UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Appunti di

Algoritmi e Strutture Dati

Si può fare meglio di così?

Autore

Emanuele Nardi

Revisori

Francesco Bozzo

Samuele Conti

Anno Accademico 2019/2020

A tutti coloro che lavorano per lasciare il mondo
un po' meglio di come lo hanno trovato.

Prologo

Caro studente,

ho lavorato a questi appunti cercando di offrirti l'esperienza di lettura e di comprensione migliore possibile, rileggendo più volte, correggendo errori e facendole revisionare da alcuni tuoi ex-compagni di corso che mi hanno aiutato a portarti un lavoro corretto e revisionato.

Ho cercato di rispettare la seguente affermazione tratta da "Pensare in Python, Come pensare da Informatico": Il commento più utili sono quelli che documentano caratteristiche del codice di non immediata comprensione. È ragionevole supporre che chi legge il codice possa capire *cosa* esso faccia; è più utile spiegare *perché*.

Questa affermazione è sicuramente vera per coloro che hanno già delle forti basi di programmazione, per mia sfortuna e per fortuna per voi, questo non era il mio caso, quindi ogni algoritmo è stato analizzato, scomposto e sviscerato di ogni sua componente fino ad arrivare al suo principio.

Non ho dato nessun preconcetto per scontato e tutti i passaggi, matematici e logici, sono stati resi espliciti in ogni momento possibile.

Non soffermarti ad una lettura passiva del testo, il quale dovrebbe essere una traccia delle lezioni, cerca invece a provare gli algoritmi prima di entrare in aula e ascoltare una lezione frontale, che privo della comprensione di quel che stai vedendo, potrebbe risultare in una completa perdita di tempo.

I seguenti appunti sono una traccia delle lezioni di Algoritmi e Strutture Dati, il mio consiglio per affrontare la materia è quella di leggere prima l'argomento e andare a lezione con le idee ben chiare, in questo modo ti verrà spontaneo fare domande su ciò che non ti è chiaro o che eventualmente non hai capito avendo già toccato quell'argomento da solo. Non aver paura di far domande! Il professore è super disponibile ed è uno dei migliori del nostro ateneo, non si farà problemi a risponderti e, anche se la domanda sarà banale, è molto probabile che tu non sia l'unico ad avere quel dubbio. Probabilmente farai un favore a qualcuno che è più timido di te! E, cosa molto più importante, interagendo trarrai molto più beneficio dalla lezione frontale in quanto porterai a casa un'esperienza individuale e ti sarà più semplice ricordarti quale fosse il tuo dubbio e la relativa risposta in sede d'esame.

Dall'anno accademico 2018/2019 il corso di Algoritmi è passato dall'essere un corso semestrale ad un corso annuale, questo ha dato a molti studenti la possibilità di avere più tempo per interiorizzare le molte nozioni richieste per una comprensione piena degli argomenti trattati, ma al contempo quando nel mese di marzo riprenderai la materia in mano (che tu abbia affrontato il primo parziale o meno) è il caso che tu riprenda in mano gli argomenti del primo semestre, anche solo per una rilettura veloce.

Se trovi errori di qualsiasi natura non indugiare a segnalarmeli, li correggerò il prima possibile e aggiornerò l'errata corrige, inoltre (se ti facesse piacere) verrai menzionato vicino all'errore segnalato. Una versione pubblica e aggiornata dell'errata corrige sarà presente alla pagina Errata corrige nella Wiki del progetto.

Vorrei ringraziare Francesco Bozzo, e Samuele Conti per aver revisionato questa dispensa, per la considerevole pazienza e meticolosità delle correzioni. Gli errori rimanenti sono, ovviamente, interamente miei. Le carenze di questa dispensa sarebbero considerevolmente maggiori e più numerose se non fosse stato per la loro assistenza.

Come ultima battuta vorrei ringraziare tutti coloro che mi hanno sostenuto in questi mesi per la stesura di questo testo e a coloro senza i quali sarebbe rimasti dei semplici appunti.

Ti auguro buono studio e di passare l'esame a pieni voti!

Emanuele

Indice

Elenco delle figure

Elenco delle tabelle

Parte I

Primo semestre

Capitolo 1

Introduzione

Definizione (problema computazionale)**.** Dato un dominio di input e un dominio di output, un *problema computazionale* è rappresentato dalla **relazione matematica** che associa un elemento del dominio in output ad ogni elemento del dominio di input.

Definizione (algoritmo)**.** Dato un problema computazionale, un *algoritmo* è un procedimento **effettivo**, espresso tramite un insieme di **passi elementari ben specificati** in un sistema **formale** di calcolo, che risolve il problema in tempo **finito**.

Un esempio classico, ma ingannevole, di algoritmo è quello della preparazione di una ricetta: l'input sono gli ingredienti, l'esecutore è il cuoco, l'algoritmo è la ricetta e l'output è rappresentato dal piatto cucinato. È ingannevole perché non esiste un modello formale del cuoco, ossia qualcosa che descriva esattamente cosa un cuoco può fare e quali sono i passi elementari dell'algoritmo.

Algoritmi nella storia

Gli algoritmi sono molto antichi ed esistevano ancor prima del concetto di elaboratore.

Nè è un esempio il Papiro di Rhind o di Ahmes che mostra l'algoritmo del contadino per la moltiplicazione. Algoritmi di tipo numerico furono studiati da matematici babilonesi ed indiani. Esistono algoritmi in uso fino a tempi recenti che furono studiati dai matematici greci più di 2000 anni fa. Come ad esempio l'algoritmo di Euclide per il massimo comune divisore e gli algoritmi geometrici come il calcolo di tangenti, sezioni di angoli, etc.

Origine del nome

L'*origine del nome* ***algoritmo*** è dovuto al matematico, astronomo, astrologo e geografo Abu Abdullah Muhammad bin Musa **al-Khwarizmi**. Ha scritto un testo chiamato "Algoritmi de numero indorum", traduzione di un testo arabo ormai perso, che ha introdotto i numeri indiani (da noi comunemente detti numeri arabi) nel mondo occidentale.

Ha inoltre scritto un'altra opera "Al-Kitab al-muhtasar fi hisab **al-gabr** wa-l-muqabala" che è stata tradotta in latino come "Liber algebrae et almucabala" dalla quale ha avuto *origine il nome* ***algebra***.

1.1 Primi esempi di problemi

Per iniziare a comprendere ciò che intendiamo per problema computazionale iniziamo a vedere degli algoritmi che sono espressi in modo volutamente banale.

1.1.1 Minimo

Definizione del problema Il minimo di un insieme S è l'elemento di S che è minore o uguale ad ogni altro elemento di S. Possiamo esprimere matematicamente il problema nel seguente modo:

$$min(S) = a \Leftrightarrow \exists a \in S : \forall b \in S : a \leqslant b$$

Partiamo da una soluzione semplice ma ingenua, che deriva dalla definizione del problema.

Algoritmo naïf Per trovare il minimo di un insieme, confronto ogni elemento con tutti gli altri; l'elemento che è minore di tutti è il minimo.

A questo punto ci accorgiamo dell'ambiguità della lingua italiana, infatti che cosa s'intende per "confronta ogni elemento con tutti gli altri" ? Non abbiamo un insieme di passi elementari ben specificati.

1.1.2 Ricerca dell'indice di un elemento

Definizione del problema Sia S una sequenza di dati $s_1, s_2, \ldots, s_n$ ordinati e distinti, ad esempio $s_1 > s_2 > \cdots > s_n$. Eseguire una ricerca della posizione di un dato valore v nell'insieme S consiste nel restituire un indice i tale che $1 \leqslant i \leqslant n$ (ossia che sia compreso fra 1 ed n, estremi inclusi), se v è presente nella posizione i, oppure 0, se v non vi è presente. Possiamo esprimere matematicamente il problema nel seguente modo:

$$lookup(S, v) = \begin{cases} i & \exists i \in \{1, \ldots, n\} : S_i = v \\ 0 & \text{altrimenti} \end{cases}$$

Notiamo che, avendo caratterizzato la sequenza in modo tale che abbia *elementi distinti*, abbiamo semplificato il problema al caso in cui non ci sono ambiguità nella restituzione dell'indice.

Algoritmo naïf Per trovare il valore v nella sequenza S, confronto v con tutti gli elementi di S, in sequenza, e restituisco la posizione corrispondente; restituisco 0 se nessuno degli elementi corrisponde.

1.1.3 Problemi riscontrati

Le descrizioni degli algoritmi precedenti presentano diversi problemi:

- la **descrizione** dei problemi è stata fatta in linguaggio naturale, il quale è intrisicamente ambiguo, abbiamo quindi bisogno di un linguaggio più formale.
- come possiamo **valutare** se esistono algoritmi migliori di quelli proposti? Per farlo dobbiamo definire il concetto di "migliore".

1.1.4 Come descrivere un algoritmo

Come abbiamo già anticipato è necessario avere una descrizione il più possibile formale, ma senza soffermarsi sulle particolarità di un determinato linguaggio di programmazione. Per questo motivo nel seguito utilizzeremo lo "Pseudo-codice".

In alcuni casi daremo anche per scontato alcuni passaggi, ad esempio scriveremo "ordina gli elementi" e non sarà di nostro interesse sapere quale sarà lo specifico algoritmo di ordinamento utilizzato.

Esempi di Pseudo-codice sono i seguenti.

Implementazione naïf della ricerca del minimo

```
int min(int[] S, int n)
    from i ← 1 until n do // per ogni elemento del vettore
        boolean isMin ← true // assumo di aver trovato il minimo
        from j ← 1 until n do // confronto l'elemento con tutti gli altri
            if i ≠ j and S[j] < S[i] then // se trovo un valore più piccolo
                isMin ← false // quell'indice non contiene l'elemento minimo
        // se dopo aver controllare ogni indice...
        if isMin then // ...trovo un valore che ha conservato il valore true
            return S[i] // l'elemento in posizione i-esima è il più piccolo
```

Nota che specificheremo sempre com'è fatto l'input, ossia passeremo in ingresso sia il vettore (S), sia la lunghezza del vettore passato (n).

Implementazione naïf della ricerca dell'indice di un elemento

```
int lookup(int[] S, int n, int v)
    from i ← 1 until n do // per tutti gli elementi del vettore
        if S[i] == v then // confronto tutti i valori con quello cercato
            return i // se lo trovo lo restituisco
    return 0 // altrimenti restituisco 0
```

I linguaggi di programmazione assumono che l'indice dei vettori parta da 0, ma per spiegarne alcuni è molto più semplice concentrarsi sugli indici che vanno da 1 a n e tralasciare questi dettagli implementativi.

Convenzioni dello pseudocodice

Di seguito sono riportate le convenzioni utilizzate sui lucidi e sugli appunti:

- l'assegnamento verrà indicato con $a \leftarrow b$;
- uno *swap* verrà indicato come $a \leftrightarrow b$, il quale corrisponde ai comandi $tmp \leftrightarrow a$; $a \leftarrow b$; $b \leftarrow tmp$;
- i vettori come $\mathbf{T}[\,]\ A = \mathsf{new}\ \mathbf{T}[1 \dots n]$;
- le matrici come $\mathbf{T}[\,][\,]\ A = \mathsf{new}\ \mathbf{T}[1 \dots n][1 \dots n]$;
- i tipi come **int**, **float**, **boolean**, ...;
- i simboli logici come **and**, **or**, **not**;
- i simboli relazionali con $==$, $\neq$, $\leqslant$, $\geqslant$;
- i simboli matematici con $+$, $-$, $\cdot$, $/$, $\lfloor x \rfloor$, $\lceil x \rceil$, log, x^2, ...;
- talvolta scriveremo l'operatore if ternario come $\mathsf{iif}(condizione,\ v_1,\ v_2)$;
- **if** *condizione* **then** *istruzione*;
- **foreach** *elemento* $\in$ *insieme* **do** *istruzione*.

1.2 Come valutare un algoritmo

Quando valutiamo un algoritmo dobbiamo chiederci se risulta *corretto* ed *efficiente*.

Per quanto riguarda l'efficienza dobbiamo ancora stabilire come valutare se un programma è efficiente, e se lo è in assoluto. Nota che alcuni problemi non possono essere risolti in modo efficiente ma esistono soluzioni "ottime", ossia non è possibile essere più efficienti di così.

Per controllare la correttezza dobbiamo domandarci se il nostro algoritmo rispetta la relazione input-output del problema computazionale. Nota che alcuni problemi non possono essere risolti, mentre alcuni vengono risolti in modo approssimato.

1.2.1 Efficienza

Definizione (complessità di un algoritmo)**.** Analisi delle *risorse* impiegate da un algoritmo per risolvere un problema, in funzione della *dimensione* e della *tipologia* dell'input.

Le risorse si possono categorizzare in:

- **tempo**: ossia il tempo impiegato per completare l'algoritmo (definiremo più avanti come lo misureremo);

- **spazio**: la quantità di memoria utilizzata;
- **banda**: la quantità di bit spediti (interessante solo per gli algoritmi distribuiti).

