\right) + d & n > 0 \end{cases}$$

Risolviamo la ricorrenza col Master Theorem:

$$\begin{aligned} a &= 2, & b &= 2, & n^{\log_b a} &= n^{\log_2 2} = n = g(n) \\ f(n) &= d \end{aligned}$$

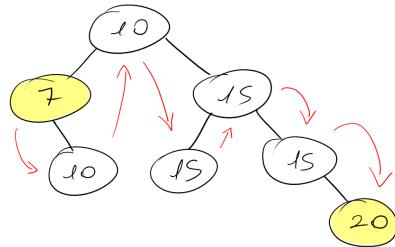
Verifichiamo di essere nel primo caso del Master Theorem: $f(n) = O(n^{1-\varepsilon})$, $\varepsilon > 0$. Se $\varepsilon = 1 \rightarrow n^{1-\varepsilon} = n^0 = 1$, dunque $T(n) = \Theta(n)$.

Nota Se l'array passato in input non dovesse essere ordinato, è sufficiente ordinarlo e applicare l'algoritmo ottimizzato, operazione che costa $\Theta(n \log n)$. Dimostreremo più avanti che l'operazione di creazione di un albero binario di ricerca partendo da un array non ordinato non può scendere sotto a $\Theta(n \log n)$.

12 Esercizi d'esame sugli alberi binari di ricerca

12.1 Conta nodi distinti

Dato un albero binario di ricerca T , scrivere un algoritmo efficiente che sostituisca il numero di elementi che occorrono una sola volta e analizzarne la complessità.



Nel caso di questo albero binario di ricerca, l'output del programma sarà 2 perché i nodi aventi chiave 7 e 20 occorrono una volta sola, mentre tutti gli altri nodi più di una.

Una strategia per risolvere questo problema può essere quello di una visita ordinata: trovo il valore minimo dell'albero, poi trovo per $n - 1$ volte il suo successore (n è il numero di nodi dell'albero) e controllo quante volte si ripetono i singoli valori. La complessità di questo algoritmo è lineare.

```

1 struct Tree{
2     PNode root;
3 };
4 typedef Tree* PTree;
5
6 int contaDistinti(PTree t){
7     int numDistinti = 0, count = 1, val;
8     PNode iter;
9     if (t->root == nullptr){
10         return 0;
11     }
12     iter = Tree_minimum(t->root);
13     val = iter->Key;
14     iter = Tree_successor(iter);
15     while (iter != nullptr){
16         if (iter->Key == val){
17             count++;
18         }
19         else{
20             if (count == 1){
21                 numDistinti++;
22             }
23             count = 1;
24             val = item->Key;
25         }
26     }
27     return numDistinti;
28 }
```

```

25     }
26     iter = Tree_successor(iter);
27 }
28 if (count == 1){
29     numDistinti++;
30 }
31 return numDistinti;
32 }
```

Complessità L'attraversamento simmetrico di un albero binario di ricerca di n nodi può essere implementato trovando l'elemento minimo dell'albero con la procedura `Tree_minimum` e poi effettuando $n - 1$ chiamate di `Tree_successor`. Dimostriamo che questo algoritmo è lineare, cioè che viene eseguito nel tempo $\Theta(n)$.

Dimostrazione La chiamata a `Tree_minimum` seguita da $n - 1$ chiamate a `Tree_successor` esegue esattamente una visita simmetrica come fa la procedura ricorsiva *in order*: questa infatti stampa prima `Tree_minimum` e per definizione il `Tree_successor` di un nodo è il prossimo nodo di una visita simmetrica. L'algoritmo ha tempo $\Theta(n)$ perché:

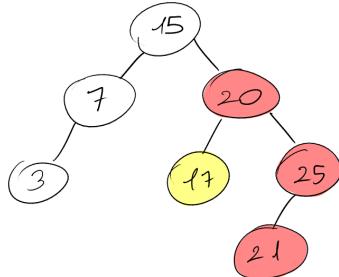
- Richiede sicuramente $\Omega(n)$ per effettuare le n chiamate di procedura.
- Attraversa ognuno dei $n - 1$ archi al più due volte, che richiede $O(n)$ (una volta scendendo l'albero, e una volta salendo).

Sia (u, v) un generico arco. Partendo dalla radice, dobbiamo attraversare l'arco (u, v) da u a v prima di attraversarlo da v a u . L'unico modo di attraversarlo "*downward*", verso il basso, è nella procedura `Tree_minimum`, mentre "*upward*" è nella procedura `Tree_successor` quando il nodo a cui applico `Tree_successor` non ha sottoalbero destro.

In conclusione, il tempo di esecuzione dell'algoritmo è $\Theta(n)$.

12.2 Elimina chiavi maggiori di k

Sia T un albero binario di ricerca contenente n chiavi intere distinte. Sia K una chiave di T . Si consideri il problema di eliminare da T tutte le chiavi maggiori di K .



Poniamo, per questo esempio, $K = 17$. Prima di tutto, è necessario osservare che conviene partire dalla radice per trovare il nodo contenente la chiave K ; in questo modo possiamo tagliare il nostro spazio di ricerca.

```

1 void eliminaMaggioriK(PTree t, int k){
2     PNode iter, temp;
3     iter = t->root;
4     while (iter->Key != K){
5         if (iter->Key > K){
6             /**/
7             rimuovi(iter->right);
8             iter->right = nullptr;
9             temp = iter;
10            transplant(t, iter, iter->left);
11            iter = iter->left;
12            delete temp;
13        }
14        else{
15            iter = iter->right;
16        }
17     /**/rimuovi(iter->right);
18     iter->right = nullptr;
19 }
20 void rimuovi(PNode u){
21     if (u != nullptr){
22         rimuovi(u->left);
23         rimuovi(u->right);
24         delete u;
25     }
26 }
27 }
```

Complessità La funzione `rimuovi` ha tempo d'esecuzione derivabile dallo studio delle sue ricorrenze:

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

Questa è la ricorrenza che abbiamo già trovato più volte nello studio degli algoritmi di visita degli alberi binari, e quindi sappiamo che per questa funzione $T(n) = \Theta(n)$.

Per quanto riguarda invece la funzione `eliminaMaggioriK`, il suo tempo d'esecuzione è $O(n)$ perché ogni nodo è visitato al più una volta.

Osservazione Se non si dovesse cancellare fisicamente i nodi ma solo logicamente, ponendo a `nullptr` i puntatori ai nodi di interesse (le relative istruzioni sono evidenziate nel codice con un commento vuoto all'inizio della relativa riga), quale sarebbe il tempo d'esecuzione?

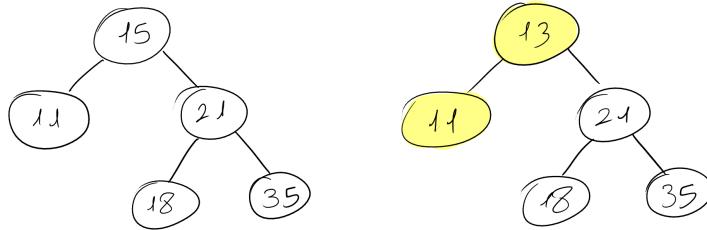
Sarebbe $O(h)$, dove h è l'altezza dell'albero, in quanto considereremmo solamente un cammino dalla radice al nodo che contiene K.

12.3 Check condition

Scrivere una funzione efficiente `check` che dato un albero binario di ricerca verifica se è soddisfatta la seguente condizione: per ogni intero K , se le chiavi K e $K + 2$ sono nell'albero, allora anche la chiave $K + 1$ è nell'albero. In termini matematici,

$$K \in T \wedge K + 2 \in T \rightarrow K + 1 \in T$$

Vediamo due esempi.



Nell'albero a sinistra la condizione è soddisfatta perché non sono presenti due nodi u e v tali che $u\text{-}>\text{Key} = K$ e $v\text{-}>\text{Key} = K + 2$ in modo da soddisfare la tesi. Al contrario, nell'albero a destra la condizione non è soddisfatta perché sono presenti due nodi u e v tali che $u\text{-}>\text{Key} = K$ e $v\text{-}>\text{Key} = K + 2$, ma non esiste un nodo w tale che $w\text{-}>\text{Key} = K + 1$.

```

1 bool check(PTree t){
2     PNode iter, succ;
3     bool ok{true};
4     iter = t->root;
5     if (iter == nullptr){
6         return true;           //Condizione vacuamente vera
7     }
8     iter = Tree_minimum(iter);
9     while (iter != nullptr && ok == true){
10         succ = Tree_successor(iter);
11         if (succ != nullptr && succ->Key == iter->Key + 2){
12             ok = false;
13         }
14         else{
15             iter = succ;
16         }
17     }
18     return ok;
19 }
```

Complessità Il tempo d'esecuzione dell'algoritmo è $O(n)$ perché stiamo applicando una visita andando a posizionarci sul minimo dell'albero e andando a eseguire, al più, $n - 1$ volte la funzione `Tree_successor`. La dimostrazione è la medesima del primo esercizio di questa sezione.

13 Cenni ad ulteriori alberi di ricerca

13.1 Alberi AVL

Sono alberi binari di ricerca bilanciati: oltre alla chiave, mantengono un'informazione aggiuntiva sul bilanciamento, il cosiddetto **fattore di bilanciamento**. Il fattore di bilanciamento di un nodo è la differenza fra l'altezza del suo sottoalbero sinistro e quella del suo sottoalbero destro. In un albero AVL, il valore assoluto del fattore di bilanciamento è minore o uguale ad 1 su ogni nodo.

Le operazioni di inserimento e di cancellazione sono più complesse in quanto necessitano di eseguire delle rotazioni per mantenere il bilanciamento dell'albero.

13.2 B_Alberi

Sono alberi di ricerca bilanciati di grado minimo t , con $t \geq 2$, che godono delle seguenti proprietà:

- Tutte le foglie hanno la stessa profondità.
- Ogni nodo v diverso dalla radice mantiene $K(v)$ chiavi ordinate

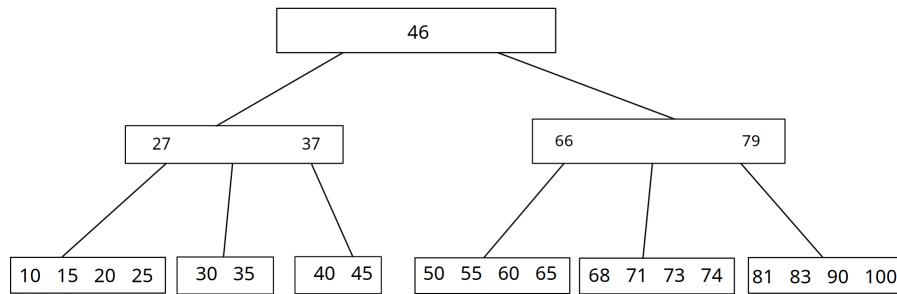
$$\text{Key}_1(v) \leq \text{Key}_2(v) \leq \dots \leq \text{Key}_{K(v)}(v)$$

tali che

$$t - 1 \leq K(v) \leq 2t - 1$$

- La radice mantiene almeno una chiave e al più $2t - 1$ chiavi ordinate.
- Ogni nodo interno v ha $K(v) + 1$ figli.
- Le chiavi $\text{Key}_i(v)$ separano gli intervalli di chiavi memorizzate in ciascun sottoalbero: se c_i è una qualunque chiave nell' i -esimo sottoalbero di un nodo v , allora

$$c_1 \leq \text{Key}_1(v) \leq c_2 \leq \text{Key}_2(v) \leq \dots \leq \text{Key}_{K(v)}(v) \leq c_{K(v)+1}$$



13.3 Alberi rossi e neri

Sono alberi binari di ricerca che contengono un'informazione aggiuntiva in ogni nodo che è il suo colore: rosso o nero. Vincolando in modo opportuno come si alternano i nodi rossi e i nodi neri, riusciamo a mantenere la seguente invariante: il cammino più lungo nell'albero è lungo al massimo il doppio del cammino più basso. Ne deriva un albero bilanciato.

14 Problema dell'ordinamento

Il problema dell'ordinamento è una delle questioni più importanti dell'algoritmica per diversi motivi.

- Consistono molto spesso in una *subroutine*: in molti casi l'ordinamento è utilizzato per ottenere delle soluzioni efficienti.
- Gli ordinamenti sono spesso basati sulla tecnica incrementale: utilizziamo una soluzione di dimensione k per costruirne una per un problema di dimensione $k + 1$.

Gli algoritmi di ordinamento sono caratterizzati dagli stessi *input* e *output*:

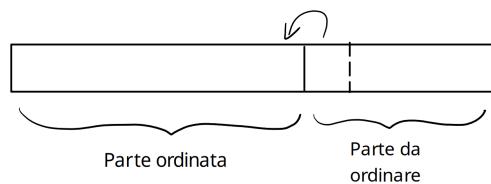
- **Input:** una sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$.
- **Output:** una permutazione $\langle a'_1, a'_2, \dots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Nello studio di questo argomento andremo ad utilizzare algoritmi basati sul confronto.

Premessa Non esiste un algoritmo basato sul confronto che esegua l'operazione di ordinamento inferiore a $O(n \log n)$.

14.1 Insertion Sort

L'*Insertion Sort* è un algoritmo basato sull'inserimento ed è di tipo incrementale: avendo k elementi già ordinati, vogliamo estendere questo insieme di elementi già ordinati al $(k + 1)$ -esimo elemento.



```

1  Insertion_sort(Array A)
2      for j = 2 to A.length
3          Key = A[j]
4          i = j - 1
5          while i > 0 AND Key < A[i]
6              A[i + 1] = A[i]
7              i = i - 1
8          A[i + 1] = Key

```

Spiegazione Prima di spiegare il funzionamento dell'algoritmo, può essere utile ricordare che in pseudocodice la posizione iniziale di un vettore è la numero 1. Di conseguenza, il ciclo **for** dell'algoritmo parte dalla seconda posizione dell'array.

L'algoritmo basa la sua correttezza su un **invariante**, secondo il quale la parte sinistra dell'array è ordinata. Per questo motivo, alla prima iterazione dell'array diamo per valido il fatto che il sottoarray costituito dal primo elemento sia ordinato. Il valore puntato da **j** diventa la chiave del nostro algoritmo: consiste nel primo elemento della parte non ordinata dell'array, quella destra, che vogliamo confrontare con tutti gli elementi della parte ordinata; quando troviamo un elemento minore di **Key** o raggiungo l'inizio dell'array, inseriamo il valore di **Key** nella posizione puntata da **i**, shiftando opportunamente a destra tutti gli elementi maggiori di **Key** nella parte ordinata. Raggiunta la fine del ciclo **while**, la parte ordinata sarà più grande di un elemento rispetto all'inizio del ciclo.

Nell'esempio di cui sotto, l'array che vogliamo ordinare è il seguente: [5, 2, 7, 3, 1]; è riportata una riga per ogni iterazione del ciclo **for** esterno, e le iterazioni del ciclo **while** sono rappresentate dalle frecce azzurre. Per ogni iterazione di questo ciclo, la chiave è l'elemento cerchiato in verde scuro; una croce arancione segna la fine del ciclo **while** (che, ripetiamo, può interrompersi quando troviamo un elemento più piccolo della chiave o quando raggiungiamo l'inizio dell'array) e una spunta verde chiaro indica la nuova posizione della chiave. Infine, la parte ordinata dall'array è sottolineata in rosso.

5 2 7 3 1

2 5 7 3 1

2 5 7 3 1

2 3 5 7 1

1 2 3 5 7

Analisi della correttezza

Invariante "Il sottoarray $A[1, \dots, j - 1]$ è formato dagli elementi ordinati che originariamente erano in posizione $A[1, \dots, j - 1]$ ".

Conclusione Quando il ciclo termina, l'invariante ci deve garantire una proprietà attraverso la quale possiamo dimostrare la correttezza dell'algoritmo; infatti, quando il ciclo `for` termina, l'indice j ha valore $A.length + 1 = n + 1$; poiché l'invariante è vero prima dell'inizio del ciclo, dopo ogni iterazione e dopo la sua fine, effettuiamo la seguente sostituzione:

$$INV\left[\frac{n+1}{j}\right]$$

"Il sottoarray $A[1, \dots, n + 1 - 1]$ è formato dagli elementi ordinati che originariamente erano in posizione $A[1, \dots, n + 1 - 1] \Rightarrow A[1, \dots, n]$ ". L'invariante al termine del ciclo ci garantisce la correttezza dell'algoritmo.

Teorema L'algoritmo *Insertion Sort* ordina *in loco* n elementi eseguendo, nel caso peggiore, $\Theta(n^2)$ confronti.

Dimostrazione Il ciclo esterno è eseguito $n-1$ volte, il numero di confronti è $\sum_{j=2}^n j - 1$. Effettuiamo il seguente cambio di variabile, $k = j - 1$, e riscriviamo il numero di confronti in funzione di k :

$$\sum_{j=2}^n j - 1 = \sum_{k=1}^{n-1} k = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2} = \Theta(n^2)$$

L'algoritmo è un ordinamento *in loco* perché un solo elemento è, in ogni istante, memorizzato all'esterno dell'array.

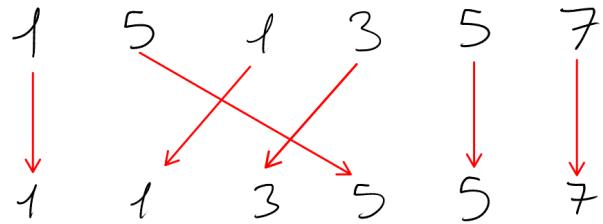
Nel caso migliore, cioè se l'array in input è ordinato in modo crescente, il tempo d'esecuzione è $\Theta(n)$.

Nel caso peggiore, cioè se l'array in input è ordinato in modo decrescente, il tempo d'esecuzione è $\Theta(n^2)$.

Definizione Un algoritmo *in loco* (o *sul posto*) è definito tale se, in ogni istante, al più un numero costante di elementi dell'array in input sono registrati all'esterno dell'array.

Vantaggi

- È un ordinamento *in loco*.
- Si tratta di un metodo stabile: i numeri che riceve in input con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano in quello di input.



- È sensibile all'ordinamento dell'input: se trova un elemento maggiore dell'array già ordinato, lo capisce e non va ad eseguire il ciclo interno.

Svantaggi

- Nel caso peggiore, la complessità risulta essere $O(n^2)$. Per questo motivo, l'*Insertion Sort* viene spesso utilizzato come algoritmo ausiliario per ordinare array di piccole dimensioni. Altre alternative sono il *Merge Sort* e il *Quick Sort*.

Good practices

- Usare l'*Insertion Sort* per input di piccole dimensioni.

14.2 Merge Sort

Il *Merge Sort* è un algoritmo di ordinamento basato sulla tecnica *divide et impera*.

Parametro di questo algoritmo è un array $A[p \dots r]$, dove inizialmente $p = 1$ e $r = A.length$.

Passi dell'algoritmo

1. **Divide:** divide l'array in due sottoarray $A[p \dots q]$ e $A[q + 1 \dots r]$ dove q è l'indice di mezzo di $A[p \dots r]$.
2. **Impera:** ordina i due sottoarray in modo ricorsivo utilizzando l'algoritmo `mergesort`. Se il problema è sufficientemente piccolo si risolve direttamente.
3. **Combina:** fonde i due sottoarray che sappiamo essere ordinati per generare un singolo vettore ordinato $A[p \dots r]$.

```

1 mergesort(array A, int p, int r)
2     if p < r
3         med = (p + r) / 2
4         mergesort(A, p, med)
5         mergesort(A, med + 1, r)
6         merge(A, p, q, r)
7
8     merge(array A, int p, int q, int r)
9         n1 = q - p + 1
10        n2 = r - q
11        crea due array di appoggio L[1 ... n1 + 1] e R[1 ... n2 + 1]
12        for i = 1 to n1
13            L[i] = A[p + i - 1]
14        for j = 1 to n2
15            R[j] = A[q + j]
16        L[n1 + 1] = infinito
17        R[n2 + 1] = infinito
18        i = 1, j = 1
19        for k = p to r
20            if L[i] <= R[j]
21                A[k] = L[i]
22                i++
23            else
24                A[k] = R[j]
25                j++

```

Spiegazione L'algoritmo basa il suo funzionamento sulla ricorsione: la funzione principale consiste in due chiamate ricorsive a sé stesso: con la prima riordina la metà sinistra dell'array, con la seconda la metà destra. Questa funzione principale è molto semplice, in quanto il riordinamento vero e proprio dell'array non è neanche fisicamente eseguito all'interno di questa, bensì nella funzione di ricombinazione degli array; pertanto, il punto sul quale ci dobbiamo focalizzare non è tanto il metodo di riordinamento in sé delle chiamate, bensì

l'effetto che queste hanno sull'array: noi sappiamo, per ipotesi, che dopo queste chiamate le due metà degli array sono ordinate. Se vogliamo, anche se non è stato esplicitamente detto dalla Professoressa a lezione, l'ordinamento vero e proprio avviene all'interno della funzione `merge`.

Questa funzione infatti riceve come input l'array e gli indici al suo inizio, alla sua fine e alla sua metà. Sappiamo per ipotesi che $p \leq q < r$, che il sottoarray $A[p \dots q]$ è ordinato, che il sottoarray $A[q + 1 \dots r]$ è ordinato e che nessuno dei due è vuoto per le condizioni su p , q e r . La funzione produce in output un unico array ordinato dato dalla fusione dei due sottoarray.

La procedura funziona nel seguente modo: vengono creati due array di appoggio aventi dimensione metà dell'array in ingresso. Il primo di questi contiene gli elementi dell'array a sinistra di q , l'altro quelli a destra. Come ultimo elemento di entrambi gli array viene messo un valore arbitrario, noi abbiamo deciso ∞ , che funge da **elemento sentinella** e ha lo scopo di semplificare il codice e dimostrarne la correttezza. Ci torneremo più avanti.

L'unione degli elementi dei due array avviene nell'ultimo ciclo `for` della procedura: un indice scorre l'array risultato in tutta la sua lunghezza, scrivendo man mano nella k -esima posizione dell'array il minore tra l' i -esimo elemento dell'array di sinistra e il j -esimo elemento dell'array di destra; ogni volta che viene inserito un elemento appartenente all'array di sinistra, viene incrementato il relativo indice i , viceversa per l'array di destra viene incrementato l'indice j .

Analisi della correttezza

Invariante "Il sottoarray $A[p \dots k - 1]$ contiene ordinati i $k - p$ elementi più piccoli di $L[1 \dots n_1 + 1]$ e $R[1 \dots n_2 + 1]$. Inoltre, $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro rispettivi array che non sono stati ancora copiati in A ."

Conclusione Alla fine del ciclo, $k = r + 1$ e vale il nostro invariante:

$$INV \left[\frac{r + 1}{k} \right]$$

"Il sottoarray $A[p \dots r + 1 - 1] = A[p \dots r]$ contiene ordinati i $r + 1 - p$ elementi più piccoli di $L[1 \dots n_1 + 1]$ e $R[1 \dots n_2 + 1]$. Inoltre, $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro array che non sono stati copiati in A ."

Questo vuol dire che il vettore $A[p \dots r]$ è ordinato, e gli unici elementi non copiati sono le due sentinelle.

Miglioramenti

- Usare una funzione *wrapper* che, in caso di input già parzialmente ordinati e/o di piccole dimensioni (solitamente dai 5 ai 25 elementi), permetta di riordinare l'array attraverso l'*Insertion Sort*. Spieghiamo perché questa strategia può tornare utile nell'analisi della complessità dell'algoritmo.

Analisi della complessità

merge Il tempo d'esecuzione della `merge` è si può derivare come segue:

- $\Theta(n_1)$ per il ciclo `for` per riempire l'array `L` (righe 12-13)
- $\Theta(n_2)$ per il ciclo `for` per riempire l'array `R` (righe 14-15)
- $\Theta(r - p + 1)$ per il ciclo `for` che congiunge i due array (righe 19-25)
- $O(1)$ per le altre operazioni costanti

Di conseguenza,

$$\begin{aligned} T(n) &= O(1) + \Theta(n_1) + \Theta(n_2) + \Theta(r - p + 1) \\ &= \Theta(n_1 + n_2) + \Theta(r - p + 1) \end{aligned}$$

Poiché $n_1 = q - p + 1$, $n_2 = r - q$ e $n = r - p + 1$,

$$\begin{aligned} T(n) &= \Theta(n_1 + n_2) + \Theta(r - p + 1) \\ &= \Theta(q - p + 1 + r - q) + \Theta(n) \\ &= \Theta(r - p + 1) + \Theta(n) \\ &= \Theta(n) + \Theta(n) = \Theta(n) \end{aligned}$$

mergesort Il tempo di esecuzione di `mergesort` è il seguente:

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

dove il $\Theta(n)$ si deve alla complessità di `merge`. Applichiamo il Master Theorem:

$$\begin{aligned} f(n) &= \Theta(n) \\ n^{\log_b a} &= n^{\log_2 2} = n^1 = n = \Theta(n) \end{aligned}$$

Siamo nel secondo caso, in quanto $f(n)$ è dello stesso ordine di grandezza di $n^{\log_b a}$. Di conseguenza,

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

Vantaggi

- $T(n) = \Theta(n \log n)$.
- È un metodo stabile.

Svantaggi

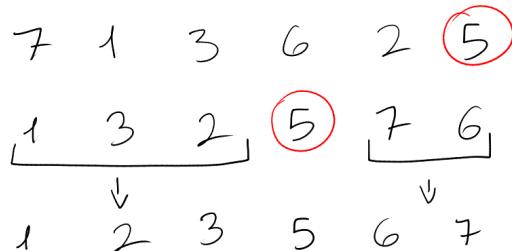
- Non è *in loco*: utilizza un vettore ausiliario proporzionale al numero di elementi da ordinare.
- Il tempo di calcolo del `mergesort` dipende essenzialmente dal numero di elementi da ordinare e non trae beneficio da eventuali ordinamenti dell'input, come fa l'*Insertion Sort*.

14.3 Quick Sort

Si tratta di un algoritmo di ordinamento basato sulla tecnica del *divide et impera*. Serve per ordinare un vettore $A[p \dots r]$, in cui inizialmente $p = 1$ e $r = A.length$.

Passi dell'algoritmo

- **Divide:** nel *Merge Sort* era banale perché andava a prendere l'indice dell'elemento in mezzo all'array, mentre ora fa un'operazione diversa. Il *Quick Sort* partiziona l'array di partenza $A[p \dots r]$ in due sottoarray $A[p \dots q - 1]$ e $A[q + 1 \dots r]$, due partizioni che possono essere eventualmente vuote e in cui, appunto, l'elemento alla posizione q non è compreso. Queste partizioni sono tali che ogni elemento di $A[p \dots q - 1]$ è minore o uguale a $A[q]$ che a sua volta è minore o uguale ad ogni elemento di $A[q + 1 \dots r]$. L'indice q è il risultato di questa procedura di partizionamento e l'elemento $A[q]$ è chiamato **pivot**.
- **Impera:** ordino i due sottoarray $A[p \dots q - 1]$ e $A[q + 1 \dots r]$ chiamando ricorsivamente la procedura `quicksort`. Se il problema è sufficientemente piccolo si risolve direttamente.
- **Combina:** è triviale, poiché i sottoarray sono ordinati sul posto, non occorre altro lavoro per combinarli. L'intero array $A[p \dots r]$ è ordinato.



```

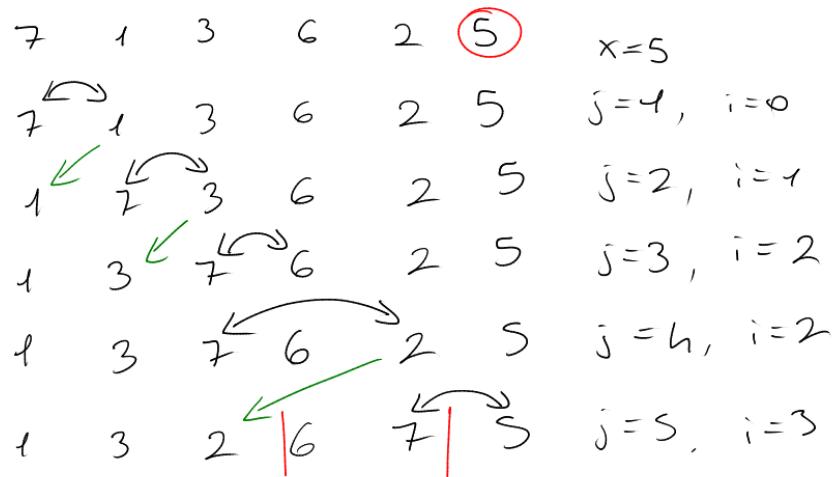
1 quicksort(array A, int p, int r)
2     if p < r
3         q = partition(A, p, r)
4         quicksort(A, p, q - 1)
5         quicksort(A, q + 1, r)
6
7 partition(array A, int p, int r) -> int
8     x = A[r]      //Per noi il pivot e' l'ultimo elemento
9     i = p - 1
10    for j = p to r - 1
11        if A[j] <= x
12            i = i + 1
13            scambia A[i] e A[j]
14    scambia A[i + 1] e A[r]
15    return i + 1

```

Spiegazione La prima istruzione dell'algoritmo consiste in una chiamata alla procedura **partition**, funzione che ritorna il valore contenuto nell'ultima posizione dell'array e svolge una serie di operazioni su di questo per inizializzare l'ordinamento, che risulta essere *in loco*.

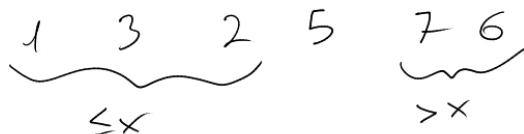
Per prima cosa, vengono confrontati tutti gli elementi dell'array con questo elemento, il **pivot**. Se questi elementi sono minori o uguali del **pivot**, vengono spostati nella parte sinistra dell'array, altrimenti restano nella loro posizione. Contando il numero di spostamenti fatti attraverso la variabile i , alla fine delle operazioni di scambio, $i + 1$ conterrà l'indice della posizione a sinistra della quale l'array conterrà solo gli elementi minori o uguali del **pivot**, mentre a destra quelli maggiori. Come ultima operazione, la **partition** scambia il **pivot** con l'ultimo elemento dell'array, e ritorna l'indice del **pivot**.

L'algoritmo termina con due chiamate ricorsive all'algoritmo stesso, una avente come range d'azione il sottoarray sinistro, l'altra il sottoarray destro.



Fine ciclo: $j = 6, i = 3$

Scambio $A[r]$ con $A[i+1-h]$



Vantaggi

- Si tratta di un ordinamento *in loco*.

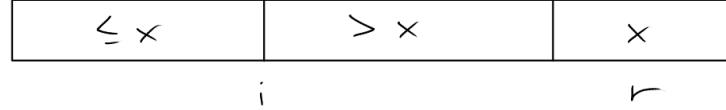
Dimostrazione della correttezza

Invariante

$$\begin{aligned} INV \equiv & x = A[r] \wedge \forall k \in [p \dots i] \rightarrow A[k] \leq x \wedge \\ & \wedge \forall k \in [i + 1 \dots j - 1] \rightarrow A[k] > x \wedge \\ & \wedge p \leq j \leq r \wedge p - 1 \leq i \leq j - 1 \end{aligned}$$

Conclusione Alla fine del ciclo `for, j = r`:

$$\begin{aligned} INT\left[\frac{r}{j}\right] \equiv & x = A[r] \wedge \forall k \in [p \dots i] \rightarrow A[k] \leq x \wedge \\ & \wedge \forall k \in [i + 1 \dots r - 1] \rightarrow A[k] > x \wedge \\ & \wedge p \leq r \leq r \wedge p - 1 \leq i \leq r - 1 \end{aligned}$$



Abbiamo così ripartito i valori dell'input in tre parti: quelli minori o uguali a x , quelli maggiori di x e una parte costituita dal solo elemento x . Le ultime due righe inseriscono il *pivot* x nella posizione corretta, cioè come elemento più a sinistra della porzione che contiene i valori maggiori di x .

Analisi della complessità

partition Sia n il numero di elementi nella porzione di array che la `partition` riceve in input. Allora, la complessità della procedura `partition` è $\Theta(n)$, per via del ciclo `for` che impiega $r - p + 1 = n$ iterazioni.

quicksort Sia k il numero di elementi del sottoarray che va da p a $q - 1$ e $n - 1 - k$ il numero di elementi del sottoarray che va da $q + 1$ a r (si noti che il -1 si deve al fatto che il *pivot* non è incluso in nessuno dei due sottoarray). Allora,

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

dove il $\Theta(n)$ è dovuto alla `partition`.

Vanno affrontati in modo separato i casi peggiore, migliore e medio, in quanto il tempo d'esecuzione del *Quick Sort* dipende strettamente dal partizionamento dei due sottoarray.

- **Caso peggiore**

In questo caso, una partizione contiene $n - 1$ elementi e l'altra non ne contiene nessuno; questa situazione si verifica ad ogni chiamata ricorsiva dell'algoritmo.

Possiamo procedere a risolvere la ricorrenza con il metodo di sostituzione:

$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n) = T(n - 1) + cn$$

$$\begin{aligned} T(n) &= T(n - 1) + cn = T(n - 2) + c(n - 1) + cn \\ &= \dots \\ &= \sum_{i=1}^n ci = c \frac{n(n + 1)}{2} = \Theta(n^2) \end{aligned}$$

Otteniamo tale tempo d'esecuzione quando il vettore è già ordinato.

- **Caso migliore**

In questo caso, un sottoarray contiene $\lfloor \frac{n}{2} \rfloor$ elementi e l'altro $\lfloor \frac{n}{2} \rfloor - 1$ elementi.

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

Notiamo che si tratta della stessa ricorrenza del *Merge Sort*, che possiamo risolvere col Master Theorem:

$$\begin{aligned} f(n) &= \Theta(n) \\ n^{\log_b a} &= n^{\log_2 2} = n^1 = n = \Theta(n) \end{aligned}$$

Siamo nel secondo caso del Master Theorem, in quanto $n^{\log_b a}$ è dello stesso ordine di $f(n)$:

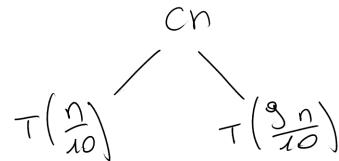
$$T(n) = \Theta(n \log n)$$

- **Caso medio**

Supponiamo che l'algoritmo produca sempre una ripartizione proporzionale 9-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn$$

Non possiamo applicare il Master Theorem; decidiamo di risolvere la ricorrenza attraverso l'albero delle ricorrenze:

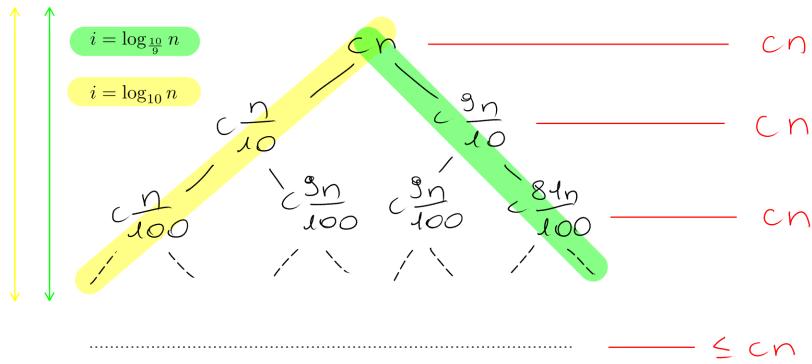


dove cn è il costo al livello più alto.

Ma noi sappiamo che

$$\begin{aligned} T\left(\frac{n}{10}\right) &= T\left(\frac{n}{100}\right) + T\left(\frac{9n}{100}\right) + c\frac{n}{10} \\ T\left(\frac{9n}{10}\right) &= T\left(\frac{9n}{100}\right) + T\left(\frac{81n}{100}\right) + c\frac{9n}{10} \end{aligned}$$

Quindi possiamo disegnare il seguente albero delle ricorsioni.



Dal momento che il numero di elementi per sottoarray ad ogni chiamata non è bilanciato, ci sarà un ramo delle ricorsioni che terminerà prima di altri, e uno più lungo che determinerà il costo temporale complessivo dell'algoritmo.

La somma delle complessità ai primi livelli è pari a cn , mentre sappiamo per certo, proprio perché alcuni cammini della ricorsione terminano prima di altri, che ai livelli inferiori il costo sarà inferiore a cn . Per trovare le lunghezze dei cammini, poniamo le seguenti condizioni ai nodi dell'albero che decrescono in modo omogeneo, basandoci sulla condizione del caso base:

$$\begin{aligned} \frac{n}{10^i} &= 1 \Rightarrow n = 10^i \Rightarrow i = \log_{10} n \\ \left(\frac{9}{10}\right)^i n &= 1 \Rightarrow n = \left(\frac{10}{9}\right)^i \Rightarrow i = \log_{\frac{10}{9}} n \end{aligned}$$

Il primo risultato è la lunghezza del cammino più corto, il secondo di quello più lungo.

Quindi l'altezza dell'albero è $\log_{\frac{10}{9}} n$, e il costo totale è:

$$T(n) \leq cn \cdot \log_{\frac{10}{9}} n = O(n \log n)$$

Generalizzando ad un caso di una qualsiasi proporzionalità costante:

$$T(n) = T(\alpha n) + T((1 - \alpha)n) + cn, \quad 0 < \alpha < 1, c > 0$$

Nel caso medio, dunque, l'algoritmo produce un albero di ricorsione di profondità $\Theta(n \log n)$, dove il costo di ogni livello è $O(n)$. Il tempo d'esecuzione si ottiene moltiplicando il costo di un livello per l'altezza del nostro albero, ma non essendo tutti i cammini della stessa lunghezza, questo è limitato superiormente da $n \log n$: quindi $T(n) = O(n \log n)$ quando la ripartizione ha proporzionalità costante.

- **Caso medio: proporzioni alterne** Ma cosa succede nel caso in cui le partizioni si alternano tra buone e cattive (bilanciate e sbilanciate)? Supponiamo di avere due possibili ricorrenze, una che chiameremo *Lucky*, $L(n)$, se esiste una proporzione bilanciata tra gli elementi dei sottoarray, e un'altra che chiameremo *Unlucky*, $U(n)$, altrimenti. Esprimiamole di seguito.

$$\begin{aligned} L(n) &= U\left(\frac{n}{2}\right) + \Theta(n) \\ U(n) &= L(n-1) + \Theta(n) \end{aligned}$$

Possiamo eseguire le seguenti operazioni algebriche:

$$\begin{aligned} L(n) &= 2\left(L\left(\frac{n}{2}-1\right) + \Theta\left(\frac{n}{2}\right)\right) + \Theta(n) \\ &= 2L\left(\frac{n}{2}-1\right) + 2\Theta\left(\frac{n}{2}\right) + \Theta(n) \\ &= 2L\left(\frac{n}{2}-1\right) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Quindi, anche se si alternano partizioni buone e cattive, nel caso medio la complessità è comunque $\Theta(n \log n)$. Per questo motivo, nonostante il caso peggiore sia $\Theta(n^2)$, è un algoritmo usato spesso.

14.3.1 Quick Sort Randomizzato

Cosa succederebbe se andassimo a scegliere un *pivot* casuale?

Invece di scegliere sempre $A[r]$ come *pivot*, scambieremo l'elemento $A[r]$ con un elemento scelto a caso da $A[p \dots r]$. Questo serve per abbassare la probabilità che si presenti la situazione del caso peggiore.

```

1 randomized_quicksort(array A, int p, int r)
2     if p < r
3         q = randomized_partition(A, p, r)
4         randomized_quicksort(A, p, q - 1)
5         randomized_quicksort(A, q + 1, p)
6
7 randomized_partition(array A, int p, int r) -> int
8     i = random(p, r)
9     scambia A[i] con A[r]
10    return partition(A, p, r)

```

dove *partition* è la stessa procedura vista nel *Quick Sort* generale.

Vantaggi

Assunzione: le chiavi sono distinte.

- Il tempo di esecuzione è indipendente dall'ordinamento dell'input.
- Non è necessario fare alcuna assunzione sulla distribuzione dell'input.
- Nessun specifico input può portare al caso pessimo.
- Il caso peggiore è determinato solamente dal generatore di numeri casuali.

Per tutti questi motivi, il `randomized_quicksort` è dalle 3 alle 4 volte più veloce della controparte generale.

Miglioramenti

1. Insertion Sort

Analogamente al *Merge Sort*, si può utilizzare l'*Insertion Sort* su vettori di piccole dimensioni. Possiamo modificare la procedura inserendo il seguente caso base, che ci garantisce migliori prestazioni per array con pochi elementi in input.

```
1 if r - p <= M
2     insertion_sort(A, p, r)
```

dove M è un valore arbitrario che dipende dalle implementazioni, solitamente $5 \leq M \leq 25$, sostituendo la condizione dell'implementazione precedente:

```
1 if p < r
2     ...
```

Un'altra possibile ottimizzazione si può ottenere inserendo il seguente caso base:

```
1 if r - p <= M
2     return
```

e utilizzando una funzione *wrapper* come la seguente:

```
1 void sort(array A, int p, int r)
2     quicksort(A, p, r)
3     insertion_sort(A, p, r)
```

In questo modo, l'*Insertion Sort* riceve un input quasi ordinato, con il quale sappiamo che l'*Insertion Sort* lavora particolarmente bene.

2. Mediana

Un'ottimizzazione si può ottenere scegliendo il *pivot* come la mediana di 3 elementi del vettore; in particolare, si può scegliere di prendere un elemento dalla parte sinistra del vettore, uno al suo centro e uno nella sua parte destra.

3. Chiavi duplicate

Se tutte le chiavi sono uguali, la randomizzazione non aiuta: avremo sempre due array fortemente sbilanciati nelle loro dimensioni.

Invece di andare a partizionare il vettore in elementi che risultano minori di x , dove $x = A[A.length]$, possiamo costruire un'ulteriore partizione contenente tutti gli elementi uguali ad x , che sappiamo per certo essere già ordinati. Perciò potremo applicare il Quick Sort limitatamente ai sottoarray contenenti chiavi diverse da x .

$< x$	$= x$	$> x$	x
-------	-------	-------	-----

Effettuare la partizione è compito della procedura

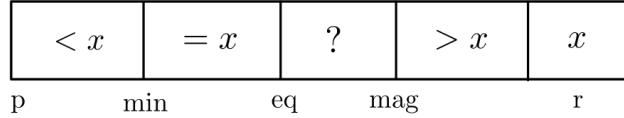
```
1 partition(array A, int p, int r)
```

che permuta gli elementi di $A[p \dots r]$ e restituisce una coppia di indici q e t che delimita la partizione in cui gli elementi compresi tra q e t sono uguali a x ; in particolare,

- $p \leq q \leq t \leq r$
- Tutti gli elementi di $A[q \dots t]$ sono uguali al *pivot*.
- Tutti gli elementi di $A[p \dots q - 1]$ sono minori di $A[q]$.
- Tutti gli elementi di $A[t + 1 \dots r]$ sono maggiori di $A[q]$.
- *partition* dovrà richiedere un tempo $\Theta(r - p)$.

```
1 partition(array A, int p, int r)
2     x = A[r]
3     min = eq = p
4     mag = r
5     while eq < mag
6         if A[eq] < x
7             scambia A[eq] con A[min]
8             min = min + 1
9             eq = eq + 1
10        else
11            if A[eq] == x
12                eq = eq + 1
13            else
14                mag = mag - 1
15                scambia A[eq] con A[mag]
16    scambia A[r] con A[mag]
17    return <min,mag>
```

In questa implementazione, il *pivot* viene nuovamente scelto come l'ultimo elemento dell'array. L'algoritmo vuole riordinare un array, servendosi di un totale di 5 partizioni impostate come segue.



- Da p a \min non compreso sono presenti tutti gli elementi strettamente minori del *pivot*.
- Da \min a \eq non compreso sono presenti tutti gli elementi uguali al *pivot*.
- Da \eq a \mag non compreso sono presenti tutti gli elementi che non sappiamo ancora in che partizione appartengono, perché vengono comparati man mano che procede l'algoritmo: alla fine della procedura, infatti, \eq e \mag avranno lo stesso valore.
- Da \mag a r non compreso sono presenti tutti gli elementi strettamente maggiori del *pivot*.
- In posizione r è presente il *pivot*.

Le comparazioni vengono fatte nel seguente modo: ad ogni iterazione, vengono effettuati dei controlli sul valore attualmente puntato dall'indice \eq (inizializzato col valore di p) e paragonato al *pivot*. Se minore di esso, vengono aumentati gli indice \eq e \min ; se uguale ad esso, viene solamente incrementato \eq ; se maggiore di esso, viene decrementato \mag e si scambiano di posto gli elementi puntati dagli indici \eq e \mag .

Il ciclo prosegue finché \eq e \mag non raggiungono lo stesso valore. Infatti, con questa proprietà, possiamo dimostrare la correttezza dell'algoritmo. Infine, con la penultima istruzione vengono scambiati di posto il primo elemento

Analisi della correttezza

Invariante

$$\begin{aligned}
 INV \equiv & x = A[r] \wedge \\
 & \wedge \forall k \in [p \dots \min), A[k] < x \wedge \\
 & \wedge \forall k \in [\min \dots \eq), A[k] = x \wedge \\
 & \wedge \forall k \in [\mag \dots r), A[k] > x \wedge \\
 & \wedge p \leq \min \leq \eq \leq \mag \leq r
 \end{aligned}$$

Conclusione Quando il ciclo termina, $\text{eq} == \text{mag}$. Possiamo riscrivere l'invariante nel seguente modo:

$$\begin{aligned} \text{INV} \left[\frac{\text{mag}}{\text{eq}} \right] \equiv & x = A[r] \wedge \\ & \wedge \forall k \in [\text{p} \dots \text{min}), A[k] < x \wedge \\ & \wedge \forall k \in [\text{min} \dots \text{mag}), A[k] = x \wedge \\ & \wedge \forall k \in [\text{mag} \dots \text{r}), A[k] > x \\ & \wedge \text{p} \leq \text{min} \leq \text{mag} \leq \text{r} \end{aligned}$$

Con la penultima istruzione andiamo a posizionare x in modo che il vettore sia diviso in tre partizioni:

$< x$	$= x$	$> x$
-------	-------	-------

La presente ottimizzazione si può implementare nel seguente modo:

```

1 quicksort(array A, int p, int r)
2     if p < r
3         <q,t> = partition(A, p, r)
4         quicksort(A, p, q - 1)
5         quicksort(A, t + 1, r)

```

Analisi della complessità Il tempo d'esecuzione di `partition` è $\Theta(r - p)$. Nel caso in cui tutti gli elementi siano uguali, andremo ad eseguire la `partition` e il *Quick Sort* verrà chiamato su due sottoarray vuoti: la complessità sarà $\Theta(n)$ perché la partition viene eseguita una volta sola su tutta la lunghezza dell'array.

Vantaggi

- È un algoritmo in loco.
- Nel caso medio il tempo d'esecuzione è $O(n \log n)$.

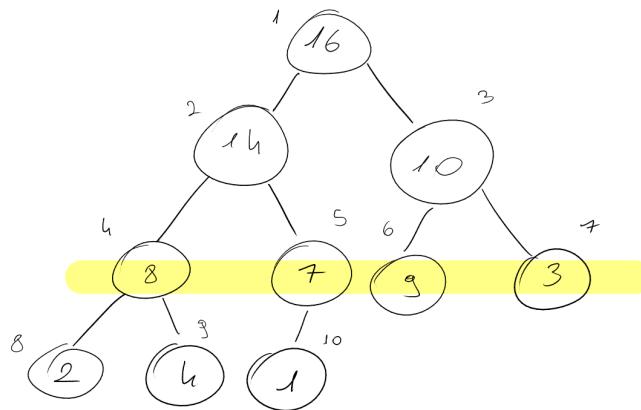
Svantaggi

- Nel caso peggiore il tempo d'esecuzione è $O(n^2)$.
- Non è stabile.

14.4 Heap e Heap Sort

L'*Heap Sort* è un algoritmo di ordinamento basato sul confronto che fa uso di una struttura dati detta *heap*.

Un **heap** (binario) è un albero quasi completo, cioè un albero in cui tutti i suoi livelli sono completamente riempiti tranne eventualmente l'ultimo, in cui tutte le foglie sono addossate a sinistra.



Un heap può essere facilmente memorizzato in un **array posizionale A** contenente due attributi:

- **A.length** indica il numero di elementi dell'array, la sua dimensione.
- **A.heap_size** indica il numero di elementi dell'heap che sono memorizzati in A. Vale la seguente proprietà: $A.heap_size \leq A.length$.

Essendo il vettore posizionale, la radice dell'albero è memorizzata nella prima posizione dell'array, **A[1]** (ricordiamo che in pseudocodice l'indice di partenza è 1), e gli altri nodi si trovano nelle posizioni **A[2] ... A.heap_size**.

Sia **i** l'indice di un nodo. Allora, le seguenti funzioni ci permettono di recuperare padre, figlio sinistro e figlio destro di **i**.

```

1 parent(Node i) -> Node
2     return floor(i/2)
3
4 left(Node i) -> Node
5     return 2 * i
6
7 right(Node i) -> Node
8     return 2 * i + 1
  
```

Queste operazioni sono eseguite dal calcolatore in maniera quasi istantanea, dal momento che per eseguire una moltiplicazione o una divisione per una potenza di 2 un calcolatore esegue uno shift a sinistra di **n** bit, dove **n** è l'esponente del dividendo (1 nel caso delle funzioni di cui sopra) per eseguire una moltiplicazione, mentre esegue uno shift a destra di **n** bit per eseguire una divisione.

Dunque, la procedura `parent` consiste nell'eseguire uno shift a destra di 1 bit su i , la procedura `left` nell'eseguire uno shift a sinistra di 1 bit su i e la procedura `right` nell'eseguire uno shift a sinistra di 1 bit su i e nell'impostare il bit meno significativo (quello più a destra) ad 1.

Esistono due tipi di heap, in base a quale proprietà dei nodi soddisfano:

1. **max_heap**

Per ogni nodo i diverso dalla radice, si ha la seguente proprietà:

$$A[\text{parent}(i)] \geq A[i]$$

Si può dimostrare per induzione e per transitività dell'operazione \geq che la proprietà del max_heap garantisce che il massimo elemento di un max_heap si trova nella radice e che il sottoalbero di un nodo contiene valori non maggiori del valore contenuto nel nodo stesso.