Definizione di tempo

Il tempo effettivamente impiegato per eseguire un algoritmo dipende da troppi parametri come la bravura del programmatore, il linguaggio di programmazione utilizzato (C è più efficiente di Python), il codice generato dal compilatore, dalla velocità del processore e della memoria e dai processi attualmente in esecuzione sul sistema operativo.

È quindi necessario considerare una rappresentazione più astratta. Ad esempio potremmo considerare il *numero di operazioni "rilevanti"*, ovvero il numero di operazioni che caratterizzano lo scopo dell'algoritmo.

1.2.2 Primi esempi di calcolo della complessità

Nel caso della ricerca del minimo, l'operazione più rilevante è il numero di confronti di minoranza (<), nella caso della ricerca dell'indice invece è il numero di confronti di eguaglianza (==).

Implementazione naïf della ricerca del minimo

```
int min(int[] S, int n)
    from i ← 1 until n do // ciclo con n elementi
        boolean isMin ← true
        from j ← 1 until n do // ciclo con n elementi
            if i ≠ j and S[j] < S[i] then // il confronto avviene solo se gli indici sono diversi
                isMin ← false
        if isMin then
            return S[i]
```

Calcolo della complessità Come calcoliamo la complessità? L'operazione rilevante all'interno di questo algoritmo è il confronto fra gli elementi agli indici i e j ($S[j] < S[i]$), la quale è ripetuta all'interno di due cicli annidati che scorrono gli n elementi del vettore, tranne quelli che hanno lo stesso indice. Quindi il numero di confronti totali per questa soluzione del problema del minimo è $n \cdot n - n = n^2 - n$. Si può fare meglio di così? Certo che sì!

Implementazione efficiente per la ricerca del minimo

```
int min(int[] S, int n)
    int min ← S[1] // minimo parziale
    from i ← 2 until n do // dal secondo elemento in poi...
        if S[i] < min then // ...confronto il minimo parziale con l'elemento corrente
            min ← S[i] // aggiorno il minimo parziale
    return min // restituisco il minimo trovato
```

Con questa implementazione effettuiamo $n - 1$ confronti perché il primo elemento non deve essere confrontato.

Questo algoritmo effettua n confronti ($S[i] == v$), uno per ogni elemento del vettore. Si può fare meglio di così?

Implementazione naïf della ricerca dell'indice di un elemento

```
int lookup(int[] S, int n, int v)
    from i ← 1 until n do
        if S[i] == v then
            return i

    return 0
```

Sfruttando il fatto che la *sequenza è ordinata* possiamo applicare la **ricerca binaria**. Prendiamo l'elemento centrale (di indice m) del sottovettore considerato: se l'elemento contenuto all'indice m è pari all'elemento cercato ($A[m] == v$) allora ho trovato il valore e lo restituisco al chiamante, altrimenti se l'elemento contenuto all'indice m è più piccolo dell'elemento cercato ($A[m] < v$) allora dovrò continuare la ricerca nella "metà di destra" ($m+1$, j), infine se l'elemento contenuto all'indice m è più grande dell'elemento cercato ($A[m] > v$) dovrò continuare la ricerca nella "metà di sinistra" (i, $m-1$). Nel caso sfortunato in cui l'elemento non esistesse all'interno del vettore gli indici si incroceranno, ed è questo che il caso base controlla all'inizio di ogni chiamata della funzione.

Algoritmo 1.2.1: Ricerca binaria all'interno di un vettore

```
// Effettua una ricerca binaria su un vettore di lunghezza arbitraria
binarySearch(ITEM[] A, ITEM v, int i, int j)
    if i > j then // se i cursori si incrociano
        return 0 // l'elemento non esiste
    else
        int m ← ⌊(i+j)/2⌋ // assegna la mediana
        if A[m] == v then // abbiamo trovato l'elemento
            return m // lo restitisco al chiamante
        else if A[m] < v then// se l'elemento cercato è più grande
            return binarySearch(A, v, m+1, j) // cerco nella seconda metà
        else // altrimenti
            return binarySearch(A, v, i, m-1) // cerco nella prima metà
```

Complessità Ad ogni passo il numero di elementi che considero viene dimezzato, quindi la complessità è logaritmica (più precisamente $\lceil \log \rceil$).

Nota. Tutte le volte che scriveremo un logaritmo sarà da intendere con base 2. Siamo informatici!

1.2.3 Correttezza

Per valutare la correttezza di un algoritmo possiamo utilizzare il concetto di invariante.

Definizione (invariante)**.** Condizione sempre vera *in un certo punto* del programma.

Definizione (invariante di ciclo)**.** Una condizione sempre vera all'inizio dell'iterazione di un ciclo.

Definizione (invariante di classe)**.** Una condizione sempre vera al termine dell'esecuzione di un metodo della classe.

Facciamo un esempio di invariante di ciclo, in quanto la utilizzeremo più avanti nella nostra trattazione.

Invariante di ciclo

Il concetto di **invariante di ciclo** ci aiuta a dimostrare la correttezza di un **algoritmo iterativo**. Distinguiamo tre fasi:

1. **inizializzazione** (caso base): la condizione è vera alla prima iterazione di un ciclo;
2. **conservazione** (passo induttivo): se la condizione è vera prima di un'iterazione del ciclo, allora rimane vera anche al termine (quindi prima della successiva iterazione);
3. **conclusione**: quando il ciclo termina, l'invariante deve rappresentare la "correttezza" dell'algoritmo.

Invariante di ciclo nella ricerca del minimo

Prendiamo ancora un'ultima volta come esempio l'algoritmo della ricerca del minimo.

L'invariante di ciclo per questo algoritmo è la seguente: all'inizio di ogni iterazione del ciclo, la variabile min contiene il minimo parziale degli elementi $S[1 \dots i-1]$. Quindi quando analizzo l'elemento i-esimo so che min contiene il minimo fra tutti gli elementi precendenti.

Inizializzazione (caso base): quando entriamo nel ciclo la variabile min contiene il minimo fra $S[1 \dots i-1]$ ed è vero poiché $i = 2$ quindi $i - 1 = 1$ e la variabile contiene il minimo fra il primo elemento ed il primo elemento. **Conservazione** (passo induttivo): nel momento in cui termino il ciclo la variabile min contiene il minimo parziale fra i ed n. **Conclusione**: al termine del ciclo i è pari a $n+1$, quindi la variabile min contiene il minimo parziale fra gli elementi compresi fra 1 e $i-1$, ma visto che i vale $n+1$, allora conterrà gli elementi il minimo fra gli elementi compresi fra $n+1-1 = n$. L'invariante di ciclo risulta quindi rispettata e rappresenta la correttezza del nostro algoritmo.

Questa analisi risulta eccessivamente dettagliata per un algoritmo così semplice, ma in futuro dovremo utilizzare l'invariante di ciclo per dimostrare algoritmi più complessi, è quindi di fondamentale importanza impararne i prìncipi.

Dimostrazione per induzione nella ricerca binaria

La **dimostrazione per induzione** è utile anche nel caso in cui si abbia a che fare con un **algoritmo ricorsivo**.

Infatti è possibile dimostrare la correttezza della ricerca binaria per induzione sulla dimensione n dell'input. Dove n è il numero di elementi passati alla funzione.

Il *caso base* si avvera quando non passiamo nessun elemento alla funzione ($n = 0$), ossia quando gli indici si sono incrociati ($i > j$) poiché non abbiamo trovato l'elemento cercato all'interno del vettore. Per *ipotesi* supponiamo che l'algoritmo sia corretto per tutti i valori n' più piccoli di n. Il *passo induttivo* è quindi costituito da tre casi:

1. trovo l'elemento e lo restituisco (il caso più semplice, $A[m] == v$);
2. l'elemento confrontato è più piccolo di quello che sto cercando ($A[m] < v$) e, in quanto i valori sono ordinati, l'elemento cercato si troverà sicuramente nella metà di destra delimitata dagli indici $m+1$ e j, i quali determinano una porzione del vettore n' più piccola del vettore di partenza, quindi l'algoritmo risulta corretto per induzione;
3. si ragiona in modo speculare al secondo caso con $A[m] > v$.

Conclusioni

Noi abbiamo analizzato soltato due aspetti degli algoritmi, ossia la loro *correttezza* ed *efficienza*. Ci sono tanti altri aspetti che si potrebbero guardare come ad esempio la semplicità, la modularità, la mantenibilità, l'espandibilità e la robustezza. I quali risultano però di secondaria importanza per un corso di algoritmi, e vengono trattati in modo approndito in un corso di ingegneria del software.

Alcune proprietà hanno un costo aggiuntivo in termini di prestazioni: ad esempio per scrivere codice modulare dobbiamo pagare il costo della gestione delle chiamate o per scrivere codice Java dobbiamo pagare il costo di interpretazione. Progettare algoritmi efficienti è quindi un prerequisito per poter pagare questo costo.

Capitolo 2

Analisi di algoritmi

2.1 Introduzione

Il nostro obiettivo è *stimare la complessità in tempo* degli algoritmi. Dovremmo stimare anche quella in spazio, ma la complessità in spazio dipende da quella in tempo. Daremo delle definizioni, parleremo di modelli di calcolo, faremo qualche esempio di valutazione precisa e introdurremo una notazione. Faremo tutto questo per stimare il tempo per un dato input, per stimare il più grande input gestibile in tempi ragionevoli, per avere un metodo di confronto fra diversi algoritmi ed in particolare per ottimizzare le parti più importanti di un particolare algoritmo.

Definizione di complessità

La complessità viene definita come una funzione che, data la dimensione dell'input, restituisce il tempo, considerato come un valore intero.

Come definiamo quindi la dimensione dell'input? Come misuriamo il tempo?

2.2 Valutare la dimensione dell'input

Esistono due criteri per valutare la dimensione dell'input:

1. il criterio di **costo logaritmico**: dove la dimensione dell'input è il numero di bit necessari per rappresentarlo (un esempio è la moltiplicazione di numeri binari, lunghi n bit);
2. il criterio di **costo uniforme**: dove la dimensione dell'input è il numero di elementi di cui è costituito (un esempio è la ricerca del minimo in un vettote di n elementi).

Ad esempio, consideriamo n interi rappresentati tramite 32 bit. Nel criterio di costo uniforme hanno un costo pari ad n, mentre nel criterio di costo logaritmico hanno un costo pari a $32n$. In molti casi, infatti, possiamo assumere che gli "elementi" siano rappresentati da un numero costante di bit e che le due misure coincidano a meno di una costante moltiplicativa.

Il criterio che abbiamo utilizzano fin'ora — e che useremo d'ora in poi — è il criterio del costo uniforme (in casi particolari utilizzeremo il criterio di costo logaritmico).

2.3 Misurare il tempo

Consideriamo un'istruzione come elementare se può essere eseguita in tempo "costante" dal processore. Facciamo qualche esempio:

- `a *= 2` effettua un'operazione di *shift*, è una singola operazione macchina;
- `Math.cos(d)` può essere considerata come un'operazione elementare;
- `min(A, n)` non può essere considerata un'operazione elementare poiché si richiede il minimo di un vettore *arbitrariamente lungo*.

Ma allora come possiamo distinguere in maniera precisa un'operazione elementare da una che non lo è?

Per farlo abbiamo bisogno di un modello di calcolo, ossia una rappresentazione astratta di un calcolatore. Il quale deve:

1. permettere di nascondere i dettagli (tramite *astrazione*)
2. riflettere la situazione reale (*realismo*)
3. permettere di trarre conclusioni "formali" sul contesto.

La pagina di Wikipedia dei modelli di calcolo presenta centinaia di modelli di calcolo diversi. La macchina di Turing ne è un esempio. È una macchina ideale che manipola, seconodo un insieme prefissato di regole, i dati contenuti su un nastro di lunghezza infinita. Ad ogni passo, la Macchina di Turing:

1. legge il simbolo sotto la testina;
2. modifica il proprio stato interno;
3. scrive il nuovo simbolo nella cella;
4. muove la testina a destra o a sinistra.

Nel corso di laurea magistrale è possibile approfondire questo aspetto, per i nostri scopi questa è una trattazione dell'argomento troppo a basso livello.

Noi utilizzeremo il modello di calcolo RAM, che sta per Random Access Machine, ossia una macchina che ha una quantità infinita di celle (di dimensione finita) e accesso in tempo costante indipendentemente dalla posizione (diversamente da ciò che avviene nei nastri); un singolo processore con un set di istruzioni simile a quelli reali, i cui costi di esecuzione sono uniformi e ininfluenti ai fini della valutazione (faremo un esempio più avanti).

2.3.1 Calcolo della complessità

Proviamo a calcolare la complessità dell'algoritmo che ricerca il minimo.

Calcolo della complessità della ricerca del minimo in un vettore

```
min(ITEM[] A, int n)                          Costo   # Volte
    ITEM min ← A[i]                           c1      1
    from i ← 2 until n do                     c2      n
        if A[i] < min then                    c3      n − 1
            min ← A[i]                        c4      n − 1
    return min                                c5      1
```

Ragionamento sul calcolo della complessità L'assegnazione del minimo parziale viene eseguita solo una volta. Il ciclo viene eseguito n volte. Considerando *il caso pessimo*, ovvero un vettore ordinato in modo decrescente, il controllo $A[i] < min$ viene eseguito $n-1$ volte in quanto ogni elemento che incontreremo sarà minore del precedente. Infine l'istruzione di ritorno viene eseguita una volta sola.

Bisogna tenere ben a mente che:

- ogni istruzione richiede un tempo costante per essere eseguita e
- viene eseguita un certo numero di volte, dipendente da n e
- la costante è potenzialmente diversa da istruzione a istruzione.

Sommando tutte le costanti il costo totale risultante è:

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \quad \text{raccogliamo} \\ &= an + b \quad \text{semplifichiamo} \end{aligned}$$

Possiamo quindi notare che le costanti vanno a semplificarsi nei parametri a e b.

Proviamo a calcolare la complessità dell'algoritmo che ricerca un numero intero all'interno di un vettore ordinato.

Calcolo della complessità della ricerca di un numero intero in un vettore ordinato

```
binarySearch(ITEM[] A, ITEM v, int i, int j)
    if i > j then                                          Costo          # i > j   # i ⩽ j
        return 0                                           c1             1        1
    else                                                   c2             1        0
        int m ← ⌊(i+j)/2⌋                                  c3             0        1
        if A[m] == v then                                  c4             0        1
            return m                                       c5             0        0
        else if A[m] < v then                              c6             0        1
            return binarySearch(A, v, m + 1, j)            c7 + T(⌊(n−1)/2⌋)  0    0/1
        else
            return binarySearch(A, v, i, m − 1)            c7 + T(⌊n/2⌋)      0    1/0
```

Nota. Ci è permesso fare questo ragionamento poiché il vettore è ordinato in ordine decrescente.

Ragionamento sul calcolo della complessità Il vettore viene diviso due parti: la parte sinistra di dimensione $\lfloor \frac{n-1}{2} \rfloor$ e la parte destra di dimensione $\lfloor n/2 \rfloor$. Se n è pari allora il vettore viene diviso in due parti uguali, altrimenti il vettore "di destra" avrà un elemento in più. Si andrà cercare sulla metà sinistra o sulla metà destra a seconda che l'elemento cercato sia più grande o più piccolo rispettivamente. Anche in questo caso consideriamo il caso peggiore, ovvero il caso in cui l'elemento non sia presente. Non prendiamo in considerazione il caso fortunato, ossia quello in cui l'elemento che stiamo cercando sia l'elemento che guardiamo per primo. Nelle chiamate ricorsive dobbiamo considerare (nel costo complessivo) anche il costo delle sotto-chiamate ricorsive con dimensione dell'input pari alla dimesione del vettore passato.

Esercizio Cerca nel vettore ordinato l'elemento "0" tramite la procedura binarySearch, calcolando di volta in volta la dimensione del vettore n.

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

Calcolo del caso pessimo Assumiamo per semplicità che:

1. n sia una potenza di 2 ($n = 2^k$, $8 = 2^3$);
2. l'elemento cercato non sia precisato;
3. ad ogni passo scegliamo il vettore di destra (di dimensione $n/2$).

Si hanno due casi:

- il caso base $\boxed{i > j}$, dove $n = 0$ e la relazione di ricorrenza è pari a $T(n) = c_1 + c_2 = c$, dove c è una costante;
- il caso ricorsivo $\boxed{i \leqslant j}$, dove $n > 0$ e dobbiamo tener conto di tutte le costanti moltiplicative $T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7$, raccogliendo le costanti $T(n) = T(n/2) + d$, dove d è la costante che racchiude tutti i costi.

La relazione di ricorrenza che ne segue è la sequente:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n/2) + d & \text{se } n > 0 \end{cases}$$

Nota. Per calcolare la complessità di una funzione ricorsiva abbiamo bisogno di una funzione di ricorrenza anch'essa ricorsiva.

Le **equazioni di ricorrenza** così fatte $T(n) = d\log(n) + e$ sono dette **in** "**forma chiusa**" e rappresentano la complessità dell'algoritmo (non necessano quindi ulteriori sviluppi).

Risolviamo quindi l'equazione di ricorrenza *tramite espansione*:

$$\begin{aligned} T(n) &= T(n/2) + d && \big) \left(T\left(\tfrac{n}{2}\cdot\tfrac{1}{2}\right) + d\right) + d \\ &= T(n/4) + 2d && \big) \left(T\left(\tfrac{n}{4}\cdot\tfrac{1}{2}\right) + 2d\right) + d \\ &= T(n/8) + 3d \\ &\vdots && \big) \; n = 2^k \Rightarrow k = \log n \\ &= T(1) + kd && \big) \; T(0) = c \\ &= T(0) + (k+1)\,d \\ &= kd + (c+d) && \big) \; k = \log n \\ &= d\log n + e. \end{aligned}$$

2.4 Ordini di complessità

Finora abbiamo analizzato precisamente due algoritmi e abbiamo ottenuto due *funzioni di complessità*:

- Ricerca: $T(n) = d\log n + e$. Chiamiamo questa funzione **logaritmica**, utilizzando la notazione $\mathcal{O}(\log n)$;
- Minimo: $T(n) = a + b$. Chiamiamo questa funzione **lineare**, utilizzando la notazione $\mathcal{O}(n)$.

Abbiamo visto anche una terza funzione che deriva dall'implementazione banale (*naïf*) dell'algoritmo per la ricerca del minimo:

- Minimo: $T(n) = fn^2 + gn + h$. Chiamiamo questa funzione **quadratica**, utilizzando la notazione $\mathcal{O}(n^2)$.

Tabella 2.1: Classi di complessità

$f(n)$	$n = 10^1$	$n = 10^2$	$n = 10^3$	$n = 10^4$	Tipo
$\log n$	3	6	9	13	logaritmico
$\sqrt{n}$	3	10	31	100	sublineare
n	10	100	1000	10 000	lineare
$n\log n$	30	664	9965	132 877	loglineare
n^2	10^2	10^4	10^6	10^8	quadratico
n^3	10^3	10^6	10^9	10^{12}	cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	esponenziale

2.5 Funzioni di costo, notazione asintotica

Ora andremo a formalizzare le nozioni sui limiti superiori ed inferiori che abbiamo accennato in maniera informale nelle lezioni precedenti.

Definizione (funzione di costo)**.** Utilizziamo il termine "funzione di costo" per indicare una funzione $f: \mathbb{N} \to \mathbb{R}$ (dall'insieme dei numeri naturali ai reali).

Definizione (notazione $\mathcal{O}$)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\mathcal{O}(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cg(n), \forall n \geqslant m$$

Nota. Eventuali fattori moltiplicativi non ci interessano.

La notazione si legge $f(n)$ è "O grande" di $g(n)$ e si scrive $f(n) = \mathcal{O}(g(n))$. Questo è un abuso di notazione, dovemmo scrivere $f(n) \in \mathcal{O}(g(n))$, in quanto $\mathcal{O}$ è un insieme (più precisamente una famiglia di funzioni). Questa notazione è però diventata d'uso comune, poiché ci si può fare una specie di aritmetica sopra, infatti è la notazione che troverete nella letteratura e sta a significare che $g(n)$ è un limite asintotico superiore per $f(n)$, ossia che $f(n)$ cresce al più (al massimo) come $g(n)$.

Definizione (notazione Ω)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : f(n) \geqslant cg(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Omega grande" (nella letteratura big-O) di $g(n)$, si scrive $f(n) = \Omega(g(n))$ e sta a significare che $g(n)$ è un limite asintotico inferiore per $f(n)$, ossia che $f(n)$ cresce almeno quanto (non meno di) $g(n)$.

Definizione (notazione Θ)**.** Sia $g(n)$ una funzione di costo; indichiamo con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists c > 0, \exists m \geqslant 0 : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n), \forall n \geqslant m$$

La notazione si legge $f(n)$ è "Theta" di $g(n)$, si scrive $f(n) = \Theta(g(n))$ e sta a significare che $f(n)$ cresce *esattamente* come $g(n)$ al di là di fattori moltiplicativi. Nota che $f(n) = \Theta(g(n))$ avviene se e solo se $f(n) = \mathcal{O}(g(n))$ e $f(n) = \Theta(g(n))$.

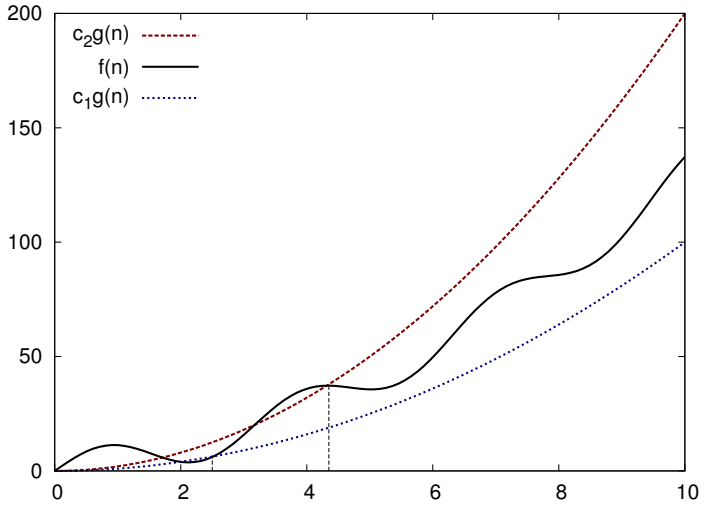


Figura 2.1: Notazione asintotica Θ.

2.6 Complessità di un algoritmo e di un problema

Definizione (complessità in tempo di un algoritmo)**.** La più grande quantità di tempo richiesta *per un input* di dimensione n.

- $\mathcal{O}(f(n))$: per tutti gli input, l'algoritmo costa al più $f(n)$;
- $\Omega(f(n))$: per tutti gli input, l'algoritmo costa almeno $f(n)$;
- $\Theta(f(n))$: l'algoritmo richiede $\Theta(f(n))$ per tutti gli input.

Definizione (complessità in tempo di un problema computazionale)**.** La complessità in tempo relativa *a tutte le possibili soluzioni.*

- $\mathcal{O}(f(n))$: complessità del miglior algoritmo che risolve il problema;
- $\Omega(f(n))$: dimostrare che nessun algoritmo può risolvere il problema in tempo inferiore a $\Omega(f(n))$;
- $\Theta(f(n))$: abbiamo trovato l'algoritmo ottimo.

2.7 Esercizi

Primo esercizio

Iniziamo con degli esercizi banali che ci permetteranno di introdurre delle tecniche che utilizzeremo con le ricorrenze. In particolare ci serviranno a renderci conto che non stiamo dimostrando equazioni, ma disequazioni.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 \\
&\leqslant 10n^3 + 2n^3 + 7 && \forall n \geqslant 1 \\
&\leqslant 10n^3 + 2n^3 + 7n^3 \\
&= 19n^3 && \textit{sommiamo i termini} \\
&\stackrel{?}{\leqslant} cn^3 && \textit{esiste una certa costante } c \textit{ per la quale } f(n) \leqslant cn^3 \ ? \\
19n^3 &\leqslant cn^3 && \textit{metto a confronto} \\
19\cancel{n^3} &\leqslant c\cancel{n^3} && \textit{semplifico}
\end{aligned}$$

che è vera per ogni $c \geqslant 19$ (abbiamo così trovato la costante moltiplicativa) e per ogni $n \geqslant 1$ (introdotta nei calcoli), quindi $m = 1$ (che deriva da $\forall n \geqslant m$).

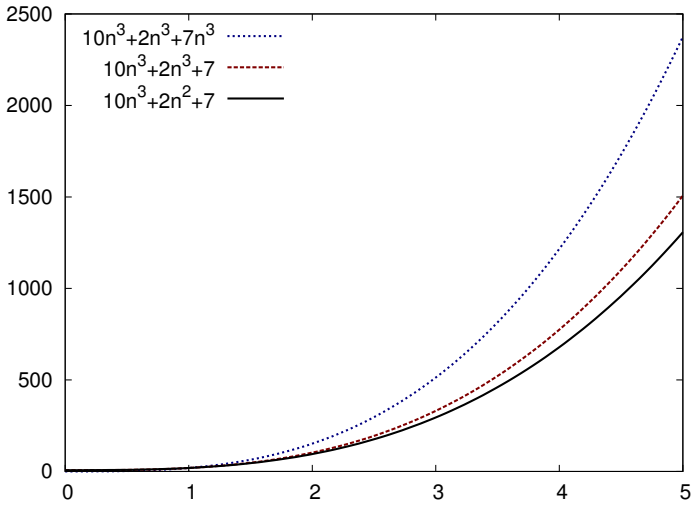


Figura 2.2: Risoluzione grafica dell'esercizio.

Nota. In generale noi considereremo solo valori di n positivi, in quanto le funzioni di costo sono definite sull'insieme dei numeri naturali, non ha alcun senso definire una funzione di costo su una dimensione dell'input negativa.