2. **min_heap**

Per ogni nodo i diverso dalla radice, si ha la seguente proprietà:

$$A[\text{parent}(i)] \leq A[i]$$

Si può dimostrare per induzione e per transitività dell'operazione \leq che la proprietà del min_heap garantisce che il minimo elemento di un min_heap si trova nella radice e che il sottoalbero di un nodo contiene valori non minori del valore contenuto nel nodo stesso.

Proprietà degli heap

Lemma 1. L'altezza di un heap di n elementi è $\lfloor \log n \rfloor$.

Dimostrazione

Un heap è un albero quasi completo. Se ha altezza h , allora il numero di nodi si può limitare inferiormente e superiormente in questo modo:

$$\begin{aligned} n &\leq \sum_{i=0}^h 2^i = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1 && \text{se l'albero è completo} \\ n &\geq \sum_{i=0}^{h-1} 2^i + 1 = 2^{h-1+1} - 1 + 1 = 2^h && \text{altrimenti} \\ 2^h &\leq n \leq 2^{h+1} - 1 < 2^{h+1} \\ h &\leq \log n \leq h + 1 \end{aligned}$$

Dal momento che h è un intero, allora vale $h = \lfloor \log n \rfloor$ per definizione.

Lemma 2. Nell'array che rappresenta un heap di n elementi, le foglie sono i nodi con indici

$$\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2, \dots, n$$

La dimostrazione avviene per induzione sul numero di nodi n . Hint: se uno di questi nodi avesse un figlio, il suo indice si troverebbe oltre la fine dell'array.

Questo comporta che le foglie di un heap costituiscono al più la metà degli elementi dell'heap.

Lemma 3. Ci sono al massimo $\lceil \frac{n}{2^{h+1}} \rceil$ nodi di altezza h in un qualsiasi heap di n elementi.

Conseguenza di questo lemma è che ci sono tanti nodi che hanno altezze molto piccole: prendendo $h = 0$, cioè il livello dei nodi foglia, si ha che

$$n_{foglie} \leq \left\lceil \frac{n}{2} \right\rceil$$

Se l'heap è completo, allora le foglie sono a tutti gli effetti pari a $\lceil \frac{n}{2} \rceil$.

Operazioni su max_heap

max_heapify

È una procedura che serve per mantenere la proprietà dei max_heap.

Pre-condizione: gli alberi binari con radice in `left(i)` e `right(i)` sono max_heap.

Post-condizione: l'albero radicato in `i` è un max_heap.

```

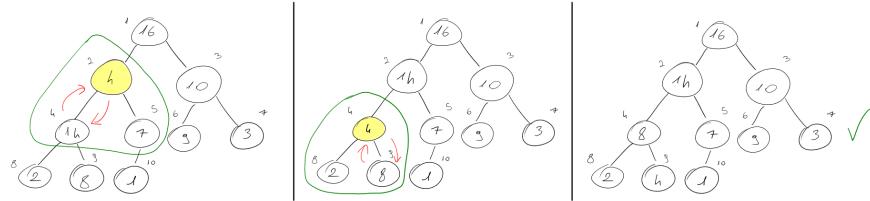
1 max_heapify(Heap A, Node i)
2     l = left(i)
3     r = right(i)
4     if l <= A.heap_size AND A[l] > A[i]
5         massimo = l
6     else
7         massimo = i
8         if r <= A.heap_size AND A[r] > A[massimo]
9             massimo = r
10        if i != massimo
11            scambia A[i] e A[massimo]
12            max_heapify(A, massimo)

```

L'algoritmo di verifica della proprietà dei max_heap consiste in un controllo che considera tre nodi alla volta: quello passato come parametro e i suoi figli. Prima di tutto si ottengono gli indici dei figli, si verifica che siano presenti nell'albero (cioè, che `i` non sia una foglia) e si comparano i loro valori, salvando l'indice del nodo avente il valore massimo. Se il valore massimo non è salvato nel nodo padre, allora vengono scambiati il nodo padre e il figlio col valore massimo, e infine viene chiamata ricorsivamente la procedura sull'indice del figlio avente il valore massimo.

Analisi della complessità Il tempo d'esecuzione è $O(h)$, dove h è l'altezza del nodo su cui chiamiamo la procedura, perché ad ogni chiamata ricorsiva scendiamo di un livello fino ai nodi foglia. Poiché un heap ha altezza $\lfloor \log n \rfloor$,

se la procedura viene chiamata sulla radice, allora il suo tempo d'esecuzione è $O(\log n)$.



Costruzione di un max_heap

Dato un vettore disordinato, vogliamo trasformarlo in un max_heap.

```

1 build_max_heap(array A)
2     A.heap_size = A.length
3     for i = floor(A.length/2) down to 1
4         max_heapify(A, i)

```

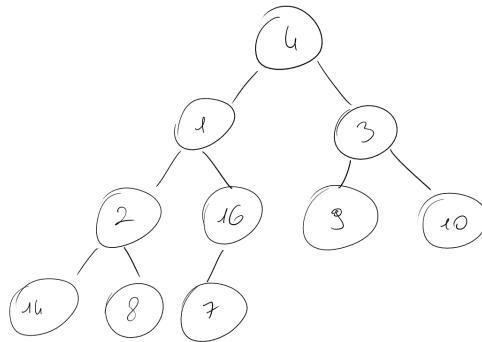
Osservazioni

- Una foglia è un max_heap banale, per cui...
- ...partiamo a verificare la proprietà da $A.length/2$ perché oltre questa soglia sappiamo per certo, essendo nodi foglia, che la proprietà di max_heap è verificata.

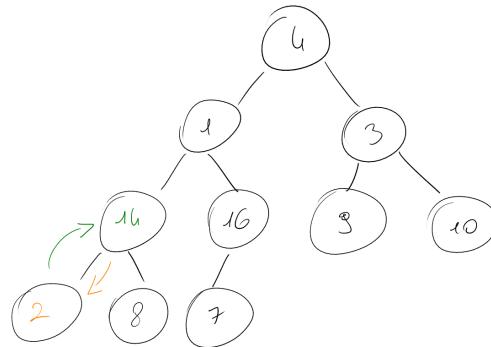
Spiegazione Vediamo il procedimento attraverso un esempio. Supponiamo di voler costruire un max_heap a partire dal seguente array:

6	1	3	2	16	9	10	16	8	7
---	---	---	---	----	---	----	----	---	---

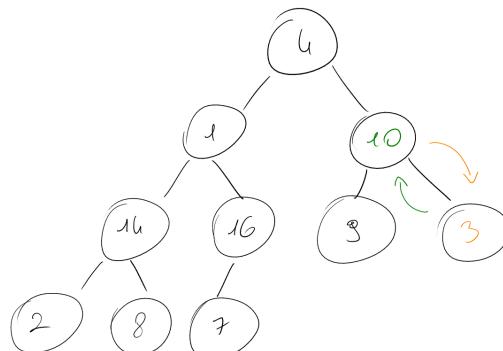
Inizialmente, prima dell'applicazione della `max_heapify`, l'albero si può visualizzare graficamente in questo modo:



- i = 5) L'albero ha 10 elementi, per cui alla prima iterazione i assumerà il valore $\lfloor \frac{10}{2} \rfloor = 5$. In questa posizione troviamo il nodo col valore 16, per cui verrà chiamata la `max_heapify` su questo nodo. Il sottoalbero avente la radice nel nodo 16 soddisfa la proprietà del max_heap, dunque in questa iterazione l'algoritmo non compie nulla e procede all'iterazione successiva.
- i = 4) Il nodo corrente è quello con valore 2: confrontandolo con i suoi figli, notiamo che il suo figlio sinistro ha valore 14, che è maggiore di 2. Perciò effettuiamo uno scambio tra i valori di questi due nodi. Le chiamate ricorsive di `max_heapify` non eseguiranno alcuna istruzione poiché i figli del nodo corrente sono foglie e quindi, vacuamente, la proprietà del max_heap è verificata.

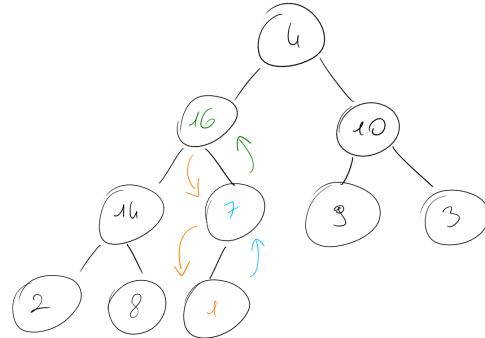


- i = 3) Il nodo corrente è quello con valore 3: notiamo che il suo figlio destro contiene un valore maggiore di 3, perciò effettuiamo lo scambio tra i valori di questi nodi e poiché questi sono foglie non abbiamo bisogno di svolgere altre operazioni in questa iterazione.



- i = 2) Il nodo corrente è quello con valore 1: prima viene scambiato con il 16, e una volta spostato in questa posizione avviene la chiamata ricorsiva di `max_heapify` sulla nuova posizione del nodo per verificare la conservazione della proprietà. Il nodo ora ha un solo figlio con valore 7, maggiore di

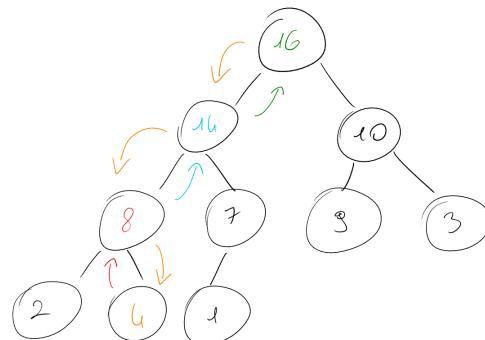
1, dunque avviene lo scambio tra questi nodi e si conclude la corrente iterazione.



$i = 1$) L'ultima iterazione del ciclo nel nostro esempio è la più impegnativa. Controlliamo che il nodo radice, contenente il valore 4, sia il maggiore tra sé stesso e i suoi figli. Questo non si verifica, perché il suo figlio sinistro ha valore 16. Questi due nodi vengono quindi scambiati di posto.

Con la nuova disposizione la proprietà non è ancora verificata, poiché ora il nodo 4 ha come figli 7 e 14. Vengono quindi scambiati il 4 e il 14.

In questa nuova posizione, il nodo 4 ha due figli: 2 e 8. Vengono infine scambiati i nodi 4 e 8, e il risultato di quest'operazione è un heap che soddisfa la proprietà del max_heap.



Analisi della correttezza

$INV \equiv$ ogni nodo $i + 1, i + 2, \dots, n$ è radice di un max_heap con $n = A.length$.

L'invariante assicura che la pre-condizione di `max_heapify` sia verificata perché i figli di i si trovano in posizioni $2 * i$ e $2 * i + 1$, entrambe maggiori di i .

Alla conclusione del ciclo, vale che $i = 0$:

$INV \left[\frac{0}{i} \right] \equiv$ ogni nodo 1, 2, ..., n è radice di un max_heap. In particolare, il nodo 1 contiene la radice del nostro albero che è dunque un max_heap.

Analisi della complessità La procedura compie $O(n)$ iterazioni ($\frac{n}{2}$ per essere precisi) in ognuna delle quali viene chiamata la funzione `max_heapify` che ha come costo temporale $O(\log n)$. Quindi il tempo di esecuzione di `build_max_heap` è sicuramente $O(n \log n)$, tuttavia questo non risulta essere un limite stretto. Cerchiamone uno più stretto.

Sappiamo che il tempo di esecuzione di `max_heapify` è $O(h)$, dove h è l'altezza del nodo su cui viene chiamata. Si noti che l'altezza della maggior parte dei nodi è piccola, perciò il tempo d'esecuzione sarà limitato superiormente dalla seguente funzione:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

Sfruttiamo una serie nota:

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2} \text{ per } |x| < 1$$

Poniamo $x = \frac{1}{2} < 1$. Allora,

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1}{2} \frac{1}{\left(\frac{1}{2}\right)^2} = \frac{4}{2} = 2$$

e quindi

$$O \left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

Dunque il tempo d'esecuzione di `build_max_heap` è $O(n)$.

Heap Sort

```

1 heapsort(array A)
2     build_max_heap(A)
3     for i = A.length down to 2
4         scambio A[i] con A[1]
5         A.heap_size = A.heap_size - 1
6         max_heapify(A, 1)

```

Osservazioni

- Poiché il primo elemento viene scambiato ad ogni iterazione, possiamo evitare di far compiere al ciclo l'iterazione relativa alla radice.
- Dopo lo scambio, la violazione della proprietà può esserci solamente nella radice: perciò chiamiamo su di essa `max_heapify` e verifichiamolo.

Spiegazione L'algoritmo trasforma l'array passatogli in un `max_heap`, in modo che il valore massimo contenuto al suo interno assuma la prima posizione. Successivamente vengono iterati quasi tutti i nodi, a partire dall'ultimo elemento fino ad arrivare ai figli della radice. In ciascun'iterazione del ciclo, il nodo radice viene scambiato di posto con l'elemento in posizione i -esima, e la dimensione dell'heap viene diminuita di uno in modo che la successiva operazione di `max_heapify` vada a riordinare l'array ponendo in prima posizione il nodo più grande dopo quello che abbiamo spostato, che d'ora in poi non considereremo più (in quanto, essendo il più grande dell'array, possiamo tranquillamente porre in ultima posizione).

In questo modo, l'elemento più grande dell'array per ogni iterazione del ciclo viene spostato in fondo fino ad arrivare alla radice dell'heap, e la correttezza dell'algoritmo è garantita dal seguente invarianto.

Analisi della correttezza

$INV \equiv$ il sottoarray $A[1 \dots i]$ è un `max_heap` che contiene gli i elementi più piccoli di $A[1 \dots n]$ e il sottoarray $A[i + 1 \dots n]$ contiene gli $(n - 1)$ elementi più grandi di $A[1 \dots n]$, ordinati.

max_heap	ORDINATI
	i

Alla conclusione del ciclo, $i = 1$:

$INV \left[\frac{1}{i} \right] \equiv$ il sottoarray $A[1 \dots 1]$ è un `max_heap` che contiene l'elemento più piccolo di $A[1 \dots n]$ e il sottoarray $A[2 \dots n]$ contiene gli $(n - 1)$ elementi più grandi di $A[1 \dots n]$, ordinati.

Analisi della complessità La prima operazione dell'algoritmo, la costruzione del `max_heap` a partire dall'array, ha costo $O(n)$. Successivamente, viene eseguita la `max_heapify` avente costo $O(\log n)$ per un totale di $n - 1$ volte. Il tempo d'esecuzione totale dell'algoritmo è

$$T(n) = O(n) + O(n \log n) = O(n \log n)$$

Teorema L'algoritmo *Heap Sort* ordina *in loco* (come il *Quick Sort*) n elementi eseguendo, nel caso peggiore, $O(n \log n)$ confronti (come il *Merge Sort*). Tuttavia, nel caso medio e attraverso le varie ottimizzazioni, il *Quick Sort* è tendenzialmente più veloce (ma non nel caso peggiore!).

15 Code di priorità

Una **coda di priorità** è una struttura dati che serve a memorizzare un insieme dinamico di elementi, ciascuno dei quali ha un valore detto **chiave** o **peso**.

Esistono due tipologie di code di priorità:

- **Code di massima priorità**

Sono usate nelle implementazioni dei sistemi di *scheduling* dei processi all'interno dei sistemi operativi.

- **Code di minima priorità**

Sono usate nella teoria dei grafi e nell'algoritmo di Prim.

15.1 Operazioni su code di massima priorità

1. **Inserimento**

```
1 insert(S, x)
```

Post-condizione: inserisce l'elemento **x** nell'insieme **S**. $S = S \cup \{x\}$

2. **Massimo**

```
1 maximum(S)
```

Post-condizione: restituisce l'elemento di **S** con chiave più grande (ma non lo rimuove da **S**, è una lettura).

3. **Estrazione del massimo**

```
1 extract_max(S)
```

Post-condizione: restituisce ed elimina da **S** l'elemento di **S** con chiave più grande.

4. **Increase**

```
1 increase_key(S, x, k)
```

Pre-condizione: $x \in S$

Post-condizione: aumenta il valore della chiave di **x** al nuovo valore **k**, che si suppone sia almeno pari al valore corrente della chiave di **x** ($k \geq$ chiave di **x**).

15.2 Operazioni su code di minima priorità

1. **Inserimento**

```
1 insert(S, x)
```

Post-condizione: inserisce l'elemento **x** nell'insieme **S**. ($S = S \cup \{x\}$)

2. Minimo

```
1 minimum(S)
```

Post-condizione: restituisce l'elemento di S con chiave più piccola (ma non lo rimuove da S , è una lettura).

3. Estrazione del minimo

```
1 extract_min(S)
```

Post-condizione: restituisce ed elimina da S l'elemento di S con chiave più piccola.

4. Decrease

```
1 decrease_key(S, x, k)
```

Pre-condizione: $x \in S$

Post-condizione: decrementa il valore della chiave di x al nuovo valore k , che si suppone sia al più pari al valore corrente della chiave di x ($k \leq$ chiave di x).

15.3 Code di massima priorità implementate con max_heap

1. Massimo

```
1 heap_maximum(Heap A)
2     if A.heap_size < 1
3         error "heap underflow"
4     return A[1]
```

L'algoritmo ritorna semplicemente il primo elemento dell'heap, che per sappiamo essere l'elemento più grande dell'heap per definizione di max_heap, dopo aver fatto un controllo che l'heap non sia vuoto.

Tempo d'esecuzione $O(1)$

2. Estrazione del massimo

```
1 heap_extract(Heap A)
2     if A.heap_size < 1
3         error "heap underflow"
4     max = A[1]
5     A[1] = A[A.heap_size]
6     A.heap_size = A.heap_size - 1
7     max_heapify(A, 1)
8     return max
```

L'algoritmo sostituisce il primo elemento dell'array, che sappiamo essere il massimo, con l'ultimo; riduce la dimensione dell'heap di un elemento e fa una chiamata a `max_heapify` sulla radice dell'heap per ripristinare la proprietà del max_heap. Infine ritorna il massimo, che abbiamo salvato.

Tempo d'esecuzione $O(\log n)$ per la chiamata a `max_heapify`.

3. Increase

```

1 heap_increase_key(Heap A, Node i, Elem Key)    //i e' un indice
2     if Key < A[i]
3         error "nuova chiave piu' piccola di quella corrente"
4     A[i] = Key
5     while i > 1 and A[parent(i)] < A[i]
6         scambia A[i] e A[parent(i)]
7         i = parent(i)

```

L'algoritmo controlla che il valore del nodo i sia minore della chiave Key , e in caso affermativo aggiorna il nodo col nuovo valore e effettua un cammino dal nodo i fino, potenzialmente, alla radice dell'albero per verificare che la proprietà del max_heap sia preservata di figlio in padre.

Analisi della correttezza

$INV \equiv$ L'array $A[1 \dots A.heap_size]$ soddisfa la proprietà del max_heap, tranne una possibile violazione in posizione $A[i]$.

Conclusione: $INV \wedge \neg \text{Guardia} \Rightarrow \text{Asserzione finale}$

$$\neg \text{Guardia} \equiv i == 1 \vee A[\text{parent}(i)] \geq A[i]$$

(a) $i == 1$

Dall'invariante, $INV \left[\frac{1}{i} \right]$, e poiché i è la radice non c'è alcuna violazione.

(b) $A[\text{parent}(i)] \geq A[i]$

Dall'invariante, l'unica violazione potrebbe essere in i , ma questo non è possibile perché la guardia ci permette di concludere che la proprietà del max_heap è verificata.

Tempo d'esecuzione $O(\log n)$, perché stiamo effettuando un cammino da un nodo fino alla radice, che nel caso più lungo (il nodo è una foglia) percorre tutta la sua altezza, cioè $h = \log n$.

4. Inserimento

```
1 max_heap_insert(Heap A, Elem Key)
2     A.heap_size = A.heap_size + 1
3     A[A.heap_size] = -infinito
4     heap_increase_key(A, A.heap_size, Key)
```

L'algoritmo aumenta la dimensione dell'heap di un elemento, pone in ultima posizione una sentinella (abbiamo scelto $-\infty$) sulla quale viene chiamata la `heap_increase_key` col valore del nuovo elemento che vogliamo inserire nell'heap. In questo modo, ci penserà questa procedura a ripristinare lo status di `max_heap` qualora si presentassero delle violazioni.

Non potendo fisicamente rappresentare $-\infty$ in codice reale e utilizzabile, in un'implementazione reale al posto di $-\infty$ potremmo inserire il valore `Key` stesso.

Tempo d'esecuzione $O(\log n)$, per la chiamata a `heap_increase_key`.

In conclusione Un heap può svolgere ciascuna operazione delle code di priorità nel tempo $O(\log n)$, dove n è il numero di elementi dell'insieme dinamico.

16 Esercizi d'esame su ordinamento e varie

16.1 Eliminazione nodo max_heap

Implementare l'algoritmo di eliminazione di un nodo da un max_heap.

```

1 heap_delete(Heap A, Node i)

Pre-condizione:  $i \in A$ 
Post-condizione:  $i \notin A$ 

1 heap_delete(Heap A, Node i)
2     if A.heap_size == 1
3         A.heap_size = A.heap_size - 1
4     else
5         val = A[i]
6         A[i] = A[A.heap_size]
7         A.heap_size = A.heap_size - 1
8         if val > A[i]
9             max_heapify(A, i)
10        else
11            while i > 1 and A[i] > A[parent(i)]
12                scambio A[i] e A[parent(i)]
13                i = parent(i)

```

Spiegazione L'algoritmo controlla la dimensione dell'heap: se costituito da un solo elemento, sarà sufficiente porre a 0 il numero dei suoi elementi.

Altrimenti, salva il valore del nodo da rimuovere, e scambia tale nodo con l'ultimo presente nell'heap. Successivamente riduce la dimensione dell'heap e verifica se il nuovo elemento è maggiore o minore rispetto al valore che aveva precedentemente: se minore, è possibile che si verifichi una violazione della proprietà del max_heap verso il basso, verso le foglie dell'albero, per cui viene chiamata la `max_heapify`; se maggiore, è possibile che si verifichi una violazione questa volta non verso il basso, bensì verso l'alto: viene eseguito lo stesso blocco di codice presente nella `heap_increase_key` per verificare la presenza di violazioni a partire dal nodo corrente salendo fino alla radice.

Tempo d'esecuzione $O(\log n)$. Entrambi i rami dell'`if-statement` hanno tale costo, infatti la `max_heapify` esegue un cammino lungo al più l'altezza dell'albero partendo dalla radice fino alle foglie, mentre l'altro blocco esegue un cammino lungo al più l'altezza dell'albero partendo dalle foglie fino alla radice.

16.2 Trovare il minimo

Sia dato un insieme S di n numeri interi e distinti. Gli elementi di S sono memorizzati in parte in un `max_heap` A e in parte in un `min_heap` B , entrambi non vuoti.

- (a) Se tutte le chiavi di A sono minori di quelle di B , quanto costa determinare il minimo di S ? Quanto costa eliminarlo?
- (b) Se tutte le chiavi di A sono maggiori di quelle di B , quanto costa determinare il minimo di S ? Quanto costa eliminarlo?

Soluzione

Supponiamo che A abbia n_1 elementi, B ne abbia n_2 in modo che $n_1 + n_2 = n$.

- (a) A è un `max_heap` tale che $\forall x \in A, x < y \forall y \in B$, che è un `min_heap`.
Il minimo in un `max_heap` si trova nelle foglie, che occupano le posizioni $\lfloor \frac{n_1}{2} \rfloor + 1, \dots, n_1$. Devo dunque scorrere da $\lfloor \frac{n_1}{2} \rfloor + 1$ a n_1 per determinare l'elemento minimo. Il tempo d'esecuzione è $\Theta(n_1)$, e nel caso peggiore A contiene tutti gli elementi di S tranne uno (perché A e B non possono essere vuoti), cioè $n_1 = n - 1$, quindi $T(n) = O(n)$.
Per cancellare il minimo, devo prima posizionarmi su di esso, dunque effettuare la ricerca come nel caso precedente, che ha costo $O(n)$, e poi rimuovere il nodo tramite la `heap_delete`, che costa $O(\log n_1) = O(\log n)$. Dunque il costo totale è $O(n)$.
- (b) $\forall x \in A, x > y \forall y \in B$
Il minimo in un `min_heap` si trova nella radice, quindi la sua ricerca ha tempo d'esecuzione costante pari a $O(1)$.
Per eliminarlo, applico la `extract_min` su B , che costa $O(\log n_2) = O(\log n)$.

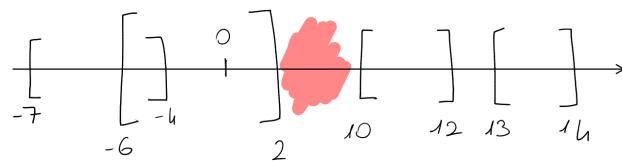
16.3 Unione di intervalli

Si scriva una funzione efficiente che, ricevuto in input un insieme di n intervalli chiusi $[a_i, b_i]$, con a_i, b_i numero interi e $a_i \leq b_i$ e $1 \leq i \leq n$, stabilire se la loro unione è un intervallo (nel qual caso, restituirlo).

```
1 bool unione(vector<pair<int,int>>& arr, pair<int,int>& res)
```

Pre-condizione: `arr.size() ≥ 1`

Ci conviene graficare il problema.



$$\langle [-7, 4], [10, 12], [-6, 2], [13, 14] \rangle$$

Dobbiamo verificare che, non siano presenti buchi tra gli intervalli presenti nell'array passati come parametro. In questo caso, dobbiamo ritornare `false` perché, come vediamo, sono presenti dei buchi evidenziati in rosso.

Se la condizione è verificata, dobbiamo tornare gli estremi dell'intervallo unione di tutti gli intervalli:

$$[a, b] = \bigcup_{i=1}^n [a_i, b_i]$$

```
1 bool unione(vector<pair<int,int>>& arr, pair<int,int>& ris){
2     size_t i = 1;
3     int endtemp = arr[0].second;
4     ordina(arr);
5     while(i < arr.size() && endtemp + 1 >= arr[i].first){
6         if (arr[i].second > endtemp){
7             endtemp = arr[i].second;
8         }
9         i++;
10    }
11    if (i == arr.size()){
12        ris = {arr[0].first, endtemp};
13        return true;
14    }
15    return false;
16 }
```

Spiegazione Per la risoluzione di questo problema, torna molto utile (se non indispensabile) ordinare l'array; decidiamo di ordinarlo sulla base del primo

estremo dell'intervallo; per esempio, se l'array inizialmente è quello della figura di cui sopra, dopo l'ordinamento sarà permuto nel seguente modo:

$$\langle [-7, 4], [-6, 2], [10, 12], [13, 14] \rangle$$

Vogliamo ora scorrere l'array ordinato confrontando l'estremo destro di ciascun intervallo con l'estremo sinistro dell'intervallo successivo: finché è verificata la proprietà che fa da guardia del ciclo, sappiamo per certo che non ci sono buchi di interi tra gli intervalli passati, in quanto quello che vogliamo verificare è proprio che l'estremo destro dell'intervallo precedente "copra" (o almeno sia pari a) l'estremo sinistro dell'intervallo corrente (o di quelli correnti nel caso un intervallo abbia estremo destro distanziato abbastanza da coprire anche altri intervalli); quando troviamo un estremo destro superiore a quello che stiamo considerando, lo aggiorniamo.

Una volta interrotto il ciclo, se è arrivato alla fine, siamo sicuri che non ci sono buchi e l'unione degli intervalli si può esprimere come un unico intervallo, avente come estremo sinistro l'estremo sinistro del primo intervallo, elemento dell'array ordinato, e come estremo destro l'ultimo estremo destro che abbiamo usato per aggiornare `endtemp`. Aggiorniamo di conseguenza le variabili dove salviamo questi valori e torniamo il booleano corretto.

Tempo d'esecuzione Sappiamo che la procedura di ordinamento dell'array non può scendere sotto l'ordine di $\Theta(n \log n)$, mentre il ciclo che scorre l'array ordinato ha tempo d'esecuzione $O(n)$. Quindi la complessità temporale dell'algoritmo sarà $T(n) = \Theta(n \log n)$.

Osservazioni sull'ordinamento In esame sarà richiesto di scrivere esplicitamente il corpo di ogni procedura utilizzata, compresa `ordina`, che dovrà implementare un algoritmo di ordinamento a nostra scelta. Anche il calcolo della complessità di questa procedura dovrà essere espressa di conseguenza, tenendo conto dei tempi di esecuzione nel caso medio e nel caso peggiore.

La Professoressa invita caldamente ad evitare l'*Insertion Sort*, in quanto nel caso peggiore ha complessità $\Theta(n^2)$, e suggerisce il *Quick Sort* randomizzato con tripartizione, soluzione che difficilmente porta al caso peggiore (avente anch'esso complessità $\Theta(n^2)$), portando il problema a risolversi nel caso medio di complessità $O(n \log n)$. È anche apprezzato, usando il *Quick Sort*, riportare due complessità, una per ciascun caso in cui l'algoritmo può finire.

16.4 Somma k

Dato un intero k e un array di n interi, scrivere un algoritmo per trovare una coppia di posizioni distinte i e j , tali che $0 \leq i < j \leq (n-1)$ e $v[i] + v[j] = k$. La funzione deve restituire `true` e i valori $v[i]$ e $v[j]$ se i e j esistono, `false` altrimenti.

Possiamo implementare diverse soluzioni.

Soluzione 1 La soluzione più semplice consiste nell'utilizzare la forza bruta e scorrere l'array in due cicli innestati per provare tutte le possibili combinazioni di valori che danno come somma il risultato cercato. Chiaramente si tratta di una soluzione inefficiente, in quanto il tempo d'esecuzione è $O(n^2)$.

Soluzione 2 Una soluzione è quella di ordinare l'array ricevuto in input e applicare un algoritmo di ricerca binaria nel seguente modo.

```

1 bool sommaK(Array A, int k)
2     for i = 1 to A.length - 1
3         temp = v[i]
4         x = k - temp
5         binary search in A[i + 1, ..., n] di x

```

La ricerca binaria ha costo $O(\log n)$, ed essendo ripetuta per $n-1$ volte, la complessità dell'algoritmo risulta pari a quella dell'ordinamento, quindi in questa soluzione $T(n) = O(n \log n)$.

Soluzione 3

```

1 bool sommaK(vector<int>& v, int k, int& val1, int& val2){
2     size_t p = 0, f = v.size() - 1;
3     bool trovati = false;
4
5     ordina(v);
6
7     while (p < f && !trovati){
8         if (v[p] + v[f] == k){
9             trovati = true;
10            val1 = p;
11            val2 = f;
12        }
13        else{
14            if (v[p] + v[f] < k){
15                p++;
16            }
17            else{
18                f--;
19            }
20        }
21    }
22 }

```

Spiegazione Ordinando l'array, possiamo ridurre fortemente lo spazio di ricerca, sviluppando un algoritmo che trovi i valori cercati in tempo lineare rispetto al numero di elementi dell'array. Possiamo infatti prendere gli estremi dell'array e, sulla base della loro somma, capire quali combinazioni di elementi possiamo scartare.

L'algoritmo di per sé consiste in un ciclo che compie al più n iterazioni (dove n è il numero di elementi dell'array), in ciascuna delle quali o viene incrementato l'indice sinistro (se la somma è inferiore al valore cercato) o viene decrementato quello destro (se la somma è superiore al valore cercato). Il ciclo termina se i due indici raggiungono la stessa posizione (in tal caso non abbiamo trovato delle soluzioni) o quando, appunto, la somma degli elementi negli indici correnti dà come risultato k . In tal caso, vengono aggiornate le variabili di ritorno e il booleano che ci fa da guardia del ciclo.

Tempo d'esecuzione Tenendo gli stessi accorgimenti che abbiamo fatto per l'esercizio precedente, ammesso che l'algoritmo di ordinamento abbia tempo d'esecuzione $\Theta(n \log n)$, poiché il ciclo compie $O(n)$ iterazioni, allora il tempo d'esecuzione della funzione `sommaK` sarà $T(n) = \Theta(n \log n)$.

16.5 Somma 21

Dato un array v di n interi non negativi, scrivere un algoritmo per trovare una coppia di elementi in v tali che la loro somma sia 21.

Per risolvere questo problema, la soluzione ottimale prevede l'uso di un **vettore delle occorrenze**, cioè un vettore che conta il numero di occorrenze di un elemento dentro un vettore.

```

1 bool somma21(vector<int>& v, int& val1, int& val2){
2     vector<int> occ(22);
3     bool trovati = false;
4     size_t i = 0;
5     while (i < v.size() && !trovati){
6         if (v[i] <= 21){
7             occ[v[i]]++;
8             if (occ[21 - v[i]] > 0){
9                 trovati = true;
10                val1 = v[i];
11                val2 = 21 - v[i];
12            }
13        }
14        i++;
15    }
16    return trovati;
17 }
```

Spiegazione Notiamo come in questa soluzione non è necessario ordinare l'array, e possiamo sfruttare a nostro vantaggio la condizione che gli elementi dell'array sono tutti non negativi (quindi maggiori o uguali a 0).

L'algoritmo inizializza un vettore di 22 elementi, contenente un contatore per ciascun numero da 0 a 21 compresi; successivamente itera tutti gli elementi dell'array, andando ad incrementare il relativo contatore nel vettore delle occorrenze ogni qualvolta trova un numero minore o uguale a 21.

Poiché stiamo utilizzando il valore del numero come indice nel vettore delle occorrenze, sappiamo per certo che se due posizioni x e y in questo vettore tali che $x + y = 21$ hanno i contatori maggiori di 0, abbiamo trovato due numeri la cui somma fa 21. Per questo motivo, ogni volta che troviamo un numero $x < 21$, possiamo controllare la posizione $y = 21 - x$ del vettore e verificare che il contatore sia maggiore di 0; in tal caso, gli elementi che stiamo cercando sono proprio l'elemento puntato nella corrente iterazione dell'array e la sua differenza rispetto a 21.

Tempo d'esecuzione Il ciclo compie al più n iterazioni, quindi $T(n) = O(n)$.

Osservazioni Una piccola modifica che possiamo fare per rendere il codice più raffinato è quella di utilizzare, come vettore delle occorrenze, non un vettore di interi, bensì un vettore di variabili booleane, in quanto non ci interessa sapere esattamente il numero di volte che tale elemento è stato trovato, ma solamente

il fatto che sia stato trovato oppure no.

Ricordiamo che il vettore delle occorrenze può essere utilizzato solo se abbiamo un insieme di elementi limitato, $[-m, K] \rightarrow [0, K - (-m)]$.

Una qualsiasi posizione nel vettore delle occorrenze, $\text{occ}[i]$, contiene la seguente informazione:

$$\text{occ}[i] = |\{j \in [0, \dots, n-1] : v[j] = i\}|$$

17 Ordinamenti lineari

17.1 Complessità algoritmi di ordinamento

Riassumiamo le complessità degli algoritmi di ordinamento studiati finora:

1. *Insertion Sort*: $O(n^2)$
2. *Merge Sort*: $O(n \log n)$
3. *Quick Sort*: $O(n^2)$ nel caso peggiore, $O(n \log n)$ nel caso medio
4. *Heap Sort*: $O(n \log n)$

Sorge spontanea la seguente domanda: è possibile trovare algoritmi di ordinamento più efficienti?

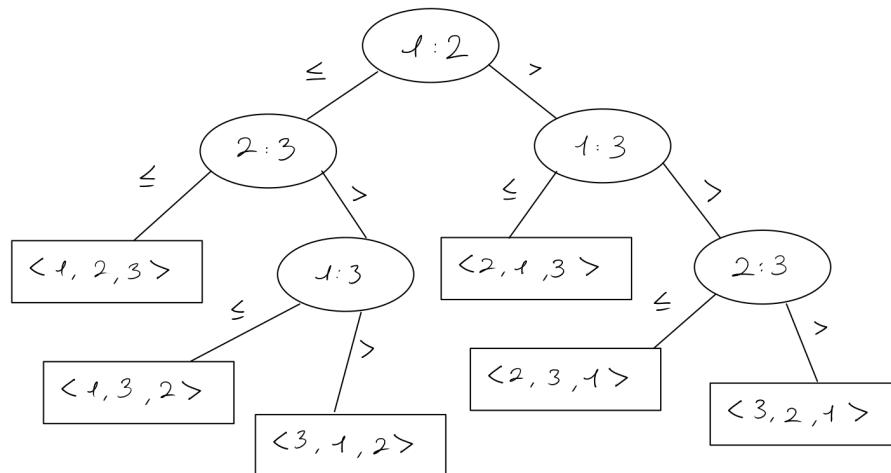
La risposta non è univoca, in quanto dipende dal modello computazionale che stiamo considerando. Cerchiamo di trovare una risposta per il modello computazionale degli algoritmi sopracitati, cioè il **confronto**. Cominciamo col considerare due limiti dai quali trarremo delle importanti conclusioni:

- $\Omega(n)$, cioè il numero di operazioni necessarie per esaminare gli elementi in input; scopriremo che questo non può rappresentare un limite inferiore per gli algoritmi basati sul confronto.
- $O(n \log n)$, cioè il limite che abbiamo già trovato in diversi algoritmi.

Da questi due limiti possiamo supporre, e più avanti verificheremo, che $\Omega(n \log n)$ è il limite inferiore degli algoritmi basati sul confronto.

Possiamo dimostrarlo attraverso gli **alberi di decisione**, una struttura basata sugli alberi binari che consiste in un'astrazione che ci permette di mettere in evidenza i passaggi di un algoritmo di ordinamento.

Prendiamo per esempio il caso dell'ordinamento di un array di tre elementi, $\langle a_1, a_2, a_3 \rangle$:

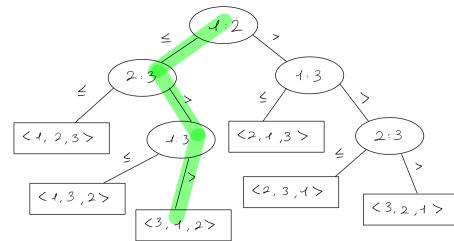


In generale, per un input di dimensione $\langle a_1, \dots, a_n \rangle$:

- ogni nodo interno è etichettato da $i:j$, $i,j \in \{1, \dots, n\}$
 - significa verificare se $a_i \leq a_j$
 - il sottoalbero sinistro dà i successivi confronti se $a_i \leq a_j$
 - il sottoalbero destro dà i successivi confronti se $a_i > a_j$
- ogni nodo foglia dà una permutazione

$$\langle \pi(1), \dots, \pi(n) \rangle \text{ tale che } a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$$

Prendiamo l'esempio dell'array $\langle 6, 8, 5 \rangle$: vengono confrontati il 6 e l'8 e si scende nel sottoalbero sinistro della radice; successivamente avvengono i confronti tra l'8 e il 5 (si va nel sottoalbero destro) e tra il 6 e il 5 (si va nel sottoalbero destro). La traccia per questo array è riportata di fianco.



Dato un qualsiasi ordinamento per confronti:

- È possibile costruire un albero di decisione per ogni n .
- L'albero modella tutte le possibili **tracce d'esecuzione**, cioè ciascun cammino che parte dalla radice ed arriva ad una foglia. La traccia ci dice quali sono i confronti che il nostro algoritmo compie per arrivare ad una soluzione.
- Il tempo d'esecuzione, cioè il numero di confronti che vengono effettuati, è la lunghezza della relativa traccia.
- Il tempo d'esecuzione nel caso peggiore, cioè la complessità asintotica, è quindi l'altezza dell'albero.

Vogliamo quindi cercare un limite inferiore sulle altezze di tutti gli alberi di decisione dove ogni permutazione compare come foglia (non potrebbe essere altrimenti: se le permutazioni non comparissero come nodi foglia, l'algoritmo non sarebbe corretto).

Per definizione di permutazione, sappiamo che dati n elementi esistono $n!$ permutazioni degli elementi: di conseguenza, il **numero di foglie** è $\geq n!$ perché per essere corretto ogni permutazione deve comparire almeno una volta.

Lemma Un qualsiasi albero binario di altezza h ha al più 2^h foglie.

Dimostrazione Avviene per induzione su h .

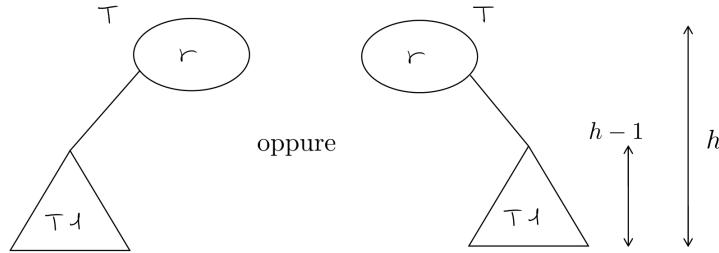
Caso base Un albero di altezza uguale a 0 ($h = 0$) rappresenta un albero che ha un solo nodo, la radice, che è l'unica foglia. Dunque,

$$f = 1 \leq 2^0 = 1, \text{ c.v.d.}$$

Passo induttivo Assumiamo che per alberi binari di altezza $k < h$ il numero di foglie è $\leq 2^k$ e lo dimostro per h .

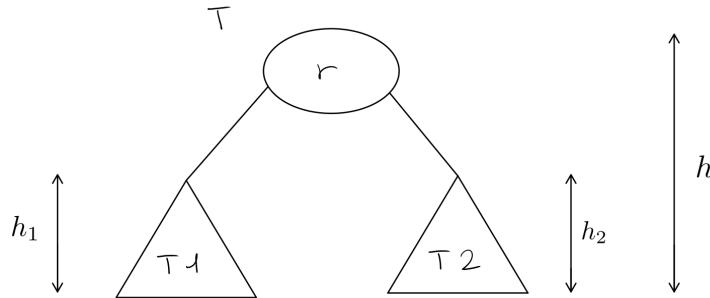
Sia r la radice dell'albero T di altezza h : se r ha un solo figlio, allora il numero di foglie di T è uguale al numero di foglie dell'albero T_1 (il suo unico sottoalbero). Quindi,

$$f = f_{T_1} \leq 2^{h-1} \leq 2^h, \text{ c.v.d.}$$



Se invece r ha due figli, allora il numero di foglie di T è uguale alla somma delle foglie di T_1 e T_2 . Poiché T_1 ha altezza $h_1 < h$ e T_2 ha altezza $h_2 < h$:

$$f = f_{T_1} + f_{T_2} \stackrel{\text{Per ip. ind.}}{\leq} 2^{h_1} + 2^{h_2} \leq 2 \cdot 2^{\max\{h_1, h_2\}} \stackrel{h=1+\max\{h_1, h_2\}}{=} 2^{1+\max\{h_1, h_2\}} = 2^h$$



Quindi, $f \leq 2^h$, c.v.d..

Teorema Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \log n)$ confronti nel caso peggiore.

Dimostrazione Bisogna determinare l'altezza di un albero di decisione dove ogni permutazione appare come foglia. Si consideri un albero di decisione con tale caratteristica di altezza h , con l foglie che corrisponde ad un ordinamento per confronti di n elementi. Allora,

$$n! \leq l \leq 2^h \Rightarrow h \geq \log n!$$

Utilizziamo l'approssimazione di Stirling:

$$n! \cong \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Perciò, per n sufficientemente grande, considero il termine più grande:

$$h \geq \log n! \geq \log \left(\frac{n}{e}\right)^n = n \log \left(\frac{n}{e}\right) = n (\log n - \log e) = n(\log n - \Theta(1)) = \Omega(n \log n)$$

Corollario *Heap Sort* e *Merge Sort* sono algoritmi di ordinamento per confronti asintoticamente ottimali. La dimostrazione si deriva dal fatto che il loro limite superiore del tempo di esecuzione, $O(n \log n)$, corrisponde al limite inferiore $\Omega(n \log n)$ nel caso peggiore.

17.2 Counting Sort

Gli algoritmi di ordinamento lineari non si basano sul confronto, bensì sulla validità di alcune assunzioni sulla natura dell'input. Il **Counting Sort** ne è un valido esempio:

Assunzione I numeri da ordinare sono interi in un intervallo che va da 0 a k , per qualche intero k .

- **Input:** $A[1, \dots, n]$ dove $A[j] \in [0, \dots, k]$; n e k sono parametri dell'algoritmo.
- **Output:** $B[1, \dots, n]$, una permutazione ordinata di A .
- **Memoria ausiliaria:** $C[0, \dots, k]$.

```

1 countingSort(array A, array B, int n, int k)
2     creo C[0, ..., k]
3     for i = 0 to k           //init vettore occorrenze
4         C[i] = 0
5     for j = 1 to n
6         C[A[j]]++
7     for i = 1 to k
8         C[i] = C[i] + C[i - 1]
9     for j = n down to 1
10        B[C[A[j]]] = A[j]
11        C[A[j]]--

```

Spiegazione L'algoritmo è composto da 4 cicli **for** che operano su tre array:

- **A:** l'array da ordinare;
- **B:** l'array risultato, contenente gli elementi di **A** ordinati in modo crescente;
- **C:** un vettore ausiliario delle occorrenze, contenente $k + 1$ elementi (da 0 a k).

Il primo ciclo serve per inizializzare a 0 tutti gli elementi del vettore delle occorrenze.

Il secondo ciclo incrementa gli elementi del vettore delle occorrenze: alla fine di questo ciclo, l'elemento in posizione i -esima del vettore **C** contiene il numero di volte che il numero i compare in **A**; formalmente, scriviamo:

$$\begin{aligned} A[j] &\in [0, \dots, k] \\ C[i] &= |\{x \in \{1, \dots, n\} | A[x] = i\}| \end{aligned}$$

Il terzo ciclo compie il calcolo delle **somme prefisse** sugli elementi del vettore delle occorrenze: somma ad ogni elemento del vettore delle occorrenze il numero di occorrenze dell'elemento precedente. Questo passaggio è importante perché permette di capire quanti elementi minori o uguali di i sono presenti in **A**, dove

i è l'indice che scorre il vettore delle occorrenze, e quindi assume tutti i valori contenuti in \mathbf{A} ; formalmente, scriveremo:

$$\mathbf{C}[i] = |\{x \in \{1, \dots, n\} \mid \mathbf{A}[x] \leq i\}|$$

L'ultimo ciclo è quello che va a riempire l'array di output. Consideriamo di avere, a questo punto dell'algoritmo, un vettore delle occorrenze contenente, in ogni posizione, la somma cumulativa delle occorrenze dei valori minori o uguali dell'indice della posizione corrente. Decidiamo di scorrere il vettore in input dalla fine all'inizio: sia x il numero i -esimo presente in \mathbf{A} , che vogliamo inserire ordinato in \mathbf{B} , e y il numero di occorrenze in \mathbf{A} di valori minori o uguali ad x . L'operazione di addizione cumulativa sul vettore di occorrenze ci garantisce che possiamo inserire x nella posizione y dell'array di output ed essere sicuri che quella sarà la posizione corretta per x .

Per assicurarci di inserire nella posizione corretta eventuali occorrenze ulteriori di x , decrementiamo y di 1 e procediamo in questo modo per tutti gli elementi rimasti in \mathbf{A} . Alla fine di questo quarto ciclo, \mathbf{B} conterrà tutti gli elementi ordinati di \mathbf{A} .

Analisi della complessità

- Il primo ciclo compie k iterazioni: ha costo $\Theta(k)$.
- Il secondo ciclo compie n iterazioni: ha costo $\Theta(n)$.
- Il terzo ciclo compie k iterazioni: ha costo $\Theta(k)$.
- Il quarto ciclo compie n iterazioni: ha costo $\Theta(n)$.

Dunque il tempo d'esecuzione dell'algoritmo è $\Theta(n + k)$. Nella pratica, questo algoritmo è usato quando $k = O(n)$, condizione che rende la complessità dell'algoritmo $\Theta(n)$.

Osservazioni

- È importante calcolare le somme prefisse e iniziare la lettura dalla fine dell'array in input per produrre quello di output per garantire **stabilità** e **correttezza**: non si può costruire l'output generando i dati a partire dal vettore delle occorrenze senza calcolare le somme prefisse, perché i numeri da ordinare potrebbero contenere dati satellite che non possiamo ignorare, vogliamo conservarne l'ordine.
- **Attenzione!** La seguente implementazione dell'algoritmo di *Counting Sort* è errata: supponiamo di prendere $\min(\mathbf{A})$ e $\max(\mathbf{A})$ ed applicare il *Counting Sort* usando i due risultati come parametri del vettore delle occorrenze. In questo modo, qualora il massimo di \mathbf{A} sia molto alto, supponiamo n^2 , n^3 et cetera, la complessità dell'algoritmo non sarebbe lineare rispetto a n , bensì quadratica o peggio.

- L'intervallo può contenere valori negativi. È possibile implementare questa casistica traslando il valore iniziale del vettore delle occorrenze del valore assoluto del minimo negativo:

$$\begin{aligned} [-n, M] &\rightarrow [0, M - (-n)] \\ &\rightarrow [0, M + n] \end{aligned}$$

Di conseguenza,

$$x \in [-n, M] \rightarrow x - -(n) = x + n$$

Questa soluzione è implementabile a patto che valga la seguente condizione: l'algoritmo deve essere limitato superiormente da n ($M + n = O(n)$), altrimenti risulterebbe inefficiente.

17.3 Radix Sort

Come abbiamo visto, nel *Counting Sort* se $k \gg n$ l'algoritmo perde la peculiarità di avere complessità lineare rispetto al numero di elementi dell'array in input. Per questo motivo, una valida alternativa al *Counting Sort* è il ***Radix Sort***.

Questo algoritmo fu ideato dall'ingegnere americano Herman Hollerith per risolvere il problema dell'elaborazione dei dati del censimento degli Stati Uniti del 1890: i dati del censimento precedente erano stati elaborati in 7 anni, perciò serviva un modo più veloce per elaborarli entro il censimento successivo. Hollerith, vedendo il modo in cui venivano convalidati i biglietti dei treni, suppose che attraverso delle schede perforate fosse possibile memorizzare delle informazioni sulla base della presenza, dell'assenza o della posizione dei fori. Questa idea portò alla nascita delle prime macchine calcolatrici, e insieme ad esse della IBM, azienda che tuttogi è tra le maggiori aziende di produzione e commercializzazione hardware e software, di cui Hollerith fu uno dei primi fondatori.

L'idea di Hollerith diede grandi risultati: i dati del censimento del 1890 furono elaborati in appena due mesi, una tempistica notevole per i tempi.