Soluzione alternativa all'esercizio precedente

Dato lo stesso esercizio posso esserci passaggi risolutivi diversi. Risolviamo quindi l'esercizio precedente in modo diverso.

$$\boxed{f(n) = 10n^3 + 2n^2 + 7 \stackrel{?}{=} \mathcal{O}(n^3)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^3, \forall n \geqslant m$

$$\begin{aligned}
f(n) &= 10n^3 + 2n^2 + 7 && \text{(}\forall n \geqslant 1\text{)}\\
&\leqslant 10n^3 + 2n^3 + 7 && \text{(}\forall n \geqslant \sqrt[3]{7}\text{)}\\
&\leqslant 10n^3 + 2n^3 + n^3 && \text{(sommiamo i termini)}\\
&= 13n^3 && \text{(esiste una certa costante } c\\
&\stackrel{?}{\leqslant} cn^3 && \text{per la quale } f(n) \leqslant cn^3 \text{ ?)}\\
13n^3 &\leqslant cn^3 && \text{(metto a confronto)}\\
13\cancel{n^3} &\leqslant c\cancel{n^3} && \text{(semplifico)}
\end{aligned}$$

che è vera per ogni $c \geqslant 13$ e per ogni $n \geqslant \sqrt[3]{7}$ (ad esempio con $n = 2$ abbiamo $n^3 = 2^3 = 8$ che soddisfa la nostra condizione), quindi usiamo $m = 2$ (abbiamo semplificato, sarebbe $m = \sqrt[3]{7}$, ma possiamo prendere un qualunque valore si trovi dopo n in modo totalmente arbitrario).

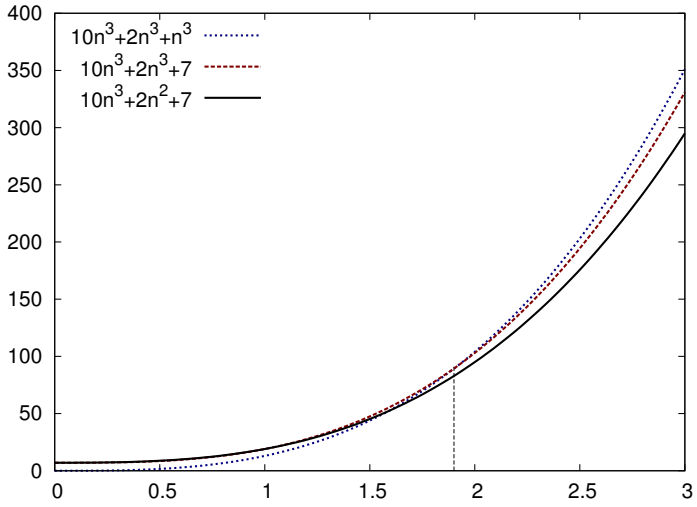


Figura 2.3: Risoluzione grafica dell'esercizio.

Secondo esercizio

$$\boxed{f(n) = 3n^2 + 7n \stackrel{?}{=} \Theta(n^2)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c_1 > 0, \exists m_1 \geqslant 0 : f(n) \geqslant c_1 n^2, \forall n \geqslant m_1$

$$\begin{aligned} f(n) &= 3n^2 + 7n && \text{(} \forall n \geqslant 0 \text{)} \\ &\geqslant 3n^2 && \text{(esiste una certa costante } c \\ &\stackrel{?}{\geqslant} c_1 n^2 && \text{per la quale } f(n) \leqslant c_1 n^2 \text{ ?)} \\ 3n^2 &\leqslant c_1 n^2 && \text{(metto a confronto)} \\ 3\cancel{n^2} &\leqslant c_1 \cancel{n^2} && \text{(semplifico)} \end{aligned}$$

che è vera per ogni $c_1 \leqslant 3$ e per ogni $n \geqslant 0$ (introdotta nei calcoli), quindi $m_1 = 0$.

Nota. Abbiamo dimostrato quindi che $f(n) = \Omega(n^2)$.

Limite superiore: Dobbiamo dimostrare che $\exists c_2 > 0, \exists m_2 \geqslant 0 : f(n) \leqslant c_2 n^2, \forall n \geqslant m_2$

$$\begin{aligned} f(n) &= 3n^2 + 2n^2 + 7n && \text{(} \forall n \geqslant 1 \text{)} \\ &\leqslant 3n^2 + 7n^2 && \text{(raccogliamo)} \\ &\leqslant 10n^2 && \text{(esiste una certa costante } c \\ &\stackrel{?}{\leqslant} c_2 n^2 && \text{per la quale } f(n) \leqslant c_2 n^2 \text{ ?)} \\ 10n^2 &\leqslant c_2 n^2 && \text{(metto a confronto)} \\ 10\cancel{n^2} &\leqslant c_2 \cancel{n^2} && \text{(semplifico)} \end{aligned}$$

che è vera per ogni $c_2 \geqslant 10$ e per ogni $n \geqslant 1$, quindi $m_2 = 1$.

Nota. Abbiamo dimostrato quindi che $f(n) = \mathcal{O}(n^2)$.

Notazione Θ: $\exists c_1 > 0, \exists c_2 > 0, \exists m \geqslant 0 : c_1 n^2 \leqslant f(n) \leqslant c_2 n^2, \forall n \geqslant m$.

Con questi paramentri:

- $c_1 = 3$;
- $c_1 = 10$;
- $m = max\{m_1, m_2\} = max\{0, 1\} = 1$, ossia un valore dopo il quale la nostra proprietà è provata.

Nota. Abbiamo dimostrato quindi che $f(n) = \Theta(n^2)$.

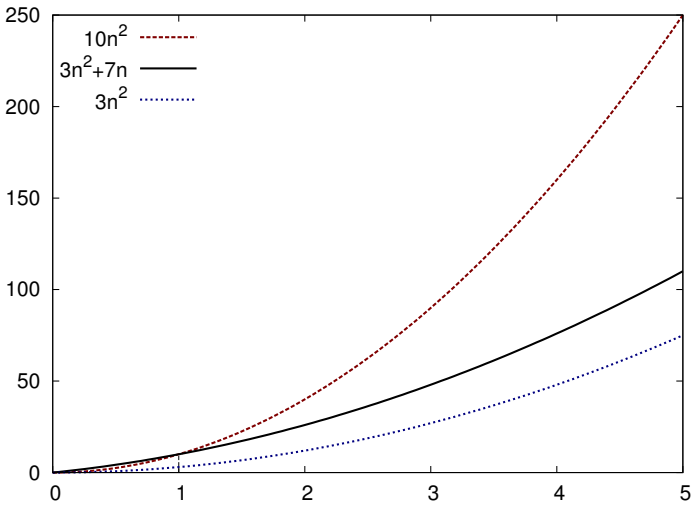


Figura 2.4: Risoluzione grafica dell'esercizio.

Errori comuni durante la risoluzione degli esercizi

Dimostrare un limite superiore inesistente

$$\boxed{f(n) = n^2 \stackrel{?}{=} \mathcal{O}(n)}$$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : n^2 \geqslant cn, \forall n \geqslant m$.

Otteniamo che $n^2 \leqslant cn \Leftrightarrow c \geqslant n$ (ad esempio con $n = 2$, $2^2 \leqslant c \cdot 2 \Leftrightarrow c \geqslant 2$), questo significa che c cresce con il crescere di n, non possiamo quindi scegliere una *costante* c che verifichi la proprietà.

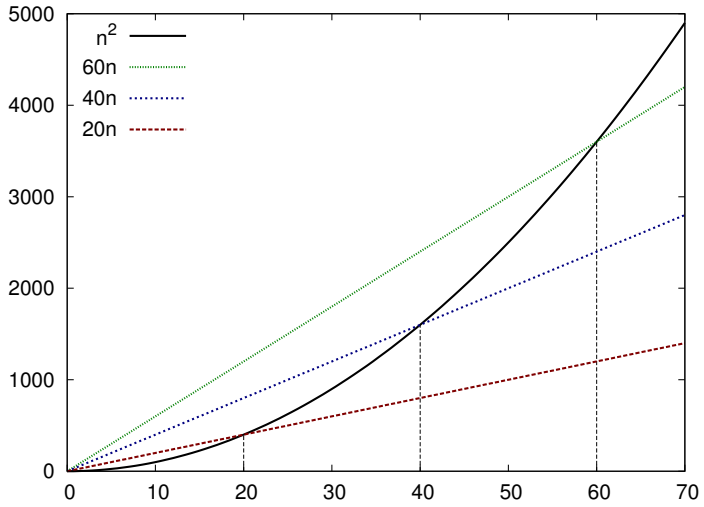


Figura 2.5: Per qualunque fattore c scelto (ossia la pendenza della retta) la curva quadratica crescerà sempre più velocemente da un certo punto in poi.

Dimostrare un limite inferiore inesistente

$$\boxed{f(n) = n^2 \stackrel{?}{=} \Omega(n^3)}$$

Limite inferiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0, n^2 \geqslant cn^3, \forall n \geqslant m$.

Otteniamo che $n^2 \geqslant cn^3 \Leftrightarrow c \leqslant \frac{1}{n}$ (ad esempio con $n = 2$, $2^2 \geqslant c \cdot 2 \Leftrightarrow c \leqslant \frac{1}{2}$), questo significa che c diminuisce al crescere di n, non possiamo quindi scegliere una costante c che verifichi la proprietà.

2.8 Complessità degli algoritmi e dei problemi a confronto

In questa sezione ragioneremo su alcuni algoritmi risolutivi che ci sono stati insegnati, in alcuni casi si può migliorare la complessità, in altri è impossibile fare di meglio.

Qual è il rapporto fra un problema computazionale e l'algoritmo?

2.8.1 Moltiplicare numeri complessi

La moltiplicazione fra numeri complessi avviene nel seguente modo: $(a+bi)(c+di) = [ac-db]+[ad+bc]i$. Abbiamo in input a, b, c, d e dobbiamo restituire in output $ac-bd$ e $ad+bc$.

Consideriamo un modello di calcolo dove la moltiplicazione costa 1 e le addizioni e sottrazioni costano 0,01.

1. Quanto costa l'algoritmo dettato dalla definizione?
2. Riesci a fare meglio di così?
3. Qual è il ruolo del modello di calcolo?

L'algoritmo banale dettato dalla definizione costa 4,02, in quanto bisogna fare 4 moltiplicazioni, 1 somma ed una sottrazione.

Soluzione di Gauss per la moltipicazione di numeri complessi

La seguente è la soluzione di Gauss al problema, datata 1805.

Input: a, b, c, d, Output: $A1 = ac-bd$, $A2 = ad+bc$

$$
\begin{aligned}
m_1 &= a \times c \\
m_2 &= b \times d \\
A_1 &= m_1 - m_2 \\
m_3 &= (a+b) \times (c+d) = ac+ad+bc+bd \\
A_2 &= m_3 - m_1 - m_2 = ad+bc
\end{aligned}
$$

calcolo i valori intermedi

evito una moltiplicazione

Il costo totale è 3,05.

Si può fare ancora meglio di così? Oppure è possibile dimostrare che non si può?

2.8.2 Sommare numeri binari

Nota. In questo caso usiamo il criterio del costo logaritmico.

L'algoritmo elementare della somma richiede di esaminare tutti gli n bit, il costo totale risulta cn, dove c è il costo per sommare due bit e generare il riporto.

Esiste un metodo più efficiente?

È dimostrabile per assurdo che *non è possibile fare di meglio* di una soluzione lineare, poiché non è possibile sommare due numeri binari senza esaminare tutti gli n bit.

Limiti alla complessità di un problema

Definizione (limite superiore, $\mathcal{O}(f(n))$)**.** Un problema ha complessità $\mathcal{O}(f(n))$ *se esiste almeno un algoritmo* che ha complessità $\mathcal{O}(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\mathcal{O}(n)$.

Definizione (limite inferiore, $\Omega(f(n))$)**.** Un problema ha complessità $\Omega(f(n))$ *se tutti i possibili algoritmi* che lo risolvono hanno complessità $\Omega(f(n))$.

Nota. Il problema della somma dei numeri binari ha complessità $\Omega(n)$.

2.8.3 Moltiplicare numeri binari

L'algoritmo elementare del prodotto richiede di moltiplicare ogni bit con ogni altro bit, per un costo totale di cn^2.

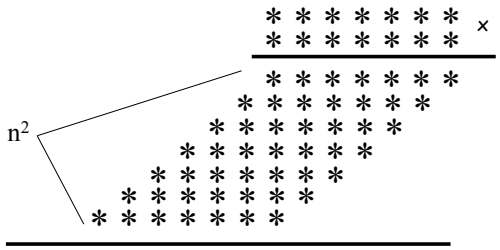


Figura 2.6: Moltiplicazione di due numeri binari.

Si potrebbe concludere che il problema della moltiplicazione è molto più costoso del problema dell'addizione: ne è conferma la nostra esperienza.

Nota. Per provare che il problema del prodotto è più costoso del problema della somma, dobbiamo provare che non esiste una soluzione in tempo lineare al problema del prodotto.

Abbiamo infatti erroneamente confrontato gli algoritmi, non i problemi! Sappiamo solo che l'algoritmo della somma che ci hanno insegnato è più efficiente di quello della moltiplicazione.

Nel 1960, Kolmogorov enunciò che la moltiplicazione avesse limite inferiore pari a $\Omega(n^2)$, una settimana dopo un suo studente Karatsuba riuscì a provare il contrario. Osserviamo la sua soluzione.

Approccio divide-et-impera

Karatsuba adottò un approccio divide-et-impera.

Definizione 2.8.1 (approccio divide-et-impera)**.** Si svolge in tre parti:

- **Divide**: dividi il problema in sottoproblemi di dimensione inferiore;
- **Impera**: risolvi i sottoproblemi in maniera ricorsiva;
- (**Combina**): unisci le soluzioni dei sottoproblemi in modo da ottenere la risposta del problema principale.

$$X = a \cdot 2^{n/2} + b$$
$$Y = c \cdot 2^{n/2} + d$$
$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + db$$

Algoritmo 2.8.1: Moltiplicazione di due numeri binari

```
boolean[] pdi(boolean[] X, boolean[] Y, int n)
    // X:      numero binario
    // Y:      numero binario
    // n:      numero di bit contenuti
    if n == 1 then
        return X[1] · Y[1] // eseguo la moltiplicazione di due bit
    else
        spezza X in a;b e Y in c;d
        return pdi(a, c, n/2) · 2^n + (pdi(a, d, n/2) + pdi(b, c, n/2)) · 2^(n/2) + pdi(b, d, n/2)
```

Complessità Moltiplicare per 2^t è equivalente ad eseguire uno *shift* di t posizioni, in tempo lineare, quindi l'equazione di ricorrenza risultante è

$$T(n) = \begin{cases} c_1 & n = 1 \\ 4T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Nota. Non sappiamo ancora trattare questo genere di problemi, quindi facciamo solo degli accenni.

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 4 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $4^2 = 16$ chiamate ricorsive su una dimesione $n/2^2$... al livello i-esimo vengono effettuate 4^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $4^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $4^{\log n} = n^{\log 4} = n^2$, le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^2)$.

Moltiplicazione di Karatsuba

È possibile ridurre ulteriormente la complessità.

$$\begin{aligned} A_1 &= a \times c \\ A_2 &= b \times d \\ m &= (a+b) \times (c+d) = ac + ad + bc + bd \\ A_3 &= m - A_1 - A_2 = ad + bc \end{aligned}$$

calcolo un valore intermedio

evito una moltiplicazione

Principio Effettuo un'unica moltiplicazione che mi permette di calcolare un valore intermedio che contiene la somma di tutte le combinazioni (m), e ricavo $ad + bc$ tramite due sottrazioni (nello stesso modo in cui lavorava Gauss) evitando così una moltiplicazione.

Algoritmo 2.8.2: Moltiplicazione di Karatsuba

```
boolean[] karatsuba(boolean[] X, boolean[] Y, int n)
    if n == 1 then
        return X[1] · Y[1] // rimane invariato
    else
        spezza X in a;b e Y in c;d
        boolean[] A1 = karatsuba(a, c, n/2)
        boolean[] A3 = karatsuba(b, d, n/2)
        boolean[] m = karatsuba(a + b, c + d, n/2) // potrebbe essere n/2 + 1
        boolean[] A2 = m − A1 − A3 // ottengo A2 tramite sottrazione
        return A1 · 2^n + A2 · 2^(n/2) + A3 // effettuo degli shift
```

Complessità L'equazione di ricorrenza risultante è:

$$T(n) = \begin{cases} c_1 & n = 1 \\ 3T(n/2) + c_2 \cdot n & n > 1 \end{cases}$$

Analisi della ricorsione Al primo passo la chiamata ricorsiva avviene su una dimesione n, al secondo passo vengono effettuate 3 chiamate ricorsive su una dimesione $n/2$, al terzo passo vengono effettuate $3^2 = 9$ chiamate ricorsive su una dimesione $n/3^2 \ldots$ al livello i-esimo vengono effettuate 3^i chiamate ricorsive su una dimesione $n/2^i$. Una volta arrivati al passo $\log_2 n$ vengono effettuate $3^{\log n}$ chiamate ricorsive su una dimesione pari al caso base $T(1)$, per la proprietà dei logaritmi $3^{\log n} = n^{\log 3} = n^{1.58\ldots}$. Le dimensioni delle chiamate ricorsive vengono ridotte ad una semplice costante. Possiamo quindi concludere che $T(n) = \mathcal{O}(n^{1.58\ldots})$.

Nota. L'algoritmo ingenuo (*naïf*) non è sempre il migliore a meno che non sia possibile dimostrare il contrario.

Negli anni sono stati proposti diversi algoritmi, che il limite inferiore al problema della moltiplicazione sia $\Omega(n \log n)$ è una congettura. Una congettura è un'affermazione o un giudizio fondato sull'intuito, ritenuto probabilmente vero, ma non ancora rigorosamente dimostrato, cioè dunque relegato solamente a rango di ipotesi.

Nella GNU Multiple Precision Arithmetic Library vengono utilizzati diversi algoritmi al crescere di n, il valore soglia per cui si predilige un algoritmo rispetto ad un altro dipende dal tipo di architettura.

2.9 Algoritmi di ordinamento

In questa lezione impareremo a capire quando è meglio utilizzare un algoritmo di ordinamento rispetto ad un altro.

In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input. Conoscere in anticipo tali caratteristiche permette di scegliere l'algoritmo migliore in quella determinata situazione.

Tipologia di analisi

Esistono tre tipi di analisi:

1. analisi del **caso pessimo**: è la tipologia più importante, il tempo di esecuzione nel caso peggiore è il limite superiore al tempo di esecuzione per qualsiasi input. Per alcuni algoritmi il caso peggiore si verifica molto spesso (ad esempio nella ricerca di dati non presenti nel database);
2. analisi del **caso medio**: è difficile da definire (cosa si intende per "medio" ?), dobbiamo avere una conoscenza pregressa sulle distribuzioni;
3. analisi del **caso ottimo**: può avere senso se si conoscono informazioni particolari sull'input.

Problema dell'ordinamento

Data una sequenza $A = a_1, a_2, \ldots, a_n$ di n valori in input, il problema dell'ordinamento consiste nel restituire in output una sequenza $B = b_1, b_2, \ldots, b_n$ che sia una permutazione di A, tale per cui $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$ (ovvero che ci sia un ordinamento *totale*).

Un approccio "demente" è quello di generare tutte le possibili permutazioni (complessità $n!$) fino a quando non se ne trova una ordinata.

2.9.1 Selection sort

Un approccio banale (*naïf*) è quello di cercare il minimo e metterlo nella posizione corretta, riducendo il problema agli $n-1$ valori rimanenti.

Algoritmo 2.9.1: selectionSort

```
// effettua l'ordinamento di un vettore
selectionSort(ITEM[] A, int n)
    from int i ← 1 until n − 1 do // l'ultimo elemento è ordinato
        int j ← min(A, i, n) // ricerca il nuovo minimo
        A[i] ↔ A[j] // lo metto nella posizione corretta

// cerca l'indice dell'elemento più piccolo
int min(ITEM[] A, int i, int j)
    int min ← i // posizione del minimo parziale
    from int j ← i + 1 until n do
        if A[j] < A[min] then // ho trovato un nuovo minimo
            min ← j // nuovo minimo parziale
    return min // restituisco l'indice dell'elemento più piccolo
```

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$i=1$	7	4	2	1	8	3	5
$i=2$	1	4	2	7	8	3	5
$i=3$	1	2	4	7	8	3	5
$i=4$	1	2	3	7	8	4	5
$i=5$	1	2	3	4	8	7	5
$i=6$	1	2	3	4	5	7	8
$i=7$	1	2	3	4	5	7	8

Figura 2.7: Funzinamento dell'algoritmo selectionSort.

Suggerimento. Provalo su carta! Un algoritmo dev'essere provato per essere capito!

Analisi della complessità Il ciclo effettua n chiamate della funzione min (una per ciascuna iterazione). Ad ogni iterazione il vettore su cui viene calcolato il minimo risulta più piccolo di un elemento.

$$
\begin{aligned}
&\textstyle\sum_{i=1}^{n-1}(n-1) \\
&= \textstyle\sum_{i=1}^{n-1} i && \text{(} 10+9+\cdots+1 \Leftrightarrow 1+2+\cdots+10 \text{)} \\
&= \tfrac{n(n-1)}{2} \\
&= n^2 - \tfrac{n}{2} && \text{(} \textit{svolgo i calcoli} \text{)} \\
&= \Theta(n^2)
\end{aligned}
$$

Posso dire che è $\Theta(n^2)$, e non solo che $\Omega(n^2)$, perché indipendentemente dall'ordine iniziale ci metterà sempre lo stesso tempo. In altre parole, il caso migliore, peggiore e medio coincidono.

2.1.2 Insertion sort

Un algoritmo che si basa sul principio di ordinamento di una "mano" di carte da gioco è il seguente:

Algoritmo 2.1.2: insertionSort

```
// effettua l'ordinamento di un vettore
insertionSort(ITEM[] A, int n)
    from int i = 2 until n do // il 1° elemento verrà ordinato in seguito
        ITEM temp ← A[i] // elemento da ordinare
        int j ← i
        while j > 1 and A[j − 1] > temp do
            A[j] ← A[j − 1] // copio l'elemento
            j++ // mi sposto
        A[j] ← temp
```

Questo è un algoritmo molto efficiente per ordinare piccoli insiemi di elementi.

Analisi della complessità Il *costo di esecuzione* di questo algoritmo non *dipende* solo dalla dimensione del vettore, ma anche *dalla distribuzione dei dati in ingresso*. Nel caso in cui il vettore sia già *ordinato* il costo è $\mathcal{O}(n)$, in quanto non si entra mai nel secondo ciclo dato che la condizione risulta falsa. Nel caso in cui il vettore sia *ordinato in ordine inverso* è $\Omega(n^2)$. In media (informalmente) possiamo assumere che metà dei valori sia ordinata rispetto la loro disposizione finale e quindi metà di loro dovrà fare n passi per arrivare alla destinazione per una complessità di $n \cdot n/2 = \mathcal{O}(n^2)$.

Infine quando sappiamo che i valori sono quasi ordinati o che n è molto piccolo (nell'ordine di 16 o 32 elementi) allora questo algoritmo risulta efficiente.

2.1.3 Merge sort

MergeSort è basato sulla tecnica divide-et-impera vista in precendenza. Ma come la utilizza?

Definizione 2.1.1 (approccio divide-et-impera di MergeSort)**.** Si svolge in tre parti:

- **divide**: spezza il vettore di n elementi in 2 sottovettori di $\frac{n}{2}$ elementi;
- **impera**: chiama mergeSort ricorsivamente sui due sottovettori (ottenendo due metà ordinate);
- (**combina**): unisce (da questo deriva *merge*) le due sequenze ordinate.

L'idea alla base di questo algoritmo sfrutta il fatto che è possibile *unire due sottovettori ordinati* in un vettore ordinato *in tempo lineare*.

Algoritmo 2.1.3: mergeSort

```
// ordina i sottovettori
mergeSort(ITEM[] A, int primo, int ultimo)
    if primo < ultimo then // devono esistere almeno due elementi
        int mezzo ← ⌊(primo+ultimo)/2⌋
        mergeSort(A, primo, mezzo)
        mergeSort(A, mezzo + 1, ultimo)
        merge(A, primo, ultimo, mezzo) // unisce le soluzioni

// effettua l'ordinamento dei sotto-vettori
merge(ITEM A, int primo, int ultimo, int mezzo)
    int i, j, k, h

    // inizializzo i puntatori
    i ← primo
    j ← mezzo
    k ← primo // k:  indica la prossima posizione di scrittura

    while i ⩽ mezzo and j ⩽ ultimo do
        // B è il vettore di appoggio in cui memorizzo la porzione di vettore già ordinata
        if A[i] ⩽ A[j] then
            // l'elemento è gia ordinato
            B[k] ← A[i]
            i++
        else
            B[k] ← A[j]
            j++

        // in entrambi i casi ho inserito un valore
        k++

    // se uno dei due vettori finisce ricopio la parte ordinata alla fine del vettore d'appoggio
    j ← ultimo
    from h ← mezzo until i do
        A[j] ← A[h]
        j++

    // ricopio il vettore d'appoggio del vettore originale
    from j ← primo until k − 1 do
        A[j] ← B[j]
```

Analisi della complessità Assumiamo (per semplicità) che $n = 2^k$ (ovvero che l'altezza dell'albero di suddivisioni sia esattamente $k = \log_2 n$) e che tutti i sottovettori abbiano dimensioni che sono potenze esatte di 2. L'equazione di ricorrenza risultante è la seguente:

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

Qual è il costo computazionale di mergeSort?