L'algoritmo

Assunzione Tutti i numeri da ordinare sono composti da d cifre. La cifra in posizione 1 è la meno significativa, quella in posizione d la più significativa.

```

1 radixsort(array A, int d)
2   for i = 1 to d
3     usa un algoritmo di ordinamento stabile
4     per ordinare l'array A sulla cifra i

```

Solitamente, l'algoritmo stabile usato per ordinare i numeri sulla base della i -esima cifra è proprio il *Counting Sort*. Vediamo un esempio di come questo algoritmo opera su un array di numeri di tre cifre.

$A = [326,$	$A = [690,$	$A = [704,$	$A = [326,$
453,	751,	608,	435,
608,	453,	326,	453,
835, <i>i=1</i>	704, <i>i=2</i>	835, <i>i=3</i>	608,
751, <i>⇒</i>	835, <i>⇒</i>	751, <i>⇒</i>	690,
435,	435,	751,	704,
704,	326,	453,	751,
690]	608]	690]	835]

La stabilità dell'algoritmo ausiliario è fondamentale affinché l'ordinamento sia corretto: se non fosse stabile, per esempio, all'ultima iterazione i numeri 435 e 453 potrebbero essere scambiati di posto, risultato in un array non ordinato.

Analisi della correttezza Avviene per induzione sulla colonna da ordinare.

Caso base Se $i = 1$, ordino l'unica colonna e non devo fare altro, perché i numeri sono costituiti da una sola cifra.

Passo induttivo Assumiamo l'**ipotesi induttiva** che le cifre delle colonne $1, 2, \dots, i - 1$ siano ordinate e dimostriamo che un algoritmo stabile sulla colonna i lascia le colonne $1, \dots, i$ ordinate.

- Se due cifre in posizione i sono uguali, per la stabilità rimangono nello stesso ordine e per l'ipotesi induttiva sono ordinate.
- Se due cifre in posizione i sono diverse, allora l'algoritmo di ordinamento sulla colonna i le ordina e le mette in posizione corretta.

Dunque l'algoritmo è corretto, come volevasi dimostrare.

Analisi della complessità

Lemma Dati n numeri di d cifre, dove ogni cifra può avere fino a k valori possibili, la procedura `radixsort` correttamente ordina i numeri nel tempo $\Theta(d(n+k))$, se l'ordinamento stabile utilizzato dalla procedura impiega un tempo $\Theta(n+k)$.

Dimostrazione

- Abbiamo dimostrato per induzione la correttezza dell'algoritmo.
- Per ogni iterazione, il costo è dato da $\Theta(n+k)$.
- Il ciclo `for` compie d iterazioni.
- Di conseguenza, il costo totale è $\Theta(d(n+k))$.

Osservazioni

- Quando viene eseguito in tempo lineare questo algoritmo?
 - Se $k = O(n)$, il tempo d'esecuzione diventa $\Theta(dn)$.
 - Se, inoltre, d è costante, il tempo d'esecuzione diventa $\Theta(n)$.

Per questo motivo, in contesti specifici, il *Radix Sort* è molto usato nel mondo della programmazione.

L'algoritmo completo

È di seguito proposta un'implementazione completa dell'algoritmo *Radix Sort*, composta da una procedura principale, una subroutine ausiliaria *Counting Sort* e un'ulteriore procedura ausiliaria **cifra** che ritorna la d -esima cifra del numero che stiamo considerando nell'array da ordinare.

In questa implementazione, sia d il numero di cifre di cui ogni numero dell'array da ordinare è composto, e k il range di valori che ciascuna di queste cifre può assumere.

```

1 radixsort(array A, int d, int k)
2     for i = 1 to d
3         countingsort(A, k, i)
4
5 countingsort(array A, int k, int i)
6     alloco un vettore di appoggio B[1, ..., A.size]
7     alloco un vettore delle occorrenze C[0, ..., k]
8
9     for j = 0 to k
10        C[j] = 0
11
12    for j = 1 to A.size
13        C[cifra(A[j], k, i)]++
14
15 #ifdef ASCENDING_ORDER // somme prefisse
16    for j = 1 to k
17        C[j] = C[j] + C[j - 1]
18 #endif
19
20 #ifdef DESCENDING_ORDER // somme postfisse
21    for j = k - 1 down to 0
22        C[j] = C[j] + C[j + 1]
23 #endif
24
25    for j = A.size down to 1
26        B[C[cifra(A[j], k, i)]] = A[j]
27        C[cifra(A[j], k, i)]--
28
29    copio il vettore B in A
30
31 cifra(int x, int n, int i)
32     return (x / (n^(i-1))) % n           //  $\frac{x}{n^{i-1}} \% n$ 
```

Analisi della complessità La procedura **radixsort** è costituita da un ciclo che esegue d iterazioni, in ciascuna delle quali viene chiamata la procedura **countingsort**.

Quest'ultima è costituita da 4 cicli, due dei quali compionti k iterazioni e gli altri due $n = A.size$ iterazioni; dunque, per il **countingsort**, il tempo d'esecuzione è il seguente:

$$T(n, k) = \Theta(n + k)$$

Con l'assunzione che $k = O(n)$, dato che la procedura **cifra** viene eseguita in tempo costante, allora il tempo d'esecuzione di **countingsort** diventa lineare

rispetto alla dimensione dell'input: $T(n) = \Theta(n)$.

Di conseguenza, il `radixsort` avrà la seguente complessità:

$$T(n, d) = \Theta(d \cdot n)$$

Se, tuttavia, d è costante, allora anche il `radixsort` avrà tempo d'esecuzione lineare.

Il risultato finale, date le due assunzioni di cui sopra, è che la combinazione dei due algoritmi ordina un array in tempo lineare rispetto all'input.

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

Notare che l'algoritmo è stato adattato, attraverso le direttive presenti alle righe 15 e 20, per ordinare l'array in ordine crescente o decrescente: se il programmatore ha definito la macro `ASCENDING_ORDER`, l'algoritmo computa le somme prefisse, e l'array risultante verrà ordinato in modo crescente; altrimenti, se l'utente ha definito la macro `DESCENDING_ORDER`, l'algoritmo calcolerà le somme postfisse, portando come risultato ad un array ordinato in modo decrescente.

Caso di studio: ordinamento di numeri binari di 32 bit

Immaginiamo di avere n interi da ordinare, ciascuno rappresentato su b bit, supponiamo $b = 32$. Applicare 32 volte il *Radix Sort* su cifre che possono assumere due valori (0 e 1) risulta costoso.

È possibile trovare un compromesso: scegliamo $r = 8$ bit (1 byte) e otteniamo $d = \frac{32}{8} = 4$ cifre, ognuna delle quali varia da 0 a 255 (che equivale a $2^8 - 1$), quindi $k = 256$.

In altre parole, possiamo spezzare ogni intero in $\lceil \frac{b}{r} \rceil$ cifre, ognuna costituita da r bit, e il numero possibile di valori di ciascuna cifra è $k = 2^r$. Il tempo d'esecuzione del *Radix Sort* diventa quindi:

$$\Theta(d(n+k)) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Dati i valori di n e b , vogliamo scegliere un valore arbitrario di r , con $r \leq b$, in modo tale da minimizzare questa funzione affinché non sia dispendiosa. In particolare, vogliamo che r sia grande (perché denominatore) ma non grande al punto da rendere $2^r \gg n$.

Nota Nelle seguenti analisi, il logaritmo è in base 2 perché stiamo operando con numeri binari.

Distinguiamo due casi:

- Se $b < \lfloor \log n \rfloor$, allora per qualche valore $r \leq b$ si ha $(n + 2^r) = O(n)$. In questo caso, conviene scegliere r il più grande possibile, cioè b . Si ottiene quindi il seguente tempo d'esecuzione:

$$\frac{b}{b}(n + 2^b) = \Theta(n)$$

- Il caso più frequente, però, vede la seguente condizione: $b \geq \lfloor \log n \rfloor$. In questo caso, conviene scegliere r tale che sia massimo ma sotto la condizione che $n \geq 2^r \Rightarrow r = \lfloor \log n \rfloor$. Il tempo d'esecuzione risultante è il seguente:

$$\frac{b}{\log n}(n + 2^{\log n}) = \frac{b}{\log n}(n + n) = \frac{b}{\log n}(2n) = \Theta\left(\frac{b}{\log n}n\right)$$

Esercizio d'esame

Dimostrare come ordinare n numeri interi compresi nell'intervallo che va da 0 a $n^4 - 1$ nel tempo $O(n)$ in modo NON crescente.

Cominciamo col trarre alcune considerazioni di base, e a costruire a partire da queste la soluzione.

Poiché l'esercizio richiede un tempo d'esecuzione lineare rispetto a n , dovremo utilizzare uno dei due algoritmi di ordinamento lineare che abbiamo studiato: il *Counting Sort* e il *Radix Sort*.

Non risulta possibile utilizzare il primo, poiché sappiamo che il suo tempo d'esecuzione dipende dall'intervallo dei valori contenuti nell'array:

$$T(n) = \Theta(n + k) = \Theta(n + n^4) = \Theta(n^4)$$

che non è lineare rispetto a n .

Ad esclusione, dovremo utilizzare il *Radix Sort*. Servendoci dei ragionamenti compiuti nel paragrafo precedente, decidiamo di rappresentare i numeri contenuti nell'array in base n : dal momento che $\log_n n^4 = 4$, sappiamo che tutti i numeri contenuti nell'intervallo $[0, n^4 - 1]$ sono di esattamente 4 cifre, se rappresentati in base n .

Quindi, $d = 4$ e $k = n$; quest'ultima considerazione soddisfa l'assunzione che $k = O(n)$, condizione necessaria affinché l'algoritmo ausiliario del *Radix Sort*, il *Counting Sort*, abbia tempo d'esecuzione lineare rispetto a n .

```

1  radixsort(int A[], int n)
2      for i = 1 to 4
3          countingsort(A, n, i)
4
5  countingsort(int A[], int n, int i)
6      alloco un vettore B[1, ..., n]
7      alloco un vettore delle occorrenze C[0, ..., n-1]
8      for j = 0 to n - 1
9          C[j] = 0
10     for j = 1 to n
11         C[cifra(A[j], n, i)]++
12     for j = n - 2 down to 0
13         C[j] = C[j] + C[j + 1]
14     for j = n down to 1
15         B[C[cifra(A[j], n, i)]] = A[j]
16         C[cifra(A[j], n, i)]--
17     copia vettore B in A
18
19 cifra(int x, int n, int i)
20     return  $\left(\frac{x}{n^{i-1}}\right) \% n$ 
```

Osservazioni

- Poiché $k = n$, tutti i cicli del *Counting Sort* hanno costo $\Theta(n)$. Inoltre, poiché la procedura **cifra** ha tempo d'esecuzione costante e il *Radix Sort* chiama l'algoritmo ausiliario 4 volte, il tempo d'esecuzione complessivo sarà $\Theta(4n) = \Theta(n)$, lineare.
- Dal momento che l'esercizio richiede l'ordinamento non crescente, invece di calcolare le somme prefisse come nell'algoritmo di base, noi nel ciclo delle righe 12-13 abbiamo calcolato le **somme postfisse**: alla fine di questo ciclo, il vettore delle occorrenze ci fornisce l'informazione su quanti elementi maggiori o uguali di j sono presenti nel vettore da ordinare, dove j è l'indice che scorre il vettore; formalmente, scriveremo:

$$C[i] = |\{x \in \{1, \dots, n\} | A[x] \geq i\}|$$

- Questa soluzione può essere generalizzata per ogni intervallo $[0, n^c - 1]$, dove c è una costante.

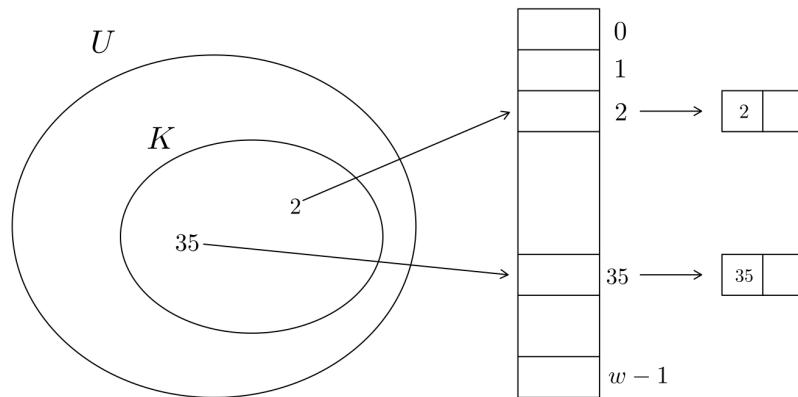
18 Tabelle Hash

18.1 Tabelle ad indirizzamento diretto

Una tabella ad indirizzamento diretto è un tipo di dato utile alla memorizzazione di dati indicizzati su un numero che prende il nome di **chiave**. Un'applicazione delle tabelle ad indirizzamento diretto prevede le seguenti caratteristiche sui suoi elementi:

- Sono costituite da un insieme dinamico al fine dell'implementazione delle operazioni di inserimento, ricerca e cancellazione.
- Ogni elemento ha una chiave estratta dall'universo $U = \{0, 1, \dots, w - 1\}$, dove w non è troppo grande.
- Nessun elemento ha la stessa chiave.

La struttura dati che permette di gestire in modo efficiente questo tipo di dato è un array T con posizioni $[0, \dots, w - 1]$, dove w corrisponde alla cardinalità dell'universo delle chiavi, rappresentato con U .



Ogni posizione corrisponde ad una chiave in U : se nella tabella è presente l'elemento x con chiave k , allora $T[k]$ contiene un puntatore a x ; altrimenti, se la tabella non contiene l'elemento x , $T[k] = \text{NIL}$.

Poiché la tabella non memorizza necessariamente tutte le chiavi dell'universo bensì solamente un suo sottoinsieme, che definiamo K , è possibile assegnare un valore arbitrario, diciamo -1, alle posizioni dell'array che non memorizzano informazioni relative alle chiavi che corrispondono al loro indice.

Operazioni

- **Ricerca**

```
1 direct_address_search(T, k)
2   return T[k]
```

- **Inserimento**

```
1 direct_address_insert(T, x)
2   T[x.Key] = x
```

- **Cancellazione**

```
1 direct_address_delete(T, x)
2   T[x.Key] = NIL
```

Un importante vantaggio delle tabelle ad indirizzamento diretto è costituito dal tempo d'esecuzione delle possibili operazioni su di esse: le operazioni di inserimento, ricerca e cancellazione hanno infatti tempo d'esecuzione costante.

Tuttavia, uno svantaggio non trascurabile è dato dalla complessità spaziale di questo tipo di dato: infatti, lo spazio utilizzato è proporzionale all'universo di tutte le chiavi ($w = |U|$) e non al numero n di elementi effettivamente memorizzati; questo porta ad un inevitabile spreco di memoria.

18.2 Tabelle Hash

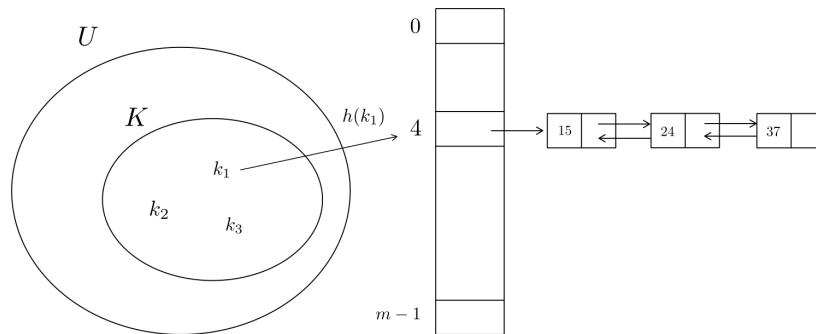
Le **Tabelle Hash** offrono un'importante alternativa alle tabelle ad indirizzamento diretto, da cui non ereditano l'elevata complessità spaziale.

Le tabelle hash si basano sul seguente principio: invece di memorizzare la chiave k nella posizione k , applichiamo a k una **funzione hash** h che ritorna un indice, $h(k)$, dove andremo a memorizzare l'elemento. Questo serve a ridurre il numero di elementi della tabella hash da w a m : la funzione hash infatti è una funzione definita in questo modo:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

Significa che ha come dominio l'universo U e restituisce un intero compreso nell'intervallo $[0, m - 1]$, dove m , la dimensione della tabella, è generalmente molto più piccola di $|U|$: infatti, se l'insieme delle chiavi da memorizzare è molto più piccolo dell'universo di tutte le possibili chiavi U , allora una tabella hash richiede molto meno spazio di una tabella ad indirizzamento diretto; più precisamente, lo spazio di una tabella hash può essere ridotto a $\Theta(|K|)$, dove $|K|$ è la cardinalità dell'insieme delle chiavi utilizzate.

Il tempo di ricerca in una tabella hash è costante nel caso medio, ma non lo è nel caso peggiore. Affronteremo meglio la questione quando parleremo dei diversi metodi di hashing.



In questo esempio di tabella hash, ogni cella dell'array che costituisce la tabella hash contiene un puntatore ad una lista singolarmente o doppiamente concatenata, all'interno della quale sono contenuti tutti gli elementi le cui chiavi vengono mappate con lo stesso valore hash. Questa riflessione anticipa la questione delle **collisioni**, di cui ci occuperemo subito.

Nota Ricordiamo la distinzione tra i diversi campi di una tabella hash:

- $w = |U|$: la cardinalità dell'universo, cioè il numero di tutte le possibili chiavi.
- $n = |K|$: la cardinalità dell'insieme delle chiavi memorizzate.
- m : la dimensione della tabella hash.

18.3 Collisioni

Le tabelle hash possono soffrire del problema delle **collisioni**, situazione che si verifica quando un elemento che vogliamo inserire è mappato in una cella già occupata. Può succedere che si verifichi una collisione quando $|U| > m$, mentre succedono sicuramente se $|K| > m$.

Nell'esempio nella pagina precedente, è possibile notare come l'elemento avente chiave k_1 venga mappato da h nella posizione dell'array avente indice 4. Questa posizione però contiene già un puntatore ad una lista di elementi aventi chiavi 15, 24 e 37: ciò significa che $h(15) = h(24) = h(37) = 4$.

Esistono due modi per risolvere le collisioni:

1. **Concatenamento (o Liste di collisione)**: ad ogni cella corrisponde una lista di elementi contenenti tutte le chiavi mappate con lo stesso indice.
2. **Indirizzamento aperto**: non viene utilizzata nessuna struttura esterna, solamente la tabella hash stessa; se un elemento da inserire viene mappato su una cella occupata, ci sarà bisogno di trovare un metodo per mappare l'elemento in una cella diversa.

18.3.1 Concatenamento

Per risolvere il problema delle collisioni, il metodo del concatenamento ci offre la possibilità di inserire tutti gli elementi associati alla stessa cella in una lista concatenata. Diciamo che la j -esima cella contiene un puntatore alla testa della lista di tutti gli elementi mappati nella posizione j , se esistono in numero pari o superiore ad uno, altrimenti contiene un puntatore a NIL.

Operazioni con il concatenamento

- **Inserimento**

```
1 chained_hash_insert(T, x)
2     inserisce x in testa alla lista T[h(x.Key)]
```

Il tempo d'esecuzione è costante perché:

- L'inserimento in testa ad una lista è costante.
- Assumiamo l'ipotesi che il calcolo della funzione hash sia costante.
- Si suppone che l'elemento da inserire non sia presente nella lista; se necessario, si può cercare se esiste un elemento con chiave $x.Key$: in tal caso, il costo dell'operazione diventa pari al costo della sua ricerca.

- **Ricerca**

```

1  chained_hash_search(T, k)
2      ricerco un elemento con chiave k nella lista T[h(k)]

```

Il tempo d'esecuzione nel caso peggiore è proporzionale alla lunghezza della lista nella cella $h(k)$, ma affronteremo meglio la questione dopo aver parlato dell'operazione di cancellazione.

- **Cancellazione**

```

1  chained_hash_delete(T, x)
2      cancella x dalla lista T[h(x.Key)]

```

In questa implementazione dell'operazione di cancellazione, x è un puntatore all'elemento da cancellare. Questa considerazione fa sì che per cancellare l'elemento x non è necessaria la sua ricerca.

Inoltre, se le liste sono doppiamente concatenate, l'operazione di cancellazione ha tempo d'esecuzione costante, in quanto non è necessario ricercare i puntatori degli elementi precedente e successivo: se le liste non dovessero essere doppiamente concatenate, bensì solo singolarmente, allora bisognerebbe compiere l'operazione di ricerca per trovare gli elementi precedente e successivo di x in modo tale da poter sistemare i puntatori.

18.3.2 Analisi dell'hashing con concatenamento

Vogliamo stimare il tempo d'esecuzione della ricerca di un elemento con chiave k .

Il caso peggiore si verifica quando tutte le chiavi sono mappate nella stessa cella: in questo caso, è presente un'unica lista di lunghezza n e il tempo d'esecuzione è $O(n)$, più il tempo necessario al calcolo della funzione hash.

Per quanto riguarda il caso medio, questo dipende dal modo in cui la funzione hash distribuisce mediamente le chiavi fra le m celle.

Hashing uniforme semplice La tecnica dell'hashing uniforme semplice si avvale della seguente ipotesi.

Ipotesi Qualsiasi elemento ha la stessa probabilità di essere mandato in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono mandati gli altri elementi.

$$\forall i \in [0, \dots, m-1], \quad Q(i) = \frac{1}{m}$$

$Q(i)$ indica la probabilità che una chiave qualsiasi finisca nella i -esima cella.

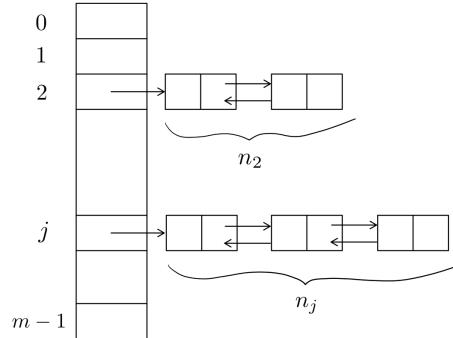
Definizione Il **fattore di carico** di una tabella hash costituita da m celle e n chiavi è $\alpha = \frac{n}{m}$.

Osservazione $\alpha < 1 \vee \alpha > 1 \vee \alpha = 1$. Quando però $\alpha > 1$, bisogna fare attenzione che non sia troppo grande, altrimenti si perderebbe in prestazioni.

Sotto le ipotesi di hashing uniforme semplice, indichiamo con n_j la lunghezza della lista $T[j]$.

Il valore medio di n_j è:

$$\frac{n_0 + n_1 + \dots + n_{m-1}}{m} = \frac{n}{m} = \alpha$$



Ipotesi La funzione hash è calcolata in tempo $O(1)$.

Teorema In una tabella hash in cui le collisioni sono risolte col metodo del concatenamento, una **ricerca senza successo** richiede tempo $\Theta(1 + \alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

Dimostrazione Per intuizione,

- Calcolo $j = h(k)$, operazione che costa $O(1)$
- Accedo alla lista $T[j]$, operazione che costa $O(1)$
- Scorro la lista $T[j]$, operazione che costa $\Theta(\alpha)$

Dunque, il tempo d'esecuzione è $\Theta(1 + \alpha)$, dove l'1 si deve al calcolo della funzione hash.

Teorema In una tabella hash in cui le collisioni sono risolte col metodo del concatenamento, una **ricerca con successo** richiede tempo $\Theta(1 + \alpha)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

Dimostrazione Per intuizione,

- Calcolo $j = h(k)$, operazione che costa $O(1)$
- Accedo alla lista $T[j]$, operazione che costa $O(1)$
- Scorro la lista fino a trovare k , operazione che costa mediamente $\Theta(\frac{\alpha}{2})$

Dunque, il tempo d'esecuzione è $\Theta(1 + \frac{\alpha}{2}) = \Theta(1 + \alpha)$.

Interpretazione Se n è limitato superiormente da m , cioè $n = O(m)$, si verifica che $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$ e quindi la ricerca risulta costante. Questo implica che, quando n cresce, è necessario riallocare raddoppiandone le dimensioni affinché sia possibile mantenere questo rapporto costante, portando tuttavia ad un inevitabile calo di prestazioni.

18.3.3 Funzioni hash

Una funzione hash (dall'inglese "tritare", "polpetta") è una funzione che "spezzetta" la nostra chiave e la ricomponga casualmente in modo tale da ottenere un intero che risulti il più casuale possibile. Lo scopo è quello di distribuire in modo uniforme gli elementi nella nostra tabella.

Nella situazione più "fortunata" che ci può capitare, le chiavi sono numeri reali k , casuali e distribuiti in modo indipendente e uniforme nell'intervallo $0 \leq k < 1$. Quando le nostre chiavi rispettano queste condizioni, la seguente funzione hash offre risultati ideali:

$$h(k) = \lfloor k \cdot m \rfloor,$$

funzione che soddisfa la condizione di hashing uniforme semplice.

Codifica delle informazioni Prima di affrontare i vari metodi di hashing, è opportuno fare una digressione sul modo in cui vengono processate le chiavi. Solitamente, le funzioni hash assumono che le chiavi siano numeri naturali, cioè appartenenti a $\mathbb{N} = \{0, 1, \dots\}$. Qualora le chiavi non dovessero essere numeri naturali, occorre interpretarli come tali.

Prendiamo come esempio delle stringhe di caratteri. Supponiamo di avere la stringa "CLRS"; come sappiamo, ogni carattere della stringa è codificato secondo lo standard ASCII, che ci permette di rappresentare $128 = 2^7$ possibili caratteri: al carattere 'C' equivale l'intero 67, a 'L' 76, a 'R' 82 e a 'S' 83. Una stringa, interpretata attraverso la **notazione posizionale**, può essere convertita nel seguente modo: nella stringa d'esempio, "CLRS", il carattere 'C' assume la posizione 3, 'L' la posizione 2, 'R' la posizione 1 e 'S' la posizione 0. Attuiamo la conversione:

$$\begin{aligned} \text{CLRS} &= 128^0 \cdot S + 128^1 \cdot R + 128^2 \cdot L + 128^3 \cdot C \\ &= 128^0 \cdot 83 + 128^1 \cdot 82 + 128^2 \cdot 76 + 128^3 \cdot 67 \\ &= 141.764.947 \end{aligned}$$

Metodo della divisione

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

Un famoso metodo di hashing consiste nel prendere come valore hash il resto della divisione tra la chiave e la dimensione della tabella. Per esempio, con $m = 19$ e $k = 31$, la funzione hash produce il risultato $h(31) = 12$.

Il più importante vantaggio di questa implementazione è sicuramente la semplicità della realizzazione, tuttavia questo beneficio non basta per compensare gli innumerevoli svantaggi: primo fra tutti, è importante sottolineare il fatto che questa tecnica rende la dimensione della tabella un dato critico; è dunque bene evitare certi valori per m , altrimenti si rischia di produrre un hashing scorretto.

- m non può assumere valori pari a potenze di 2: se $m = 2^p$, $h(k)$ rappresenta i p bit meno significativi di k ; noi vogliamo che il valore hash dipenda da tutti i bit della divisione, per cui occorre evitare le potenze di 2.

- Nel caso in cui dovessimo hashare una stringa k interpretata nella base 2^p , una dimensione della tabella pari a $m = 2^p - 1$ non è idonea al nostro scopo: questa dimensione farebbe in modo che una permutazione dei caratteri di k non cambi il valore hash.

Per esempio, se $k = \text{"CLRS"}$ codificata secondo lo standard ASCII e $m = 127 = 2^7 - 1$, si ha che $h(k) = k \bmod 127 \Rightarrow h(\text{cod(CLRS)}) = h(\text{cod(SRCL)})$.

- Una buona scelta per m è un numero primo non troppo vicino a una potenza esatta di 2 o di 10.

Per esempio, si vogliono memorizzare $n = 2000$ chiavi, prevedendo una media di 3 collisioni: poniamo come dimensione della tabella un numero primo che rispetti le caratteristiche di cui sopra vicino a $\frac{2000}{3} = 666.6$; scegliamo arbitrariamente $m = 701$: questo valore ci permette di spargere in modo opportuno e ben distribuito gli elementi nella tabella.

Metodo della moltiplicazione

$$h(k) = \lfloor m \cdot k \rfloor, \text{ con } 0 \leq k < 1$$

Notiamo che questa corrisponde alla prima formulazione di funzione hash che abbiamo descritto nel capitolo.

L'idea di questa strategia sta nel fatto che data una chiave che è un numero naturale $k \in U$, decidiamo di trasformarla in un numero $\bar{k} \in [0, 1[$ per poi applicare la funzione hash di cui sopra.

I passaggi di questo algoritmo di hashing sono i seguenti:

- Fisso una costante A , $0 < A < 1$
- Calcolo $k \cdot A$, che dà una parte intera e una parte frazionaria
- Estraiamo la parte frazionaria calcolata precedentemente:

$$k \cdot A \bmod 1 = k \cdot A - \lfloor k \cdot A \rfloor \in [0, 1]$$

Dunque, la nostra funzione hash $h(k)$, data una chiave k , produrrà il seguente risultato:

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

Vantaggi

- Il valore di m non è critico.
- Funziona bene con potenzialmente tutti i possibili valori di A .
- In particolare, il matematico Donald Knuth ha osservato che tale algoritmo di hashing funziona particolarmente bene se $A \simeq \hat{\Phi}$, cioè l'inverso del rapporto aureo:

$$A \simeq \frac{\sqrt{5} - 1}{2} = 0.610033$$

Semplificare il calcolo della funzione hash

Sia w la lunghezza di una parola di memoria, e k entri in una singola parola di memoria. Scegliamo un intero q , $0 < q < 2^w$, e $m = 2^p$, cioè una potenza di due. Poniamo quindi $A = \frac{q}{2^w} < 1$. La nostra \bar{k} , cioè il prodotto di k e A , diventerà di conseguenza:

$$\bar{k} = k \cdot A = \frac{k \cdot q}{2^w}$$

Che, come abbiamo visto, è composta da una parte intera e una parte frazionaria. Graficamente, possiamo interpretarla nel seguente modo:

$$\begin{array}{r}
 k \quad \boxed{} \\
 q \quad \boxed{} \\
 \hline
 k \cdot q \quad \boxed{}, \boxed{} \\
 \underbrace{}_{\frac{\lfloor k \cdot q \rfloor}{2^w}} \quad \underbrace{}_{\frac{k \cdot q \bmod 1}{2^w}}
 \end{array}$$

E quindi, dal momento che noi siamo interessati alla parte frazionaria del risultato di questa operazione, possiamo scrivere la funzione hash come segue:

$$\begin{aligned}
 h(k) &= \lfloor m \cdot (k \cdot A \bmod 1) \rfloor \\
 &= \lfloor 2^p \cdot (k \cdot A \bmod 1) \rfloor
 \end{aligned}$$

Possiamo concludere osservando che tale operazione di hashing rappresenta i p bit più significativi della parola meno significativa del prodotto di $k \cdot q$.

Hashing universale Possiamo compiere un’ulteriore digressione sui metodi di hashing ponendo la seguente osservazione: ciò che noi vogliamo più di ogni altra cosa, quando produciamo dei valori hash, è evitare determinati input che diano come risultato sempre lo stesso output; se ciò si dovesse verificare, avremmo tutti gli elementi della nostra tabella mappati in una sola cella.

Un modo per evitare questa situazione consiste nell’avere a nostra disposizione un insieme H di funzioni hash opportunamente costituito, e nel fare in modo che il programma, all’inizio dell’esecuzione, scelga casualmente una funzione $h \in H$.

18.3.4 Indirizzamento aperto

L'altro modo per gestire il problema delle collisioni prende il nome di indirizzamento aperto. L'idea alla base di questa tecnica prevede che tutti gli elementi siano memorizzati nella tabella stessa, senza la necessità di una memoria ausiliaria esterna. In questa implementazione, ogni cella contiene un elemento dell'insieme dinamico oppure un valore arbitrario, che può essere **NIL** o **DELETED**, che vedremo nel dettaglio più avanti.

Per cercare un elemento di chiave k all'interno della tabella,

- a) Calcolo $h(k)$ ed esamino la cella con indirizzo $h(k)$; questa operazione prende il nome di **ispezione**.
- b) Se la cella $h(k)$ contiene la chiave k , la ricerca ha successo. Se la cella contiene **NIL**, la ricerca termina con insuccesso.
- c) Se la cella dovesse contenere una chiave che non è k , allora bisogna calcolare l'indice di un'altra cella, in base alla chiave k e all'**ordine di ispezioni**, cioè al numero (naturale e intero) di ispezioni compiute fino a questo momento.
- d) Si continua la scansione della tabella finché non si trova la chiave k , condizione di successo della ricerca, oppure si trova una cella che contiene **NIL** oppure ho eseguito m ispezioni senza successo, condizioni di insuccesso della ricerca.

Questo implica che la funzione hash ora vada modificata in modo tale da prendere due argomenti: diremo che $h(k, i)$ rappresenta la posizione della chiave k dopo i ispezioni fallite (se non avessi fallito, non sarei andato all'ispezione successiva), dove inizialmente $i = 0$. Formalmente, scriveremo:

$$h(k, i) : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

Si richiede, che per ogni possibile chiave $k \in U$, la sequenza di ispezioni $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ sia una permutazione di $\langle 0, 1, \dots, m-1 \rangle$, cioè gli indici della tabella, in modo che ogni posizione della tabella hash possa essere considerata come possibile cella in cui inserire una nuova chiave mentre la tabella si riempie.

Operazioni con indirizzamento aperto

Ipotesi Per motivi di semplificazione del codice, supponiamo che gli elementi contengano solo la chiave, cioè che non abbiano dei dati satelliti.

- **Inserimento**

Post-condizione Restituisce l'indice della cella dove ha memorizzato k oppure segnala un errore se la tabella è piena.

```

1 hash_insert(T, k)
2     i = 0
3     trovata = false
4     repeat
5         j = h(k, i)
6         if T[j] == NIL or T[j] == DELETED
7             T[j] = k
8             trovata = true
9         else
10            i = i + 1
11        until trovata or i == m
12        if trovata
13            return j
14        else error "overflow della tabella hash"

```

Osservazioni

- Il motivo per cui è presente la condizione `if ... T[j] == DELETED` si deve al problema che nasce dalla cancellazione di un elemento da una tabella hash ad indirizzamento aperto, che vedremo tra poco.

- **Ricerca**

Post-condizione Restituisce j se la cella j contiene la chiave k oppure NIL se la chiave k non si trova nella tabella T .

```

1 hash_search(T, k)
2     i = 0
3     trovata = false
4     repeat
5         j = h(k, i)
6         if T[j] == k
7             trovata = true
8         else
9             i = i + 1
10        until trovata or i == m or T[j] == NIL
11        if trovata
12            return j
13        else
14            return NIL

```

Osservazioni

- Il motivo per cui è presente la condizione $\dots \text{ or } T[j] == \text{NIL}$ si deve al fatto che se, effettuando la ricerca di un elemento con chiave k , si trova una cella senza nessuna chiave, allora la chiave k dovrebbe essere mappata in quella cella, ma essendo essa vuota, l'assenza della chiave in quella cella implica l'assenza della chiave nell'intera tabella, per cui la ricerca può terminare.

• Cancellazione

La cancellazione da una tabella hash con indirizzamento aperto, come abbiamo già anticipato, causa una scomoda problematica.

Supponiamo di avere la tabella hash di cui sotto, e di voler effettuare l'inserimento della chiave 12.

	101			32			45
0	1	2	3	4	5	6	7

1.	0	Al primo tentativo di inserimento, la funzione hash produce il risultato $h(12, 0) = 4$, ma la cella 4 è già occupata: si verifica una collisione.
	1	
	2	
	3	Al secondo tentativo, $h(12, 1) = 1$, ma la cella 1 è già occupata: si verifica un'altra collisione.
	4	
	5	Al terzo tentativo, $h(12, 2) = 3$: la cella è libera, quindi l'elemento con chiave 12 viene inserito in posizione 3.
	6	
	7	

2.	0	Supponiamo ora di voler eliminare l'elemento avente chiave 101: ora, in posizione 1 troveremo il valore NIL .
	1	
	2	Se volessimo compiere l'operazione di ricerca per l'elemento avente chiave 12, chiaramente, andremmo a compiere tutte le ispezioni già compiute per l'inserimento: $h(12, 0) = 4$, ma in posizione 4 non si trova la chiave 12, quindi proseguiamo all'ispezione successiva.
	3	
	4	
	5	
	6	
	7	$h(12, 1) = 1$, ma in posizione 1 troviamo il valore NIL .

L'aver trovato **NIL** invece della chiave corretta è un problema, perché stando alle nostre ipotesi, **NIL** ci segnala che l'elemento con chiave 12 non sia presente in T (se fosse presente, sarebbe mappato nella cella che contiene **NIL**), quando in realtà è presente.

Per risolvere questo problema, invece di porre a **NIL** un elemento rimosso, utilizzeremo un altro valore arbitrario, **DELETED**, che ci serve per

distinguere il caso in cui un elemento *potrebbe* essere presente in T ma mappato in una cella diversa, e il caso in cui l'elemento cercato non è proprio presente in T .

È questo il motivo per cui abbiamo un ulteriore controllo nella procedura di inserimento: infatti, per prevenire questa eventualità, la procedura `hash_insert` viene modificata alla riga 6 aggiungendo `or T[j] == DELETED` (prima era solamente `if T[j] == NIL`); la procedura `hash_search` invece non richiede alcuna modifica, perché k non è mai uguale a NIL o DELETED.

Svantaggio Con questa implementazione, l'impiego del valore speciale DELETED comporta il fatto che il tempo di ricerca non dipende più dal fattore di carico $\alpha = \frac{n}{m}$. Questo, chiaramente, porta ad un calo delle prestazioni. Per questo motivo, in caso di collisioni, se sappiamo che delle chiavi dovranno essere cancellate è preferibile usare la strategia del concatenamento invece della strategia dell'indirizzamento aperto.

18.3.5 Metodi di scansione/ispezione

La situazione ideale per l'hashing con indirizzamento aperto è l'**hashing uniforme**: ogni chiave ha la stessa probabilità di avere come sequenza di ispezioni una delle $m!$ permutazioni di $< 0, 1, \dots, m - 1 >$. Significa che, ad ogni iterazione,

- $h(k, 0)$ si distribuisce uniformemente sulle m celle
- $h(k, 1)$ si distribuisce uniformemente sulle $m - 1$ celle rimaste
- $h(k, 2)$ si distribuisce uniformemente sulle $m - 2$ celle rimaste
- \vdots
- \simeq hashing uniforme semplice ad ogni iterazione

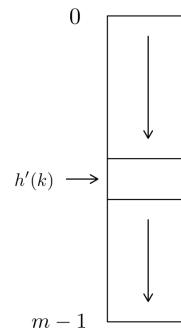
Questo risultato è difficile da ottenere, motivo per cui nella realtà sono usate delle approssimazioni di questa tecnica.

Ispezione lineare

Data una funzione hash ordinaria $h' : U \rightarrow \{0, 1, \dots, m - 1\}$ che chiameremo **funzione hash ausiliaria**, il metodo dell'ispezione lineare usa la seguente funzione hash:

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

La prima cella esaminata è $T[h'(k)]$, poi continua a scandire tutte le celle sequenzialmente fino alla cella $m - 1$; infine riprende dalla cella 0 fino alla cella $T[h'(k) - 1]$.



Esempio Supponiamo di avere una tabella hash con $m = 13$ posizioni, e di voler inserire in essa, nell'ordine, le seguenti chiavi: 69, 4, 31, 43. Le funzioni hash sono definite di seguito:

$$h_1(k) = k \bmod m$$

$$h(k, i) = (h_1(k) + i) \bmod 13$$

Allora, l'ordine degli inserimenti sarà il seguente:

1. $h(69, 0) = (h_1(69) + 0) \bmod 13 = 4 \bmod 13 = 4$
Non c'è collisione, quindi $T[4] = 69$.
2. $h(4, 0) = (h_1(4) + 0) \bmod 13 = 4 \bmod 13 = 4$
C'è collisione, quindi $i = 1$.
3. $h(4, 1) = (h_1(4) + 1) \bmod 13 = 5 \bmod 13 = 5$
Non c'è collisione, quindi $T[5] = 4$.
4. $h(31, 0) = (h_1(31) + 0) \bmod 13 = 5 \bmod 13 = 5$
C'è collisione, quindi $i = 1$.
5. $h(31, 1) = (h_1(31) + 1) \bmod 13 = 6 \bmod 13 = 6$
Non c'è collisione, quindi $T[6] = 31$.
6. $h(43, 0) = (h_1(43) + 0) \bmod 13 = 4 \bmod 13 = 4$
C'è collisione, quindi $i = 1$.
7. $h(43, 1) = (h_1(43) + 1) \bmod 13 = 5 \bmod 13 = 5$
C'è collisione, quindi $i = 2$.
8. $h(43, 2) = (h_1(43) + 2) \bmod 13 = 6 \bmod 13 = 6$
C'è collisione, quindi $i = 3$.
9. $h(43, 3) = (h_1(43) + 3) \bmod 13 = 7 \bmod 13 = 7$
Non c'è collisione, quindi $T[7] = 7$.

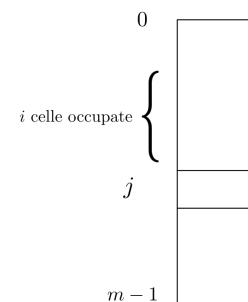
0	
1	
2	
3	
4	69
5	4
6	31
7	43
8	
9	
10	
11	
12	

Vantaggio Facilità di implementazione.

Svantaggio Questa tecnica vede la formazione di **agglomerazioni/addensamenti primari**, cioè si formano lunghe file di celle adiacenti occupate, che occupano il tempo di ricerca.

Si può notare come la prima cella ispezionata determini l'intera sequenza di ispezioni: questo significa che per ogni chiave ci sono soltanto m sequenze di ispezioni distinte.

Data una generica cella j preceduta da i celle occupate, la probabilità che la cella j sia occupata è pari a $\frac{i+1}{m}$.



Ispezione quadratica

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

dove h' è una funzione hash ausiliaria, c_1 e c_2 sono due costanti ausiliarie diverse da 0 e i può assumere tutti i valori da 0 a $m - 1$.

Dei valori esemplificativi che possiamo adottare per questo metodo di hashing sono i seguenti: $c_1 = c_2 = \frac{1}{2}$, $m = 2^p$ per un qualche p (m è una potenza di 2). L'hashing quadratico soffre, analogamente alla sua variante lineare, della formazione di **addensamenti secondari**: se due chiavi distinte k_1 e k_2 che sono mappate tramite la funzione ausiliaria h' nello stesso valore, cioè $h'(k_1) = h'(k_2)$, allora le due chiavi generano la stessa sequenza di ispezioni. Si nota quindi che anche in questo caso la prima posizione determina l'intera sequenza di ispezioni, e dunque per una chiave ci sono soltanto m sequenze di ispezioni distinte.

Doppio hashing

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

dove h_1 e h_2 sono due funzioni hash ausiliarie e i può assumere tutti i valori da 0 a $m - 1$. In questa tecnica di hashing, h_1 serve per determinare il punto di partenza (è un valore che non cambia mai in funzione di i), mentre h_2 serve per determinare il passo delle ispezioni.

Vediamo che risultati produce questo metodo di scansione ponendoci nelle stesse condizioni dell'esempio visto nel caso dell'ispezione lineare; vogliamo inserire in una tabella hash con $m = 13$ le chiavi 69, 4, 31 e 43, e abbiamo le seguenti funzioni hash:

$$h_1(k) = k \bmod 13, \quad h_2(k) = 1 + (k \bmod 11)$$

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod 13$$

- | | |
|----|----|
| 0 | |
| 1 | |
| 2 | 43 |
| 3 | |
| 4 | 69 |
| 5 | 31 |
| 6 | |
| 7 | |
| 8 | |
| 9 | 4 |
| 10 | |
| 11 | |
| 12 | |
1. $h(69, 0) = (h_1(69) + 0) \bmod 13 = 4 \bmod 13 = 4$
Non c'è collisione, quindi $T[4] = 69$.
 2. $h(4, 0) = (h_1(4) + 0) \bmod 13 = 4 \bmod 13 = 4$
C'è collisione, quindi $i = 1$.
 3. $h(4, 1) = (h_1(4) + h_2(4)) \bmod 13 = 9 \bmod 13 = 9$
Non c'è collisione, quindi $T[9] = 4$.
 4. $h(31, 0) = (h_1(31) + 0) \bmod 13 = 5 \bmod 13 = 5$
Non c'è collisione, quindi $T[5] = 31$.
 5. $h(43, 0) = (h_1(43) + 0) \bmod 13 = 4 \bmod 13 = 4$
C'è collisione, quindi $i = 1$.
 6. $h(43, 1) = (h_1(43) + h_2(43)) \bmod 13 = 15 \bmod 13 = 2$
Non c'è collisione, quindi $T[2] = 43$.

Come possiamo notare, a parità di dimensione della tabella hash, di numero e di valore delle chiavi da inserire, il doppio hashing ha impiegato 2 collisioni invece delle 5 della scansione lineare, e la tabella hash risultante vede le chiavi distribuite in maniera più uniforme, aspetti che mettono in risalto le migliori prestazioni della tecnica del doppio hashing: usa infatti $\Theta(m^2)$ sequenze di ispezioni, perché ogni possibile coppia $(h_1(k), h_2(k))$ produce una distinta sequenza di ispezioni. Poiché è il metodo che si avvicina di più alla strategia dell'hashing uniforme, il doppio hashing è molto usato nel mondo reale.

Come si costruisce la funzione hash?

Per costruire una funzione hash efficiente, è necessario che il valore $h_2(k)$ sia relativamente primo con la dimensione m della tabella hash, affinché questa possa essere ispezionata completamente.

Sono esposte di seguito due possibilità per la costruzione della funzione $h_2(k)$:

1. Si può scegliere $m = 2^p$ per un qualche p (m è una potenza di 2). Questo vuol dire che m è un numero pari. Definisco quindi $h_2(k)$ in modo che produca sempre numeri dispari: in questo modo m e $h_2(k)$ saranno sempre relativamente primi.

Per esempio, $m = 2^p$ e $h_2(k) = 2 \cdot h'(k) + 1$, dove h' è una qualsiasi funzione hash già vista.

2. Si può scegliere m numero primo, e definire $h_2(k)$ in modo che generi sempre un intero positivo minore strettamente di m .

Per esempio, scegliamo m primo, $h_1(k) = k \bmod m$ e $h_2(k) = 1 + (k \bmod m')$, dove $m' < m$. Notiamo che questa è la soluzione implementata nell'esempio usato per confrontare la scansione lineare e il doppio hashing.

Esercizio Dimostrare che se $h_2(k)$ e m sono relativamente primi, e $h(k, i) = h(k, i')$ con $i, i' < m$, allora $i = i'$.

18.3.6 Analisi dell'hashing a indirizzamento aperto

Per analizzare il tempo d'esecuzione delle operazioni di ricerca e inserimento nel caso di hashing con indirizzamento aperto, è necessario porre alcune ipotesi sulla natura dell'hashing e sulle operazioni sulle chiavi: in particolare,

Hp. 1 Si assume l'utilizzo di hashing uniforme.

Hp. 2 Si assume che non venga compiuta l'operazione di cancellazione.

L'ipotesi sull'operazione di cancellazione è essenziale affinché possiamo compiere l'analisi in termini del **fattore di carico** $\alpha = \frac{n}{m}$: come abbiamo visto nel caso del concatenamento, la cancellazione delle chiavi comporta un aumento del tempo d'esecuzione, per cui decidiamo di attenerci alla sola considerazione del fattore di carico come indice prestazionale.

In particolare, notiamo che nel caso di indirizzamento aperto, $0 \leq \alpha \leq 1$: a differenza del concatenamento, infatti, ora il numero massimo di chiavi che possiamo memorizzare è m , dunque quando $n = m$, $\alpha = 1$.

Teorema Nell'ipotesi di hashing uniforme, data una tabella hash a indirizzamento aperto con un fattore di carico $\alpha = \frac{n}{m} < 1$, il numero atteso di ispezioni in una ricerca senza successo (*i.e.* nel caso peggiore) risulta essere al massimo $\frac{1}{1-\alpha}$.

(Sketch della) Dimostrazione Se $\alpha < 1$, allora ci sono delle celle vuote ($n < m$), quindi nel compiere la mia ricerca posso fermarmi alla prima cella vuota. Ragioniamo in termini di probabilità:

1. Una prima ispezione è sempre compiuta: $\mathbb{P}[I] = 1$.
2. La probabilità che venga compiuta la seconda ispezione equivale alla probabilità che la prima cella sia occupata: $\mathbb{P}[II] = \frac{n}{m}$.
3. La probabilità che venga compiuta la terza ispezione equivale alla probabilità che anche la seconda cella sia occupata: $\mathbb{P}[III] = \frac{n}{m} \cdot \frac{n-1}{m-1}$.
4. La probabilità che venga compiuta la quarta ispezione equivale alla probabilità che anche la terza cella sia occupata: $\mathbb{P}[IV] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2}$.

$n. \quad :$

Notiamo che, dal momento che $\frac{n}{m} = \alpha$, possiamo maggiorare ad α tutti i termini che costituiscono le probabilità che le singole celle siano occupate:

$$\mathbb{P}[I] = 1, \quad \mathbb{P}[II] = \alpha, \quad \mathbb{P}[III] \simeq \alpha^2, \quad \mathbb{P}[IV] \simeq \alpha^3, \quad \dots$$

e approssimare di conseguenza il valore atteso:

$$1 + \alpha + \alpha^2 + \dots \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Interpretazione Se α è costante, una ricerca senza successo viene eseguita in tempo medio $O(1)$.

- Se $\alpha = 0.5$ (tabella piena a metà), il numero medio di ispezioni è al massimo $\frac{1}{1-\frac{1}{2}} = 2$.
- Se $\alpha = 0.9$ (tabella quasi piena), il numero medio di ispezioni è al massimo $\frac{1}{1-\frac{9}{10}} = 10$.

Osserviamo che più la tabella è piena, più è costosa la ricerca.

Corollario L'inserimento di un elemento in una tabella hash a indirizzamento aperto con un fattore di carico α richiede in media non più di $\frac{1}{1-\alpha}$ ispezioni, nell'ipotesi di hashing uniforme.

Dimostrazione Un elemento è inserito nella tabella solo se essa non è satura, cioè se $\alpha < 1$ (condizione del teorema di cui sopra).

L'inserimento di una chiave richiede una ricerca senza successo (perché dobbiamo trovare una posizione vuota) seguita dalla sistemazione della chiave nella prima cella vuota che viene trovata.