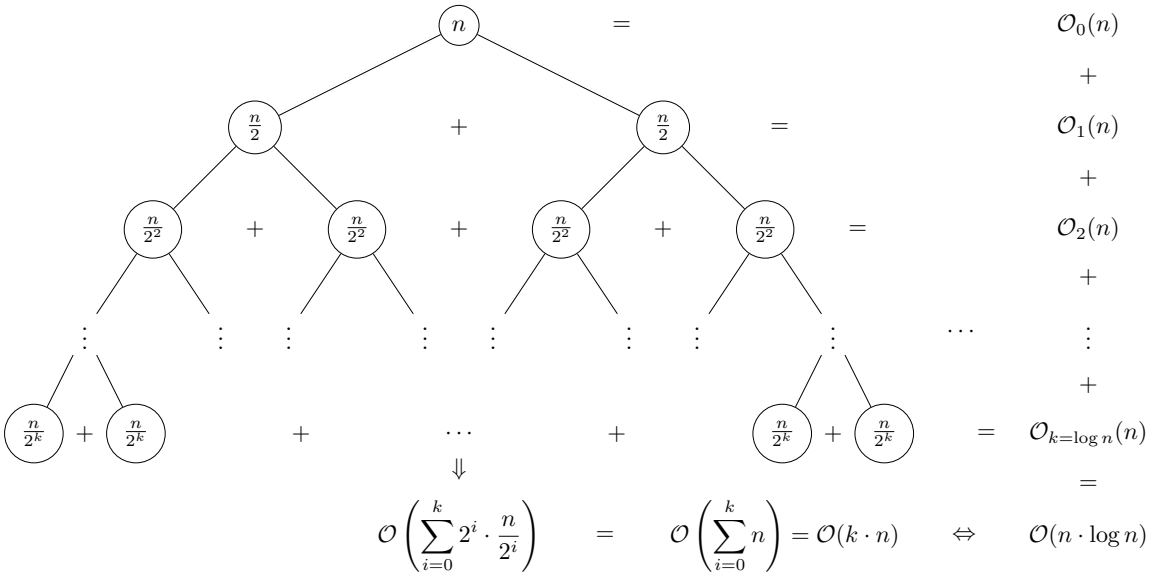


Figura 2.8: Analisi per livelli del costo di mergeSort

L'analisi per livelli è la seguente:

$$\begin{aligned} & \mathcal{O}\left(\textstyle\sum_{i=0}^{k} 2^i \frac{n}{2^i}\right) && \text{semplifico} \\ &= \mathcal{O}\left(\textstyle\sum_{i=0}^{k} n\right) && \text{equivalente} \\ &= \mathcal{O}((k+1) \cdot n) && k+1 \text{ elementi, semplico perché è una costante} \\ &= \mathcal{O}(k \cdot n) && k = \log n \\ &= \mathcal{O}(n \log n) \end{aligned}$$

$\mathcal{O}(n \log n)$ è asintoticamente migliore di $\mathcal{O}(n^2)$. Questo algoritmo è preferibile — per grandi dimensioni di n — al selectionSort e all'insertionSort.

Capitolo 3

Analisi delle funzioni di costo

Abbiamo concluso la prima parte delle lezioni che ci introduceva alle equazioni di ricorrenza: ora andremo a capire i fondamenti matematici che ci permetteranno di analizzarne una famiglia più grande.

3.1 Proprietà della notazione asintotica

3.1.1 Regola generale

Teorema 1 (regola generale)**.** $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots a_1 n + a_0, a_k > 0 \Rightarrow f(n) = \Theta(n^k)$

Dimostrazione. **Limite superiore**: $\exists c > 0, \exists m \geqslant 0 : f(n) \leqslant cn^k, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\
&\leqslant a_k n^k + |a_{k-1}| n^{k-1} + \dots + |a_1| n + |a_0| \\
&\leqslant a_k n^k + |a_{k-1}| n^k + \dots + |a_1| n^k + |a_0| n^k \\
&= (a_k + |a_{k-1}| + \dots + |a_1| + |a_0|)\, n^k \\
&\overset{?}{\leqslant} cn^k
\end{aligned}
$$

$a_k > 0$ per def., rendo gli altri positivi
$\forall n \geqslant 1$, elevo tutte le potenze a k
raccolgo n^k
esiste una costante c che rende la disequazione vera?

che è vera per $c \geqslant (|a_k| + |a_{k-1}| + \dots + |a_1| + |a_0|)$ (il coefficiente) e per $m = 1$. □

Dimostrazione. **Limite inferiore**: $\exists d > 0, \exists m \geqslant 0 : f(n) \geqslant dn^k, \forall n \geqslant m$

$$
\begin{aligned}
f(n) &= a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 \\
&\geqslant a_k n^k - |a_{k-1}| n^{k-1} - \dots - |a_1| n - |a_0| \\
&\geqslant a_k n^k - |a_{k-1}| n^{k-1} - \dots - |a_1| n^{k-1} - |a_0| n^{k-1} \\
&\overset{?}{\geqslant} dn^k
\end{aligned}
$$

normalizzo aggiungendo un segno negativo
$\forall n \geqslant 1$, elevo alla $k-1$

L'ultima equazione è vera se:

$$
d \leqslant a_k - \frac{|a_{k-1}|}{n} - \frac{|a_{k-2}|}{n} - \dots - \frac{|a_1|}{n} - \frac{|a_0|}{n} > 0 \iff n > \frac{|a_{k-1}| + \dots + |a_0|}{a_k} = m
$$

□

Abbiamo dimostrato sia il limite superiore che quello inferiore, possiamo affermare che è un Θ.

Ad esempio $17n^3 - 47n^2 + 123n + 17 = \Theta(n^3)$. Oppure $2n^3 + 7 = \Theta(n^2)$.

3.1.2 Funzioni di costo particolari

La complessità di $f(n) = 5$ è pari a $\Theta(1)$, questa classe di complessità rappresenta quegli algoritmi che sono così ben congegnati che indipendentemente dalla dimensione dell'input impiegano un tempo costante a risolvere il problema. Ad esempio richiedere il minimo in un vettore ordinato.

Dimostriamolo.

$$f(n) = 5 \geqslant c_1 n^0 \Rightarrow c_1 \leqslant 5$$
$$f(n) = 5 \leqslant c_2 n^0 \Rightarrow c_2 \geqslant 5$$
$$f(n) = \Theta(n^0) = \Theta(1)$$

La complessità di $f(n) = 5 + \sin(n)$ è pari a $\Theta(1)$, in quanto $\sin(n)$ oscilla fra 1 e -1.

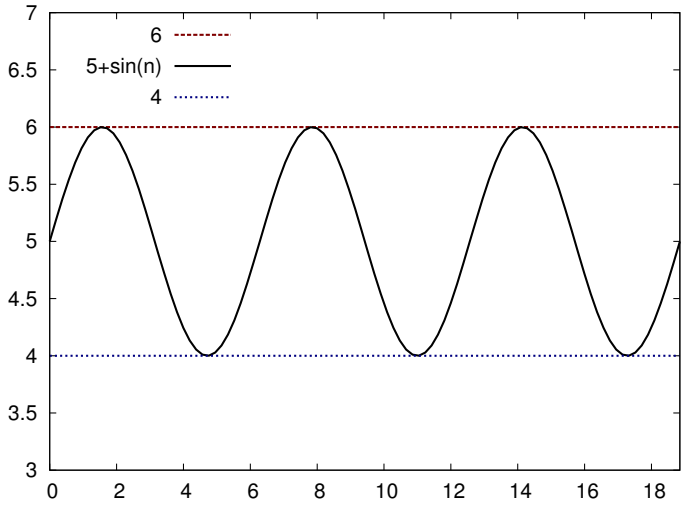


Figura 3.1: $\sin(n)$ oscilla fra 1 e -1.

3.1.3 Proprietà delle notazioni

Teorema 2 (dualità)**.** *Se $f(n)$ è limitata superiormente da $g(n)$, allora $g(n)$ è limitata inferiormente da $f(n)$.*

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

La dimostrazione avviene tramite passaggi algebrici.

Dimostrazione.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) &\Leftrightarrow f(n) \leqslant \boldsymbol{c}\ g(n), \forall n \geqslant m \\ &\Leftrightarrow g(n) > \tfrac{\mathbf{1}}{\boldsymbol{c}}\ f(n), \forall n \geqslant m \\ &\Leftrightarrow g(n) > \boldsymbol{c'}\ f(n), \forall n \geqslant m, c' = \tfrac{1}{c} \\ &\Leftrightarrow g(n) = \Omega(f(n)) \end{aligned}$$

ribalto la disequazione, $c > 0$
rinomino $\frac{1}{c}$ a c'
passo alla notazione dei quantificatori

□

Teorema 3 (eliminazione delle costanti)**.**

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow af(n) = \mathcal{O}(g(n)), \forall a > 0$$
$$f(n) = \Omega(g(n)) \Leftrightarrow af(n) = \Omega(g(n)), \forall a > 0$$

La dimostrazione avviene sempre tramite semplici passaggi algebrici.

Dimostrazione.

$$\begin{aligned} f(n) = \mathcal{O}(g(n)) &\Leftrightarrow f(n) \leqslant c\,g(n), \forall n \geqslant m && \text{Introduco una costante } a \\ &\Leftrightarrow f(n) \leqslant ac\,g(n), \forall n \geqslant m, \forall a \geqslant 0 && \text{raccolgo } ac \text{ sotto un'unica costante } c' \\ &\Leftrightarrow f(n) \leqslant c'g(n), \forall n \geqslant m, c' = ac > 0 && \text{per definizione} \\ &\Leftrightarrow af(n) = \mathcal{O}(g(n)) \end{aligned}$$

□

Ad esempio $2\log n = \Theta(\log n)$, ignoriamo quindi le costanti numeriche.

Teorema 4 (sommatoria, sequenza di algoritmi)**.**

$$\begin{cases} f_1(n) = \mathcal{O}(g_1(n)) \\ f_2(n) = \mathcal{O}(g_2(n)) \end{cases} \Rightarrow f_1(n) + f_2(n) = \mathcal{O}(\max(g_1(n), g_2(n)))$$

$$\begin{cases} f_1(n) = \Omega(g_1(n)) \\ f_2(n) = \Omega(g_2(n)) \end{cases} \Rightarrow f_1(n) + f_2(n) = \Omega(\max(g_1(n), g_2(n)))$$

Dimostrazione.

$$\begin{aligned} f_1(n) = \mathcal{O}(g_1(n)) \wedge f_2 n = \mathcal{O}(g_2(n)) \Rightarrow & && \text{per definizione} \\ f_1(n) \leqslant c_1 g_1(n) \wedge f_2(n) \leqslant c_2 g_2(n) \Rightarrow & && \\ f_1(n) + f_2(n) \leqslant c_1 g_1(n) + c_2 g_2(n) \Rightarrow & && \text{raccolgo} \\ f_1(n) + f_2(n) \leqslant \max\{c_1, c_2\}(2 \cdot \max(g_1(n), g_2(n)) \Rightarrow & && \text{per definizione} \\ f_1(n) + f_2(n) = \mathcal{O}(g_1(n) + g_2(n)) & \end{aligned}$$

□

Ad esempio se ripeto due volte un algoritmo lineare, l'algoritmo risultante sarà comunque lineare. Mentre se ho un algoritmo lineare ed un algoritmo quadratico, la complessità risultante è una combinazione delle due, il limite superiore della complessità totale sarà quindi quadratica.

Teorema 5 (cicli annidati)**.** *Se $f_1(n)$ viene ripetuto $f_2(n)$ volte, allora $f_1(n) \cdot f_2(n)$ è limitato superiormente dal prodotto delle complessità e limitato inferiormente dal prodotto delle complessità.*

$$\begin{cases} f_1(n) = \mathcal{O}(g_1(n)) \\ f_2(n) = \mathcal{O}(g_2(n)) \end{cases} \Rightarrow f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$$

$$\begin{cases} f_1(n) = \Omega(g_1(n)) \\ f_2(n) = \Omega(g_2(n)) \end{cases} \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

La dimostrazione è molto semplice.

Dimostrazione.

$$\begin{aligned} f_1(n) = \mathcal{O}(g_1(n)) \wedge f_2 n = \mathcal{O}(g_2(n)) \Rightarrow & && \text{per definizione} \\ f_1(n) \leqslant c_1 g_1(n) \wedge f_2(n) \leqslant c_2 g_2(n) \Rightarrow & && \text{raccolgo, } c_1 c_2 > 0 \\ f_1(n) \cdot f_2(n) \leqslant c_1 c_2 g_1(n) g_2(n) & \end{aligned}$$

□

Ad esempio, se ripeto n volte un algoritmo di costo $\log n$, l'algoritmo complessivo avrà un costo di $n \log n$.

Teorema 6 (simmetria)**.** *Se $f(n)$ è limitata superiormente ed inferiormente da $g(n)$, allora anche $g(n)$ è limitata inferiormente e superiormente da $f(n)$.*

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Vuol dire semplicemente che $2n^2 + 7 = \Theta(n^2) \Leftrightarrow n^2 = \Theta(2n^2 + 7)$.

La dimostrazione avviene tramite la proprietà di dualità:

Dimostrazione.

$$\begin{array}{lcl} f(n) = \Theta(g(n)) & \Rightarrow & f(n) = \mathcal{O}(g(n)) \quad \Rightarrow \quad g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) & \Rightarrow & f(n) = \Omega(g(n)) \quad \Rightarrow \quad g(n) = \mathcal{O}(f(n)) \end{array}$$

□

Teorema 7 (transitività)**.** *Se $f(n)$ è limitata superiormente da $g(n)$, e $g(n)$ è limitata superiormente da $h(n)$, allora $f(n)$ è limitata superiormente da $h(n)$.*

$$\begin{cases} f(n) = \mathcal{O}(g(n)) \\ g(n) = \mathcal{O}(h(n)) \end{cases} \Rightarrow f(n) = \mathcal{O}(h(n))$$

La dimostrazione è banale:

Dimostrazione.

$$\begin{array}{rll} f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n)) \Rightarrow & \text{applico la definizione} \\ f(n) \leqslant c_1 g(n) \wedge g(n) \leqslant c_2 h(n) \Rightarrow & \text{sostituisco } g(n) \text{ con } c_2 h(n) \\ f(n) \leqslant c_1 c_2 h(n) \Rightarrow & c_1 c_2 > 0, \text{ elimino la costante} \\ f(n) = \mathcal{O}(h(n)) \end{array}$$

□

Altre funzioni di costo

Vogliamo provare che $\boxed{\log n = \mathcal{O}(n)}$

Dimostriamo per induzione che $\exists c > 0, \exists m \geqslant 0 : \log n \leqslant cn, \forall n \geqslant m$.

- **caso base** ($n = 1$):

$$\begin{array}{rl} \log n \leqslant cn & \text{n = 1} \\ \log 1 \leqslant c \cdot 1 & \text{semplifico} \\ 0 \leqslant c \end{array}$$

- **ipotesi induttiva**: $\log k \leqslant ck, \forall k \leqslant n$
- **passo induttivo** dimostriamo la proprietà per $n + 1$:

$$\begin{array}{rll} \log(n+1) \leqslant \log(n+n) = \log 2n \quad \forall n \geqslant 1 & \log ab = \log a + \log b \\ = \log 2 + \log n & \log_2 2 = 1 \\ = 1 + \log n & \text{per ipotesi induttiva} \\ \leqslant 1 + cn & \text{obiettivo} \\ \overset{?}{\leqslant} c(n+1) & \text{metto a confronto} \\ 1 + cn \leqslant c(n+1) & \text{moltiplico} \\ 1 + cn \leqslant cn + c & \text{semplifico} \\ 1 \leqslant c \end{array}$$

Classificazione delle funzioni

È possibile trarre un ordinamento dalle principali espressioni, estendendo le relazioni che abbiamo dimostrato fino ad ora. Per ogni $r < s, h < k, a < b$:

$$\mathcal{O}(1) \subset \mathcal{O}(\log^r n) \subset \mathcal{O}(\log^s n) \subset \mathcal{O}(n^h) \subset \mathcal{O}(n^h \log^r n) \subset \mathcal{O}(n^h \log^s n) \subset \mathcal{O}(n^k) \subset \mathcal{O}(a^n) \subset \mathcal{O}(b^n)$$

Ricordati che $\log^r(n) = (\log n)^r$. Non è detto che gli esponenti siano interi. $\sqrt[1000]{n}$ cresce più velocemente di $\log^2 n$. n cresce meno velocemente di $n \log n$, il quale a sua volta cresce meno velocemente di $n \log^2 n$.

3.2 Analisi per livelli

Nell'analisi per livelli (o dell'albero di ricorsione) andiamo sostanzialmente a "srotolare" la ricorrenza in un albero, i cui nodi rappresentano i costi dei vari livelli della ricorsione.

3.2.1 Esempi di analisi per livelli

Analisi per livelli dell'algoritmo della ricerca binaria

Equazione di ricorrenza della ricerca binaria:

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

Assumiamo per semplicità che $n = 2^k$, ovvero che $k = \log n$. È possibile quindi risolvere questa ricorrenza nel seguente modo:

$$\begin{aligned} T(n) &= b + T\left(\tfrac{n}{2}\right) && \text{espando } T\left(\tfrac{n}{2}\cdot\tfrac{1}{2}\right)+b \\ &= b + b + T\left(\tfrac{n}{4}\right) && \text{espando } T\left(\tfrac{n}{4}\cdot\tfrac{1}{2}\right)+b \\ &= b + b + b + T\left(\tfrac{n}{8}\right) \\ &\quad\vdots && \text{svolgo } \log n \text{ operazioni} \\ &= \underbrace{b + b + \ldots + b}_{\log n} + T(1) && \text{semplifico} \\ &= b \log n + T(1) \\ &= \Theta(\log n) && \text{eliminazione delle costanti} \end{aligned}$$

Analisi per livelli del primo tentativo della moltiplicazione di Karatsuba

Equazione di ricorrenza:

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

È possibile risolvere questa ricorrenza nel modo seguente:

$$\begin{aligned} T(n) &= n + 4T(n/2) \\ &= n + 4\left(4T\left(\frac{n}{2}\cdot\frac{1}{2}\right) + \frac{n}{2}\right) && 4T\left(\tfrac{n}{2}\cdot\tfrac{1}{2}\right)+\tfrac{n}{2} \\ &= n + 4n/2 + 16T(n/4) && \text{semplifico} \\ &= n + 2n + 16\left(4T\left(\frac{n}{4}\cdot\frac{1}{2}\right) + \frac{n}{4}\right) && 4T\left(\tfrac{n}{4}\cdot\tfrac{1}{2}\right)+\tfrac{n}{4},\ 4n/2 = 2n \\ &= n + 2n + 16n/4 + 64T(n/8) && \text{semplifico} \\ &\quad\vdots && \text{svolgo } \log n \text{ operazioni} \\ &= \underbrace{n + 2n + 4n + 8n + \cdots + 2^{\log n - 1} n}_{sommatoria} + 4^{\log n} T(1) \\ &= n \sum_{j=0}^{\log n - 1} 2^j + 4^{\log n} && \text{raccolgo} \end{aligned}$$

Ciò che abbiamo ottenuto è una forma chiusa, non più un'equazione di ricorrenza: non è ancora nella sua forma definitiva, dobbiamo ancora trattarla.

$$
\begin{aligned}
T(n) &= n \sum_{j=0}^{\log n - 1} 2^j + 4^{\log n} && \text{Applico } \forall x \neq 1 : \textstyle\sum_{j=0}^{k} x^j = \frac{x^{k+1}-1}{x-1}, \text{ dove } k = \log n - 1 \\
&= n \cdot \frac{2^{\log n} - 1}{2 - 1} + 4^{\log n} && 2^{\log n} = n^{\log_2 2} = n^1 \\
&= n(n-1) + 4^{\log n} && \text{moltiplico: } n(n-1) = n^2 - n \\
&= n^2 - n + n^2 && \text{semplifico: } 4^{\log n} = n^{\log 4} = n^2 \\
&= 2n^2 - n && \text{raccolgo } n^2 \\
&= \Theta(n^2) && \text{regola generale}
\end{aligned}
$$

Possiamo concludere che $T(n) = n + 4T(n/2) = \Theta(n^2)$.

Modifichiamo l'esempio precendente

Esaminiamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Per semplicità consideriamo $n = 2^\ell$.

livello	dim.	costo chiam.	no. chiamate	costo livello
0	n	n^3	1	n^3
1	$n/2$	$(n/2)^3$	4	$4(n/2)^3$
2	$n/4$	$(n/4)^3$	16	$16(n/4)^3$
3	$n/8$	$(n/8)^3$	64	$64(n/8)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$n/2^i$	$(n/2^i)^3$	4^i	$4^i(n/2^i)^3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\ell - 1$	$n/2^{\ell-1}$	$(n/2^{\ell-1})^3$	$4^{\ell-1}$	$4^{\ell-1}(n/2^{\ell-1})^3$
$\ell = \log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Sommando il costo di tutti i $\ell - 1$ livelli ed il livello ℓ-esimo otteniamo:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log n - 1} 4^i \cdot \frac{n^3}{2^{3i}} + 4^{\log n} && \text{semplifico: } 4^i = 2^{2i}, \text{ porto fuori: } n^3 \\
&= n^3 \sum_{i=0}^{\log n - 1} \frac{2^{2i}}{2^{3i}} + 4^{\log n} && \text{semplifico: } \tfrac{2^{2i}}{2^{3i}} = \left(\tfrac{1}{2}\right)^i \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i + 4^{\log n} && \text{cambio di base, } 4^{\log n} = n^{\log 4} = n^2 \\
&= n^3 \sum_{i=0}^{\log n - 1} \left(\frac{1}{2}\right)^i + n^2 && \text{estensione della sommatoria ad } \infty \\
&\leqslant n^3 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + n^2 && \text{Serie geometrica infinita decrescente: } \forall x, |x| < 1 : \textstyle\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \text{ dove } x = \frac{1}{2} \\
&= n^3 + \frac{1}{1 - \frac{1}{2}} + n^2 && \text{semplifico: } \tfrac{1}{1-\frac{1}{2}} = 2 \\
&= 2n^3 + n^2
\end{aligned}$$

Abbiamo dimostrato che $T(n) \leqslant 2n^3 + n^2$, possiamo quindi affermare che $T(n) = \mathcal{O}(n^3)$, ma non possiamo affermare che $T(n) = \Theta(n^3)$ poiché abbiamo dimostrato solo un limite superiore.

Suggerimento. Tutte le volte che notiamo in un'equazione di riccorrenza una parte polinomiale, come ad esempio n^3, possiamo dire con certezza che l'equazione è $\Omega(n^3)$.

Ora possiamo affermare che $T(n) = \Theta(n^3)$.

Modifichiamo l'esempio precendente ulteriormente

Esaminiamo la seguente equazione di ricorrenza:

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

Cambia il costo della chiamata, che passa dall'essere n^3 all'essere n^2. Di conseguenza cambia anche il costo del livello.

Livello	Dim.	Costo chiam.	no. chiamate	Costo livello
0	n	n^2	1	n^2
1	$n/2$	$(n/2)^2$	4	$4(n/2)^2$
2	$n/4$	$(n/4)^2$	16	$16(n/4)^2$
3	$n/8$	$(n/8)^2$	64	$64(n/8)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	$n/2^i$	$(n/2^i)^2$	4^i	$4^i(n/2^i)^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\ell - 1$	$n/2^{\ell-1}$	$(n/2^{\ell-1})^2$	$4^{\ell-1}$	$4^{\ell-1}(n/2^{\ell-1})^2$
$\ell = \log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Sommando il costo di tutti i $\ell - 1$ livelli ed il livello ℓ-esimo otteniamo:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n - 1} \frac{n^2}{2^{2i}} \cdot 4i + 4^{\log_2 n} && \text{semplifico: } 4^i = 2^{2i},\ 4^{\log n} = n^{\log_2 4} = n^2 \\ &= n^2 \sum_{i=0}^{\log n - 1} \frac{2^{2i}}{2^{2i}} + n^2 && \text{porto fuori: } n^2 \\ &= n^2 \sum_{i=0}^{\log n - 1} 1 + n^2 && \text{semplifico: } \tfrac{2^{2i}}{2^{2i}} = 1 \\ &= n^2 \log n + n^2 && \text{svolgo la sommatoria: } \textstyle\sum_{i=0}^{\log n - 1} 1 = \log n \\ &= \Theta(n^2 log n) && \text{regola generale} \end{aligned}$$

Conclusioni

Per riassumere, se consideriamo i termini non ricorrenti, n ha prodotto n^2, n^2 ha prodotto $n^2 \log n$ ed n^3 ha prodotto n^3. Quando avremo a disposizione lo strumento "master theorem" riusciremo semplicemente guardando l'equazione di ricorrenza a capire quale complessità essa produce.

3.3 Metodo di sostituzione

Vediamo ora un ulteriore meccanismo per risolvere le equazioni di ricorrenza in quanto il metodo precendente in alcuni casi può non esserci di aiuto. Il metodo di sostituzione è un metodo in cui si cerca di "indovinare" (*guess*) una soluzione in base alla propria esperienza ed in seguito si dimostra che questa intuizione è corretta tramite dimostrazione per induzione.