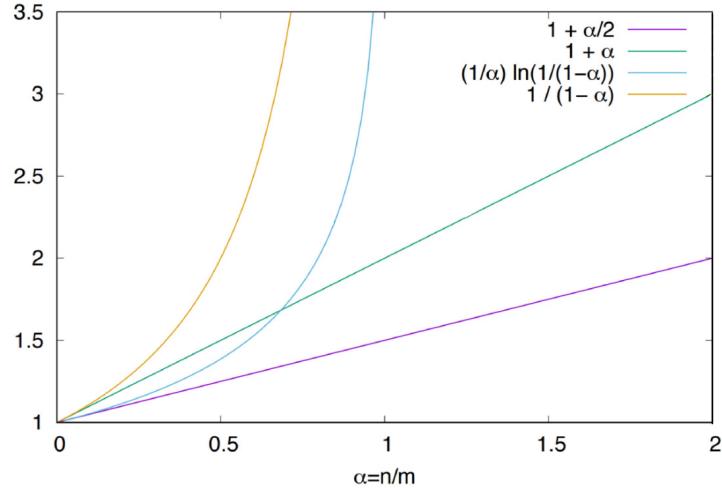
Quindi il numero atteso di ispezioni è dato al massimo da $\frac{1}{1-\alpha}$.

Teorema Data una tabella hash a indirizzamento aperto con un fattore di carico $\alpha < 1$, il numero atteso di ispezioni in una ricerca con successo è al massimo $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$, supponendo che l'hashing sia uniforme e che ogni chiave nella tabella abbia la stessa probabilità di essere cercata.

Interpretazione Se α è costante, una ricerca con successo viene eseguita in tempo medio $O(1)$.

- Se $\alpha = 0.5$, il numero medio di ispezioni in una ricerca con successo è minore di 1.387.
- Se $\alpha = 0.9$, il numero medio di ispezioni in una ricerca con successo è minore di 2.559.

18.4 Concatenamento e indirizzamento aperto a confronto



Il grafico in figura mostra, al crescere di α , il tempo d'esecuzione delle operazioni di ricerca e inserimento in tabelle hash in cui le collisioni sono risolte con il concatenamento e con l'indirizzamento. In particolare,

- Concatenamento - Ricerca con successo
- Concatenamento - Ricerca senza successo
- Indirizzamento aperto - Ricerca con successo
- Indirizzamento aperto - Ricerca senza successo

Ristrutturazione Come si può notare dal grafico, per valori di α superiori a $\frac{1}{2}$, il tempo d'esecuzione delle operazioni su tabelle hash con indirizzamento aperto cresce a ritmo esponenziale. Per questo motivo, quando il valore di α cresce sopra tale soglia, è necessario effettuare una **ristrutturazione** della tabella hash, cioè una riallocazione della tabella con il raddoppiamento della sua dimensione. Si tratta di un'operazione relativamente costosa, perché comporta il cambio delle funzioni hash e quindi una mappatura delle chiavi completamente diversa rispetto a quella precedente.

Quando conviene eseguire la ristrutturazione?

- Nel caso del concatenamento, quando $\alpha \geq 2$; in tal caso, l'operazione di ristrutturazione costerà, nel caso peggiore, $\Theta(m + n)$, dal momento che dobbiamo scorrere tutte le celle della tabella e tutte le liste contenute nelle celle.
- Nel caso dell'indirizzamento aperto, quando $\alpha \geq \frac{1}{2}$; in tal caso, l'operazione di ristrutturazione costerà, nel caso peggiore, $\Theta(m)$.

18.4.1 Strutture dati a confronto

Viene di seguito riportata una tabella utile al confronto del costo delle tradizionali operazioni compiute sulle strutture dati viste fino a questo punto.

Struttura Dati	Ricerca	Predecessore	Massimo	Costruzione
Hash Table - Chaining (caso medio)	$O(1 + \alpha)$	$O(m + n)$	$O(m + n)$	$\Theta(n)$
Hash Table - Chaining (caso peggiore)	$O(n)$	$O(m + n)$	$O(m + n)$	$\Theta(n)$
Hash Table - Ind. aperto (caso medio)	$O\left(\frac{1}{1-\alpha}\right)$	$O(m)$	$O(m)$	$O\left(\frac{n}{1-\alpha}\right)$
Hash Table - Ind. aperto (caso peggiore)	$O(n)$	$O(m)$	$O(m)$	$O(n^2)$
Max Heap	$O(n)$	$O(n)$	$\Theta(1)$	$\Theta(n)$
Alberi binari di ricerca	$O(h)$	$O(h)$	$O(h)$	$\Theta(n \log n)$

19 Programmazione dinamica

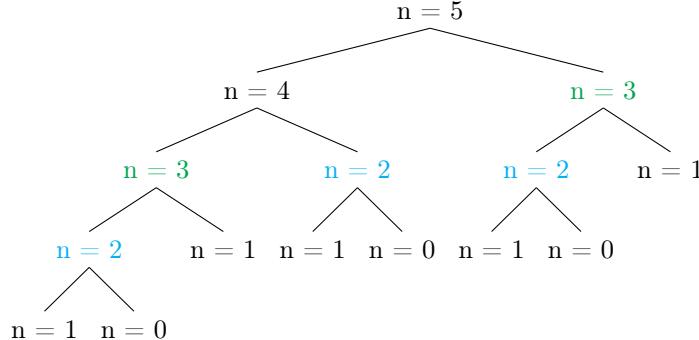
La **programmazione dinamica** è una tecnica di progettazione di algoritmi che si applica in presenza delle seguenti condizioni:

- Un problema si può ridurre ad un insieme di problemi più piccoli (in modo analogo al *Divide et impera*);
- Ma, diversamente dal *Divide et impera*, i sottoproblemi non sono indipendenti, cioè questi sottoproblemi hanno in comune ulteriori sottoproblemi.

Prendiamo come esempio la successione di Fibonacci, che abbiamo già incontrato all'inizio del corso. La sua definizione ricorsiva è la seguente:

$$f(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

Se volessimo calcolare $f(5)$ notiamo, attraverso l'albero delle ricorsioni di seguito riportato, che sottoproblemi come $f(3)$ o $f(2)$ si ripetono più volte, e con un approccio analogo agli schemi del *Divide et impera*, anche i calcoli compiuti per risolverli si ripetono. Il risultato, se calcolato secondo gli schemi tradizionali, è estremamente inefficiente.



L'idea alla base della programmazione dinamica è la seguente: ogni volta che risolvo un problema, salvo la soluzione per evitare di doverla ricalcolare in futuro, in modo tale da risolvere ogni sottoproblema una volta sola.

Questa idea permette di creare algoritmi risolutivi con tempi d'esecuzione polinomiali, o perfino di adattare algoritmi dotati di tempi esponenziali a tempistiche polinomiali rispetto alla dimensione dell'input.

La programmazione dinamica è una tecnica in genere adatta a **problematiche di ottimizzazione**: in numerosissime occasioni, nello sviluppo di un algoritmo, potremmo avere a disposizione diverse possibili soluzioni per il problema, ognuna delle quali ha un costo in termini di tempo; attraverso la programmazione dinamica, siamo in grado di identificare e scegliere una soluzione ottima, cioè di costo minimo o massimo (sulla base delle nostre esigenze), tenendo conto del

fatto che la soluzione ottima può non essere unica.

La memorizzazione delle soluzioni dei sottoproblemi impiega, chiaramente, della memoria ausiliaria: possiamo vedere la programmazione dinamica come un compromesso tra tempo d'esecuzione e spazio utilizzato.

Proprietà della sottostruttura ottima La soluzione ottima è esprimibile come combinazione di soluzioni ottime di sottoproblemi.

La programmazione dinamica è infatti utile in contesti dove:

- I sottoproblemi distinti sono in numero polinomiale
- Ciascun sottoproblema si risolve in tempo polinomiale

Come sviluppare un algoritmo di programmazione dinamica

1. Caratterizzazione della struttura di una soluzione ottimale
2. Fornire una definizione ricorsiva del valore di una soluzione ottima
3. Calcolo del valore di una soluzione ottima
4. Individuazione di una soluzione ottima sulla base delle informazioni calcolate nel punto precedente

Tecniche di costruzione dell'algoritmo

- **Top-Down** Memorizzo in una tabella (vettore, tabella hash, ...) le soluzioni dei problemi già risolto (*memoization*); per questa tecnica serve la ricorsione.
- **Bottom-Up** Ordiniamo i problemi in base alla dimensione e, partendo da quelli più piccoli, li risolviamo e memorizziamo le soluzioni ottenute; le soluzioni di questa categoria solitamente non richiedono la ricorsione.

Dal punto di vista asintotico, le due tecniche sono equivalenti, tuttavia dal punto di vista delle costanti moltiplicative la strategia Top-Down solitamente risulta essere la meno costosa, nonostante possa andare a considerare meno sottoproblemi (laddove invece la Bottom-Up li risolve sempre tutti).

19.1 Problema del taglio delle aste

Vediamo come si produce un algoritmo di programmazione dinamica risolvendo un problema.

Un'azienda produce aste d'acciaio e le vende a pezzi. Le aste prodotte hanno una certa lunghezza n , e sul mercato i pezzi hanno un prezzo che dipende dalla loro lunghezza. Trovare il modo di tagliare le aste che massimizzi il guadagno, assumendo che il costo di taglio sia irrilevante.

Data un'asta di lunghezza n e una tabella di prezzi p_i , con $i = 1, \dots, m$ e $m \geq n$, vogliamo determinare:

- In primo luogo, r_n , cioè il ricavo massimo che si può ottenere tagliando un'asta di lunghezza n e vendendone i pezzi.
- In seconda istanza, le posizioni dove effettuare i tagli.

Lunghezza i	1	2	3	4	5	6	7	...
Prezzo p_i	1	5	8	9	10	17	17	...

Supponiamo di voler effettuare dei tagli in diverse posizioni dell'asta.

- $1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$
- $2 + 2 + 2 + 1 = 16$
- $2 + 2 + 3 = 18$

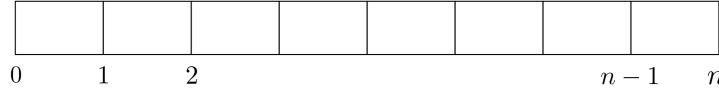
Come possiamo notare, la soluzione che ci permette di ottenere il ricavo massimo prevede di dividere l'asta in due aste più piccole di lunghezza 2 e in una di lunghezza 3.

Ma la soluzione non è unica: possiamo ottenere un ricavo di 18 anche cambiando l'ordine dei tagli della sbarra, oppure effettuando addirittura tagli in posizioni diverse:

- $3 + 2 + 2 = 18$
- $2 + 3 + 2 = 18$
- $1 + 6 = 18$
- $6 + 1 = 18$

19.1.1 Approccio *Divide et impera*

In quanti modi diversi possiamo tagliare un'asta di lunghezza n ?



Per ogni posizione i , $1 \leq i \leq n - 1$, possiamo decidere di effettuare o meno un taglio: questo significa che, in totale, possiamo effettuare $2 \cdot 2 \cdots 2 = 2^{n-1}$ tagli. Analizzare esplicitamente tutti i possibili modi di taglio non è efficiente: se è possibile fare 2^{n-1} tagli diversi, vuol dire che la complessità di tale analisi è dell'ordine di $\Theta(2^n)$.

Esprimiamo il ricavo massimo r_n per un'asta di lunghezza n in modo ricorsivo nel seguente modo:

$$\begin{cases} r_0 = 0 \\ r_n = \max\{ \underbrace{p_n}_{\substack{\text{prezzo} \\ \text{senza} \\ \text{tagliare} \\ \text{l'asta}}}, \underbrace{r_1 + r_{n-1}}_{\substack{\text{taglio in } i=1}}, \underbrace{r_2 + r_{n-2}}_{\substack{\text{taglio in } i=2}}, \dots, \underbrace{r_{n-1} + r_1}_{\substack{\text{taglio in } i=n-1}} \} \end{cases}$$

Quando la soluzione ottima, in questo caso il ricavo massimo r_n , è esprimibile come combinazione di soluzioni ottime di sottoproblemi, si dice che vale la **proprietà della sottostruttura ottima**. Tale proprietà è la condizione esistenziale per cui è possibile elaborare algoritmi di programmazione dinamica.

Implementare la formulazione ricorsiva di cui sopra in codice eseguibile da un calcolatore può risultare complicato; possiamo trovare una caratterizzazione più semplice per risolvere il problema, tagliando un pezzo in modo definitivo (quello sinistro) e suddividendo ulteriormente la parte che resta (quella destra). Formalmente, scriveremo:

$$\begin{cases} r_0 = 0 \\ r_n = \max\{ \underbrace{p_i}_{\substack{\text{non} \\ \text{ulteriormente} \\ \text{diviso}}} + \underbrace{r_{n-i}}_{\substack{\text{suddiviso in modo ottimo}}}, \quad 1 \leq i \leq n \end{cases}$$

Alla luce di tale caratterizzazione, siamo in grado di elaborare una soluzione al problema proposto. L'algoritmo prende in input i seguenti parametri:

- $p[1, \dots, m]$, con $m \geq n$, è il vettore contenente i prezzi delle aste: in $p[i]$ è contenuto il valore di un'asta di lunghezza i , $i \geq 0$.
- n è la lunghezza dell'asta da tagliare.

L'algoritmo produce in output r_n , un intero che rappresenta il massimo guadagno ricavabile dall'asta di lunghezza n .

```

1 Cut_Rod(p, n)
2     if n == 0
3         return 0
4     else
5         q = -1
6         for i = 1 to n
7             q = max(q, p[i] + Cut_Rod(p, n - i))
8         return q

```

Note

- Abbiamo inizializzato q a riga 5 con -1 perché sappiamo che le aste non possono avere prezzi negativi. Se potessero assumere valori negativi, allora andrebbe inizializzato con $-\infty$.

Analisi della complessità Sia $T(n)$ il numero di chiamate di `Cut_Rod` quando la chiamata viene fatta con il secondo parametro uguale a n . Allora,

$$T(n) = \begin{cases} \overbrace{1}^{\text{chiamata iniziale}} & n = 0 \\ 1 + \sum_{i=1}^n T(n-1) & n > 0 \end{cases}$$

Poniamo $j = n - i$ e otteniamo:

$$T(n) = 1 + \sum_{i=1}^n T(n-1) = 1 + \sum_{j=0}^{n-1} T(j)$$

Dimostriamo per induzione su n che $T(n) = 2^n$.

Caso base Se $n = 0$, $T(0) = 2^0 = 1$, che è vero per definizione.

Passo induttivo Assumiamo l'ipotesi induttiva che $T(n) = 2^n$ e lo dimostriamo per $n + 1$.

$$T(n+1) \stackrel{\text{Per def.}}{=} 1 + \sum_{j=0}^{n+1-1} T(j) = 1 + \underbrace{\sum_{j=0}^{n-1} T(j)}_{= T(n)} + T(n) = T(n) + T(n) = 2T(n)$$

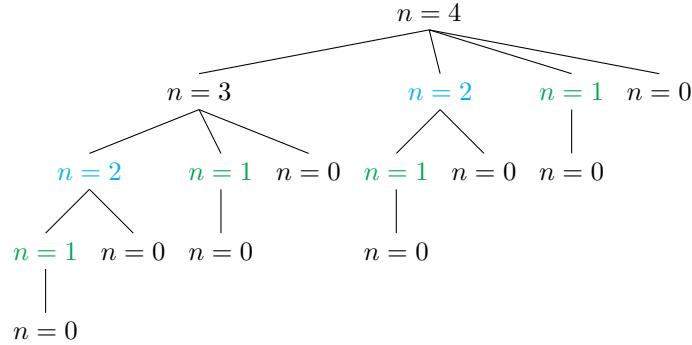
Per definizione

Concludiamo attraverso l'ipotesi induttiva:

$$T(n+1) = 2T(n) = 2 \cdot 2^n = 2^{n+1}, \quad \text{c.v.d.}$$

La complessità di `Cut_Rod` è quindi esponenziale, $T(n) = O(2^n)$.

Lo possiamo verificare anche attraverso l'albero di ricorsione:



Il numero di nodi è esattamente pari a 2^n , con n iniziale uguale a 4.

Notiamo che ogni cammino dalla radice ad una foglia ci dà una permutazione dei possibili tagli effettuabili sull'asta: il cammino più lungo, quello più a sinistra, indica un taglio compiuto, ad ogni chiamata di `Cut_Rod`, in posizione 1 della sottosbarra destra.

19.1.2 Approccio di programmazione dinamica

Se i sottoproblemi distinti sono in numero polinomiale e ciascuno si risolve in tempo polinomiale (data la soluzione dei sottoproblemi), allora memorizzando le soluzioni ed evitando di ricalcolarle si ottiene un algoritmo polinomiale.

Ricordiamo tuttavia che per una soluzioni di questo tipo occorre scendere ad un compromesso tra spazio e tempo: avremo bisogno di una memoria ausiliaria per memorizzare le soluzioni dei sottoproblemi.

Soluzione Top-Down

```

1 Memoized_Cut_Rod(p, n)
2     Sia r[0, ..., n] un nuovo vettore
3     for i = 0 to n
4         r[i] = -1
5     return Memoized_Cut_Rod_Aux(p, n, r)
6
7 Memoized_Cut_Rod_Aux(p, j, r)
8     if r[j] < 0
9         if j == 0
10            r[j] = 0
11        else
12            q = -1
13            for i = 1 to j
14                q = max(q, p[i] + Memoized_Cut_Rod_Aux(p, j - i, r))
15            r[j] = q
16    return r[j]

```

Spiegazione L'algoritmo inizializza un vettore ausiliario r di n elementi dove vengono memorizzati i ricavi massimi per ogni posizione del vettore ($r[i]$ contiene il ricavo ottimo per una lunghezza i). Questo vettore, con gli elementi inizialmente posti a -1 , viene passato ad una funzione ausiliaria, che man mano che risolve i sottoproblemi aggiorna r con i risultati parziali.

L'inizializzazione ad un numero negativo ci serve per capire subito se è già stato calcolato il ricavo ottimo per quella lunghezza (se il valore è negativo non è stata ancora calcolato); questo controllo viene fatto nelle prime righe della funzione ausiliaria, e procede a ritornare direttamente il ricavo se presente.

Altrimenti, viene compiuto un algoritmo simile a quello dell'approccio *Divide et impera*, dove viene impiegata la ricorsione per calcolare il ricavo ottimo per una data lunghezza.

Va osservato che si può eliminare il controllo delle righe 9 e 10 inizializzando, nella funzione principale, gli elementi di r a 0 piuttosto che a -1 , ma si tratta di scelte di implementazione: l'algoritmo è corretto in entrambe le sue varianti.

Analisi della complessità Una chiamata ricorsiva per risolvere un problema precedentemente risolto termina immediatamente, dunque si giunge al ramo `else` a riga 5 nella funzione ausiliaria una sola volta per ciascun sottoproblema, presenti in numeri $j = 1, \dots, n$.

Per risolvere un sottoproblema di dimensione j , il ciclo `for` effettua j iterazioni.

Quindi il numero totale di iterazioni di questo ciclo **for**, per tutte le chiamate ricorsive della funzione ausiliaria, è:

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

Considerando anche il fatto che la funzione principale compie n iterazioni per inizializzare **r**, diremo che il costo totale dell'algoritmo è dell'ordine di $\Theta(n) + \Theta(n^2) = \Theta(n^2)$ nel caso peggiore.

Soluzione Bottom-Up

```

1 Bottom_Up_Cut_Rod(p, n)
2     Sia r[0, ..., n] un nuovo vettore
3     r[0] = 0
4     for j = 1 to n
5         q = -1
6         for i = 1 to j
7             q = max(q, p[i] + r[j - i])
8         r[j] = q
9     return r[n]

```

Analisi della complessità Il ciclo esterno compie esattamente n iterazioni, mentre il ciclo più interno compie ogni volta j iterazioni, dove j assume tutti i valori da 1 a n . Di conseguenza,

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

Posizioni dei tagli Andiamo ora a costruire una funzione che ci dice dove posizionare i tagli per ottenere il ricavo massimo. Per questo obiettivo dovremo usare un'ulteriore struttura dove memorizzare tale risultato.

Abbiamo già usato il vettore **r**[0, ..., n], il cui i -esimo elemento memorizza il ricavo massimo per il problema di dimensione i , mentre ora dovremo servirci anche del vettore **s**[1, ..., n], la cui posizione i -esima memorizza la posizione del primo taglio che determina la soluzione ottima per il problema di dimensione i .

```

1 Ext_Bottom_Up_Cut_Rod(p, n)
2     Siano r[0, ..., n] e s[1, ..., n] due nuovi vettori
3     r[0] = 0
4     for j = 1 to n
5         q = -1
6         for i = 1 to j
7             if q < p[i] + r[j - i]
8                 q = p[i] + r[j - i]
9                 s[j] = i
10            r[j] = q
11    return r, s

```

La complessità è la medesima dell'algoritmo precedente, $T(n) = \Theta(n^2)$.

Per visualizzare la corretta sequenza dei tagli, è necessario definire un'ulteriore funzione.

```

1 Print_Cut_Rod_Solution(p, n)
2     r, s = Ext_Bottom_Up_Cut_Rod(p, n)
3         while n > 0
4             print(s[n])
5             n = n - s[n]

```

Spiegazione L'algoritmo stampa il taglio ottimo per un'asta di lunghezza n . Una volta effettuato il taglio, cioè una volta ottenuto il valore n -esimo di s , quello che ci resta è la lunghezza della barra a destra, che noi vogliamo continuare a suddividere, e che è pari proprio alla lunghezza iniziale meno il valore contenuto in $s[n]$.

Cerchiamo di visualizzare la soluzione. Il seguente è il contenuto di s dopo l'esecuzione di `Ext_Bottom_Up_Cut_Rod`:

$s[i]$	1	2	3	2	2	6	1
i	1	2	3	4	5	6	7

Supponiamo di avere l'asta dell'esempio iniziale, di lunghezza $n = 7$. $s[7]$ ci ritorna 1, per cui sappiamo che il primo taglio verrà eseguito nella posizione 1; la lunghezza che ci resta è pari a $7 - 1 = 6$.

$s[6]$ ci ritorna 6, che significa che dobbiamo conservare l'asta nella sua lunghezza attuale; dopo questo taglio la lunghezza è arrivata a 0: non abbiamo più parti di asta da tagliare.

Ricordiamoci ora che tra le diverse possibili soluzioni dell'esempio in questione c'era proprio la sequenza di taglio [1, 6], che ci ritornava un ricavo di 18.

Analisi della complessità La funzione è costituita da un ciclo `while` che compie al più n iterazioni, dunque il suo tempo d'esecuzione è $O(n)$. Sommando a questa complessità quella del resto dell'algoritmo (dobbiamo infatti ottenere il vettore dei ricavi s per poterlo leggere), il tempo d'esecuzione totale è $\Theta(n^2)$.

19.2 Longest Common Subsequence

Continuiamo lo studio della programmazione dinamica analizzando un famoso problema del campo dell'informatica: trovare la *Longest Common Subsequence*, o Massima Sottosequenza Comune, dei caratteri presenti in due stringhe.

Si tratta di una questione che trova diverse applicazioni pratiche, prima fra tutte nella bioinformatica, dove viene implementata per lo studio di sequenze di filamenti di DNA: il DNA infatti, essendo composto da una sequenza in cui si alternano le quattro basi azotate (adenina, guanina, citosina e timina), può essere rappresentato come una stringa in cui l'alfabeto dei caratteri è costituito dalle iniziali delle basi azotate: {A, G, C, T}.

Edit Distance Prende il nome di *Edit Distance* il minimo numero di modifiche da apportare ad una stringa per renderla uguale ad un'altra stringa.

Date le stringhe $S_1 = \text{"ACTACCTG"}$ e $S_2 = \text{"ATCACC"}$, calcoliamo la loro *Edit Distance*.

1. Inseriamo una "C" tra la prima "A" e la "T" in S_2 .
2. Eliminiamo la prima "C" dopo la "T" in S_2 .
3. Inseriamo una "T" in coda a S_2 .
4. Inseriamo una "G" in coda a S_2 .

Le due stringhe ora sono uguali, dunque hanno una *Edit Distance* di 4.

Il problema Prendiamo le stesse stringhe del paragrafo precedente, S_1 e S_2 , e cerchiamo la loro massima sottosequenza comune.

Possiamo indicare una stringa come una sequenza di n caratteri $X = x_1x_2 \dots x_n$. Di conseguenza, una sottosequenza di X è un'altra sequenza $X_k = x_{i_1}, \dots, x_{i_k}$ tale che $i_1, \dots, i_k \in \{1, \dots, n\}$ e $i_1 < i_2 < \dots < i_k$.

Per definizione, una sottosequenza può non essere lineare, cioè può essere costituita da caratteri non adiacenti nella stringa originaria, purché la sottosequenza mantenga l'ordine dei caratteri.

Il problema della *Longest Common Subsequence*, abbreviato in *LCS*, si pone il seguente problema: date due sequenze $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$, vogliamo trovare una sequenza W tale che sia sottosequenza di X e Y e che sia di lunghezza massima. Ad occhio, possiamo notare che le stringhe S_1 e S_2 presentano non una, bensì due sottosequenze comuni di lunghezza massima:

$$\text{LCS}(S_1, S_2) = \{\text{ATACC, ACACC}\}$$

Infatti, la massima sottosequenza comune solitamente non è unica, per cui è meglio definire *LCS* come un insieme che come un unico elemento.

L'algoritmo risolutivo Il nostro esempio con le stringhe S_1 e S_2 rappresenta un caso estremamente semplice del problema della *LCS*, tant'è che siamo giunti alle soluzioni, appunto, ad occhio.

Di primo impatto, può venire spontaneo chiedersi se possa essere efficiente un algoritmo di *brute force* che trovi tutte le sottostringhe di X , verifichi che siano anche sottosequenze di Y e di queste conservi le più lunghe. Per renderci conto che questa non può essere una soluzione efficace, basta considerare tutte le possibili sottosequenze di $X = x_1x_2\dots x_m$: possiamo compiere un totale di due scelte su ciascun carattere $x_i \in X$, $i \in [1, m]$ (se tenerlo nella sottosequenza o meno), per cui avremo un totale di 2^m possibili scelte e quindi anche un totale di 2^m possibili sottosequenze.

Verificato che l'algoritmo di *brute force* ha costo esponenziale, cerchiamo una caratterizzazione del problema attraverso la programmazione dinamica.

Passo 1. Definizione della sottostruttura ottima

Il problema della *LCS* di X e Y può essere ridotto allo stesso problema sui prefissi di X e Y .

Data una sequenza $X = x_1 \dots x_m$, per un certo $k \leq m$, il **prefisso** di lunghezza k di X è $X^k = x_1 \dots x_k$. Per esempio, se $X = ACG$, vale che $X^0 = \varepsilon$ (la stringa vuota), $X^1 = A$, $X^2 = AC$ e $X^3 = ACG$. In generale, una sequenza di n caratteri ha un totale di $n + 1$ prefissi, contando anche la stringa vuota.

Teorema - Sottostruttura ottima per *LCS*

Siano $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$ due sequenze, e sia $W = w_1 \dots w_k \in LCS(X, Y)$. Allora,

- 1) Se $x_m = y_n$, allora $w_k = x_m = y_n$ e $W^{k-1} \in LCS(X^{m-1}, Y^{n-1})$.
- 2) Se $x_m \neq y_n$,
 - 2.1) Se $w_k \neq x_m$, allora $W \in LCS(X^{m-1}, Y)$.
 - 2.2) Se $w_k \neq y_n$, allora $W \in LCS(X, Y^{n-1})$.

L'aver ridotto il problema della *LCS* al problema sui prefissi fa sì che il numero di sottoproblemi distinti sia dell'ordine di $O(n \cdot m)$.

Passo 2. Soluzione ricorsiva per il valore della soluzione ottima

Dati $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$, indichiamo con $c[i, j]$ la lunghezza delle sottosequenze appartenenti a $LCS(X^i, Y^j)$, con $0 \leq i \leq m$ e $0 \leq j \leq n$.

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i - 1, j - 1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max \{c[i - 1, j], c[i, j - 1]\} & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Passo 3. Algoritmo Top Down o Bottom Up

Per questo algoritmo, facciamo uso di due strutture ausiliarie:

- La matrice c , avente $m + 1$ righe e $n + 1$ colonne, che in posizione $c[i, j]$ memorizza la lunghezza delle sottosequenze appartenenti a $LCS(X^i, Y^j)$.
- La matrice b , avente m righe e n colonne, che in posizione $b[i, j]$ contiene informazioni utili a recuperare la soluzione; in particolare,
 - $b[i, j] = \nwarrow$ se $x_i = y_j \Rightarrow LCS(X^i, Y^j)$ viene ridotta a $LCS(X^{i-1}, Y^{j-1})$.
 - $b[i, j] = \uparrow$ se $x_i \neq y_j \Rightarrow LCS(X^i, Y^j)$ viene ridotta a $LCS(X^{i-1}, Y^j)$.
 - $b[i, j] = \leftarrow$ se $x_i \neq y_j \Rightarrow LCS(X^i, Y^j)$ viene ridotta a $LCS(X^i, Y^{j-1})$.

Algoritmo Bottom Up

```

1  LCS(X, Y)
2      m = X.length
3      n = Y.length
4      for i = 0 to m
5          c[i, 0] = 0
6      for j = 1 to n
7          c[0, j] = 0
8      for i = 1 to m
9          for j = 1 to n
10             if x_i == y_j
11                 c[i, j] = c[i - 1, j - 1] + 1
12                 b[i, j] = \nwarrow
13             else
14                 if c[i - 1, j] ≥ c[i, j - 1]
15                     c[i, j] = c[i - 1, j]
16                     b[i, j] = \uparrow
17                 else
18                     c[i, j] = c[i, j - 1]
19                     b[i, j] = \leftarrow
20     return b, c

```

Analisi della complessità Il tempo d'esecuzione, in funzione di m e n , è dato dai due cicli che compiono rispettivamente m e n iterazioni e dai due cicli innestati che compiono un totale di $m \cdot n$ iterazioni.

$$T(m, n) = \Theta(m) + \Theta(n) + \Theta(m \cdot n) = \Theta(m \cdot n)$$

Algoritmo in azione Simuliamo l'esecuzione dell'algoritmo con le seguenti stringhe: $X = AC$, $Y = BA$.

$X \backslash Y$	ε	B	A
X	0	0	0
ε	0	$0 \uparrow$	$1 \nwarrow$
A	0	$0 \uparrow$	$1 \nwarrow$
C	0	$0 \uparrow$	$1 \uparrow$

Innanzitutto, l'algoritmo inizializza la prima riga e la prima colonna di una tabella di $m + 1$ righe e $n + 1$ colonne a 0. Questa riga e questa colonna rappresentano la sottosequenza di lunghezza 0, per cui sappiamo che consistono nel caso base della nostra soluzione.

Successivamente, partendo dalla cella (1, 1) (lettera A della stringa X e lettera B della stringa Y), procede colonna per colonna da sinistra verso destra e ad ogni cella trovata confronta i caratteri delle due stringhe: se uguali, somma 1 al contenuto della cella adiacente in alto a sinistra (perché abbiamo trovato un altro carattere appartenente alla sottosequenza di lunghezza massima, dunque aumentiamo la sua lunghezza di 1), e memorizza il carattere ausiliario \nwarrow ; altrimenti, memorizza il valore massimo tra quello contenuto nella cella adiacente in alto e quello contenuto nella cella adiacente a sinistra, e memorizza rispettivamente \uparrow o \leftarrow .

Passo 4. Costruire una soluzione

Attraverso la procedura `LCS(X, Y)` abbiamo ottenuto la soluzione, ma come facciamo a stamparla a partire dalle tabelle `b` e `c`?

```

1 printLCS(X, Y)
2   b, c = LCS(X, Y)
3   printLCSrec(X, b, X.length, Y.length)
4
5 printLCSrec(X, b, i, j)
6   if i > 0 and j > 0
7     if b[i, j] == ↘
8       printLCSrec(X, b, i - 1, j - 1)
9       print  $X_i$ 
10    else
11      if b[i, j] == ↑
12        printLCSrec(X, b, i - 1, j)
13      else
14        printLCSrec(X, b, i, j - 1)

```

Analizziamo il funzionamento delle due procedure di stampa partendo dal risultato dell'algoritmo precedente.

$X \backslash Y$	ε	B	A
X	0	$3\ 0$	0
ε	0	$0 \uparrow$	$2\ 1 \nwarrow$
A	0	$0 \uparrow$	$1 \nwarrow$
C	0	$0 \uparrow$	$1 \uparrow$

L'algoritmo parte dall'estremo inferiore destro della tabella, e segue la direzione delle frecce per ricavare il carattere da stampare e gli indici della successiva chiamata ricorsiva. In particolare, nell'esempio in questione, alla prima chiamata la funzione parte dalla cella (2, 2): vede che la freccia non è uguale a \nwarrow (ciò significa che i due caratteri sono diversi, dunque non stampa nulla), procede nella direzione della freccia

(2, 2): vede che la freccia non è uguale a \nwarrow (ciò significa che i due caratteri sono diversi, dunque non stampa nulla), procede nella direzione della freccia

contenuta in quella cella (cioè verso l'alto) e chiama sé stessa sulla cella adiacente in alto. A questo punto vede che la freccia punta in alto a sinistra, quindi sa che deve stampare il carattere di X contenuto nella posizione i (notare che l'algoritmo riceve solamente una delle due stringhe: dal momento che la sottosequenza è contenuta in entrambe le stringhe, è sufficiente passargliene una sola), ma non prima di aver chiamato la ricorsione sulla cella adiacente in alto a sinistra: in questo caso, tale cella ha come indice i pari a 0, dunque siamo arrivati al caso base, perciò l'algoritmo è libero di stampare la sottosequenza comune di lunghezza massima, costituita in questo esempio dal carattere A .

Analisi della complessità Il tempo d'esecuzione di `printLCSrec` è $O(i+j)$: ad ogni chiamata ricorsiva vado a decrementare almeno uno dei due parametri i e j , per cui il numero di chiamate complessive che possono essere fatte è $i+j$. Il tempo d'esecuzione di `printLCS` è pari alla somma dei costi di `LCS` e `printLCSrec`:

$$T(n, m) = \Theta(m \cdot n) + \Theta(m + n) = \Theta(m \cdot n)$$

Ottimizzazioni

Possiamo fare delle ottimizzazioni rispetto all'uso della memoria: il tempo d'esecuzione rimane invariato.

1. L'array \mathbf{b} può non essere memorizzato, dato che l'informazione corrispondente è derivabile da \mathbf{c} e in modo locale: $c[i, j]$ dipende solo da tre possibili valori: $c[i - 1, j - 1]$, $c[i - 1, j]$ e $c[i, j - 1]$.

```

1 printLCSrec(X, c, i, j)
2   if i > 0 and j > 0
3     if c[i, j] == c[i - 1, j]
4       printLCSrec(X, c, i - 1, j)
5     else if c[i, j] == c[i, j - 1]
6       printLCSrec(X, c, i, j - 1)
7     else
8       printLCSrec(X, c, i - 1, j - 1)
9       print X_i

```

L'ordine in cui facciamo questi confronti è importante: non possiamo, per esempio, controllare prima che la cella sia uguale a quella adiacente in alto a sinistra (*i.e.*, `if c[i, j] == c[i - 1, j - 1] ...`), perché rischieremmo di ottenere delle soluzioni errate.

Ordine corretto

X	Y	ε	B	A
ε	0	$\textcolor{blue}{3} 0$	0	
A	0	$0 \uparrow$	$\textcolor{blue}{2} 1 \nwarrow$	
C	0	$0 \uparrow$	$\textcolor{blue}{1} 1 \uparrow$	

Ordine sbagliato

X	Y	ε	B	A
ε	0	$\textcolor{blue}{3} 0$	0	
A	0	$2 0 \uparrow$	$1 \nwarrow$	
C	0	$0 \uparrow$	$\textcolor{blue}{1} 1 \uparrow$	

2. Se sono interessato esclusivamente alla lunghezza della LCS si può evitare di mantenere la tabella c perché posso calcolare la riga $i + 1$ utilizzando la riga i . Si può dunque utilizzare, al suo posto, un vettore di $\min(m, n)$ elementi, più uno spazio $O(1)$ aggiuntivo.

Algoritmo Top Down

```

1 tdLCS(X, Y)
2   m = X.length
3   n = Y.length
4   c[0 ... m, 0 ... n] = -1
5   return tdLCSaux(X, Y, c, m, n)
6
7 tdLCSaux(X, Y, c, i, j)
8   if c[i, j] == -1
9     if i == 0 or j == 0
10      c[i, j] = 0
11    else
12      if Xi == Yj
13        c[i, j] = tdLCSaux(X, Y, c, i - 1, j - 1) + 1
14      else
15        c[i, j] = max(tdLCSaux(X, Y, c, i - 1, j),
16                         tdLCSaux(X, Y, c, i, j - 1))
16
17 return c[i, j]

```

Spiegazione e analisi della complessità L'algoritmo crea una tabella di $(m+1) \cdot (n+1)$ elementi inizializzati a -1 , e procede a chiamare sé stessa ogni volta che trova un problema non risolto.

Il principio risolutivo dell'algoritmo è il medesimo della sua variante *Bottom Up*: partendo dalla fine delle due stringhe (cioè dall'estremo inferiore destro della tabella), se i due caratteri correnti sono uguali, riduce il problema di dimensione (i, j) al problema di dimensione $(i - 1, j - 1)$, altrimenti considera il maggiore tra i sottoproblemi di dimensione $(i - 1, j)$ e $(i, j - 1)$.

Ogni problema viene risolto una volta sola, dunque il numero di chiamate complessivo fatte da `tdLCS` è $m \cdot n$ nel caso peggiore. L'inizializzazione di c costa inoltre $\Theta(m \cdot n)$, dunque $T(n, m) = \Theta(m \cdot n)$.

Numero di sottoproblemi risolti

X \ Y	ε	C	A	N	E
ε	✓				
C		4			
A			3		
N				2	
E					1

In casi come questo, gli algoritmi *Bottom Up* calcolano sempre tutti i sottoproblemi, mentre invece gli algoritmo *Top Down* ne calcolano molti meno: questi ultimi, infatti, non devono per forza risolvere tutti i sottoproblemi, bensì solo quelli strettamente necessari.

Nel problema della *LCS*, l'approccio *Top Down* è tendenzialmente più efficiente.

19.3 Elementi di Programmazione Dinamica

1. A quali problemi si può applicare?

La programmazione dinamica si applica ai problemi di ottimizzazioni, cioè problemi in cui ho un insieme molto grande di soluzioni, e voglio determinare quella ottima.

2. Occorre che questi problemi abbiano due caratteristiche:

a) Il problema deve disporre di una **sottostruttura ottima**: una soluzione ottima di un problema è una combinazione di soluzioni ottime di sottoproblemi.

b) **Ripetizioni di sottoproblemi**: il numero di sottoproblemi distinti è piccolo rispetto al numero di possibili soluzioni.

Nel caso delle aste, il numero di soluzioni è $\Omega(2^n)$, mentre il numero di sottoproblemi è $\Theta(n)$.

Nel caso della *LCS*, il numero di soluzioni è $\Omega(2^n)$, mentre il numero di sottoproblemi è $\Theta(m \cdot n)$.

3. Se valgono le prime due condizioni, ci sono due possibili approcci che possiamo intraprendere:

a) Procedura di tipo *Bottom Up*: elenca i sottoproblemi in modo ordinato, e li risolve dal più piccolo fino a quello desiderato.

b) Procedura di tipo *Top Down* con *Memoization*: si fornisce una soluzione ricorsiva del problema in termini dei sottoproblemi, memorizzando le soluzioni dei problemi già risolti.

19.3.1 Confronti

Se per il calcolo della soluzione globale occorre risolvere tutti i sottoproblemi, conviene usare l'approccio *Bottom Up* perché non è ricorsiva (non c'è il costo delle chiamate ricorsive) ed evita alcuni controlli, *e.g.* se è già stato calcolato oppure no il valore del sottoproblema corrente; se invece c'è solamente bisogno di alcuni sottoproblemi, allora la soluzione *Top Down* risulta essere migliore perché evita il calcolo di sottoproblemi inutili.

Suggerimento della prof.ssa Raffaetà Se non viene specificato e se si riesce a dare un'espressione ricorsiva del problema, conviene usare l'approccio *Top Down* in quanto più "fedele" alla rappresentazione ricorsiva, mentre quella *Bottom Up* solitamente richiede un'ulteriore elaborazione.

20 Grafi

20.1 Introduzione

Un **grafo** è una rappresentazione del concetto matematico di relazione.

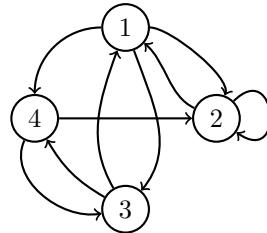
Un grafo è costituito da un insieme di **vertici** (o nodi) e da un insieme di **archi** che collegano questi nodi.

Esistono molteplici parametri secondo i quali possiamo catalogare i diversi tipi di grafi, ma la distinzione più importante che dobbiamo fare è sicuramente quella tra **grafi orientati** e **grafi non orientati**.

- **Grafi orientati**

Un grafo G è una coppia $G = (V, E)$, dove $V = \{1, 2, \dots, n\}$ è l'insieme di vertici del grafo e $E \subseteq V \times V$ è un insieme di coppie ordinate di nodi, che prendono il nome di archi (*edges* in inglese, da cui E). Nei grafi orientati, gli archi hanno un verso e le coppie sono orientate.

Per esempio, dati $V = \{1, 2, 3, 4\}$ e $E = \{(1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (3, 1), (3, 4), (4, 2), (4, 3)\}$, possiamo rappresentare il grafo in questo modo:

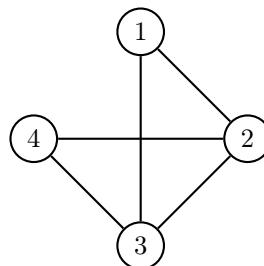


- **Grafi non orientati**

Un grafo non orientato G è una coppia $G = (V, E)$, dove V è l'insieme di vertici, $E \subseteq V \times V$ è l'insieme di archi, su cui valgono le seguenti proprietà:

- i. **Simmetria:** $(i, j) \in E \Leftrightarrow (j, i) \in E$
- ii. **Non riflessività:** $\forall i \in V, (i, i) \notin E$
È infatti possibile che esistano archi che vanno da un nodo a sé stesso: questi archi prendono il nome di **cappi** o **loophole**.

In questo tipo di grafi, gli archi si disegnano senza freccia perché la relazione tra i nodi che la compongono è sempre simmetrica.



Grafi non orientati - Definizione alternativa

Alcuni libri di testo adottano una definizione alternativa, tuttavia equivalente, per i grafi non orientati. Secondo tali testi, un grafo $G = (V, E)$ è costituito dall'insieme di vertici V e dall'insieme di archi E , dove $E \subseteq \binom{V}{2}$.

Secondo questa definizione, sembra che E sia il risultato di un coefficiente binomiale, però non è così in quanto V rappresenta un insieme, non un numero: $\binom{V}{2}$ rappresenta infatti tutti i possibili sottoinsiemi di V costituiti da due elementi. Sia $V = \{1, 2, 3\}$. Allora, $\binom{V}{2} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$, dove ogni coppia di elementi è compresa tra due parentesi graffe invece che tra due parentesi tonde per sottolineare il fatto che l'ordine di questi raggruppamenti non ha importanza. Il motivo di questa rappresentazione si deve alla cardinalità di questi sottoinsiemi:

$$\left| \binom{V}{2} \right| = \binom{|V|}{2} = \frac{|V|!}{2!(|V|-2)!}$$

Analogamente, rappresentiamo in questo modo l'insieme delle coppie ordinate: $V \times V = V^2 \Rightarrow |V^2| = |V|^2$.

20.1.1 Terminologia

- **Cardinalità di V e E**

Indicheremo con n il numero di vertici del grafo, e con m il numero dei suoi archi: $n = |V|$, $m = |E|$.

- **Adiacenza**

In un grafo non orientato, i nodi i e j sono adiacenti se esiste un arco che li collega.

In un grafo orientato, il vertice j è adiacente al vertice i se esiste un arco $(i, j) \in E$.

Sono di seguito rappresentati, rispettivamente, un grafo non orientato e un grafo orientato che rispettano le condizioni di cui sopra.



- **Incidenza**

In un grafo orientato o in un grafo non orientato, un arco che collega i nodi i e j è detto incidente a i e a j .

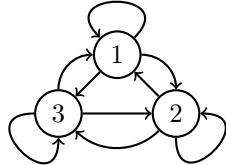
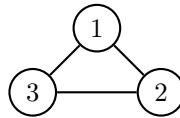
- **Densità**

La densità di un grafo è data dal rapporto tra il numero di archi e il numero totale dei suoi possibili archi.

$$0 \leq \delta(G) = \frac{|E|}{\text{numero totale di possibili archi}} \leq 1$$

Vediamo qual è il numero massimo di archi nei due tipi di grafi che abbiamo visto.

Nel caso di un grafo non orientato, il numero totale di possibili archi è dato da $\binom{n}{2} = \frac{n(n-1)}{2}$.



Nel caso di un grafo orientato, invece, il numero totale di possibili archi è dato da n^2 .

Quindi, vale che in un grafo non orientato $\delta(G) = \frac{m}{\binom{n}{2}} = \frac{2m}{n(n-1)}$, mentre in un grafo orientato $\delta(G) = \frac{m}{n^2}$.

Se $\delta(G) = 0$ si dice che il grafo è **vuoto**, cioè non ha archi; un grafo con n vertici e $\delta(G) = 0$ si rappresenta con E_n , dove la E sta per *empty*.

Se $\delta(G) = 1$ si dice che il grafo è **completo**, cioè sono presenti in esso tutti i possibili archi; un grafo con n vertici e $\delta(G) = 1$ si rappresenta con K_n .

Quanto più $\delta(G)$ tende a 0, più il grafo è **sparso**, e si verifica che $m \approx n$, da intendere che m e n sono dello stesso ordine di grandezza.

Quanto più invece $\delta(G)$ tende a 1, più il grafo è **denso**, e si verifica che $m \approx n^2$, da intendere che m e n^2 sono dello stesso ordine di grandezza.

• Peso

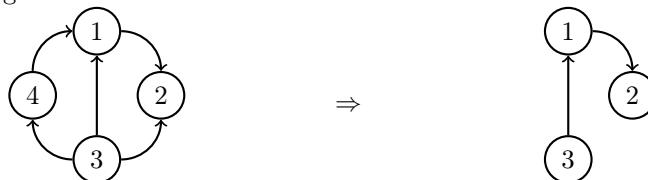
Un grafo si dice **pesato** rispetto ai suoi vertici e/o ai suoi archi quando $G = (V, E, w)$, dove w è detta **funzione peso** ed è una funzione che associa ai vertici e/o agli archi del grafo un valore reale.

Se $w : E \rightarrow \mathbb{R}$, la funzione peso si riferisce agli archi. Se $w : V \rightarrow \mathbb{R}$, la funzione peso si riferisce ai vertici.

È anche possibile creare un grafo pesato sia rispetto ai vertici che agli archi, definendolo nel seguente modo: $G = (V, E, w_1, w_2)$, dove $w_1 : E \rightarrow \mathbb{R}$ e $w_2 : V \rightarrow \mathbb{R}$.

• Sottografo

Sia $G = (V, E)$ un grafo. Allora, un suo **sottografo** è $G' = (V', E')$, con $V' \subseteq V$ e $E' \subseteq E \cap V' \times V'$. Vediamo un esempio di un grafo e di un suo sottografo.

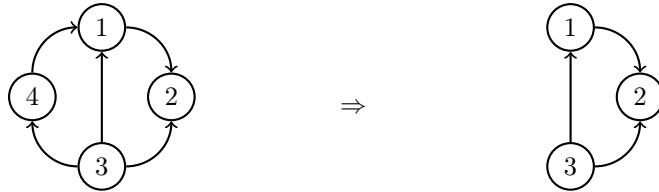


Sottografo del grafo a sinistra è anche un grafo senza archi e costituito dai soli vertici 1, 2 e 3, perché l'insieme vuoto è sempre sottoinsieme di E , ma non lo è un grafo con gli stessi vertici ma avente un arco come, *e.g.*, $(2, 3)$, in quanto $(2, 3) \notin E$.

- **Sottografo indotto**

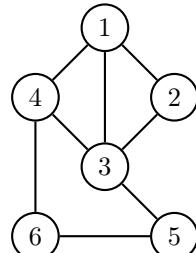
Un **sottografo indotto** è un particolare sottografo che ha la seguente caratteristica: sia $G = (V, E)$ un grafo, e $G[V'] = (V', E')$ il suo sottografo indotto; allora, $E' = E \cap V' \times V'$. Significa che, dato un sottoinsieme V' dei vertici di G , il sottoalbero indotto $G[V']$ contiene tutti gli archi incidenti ai nodi di V' .

Dall'esempio di prima, sia $V' = \{1, 2, 3\}$. Allora, G e $G[V']$ sono rappresentabili graficamente come segue.



- **Cammino**

Un **cammino** tra due vertici u e v è una sequenza di vertici $\langle x_0, x_1, \dots, x_q \rangle$ tali che $x_0 = u$, $x_q = v$ e $(x_i, x_{i+1}) \in E \forall i = 0, \dots, q - 1$.



Un cammino che va, per esempio, dal nodo 2 al nodo 6 nel grafo in figura è il seguente:

$$\langle 2, 1, 4, 3, 5, 6 \rangle$$

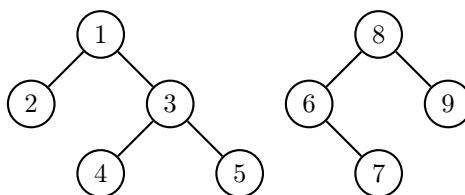
La lunghezza di un cammino è pari al numero di archi traversati dal cammino.

Un **cammino semplice** è un cammino in cui tutti i vertici sono distinti; per esempio, il cammino $\langle 2, 1, 4, 3, 5, 6 \rangle$ è un cammino semplice, mentre $\langle 2, 1, 3, 2, 3, 5, 6 \rangle$ non lo è.

Inoltre, un **ciclo** è un cammino in cui $x_0 = x_q$. Un grafo viene detto **aciclico** se non contiene cicli.

- **Connessione**

Un grafo si dice **connesso** se, presi due vertici nel grafo, esiste almeno un cammino che li collega. Analogamente, un grafo **disconnesso** presenta almeno una coppia di vertici non raggiungibili da un cammino.



Un grafo aciclico e connesso prende il nome di **albero (libero)**.

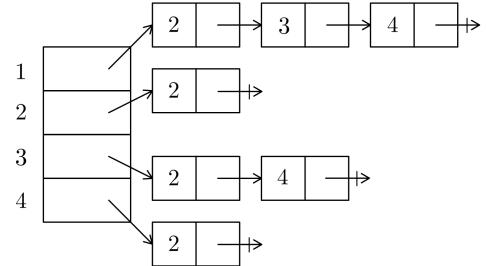
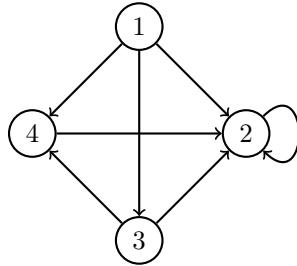
20.1.2 Rappresentazione di grafi

È possibile rappresentare algoritmamente i grafi attraverso tre diverse modalità: tramite **liste di adiacenza**, tramite una **matrice di adiacenza** e tramite una **matrice di incidenza**.

- **Liste di adiacenza**

Questa implementazione richiede l'uso di un array di dimensione n , contenente, in ciascuna posizione, un puntatore ad una lista concatenata. La posizione dell'array indica il nodo da cui parte un arco, e ciascuna cella della lista concatenata contiene l'informazione su un nodo a cui arriva l'arco partito da quella posizione.

È riportato di seguito un grafo e la sua rappresentazione mediante liste di adiacenza.



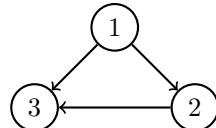
Questo tipo di rappresentazione vanta un'occupazione di memoria che cresce linearmente rispetto a n , a scapito dei tempi di accesso: per verificare l'esistenza di un arco, è necessario scorrere le liste concatenate, che sappiamo essere un'operazione potenzialmente lenta. Per questo motivo, la rappresentazione attraverso liste di adiacenza è più utile nel caso di grafi sparsi.

- **Matrice di adiacenza**

La matrice di adiacenza A_G o A è una matrice di dimensione $n \times n$ i cui elementi assumono i seguenti valori:

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{se } (i, j) \notin E \end{cases}, \quad 1 \leq i, j \leq n$$

Di seguito viene riportato un esempio di grafo orientato con relativa matrice di adiacenza.



$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Il vantaggio di questa rappresentazione è sicuramente il tempo d'accesso costante alla matrice per la verifica dell'adiacenza dei nodi, tuttavia uno svantaggio non irrilevante è la crescita quadratica dell'utilizzo della memoria rispetto a n . Per questo motivo, la rappresentazione attraverso matrici di adiacenza è utile nel caso di grafi densi.

Vediamo ora invece un altro esempio di questa rappresentazione per quanto riguarda i grafi non orientati.

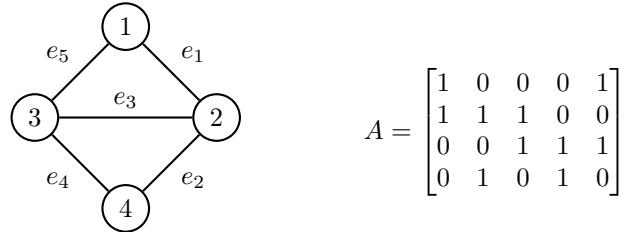


Si può notare che la matrice di adiacenza di un grafo non orientato è simmetrica: $A^T = A$.

• Matrice di incidenza

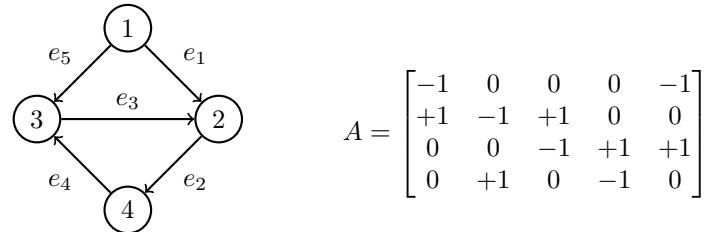
La matrice di incidenza di un grafo è una matrice di dimensione $n \times m$ in cui le righe rappresentano i vertici e le colonne rappresentano gli archi, che ci permette di capire quali vertici sono collegati da un preciso arco (e qual è il vertice di arrivo e di partenza nel caso di grafi orientati).

Vediamo prima un esempio di rappresentazione di un grafo non orientato.



Come possiamo notare, nella prima colonna, che rappresenta l'arco e_1 , sono presenti degli 1 nelle righe 1 e 2, in quanto l'arco e_1 è esattamente quello che collega i vertici 1 e 2, e degli 0 in tutte le altre righe. In modo analogo, ciascuna colonna presenta degli 1 in corrispondenza delle righe relative ai vertici collegati dall'arco della corrente colonna.

Il discorso cambia leggermente per grafi orientati: in queste matrici, viene posto -1 in corrispondenza della riga che rappresenta il vertice di partenza, e viene posto +1 in corrispondenza della riga che rappresenta il vertice di arrivo. Vediamo un esempio.

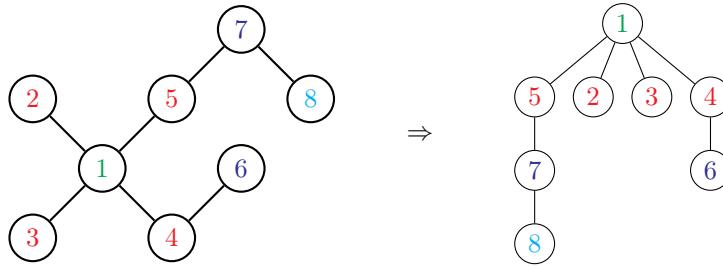


20.1.3 Ulteriore terminologia e approfondimenti

- **Approfondimento sugli alberi**

Abbiamo visto nel primo capitolo sulla terminologia che un grafo aciclico e connesso prende il nome di albero. Nella teoria dei grafi, l'albero è una struttura piatta, non c'è una gerarchia come l'abbiamo vista negli alberi studiati precedentemente (detti, appunto, **alberi radicati**).

Esiste tuttavia una relazione tra i due: nel momento in cui all'interno di un albero libero identifichiamo un vertice particolare, possiamo indurre in esso una gerarchia, ponendolo cioè il nodo radice di un albero radicato.

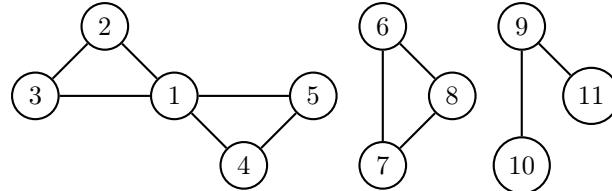


- **Componente连通**

Dato un grafo $G = (V, E)$, una componente connessa di G è un sottoinsieme di vertici V' tale che:

1. Il sottografo indotto da V' è connesso.
2. V' non è contenuto strettamente in un sottoinsieme V'' , con $G[V'']$ connesso.

In altre parole, non è possibile aggiungere un vertice esterno ad una componente connessa e ottenere un grafo connesso.



Per esempio, il grafo di cui sopra è costituito da tre componenti connesse. Non è possibile aggiungere a quella a sinistra, per esempio, il vertice 6, perché non otterremmo più un sottografo connesso. Le tre componenti costituiscono dei sottografi indotti sull'insieme di vertici che le costituiscono: $V_1 = \{1, 2, 3, 4, 5\}$, $V_2 = \{6, 7, 8\}$ e $V_3 = \{9, 10, 11\}$.

Lemma Sia G un grafo, e siano V_1, V_2, \dots, V_k delle componenti connesse. Allora, V_1, V_2, \dots, V_k sono un partizionamento di V .

Dimostrazione Il lemma è vero se riusciamo a dimostrare le seguenti proprietà.

1. $V_i \cap V_j \neq \emptyset \forall i, j \in [1, k], i \neq j$

Avviene per assurdo. Supponiamo che esista un vertice $x \in V_i, V_j$ per un certo i e un certo j , $i \neq j$. Questo è assurdo, perché per definizione di componente连通的 non è possibile aggiungere un vertice esterno ad una componente连通的 e ottenere un grafo连通的, verrebbe a mancare la seconda proprietà delle componenti connesse.

2. $V_1 \cup V_2 \cup \dots \cup V_k = V$

Sia n il numero di vertici del grafo, e, rispettivamente, n_1 la cardinalità di V_1 , n_2 la cardinalità di V_2, \dots, n_k la cardinalità di V_k .

Allora, poiché le componenti connesse rappresentano dei sottoinsiemi di V , la somma delle cardinalità delle componenti connesse è pari alla cardinalità di V , dunque l'unione delle componenti connesse è uguale a V .

• Tipologie di grafi notevoli

– Grafo vuoto

Si rappresenta con E_n , e indica un grafo con n vertici e nessun arco.

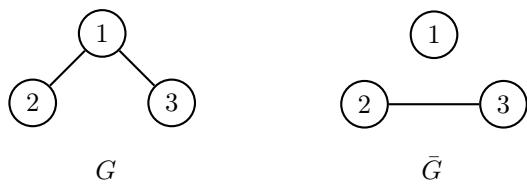
– Grafo completo

Si rappresenta con K_n , e indica un grafo con n vertici e n^2 archi se orientato, $\frac{n(n-1)}{2}$ archi se non orientato.

– Grafo complemento

Dato $G = (V, E)$, \bar{G} o G^C è un nuovo grafo costruito sugli stessi vertici ma con tutti gli archi non presenti in E :

$$\bar{G} = (V, \bar{E}) \Rightarrow (u, v) \in \bar{E} \Leftrightarrow (u, v) \notin E$$



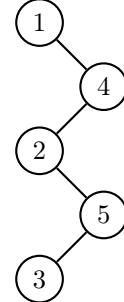
Vale che $\bar{K}_n = E_n$ e che $\bar{\bar{G}} = G$.

Nota Ricordiamo che abbiamo scelto di indicare la cardinalità di V con la lettera n , e la cardinalità di E con la lettera m : $n = |V|$, $m = |E|$.

– Grafo bipartito

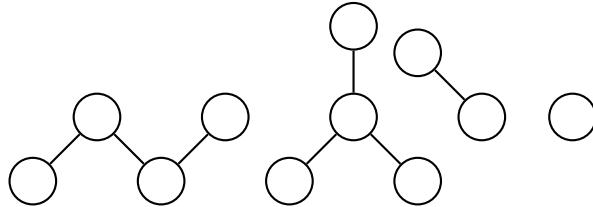
Un grafo bipartito è un grafo che può essere partizionato in due parti tali che all'interno di ciascuna di queste non ci sono archi. I sottografi indotti su queste due parti sono vuoti.

Nell'esempio in questione, siano $V_L = \{1, 2, 3\}$ e $V_R = \{4, 5\}$; allora, vale che $G[V_L] = E_{|V_L|}$ e $G[V_R] = E_{|V_R|}$.



• Foresta

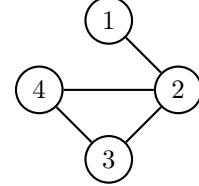
Il termine **foresta** può essere usato per indicare un grafo aciclico. Le componenti connesse di una foresta sono **alberi**, in quanto, come abbiamo visto precedentemente, un albero è un grafo aciclico e connesso. Il grafo riportato sotto è una foresta con quattro alberi.



• Grado, intorno e approfondimenti sulla matrice di adiacenza

Sia $G = (V, E)$ un grafo non orientato. Preso $u \in V$, il **grado** di u , $\deg(u)$ (dall'inglese *degree*), è il numero di vertici adiacenti al vertice u .

Nel grafo in figura, vale che $\deg(1) = 1$, $\deg(2) = 3$, $\deg(3) = 2$ e $\deg(4) = 2$.



Inoltre, l'**intorno** di u è l'insieme dei nodi adiacenti a u , ed è definito nel seguente modo: $N(u) = \{v \in V | (u, v) \in E\}$.

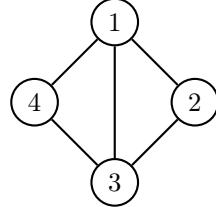
Possiamo quindi affermare che il grado di un nodo è pari alla cardinalità del suo intorno, $\deg(u) = |N(u)|$.

Data la matrice di adiacenza $A = (a_{ij})$ di G , possiamo calcolare il grado di un nodo sommando le righe o le colonne del nodo relativo (dal momento che la matrice di adiacenza di un grafo non orientato è simmetrica):

$$\deg(i) = \sum_{j=1}^n a_{ij} = \sum_{j=1}^n a_{ji}$$

Vediamo un esempio di questa proprietà.

Sia $G = (V = \{1, 2, 3, 4\}, E = \{(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)\})$ il grafo in figura, e sia A la sua matrice di adiacenza.



$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Consideriamo il quadrato di A , $A \times A = A^2 = (a_{ij}^{(2)})$. Cosa conterrà questa matrice?

$$A \times A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 2 \\ 2 & 1 & 3 & 1 \\ 1 & 2 & 1 & 2 \end{bmatrix}$$

Possiamo notare che i numeri sulla diagonale, cioè in posizione (i, i) , indicano esattamente il grado del vertice i , mentre gli altri numeri indicano il numero di cammini di lunghezza 2 tra i nodi relativi alla riga i e alla colonna j . Formalmente, scriveremo:

$$a_{ij}^{(2)} = \begin{cases} \deg(i) & \text{se } i = j \\ \text{numero di cammini di lunghezza 2 tra } i \text{ e } j & \text{se } i \neq j \end{cases}$$

1. Dimostriamo che $a_{ii} = \deg(i) \forall i = j, i = 1, \dots, n$.
Poiché $A \times B = C$, $c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$,

$$a_{ii}^{(2)} = \sum_{k=1}^n a_{ik} \cdot a_{ki} = \sum_{k=1}^n a_{ik} \cdot a_{ik} = \sum_{k=1}^n a_{ik}^2 = \sum_{k=1}^n a_{ik} = \deg(i)$$

dove vale che $a_{ik}^2 = a_{ik}$ perché la matrice è binaria.

2. Dimostriamo che $a_{ij} = \text{numero di cammini di lunghezza 2 tra } i \text{ e } j \forall i \neq j, i = 1, \dots, n$.

Per definizione di prodotto tra matrici, sappiamo che l'elemento nella riga i e nella colonna j della matrice A^2 è definito nel seguente modo:

$$a_{ij}^{(2)} = \sum_{\ell=1}^n a_{i\ell} \cdot a_{\ell j}$$

Tale prodotto è pari a 1 se e solo se $a_{i\ell} = 1$, cioè $(i, \ell) \in E$, e $a_{\ell j} = 1$, cioè $(\ell, j) \in E$, e i due archi (i, ℓ) e (ℓ, j) rappresentano proprio un cammino di lunghezza 2 che va da i a j .

In generale, vale che per ogni $i \neq j$ la matrice risultante dal prodotto di k matrici di adiacenza, A^k , ci dà informazioni sul numero di cammini di lunghezza k tra i nodi i e j . Possiamo dimostrarlo per induzione su k .

Dimostrazione Sia $A^k = \underbrace{A \times A \times A \times \cdots \times A}_k = A^{k-1} \times A$ il prodotto di k matrici di adiacenza per un certo grafo G .

Caso base Per $k = 1$, la condizione è verificata: la matrice di adiacenza $A^1 = A$ di G ci dice il numero di cammini di lunghezza 1, cioè di archi, tra i vertici i e j .

Analogamente, per $k = 2$, abbiamo dimostrato appena sopra la validità della condizione.

Passo induttivo Assumiamo l'ipotesi induttiva che la condizione valga per $k - 1$, e verifichiamolo per k . Siccome $A^k = A^{k-1} \times A$, possiamo fare la stessa considerazione per la dimostrazione del caso in cui $k = 2$:

$$a_{ij}^k = \sum_{\ell=1}^n a_{i\ell}^{(k-1)} \cdot a_{\ell j}$$

dove $a_{i\ell}^{(k-1)}$, per l'ipotesi induttiva, indica il numero di cammini di lunghezza $k - 1$ tra i e ℓ , e $a_{\ell j} = 1$ se e solo se $(\ell, j) \in E$. Dunque, la condizione è verificata.

Analogamente, sulla diagonale principale, cioè quando $i = j$, avremo il numero di cicli di lunghezza k che partono dal nodo i e arrivano a sé stesso. Notiamo che, nei grafi non orientati, la diagonale principale contiene il grado del nodo i -esimo in quanto tutti gli archi di un nodo di un grafo non orientato possono essere visti come un cammino di lunghezza 2 dal nodo a sé stesso.

Lemma Non è possibile costruire un grafo con tutti i gradi distinti.

Dimostrazione Supponiamo per assurdo che il grafo $G = (V, E)$ abbia nodi con tutti i gradi distinti, cioè $\forall i, j = 1, 2, \dots, n : i \neq j \Rightarrow \deg(i) \neq \deg(j)$.

Sappiamo che $\forall i \in V, 0 \leq \deg(i) \leq n - 1$, cioè ci sono tanti gradi possibili quanti sono i nodi. Ciò significa che ogni nodo deve avere un grado diverso nell'intervallo $[0, n - 1]$, cioè devono esistere, tra gli altri, due nodi in particolare: uno con grado 0 e uno con grado $n - 1$. Il primo, quindi, non deve essere connesso ad un altro nodo, e il secondo, per avere grado massimo, deve essere connesso a tutti gli altri nodi. Questo è assurdo, perché non è possibile che esistano contemporaneamente un nodo senza archi e un nodo collegato a tutti gli altri.

Lemma della stretta di mano Sia $G = (V, E)$ un grafo non orientato. Allora, la somma dei gradi dei nodi di G è un numero pari, e in particolare è pari al doppio della cardinalità di E . Formalmente, scriveremo:

$$\sum_{i=1}^n \deg(i) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} = 2|E| = 2m$$

Corollario 1 Conseguenza di tale lemma è la seguente proprietà: in un grafo non orientato $G = (V, E)$, il numero di vertici di grado dispari è sempre pari.

Per dimostrarlo, suddividiamo V in due sottoinsiemi di vertici:

$$\begin{aligned} P &= \{u \in V \mid \deg(u) \text{ è pari}\} \\ D &= \{u \in V \mid \deg(u) \text{ è dispari}\} \end{aligned}$$

P e D costituiscono due partizioni di V : $P \cup D = V$ e $P \cap D = \emptyset$. Sviluppiamo il lemma della stretta di mano:

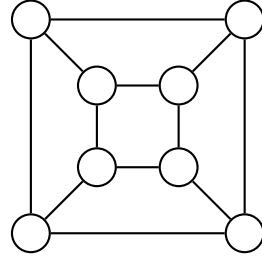
$$\begin{aligned} 2m &= \sum_{u \in V} \deg(u) \\ &= \sum_{u \in P} \deg(u) + \sum_{u \in D} \deg(u) \\ &= \sum_{u \in P} 2 \cdot h(u) + \sum_{u \in D} (2 \cdot h(u) + 1) \\ &= 2 \cdot \sum_{u \in P} h(u) + 2 \cdot \sum_{u \in D} h(u) + \sum_{u \in D} 1 \\ &= 2 \cdot \left(\sum_{u \in P} h(u) + \sum_{u \in D} h(u) \right) + |D| \\ &= 2 \cdot \sum_{u \in V} h(u) + |D| \\ \Rightarrow |D| &= 2m - 2 \cdot \sum_{u \in V} h(u) = 2 \cdot \left(m - \sum_{u \in V} h(u) \right) \end{aligned}$$

dove $h(u)$ è un intero che dipende dai vertici di G e quindi anche $(m - \sum_{u \in V} h(u))$ è un intero, dunque $|D|$ è pari.

Corollario 2 - Grafi k -regolari Un'ulteriore conseguenza del lemma della stretta di mano riguarda i cosiddetti **grafi k -regolari**, cioè i grafi tali che $\forall u \in V, \deg(u) = k$.

Un caso particolare di grafo k -regolare è il **grafo cubico**, un grafo in cui tutti i nodi hanno grado, cioè k , uguale a 3. Per il corollario appena visto, il numero di vertici in questo tipo di grafo è necessariamente pari.

È riportato a lato un grafo cubico.



Il lemma della stretta di mano ci permette di fare le seguenti considerazioni sui grafi k -regolari.

1. $G = (V, E)$ è 2-regolare $\Rightarrow |V| = |E|$. Infatti,

$$2m = \sum_{u \in V} \deg(u) = \sum_{u \in V} 2 = 2n \Rightarrow m = n$$

2. $G = (V, E)$ è 3-regolare $\Rightarrow |V|$ è pari. Infatti,

$$2m = \sum_{u \in V} \deg(u) = \sum_{u \in V} 3 = 3n \Rightarrow n = \frac{2}{3}m$$

Poiché il numero di vertici $n = |V|$ è pari al doppio di $\frac{m}{3}$, e il doppio di un numero è sempre pari, allora si avrà che $|V|$ è pari.

3. $G = (V, E)$ è 4-regolare $\Rightarrow |E|$ è pari. Infatti,

$$2m = \sum_{u \in V} \deg(u) = \sum_{u \in V} 4 = 4n \Rightarrow 2m = 4n \Rightarrow m = 2n$$

Poiché $m = |E|$ è il doppio del numero di vertici, sarà sempre un numero pari.

4. $G = (V, E)$ è 6-regolare $\Rightarrow \begin{cases} |V| \text{ è pari} \Rightarrow |E| \text{ è pari} \\ |V| \text{ è dispari} \Rightarrow |E| \text{ è dispari} \end{cases}$. Infatti,

$$2m = \sum_{u \in V} \deg(u) = \sum_{u \in V} 6 = 6n \Rightarrow 2m = 6n \Rightarrow m = 3n$$

Poiché il numero di archi è pari al prodotto del numero di vertici per 3, che è un numero dispari, se il numero di vertici è pari allora anche il numero di archi sarà pari, altrimenti, se il numero di vertici è dispari, allora anche il numero di archi sarà dispari.

Il lemma della stretta di mano, insieme ai suoi corollari e alle considerazioni fatte finora sul grado di un vertice, non vale per i grafi orientati.

Grado in grafi orientati

Il concetto di grado di un nodo cambia sensibilmente nei grafi orientati. La prima considerazione che va fatta è la distinzione tra ***in-degree***, cioè il numero di archi entranti nel nodo, e ***out-degree***, cioè il numero di archi uscenti dal nodo. Data la matrice di adiacenza di un grafo orientato, l'*in-degree* e l'*out-degree* di un nodo, che possono assumere valori diversi, sono calcolabili in questo modo:

$$\text{in-deg}(i) = \underbrace{\sum_{j=1}^n a_{ji}}_{\text{Sommo gli } 1 \text{ nella } i\text{-esima colonna}}$$

$$\text{out-deg}(i) = \underbrace{\sum_{j=1}^n a_{ij}}_{\text{Sommo gli } 1 \text{ nella } i\text{-esima riga}}$$

Vale che la somma degli *in-degree* e degli *out-degree* dei vertici di un grafo orientato è uguale, ed è uguale anche alla cardinalità di E . Formalmente, scriveremo:

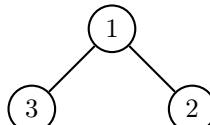
$$\sum_{i=1}^n \text{in-deg}(i) = \sum_{i=1}^n \text{out-deg}(i) = m$$

- **Isomorfismo di grafi**

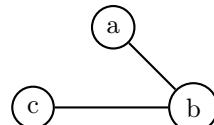
Dati due grafi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, $\phi : V_1 \rightarrow V_2$ è un isomorfismo se valgono le seguenti proprietà:

1. ϕ è biunivoca (o bigettiva)
2. Deve preservare l'adiacenza: $(u, v) \in E_1 \Leftrightarrow (\phi(u), \phi(v)) \in E_2$

Vediamo un esempio.



G



G'

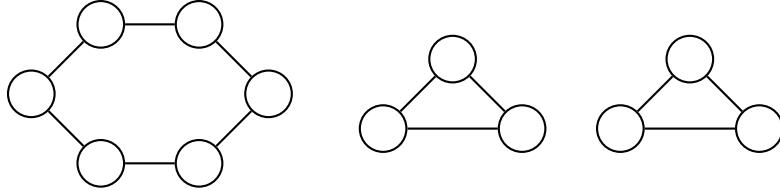
I due grafi G e G' sono isomorfi perché rappresentano la stessa relazione, a prescindere dal nome che diamo ai vertici. Possiamo dire, su entrambi, che esiste un vertice adiacente agli altri due, ma questi ultimi non sono adiacenti tra di loro. In questo caso, scriveremo $G \simeq G'$.

L'isomorfismo di grafi è un tema molto discusso nell'algoritmica, perché non esiste un algoritmo di tempo polinomiale che, dati due grafi, ritorni la funzione di isomorfismo tra di essi (qualora esistesse).

Condizioni necessarie per l'isomorfismo

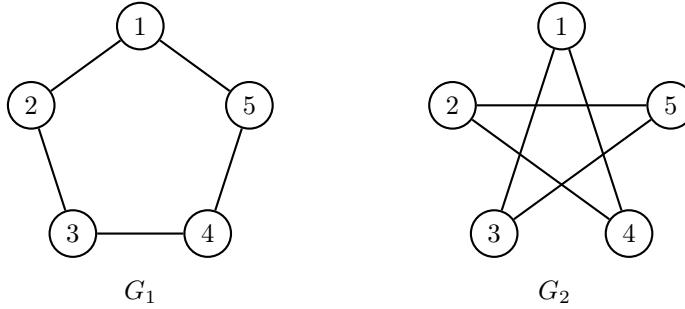
1. $G_1 \simeq G_2 \Rightarrow |V_1| = |V_2|$
2. $G_1 \simeq G_2 \Rightarrow |E_1| = |E_2|$; infatti, affinché ϕ sia un isomorfismo, $\forall u \in V_1, \deg(u) = \deg(\phi(u))$.
3. La **degree-sequence** di un grafo $G = (V, E)$, $\deg\text{-seq}(G)$, è un vettore di n elementi contenente i gradi dei vertici in un certo ordine, solitamente in ordine crescente.
In caso di isomorfismo, vale che $G_1 \simeq G_2 \Rightarrow \deg\text{-seq}(G_1) = \deg\text{-seq}(G_2)$.
4. $G_1 \simeq G_2 \Rightarrow G_1$ e G_2 hanno lo stesso numero di componenti connesse.

Le presenti condizioni sono tutte necessarie affinché due grafi siano isomorfi, tuttavia non sono sufficienti se prese singolarmente: di seguito, presentiamo due grafi che rispettano le prime tre condizioni, ma non l'ultima. È evidente che i due grafi non sono isomorfi, nonostante rispettino le prime tre condizioni, perché il grafo a sinistra è costituito da una sola componente连通的, mentre il secondo è costituito da due componenti connesse.



Dimostrazione isomorfismo tra due grafi

Siano G_1 e G_2 i grafi in figura.



E sia ϕ definita nel seguente modo:

$$\phi(1) = 2 \quad \phi(2) = 4 \quad \phi(3) = 1 \quad \phi(4) = 3 \quad \phi(5) = 5$$

Per verificare che ϕ sia un isomorfismo per G_1 e G_2 , bisogna verificare le due proprietà degli isomorfismi:

1. ϕ mappa ciascun vertice di G_1 ad uno e un solo vertice di G_2 , quindi è una funzione biunivoca.

2. Verifichiamo la preservazione delle adiacenze:

- (a) $(1, 2) \in E_1 \rightarrow (\phi(1), \phi(2)) = (2, 4) \in E_2$
- (b) $(2, 3) \in E_1 \rightarrow (\phi(2), \phi(3)) = (4, 1) \in E_2$
- (c) $(3, 4) \in E_1 \rightarrow (\phi(3), \phi(4)) = (1, 3) \in E_2$
- (d) $(4, 5) \in E_1 \rightarrow (\phi(4), \phi(5)) = (3, 5) \in E_2$
- (e) $(5, 1) \in E_1 \rightarrow (\phi(5), \phi(1)) = (5, 2) \in E_2$

Quindi ϕ è un isomorfismo per G_1 e G_2 .

In questo particolare caso, possiamo verificare che G_1 è isomorfo al suo complemento: $G_1 \simeq \bar{G}_1$. Per dimostrarlo, bisogna verificare che G_2 è effettivamente il grafo complemento di G_1 .

Notiamo innanzitutto che $V_1 = V_2 = \{1, 2, 3, 4, 5\}$, e che $|E_1| = |E_2|$. Possiamo proseguire nella dimostrazione verificando che in G_2 sono presenti gli archi assenti in G_1 .

- In G_1 , il vertice 1 è adiacente ai vertici 2 e 5, mentre in G_2 è adiacente ai vertici 3 e 4.
- In G_1 , il vertice 2 è adiacente ai vertici 1 e 3, mentre in G_2 è adiacente ai vertici 4 e 5.
- In G_1 , il vertice 3 è adiacente ai vertici 2 e 4, mentre in G_2 è adiacente ai vertici 1 e 5.
- In G_1 , il vertice 4 è adiacente ai vertici 3 e 5, mentre in G_2 è adiacente ai vertici 1 e 2.
- In G_1 , il vertice 5 è adiacente ai vertici 1 e 4, mentre in G_2 è adiacente ai vertici 2 e 5.

Possiamo concludere affermando quindi che $G_1 \simeq \bar{G}_1 = G_2$, con funzione di isomorfismo ϕ .

• **Condizione necessaria per la connettività**

Se $G = (V, E)$ è un grafo non orientato连通的, allora $|E| \geq |V| - 1$.

Dimostrazione Avviene per induzione su $n = |V|$.

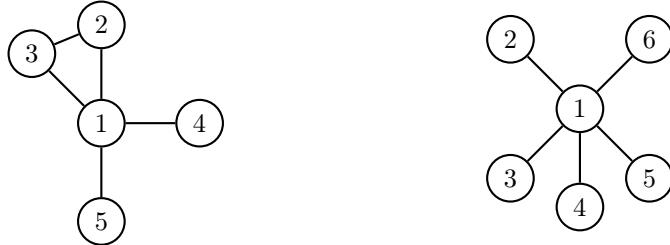
Caso base Se $n = 1$, il grafo è banalmente连通的, perché costituito da un solo nodo e nessun arco.

Se $n = 2$, il grafo è连通的 se esiste un arco che collega i due vertici. Quindi il caso base è verificato.

Passo induttivo Assumiamo l'ipotesi induttiva che la proprietà valga per ogni grafo G con $|V|$ fino a n , e verifichiamolo per $n + 1$. Supponiamo quindi che $|V| = n + 1$, e prendiamo da G un nodo z .

Rimuoviamo z da G : otteniamo un nuovo insieme di vertici $V' = V \setminus \{z\}$.

Ragioniamo sul sottografo indotto su V' , $G[V']$. Non possiamo applicare l'ipotesi induttiva su $G[V']$ nella sua interezza, in quanto tale sottografo indotto può essere connesso o disconnesso sulla base del nodo che noi decidiamo di rimuovere.



Se volessimo rimuovere, per esempio, il nodo 5 dal grafo a sinistra, otterremmo un sottografo connesso, ma rimuovendo il nodo 1 dal grafo a destra otterremmo ben 5 componenti connesse costituite da un solo nodo. Non avendo la certezza sulla natura del sottografo indotto su V' , decidiamo di considerare tutte le componenti connesse di tale sottografo, di numero k , e di applicare su di esse l'ipotesi induttiva: siano V_1, V_2, \dots, V_k le k componenti connesse di V' ; vogliamo dimostrare che, $\forall V_i \in V', |E_i| \geq |V_i| - 1$. Dal momento che il numero totale di archi è dato dalla somma tra gli archi delle componenti connesse e gli archi incidenti a z , possiamo affermare con certezza che

$$|E| = \sum_{i=1}^k |E_i| + \deg(z)$$

e applicare l'ipotesi induttiva sulle componenti connesse di $G[V']$:

$$|E_i| \geq |V_i| - 1$$

Combiniamo le suddette considerazioni:

$$\begin{aligned} |E| &\geq \sum_{i=1}^k (|V_i| - 1) + \deg(z) \\ &= \sum_{i=1}^k |V_i| - k + \deg(z) \\ &= |V| - 1 + \deg(z) - k \end{aligned}$$

dove il -1 nell'ultima riga si deve al fatto che z è stato tolto, per cui non appartiene a V' .

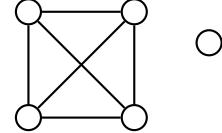
Possiamo concludere la dimostrazione se riusciamo a provare che $\deg(z) - k \geq 0 \Rightarrow \deg(z) \geq k$. È assurdo che $\deg(z) < k$, perché significa che il grafo di partenza è disconnesso: affinché la rimozione di z abbia generato k componenti connesse, vuol dire che z deve essere stato adiacente ad almeno un nodo in ciascuna componente connessa.

Quindi, $\deg(z) \geq k$ e $|E| \geq |V| - 1$, come volevasi dimostrare.

- **Condizione sufficiente per la connettività**

La condizione di cui sopra non è sufficiente per stabilire se un grafo è connesso, basta trovare un controsenso per verificarlo.

Nel grafo di lato, $|V| = 5$, $|E| = 6$.



Una condizione sufficiente per la connettività è la seguente: sia $G = (V, E)$ un grafo non orientato. Se $\forall u \in V, \deg(u) > \frac{n-1}{2}$, allora G è connesso.

Dimostrazione Avviene per assurdo. Supponiamo che G non sia connesso, e supponiamo che sia costituito da k componenti connesse. Per semplicità, imponiamo $k = 2$. Allora, V_1 e V_2 sono due partizioni di V . Prendiamo due nodi $u \in V_1$ e $v \in V_2$: questi sono collegati con $\frac{n-1}{2}$ nodi appartenenti al loro stesso insieme. Formalmente, scriveremo:

$$|V_1| \geq \frac{n-1}{2} + 1 = \frac{n+1}{2}, \quad |V_2| \geq \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

Dove il $+1$ si deve ai nodi u e v rispettivamente. Allora, dovrebbe valere che

$$|V| = |V_1| + |V_2| \geq \frac{n+1}{2} + \frac{n+1}{2} = n+1$$

che è assurdo perché $|V| = n$.

- **Condizione necessaria per l'aciclicità**

Sia $G = (V, E)$ un grafo non orientato aciclico. Allora $|E| \leq |V| - 1$.

Dimostrazione Avviene per induzione su n , in modo analogo alla dimostrazione della condizione necessaria per la connettività.

Caso base Per $n = 1$, un grafo con un solo nodo e senza archi è banalmente aciclico, quindi la proprietà è verificata.

Per $n = 2$, un grafo con due nodi e al più un arco è aciclico. Dunque la condizione è verificata.

Passo induttivo Supponiamo che la proprietà valga per ogni grafo $G = (V, E)$ con $|V|$ fino a n , e verifichiamolo per $n + 1$. Prendiamo un grafo G con $|V| = n + 1$ avente un vertice z , e rimuoviamo tale nodo. Otteniamo il sottoinsieme $V' = V \setminus \{z\}$; sia $G' = G[V']$ il sottografo indotto da V' e costituito dalle k componenti connesse V_1, V_2, \dots, V_k .

Applichiamo l'ipotesi induttiva sulle componenti connesse: $\forall i = 1, \dots, k$, $|E_i| \leq |V_i| - 1$.

$$\begin{aligned} |E| &= \sum_{i=1}^k |E_i| + \deg(z) \\ &\leq \sum_{i=1}^k (|V_i| - 1) + \deg(z) \\ &= \sum_{i=1}^k |V_i| - k + \deg(z) \\ &= |V| - 1 + \deg(z) - k \end{aligned}$$

Dobbiamo riuscire a verificare che $\deg(z) - k \leq 0 \Rightarrow \deg(z) \leq k$.

Non può essere altrimenti, perché se $\deg(z) > k$ vorrebbe dire che nel grafo di partenza avremmo un ciclo: z sarebbe cioè connesso ad almeno due nodi in una delle k componenti connesse, che è assurdo per ipotesi. Quindi $\deg(z) \leq k$, $|E| \leq |V| - 1$ come volevasi dimostrare.

Anche questa, analogamente, è una condizione necessaria ma non sufficiente per l'aciclicità:

$$|E| \leq |V| - 1 \not\Rightarrow G \text{ aciclico}$$

Dimostrazioni per casa

- Se $\forall u \in V : \deg(u) \geq 2$, allora G è ciclico.

Dimostrazione Per il lemma della stretta di mano, abbiamo che:

$$2m = \sum_{i=1}^n \deg(i) \geq \sum_{i=1}^n 2 = 2n \Rightarrow 2m \geq 2n \Rightarrow m \geq n$$

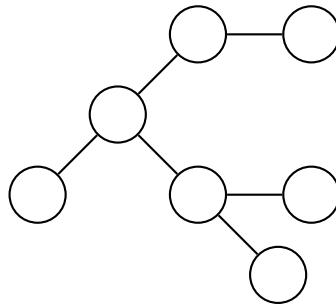
Poiché $|E| \geq |V|$, viene a mancare la condizione sufficiente per l'aciclicità ($|E| \leq |V| - 1$), quindi il grafo è ciclico.

- Se G è aciclico, allora $\exists u \in V \ni \deg(u) \leq 1$.

Dimostrazione Neghiamo la tesi, e supponiamo per assurdo che il grafo non contenga vertici con grado inferiore o uguale a 1. Possiamo applicare il lemma della stretta di mano allo stesso modo di come abbiamo fatto per la dimostrazione precedente, e verificare che in questo caso avverrebbe che $m \geq n$, cioè $|E| \geq |V|$. Ma questo è assurdo, perché affinché un grafo sia aciclico è necessario che $|E| \leq |V| - 1$, dunque la tesi è verificata.

20.2 Alberi

Come abbiamo già visto nel capitolo introduttivo, un **albero** è un particolare tipo di grafo, che ha contemporaneamente le proprietà di essere aciclico e connesso. Ricordiamo che nella teoria dei grafi un albero non ha un nodo radice dal quale vengono "generati" i figli, come nel caso degli alberi radicati, bensì si tratta di una struttura piana (a cui, volendo, possiamo imporre una gerarchia per renderla un albero radicato, ma ai fini del nostro studio questa considerazione non è di nostro interesse).



Ciò che invece ci può interessare di questo tipo di dato è che si tratta di una struttura fragile: supponiamo di voler prendere un arco qualsiasi all'interno di un albero e rimuoverlo; il grafo si dividerebbe in due componenti connesse, mentre aggiungendo un arco a caso tra i nodi dell'albero si formerebbe un ciclo. In entrambi i casi, il risultato dell'inserimento o della cancellazione di un arco non sarebbe più un albero.

Affermazioni sugli alberi Sia $G = (V, E)$ un grafo non orientato. Le seguenti affermazioni sono equivalenti:

- 1) G è un albero.
- 2) Due vertici qualsiasi di G sono connessi da un unico cammino.
- 3) G è connesso, ma se rimuoviamo un qualsiasi arco di G , il grafo diventa disconnesso.
- 4) G è connesso e $|E| = |V| - 1$.
- 5) G è aciclico e $|E| = |V| - 1$.
- 6) G è aciclico, ma se aggiungiamo un qualsiasi arco a G , il grafo diventa ciclico.

Le dimostrazioni delle suddette affermazioni non sono di nostro interesse, se il lettore si volesse cimentare nella dimostrazione è sufficiente provare che l'affermazione 1) implica l'affermazione 2), l'affermazione 2) implica l'affermazione 3) *et cetera*, fino a dimostrare che l'affermazione 6) implica l'affermazione 1).

È interessante osservare che le affermazioni 4) e 5) sono la diretta conseguenza delle condizioni necessarie di connettività e aciclicità: dal momento che in un grafo connesso $|E| \geq |V| - 1$ e in un grafo aciclico $|E| \leq |V| - 1$, in un albero queste condizioni valgono entrambe, per cui $|E| = |V| - 1$.

Va tuttavia sottolineato che questa non è una condizione sufficiente affinché un grafo possa essere un albero: G è un albero $\Rightarrow |E| = |V| - 1$, ma $|E| = |V| - 1 \not\Rightarrow G$ è un albero.

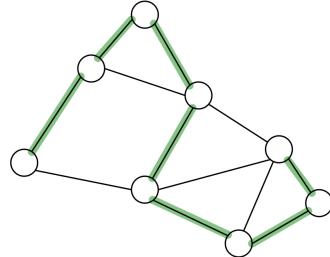
Vale inoltre che G è un albero se e solo se esiste ed è unico un cammino tra ogni coppia di vertici. La dimostrazione avviene per assurdo: supponiamo che esistano due vertici u e v , tra i quali esistono almeno due cammini distinti. Questo è assurdo, perché vuol dire che si creerebbe un ciclo ad un certo punto del grafo.

20.2.1 Alberi di copertura minimi

Un **albero di copertura** (in inglese *Spanning Tree*) di un grafo $G = (V, E)$ non orientato e connesso è un insieme di archi $T \subseteq E$ tale che (V, T) è un albero.

Dato un grafo, possono esistere più alberi di copertura. Inoltre, vale che $|T| = |V| - 1$.

Del grafo a fianco, è evidenziato in verde un possibile albero di copertura.



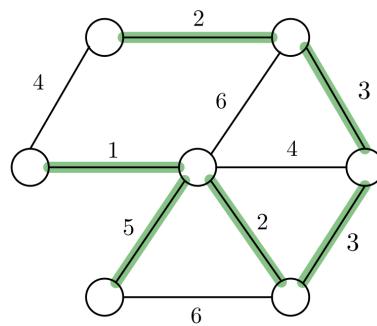
Se per un grafo G esiste una funzione peso $w : E \rightarrow \mathbb{R}$ relativa agli archi di G , allora possiamo esprimere il concetto di albero di copertura minimo.

Dato $G = (V, E, w)$, con $w : E \rightarrow \mathbb{R}$, l'**albero di copertura minimo** (in inglese *Minimum Spanning Tree*, o *MST*) di G è un albero di copertura avente somma dei pesi sugli archi minima.

Il peso di ciascun arco si rappresenta con la formulazione $w(u, v)$, e la somma dei pesi di un albero di copertura si rappresenta in questo modo:

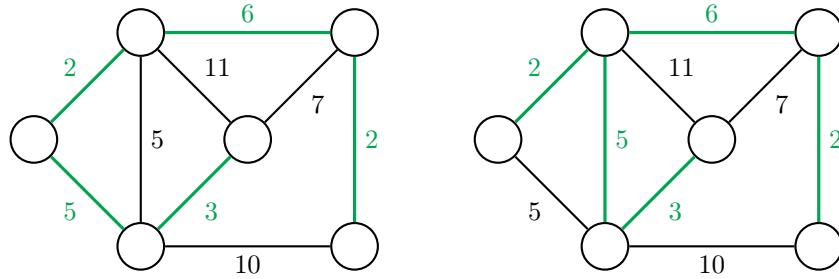
$$W(T) = \sum_{(u, v) \in T} w(u, v)$$

Quindi, l'albero di copertura minimo di un grafo ha come somma dei pesi degli archi $\min(W(T))$.



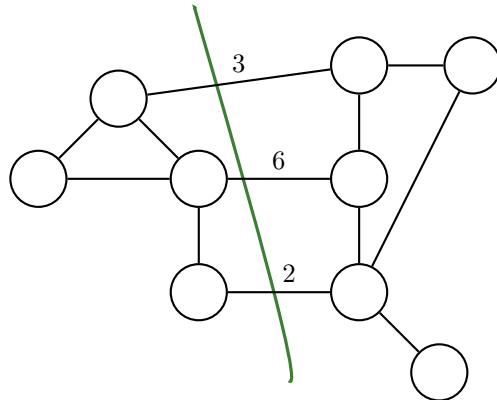
Essendo appunto un albero, il *Minimum Spanning Tree* di un grafo ha come cardinalità dell'insieme degli archi $|V| - 1$, e vale che per un grafo possono esistere più alberi di copertura minimi.

Nel seguente albero sono mostrati due possibili alberi di copertura minimi aventi $W(T) = 18$.



20.2.2 Fatto cruciale degli *MST*

Un **taglio** è una partizione del grafo in due sottoinsiemi di vertici S e $V \setminus S$.



Una volta effettuato un taglio, esiste sempre almeno un arco che lo attraversa; nell'esempio mostrato ne esistono tre, aventi peso 3, 6 e 2. L'arco di peso minore che attraversa il taglio, nel nostro caso quello avente peso 2, è definito **arco leggero**.

Il **fatto cruciale degli *MST*** afferma che per ogni arco leggero che attraversa un taglio esiste almeno un *MST* che contiene quell'arco.

Dimostrazione Prendiamo un albero di copertura minima T . Distinguiamo due casi:

- $(u, v) \in T$. Allora, la condizione è verificata.
- $(u, v) \notin T$. Allora, cerchiamo un altro *MST* T' che contenga l'arco leggero.

Utilizziamo la strategia del *cuci-e-taglia*: consideriamo l'insieme di archi $T \cup \{(u, v)\}$; l'aggiunta di un arco ad un albero comporta, per definizione di albero, la formazione di un ciclo. Questo implica che esiste un ulteriore arco $(x, y) \in T$ che attraversa il taglio e appartiene al ciclo.

Sia quindi $T' = T \cup \{(u, v)\} \setminus \{(x, y)\}$. Per il modo in cui T' è stato costruito, cioè attraverso la rimozione di un arco che collega due componenti connesse e l'aggiunta di un altro arco che collega le stesse componenti connesse, sappiamo che T' è sicuramente un albero, ma è anche un albero di copertura, perché come T collega ancora tutto l'insieme di vertici V . Ma è minimo?

Essendo T un albero di copertura minimo e T' un albero di copertura, vale che $W(T) \leq W(T')$. Se riusciamo a dimostrare che vale contemporaneamente il fatto che $W(T) \geq W(T')$, che implica $W(T) = W(T')$, allora la dimostrazione è conclusa. Ragioniamo sul peso di T' :

$$T' = T \cup \{(u, v)\} \setminus \{(x, y)\} \Rightarrow W(T') = W(T) + w(u, v) - w(x, y)$$

La condizione che vogliamo dimostrare è vera se

$$w(u, v) - w(x, y) \leq 0 \Rightarrow w(u, v) \leq w(x, y),$$

ma questo sarà sempre vero, perché abbiamo scelto di aggiungere l'arco leggero (u, v) avente peso minore o uguale rispetto a quelli degli altri archi che attraversano il taglio.

Vale dunque che $W(T) = W'(T)$, perché vale contemporaneamente che $W(T) \geq W(T')$ e che $W(T) \leq W(T')$. Quindi anche T' è un albero di copertura minimo.

Questo fatto sta alla base di numerosi algoritmi per la ricerca di un *MST* in un grafo.

20.2.3 Teorema fondamentale degli *MST*

Sia $G = (V, E, w)$ un grafo non orientato e connesso, dove $w : E \rightarrow \mathbb{R}$ è una funzione peso sugli archi. Valgano le seguenti ipotesi:

- Sia $A \subseteq E$ contenuto in qualche *MST*.
- Sia $(S, V \setminus S)$ un taglio che "rispetta" A , cioè tale che nessun arco di A attraversa il taglio.
- Sia (u, v) un arco leggero che attraversa il taglio.

Allora, il **teorema fondamentale degli *MST*** afferma che $A \cup \{(u, v)\}$ è contenuto in qualche *MST*. Si può anche dire che (u, v) è **sicuro** per A .

Dimostrazione La dimostrazione è analoga a quella del fatto cruciale degli *MST*. Chiamiamo T l'albero di copertura minima che contiene A , per ipotesi. Distinguiamo due casi:

- $(u, v) \in T \Rightarrow A \cup \{(u, v)\} \in T$. Allora la condizione è banalmente verificata.
- $(u, v) \notin T$. In modo analogo alla dimostrazione precedente, notiamo che l'aggiunta di (u, v) a T provoca la formazione di un ciclo. Indichiamo quindi con (x, y) un arco che si trova nel ciclo e che attraversa il taglio, e decidiamo di toglierlo.

Sia $T' = T \cup \{(u, v)\} \setminus \{(x, y)\}$ un nuovo albero. Allora, per ipotesi vale che $W(T) \leq W(T')$ (essendo T un *MST*), ma vale anche che $W(T) \geq W(T')$:

$$\begin{aligned} W(T') &= W(T) + \underbrace{w(u, v) - w(x, y)}_{\leq 0} \\ &\leq W(T) \end{aligned}$$

Quindi, $W(T) = W(T')$, e dunque T' è un *MST*.

Bisogna ancora dimostrare che $A \cup \{(u, v)\} \subseteq T'$. Questo è sempre vero, perché:

- $A \subseteq T'$ perché A rispetta il taglio per ipotesi (quindi nemmeno (x, y) apparteneva ad A);
- $(u, v) \in T'$ per costruzione.

Ciò significa che se togliendo e aggiungendo un arco la somma dei pesi non cambia, allora vuol dire che $w(u, v) = w(x, y)$. Ma com'è possibile che questo accada nel caso in cui gli archi che attraversano un taglio hanno tutti peso diverso, come nell'esempio mostrato nella spiegazione del fatto cruciale degli *MST*?

Tale situazione si risolve nel primo caso: l'arco leggero appartiene già all'albero di copertura minima, per cui non c'è bisogno di cercare altri *MST* attraverso la strategia del "cuci-e-taglia".

Esercizio per casa Dimostrare che se l'arco leggero che attraversa un taglio è unico, allora appartiene a tutti gli *MST*. Si consideri il problema come conseguenza del fatto cruciale degli *MST*.

Supponiamo, per assurdo, che esista un *MST* T tale che $(u, v) \notin T$. Consideriamo $T \cup (u, v)$: questo insieme contiene un ciclo. Consideriamo ora l'arco (x, y) appartenente al ciclo, e decidiamo di rimuoverlo. Sia $T' = T \cup \{(u, v)\} \setminus \{(x, y)\}$. Allora, varrà che $W(T') = W(T) + w(u, v) - w(x, y)$, e affinché T' sia un *MST* deve valere che $w(u, v) < w(x, y)$ (e non $w(u, v) \leq w(x, y)$, come nelle dimostrazioni precedenti, perché per ipotesi (u, v) è l'unico arco di peso minimo che attraversa il taglio), quindi $W(T') < W(T)$, che è assurdo per ipotesi perché vorrebbe dire che T non era un *MST*.

20.2.4 Corollario del teorema fondamentale degli MST

Sia $G = (V, E)$ un grafo non orientato e connesso, e valgano le seguenti ipotesi:

- Sia $A \subseteq E$ contenuto in qualche MST.
- Sia $C = (V_C, E_C)$ una componente连通 della foresta $G_A = (V, A)$.
- Sia (u, v) un arco leggero che connette C ad un'altra componente连通 di G_A .

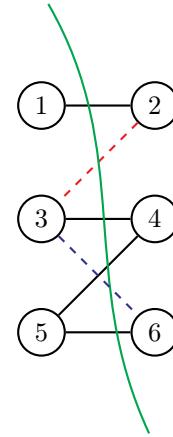
Allora, (u, v) è sicuro per A , cioè $A \cup \{u, v\}$ è contenuto in qualche MST.

Si può dimostrare in due modi: sfruttando il teorema fondamentale degli MST oppure facendo finta di non conoscerlo, e dimostrarlo di conseguenza.

Dimostrazione Supponendo di conoscere il teorema fondamentale degli MST, facciamo la seguente considerazione: poiché C è una componente连通 all'interno della foresta $G_A = (V, A)$, allora il taglio $(V_C, V \setminus V_C)$ rispetta A . Di conseguenza, per il teorema fondamentale degli MST, l'arco leggero (u, v) , che è incidente a due vertici presenti in V_C e in $V \setminus V_C$, è un arco sicuro per A .

Nota Nella strategia del "cuci-e-taglia", è importante seguire l'ordine delle operazioni: prima è necessario inserire un arco, considerare il ciclo che si è creato e poi rimuovere un arco che attraversa il taglio e che fa parte del ciclo. Se non seguissimo l'ordine, rischieremmo di creare una struttura che non è un albero, dunque non è utile per la ricerca di un MST.

Immaginiamo di avere il grafo in figura, che presenta un taglio che attraversa tutti gli archi. Se rimuovessimo prima l'arco $(2, 3)$ (tratteggiato in rosso, già presente nel grafo), non avremmo un ciclo dove inserire un ulteriore arco. Potremmo posizionarlo, per esempio, al posto dell'arco $(3, 6)$ (tratteggiato di blu, non già presente nel grafo), ma la struttura ottenuta non è un albero di copertura minima, né tantomeno un albero.



20.2.5 Algoritmi (propedeutici) sugli *MST*

Generic-MST L'algoritmo **Generic-MST** non è un vero e proprio algoritmo, bensì un'idea, un modello su cui si basano gli algoritmi implementabili per la ricerca di un *MST*.

```

1 Generic-MST(G, w)
2   A ← ∅
3   while A non forma un MST           // oppure (while |A| < |V| - 1)
4     trova un arco (u, v) "sicuro" per A
5     A ← A ∪ {(u, v)}
6   return A

```

Strutture dati per insiemi disgiunti Compiamo una digressione per presentare delle strutture dati avanzate che utilizzeremo per memorizzare e compiere operazioni su collezioni dinamiche di insiemi.

Siano $S_1 = \{4, 6, 7, 1\}$, $S_2 = \{2, 3, 5\}$ e $S_3 = \{8, 9, 10\}$. Sono disgiunti, e la loro unione contiene i numeri naturali da 1 a 10. Supponiamo che questi si evolvano nel tempo (*e.g.*, S_1 si divide in due, ecc...) e che ogni insieme è caratterizzato da un **rappresentante**, immaginiamo che per S_1 sia 7, per S_2 sia 2 e per S_3 sia 10.

Operatori

- **Make_Set(x)** → {x} - Usato in fase di inizializzazione, crea un insieme il cui unico elemento è x.
- **Union(x, y)** → $S_x \cup S_y$ - x e y sono i rappresentanti degli insiemi S_x e S_y .
- **Find_Set(x)** → y: $x \in S_y$ - Dato x, ritorna il rappresentante dell'insieme che contiene x.

Rappresentazioni

• Liste concatenate

In questa rappresentazione, *e.g.*, $S_1 = 7 \rightarrow 4 \rightarrow 6 \rightarrow 1 \rightarrow \text{NIL}$, cioè è rappresentabile attraverso una lista concatenata avente in testa il rappresentante dell'insieme. Il vantaggio di questa rappresentazione rispetto agli array è dovuto alla dinamicità delle liste concatenate, pur avendo lo svantaggio del relativamente alto tempo d'esecuzione per la verifica dell'appartenenza di un elemento.

• Heap binari

Questa rappresentazione prevede che gli insiemi siano degli heap binari, i cui nodi hanno dei puntatori al loro padre eccezion fatta per la radice (che coincide con il rappresentante?).

Occorre fare attenzione alle operazioni di unione, in quanto potrebbe essere necessario risolvere eventuali violazioni delle proprietà degli heap.

Verifica componenti connesse Vogliamo trovare un algoritmo che dato un grafo verifichi se si tratta di un'unica componente连通, e nel caso in cui non lo sia restituisca le sue componenti connesse.

```

1 Connected_Components(G)
2   for each v ∈ V[G]
3     Make_Set(v)
4   for each (u, v) ∈ E[G]
5     if Find_Set(u) ≠ Find_Set(v)
6       Union(u, v)

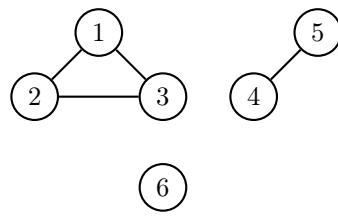
```

Con $V[G]$ e $E[G]$ indichiamo rispettivamente l'insieme dei vertici e l'insieme degli archi di G .

Le prime due righe dell'algoritmo fungono da inizializzazione: creano tanti singoletti quanti sono i vertici del grafo G .

Le ultime tre righe iterano lungo tutti gli archi, controllano se gli insiemi che contengono i nodi sono diversi e in tal caso li unisce.

Algoritmo in azione Simuliamo l'esecuzione dell'algoritmo sul seguente grafo:



Step	Insiemi	Arco
0	{1}, {2}, {3}, {4}, {5}, {6}	
1	{1, 2}, {3}, {4}, {5}, {6}	(1, 2)
2	{1, 2}, {3}, {4, 5}, {6}	(4, 5)
3	{1, 2, 3}, {4, 5}, {6}	(1, 3)
4	{1, 2, 3}, {4, 5}, {6}	(2, 3)

20.2.6 Algoritmo di Kruskal

```

1 Kruskal(G, w)
2   A ← ∅
3   for each v ∈ V[G]
4     Make_Set(v)
5   ordino gli archi di G in modo crescente
6   for each (u, v) ∈ E[G]
7     if Find_Set(u) ≠ Find_Set(v)
8       Union(u, v)
9     A ← A ∪ {(u, v)}
10  return A

```

Spiegazione L'algoritmo inizializza un insieme A vuoto e tanti singoletti quanti sono i vertici di G , ordina in modo crescente gli archi sulla base del loro peso e itera su questi archi: se i nodi che li costituiscono non sono nello stesso insieme li unisce, e inserisce l'arco in A .

Alternativamente, si può dire che l'algoritmo estrae gli archi uno alla volta, e ogni volta verifica se si crea un ciclo con gli archi già estratti. Se si crea un ciclo, l'arco viene buttato via, altrimenti viene aggiunto all'albero.

La **correttezza** è immediata, ed è garantita dal fatto che vengono rispettate le condizioni viste fino ad ora sugli *MST*, perché, per esempio, la riga 2 è vera per ogni G poiché l'insieme vuoto è sottoinsieme di ogni *MST* e perché il fatto che gli archi siano ordinati in modo crescente implica che vengano considerati per primi gli archi leggeri. Inoltre, la correttezza è garantita anche dal fatto che l'algoritmo rispetta il corollario sulle componenti connesse del teorema fondamentale degli *MST*: la foresta $G_A = (V, A)$ è costituita da sempre meno componenti connesse (che, all'inizio dell'algoritmo, sono presenti nel numero di n) man mano che A include nuovi archi leggeri; quando, alla fine dell'algoritmo, G_A è costituita da una sola componente连通的, vuol dire che sono stati estratti tutti i $|V| - 1$ archi necessari alla formazione di un albero di copertura minima.

Per quanto riguarda la **complessità**, il primo ciclo **for each** (righe 3-4) compie n iterazioni, l'ordinamento ha costo $m \log(m)$ (perché $|E| = m$) e il secondo ciclo **for each** compie m iterazioni e le operazioni che compie (**Find_Set** e **Union**) hanno costo logaritmico rispetto a m , dunque la complessità dell'algoritmo sarà la seguente:

$$T(n, m) = O(n + m \log(m) + m \log(m)) = O(m \log(m)),$$

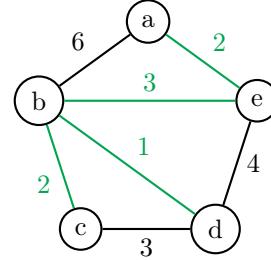
dove $n = O(m \log(m))$ perché il grafo è connesso, e quindi vale che $m \geq n - 1$.

Algoritmo in azione

Simuliamo l'esecuzione dell'algoritmo sul grafo in figura.

Come possiamo notare, vengono considerati prima gli archi di peso minore, viene controllata l'appartenenza allo stesso insieme dei vertici che li costituiscono e viene costruito l'albero di copertura minima di conseguenza.

È evidenziato l'*MST* risultante.



Step	A	Insiemi	Arco
0	\emptyset	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$	(b, d)
1	$\{(b, d)\}$	$\{a\}, \{b, c\}, \{d\}, \{e\}$	(b, c)
2	$\{(b, d), (b, c)\}$	$\{a\}, \{b, c, d\}, \{e\}$	(a, e)
3	$\{(b, d), (b, c), (a, e)\}$	$\{a, e\}, \{b, c, d\}$	(c, d)
4	$\{(b, d), (b, c), (a, e)\}$	$\{a, e\}, \{b, c, d\}$	(b, e)
5	$\{(b, d), (b, c), (a, e), (b, e)\}$	$\{a, b, c, d, e\}$	(e, d)
6	$\{(b, d), (b, c), (a, e), (b, e)\}$	$\{a, b, c, d, e\}$	(a, b)
7	$\{(b, d), (b, c), (a, e), (b, e)\}$	$\{a, b, c, d, e\}$	/

20.2.7 Algoritmo di Prim

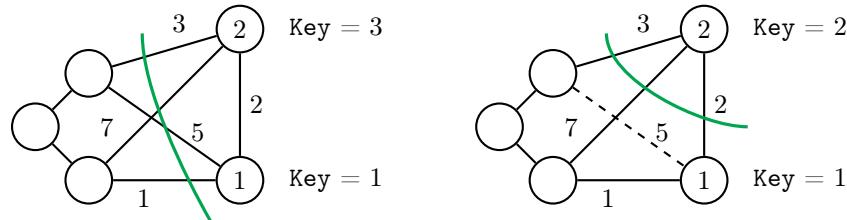
L'**Algoritmo di Prim** è un algoritmo su grafi connessi, non orientati e pesati sugli archi che, dato un grafo, la sua funzione peso e un nodo detto **radice**, ritorna un albero di copertura minima per tale grafo.

La differenza sostanziale tra questo algoritmo e quello di Kruskal consiste nel fatto che quest'ultimo, in ogni istante di tempo, memorizza un insieme di archi che forma una foresta e che alla fine diventa un albero, unendo le varie componenti connesse date dall'aggiunta di archi su tutta la superficie dell'albero. Dal canto suo, l'algoritmo di Prim considera il grafo come un albero radicato, e dalla radice r di questo fa crescere un albero di copertura.

Possiamo anche dire che mentre in Kruskal l'albero A è rappresentato in maniera esplicita, qui A è implicito, e viene costruito alla fine dell'algoritmo servendosi di un campo particolare $\pi[u]$, $u \in V$, associato ad ogni vertice, che ci dà informazioni sulla struttura dell'albero.

L'algoritmo di Prim si serve di una struttura dati particolare, una coda di minima priorità Q che memorizza i vertici del grafo per estrarli, uno alla volta, sulla base di un ulteriore campo $\text{Key}[u]$, $u \in V$, che contiene il minore tra i pesi degli archi di u che attraversano il taglio. Spieghiamo il significato di questa frase analizzando i passaggi dell'algoritmo:

1. **Inizializzazione** All'inizio dell'algoritmo, Q corrisponde all'insieme dei vertici V , il campo Key di ogni nodo viene inizializzato a ∞ , tranne per la radice il cui campo Key viene inizializzato a 0, e il campo π di ogni nodo viene inizializzato a NIL.
2. **Estrazione** Finché Q non è vuoto, viene estratto il minimo da Q , cioè il vertice del grafo che non è ancora stato estratto avente campo Key minore. Il processo di estrazione prevede la formazione, o meglio, l'aggiornamento di un **taglio** che divide i nodi che non sono stati ancora estratti, cioè i nodi appartenenti a Q , dai nodi che sono già stati estratti, cioè quelli appartenenti a $V \setminus Q$.
3. **Aggiornamento campi** L'estrazione di un nodo da Q causa, chiaramente, la rimozione dell'elemento da Q e la sua aggiunta in $V \setminus Q$. Conseguenza di ciò è anche la modifica del taglio, per cui è necessario controllare la validità del campo Key dei vertici adiacenti a quello appena estratto e all'occorrenza modificarlo. Il seguente esempio mostra come l'estrazione del vertice con peso minore (1, avente Key pari a 1) da Q (a destra del taglio) comporta la modifica della chiave del vertice 2, che passa da 3 a 2.



L'algoritmo

```

1 Prim(G, w, r)
2   Q ← V[G]
3   for each u ∈ V[G]
4     Key[u] ← ∞
5     π[u] ← NIL
6   Key[r] ← 0
7   while Q ≠ ∅
8     u ← Extract_Min(Q)
9     for each v ∈ Adj[u]
10       if v ∈ Q ∧ w(u, v) < Key[v]
11         Key[v] ← w(u, v)
12         π[v] ← u
13   return A = {(\π[v], v) ∈ E | v ∈ V \ {r}}

```

Correttezza L'algoritmo rispetta le condizioni del teorema fondamentale degli *MST*: innanzitutto, una volta estratto un vertice u , viene scelto come successivo il vertice v tale per cui l'arco (u, v) è leggero.

Inoltre, per via della distinzione tra Q e $V \setminus Q$, il taglio rispetta sicuramente A , dato dall'insieme degli archi leggeri che creano un albero tra i nodi di $V \setminus Q$; infatti, la definizione di A che diamo nell'istruzione di ritorno dell'algoritmo è corretta, ma se vogliamo essere più precisi, la formulazione di A in ogni istante dell'algoritmo è la seguente:

$$A = \{(\pi[u], u) \in E \mid u \in V \setminus \{r\} \setminus Q\}$$

Q è omesso nell'istruzione di ritorno dal momento che alla fine dell'algoritmo $Q = \emptyset$.

Notare inoltre che la notazione `Adj[u]` a riga 9 indica l'insieme dei vertici adiacenti al nodo estratto u .

Complessità Analizziamo una sezione alla volta.

1. **Inizializzazione (righe 2-6)** Il ciclo `for each` che inizializza i campi dei nodi li scorre tutti, compiendo quindi n iterazioni.
2. **Estrazione (righe 7-8)** Poiché il ciclo `while` di riga 7 itera finché Q non è vuoto, e poiché ad ogni iterazione viene estratto un nodo, allora il ciclo compie n iterazioni; inoltre, l'estrazione da una coda di minima priorità (che ricordiamo essere solitamente implementata con un *heap*) ha costo $\log(n)$ per eventuali riordinamenti interni all'*heap*. Quindi questa sezione ha costo $n \log(n)$.
3. **Aggiornamento campi (righe 9-13)** Essendo questo blocco di codice interno al ciclo `while` di riga 7, sorgerebbe spontaneo pensare che, qualunque sia la complessità di questa sezione, essa vada moltiplicata per il costo del `while`; tuttavia, queste porzioni sono tra loro indipendenti, in quanto il ciclo `for each` di riga 9 compie, per ogni $u \in V$, $\deg(u)$ iterazioni, per un totale di $\sum_{i=1}^n \deg(u_i)$ iterazioni, che per il lemma della stretta

di mano abbiamo visto essere pari a $2m$. Infine, va sottolineato anche il fatto che l'assegnazione a riga 11 non ha costo costante, bensì logaritmico rispetto a n , in quanto ad ogni aggiornamento delle chiavi si potrebbe dover riorganizzare l'ordine degli elementi che compongono l'*heap*. Dunque quest'ultima sezione ha costo $m \log(n)$.

Tirando le somme, abbiamo che

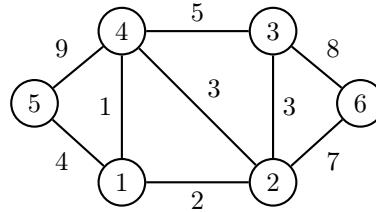
$$T(n, m) = \underbrace{n}_{\text{Init}} + \underbrace{n \log(n)}_{\text{Estrazione}} + \underbrace{m \log(n)}_{\text{Aggiornamento}}$$

E poiché il nostro grafo è connesso, vale che $m \geq n - 1$, quindi la funzione che domina è $m \log(n)$.

$$T(n, m) = O(m \log(n))$$

20.2.8 Esercizio d'esame - Compito del 06/06/2022

Si scriva l'algoritmo di Prim, si dimostri la sua correttezza (ovvero: si enunci e si dimostri il teorema fondamentale degli alberi di copertura minimi e si spieghi come questo garantisca la correttezza dell'algoritmo), si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sul seguente grafo utilizzando il vertice 1 come "sorgente":



In particolare: si indichi l'ordine con cui vengono estratti i vertici e si riempia la tabella seguente con i valori dei vettori key e π , iterazione per iterazione.

Soluzione Poiché algoritmi, teoremi, dimostrazioni e complessità sono riportati nelle pagine precedenti, ci limiteremo a completare la tabella che esplica l'esecuzione dell'algoritmo.

L'ordine di estrazione dei vertici è il seguente: $1 - 4 - 2 - 3 - 5 - 6$. Gli archi che compongono il *MST* risultato sono i seguenti: $(1, 4)$, $(1, 2)$, $(2, 3)$, $(1, 5)$ e $(2, 6)$.

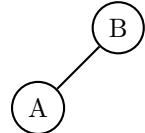
	Vertice 1		Vertice 2		Vertice 3		Vertice 4		Vertice 5		Vertice 6	
	key[1]	$\pi[1]$	key[2]	$\pi[2]$	key[3]	$\pi[3]$	key[4]	$\pi[4]$	key[5]	$\pi[5]$	key[6]	$\pi[6]$
Dopo inizializzazione	0	NIL	∞	NIL								
Iterazione 1	0	NIL	2	1	∞	NIL	1	1	4	1	∞	NIL
Iterazione 2	0	NIL	2	1	5	4	1	1	4	1	∞	NIL
Iterazione 3	0	NIL	2	1	3	2	1	1	4	1	7	2
Iterazione 4	0	NIL	2	1	3	2	1	1	4	1	7	2
Iterazione 5	0	NIL	2	1	3	2	1	1	4	1	7	2
Iterazione 6	0	NIL	2	1	3	2	1	1	4	1	7	2

20.3 Cammini minimi

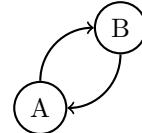
La sezione successiva affronta il problema di determinare i cammini minimi di grafi orientati, anche potenzialmente disconnessi, e pesati sugli archi, dove i pesi degli archi possono anche essere negativi.

Immaginiamo il caso di una ferrovia, in cui i nodi rappresentano le stazioni e gli archi rappresentano la distanza nello spazio o nel tempo tra le diverse stazioni. Gli algoritmi che studieremo si possono anche applicare a grafi non orientati, perché gli archi di un grafo non orientato possono essere espressi come archi di un grafo orientato:

Grafo non orientato



Grafo orientato



Definizione di cammino Ripetiamo la definizione di cammino:

Dati due vertici $u, v \in V$, un cammino tra u e v è una sequenza di vertici $p = < x_0, x_1, \dots, x_q > \in V : x_0 = u, x_q = v \text{ e } \forall i = 1, \dots, q, (x_{i-1}, x_i) \in E$.

Definiamo di conseguenza la **lunghezza di un cammino**:

$$w(p) = \sum_{i=1}^q w(x_{i-1}, x_i)$$

Inoltre, possiamo dare anche le definizioni di **insieme di cammini**:

$$\mathcal{C}(u, v) = \{p \mid p \text{ è un cammino tra } u \text{ e } v\}$$

e di **distanza** tra u e v :

$$\delta(u, v) = \begin{cases} \min_{p \in \mathcal{C}(u, v)} w(p) & \text{se } \mathcal{C}(u, v) \neq \emptyset \\ +\infty & \text{se } \mathcal{C}(u, v) = \emptyset \\ -\infty & \text{se } \mathcal{C}(u, v) \neq \emptyset \text{ ed } \exists \text{ cicli di costo negativo tra } u \text{ e } v \end{cases}$$

La cardinalità di $\mathcal{C}(u, v)$ è indicatore della distanza in quanto due nodi non raggiungibili da un cammino hanno effettivamente distanza infinita.

Nota La scelta di utilizzare $+\infty$ e $-\infty$ per indicare questi casi è una convenzione arbitraria.

20.3.1 Quattro tipi di problemi

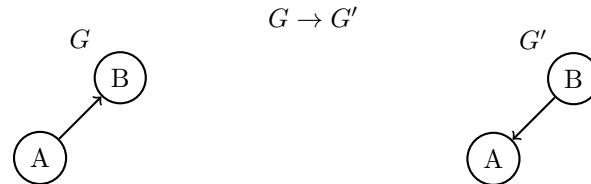
1. Dati $G = (V, E)$ orientato e pesato sugli archi e $u, v \in V$, si cerchi il cammino da u a v .
2. Dati $G = (V, E)$ orientato e pesato sugli archi e $s \in V$, si cerchi il cammino minimo da s a tutti gli altri vertici. Questo problema consiste in una generalizzazione del problema precedente.
3. Dati $G = (V, E)$ orientato e pesato sugli archi e $d \in V$, si cerchi il cammino minimo da tutti i vertici a d .
4. Dato $G = (V, E)$ orientato e pesato sugli archi, si cerchi il cammino minimo tra tutte le possibili coppie di nodi.

Possiamo schematizzare questo insieme di problemi nella seguente tabella:

Sorgente \ Destinazione	Singola	Multipla
Singola	In: $G = (V, E)$, $u, v \in V$ Out: $\delta(u, v)$	In: $G = (V, E)$, $s \in V$ Out: $\forall v \in V, \delta(s, v)$
Multipla	In: $G = (V, E)$, $d \in V$ Out: $\forall u \in V, \delta(u, d)$	In: $G = (V, E)$ Out: $\forall u, v \in V, \delta(u, v)$

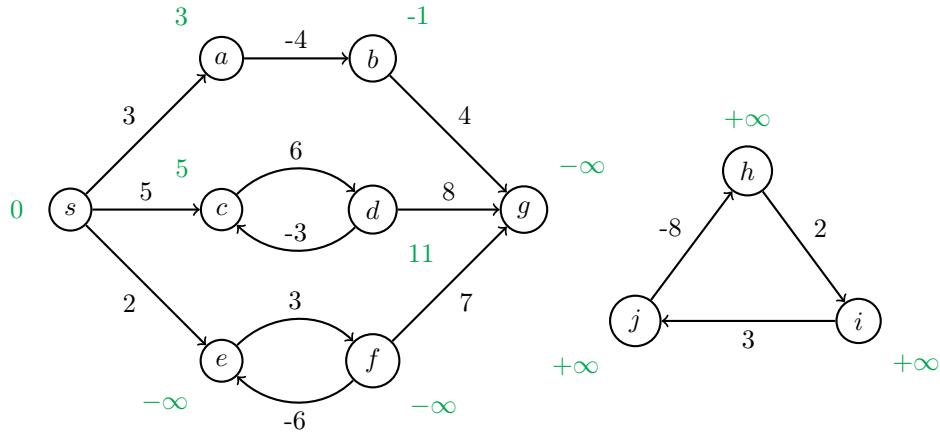
Chiaramente, in questi problemi con δ intendiamo anche il cammino minimo vero e proprio composto dall'insieme di vertici attraversati dal cammino, non solamente il valore della lunghezza di tale cammino.

Possiamo notare che il problema con sorgente multipla e destinazione singola può essere risolto con lo stesso algoritmo risolutivo del problema con sorgente singola e destinazione multipla: è infatti sufficiente invertire la direzione degli archi.



Per questo motivo, questa versione del problema non verrà studiata; in particolare, studieremo quasi esclusivamente le versioni con sorgente singola e destinazione multipla e con sorgenti e destinazioni multiple: questo perché non esistono algoritmi risolutivi asintoticamente più efficienti di quelli che risolvono i suddetti problemi che risolvano anche quelli che abbiamo deciso di ignorare; infatti, sebbene la prima variante del problema ritorni un'unica soluzione invece che un insieme, per trovarla è comunque necessario attraversare tutto il grafo, di conseguenza conviene studiare prevalentemente le soluzioni "più complesse".

Vediamo un esempio di grafo e del calcolo della distanza tra i suoi nodi. Sia s il nodo sorgente: a fianco ad ogni nodo, è riportata in verde la distanza tra la sorgente e quel nodo.



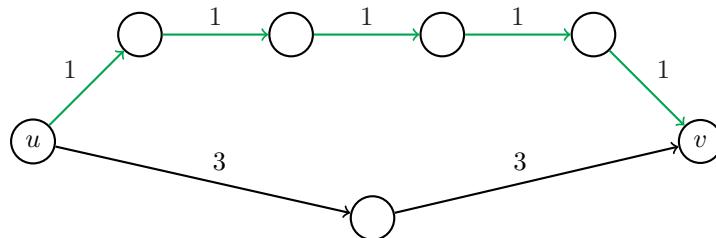
- Il nodo sorgente ha distanza 0 da sé stesso.
- I nodi a e b sono raggiungibili dalla sorgente attraverso un solo cammino, per cui avranno distanza, rispettivamente, 3 e $3 - 4 = -1$.
- Il nodo c si trova in un ciclo: esistono infiniti cammini tra s e c (*e.g.*, $< s, c >$, $< s, c, d, c >$, $< s, c, d, c, d, c >$...); dal momento che ogni volta che attraversiamo il ciclo la distanza aumenta, diremo che il cammino minimo è quello avente lunghezza 5 ($< s, c >$). Possiamo fare lo stesso discorso per il nodo d , per cui la sua distanza dalla sorgente è $5 + 6 = 11$.
- Sembra che i nodi e e f presentino le stesse condizioni dei nodi precedenti, tuttavia la lunghezza del cammino ad ogni attraversamento del ciclo che si crea tra tali nodi diminuisce sempre di più: quando i vertici si trovano in un ciclo con peso negativo, per convenzione diremo che la distanza tra la sorgente e tali nodi è $-\infty$. A causa di questo ciclo negativo, anche la distanza tra s e g è pari a $-\infty$.
- Per quanto riguarda i vertici h , i e j , potremmo pensare che anche la loro distanza dalla sorgente sia $-\infty$ per via del ciclo di peso negativo che si crea tra gli archi che li collegano, tuttavia, dal momento che questo ciclo non è raggiungibile dalla sorgente attraverso nessun cammino, la loro distanza da s è pari a $+\infty$.

20.3.2 Panoramica sugli algoritmi

- **Dijkstra** Sicuramente uno dei temi più celebri nel campo dell'algoritmica, l'algoritmo di *Dijkstra* permette di risolvere il problema dei cammini minimi con sorgente singola. Tuttavia, l'algoritmo funziona esclusivamente in grafi aventi archi con **pesi positivi**.
- **Bellman-Ford** Come il precedente, anche l'algoritmo di *Bellman-Ford* risolve il problema dei cammini minimi con sorgente singola, e in aggiunta il suo funzionamento è garantito nel caso di pesi negativi (ma non nel caso di cicli negativi).
- **Floyd-Warshall** Tale algoritmo risolve il problema dei cammini minimi tra tutte le coppie del grafo.

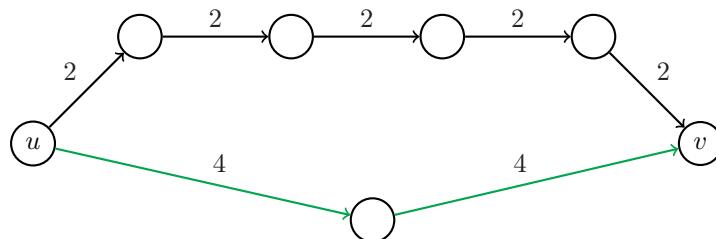
Alla luce delle osservazioni fatte sul funzionamento degli algoritmi di ricerca di cammini minimi, potremmo pensare che sia possibile adattare l'algoritmo di *Dijkstra* a tutti i grafi, anche quelli aventi pesi negativi, *shiftando* il valore di tutti gli archi in modo da avere unicamente archi con peso positivo.

Questa soluzione tuttavia è incorretta, e per dimostrarlo è sufficiente trovare un controesempio.



In questo caso, il cammino minimo attraversa tutti i vertici collegati dagli archi di peso 1, per cui $\delta(u, v) = 5$.

Vediamo cosa succede sommando 1 al peso di tutti gli archi.



Possiamo notare che a seguito di questa traslazione il cammino minimo cambia, e ora $\delta(u, v) = 8$. Questo problema non ci ha causato problemi nello studio degli alberi di copertura minimi perché il numero di archi è costante e i pesi negativi non erano un problema per la ricerca degli alberi di copertura.

Quando risolviamo questi problemi, i risultati che vogliamo ottenere sono due:

1. Il valore della lunghezza del cammino minimo.
2. L'insieme dei vertici attraversati dal cammino minimo.

A tale fine, memorizziamo due campi per ogni vertice $u \in V$:

1. **Stima di cammino minimo** $d[u]$ - Un campo numerico che si evolve nel tempo e ci consente di capire la distanza tra due vertici.
Alla fine dell'algoritmo deve valere che $d[u] = \delta(s, u)$, cioè la stima della distanza deve essere pari alla distanza vera.
2. **Predecessore** $\pi[u]$ - Un puntatore ad un altro vertice, che ci serve per ricostruire l'insieme dei vertici del cammino.

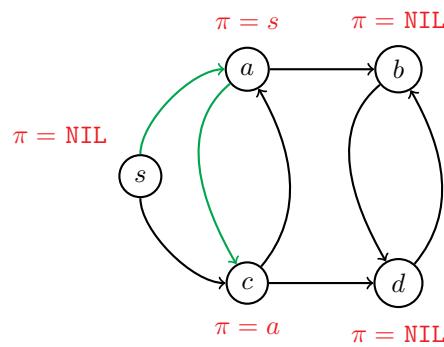
Grafo dei predecessori G_π Il grafo dei predecessori $G_\pi = (V_\pi, E_\pi)$ è un sottografo di $G = (V, E)$ che dipende dal campo predecessore dei vertici di G . È un grafo che si evolve nel tempo man mano che l'algoritmo procede, per cui possiamo immaginarlo come una "istantanea" del risultato dell'algoritmo.

Gli insiemi V_π e E_π sono definiti nel seguente modo:

1. $V_\pi = \{u \in V : \pi[u] \neq \text{NIL}\} \cup \{s\}$
2. $E_\pi = \{(\pi[u], u) \in E : u \in V_\pi \setminus \{s\}\}$

Il grafo dei predecessori è quindi un grafo in cui sono presenti solamente i vertici che hanno un predecessore ben definito e i corrispondenti archi col predecessore, i quali formano un cammino minimo.

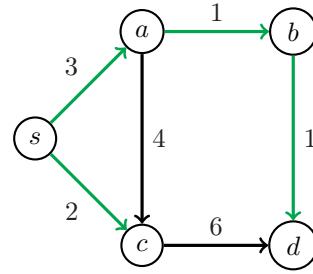
Vediamo un esempio: nel grafo in questione, è stato impostato il predecessore dei vertici a e c , per cui varrà che $V_\pi = \{s, a, c\}$ e $E_\pi = \{(s, a), (a, c)\}$.



Albero dei cammini minimi G' Un albero dei cammini minimi è un grafo $G' = (V', E')$, sottografo di $G = (V, E, w)$, con $w : E \rightarrow \mathbb{R}$, $s \in V$ e $E' \subseteq E$, dove valgono le seguenti condizioni:

- $V' = \{u \in V \mid \delta(s, u) < +\infty\}$, cioè i vertici del grafo sono raggiungibili dalla sorgente.
- G' forma un albero con radice s .
- Per ogni vertice $u \in V'$, l'unico cammino tra s e u in G' è un cammino minimo tra s e u in G .

Vediamo un esempio: nel grafo di cui sotto, sono evidenziati in verde gli archi che costituiscono l'albero dei cammini minimi.



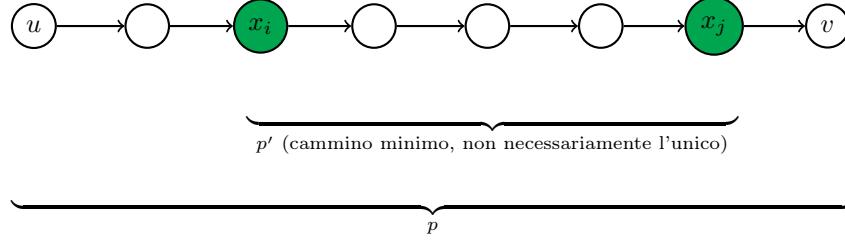
Alla fine del nostro algoritmo, vogliamo che G_π sia un albero di cammini minimi.

20.3.3 Proprietà dei sottocammini minimi

La proprietà dei sottocammini minimi afferma che sottocammini di cammini minimi sono minimi.

Dimostrazione Sia $p = < x_0, \dots, x_q >$ un cammino minimo tra i nodi u e v , dove quindi $x_0 = u$ e $x_q = v$. Sia $p' = < x_i, \dots, x_q >$, $0 \leq i \leq j \leq q$, un sottocammino di p .

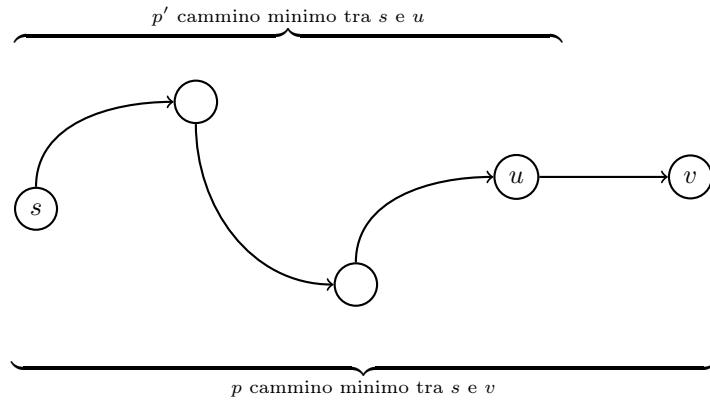
Supponiamo per assurdo che p' non sia un cammino minimo: allora, neanche p sarebbe un cammino minimo, perché vuol dire che esisterebbe un altro cammino minimo tra u e v . Quindi anche p' è un cammino minimo tra x_i e x_j .



20.3.4 Proprietà

Sia $G = (V, E)$ un grafo con $s, u, v \in V$, tale che u è il penultimo nodo del cammino minimo tra s e v . Allora, vale che

$$\delta(s, v) = \delta(s, u) + w(u, v)$$



Dimostrazione Sappiamo che

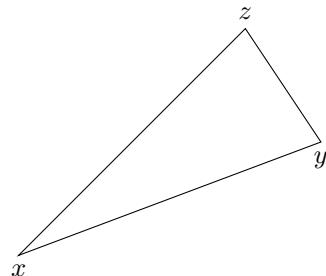
$$\begin{aligned} \delta(s, v) &= w(p) \\ &= w(p') + w(u, v) \\ &= \delta(s, u) + w(u, v) \end{aligned}$$

20.3.5 Disuguaglianza triangolare

Sia $G = (V, E)$ un grafo con $s \in V$, $(u, v) \in E$. Allora,

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Possiamo interpretare questa proprietà facendo riferimento alla disuguaglianza triangolare della geometria euclidea che riguarda le distanze tra i vertici di un triangolo degenere:



$$d(x, z) \leq d(x, y) + d(y, z)$$

Dimostrazione Distinguiamo tre casi:

- a) $\delta(s, u) = +\infty$, cioè non esiste un cammino tra s e u . Allora, varrà che $\delta(s, v) \leq +\infty + w(u, v)$, e siccome $w(u, v) \in \mathbb{R}$, allora la condizione è banalmente vera.
- b) $\delta(s, u) = -\infty$, cioè esiste un ciclo negativo tra s e u . Allora, varrà che $\delta(s, v) \leq -\infty + w(u, v)$, ma quindi anche $\delta(s, v) = -\infty$ per via del ciclo negativo. Dunque la condizione è verificata.
- c) $\delta(s, u) \in \mathbb{R}$, cioè esiste un cammino senza cicli negativi tra s e u . Questo è il caso più "standard": sia p il cammino da s a u ; allora, se l'unico cammino tra s e v è costituito dall'unione di p e dell'arco (u, v) , allora il costo di tale cammino è esattamente pari alla somma del costo di p e del costo di (u, v) . È infatti possibile che esistano altri cammini da s a v , che comprendono o meno (parti di) p , il cui costo può essere minore di $w(p) + w(u, v)$, e di conseguenza $w(p) + w(u, v)$ costituisce un limite superiore per la distanza tra s e v .

20.3.6 Funzioni ausiliarie degli algoritmi

Come abbiamo visto, gli algoritmi di ricerca del cammino minimo utilizzano due strutture dati ausiliarie per memorizzare la stima di cammino minimo e il predecessore di ogni nodo.

Vediamo due procedure ausiliarie che si occupano rispettivamente dell'inizializzazione e dell'aggiornamento di tali strutture dati.

Init_SS (Single Source)

```

1 Init_SS(G, s)
2   for each u ∈ V[G]
3     d[u] = +∞
4     π[u] = NIL
5   d[s] = 0

```

La procedura inizializza il campo d di ogni vertice a $+\infty$ (eccetto per la radice, il cui campo viene inizializzato a 0) e il campo π di ogni vertice a NIL .

Il tempo d'esecuzione di questa procedura è lineare.

Relax

```

1 Relax(u, v, w(u, v))
2   if d[v] > d[u] + w(u, v) then
3     d[v] = d[u] + w(u, v)
4     π[v] = u

```

La procedura aggiorna i campi d e π del vertice destinazione dell'arco passato come parametro, qualora si verificasse la condizione descritta dalla proprietà della disuguaglianza triangolare.

Il tempo d'esecuzione della procedura dipende dall'implementazione delle strutture dati, e verrà discusso in seguito.

Proprietà della Relax Alla fine di una `Relax(u, v, w(u, v))`, varrà sempre che

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

Attenzione a non confondere questa proprietà con la diseguaglianza triangolare, perché questa proprietà si basa sulle stime, e non sulla vera distanza tra s e v .

20.3.7 Proprietà del limite inferiore (o superiore)

La seguente proprietà, definita "*Proprietà del limite inferiore*" dal professor Pelillo e "*Proprietà del limite superiore*" da alcuni libri di testo, afferma la veridicità della seguente condizione:

$$\delta(s, v) \leq d[v], \quad \forall v \in V$$

Significa che qualunque sequenza di `Relax` noi eseguiamo, la vera distanza tra s e v costituirà sempre un limite inferiore per la stima della distanza tra s e v . Inoltre, se $\delta(s, v) = d[v]$, nessuna `Relax` potrà cambiare il valore di $d[v]$.

L'ambiguità sul nome di questa proprietà deriva dal fatto che possiamo considerare la vera distanza tra i due vertici come un limite inferiore per la stima della distanza, o, al contrario, la stima per la distanza come un limite superiore per la vera distanza.

Dimostrazione Distinguiamo due momenti dell'esecuzione dell'algoritmo:

1. Subito dopo la `Init_SS`,

- se $v \neq s$, varrà che $d[v] = +\infty \geq \delta(s, v)$, che verifica banalmente la proprietà;
- se $v = s$, sappiamo che $\delta(s, s) = 0$ se non esistono cicli negativi che attraversano la sorgente, altrimenti $\delta(s, s) = -\infty$.
In entrambi i casi, $\delta(s, s) \leq 0 = d[s]$.

Quindi subito dopo l'inizializzazione la proprietà è sempre vera.

2. Supponiamo per assurdo che v sia il primo vertice per cui la proprietà è violata, cioè ci troviamo in questa situazione:



Al termine di questa `Relax`, varrà che

$$\begin{aligned} d[u] + w(u, v) &= d[v] \\ &< \delta(s, v) && \text{per ipotesi} \\ &\leq \delta(s, u) + w(u, v) && \text{per proprietà triangolare} \end{aligned}$$

Quindi, vale che $d[u] + w(u, v) < \delta(s, u) + w(u, v) \Rightarrow d[u] < \delta(s, u)$, ma questo è assurdo, perché allora v non sarebbe il primo vertice che infrange la proprietà, ma sarebbe u .

20.3.8 Proprietà dell'assenza di cammino

Se non esiste un cammino tra s e u , cioè $\delta(s, u) = +\infty$, allora immediatamente dopo la `Init_SS`, $\delta(s, u) = d[u]$.

20.3.9 Proprietà della convergenza

In un cammino minimo $p = < s, \dots, u, v >$ per una qualche coppia di vertici $u, v \in V$, supponiamo che, ad un certo punto (cioè dopo una certa sequenza di `Relax`), $d[u] = \delta(s, u)$. Allora, chiamare `Relax(u, v, w(u, v))` farà sì che $d[v] = \delta(s, v)$.

Dimostrazione

$$\begin{aligned}
 \delta(s, v) &\leq d[v] && \text{per la proprietà del limite inferiore} \\
 &\leq d[u] + w(u, v) && \text{per la proprietà della Relax} \\
 &= \delta(s, u) + w(u, v) && \text{per ipotesi} \\
 &= \delta(s, v) && \text{perché siamo su un cammino minimo}
 \end{aligned}$$

20.3.10 Proprietà del grafo dei predecessori

Se alla fine di un algoritmo di ricerca dei cammini minimi vale che $d[u] = \delta(s, u) \forall u \in V$, allora G_π è un albero di cammini minimi.

20.3.11 Algoritmo di Dijkstra

```

1 Dijkstra(G, w, s)
2   Init_SS(G, s)
3   Q ← V[G]
4   S ← ∅
5   while Q ≠ ∅
6     u ← Extract_Min(Q)
7     S ← S ∪ {u}
8     for each v ∈ Adj[u]
9       Relax(u, v, w(u, v))
10  return (d, Gπ)

```

Spiegazione Dijkstra mantiene una coda di priorità Q in cui memorizza le informazioni del campo d per ciascun vertice del grafo G . Questo tipo di struttura dati è necessario affinché i vertici possano essere estratti uno ad uno in ordine crescente di d .

L'algoritmo fa anche uso di un insieme S , dove vengono memorizzati senza un ordine particolare i vertici già estratti: in ogni istante, vale che $Q = V \setminus S$.

Le righe 2-4 costituiscono l'inizializzazione dell'algoritmo: qui avviene la chiamata alla procedura ausiliaria `Init_SS` e viene dato il valore iniziale alle strutture Q e S .

Nelle righe 5-7 viene estratto un vertice da Q e aggiunto in S , mentre nelle righe 8-9 vengono iterati tutti i nodi adiacenti a quello appena estratto e viene compiuta la procedura di `Relax`.

Complessità

- Righe 2-4: la procedura `Init_SS` ha tempo d'esecuzione $\Theta(n)$, per cui la fase di inizializzazione ha costo lineare.
- Righe 5-7: il ciclo `while` di riga 5 compie n iterazioni, una per ogni nodo estratto, e l'operazione di estrazione da Q dipende dall'implementazione di Q : se Q è implementata con un **array lineare**, l'estrazione del minimo ha costo lineare; se invece Q è implementata con un **heap binario**, allora la `Extract_Min` ha costo logaritmico.
- Righe 8-9: il ciclo `for each` di riga 8 compie tante iterazioni quanto è l'*out-degree* del nodo appena estratto u . Quando abbiamo introdotto il concetto di *out-degree*, abbiamo osservato che

$$\sum_{i=1}^n \text{out-deg}(i) = m$$

Dunque il ciclo `for each` di riga 8 compie in totale m iterazioni, a prescindere dal fatto che sia un ciclo innestato nel `while` di riga 5.

Come abbiamo già anticipato, il tempo d'esecuzione della procedura `Relax` dipende dall'implementazione di Q : se è un array lineare, la sua complessità è costante, mentre se è un heap binario la sua complessità è logaritmica, in quanto può comportare una riorganizzazione della struttura interna.

Concludiamo esprimendo la complessità dell'algoritmo di Dijkstra in base all'implementazione della coda di priorità:

- **Array lineare**

$$T(n, m) = \underbrace{n}_{\text{Init}} + \underbrace{n^2}_{\text{Extract}} + \underbrace{c \cdot m}_{\text{Relax}} = O(n^2)$$

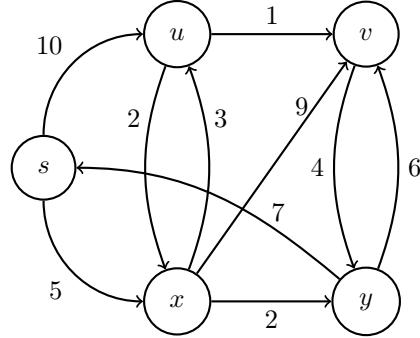
- **Heap binario**

$$T(n, m) = \underbrace{n}_{\text{Init}} + \underbrace{n \log n}_{\text{Extract}} + \underbrace{m \log n}_{\text{Relax}} = O(m \log n)$$

La funzione $m \log n$ domina sulla funzione $n \log n$ perché, con l'assunzione che il grafo sia connesso, vale che $m \geq n - 1$.

Non c'è una risposta netta su quale sia la migliore implementazione per la coda di priorità, ma in linea di massima possiamo affermare che quando il grafo è denso (cioè $m \approx n^2$) può risultare più conveniente usare un array, mentre se il grafo è sparso (cioè $m \approx n$) un heap potrebbe portare a prestazioni più elevate.

Esempio Sia dato il seguente grafo $G = (V, E)$, e vogliamo applicare su di esso l'algoritmo di Dijkstra usando il vertice s come sorgente.



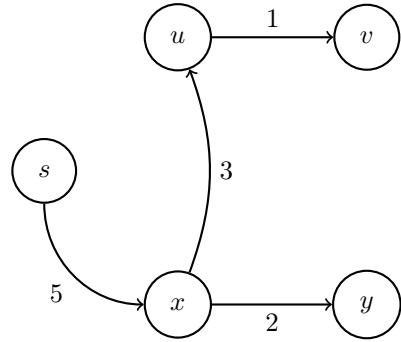
A seguito della chiamata a `Init_SS`, i valori d e π per ogni vertice varranno rispettivamente 0 e `NIL` per la radice s e $+\infty$ e `NIL` per gli altri vertici.

1. **Estrazione di s :** poiché l'algoritmo estrae il vertice avente campo d minimo, subito dopo l'inizializzazione solamente la radice avrà campo d diverso da $+\infty$, quindi viene estratta e vengono considerati i suoi archi uscenti verso i nodi u e x : attraverso le `Relax`, varrà che $d[x] = 5$, $\pi[x] = s$, $d[u] = 10$ e $\pi[u] = s$.
2. **Estrazione di x :** a questo punto, x è il vertice che non è ancora stato estratto avente campo d minimo, per cui lo estraiamo e valutiamo i suoi archi uscenti verso i vertici u , v e y : dopo le `Relax`, varrà che $d[u] = 8$, $\pi[u] = x$, $d[v] = 14$, $\pi[v] = x$, $d[y] = 7$ e $\pi[y] = x$.
3. **Estrazione di y :** estraiamo il vertice y perché $d[y] = 7$; questo nodo ha archi uscenti verso la radice e verso il vertice v : non abbiamo motivo di chiamare la `Relax` sulla sorgente, e più in generale non abbiamo motivo di chiamarla su un vertice che è già stato estratto, perciò la chiamiamo solamente sul vertice v : varrà che $d[v] = 13$ e $\pi[v] = y$.
4. **Estrazione di u :** viene ora estratto il nodo vertice, il cui campo d è rimasto pari a 8, invariato da quando è stato estratto x : per lo stesso motivo del nodo precedente, non applicheremo la `Relax` sul vertice x perché già stato estratto, e ci limitiamo ad applicarla nuovamente sul vertice v : in seguito a tale chiamata, varrà che $d[v] = 9$ e $\pi[v] = u$.
5. **Estrazione di v :** l'ultimo nodo rimasto da estrarre è il nodo v : lo estraiamo, ma non avendo senso applicare delle `Relax` su nodi già estratti, in questa iterazione l'algoritmo non fa nulla.

L'algoritmo è terminato e i seguenti sono i valori finali delle strutture dati ausiliarie, nell'ordine di estrazione dei nodi:

Nodo	s	x	y	u	v
$d[p]$	0	5	7	8	9
$\pi[p]$	NIL	s	x	x	u

Notare che G_π è un albero dei cammini minimi:



20.3.12 Dijkstra - Correttezza

Sia $G = (V, E)$ un grafo orientato pesato sugli archi, cioè con $w : E \rightarrow \mathbb{R}$ tale che $\forall (u, v) \in E : w(u, v) \geq 0$.

Allora, alla fine dell'algoritmo di Dijkstra, si ha:

1. $\forall v \in V : d[v] = \delta(s, v)$
2. G_π è un albero di cammini minimi

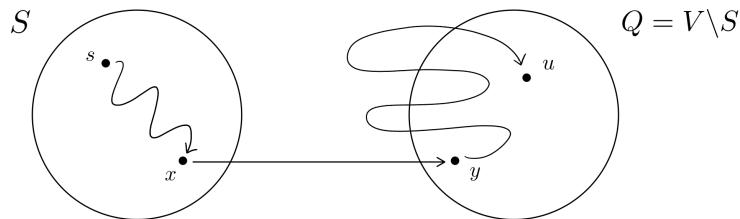
Dimostreremo la correttezza del primo punto attraverso la dimostrazione di un'affermazione più forte ma equivalente, cioè che per ogni $u \in V$, al momento dell'estrazione di u , risulta $d[u] = \delta(s, u)$.

La dimostrazione avviene per assurdo e si avvale di 9 osservazioni.

Dimostrazione Supponiamo per assurdo che esista un vertice $u \in V$ tale che al momento della sua estrazione $d[u] \neq \delta(s, u)$, e che u sia il primo vertice per cui questo accade.

Osservazioni

1. u non può essere la sorgente: dopo la `Init_SS`, $d[s] = 0 = \delta(s, s)$, e $\delta(s, s)$ non può valere $-\infty$ perché per ipotesi non ci sono archi con pesi negativi, quindi neanche cicli negativi.
2. Al momento dell'estrazione di u , $S \neq \emptyset$, perché in S ci sarà almeno la sorgente s .
3. u non è irraggiungibile dalla sorgente: se così fosse, $\delta(s, u) = +\infty = d[u]$, che sarebbe corretto e non violerebbe la proprietà che abbiamo voluto violare per assurdo; quindi, $\delta(s, u) \neq +\infty$.
4. Ci poniamo nell'istante in cui s è già stato estratto ($s \in S$) ma u no ($u \in Q = V \setminus S$). Per il punto precedente, esiste un cammino minimo p tra s e u : sia (x, y) un arco appartenente a p che attraversa il taglio, cioè tale che $x \in S$ e $y \in Q$.



5. Per ipotesi, vale che $d[x] = \delta(s, x)$: infatti, u è il primo vertice per cui questa proprietà non vale.

6. Poiché siamo su un cammino minimo, possiamo applicare la proprietà della convergenza: al momento dell'estrazione di x , applichiamo la **Relax** su y e otteniamo che $d[y] = \delta(s, x) + w(x, y) = \delta(s, y)$.
7. Dal momento che stiamo per estrarre il nodo u , siccome l'algoritmo di Dijkstra estrae il vertice avente campo d più piccolo, dovrà valere che $d[u] \leq d[y]$.
8. Non può accadere che $\delta(s, y) > \delta(s, u)$: in virtù del fatto che ci troviamo in un cammino minimo e che i pesi sono tutti maggiori o uguali a zero, allora $\delta(s, y) \leq \delta(s, u)$.
9. Per la proprietà del limite inferiore, vale sicuramente che $\delta(s, u) \leq d[u]$.

Ricostruiamo l'assurdo sulla base delle precedenti osservazioni:

$$\begin{aligned}
 \delta(s, u) &\leq d[u] && \text{per oss. 9} \\
 &\leq d[y] && \text{per oss. 7} \\
 &= \delta(s, y) && \text{per oss. 6} \\
 &\leq \delta(s, u) && \text{per oss. 8}
 \end{aligned}$$

Avvalendoci delle osservazioni, siamo riusciti a "rinchiudere" il valore di $d[u]$ tra $\delta(s, u)$ e $\delta(s, u)$:

$$\delta(s, u) \leq d[u] \leq \delta(s, u) \Rightarrow d[u] = \delta(s, u)$$

che è assurdo in quanto andiamo ad invalidare l'ipotesi che $d[u] \neq \delta(s, u)$.

20.3.13 Dijkstra - Ottimizzazioni

È possibile applicare delle ottimizzazioni all'algoritmo di Dijkstra per migliorarne il tempo d'esecuzione; si tratta tuttavia di ottimizzazioni che vanno a incidere strettamente sul *CPU-time*, non andremo a modificare la complessità asintotica dell'algoritmo.

- È possibile porre una condizione per limitare l'uso della **Relax** ai vertici che non sono ancora stati estratti, cioè che non appartengono ad S , in quanto sappiamo che una volta estratto, la stima della loro distanza non potrà essere abbassata ulteriormente.
- Analogamente al punto precedente, si può interrompere l'algoritmo prima dell'estrazione dell'ultimo vertice: nel momento in cui viene estratto quest'ultimo, tutti gli altri apparterranno a S e non verrà eseguita nessuna **Relax**, perciò in questo modo possiamo risparmiare un ciclo d'esecuzione.

20.3.14 Dijkstra - Ulteriori osservazioni ed esercizi

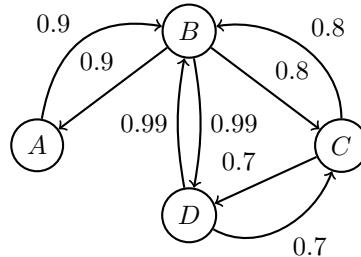
Cappi - Ragionamento personale La presenza di cappi non dovrebbe intaccare il funzionamento dell'algoritmo di Dijkstra, e possiamo provarlo sulla base del peso del cappio.

- Se il peso del cappio è negativo, non è un problema di competenza dell'algoritmo di Dijkstra, in quanto sappiamo che il suo funzionamento non è garantito in presenza di archi con pesi negativi.
- Se il peso del cappio è positivo, possiamo arbitrariamente scartarlo, in quanto nessun cammino minimo potrà mai attraversarlo: sia v il vertice in questione, e (v, v) il cappio; allora, tutti i vertici raggiungibili da v dopo aver attraversato (v, v) sono raggiungibili anche prima di attraversarlo: decidendo di attraversarlo, si aumenterebbe solamente il peso del cammino.
- Se il peso del cappio è 0, l'algoritmo potrebbe creare infiniti cammini minimi, in quanto l'attraversamento del cappio un numero indefinito di volte non aumenta il peso dei cammini. Per una questione di semplicità e per evitare errori nella creazione di G_π , conviene rimuoverlo.

Bisogna anche considerare il fatto che nel momento in cui estraiamo un nodo non ha senso cercare di compiere la Relax su sé stesso, in quanto abbiamo dimostrato che un nodo che è già stato estratto non può abbassare ulteriormente la stima della sua distanza.

Esercizio Sia $G = (V, E)$ un grafo orientato. Vale che $\forall (u, v) \in E, r(u, v) \in [0, 1]$ rappresenta il valore di affidabilità di un canale di comunicazione (l'arco (u, v)) tra una stazione e l'altra (i vertici u e v); in altre parole, $r(u, v)$ è la probabilità che un messaggio venga trasmesso con successo da u a v .

Assumendo che queste probabilità siano statisticamente indipendenti tra di loro, vogliamo trovare un algoritmo che dati due vertici ci ritorni il cammino avente affidabilità massima.



Si tratta di un problema di massimizzazione che possiamo riformulare nei termini del problema dei cammini minimi: dato un cammino, *e.g.* $p = < A, B, C >$, la sua affidabilità si può vedere come il prodotto dell'affidabilità degli archi che lo costituiscono; indichiamo l'affidabilità di p con $\alpha(p)$.

Il problema ci chiede di trovare, tra tutti i possibili cammini $p = < x_0 \ x_1 \ \dots \ x_q > \in \mathcal{C}(u, v)$, con $x_0 = u$ e $x_q = v$, quello con affidabilità massima, cioè

$$\max_{p \in \mathcal{C}(u, v)} \prod_{i=1}^q r(x_{i-1}, x_i)$$

Possiamo notare che anche il problema dei cammini minimi si può esprimere attraverso una formulazione molto simile: attraverso l'algoritmo di Dijkstra, noi troviamo, tra tutti i cammini, quelli aventi somma dei pesi sugli archi minima, cioè

$$\min_{p \in \mathcal{C}(u, v)} \sum_{i=1}^q r(x_{i-1}, x_i)$$

Applichiamo alcune trasformazioni. Sappiamo che invece di massimizzare una funzione $f(x)$, possiamo ottenere lo stesso risultato massimizzando $g(f(x))$, purché $g(x)$ sia crescente; sfruttiamo la funzione logaritmica, e massimizziamo la seguente funzione:

$$\max_{p \in \mathcal{C}(u, v)} \log \prod_{i=1}^q r(x_{i-1}, x_i) = \max_{p \in \mathcal{C}(u, v)} \sum_{i=1}^q \log r(x_{i-1}, x_i)$$

e invertiamo il segno per ottenere il minimo:

$$\min_{p \in \mathcal{C}(u, v)} - \sum_{i=1}^q \log r(x_{i-1}, x_i)$$

che, attraverso le proprietà del logaritmo, diventa

$$\min_{p \in \mathcal{C}(u, v)} \sum_{i=1}^q \log \left(\frac{1}{r(x_{i-1}, x_i)} \right)$$

Dal momento che l'affidabilità di ogni arco è positiva, siamo sicuri che il logaritmo del reciproco dell'affidabilità è una quantità positiva, quindi possiamo costruire un nuovo grafo G' avente gli stessi vertici e gli stessi archi di G ma con funzione peso $w(u, v) = \log \left(\frac{1}{r(u, v)} \right)$. In questo modo, possiamo applicare liberamente l'algoritmo di Dijkstra così come lo conosciamo e risolvere il problema.

20.3.15 Algoritmo di Bellman-Ford

Come abbiamo visto, la correttezza dell'algoritmo di Dijkstra è garantita solamente in presenza di pesi maggiori o uguali a 0, dal momento che un arco di peso negativo potrebbe diminuire la lunghezza del cammino di un nodo già estratto dalla coda con priorità, e abbiamo dimostrato che non è possibile cercare di applicare il suddetto algoritmo in presenza di pesi negativi *shiftandoli* di una costante.

In caso di pesi negativi, una valida alternativa a Dijkstra è l'algoritmo di Bellman-Ford: anche quest'ultimo è un algoritmo che risolve il problema dei cammini minimi con sorgente singola, che usa le stesse strutture dati usate di Dijkstra (d e π) e basa la ricerca dei cammini minimi sull'operazione di `Relax`. Tuttavia, la struttura dell'algoritmo è completamente diversa: se, da un lato, algoritmi come quello di Kruskal, Prim e Dijkstra si possono definire *greedy*, in quanto intraprendono la strada che sembra essere la più promettente in quel momento, dall'altro lato, algoritmi come quello di Bellman-Ford sono algoritmi di "forza bruta": tale algoritmo, infatti, fa $n - 1$ passate a tappeto sul grafo, e in ciascuna di queste rilassa tutti gli archi.

L'algoritmo

```

1 Bellman-Ford(G, w, s)
2   Init_SS(G, s)
3   for i = 1 to |V[G]| - 1
4     for each (u, v) ∈ E[G]
5       Relax(u, v, w(u, v))
6   for each (u, v) ∈ E[G]
7     if d[v] > d[u] + w(u, v) then
8       return (FALSE, d, π)
9   return (TRUE, d, π)
```

Spiegazione Possiamo dividere l'algoritmo in tre sezioni distinte.

- **Inizializzazione - Riga 2**

In questa fase, avviene l'inizializzazione delle strutture dati in modo analogo a quanto accade in Dijkstra.

- **Fase di Relax - Righe 3-5**

In questa sezione l'algoritmo compie le operazioni di `Relax` sugli archi di G . Sono presenti due cicli innestati: il primo compie $n - 1$ iterazioni, il secondo itera su tutti gli archi di G , e quindi esegue un totale di m iterazioni, in ciascuna delle quali rilassa l'arco corrente.

Se non fosse per la presenza di cicli negativi, l'algoritmo potrebbe concludersi qui.

- **Controllo e ritorno - Righe 6-9**

Questo ulteriore ciclo sugli archi di G avviene per controllare la corretta esecuzione dell'algoritmo: se l'`if-statement` di riga 7 dovesse risultare

verificato, significherebbe che l'algoritmo ha trovato dei cicli negativi all'interno del grafo, e lo segnala all'utente ritornando le strutture dati insieme ad una flag FALSE, non garantendo dunque la correttezza di d e π . Se invece l'`if-statement` non dovesse mai essere soddisfatto, l'algoritmo ritorna le strutture dati e la flag TRUE, indice del fatto che possiamo fidarci dei risultati ottenuti.

Analisi della complessità Come sappiamo da Dijkstra, l'operazione di inizializzazione viene eseguita in tempo lineare rispetto a n ; abbiamo appena verificato che la fase di `Relax` compie in totale $(n - 1) \cdot m$ cicli e l'ultima sezione, dovendo iterare su tutti gli archi, esegue m iterazioni.

Il tempo d'esecuzione sarà dunque il seguente:

$$T(n, m) = \Theta \left(\underbrace{n}_{\text{Init}} + \underbrace{(n - 1) \cdot m}_{\text{Relax}} + \underbrace{m}_{\text{Check}} \right) = \Theta(n \cdot m)$$

Dijkstra e Bellman-Ford a confronto

- Quante passate di `Relax` effettua Dijkstra? m
- Quante passate di `Relax` effettua Bellman-Ford? $(n - 1) \cdot m$

	Sparso ($m \approx n$)	Denso ($m \approx n^2$)
Dijkstra (<i>array</i>)	n^2	n^2
Dijkstra (<i>heap</i>)	$n \log n$	$n^2 \log n$
Bellman-Ford	n^2	n^3

20.3.16 Bellman-Ford - Correttezza

Si esegua l'algoritmo di Bellman-Ford su un grafo $G = (V, E)$, con $w : E \rightarrow \mathbb{R}$ e sorgente $s \in V$. Se in G non ci sono cicli negativi raggiungibili da s , allora alla fine dell'algoritmo varrà che:

- a) $d[u] = \delta(s, u) \quad \forall u \in V$
- b) G_π è un albero di cammini minimi
- c) L'algoritmo restituisce TRUE

Se invece in G esiste un ciclo negativo raggiungibile da s , allora l'algoritmo restituisce FALSE.

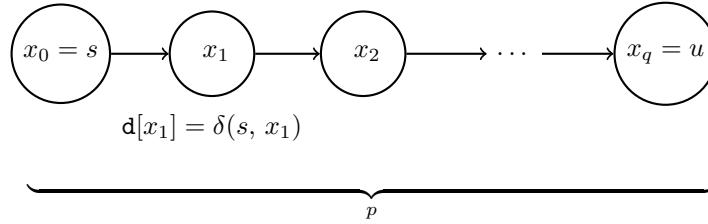
Dimostreremo i punti a) e c) e il fatto che l'algoritmo restituisce FALSE in presenza di cicli negativi, mentre non dimostreremo il punto b) (come non abbiamo fatto neanche per Dijkstra, essendo b), in Dijkstra come in Bellman-Ford, diretta conseguenza di a)).

Dimostrazione a) Sappiamo che

$$\forall u \in V, \quad \delta(s, u) = \begin{cases} +\infty \\ \in \mathbb{R} \\ -\infty \end{cases}$$

Non prenderemo in considerazione i casi in cui la distanza è pari a $+\infty$ (vorrebbe dire che il vertice non è raggiungibile dalla sorgente, per cui varrebbe che $d[u] = \delta(s, u)$ subito dopo l'inizializzazione) o a $-\infty$ (per ipotesi, sappiamo che non ci sono cicli negativi raggiungibili dalla sorgente). Se $\delta(s, u) \in \mathbb{R}$, esiste un cammino tra s e u . Sia $p = < x_0 = s, x_1, \dots, x_q = u >$ un cammino minimo semplice tra s e u (è importante che il cammino sia semplice: potrebbero esistere cammini minimi non semplici, nel caso in cui il ciclo che lo contiene avesse costo totale pari a 0; notare che esiste sempre almeno un cammino minimo semplice, ottenibile rimuovendo tutti i cicli aventi somma 0). Tale cammino può avere fino a $n - 1$ archi in totale, nel caso in cui attraversi tutti i vertici del grafo. Ai fini della dimostrazione, supponiamo sia questo il caso. Poiché sottocammini di cammini minimi sono a loro volta minimi, possiamo applicare il teorema della convergenza tante volte quante sono le passate dell'algoritmo, e verificare che alla fine dell'algoritmo $d[u] = \delta(s, u)$.

$$d[x_0] = 0 = \delta(s, s) \quad d[x_2] = \delta(s, x_2) \quad d[x_q] = d[u] = \delta(s, u)$$



Dimostriamo che al momento della terminazione dell'algoritmo abbiamo $d[v] = \delta(s, v)$ per ogni nodo v , ragionando per induzione: verifichiamo che, all'inizio dell'iterazione i nel ciclo più esterno (righe 3-5), abbiamo che $d[v] = \delta(s, v)$ per ogni nodo v per il quale il cammino minimo da s a v è composto da al più i archi.

Questa proprietà è vera per $i = 0$, in quanto s è il solo nodo tale che il cammino minimo da s a s è composto da 0 archi, e in effetti $d[s] = 0 = \delta(s, s)$.

Per il passo induttivo, ipotizziamo ora che la proprietà sia vera all'inizio dell'iterazione i : se consideriamo un qualunque nodo v tale che il cammino minimo $< s, \dots, u, v >$ comprende $i + 1$ archi, ne consegue che per il nodo u il cammino minimo $< s, \dots, u >$ comprende i primi i archi del cammino precedente, e quindi abbiamo $d[u] = \delta(s, u)$. Ma allora, nel corso dell'iterazione i , abbiamo che $d[v]$ è aggiornato in modo che $d[v] = d[u] + w(u, v) = \delta(s, u) + w(u, v)$, che è uguale a $\delta(s, v)$ per l'ipotesi che il cammino $< s, \dots, u, v >$ sia minimo. Quindi la proprietà risulta soddisfatta quando inizia l'iterazione $i + 1$.

Se osserviamo che, dato che non esistono per ipotesi cicli di lunghezza negativa in G , un cammino minimo comprende al più $n - 1$ archi, possiamo concludere che quando $i = n$ tutti i cammini minimi sono stati individuati.

Dimostrazione c) Se alla fine l'algoritmo restituisce TRUE, allora dovrà valere che

$$\forall (u, v) \in E, \quad d[v] \leq d[u] + w(u, v)$$

La diseguaglianza triangolare afferma che

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Ma avendo dimostrato nel punto a) che $d[u] = \delta(s, u)$, possiamo sostituire le vere distanze con le stime delle distanze:

$$\delta(s, v) \leq \delta(s, u) + w(u, v) \Rightarrow d[v] \leq d[v] + w(u, v)$$

Che è quello che volevamo dimostrare.

Dimostrazione - Cicli negativi

Sappiamo, per ipotesi, che esiste un ciclo negativo raggiungibile da s , e vogliamo dimostrare che in questo caso l'algoritmo di Bellman-Ford restituisce FALSE. Supponiamo per assurdo che l'algoritmo restituisca TRUE. Allora,

$$\forall (u, v) \in E, \quad d[v] \leq d[u] + w(u, v)$$

Sia $c = < x_0, x_1, \dots, x_q >$, con $x_0 = x_q$, il ciclo negativo raggiungibile da s . Allora,

$$\sum_{i=1}^q w(x_{i-1}, x_i) < 0$$

Dal momento che l'algoritmo ha restituito **TRUE**, vale per tutti i nodi la proprietà di cui sopra, e quindi anche per i nodi del ciclo:

$$\forall i = 1, \dots, q, \quad d[x_i] \leq d[x_{i-1}] + w(x_{i-1}, x_i)$$

Disequazione che diventa, ragionando in termini di sommatorie:

$$\sum_{i=1}^q d[x_i] \leq \sum_{i=1}^q d[x_{i-1}] + \sum_{i=1}^q w(x_{i-1}, x_i)$$

Notiamo che, siccome $x_0 = x_q$, le due sommatorie sono equivalenti:

$$\begin{aligned} \sum_{i=1}^q d[x_i] &= d[x_1] + d[x_2] + d[x_3] + \dots + d[x_{q-1}] + d[x_q] \\ \sum_{i=1}^q d[x_{i-1}] &= d[x_0] + d[x_1] + d[x_2] + d[x_3] + \dots + d[x_{q-1}] \end{aligned}$$

Eliminando quindi le sommatorie dalla disequazione iniziale, otteniamo

$$0 \leq \sum_{i=1}^q w(x_{i-1}, x_i)$$

che è assurdo, in quanto per ipotesi il ciclo ha peso negativo.

20.3.17 Bellman-Ford - Esercizio

Dato un grafo G orientato con pesi positivi, $w(u, v) > 0 \forall (u, v) \in E$, esiste un ciclo $\langle x_0, x_1, \dots, x_q \rangle$ raggiungibile da $s \in V$ tale che valga la seguente diseguaglianza?

$$\prod_{i=1}^q w(x_{i-1}, x_i) < 1$$

Decidiamo di applicare la funzione logaritmo a entrambi i lati della disequazione:

$$\log \left(\prod_{i=1}^q w(x_{i-1}, x_i) \right) < \log 1 = 0$$

Disequazione che, per le proprietà dei logaritmi, può essere riscritta nel seguente modo:

$$\sum_{i=1}^q \log w(x_{i-1}, x_i) < 0$$

Giunti a questo punto, possiamo vedere l'esercizio nei termini del problema dei cammini minimi: creiamo un nuovo grafo G' con stessi vertici e stessi archi, ma con funzione peso $w' = \log w$ (cioè, il peso di ciascun arco (u, v) , invece di valere $w(u, v)$, varrà $\log w(u, v)$, che può assumere valori negativi se $w(u, v) \in [0, 1]$) e applichiamo su di esso l'algoritmo di Bellman-Ford: se, alla fine dell'esecuzione, questo avrà ritornato FALSE, allora vuol dire che esiste un ciclo che soddisfa questa condizione.

20.3.18 Dijkstra e BF per i cammini tra tutte le coppie

Prima di studiare il problema dei cammini minimi tra tutte le coppie di vertici di un grafo tramite algoritmi specifici, notiamo che questo problema può essere risolto chiamando iterativamente n volte gli algoritmi di Dijkstra e Bellman-Ford, in modo tale che in ciascuna iterazione il nodo corrente sia la sorgente.

Riportiamo di seguito la versione iterativa dell'algoritmo di Dijkstra che risolve questo problema, tuttavia la versione con Bellman-Ford è analoga: è sufficiente sostituire l'algoritmo da chiamare a riga 3.

```

1 Iterated_Dijkstra(G, w)
2   for each u ∈ V[G]
3     Dijkstra(G, w, u)
```

Dijkstra e Bellman-Ford a confronto

	Sparsa ($m \approx n$)	Densa ($m \approx n^2$)
It_Dijkstra (array)	n^3	n^3
It_Dijkstra (heap)	$n^2 \log n$	$n^3 \log n$
It_Bellman-Ford	n^3	n^4

20.3.19 Algoritmo di Floyd-Warshall

L'algoritmo di Floyd-Warshall risolve il problema dei cammini minimi tra tutte le coppie di vertici di un grafo $G = (V, E)$ orientato e pesato sugli archi da una funzione peso $w : E \rightarrow \mathbb{R}$. L'algoritmo supporta gli archi con pesi negativi, ma non supporta i cicli negativi, sebbene si possa adattare per notificare l'utente qualora si accorgesse della loro presenza, in modo analogo a quanto fa l'algoritmo di Bellman-Ford.

La grande novità introdotta da questo algoritmo rispetto ai precedenti è il fatto che si tratta di un algoritmo che opera su matrici: riceve in input un grafo $W = (w_{ij}) \in \mathbb{R}^{n \times n}$, il cui insieme V contiene ***n vertici etichettati da 1 a n*** ($V = \{1, 2, \dots, n\}$), rappresentato tramite una matrice, e costruisce una nuova matrice $D = (d_{ij}) \in \mathbb{R}^{n \times n}$, detta ***matrice delle distanze***, senza aver bisogno delle strutture ausiliarie δ e π utilizzate dagli algoritmi precedenti.

Notazioni usate Indicheremo con $w(i, j)$ il peso dell'arco (i, j) , e con w_{ij} l'elemento della matrice W presente nella riga i e nella colonna j ; analogamente, d_{ij} rappresenta l'elemento della matrice D presente nella riga i e nella colonna j .

L'elemento w_{ij} può assumere i seguenti valori:

$$w_{ij} = \begin{cases} 0 & \text{se } i = j \\ w(i, j) & \text{se } i \neq j \text{ e } (i, j) \in E \\ +\infty & \text{se } i \neq j \text{ e } (i, j) \notin E \end{cases}$$

Alla fine dell'algoritmo, vale che $d_{ij} = \delta(i, j)$.

L'algoritmo

```

1 Floyd-Warshall(W)
2   n ← rows(W)
3   D(0) ← W
4   for k = 1 to n
5     for i = 1 to n
6       for j = 1 to n
7         dij(k) = min {dik(k-1) + dkj(k-1), dij(k-1)}
8   return D(n)

```

Analisi della complessità L'algoritmo consiste in tre cicli **for** innestati che compiono ciascuno n iterazioni, per cui il tempo d'esecuzione è il seguente:

$$T(n) = \Theta(n^3)$$

20.3.20 Floyd-Warshall - Correttezza e spiegazione

Sia data la seguente definizione:

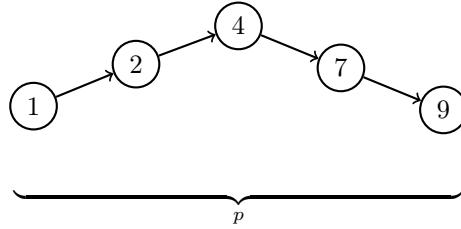
$$\mathcal{D}_{i,j} = \{p \mid p \text{ è un cammino semplice tra } i \text{ e } j\}$$

Quello che noi vogliamo calcolare, con l'algoritmo di Floyd-Warshall, è il peso del cammino minimo tra i e j , cioè

$$\delta(i, j) \stackrel{\text{def.}}{=} \min_{p \in \mathcal{D}_{i,j}} w(p)$$

Sia inoltre data la definizione di **vertice intermedio**: sia p un cammino semplice, $p = \langle x_0, x_1, \dots, x_q \rangle$; allora, un vertice intermedio in p è un qualsiasi vertice x di p tale che $x \neq x_0$ e $x \neq x_q$.

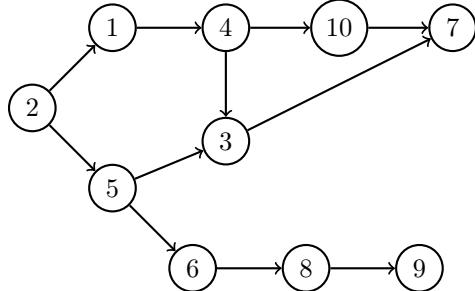
Nel seguente grafo, dato il cammino semplice $p \in \mathcal{D}_{1,9}$, sono vertici intermedi i vertici 2, 4 e 7.



Uniamo i due concetti per definire il seguente insieme:

$$\mathcal{D}_{i,j}^{(k)} = \{p \mid p \in \mathcal{D}_{i,j}, \text{ e i vertici intermedi hanno etichetta } \leq k\}$$

Vediamo un'esempio di come utilizzare tale espressione.



Nel grafo di cui sopra, vale che $\mathcal{D}_{2,7}^{(1)} = \mathcal{D}_{2,7}^{(2)} = \mathcal{D}_{2,7}^{(3)} = \emptyset$, in quanto non esistono cammini dal nodo 2 al nodo 7 i cui vertici intermedi siano minori o uguali a 1, 2 o 3.

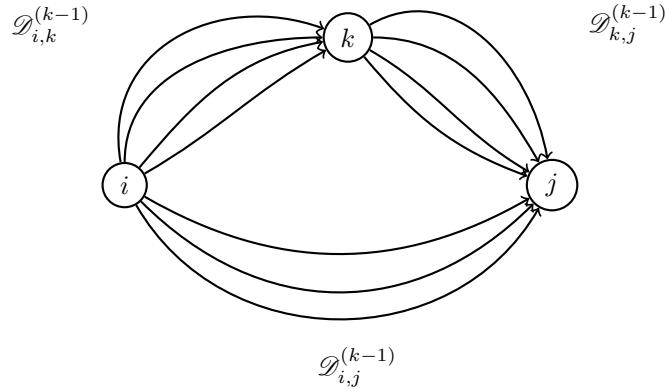
Tuttavia, per $k = 4$ avremo che $\mathcal{D}_{2,7}^{(4)} = \{<2, 1, 4, 3, 7>\}$ e per $k = 5$ avremo invece che $\mathcal{D}_{2,7}^{(5)} = \{<2, 1, 4, 3, 7>, <2, 5, 3, 7>\}$: possiamo notare che, per ogni k compreso tra 1 e $n - 1$, avviene che $\mathcal{D}_{i,j}^{(k)} \subseteq \mathcal{D}_{i,j}^{(k+1)}$, e in particolare, quando $k = n$, $\mathcal{D}_{i,j}^{(k)}$ e $\mathcal{D}_{i,j}$ coincidono.

Definiamo ora $d_{ij}^{(k)}$ come il peso del cammino minimo tra i e j dove i valori dei vertici intermedi hanno valore minore o uguale a k (in modo analogo a come abbiamo definito $\delta(i, j)$ nella pagina precedente), cioè

$$d_{ij}^{(k)} = \min_{p \in \mathcal{D}_{i,j}^{(k)}} w(p)$$

Dal momento che siamo riusciti a dimostrare che per $k = n$ vale che $\mathcal{D}_{i,j}^{(k)} = \mathcal{D}_{i,j}$, allora è altresì vero che $\delta(i, j) = d_{ij}^{(n)}$, che è l'elemento i, j -esimo della matrice risultato dell'algoritmo di Floyd-Warshall.

Generazione matrici successive Prima di capire come avviene la generazione delle matrici successive, spostiamo la nostra attenzione su un altro problema: sia X un insieme di numeri naturali, diviso in due partizioni Y e Z . Allora, $\min X = \min \{\min Y, \min Z\}$. Il fatto di poter ricondurre il problema della ricerca di un numero come minimo tra due minimi, precisamente come minimo di due sottoinsiemi, ci dà un importante spunto per applicare un algoritmo *divide et impera* o, in questo caso, di programmazione dinamica al problema dei cammini minimi. Vediamo in che modo.



Dividiamo l'insieme dei cammini da i a j in due sottocammini, quelli che passano per k e quelli che non passano per k , come nel grafo in figura (si supponga dunque che esistano altri vertici intermedi oltre a quelli mostrati nella figura). Poiché stiamo assumendo che **i cammini siano semplici**, non essendoci cicli, abbiamo la sicurezza che k non è compreso in nessuno dei due insiemi di sottocammini $\mathcal{D}_{i,k}^{(k-1)}$ e $\mathcal{D}_{k,j}^{(k-1)}$, e che tutti i vertici compresi in tali sottocammini abbiano valore minore o uguale a $k - 1$.

Definiamo il seguente insieme:

$$\hat{\mathcal{D}}_{i,j}^{(k)} = \left\{ p \mid p \in \mathcal{D}_{i,k}^{(k)} \text{ passanti per } k \right\}$$

Quindi,

$$\mathcal{D}_{i,j}^{(k)} = \hat{\mathcal{D}}_{i,j}^{(k)} \cup \mathcal{D}_{i,j}^{(k-1)}$$

E quindi, ricordando la definizione di $d_{ij}^{(k)}$,

$$\begin{aligned} d_{ij}^{(k)} &= \min_{p \in \mathcal{D}_{i,j}^{(k)}} w(p) \\ &= \min \left\{ \min_{p \in \hat{\mathcal{D}}_{i,j}^{(k)}} w(p), \min_{p \in \mathcal{D}_{i,j}^{(k-1)}} w(p) \right\} \\ &= \min \left\{ \min_{p \in \mathcal{D}_{i,k}^{(k-1)}} w(p) + \min_{p \in \mathcal{D}_{k,j}^{(k-1)}} w(p), d_{ij}^{(k-1)} \right\} \\ &= \min \left\{ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, d_{ij}^{(k-1)} \right\} \end{aligned}$$

$d_{ij}^{(k)}$ rappresenta dunque il peso del cammino minimo tra i e j i cui vertici intermedi sono di valore minore o uguale a k , e può assumere i seguenti valori:

$$d_{ik}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0 \\ \min \left\{ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, d_{ij}^{(k-1)} \right\} & \text{altrimenti} \end{cases}$$

Tale elemento si trova nella riga i e nella colonna j della matrice prodotta alla k -esima iterazione dell'algoritmo di Floyd-Warshall: l'algoritmo, infatti, produce n matrici aggiornando la riga e la colonna corrente secondo un approccio di programmazione dinamica, fino al punto in cui $k = n$ e la matrice finale, D , è quella che contiene le distanze reali tra tutte le coppie di vertici del grafo.

Ottimizzazioni Non è necessario produrre k matrici distinte, ne bastano due: quella che viene costruita nell'iterazione corrente e quella dell'iterazione precedente, da cui l'algoritmo prende gli elementi da confrontare, ma è possibile limitare l'uso spaziale ad una sola tabella, purché vengano soddisfatte le due seguenti condizioni:

1. Se non esistono cicli negativi, $\forall k = 1, \dots, n : d_{ii}^{(k)} = 0 \ \forall i = 1, \dots, n$.

Dimostrazione Avviene per induzione su k .

Caso base Per $k = 0$, la condizione è verificata per definizione di W .

Passo induttivo Assumiamo l'ipotesi induttiva che la proprietà valga fino a $k - 1$, e dimostriamolo per k .

$$d_{ii}^{(k)} = \min \left\{ \underbrace{d_{ik}^{(k-1)} + d_{ki}^{(k-1)}}_{\geq 0: \nexists \text{ cicli negativi}}, \underbrace{d_{ii}^{(k-1)}}_{=0 \text{ per ip. ind.}} \right\} = 0$$

Tale proprietà ci dice inoltre che se ci sono cicli negativi, i valori sulla diagonale possono assumere valori negativi; altrimenti, saranno sempre tutti zeri.

2. Presi due vertici $i, j \in V$ e un ulteriore vertice $k \in V$, vale che:

$$\begin{cases} d_{ik}^{(k)} = d_{ik}^{(k-1)} \\ d_{kj}^{(k)} = d_{kj}^{(k-1)} \end{cases}$$

cioè la k -esima riga e la k -esima colonna restano inalterate nella creazione della matrice k -esima a partire dalla matrice $(k - 1)$ -esima.

Dimostrazione Per definizione di $d_{ik}^{(k)}$, si ha che

$$d_{ik}^{(k)} = \min \{ d_{ik}^{(k-1)}, d_{ik}^{(k-1)} + d_{kk}^{(k-1)} \} = d_{ik}^{(k-1)}$$

dove $d_{kk}^{(k-1)} = 0$ perché si trova sulla diagonale principale.

Verificata la correttezza delle proprietà di cui sopra, possiamo riscrivere l'algoritmo di Floyd-Warshall ottimizzato in modo tale da utilizzare una sola matrice.

```

1  Floyd-Warshall(W)
2      n ← rows(W)
3      D ← W
4      for k = 1 to n
5          for i = 1 to n
6              for j = 1 to n
7                  dij = min{dik + dkj, dij}
8      return D

```

Il tempo di esecuzione dell'algoritmo rimane invariato, $T(n) = \Theta(n^3)$, tuttavia la complessità spaziale ora è quadratica: $S(n) = \Theta(n^2)$.

20.3.21 Floyd-Warshall - Esercizio

Sia data la matrice W che rappresenta il grafo G . Si calcolino le matrici create dall'algoritmo di Floyd-Warshall, iterazione per iterazione.

$$W = D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Durante lo svolgimento di questo esercizio, verranno forniti dei consigli utili su come calcolare a mano in maniera rapida le matrici risultanti dell'algoritmo.

In primo luogo, possiamo avvalerci delle due proprietà mostrate in precedenza:

1. Se non ci sono cicli negativi, i numeri della diagonale principale sono sempre pari a 0.
2. La k -esima riga e la k -esima colonna restano invariate nel passaggio dalla matrice $(k-1)$ -esima alla matrice k -esima.

Ragioniamo ora su come funziona Floyd-Warshall: alla luce delle proprietà di cui sopra, siamo certi che per calcolare la matrice di ordine $k=1$, la prima riga, la prima colonna e la diagonale principale rimarranno invariate nel passaggio da $D^{(0)}$ a $D^{(1)}$.

Per calcolare i numeri presenti nelle altre posizioni, cioè $d_{ij}^{(1)}$, dovremo valutare il minimo tra $d_{ik}^{(0)} + d_{kj}^{(0)}$ e $d_{ij}^{(0)}$. Prendiamo il caso di $d_{2,4}^{(1)}$: per calcolarlo, dobbiamo valutare il minimo tra $d_{2,1}^{(0)} + d_{1,4}^{(0)} = \infty + \infty = \infty$ e $d_{2,4}^{(0)} = 1$. Notiamo subito che $1 < \infty$, dunque $d_{2,4}^{(1)} = 1$.

Da questo esempio possiamo fare una considerazione molto vantaggiosa: se nella riga k -esima o nella colonna k -esima della matrice calcolata all'iterazione $k-1$ è presente ∞ , allora tale riga e/o tale colonna rimarrà invariata nella matrice calcolata nella k -esima iterazione. Il motivo, banale, è che ogni numero è sicuramente minore o uguale a ∞ , qualunque sia il suo valore.

3. Se in posizione (i, k) o (k, j) è presente ∞ per un qualche i indice di riga o j indice di colonna, allora la colonna j o la riga i rimarranno invariate nel passaggio dalla matrice di ordine $k-1$ alla matrice di ordine k .

Siamo ora pronti per calcolare $D^{(1)}$; per mostrare graficamente la valenza delle proprietà di cui sopra, sono evidenziati in verde gli elementi che restano **invariati per la prima proprietà**, in arancione gli elementi che restano **invariati per la seconda proprietà** e in azzurro gli elementi che restano **invariati per la terza proprietà**. Gli elementi per cui è necessario eseguire dei calcoli sono riportati in nero.

$$W = D^{(0)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty > 2 + 3 = \boxed{5} & 2 + 8 = 10 > \boxed{-5} & 0 & \infty > 2 - 4 = \boxed{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Le tre proprietà ci hanno consentito di calcolare la matrice $D^{(1)}$ effettuando solamente tre confronti, dato che il resto della matrice è rimasto invariato.
Di seguito le matrici $D^{(2)}$, $D^{(3)}$, $D^{(4)}$ e $D^{(5)}$.

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad D^{(3)} = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \quad D^{(5)} = \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

21 Algoritmi *Greedy*

Un algoritmo *greedy*, traducibile in italiano con i termini "goloso" o "miope", è un algoritmo iterativo, che ad ogni passo dell'iterazione compie una scelta, intraprende una strada che in quel momento sembra essere la più promettente per determinare il proseguimento dell'algoritmo.

Nello studio della teoria dei grafi abbiamo incontrato, senza saperlo, diversi algoritmi *greedy*:

- L'algoritmo di Kruskal, nel momento in cui deve scegliere un arco, prende sempre quello più leggero.
- L'algoritmo di Prim, quando estraе un vertice, sceglie sempre quello con campo **Key** minore.
- L'algoritmo di Dijkstra, quando estraе un vertice, sceglie sempre quello con campo **d** minore.

Non sono algoritmi *greedy* invece gli algoritmi di Bellman-Ford e di Floyd-Warshall, in quanto la loro struttura non prevede la presa di una decisione.

Analizziamo la struttura fondamentale degli algoritmi *greedy* attraverso lo studio di un problema.

21.1 Problema della selezione delle attività

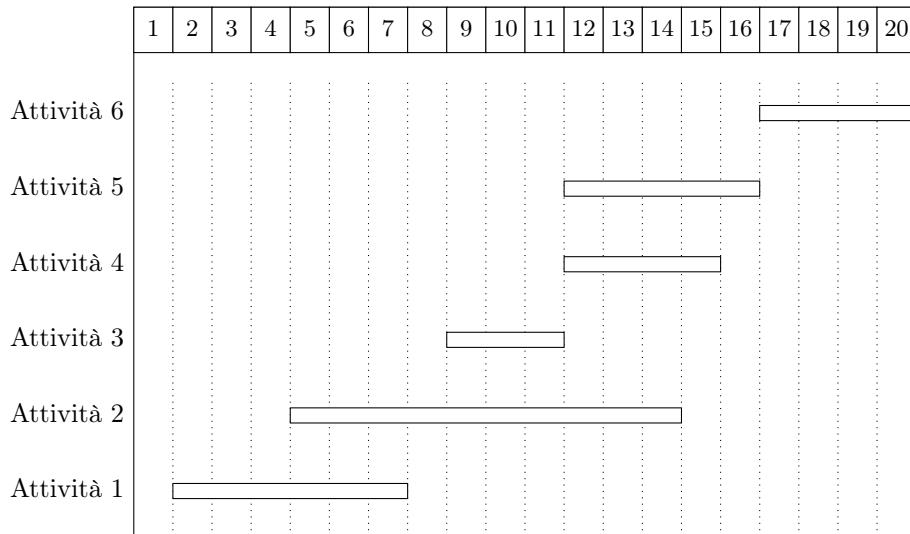
Supponiamo di avere n attività, indicizzate per numero da 1 a n , di cui sappiamo l'orario di inizio s_i e di fine f_i . Siano date due attività i e j ; tali attività si dicono compatibili se gli insiemi temporali che occupano sono disgiunti; formalmente, scriveremo:

$$\left. \begin{array}{l} i : [s_i, f_i[\\ j : [s_j, f_j[\end{array} \right\} [s_i, f_i[\cap [s_j, f_j[= \emptyset \Rightarrow i \text{ e } j \text{ sono compatibili}$$

Possiamo anche dire che i e j sono compatibili se $f_i \leq s_j \vee f_j \leq s_i$.
Si voglia determinare il massimo numero di attività che sono tra loro compatibili.

Un esempio reale di questo problema può essere l'associazione processi-risorse all'interno di un calcolatore o l'associazione lezioni-aule in un'università.

Visualizziamo il problema tramite un grafico di Gantt.



Come possiamo notare, attività come la 1, la 4 e la 6 sono tra loro compatibili, come lo sono anche la 3, la 4 e la 6 oppure la 1, la 5 e la 6: abbiamo diverse possibilità di scelta, ma il numero totale di attività tra loro compatibili non è mai superiore a 3 per qualsiasi combinazione di attività.

Cerchiamo di capire se è possibile scrivere un algoritmo *greedy* che risolva il problema. Innanzitutto, per trovare il numero massimo di attività compatibili dovremo, per forza di cose, sceglierne alcune e scartarne altre, e questa premessa ci suggerisce che è effettivamente possibile risolvere il problema tramite un algoritmo *greedy*.

Abbiamo bisogno di un criterio per confrontare le diverse attività: sceglieremo arbitrariamente il tempo di fine, e decidiamo di ordinare le diverse attività che abbiamo sulla base di tale criterio; alla fine dell'ordinamento, le attività saranno ordinate come segue:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_n$$

Per definizione di attività compatibili, non dovrà essere difficile distinguere le attività da tenere da quelle da scartare:

```

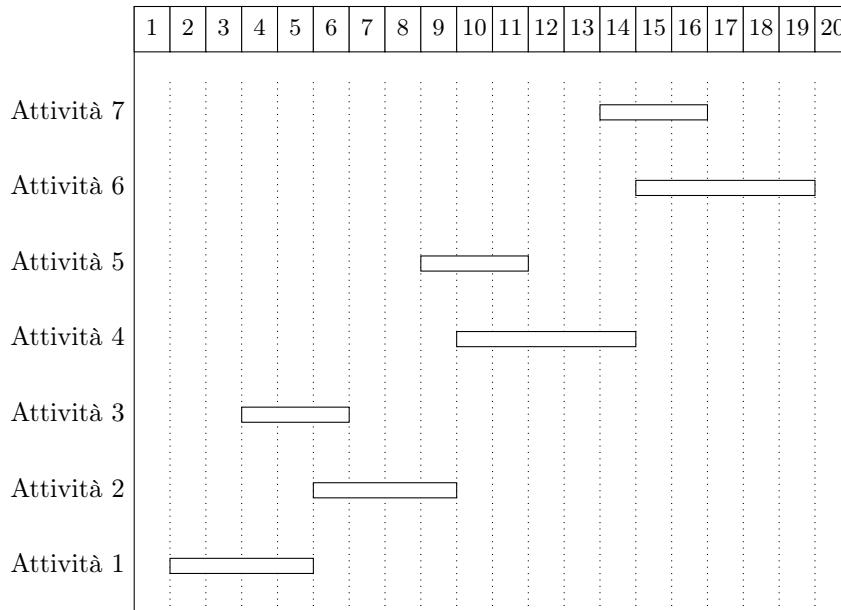
1 Greedy_Activity_Selector(s, f)
2   n ← length[s]
3   ordina le  $n$  attività per tempo di fine non decrescente
4   A ← 1
5   j ← 1
6   for i = 2 to n
7     if  $s_i \geq f_j$  then
8       A ← A ∪ {i}
9       j ← i
10  return A

```

Nell'algoritmo risolutivo, s e f sono rispettivamente i vettori di inizio e di fine attività, A è l'insieme delle attività massime e j è una variabile ausiliaria che ci ricorda qual è stata l'ultima attività inserita in A . Il controllo sulla compatibilità tra l'attività corrente e quella appena inserita viene fatto alla riga 7, ed è un controllo corretto in quanto riprende la definizione di attività compatibili. Il costo dell'algoritmo si deve all'ordinamento delle attività e al ciclo delle righe 6-9 che le itera: $T(n) = O(n \log n + n) = O(n \log n)$.

Non studieremo la correttezza dell'algoritmo, ma vogliamo soffermarci sul criterio col quale abbiamo ordinato le attività e di conseguenza risolto il problema: cosa succederebbe all'algoritmo se lo lasciassimo invariato, ma cambiassimo solamente il criterio di ordinamento?

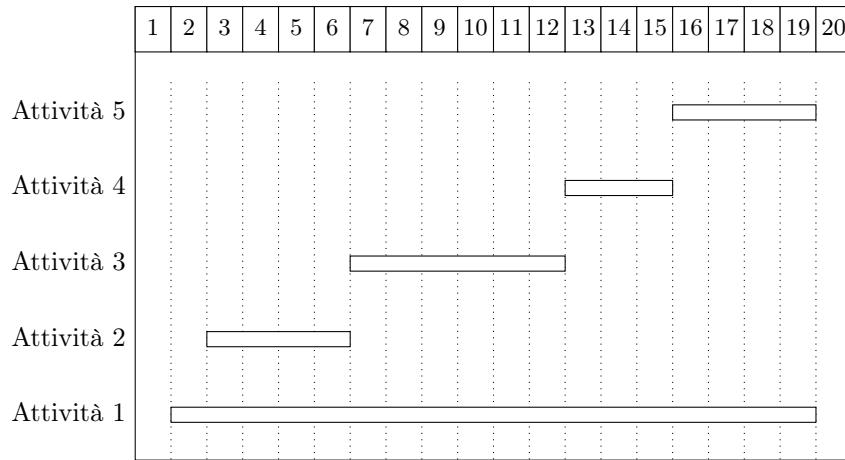
- **Ordinamento per durata**



Con l'ordinamento criterizzato secondo il tempo di fine, l'algoritmo ritornerebbe (correttamente) 4, in quanto l'insieme massimo è dato dalle attività 1, 2, 4 e 6, che hanno lunghezze 4 e 5.

Se tuttavia decidessimo di ordinare le attività secondo la loro durata, verrebbero considerate per prime le attività 3, 5 e 7, di durata 3, inferiore alle altre, ma essendo compatibili unicamente tra di loro e non con le altre, l'algoritmo ritornerebbe il risultato errato 3.

- **Ordinamento per tempo d'inizio**



Analogamente al caso precedente, l'algoritmo considererebbe per prima l'attività 1, e non essendo questa compatibile con nessun'altra attività ritornerebbe erroneamente 1, mentre possiamo vedere che in realtà il numero massimo di attività compatibili in questo caso è 4 e include le attività 2, 3, 4 e 5.

Questo ragionamento è servito a sottolineare l'importanza del criterio utilizzato nella creazione di algoritmi *greedy*.

Le osservazioni che possiamo compiere sull'algoritmo non sono tuttavia terminate: se mettessimo questo algoritmo e quello di Kruskal fianco a fianco, noteremmo non poche similitudini: in entrambi c'è un'operazione di ordinamento (degli archi e delle attività) e c'è un ciclo principale che itera ed estrae gli elementi ordinati; ciò che giustifica la somiglianza tra i due algoritmi è proprio il fatto che entrambi sono *greedy*, condividono dunque una medesima struttura di base. Ciò non significa che tutti gli algoritmi *greedy* siano uguali, in quanto, per esempio, Dijkstra e Prim si distinguono da Kruskal e dal corrente algoritmo dal momento che nei primi l'ordinamento è dinamico, avviene man mano che l'algoritmo prosegue la sua esecuzione, ma è comunque possibile trovare uno scheletro che accomuna gran parte degli algoritmi appartenenti a questa categoria.

21.2 Struttura comune degli algoritmi *greedy*

1. Ordinamento (secondo un qualche criterio a nostra discrezione)
2. $A \leftarrow \emptyset$ (inizializzazione delle strutture dati ausiliarie, solitamente poste uguali all'insieme vuoto o con elementi base)
3. **for each element** x preso secondo l'ordine stabilito nel punto 1.
4. **if** $A \cup \{x\}$ è ok **then** (test/controllo sulla condizione cercata)
5. $A \leftarrow A \cup \{x\}$ (aggiunta dell'elemento che soddisfa la condizione)
6. **return** A

Tale struttura prende il nome di **metodo** più che di algoritmo, perché indica una struttura basilare che poi, quando dovrà essere implementata, non potrà presentare condizioni generiche come l'"ordinamento secondo un qualche criterio" o il controllo se un elemento è "ok", ma è a cura di chi implementa il metodo chiarire quali sono i criteri e le condizioni generali che soddisfano il problema specifico.

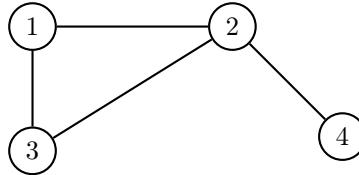
21.3 Problema della *clique* massima

Proseguiamo il nostro studio sugli algoritmi *greedy* analizzando un altro problema dalle proprietà teoriche interessanti e dalle applicazioni pratiche molto vaste: il problema della *clique* massima.

Nella teoria dei grafi, una *clique* (o, in italiano, "cricca") si può definire nel seguente modo: sia $G = (V, E)$ un grafo non orientato e non pesato. Prendiamo un sottinsieme di vertici $C \subseteq V$. Allora,

- C è una *clique* se $G[C]$ è completo.
- C è una *clique* massimale se non esiste una clique D tale che $C \not\subseteq D$ (cioè se non è sottinsieme di una *clique* più grande).
- C è una *clique* massima se la cardinalità di C , $|C|$, è massima tra tutte le possibili *clique* di G . Una *clique* massima è anche una *clique* massimale, ma non vale necessariamente il contrario.

Per ogni grafo G , identifichiamo con $\omega(G)$ il suo *clique number*, cioè la cardinalità della *clique* massima.



In questo esempio, i vertici 2 e 3 formano una *clique*, ma non la formano i vertici 1 e 4 oppure i vertici 1, 2 e 4. Inoltre, i vertici 1 e 2 non formano una *clique* massimale perché sono contenuti in una *clique* più grande, composta dai vertici 1, 2 e 3. Al contrario, i vertici 2 e 4 formano una *clique* massimale. Dal momento che i vertici 1, 2 e 3 non sono contenuti in nessun'altra *clique* e costituiscono la più grande del grafo, allora formano una *clique* massima, e vale che $\omega(G) = 3$.

21.3.1 Trovare la *clique* massima

Proviamo a implementare un algoritmo *greedy* che trovi la *clique* massima dato un grafo G in input.

```

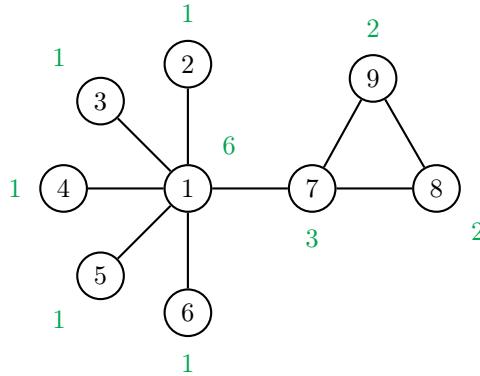
1 Greedy_Clique(G)
2   Ordina i vertici di  $G$  secondo il grado
3    $C \leftarrow \emptyset$ 
4   Per ogni vertice  $u$  di  $G$  estratto secondo il criterio di ordinamento
5     if Is_A_Clique( $C$ ,  $u$ ) then
6        $C \leftarrow C \cup \{u\}$ 
7   return  $C$ 
8
9 Is_A_Clique( $C$ ,  $u$ )
10  for each  $v \in C$ 
11    if  $(u, v) \notin E$  then
12      return FALSE
13  return TRUE

```

Il tempo d'esecuzione dell'algoritmo è dato dall'ordinamento e dallo scorriamento dei vertici, scorimenti in cui, ad ogni iterazione, la chiamata ad `Is_A_Clique` compie al più n iterazioni:

$$T(n) = O(n \log n + n^2) = O(n^2)$$

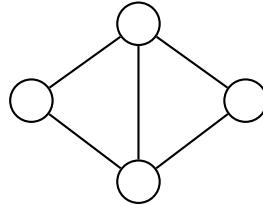
Sappiamo che, per come è costruito l'insieme C , il risultato è sicuramente una *clique*, ma è massima? La risposta è no, e per dimostrarlo basta trovare un controsenso.



L'algoritmo estrarrebbe il vertice 1 e lo aggiungerebbe in C , e successivamente estrarrebbe il vertice 7 e lo aggiungerebbe in C . A questo punto l'algoritmo terminerebbe, perché i due vertici aggiunti formano una *clique*, ma provando a estrarre i vertici 8 e 9 (aventi grado 2), questi non la formerebbero con i vertici già estratti, e di conseguenza verrebbero scartati; lo stesso discorso si può fare con i vertici di grado 1. Tale algoritmo dunque restituirebbe la *clique* $\{1, 7\}$, che è massimale, ma non è massima, in quanto quella massima è costituita dai vertici $\{7, 8, 9\}$. Possiamo supporre la seguente analogia: una *clique* massimale sta ad una *clique* massima come un minimo locale sta ad un minimo globale nello studio di funzione nell'analisi matematica.

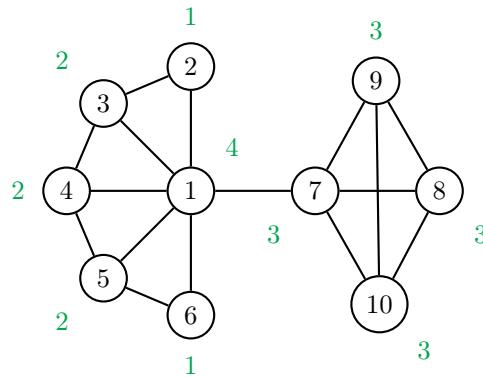
E se l'algoritmo fosse scorretto perché il grado dei vertici è un criterio di scelta errato? Può esistere un altro criterio tale per cui si può mantenere la stessa struttura e ritornare la *clique* massima?

Introduciamo il concetto di triangolo: un triangolo è una *clique* (o un ciclo) costituita da tre vertici; il grafo in figura è costituito da due triangoli incidenti.



Provando ad estrarre i vertici del grafo in ordine decrescente di numero di triangoli incidenti, sul grafo precedente l'algoritmo funzionerebbe, in quanto gli unici vertici incidenti ad un triangolo (7, 8 e 9) verrebbero estratti per primi ed inseriti in C in quanto formano una *clique*.

Purtroppo, è possibile trovare un altro controesempio che dimostra la scorrettezza di questa idea:



In tal caso, l'algoritmo estrarrebbe e terrebbe solamente i vertici 1 e 7, mentre la *clique* massima è formata dai vertici 7, 8, 9 e 10.

La dura e triste verità è che finché il criterio con cui ordiniamo i vertici è polinomiale, un algoritmo che trovi la *clique* massima non può sicuramente funzionare: possiamo contare i gradi, il numero di triangoli, o qualsiasi altro aspetto che desideriamo sui vertici, però è molto inverosimile, quasi impossibile, che un algoritmo *greedy* possa risolvere il problema della *clique* massima, e nel capitolo successivo ne studieremo il motivo.

22 Teoria della NP completezza

Quasi tutti i problemi che abbiamo trovato fino a questo momento hanno in comune un'importante caratteristica: è stato possibile risolverli tutti attraverso un algoritmo che impiega un tempo, la maggior parte delle volte, polinomiale rispetto alla dimensione dell'input.

Nel capitolo precedente, tuttavia, ci siamo imbattuti in un problema molto particolare, quello di trovare in un grafo la *clique* massima: questa questione, insieme ad un'altra migliaia di problemi tutt'ora aperti, appartiene ad una categoria speciale di problemi per cui non conosciamo una procedura risolutiva che impiega tempo polinomiale. Nel mondo reale, infatti, esistono problemi detti **intrattabili**, per cui sarebbe molto inverosimile, quasi impossibile, credere che si possa trovare una soluzione tramite algoritmi polinomiali.

22.1 Problemi

Parliamo in modo generico di problemi. Un problema può essere rappresentato in maniera astratta come una relazione binaria tra le sue possibili istanze e le sue possibili soluzioni:

$$\mathcal{P} \subseteq \underbrace{\mathcal{I}}_{\text{Istanze}} \times \underbrace{\mathcal{S}}_{\text{Soluzioni}}$$

Per esempio, nel problema della *clique* massima, l'insieme delle istanze è costituito da tutti i possibili grafi non orientati e non pesati sugli archi, mentre l'insieme delle soluzioni è dato da tutte le possibili combinazioni di vertici che costituiscono una *clique* massima all'interno del grafo dato in input, oppure, analogamente, nel problema della ricerca dei cammini minimi l'insieme delle istanze è dato dall'insieme di tutti i grafi orientati e pesati sugli archi, e l'insieme delle soluzioni è dato da tutti i cammini minimi possibili.

Esistono due principali partizioni per i problemi:

- **Problemi decidibili**

Sono problemi per cui esiste un algoritmo che dia una soluzione in tempo finito. Tutti i problemi che abbiamo incontrato finora appartengono a tale categoria.

- **Problemi indecidibili**

Sono problemi per cui non esiste un algoritmo produca una soluzione in tempo finito. Un esempio di problema indecidibile è il **problema della fermata**: supponiamo di voler creare un algoritmo A_1 che prenda in input un altro algoritmo A_2 e dei dati, e restituisca 1 se l'algoritmo A_2 termina con i dati in ingresso o 0 altrimenti. Non è possibile implementare un algoritmo del genere, perché se A_2 non dovesse mai terminare, allora neanche A_1 terminerà mai.

Per quanto riguarda i problemi decidibili, possiamo compiere un'ulteriore distinzione:

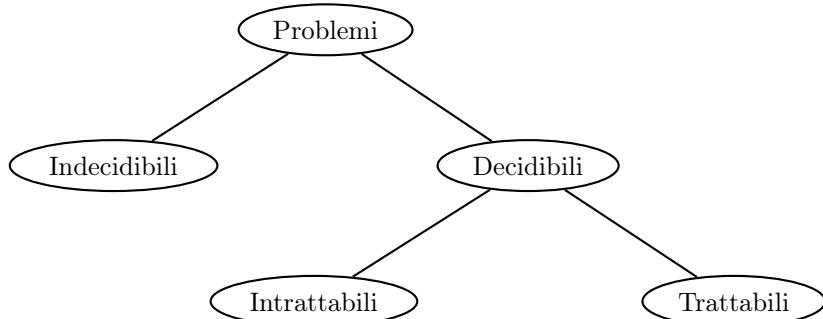
- **Problemi trattabili**

Sono problemi per cui esistono algoritmi risolutivi al più polinomiali, cioè risolvibili in un tempo $O(n^k)$, dove n è la dimensione dell'input e k è una qualche costante. Nella letteratura, la maggior parte dei problemi trattabili viene risolto entro un tempo polinomiale con $k \leq 4$.

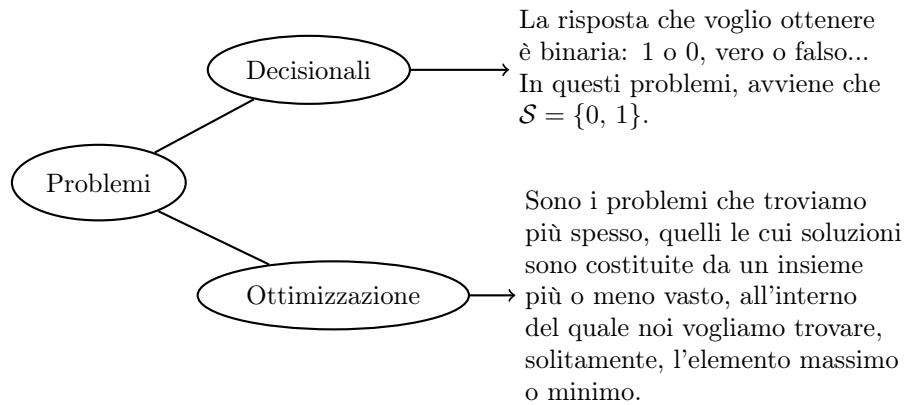
- **Problemi intrattabili**

Sono problemi i cui algoritmi risolutivi hanno tempi d'esecuzione almeno esponenziali, cioè risolvibili in un tempo $O(k^n)$, dove n è la dimensione dell'input e k è una qualche costante.

Questa distinzione tuttavia è, se vogliamo, un po' volgare: un giorno potremmo scoprire un algoritmo di tempo polinomiale che risolve un problema che ora sappiamo risolvere solamente in tempo esponenziale; per effettuare una distinzione più formale, è necessario capire il limite oltre il quale l'intrattabilità del problema è dovuta alla nostra (in)capacità di risolverlo o alla natura stessa del problema.



La terminazione e il tempo d'esecuzione dell'algoritmo risolutivo di un problema non sono gli unici criteri che ci permettono di categorizzare i problemi, ma possiamo valutarli anche in base al valore del loro eventuale risultato.



Un esempio di problema decisionale può essere, appunto, il problema della fermata visto poco fa, ma ce ne possono essere moltissimi altri: dato un grafo, dire se esiste una *clique* di 5 vertici, oppure dato un sistema lineare $Ax = b$, dire se esiste una soluzione $x \in \mathbb{R}^n$ tale che $Ax = b$ per un dato A e un dato b (non ci interessa sapere la soluzione, bensì se esiste oppure no).

La maggior parte degli algoritmi che abbiamo visto in questo corso risolvono problemi di ottimizzazione: gli algoritmi di Kruskal e Prim, per dirne alcuni, ne sono un esempio.

I problemi si possono presentare in due varianti, una decisionale e una di ottimizzazione.

Un esempio è il **problema del ciclo Hamiltoniano**: dato un grafo G non orientato e non pesato, si voglia determinare se esiste un ciclo che attraversa tutti i nodi di G una ed una sola volta. Una specializzazione di questo problema è il **problema del commesso viaggiatore**, in inglese *travelling salesman problem* o *TSP*, che date una lista di città e le distanza tra ogni coppia di città, si chiede quale sia la lunghezza della strada più corta che attraversa ogni città una ed una sola volta e ritorna alla città di partenza. Dal momento che quest'ultimo problema richiede, tra tutti i possibili cammini circolari, quello più breve, il *TSP* è un problema di ottimizzazione.

Anche il problema della *clique* massima è un problema di ottimizzazione, poiché richiede la cardinalità della *clique* massima. Ne esiste anche una variante decisionale, che si può esprimere in questo modo: dato un grafo non orientato e un intero k , esiste una *clique* con almeno k vertici?

Le complessità dei problemi della *clique* sono identiche: se qualcuno dovesse trovare un algoritmo polinomiale per risolvere il problema della *clique* massima, risolverebbe anche la variante decisionale confrontando la cardinalità della *clique* trovata col parametro k della versione decisionale. In altre parole, se il problema è trattabile, la sua versione decisionale è altresì trattabile.

22.2 Classi di problemi decisionali

22.2.1 Classe P

La **classe P**, dove P sta per "polinomiale", è definita nel seguente modo:

$$P = \{\mathcal{P} \mid \mathcal{P} \text{ è un problema decisionale } \mathbf{risolvibile} \text{ polinomialmente}\}$$

Intuitivamente, tutti i problemi che siamo riusciti a risolvere fino a questo momento appartengono a questa classe.

22.2.2 Classe NP

La **classe NP** è definita nel seguente modo:

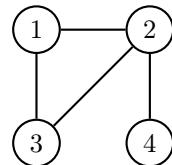
$$NP = \{\mathcal{P} \mid \mathcal{P} \text{ è un problema decisionale } \mathbf{verificabile} \text{ polinomialmente}\}$$

Spieghiamo il significato del termine "verificabile" relativamente all'ambito della teoria computazionale.

Innanzitutto, occorre fare una distinzione tra le istanze di un problema decisionale. Per definizione, ogni problema decisionale può essere risolto ottenendo come risultato un 1 o uno 0, un "Si" o un "No", senza ammettere valori intermedi: possiamo quindi partizionare l'insieme delle istanze \mathcal{I} in due sottinsiemi, \mathcal{I}_+ , che include le istanze positive (che ritornano 1, "Si"...), e \mathcal{I}_- , che include le istanze negative (che ritornano 0, "No"...).

Immaginiamo ora di avere un algoritmo A_{VER} , detto **algoritmo di verifica**, che prende in input due elementi: un'**istanza positiva** $i \in \mathcal{I}_+$ di un problema e un **certificato**, cioè una presunta soluzione del problema. Compito dell'algoritmo di verifica è, appunto, verificare che l'istanza passata in input sia effettivamente un'istanza positiva per il problema attraverso il certificato, e ritornare "Si" in tal caso, "No" altrimenti.

Esemplifichiamo la questione, dimostrando che il problema del ciclo Hamiltoniano appartiene alla classe NP.
Si supponga di avere il grafo G in figura, e di utilizzare come certificato la seguente permutazione dei vertici che dovrebbe rappresentare il ciclo Hamiltoniano: $(2, 4, 3, 1)$.



Scopo dell'algoritmo di verifica è provare che questa permutazione di vertici consista in un ciclo Hamiltoniano per G oppure no. Se lo dovesse essere, allora proverebbe che G sia un grafo Hamiltoniano, ma se per caso dovesse mancare un arco (come in questo caso, in cui non esiste l'arco tra i nodi 4 e 3), ciò non proverebbe che il grafo non sia Hamiltoniano, bensì che questo certificato non permette di dimostrare che G sia un'istanza valida per questo problema.

Immaginiamo un altro esempio: se in esame ci venisse data una proprietà, e ci venisse chiesto di dimostrarla, staremmo affrontando un problema di risoluzione; se invece ci venissero date la proprietà ed una presunta dimostrazione, e ci

venisse chiesto di verificare che la proprietà sia corretta, si tratterebbe di un problema di verifica. Se la dimostrazione è falsa, non è detto che la proprietà sia falsa, ma sappiamo che la dimostrazione data (il certificato) non ci permette di provarla.

Per determinare se il problema del ciclo Hamiltoniano appartiene a NP, bisogna analizzare il funzionamento dell'algoritmo di verifica: sia $p = \langle x_1, x_2, \dots, x_n \rangle$ la permutazione che costituisce il certificato; allora, per controllare la correttezza del certificato basta verificare che $\forall i = 1, \dots, n-1, (x_i, x_{i+1}) \in E \wedge (x_n, x_1) \in E$. Dal momento che questa operazione può essere eseguita in tempo polinomiale, addirittura lineare, il problema appartiene a NP.

Possiamo dimostrare che anche il problema della *clique* appartiene a NP. Innanzitutto, sappiamo che le istanze del problema sono costituite da tutti i possibili grafi non orientati e non pesati $G = (V, E)$ e da tutti gli interi k , e le soluzioni consistono in tutti i possibili sottinsiemi di vertici $C \subseteq V$. Il nostro algoritmo di verifica prenderà quindi in input un grafo, un intero e un sottinsieme di vertici, e dovrà verificare se questi costituiscono una *clique* per il grafo in input.

Dovrebbe in primo luogo verificare che la cardinalità di C sia uguale a k , e successivamente iterare attraverso due cicli innestati i vertici di C per verificare che siano tutti adiacenti tra di loro (cioè che $\forall i, j = 1, \dots, n : i \neq j, (i, j) \in E$). Questo controllo può essere eseguito in un tempo $O(n^2)$, dunque il problema appartiene a NP.

22.2.3 Classe Co-NP

La classe **Co-NP** è definita nel seguente modo:

$$\text{Co-NP} = \{ \mathcal{P} \mid \mathcal{P} \text{ è un problema decisionale tale che } \bar{\mathcal{P}} \text{ è verificabile polinomialmente} \}$$

dove $\bar{\mathcal{P}}$ è il **complemento** di \mathcal{P} . Vediamo di cosa si tratta.

Complemento di un problema decisionale Dato un problema con istanze positive e istanze negative, il complemento di un problema è un nuovo problema che ha come istanze positive le istanze negative del problema originale e viceversa.

Che rapporto esiste tra i problemi della classe NP e quelli della classe Co-NP? Prendiamo l'esempio del ciclo Hamiltoniano: il complemento del problema del ciclo Hamiltoniano è un problema che risponde "Sì" quando il grafo passatogli non è Hamiltoniano, "No" altrimenti. Ricordiamo che il problema del ciclo Hamiltoniano aveva, come certificato, una permutazione dei vertici: tale tipo di risultato è sufficiente per dimostrare che un grafo sia Hamiltoniano.

Tuttavia, il complemento del ciclo Hamiltoniano non può avere come certificato una permutazione dei vertici, dal momento che una singola permutazione non ci permette di stabilire se un grafo sia non Hamiltoniano, ma al massimo, se

non dovesse costituire un ciclo, ci può suggerire che quella permutazione non è sufficiente per provare che non lo sia: il certificato di questo problema dovrà essere una lista di tutte le possibili permutazioni dei vertici, esistenti nel numero di $n!$.

L'algoritmo di verifica si potrebbe anche implementare, ma essendo la funzione fattoriale di ordine super-esponenziale, il problema non è verificabile polinomialmente: il complemento del problema del grafo Hamiltoniano non appartiene a NP.

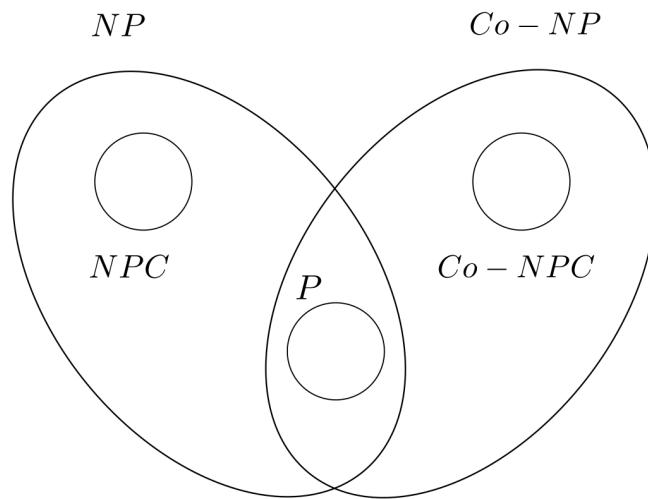
L'appartenenza alla classe Co-NP tuttavia non esclude l'appartenenza alla classe NP: un esempio è il problema del ***matching*** (o **accoppiamento**) **perfetto**: il problema richiede, dato un grafo, solitamente non orientato e non pesato, di trovare un sottinsieme di archi tali che tutti i vertici del grafo sono incidenti ad uno ed un solo arco del sottinsieme.

Possiamo ragionare ulteriormente sulle relazioni tra le diverse classi.

Innanzitutto, sicuramente $P \subseteq NP$: se un problema è risolvibile in tempo polinomiale, allora è sicuramente anche verificabile in tempo polinomiale.

Inoltre, si può dimostrare che $P \subseteq Co-NP$, tuttavia la dimostrazione di tale affermazione non è argomento del nostro studio.

Prima di introdurre la successiva classe di problemi decisionali, per la quale servirà una discussione più approfondita, ci basti sapere che, per quanto riguarda gli studi attuali (ed è importante sottolinearlo), i rapporti tra le diverse classi si possono rappresentare graficamente tramite il seguente diagramma di Venn.



22.2.4 Classe NPC

Riducibilità polinomiale Per descrivere il contenuto della classe NPC, occorre introdurre il concetto di **riducibilità polinomiale** tra problemi.

Questa non è una nozione a noi sconosciuta: quando incontriamo un nuovo problema e invece di creare un algoritmo partendo da zero cerchiamo di ricondurre il problema ad un altro che abbiamo già affrontato, stiamo inconsciamente applicando il concetto di riducibilità tra problemi; l'abbiamo visto, per esempio, tra gli algoritmi *greedy*.

Affinché si possa parlare di riducibilità polinomiale, è necessario che anche l'operazione di riduzione sia efficiente: diciamo che un problema \mathcal{P}_1 è riducibile polinomialmente al problema \mathcal{P}_2 se esiste un algoritmo polinomiale che "mappi" le istanze del problema \mathcal{P}_1 in istanze equivalenti del problema \mathcal{P}_2 . Questa condizione si riassume formalmente nel seguente modo:

$$\mathcal{P}_1 \leq_P \mathcal{P}_2$$

Chiaramente, è necessario preservare la positività e la negatività delle istanze: istanze positive di \mathcal{P}_1 possono essere mappate solamente a istanze positive di \mathcal{P}_2 per parlare di riducibilità, e lo stesso avviene per le istanze negative dei due problemi.

Proprietà della riducibilità

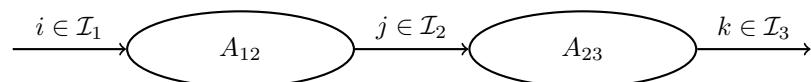
- \leq_P è **riflessiva**: dato un problema decisionale qualsiasi, possiamo dire che è riducibile a sé stesso in quanto l'algoritmo di riducibilità applica il mapping identità: ogni istanza del problema viene mappata a sé stessa.
- \leq_P è **transitiva**, ed è di seguito riportata la dimostrazione.

$$\underbrace{\begin{array}{c} \mathcal{P}_1 \leq_P \mathcal{P}_2 \\ \mathcal{P}_2 \leq_P \mathcal{P}_3 \end{array}}_{\text{Ipotesi}} \quad \underbrace{\mathcal{P}_1 \leq_P \mathcal{P}_3}_{\text{Tesi}} ?$$

Per definizione di riducibilità, $\mathcal{P}_1 \leq_P \mathcal{P}_2$ implica che esiste un algoritmo che prende le istanze di \mathcal{P}_1 e le mappa in tempo polinomiale in istanze di \mathcal{P}_2 : $A_{12} : \mathcal{I}_1 \rightarrow \mathcal{I}_2$.

Analogamente, $\mathcal{P}_2 \leq_P \mathcal{P}_3$ implica che esiste un algoritmo che prende le istanze di \mathcal{P}_2 e le mappa in tempo polinomiale in istanze di \mathcal{P}_3 : $A_{23} : \mathcal{I}_2 \rightarrow \mathcal{I}_3$.

Per cui, è sufficiente creare un algoritmo A_{13} che costituisca la concatenazione di A_{12} e A_{23} in modo tale che prenda come input una generica istanza $i \in \mathcal{I}_1$ e la trasformi prima in un'istanza $j \in \mathcal{I}_2$ e finalmente in un'istanza $k \in \mathcal{I}_3$.



Tale algoritmo avrà sicuramente complessità polinomiale, dato che la sua complessità è costituita dalla somma algebrica delle complessità di A_{12} e A_{23} , che sono polinomiali. Quindi la riducibilità è transitiva.

- \leq_P non è necessariamente simmetrica: non è detto che se un problema \mathcal{P}_1 è riducibile a \mathcal{P}_2 allora vale anche il contrario.

Siamo ora pronti per dare una definizione formale della classe NPC.

$$\text{NPC} = \left\{ \begin{array}{l} \mathcal{P} \mid \mathcal{P} \text{ è un problema decisionale tale che} \\ \quad \textbf{1)} \mathcal{P} \in \text{NP} \\ \quad \textbf{2)} \forall \mathcal{P}' \in \text{NP} : \mathcal{P}' \leq_P \mathcal{P} \end{array} \right\}$$

Un problema che fa parte della classe NPC si dice **NP completo**.

La seconda condizione di appartenenza alla classe NPC è un'assunzione molto forte: implica che tutti i problemi in NP possono essere mappati in qualsiasi problema NP completo, pur essendo completamente diverso, oltre al fatto che tutti i problemi in P sono riducibili a problemi in NPC, in quanto $P \subseteq \text{NP}$. Inoltre, essendo $\text{NPC} \subseteq \text{NP}$, vuol dire che tutti i problemi NP completi possono essere ridotti tra di loro: questo implica che in NPC la relazione di riducibilità è simmetrica.

Teorema fondamentale della NP completezza

$$P \cap \text{NPC} \neq \emptyset \Rightarrow P = \text{NP}$$

Significa che se esiste un problema che è risolvibile in tempo polinomiale, ma che è anche NP completo, allora P e NP sono lo stesso insieme. Ciò implicherebbe che tutte le migliaia di problemi NP completi che attualmente conosciamo ma non siamo in grado di risolvere in quanto intrattabili potrebbero essere risolti in tempo polinomiale.

Nessuno è mai riuscito a trovare un problema tale per cui la congettura sia vera: molto probabilmente, infatti, non lo è, e a suggerirlo ci sono diverse centinaia di indizi trovati nella letteratura, eppure nessuno è mai riuscito ad elaborare una dimostrazione valida che confutasse o verificasse la tesi del teorema.

Dimostrazione del teorema Per ipotesi, $\exists \mathcal{P} \in \text{NPC} \cap P$, cioè $\mathcal{P} \in \text{NPC} \wedge \mathcal{P} \in P$. Bisogna dimostrare che:

1. $P \subseteq \text{NP}$, ma questo è banalmente vero: un problema risolvibile polynomialmente è anche verificabile polynomialmente.

2. $\text{NP} \subseteq P$.

Sia Q un problema in NP. Allora, dovrà valere che $\forall Q \in \text{NP} : Q \in P$.

Essendo $Q \in \text{NP}$ e $\mathcal{P} \in \text{NPC}$ per ipotesi, allora:

$$\underbrace{Q \leq_P \mathcal{P}}_{\text{Def. NPC}} \in \overbrace{\text{P}}^{\text{Per ipotesi}}$$

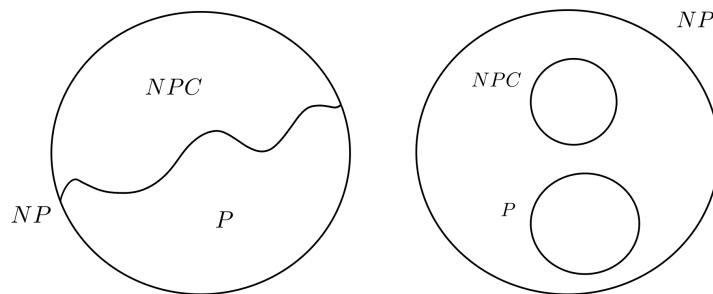
Il fatto che, per definizione di NPC, Q è riducibile a \mathcal{P} , che per ipotesi appartiene a P, vuol dire che esiste un algoritmo polinomiale che traduce le istanze di Q in istanze di \mathcal{P} , che può essere risolto polinomialmente. Significherebbe quindi che Q si può risolvere in tempo polinomiale, cioè $Q \in \text{P}$, come volevasi dimostrare.

Il teorema appena dimostrato ci suggerisce che la relazione tra le classi P, NP e NPC potrebbe essere la seguente: $\text{P} \cup \text{NPC} = \text{NP}$, $\text{P} \cap \text{NPC} = \emptyset$, cioè P e NPC sono due partizioni di NP.

Ciò sarebbe plausibile, se non fosse per i cosiddetti **problemI NPI o NP-intermedi**, cioè quei problemi che sappiamo verificare polinomialmente ma che non appartengono né alla classe P né alla classe NPC. Uno dei problemI NPI più famosi è sicuramente il problema dell'isomorfismo tra grafi, che abbiamo già incontrato nello studio della teoria dei grafi: dati due grafi e una presunta funzione di isomorfismo ϕ , è possibile dimostrare in tempo polinomiale se ϕ è un isomorfismo tra i due grafi (quindi il problema appartiene a NP), ma è impossibile determinare l'isomorfismo ϕ a partire dai due grafi (e quindi risolvere il problema in tempo polinomiale), e fino ad oggi non si è ancora riuscita a dimostrare l'appartenenza di questo problema alla classe NPC.

È più probabile, quindi, che P e NPC siano due sottinsiemi stretti di NP.

Nelle immagini che seguono, a sinistra è rappresentato NP secondo l'ipotesi suggerita dal teorema appena dimostrato, mentre a destra è rappresentato NP tenendo conto dei problemI NPI.



Riassumendo, abbiamo quindi capito che:

- $\text{NPC} \subseteq \text{NP} \Rightarrow$ Tutti i problemi NP completi sono verificabili polinomialmente.
- $\mathcal{P} \in \text{NPC}, \forall \mathcal{P}' \in P(\subseteq \text{NP}) : \mathcal{P}' \leq_P \mathcal{P} \Rightarrow$ Tutti i problemi in P (che sono quindi anche NP) sono riducibili a problemi NPC.
- $\forall \mathcal{P}', \mathcal{P}'' \in \text{NPC} : \mathcal{P}' \leq_P \mathcal{P}'' \Rightarrow$ Tutti i problemi in NPC sono riducibili ad altri problemi in NPC.

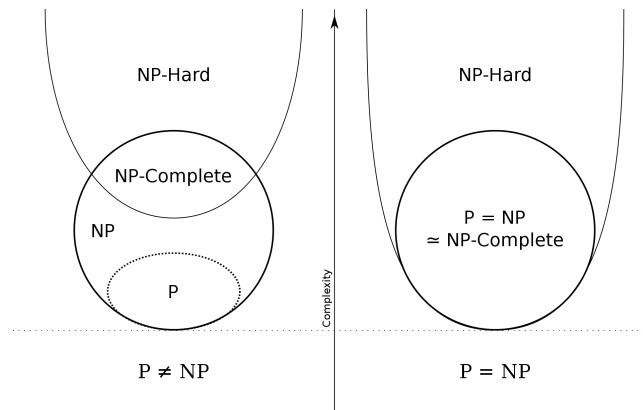
22.2.5 Classe NP-hard

La classe **NP-hard** è definita nel seguente modo:

$$\text{NP-hard} = \{\mathcal{P} \mid \mathcal{P} \text{ è un problema decisionale tale che } \forall \mathcal{P}' \in \text{NP} : \mathcal{P}' \leq_P \mathcal{P}\}$$

La proprietà determinante di appartenenza alla classe NP-hard è in realtà la seconda caratteristica dei problemi della classe NPC: un problema NP-hard è definito tale se tutti i problemi della classe NP sono riducibili ad esso, a prescindere dal fatto che questo appartenga alla classe NP o meno.

Conseguenza di ciò è che tutti i problemi NP completi sono anche NP-hard, e possiamo quindi dire molto informalmente che i problemi NP-hard sono tutti quei problemi che sono difficili da risolvere almeno quanto i problemi più difficili della classe NP.



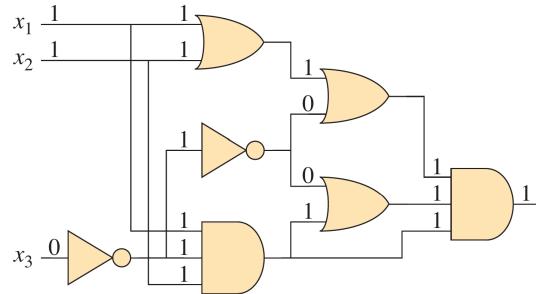
22.3 Alcuni problemi NP completi

22.3.1 Il problema CIRCUIT-SAT

Il primo problema che è stato dimostrato appartenere alla classe NPC risale agli anni '70, e attraverso questo negli anni siamo riusciti a dimostrare la NP completezza di tutti i problemi in NPC che conosciamo.

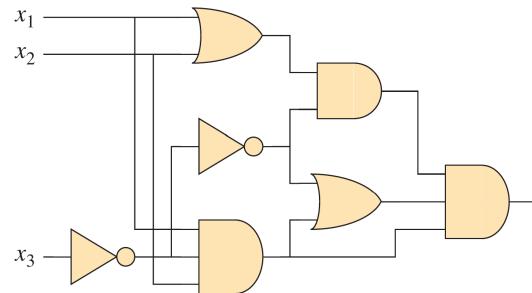
Stiamo parlando del problema del **CIRCUIT-SAT**, la cui NP completezza è garantita dal **teorema di Cook**. Il CIRCUIT-SAT è un problema che riguarda circuiti logici costruiti con le tre porte logiche di base, l'AND, l'OR e il NOT. Il problema si chiede se, dato un circuito con n linee in input, esiste una configurazione delle linee tale che l'output del circuito sia 1. Il problema in sé è di natura teorica, ma ha risvolti pratici nel momento in cui si possono individuare sottocircuiti che rispondono sempre 0, che nel mondo reale sarebbero inutili.

Vediamo un esempio.



Il circuito in questione è **soddisfacibile**, cioè è un'istanza positiva del problema del CIRCUIT-SAT (da cui, appunto, *satisfiability*) dal momento che la configurazione degli input presentata fa sì che l'output del circuito sia 1.

Proviamo a cambiare il circuito, modificando la porta OR in alto a destra in una porta AND, e vediamo se il circuito resta soddisfacibile.



Dal momento che nessuna delle $2^n = 2^3 = 8$ possibili configurazioni degli input potrà fare in modo che l'output del circuito sia 1, questo non è un circuito soddisfacibile.

Si può dimostrare che CIRCUIT-SAT è NP: l'algoritmo di verifica riceverà un circuito e come certificato una presunta configurazione di input che dovrebbe ritornare 1 dal circuito in input. Dal momento che il risultato di una singola porta logica si può calcolare in tempo costante, la presunta correttezza di un certificato si può verificare in tempo polinomiale, da cui l'appartenenza alla classe NP.

Ma è anche NP completo? Negli anni '70 Cook presentò una dimostrazione non banale, di diverse pagine di lunghezza, in quanto dovette dimostrare che tutti i problemi sono riducibili a CIRCUIT-SAT. Ciò non significa che ogni volta che vogliamo dimostrare che un problema sia NP completo dobbiamo seguire le orme di Cook e verificare che tutti i problemi NP siano riducibili ad esso, ma possiamo usare la seguente tecnica.

Supponiamo di avere un problema \mathcal{P} , e vogliamo dimostrare che sia NP completo. Allora,

1. Sicuramente, bisognerà dimostrare che il problema è verificabile in tempo polinomiale, cioè $\mathcal{P} \in \text{NP}$.
2. Se riusciamo a dimostrare che esiste un problema $Q \in \text{NPC}$ tale che $Q \leq_P \mathcal{P}$, allora avremo dimostrato che \mathcal{P} è NP completo.

È infatti sufficiente dimostrare l'esistenza di un singolo problema NP completo che soddisfi la seconda condizione per via della transitività della riducibilità polinomiale tra problemi:

$$\exists Q \in \text{NPC} \text{ s.t. } Q \leq_P \mathcal{P} \Rightarrow \forall \mathcal{P}' \in \text{NP} : \mathcal{P}' \leq_P \mathcal{P} ?$$

Si ha infatti che

$$\underbrace{\mathcal{P}' \leq_P \underbrace{Q \leq_P \mathcal{P}}_{\substack{\text{Per ipotesi} \\ \text{Def. NPC}}}}$$

22.3.2 Il problema SAT

Esiste un altro problema concettualmente simile al CIRCUIT-SAT, chiamato **SAT**. Se il primo si interroga sulla soddisfacibilità di circuiti logici, il secondo indaga sulla soddisfacibilità di formule logiche come la seguente.

$$\Phi = ((x_1 \implies x_2) \vee \neg((\neg x_1 \iff x_3) \vee x_4) \wedge \neg x_2)$$

Per questa formula, esiste una configurazione di x_1, x_2, x_3 e x_4 tale che il risultato sia 1? Sicuramente, x_2 dovrà valere 0 perché negata nell'AND alla fine della formula; quindi, per via dell'OR, basta che $(x_1 \implies x_2)$ valga 1: poniamo quindi x_1 a 0, dal momento che falso implica falso è vero, e possiamo scegliere x_3 e x_4 arbitrariamente.

Una possibile configurazione che soddisfa la formula è $x_1 = 0, x_2 = 0, x_3 = 1$ e $x_4 = 1$.

Si può dimostrare che anche SAT è NP completo, tuttavia non è di nostro interesse; per farlo è sufficiente dimostrare che è polinomialmente verificabile (il suo certificato sarà una delle 2^n configurazioni delle variabili), e che il CIRCUIT-SAT, che sappiamo essere NP completo per il teorema di Cook, è polinomialmente riducibile a SAT.

22.3.3 Forma normale congiuntiva, i problemi 3-SAT e CLIQUE

Quando abbiamo a che fare sull'algebra booleana, è comodo lavorare con certi formati particolari di formule booleane dotate di una struttura precisa. Tra le diverse formule normali, esamineremo la **forma normale congiuntiva** o FNC, cioè una formula costituita da una congiunzione di diverse clausole.

$$\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

Nel nostro esempio, Φ costituisce una congiunzione, e C_1, C_2, C_k sono le k clausole che la compongono. Ogni clausola è una disgiunzione di letterali, cioè di variabili o negazioni di variabili.

$$C = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_q$$

Date le suddette definizioni, un esempio di congiunzione può essere la seguente formula:

$$\Phi = \underbrace{(x_1 \vee \underbrace{\neg x_2}_{\text{Letterale}}) \wedge \underbrace{(x_2 \vee x_3 \vee \neg x_4)}_{\text{Clausola}} \wedge \underbrace{(x_1 \vee x_4)}_{\text{Letterale}}}_{\text{Congiunzione}} \wedge (x_2 \vee x_4)$$

Per verificare la soddisficiabilità di Φ è necessario che tutte le clausole siano vere (per via dell'AND) e che all'interno delle clausole almeno un letterale sia vero (per via dell'OR).

Il **formato k -FNC** è una specializzazione della forma normale congiuntiva in cui tutte le clausole hanno esattamente k letterali. Ai fini del nostro studio, ci interessa il caso in cui $k = 3$: tramite questa forma normale, possiamo analizzare il cosiddetto problema **SAT-3FNC**, variante del SAT che si occupa di verificare la soddisficiabilità di formule nella forma 3-FNC.

Analogamente al SAT, si può verificare che SAT-3FNC è NP completo, dimostrando che appartiene a NP e che SAT è riducibile a SAT-3FNC. Aldilà della dimostrazione, la seconda tesi è molto importante: ci dice che qualunque formula in formato arbitrario può essere sempre ridotta in tempo polinomiale in formato 3FNC, ed è infatti in questo modo che i compilatori dei linguaggi moderni interpretano le formule booleane dei programmi che compilano.

Vale banalmente che anche k -FNC è riducibile a 3-FNC per un qualsiasi $k \geq 3$, ma è interessante notare il caso in cui $k = 2$: il problema SAT-2FNC, infatti, appartiene alla classe P; ciò significa che esistono certi problemi per cui, sotto una certa soglia, il problema è risolvibile polinomialmente, mentre oltre tale soglia il problema è solamente verificabile in tempo polinomiale.

Tutta questa digressione sui problemi relativi all'algebra booleana ci è servita per poter dimostrare che il problema della *CLIQUE* è NP completo. Per dimostrarlo, è necessario verificare due proprietà:

1. *CLIQUE* ∈ NP, ma questo l'abbiamo già dimostrato: *CLIQUE* è verificabile in tempo quadratico.
2. Possiamo scegliere uno dei tre problemi appena incontrati, cioè CIRCUIT-SAT, SAT o SAT-3FNC, e dimostrare che sono tutti e tre riducibili al problema della *CLIQUE*. Sceglieremo arbitrariamente il problema SAT-3FNC.

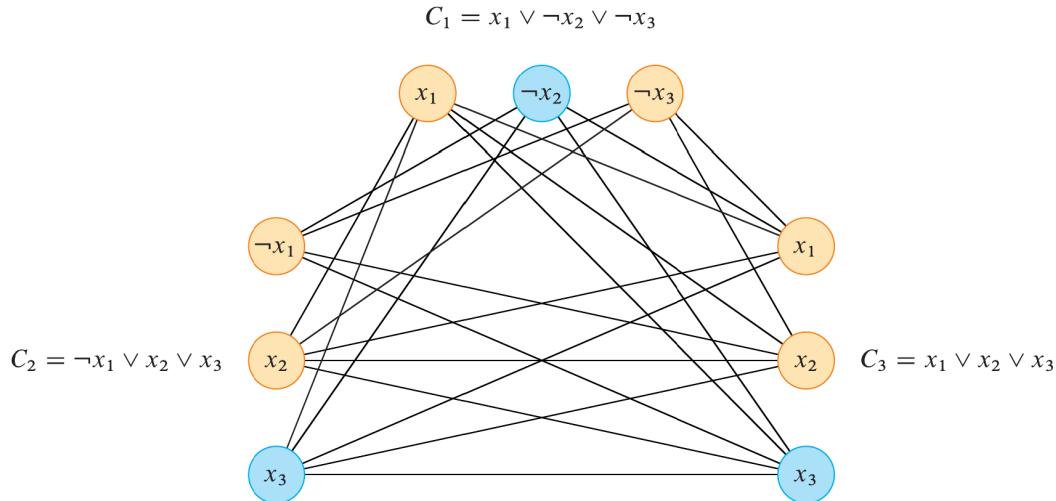
Vediamo quindi come trasformare un problema SAT-3FNC in un problema *CLIQUE*. Prendiamo una generica istanza del problema SAT-3FNC:

$$\begin{aligned}\Phi &= C_1 \wedge C_2 \wedge C_3 \\ C_1 &= x_1 \vee \neg x_2 \vee \neg x_3 \\ C_2 &= \neg x_1 \vee x_2 \vee x_3 \\ C_3 &= x_1 \vee x_2 \vee x_3\end{aligned}$$

È possibile prendere questa congiunzione e trasformarla in un'istanza di *CLIQUE*, cioè un grafo di tanti sottinsiemi di vertici quante sono le clausole di Φ , e ciascun sottinsieme ha k vertici. Gli archi si possono costruire unicamente rispettando due condizioni:

1. Non ci potrà mai essere un arco tra i letterali dello stesso gruppo. Se c'è un arco, deve collegare vertici di gruppi diversi.
2. Non ci può essere un arco tra due letterali che sono uno la negazione dell'altro.

Il risultato finale è il seguente grafo.



La congiunzione Φ è soddisfacibile se e solo se esiste una *clique* di grado 3 nel grafo risultante.

Prendiamo, per esempio, la *clique* formata da $\neg x_2$ nel sottinsieme C_1 , x_3 nel sottinsieme C_2 e x_3 nel sottinsieme C_3 (i vertici evidenziati in azzurro nel grafo): se decidiamo di dare il valore 1 ai letterali non negativi e il valore 0 ai letterali negativi che costituiscono la *clique*, avremo che con $x_2 = 0$, $x_3 = 1$ e x_1 scelto arbitrariamente, la congiunzione è verificata. Infatti, $x_2 = 0$ soddisfa la clausola C_1 , e $x_3 = 1$ soddisfa le clausole C_2 e C_3 .

La scelta di questi vertici è stata a nostra discrezione: potevamo scegliere, invece di x_3 in C_2 , $\neg x_1$ in C_2 ed avere un'altra soluzione costituita dai letterali $x_1 = 0$, $x_2 = 0$, $x_3 = 1$.

In questo modo siamo riusciti a predire il risultato di questa congiunzione senza nemmeno guardarla, se non per costruire il grafo, e a dimostrare la NP completezza del problema *CLIQUE*; notare che in questo modo è possibile dimostrare la NP completezza di molti altri problemi, tra cui quello del ciclo Hamiltoniano.

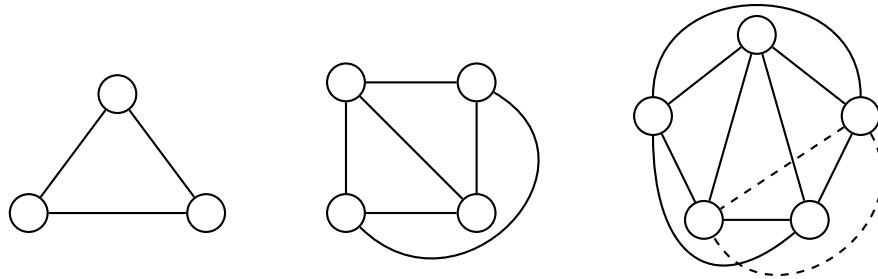
22.4 Cosa fare davanti ad un problema intrattabile

1. Fare ricorso ad **algoritmi di approssimazione**, cioè algoritmi che non necessariamente trovano una soluzione corretta, però forniscono delle soluzioni che non sono così lontane dalla soluzione ottimale. Le soluzioni di questo tipo vengono dette ε -ottimali, cioè vicine alla soluzione reale con un margine d'errore pari a ε . Esistono infatti problemi NP completi che però possono essere approssimati molto bene in tempo polinomiale, come il problema del *TSP*.

Esistono tuttavia problemi, come quello della *clique* e quello della *clique* massima, la cui approssimazione è pure NP completa: cosa fare in questi casi?

2. Sperare che il nostro problema sia un **caso speciale**: il problema della *clique* è, per certi grafi speciali detti **grafi planari**, risolvibile polinomialmente.

Un grafo planare è un grafo in cui, disegnandolo, gli archi non si intersecano; verifichiamo se K_3 , K_4 e K_5 sono grafi planari.



Notiamo che K_5 è il primo grafo completo non planare: il teorema di Kuratowski afferma infatti che tutti i grafi che contengono una *clique* di 5 vertici sono non planari.

Sebbene il problema della *clique* sia NP completo, si può risolvere su grafi planari in tempo polinomiale: si può dimostrare che Planar-*Clique* $\in P$, e la *clique* risultato potrà avere al massimo 4 vertici.

3. Fare affidamento a **euristiche**. Un'euristica è un algoritmo per risolvere un certo problema, che però non ha nessuna garanzia teorica di correttezza, e per questo motivo costituiscono l'ultima spiaggia davanti a problemi intrattabili. L'unico modo che abbiamo per fidarci di questi algoritmi è tramite la sperimentazione: siccome la teoria non ci aiuta, dobbiamo effettuare verifiche sperimentali, lanciando l'algoritmo su migliaia di istanze e verificando l'accuratezza dei risultati, conoscendoli a priori.

A Appendici

A.1 *Worst-case scenario* e algoritmo del simplex

Come abbiamo ripetuto per tutta la durata del corso, quando parliamo di complessità di un algoritmo la intendiamo quasi sempre in termini della classe asintotica O grande: se diciamo che un nostro algoritmo ha complessità $O(n^k)$ per una qualche costante k , significa che nel caso peggiore quell'algoritmo verrà eseguito in un tempo al più stimabile alla funzione n^k , e questo vale per qualsiasi esempio di funzione che ha costituito un limite superiore alla complessità dei nostri algoritmi.

Nella realtà, tuttavia, non accade spesso che un algoritmo impieghi tutto il tempo che gli definiamo ponendogli tale limite superiore: potremmo essere fortunati e assistere ad un'esecuzione molto più breve di quella che ci siamo prefissati, e l'algoritmo del simplex ne è un esempio.

L'algoritmo del simplex è un algoritmo sviluppato dall'americano George Dantzig nel 1947 per risolvere problemi di programmazione lineare, quella branca della ricerca operativa che si impegna a sviluppare algoritmi di problemi di ottimizzazioni lineari soggetti a vincoli lineari.

Un problema di programmazione lineare può essere il voler minimizzare la seguente funzione, soggetta ai seguenti vincoli.

$$f(x, y, z) = 3x - 2y + 5z, \quad \text{Vincoli: } \begin{cases} 4x + 4y = 6 \\ x - 6y \leq 7 \\ x, y, z \geq 0 \end{cases}$$

Essendo i vincoli una limitazione della regione ammissibile per la funzione, Dantzig scoprì che la soluzione del problema si può trovare proprio all'interno di tale regione, per cui è possibile "saltellare" da un vertice all'altro di tale regione fino a trovarla.

Per questo problema, è possibile creare delle istanze tali che l'algoritmo risolutivo impieghi tempo esponenziale, per esempio generando un numero esponenziale di vertici, al che l'algoritmo dovrebbe visitarli tutti.

Formalmente, dunque, l'algoritmo è esponenziale, in quanto stiamo parlando del *worst-case scenario*, tuttavia la grande maggioranza delle istanze di questo problema si può risolvere in tempo lineare, e per questo motivo tale algoritmo è largamente utilizzato.

Il paradosso avvenne quando si scoprì che tale algoritmo ha effettivamente complessità esponenziale, e si cercò un algoritmo alternativo che risolvesse il problema polinomialmente. Negli anni '60 tale algoritmo fu scoperto, però utilizzava un'aritmetica floating-point che lo rendeva poco stabile e accurato quando il numero di step diventava troppo alto, soprattutto davanti all'aritmetica floating-point finita dei calcolatori.

Per questo motivo, nonostante la complessità esponenziale del caso peggiore, algoritmi come quello del simplex e altri ancora sono ugualmente utilizzati nel mondo reale.