Primo esercizio

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

Risolvendolo tramite il metodo precendente otteniamo:

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log n} \left(\frac{1}{2}\right)^i && \text{estensione della sommatoria ad } \infty \\ &\leqslant n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i && \text{Serie geometrica decrescente infinita: } \forall x, |x| < 1 : \textstyle\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \text{ dove } x = \frac{1}{2} \\ &\leqslant n \frac{1}{1-\frac{1}{2}} && \frac{1}{1-\frac{1}{2}} = 2 \\ &= 2n \end{aligned}$$

Sapendo già il risultato proviamo – per tentativi – a dimostrare che $\boxed{T(n) = \mathcal{O}(n)}$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn, \forall n \geqslant m$

- **caso base** dimostriamo $T(1)$:
$$T(1) = 1 \overset{?}{\leqslant} c \cdot 1 \iff c \geqslant 1$$
- **ipotesi induttiva** $\forall k < n : T(n) \leqslant ck$, ossia assumiamo che per tutti i valori più piccoli di n la dimostrazione sia già stata fatta.
- **ipotesi induttiva** dimostriamo la disequazione per $T(n)$:
$$\begin{aligned} T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{2} \rfloor \\ &\leqslant c\lfloor \tfrac{n}{2} \rfloor + n && \text{semplifico l'intero inferiore} \\ &\leqslant c\tfrac{n}{2} + n && \text{raccolgo } n \\ &= (\tfrac{c}{2} + 1)n && \text{obiettivo} \\ &= (\tfrac{c}{2} + 1)n \overset{?}{\leqslant} cn \\ &\Rightarrow \tfrac{c}{2} + 1 \leqslant c && \text{semplifico} \\ &\Rightarrow c \geqslant 2 \end{aligned}$$

  Abbiamo quindi provato che $T(n) \leqslant cn$, con due diversi valori della nostra costante c: nel caso base è risultata $c \geqslant 1$, mentre nel passo induttivo è risultata pari a $c \geqslant 2$. Quindi la nostra ipotesi è valida per qualsiasi valore di c t.c. $c \geqslant a$ e $c \geqslant 2$, ovvero $\forall c \geqslant 2$.

  Quanto abbiamo dimostrato vale per $n = 1$ e per tutti i valori di n successivi, di conseguenza $m = 1$.

Al solo scopo didattico (in quanto lo potremmo dedurlo dal termine non ricorsivo) proviamo passo passo anche il limite inferiore, ossia che $\boxed{T(n) = \Omega(n)}$

Limite inferiore: Dobbiamo dimostrare che $\exists d > 0, \exists m \geqslant 0 : T(n) \geqslant dn, \forall n \geqslant m$.

Nota. Usiamo la costante d al solo scopo di non confonderla la costante c usata nella dimostrazione precendente.

- **caso base** $T(1)$:

$$T(1) = 1 \overset{?}{\geqslant} d \cdot 1 \iff d \leqslant 1$$

- **ipotesi induttiva** $\forall k < n : T(k) \geqslant ck$, ossia assumiamo che per tutti i valori più piccoli di n la mia dimostrazione sia già stata fatta.
- **ipotesi induttiva** dimostriamo la disequazione per $T(n)$

$$\begin{aligned}
T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } d = \tfrac{n}{2} \\
&\geqslant d\lfloor \tfrac{n}{2} \rfloor + n && \text{semplifico l'intero inferiore} \\
&\geqslant d\tfrac{n}{2} - 1 + n && \text{raccolgo } n \\
&= \left(\tfrac{d}{2} - \tfrac{1}{n} + 1\right) n && \text{obiettivo} \\
&= \left(\tfrac{d}{2} - \tfrac{1}{n} + 1\right) n \overset{?}{\geqslant} dn && \text{semplifico} \\
&\Rightarrow \tfrac{d}{2} - \tfrac{1}{n} + 1 \geqslant n \\
&\Rightarrow d \leqslant 2 - \tfrac{2}{n}
\end{aligned}$$

Abbiamo quindi provato che $T(n) \geqslant dn$, con due diversi valori della nostra costante d: nel caso base è risultata $d \leqslant 1$, mentre nel passo induttivo è risultata pari a $d \leqslant 2 - \frac{2}{n}$. $d = 1$ è valore che soddisfa entrambre le disequazioni, dimostra quindi la nostra tesi.

Quanto abbiamo dimostrato vale per $n = 1$ e per tutti i valori di n successivi, di conseguenza $m = 1$.

Per concludere abbiamo provato che $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n$ è limitata sia superiormente $T(n) = \mathcal{O}(n)$, sia inferiormente $T(n) = \Omega(n)$, possiamo affermare quindi con assoluta certezza che $T(n) = \Theta(n)$.

Nota che avremmo potuto evitare la dimostrazione del limite inferiore osservando che

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + n \geqslant n \overset{?}{\geqslant} dn$$

l'ultima disequazione risulta vera per $d \leqslant 1$, la quale è una condizione identica a quella del caso base. Nota che non abbiamo fatto nemmeno ricordo all'ipotesi induttiva.

Cosa succede se si sbaglia l'intuizione

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

Nota. L'equazione di riccorrenza rappresenta il caso in cui selection sort sia espresso in forma ricorsiva.

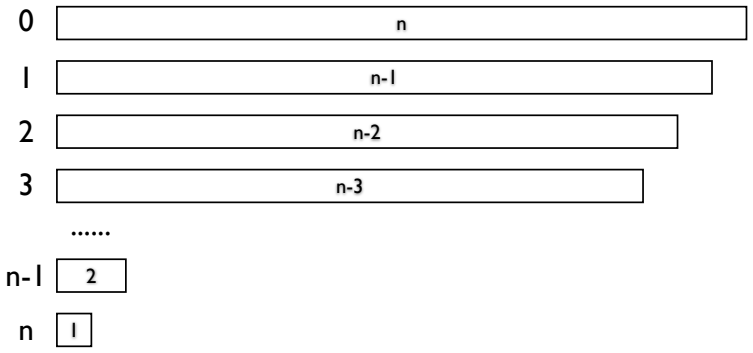


Figura 3.2: Rappresentazione della ricorrenza lineare.

Possiamo rappresentare la funzione nel seguente modo:

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

Effettuiamo un tentativo e proviamo a dimostrare che $\boxed{T(n) = \mathcal{O}(n)}$

Limite superiore: Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn, \forall n \geqslant m$.

- **caso base** lo saltiamo perché vedremmo subito che è sbagliato.
- **ipotesi induttiva** $\forall k < n : T(k) \leqslant ck$.
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned}
T(n) &= T(n-1) + n && \text{sost. ip. ind. con } k = n-1 \\
&= c(n-1) + n && \text{moltiplico} \\
&= cn - c + n && \text{raccolgo } n \\
&= (c+1)n - c && \text{rimuovo l'elemento negativo} \\
&\leqslant (c+1)n && \text{obiettivo} \\
&= (c+1)n \overset{?}{\leqslant} cn && \text{semplifico} \\
&\Rightarrow c + 1 \leqslant c
\end{aligned}$$

Possiamo notare che l'ultima disequazione risulta impossibile. Dunque quando proviamo a dimostrare qualcosa di sbagliato non riusciremo a dimostrarlo.

Difficoltà matematica

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Nota. È possibile ottenere questa equazione di riccorrenza dall'algoritmo che calcola il minimo di un vettore non ordinato in maniera ricorsiva.

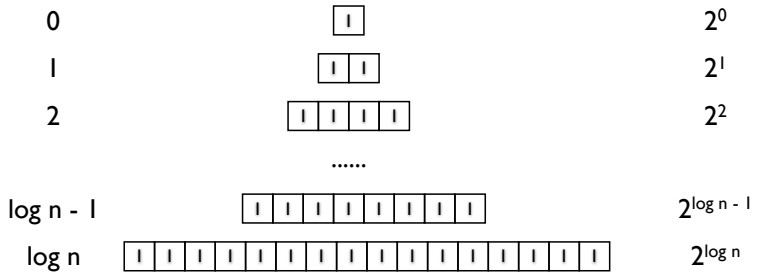


Figura 3.3: Rappresentazione della ricorsione.

$$T(n) = \sum_{i=0}^{\log n} 2^i = n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = \mathcal{O}(n)$$

Effettuiamo un tentativo per $\boxed{T(n) = \mathcal{O}(n)}$

- **ipotesi induttiva**: $\forall k < n : T(k) \leqslant ck$.
- **passo induttivo**: dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.} \\ &= c(\lfloor \tfrac{n}{2} \rfloor) + c(\lceil \tfrac{n}{2} \rceil) + 1 && \text{semplifichiamo, } \lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil = n \\ &= cn + 1 && \text{obiettivo} \\ &= cn + 1 \overset{?}{\leqslant} cn && \text{semplifico} \\ &\Rightarrow 1 \leqslant 0 \end{aligned}$$

Anche in questo caso notiamo che l'ultima disequazione risulta impossibile, ma – a differenza del caso precendente – non riusciamo a dimostrare il passo induttivo per un termine di ordine inferiore. Il tentativo risulta quindi errato.

Proviamo quindi ad utilizzare un'ipotesi induttiva *più stretta*.

- **ipotesi induttiva più stretta**: $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn - b, \forall n \geqslant m,\ b > 0$.

  Abbiamo introdotto una costante $b > 0$ nella nostra tesi, questa modifica ci permetterà di dimostrare correttamente il passo induttivo.

- **ipotesi induttiva** $\exists b > 0, \forall k < n : T(k) \leqslant ck - b$.

- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.} \\ &= c(\lfloor \tfrac{n}{2} \rfloor) - b + c(\lceil \tfrac{n}{2} \rceil) - b + 1 && \text{semplifichiamo} \\ &= cn - 2b + 1 && \text{obiettivo} \\ &= cn - 2b + 1 \overset{?}{\leqslant} cn - b && \text{semplifico} \\ &\Leftrightarrow -2b + 1 \leqslant -b \\ &\Leftrightarrow b \geqslant 1 \end{aligned}$$

- **caso base**

$$T(1) = 1 \overset{?}{\leqslant} c \cdot 1 - b \Leftrightarrow c \geqslant b + 1$$

  Per concludere abbiamo provato che $T(n) \leqslant cn - b \leqslant cn$ con diversi valori delle costanti c e b, nel passo induttivo $\forall b \geqslant 1, \forall c$, nel caso base $\forall c \geqslant b + 1$. Una coppia di valori di b e c che rispettano queste disequazioni sono $b = 1$, $c = 2$.

  Questo vale per $n = 1$, e per tutti i valori di n successivi, quindi per $m = 1$.

  Abbiamo quindi provato che $T(n) = \mathcal{O}(n)$.

Dimostriamo il limite inferiore facendo un tentativo per $\boxed{T(n) = \Omega(n)}$

Dobbiamo dimostrare che $\exists d > 0, \exists m \geqslant 0 : T(n) \geqslant dn, \forall n \geqslant m$.

- **passo induttivo**

$$\begin{aligned} T(n) &= T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + 1 && \text{sost. ip. ind.} \\ &\geqslant d(\lfloor \tfrac{n}{2} \rfloor) + d(\lceil \tfrac{n}{2} \rceil) + 1 && \text{semplifichiamo} \\ &= dn + 1 && \text{obiettivo} \\ &= dn + 1 \overset{?}{\geqslant} dn \end{aligned}$$

  L'ultima disequazione risulta vera $\forall d$.

- **caso base**

$$T(n) = 1 \geqslant d \cdot 1 \iff d \leqslant 1$$

  Abbiamo quindi provato che $T(n) = \Omega(n)$

Problemi con i casi base

$$T(n) = \begin{cases} 2T(\lfloor \frac{n}{2} \rfloor) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Proviamo a visualizzarlo graficamente così:

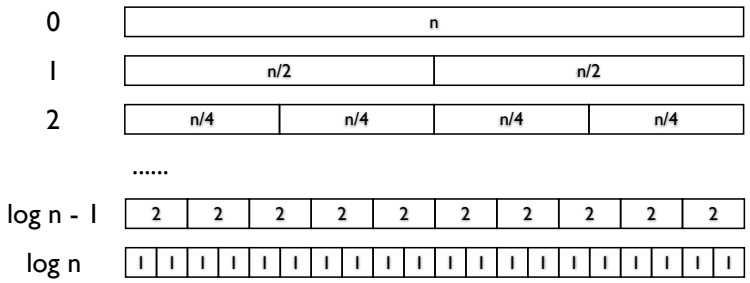


Figura 3.4: Rappresentazione della ricorsione.

Nota. È molto simile all'equazione dell'algoritmo mergeSort che sappiamo avere una complessità di $\mathcal{O}(n \log n)$.

Effettuiamo un tentativo per $\boxed{T(n) = \mathcal{O}(n \log n)}$

Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn \log n, \forall n \geqslant m$.

- **Ipotesi induttiva**: $\exists c > 0, \forall k < n : T(k) \leqslant ck \log k$
- **Passo di induzione**: Dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 2T(\lfloor \tfrac{n}{2} \rfloor) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{2} \rfloor \\ &\leqslant 2c\lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor + n && \text{rimuovo l'intero inferiore} \\ &\leqslant 2c\tfrac{n}{2} \log \tfrac{n}{2} + n && \text{semplifico} \\ &= cn \log \tfrac{n}{2} + n && \log \tfrac{n}{2} = \log n - \log_2 2 = \log n - 1 \\ &= cn(\log n - 1) + n && \text{moltiplico} \\ &= cn \log n - cn + n && \text{obiettivo} \\ &\leqslant cn \log n - cn + n \overset{?}{\leqslant} cn \log n \\ &\Leftrightarrow -cn + n \leqslant 0 && \text{semplifico} \\ &\Leftrightarrow c \geqslant 1 \end{aligned}$$

- **caso base**: dimostriamo la disequazione per $T(1)$

$$T(1) = 1 \overset{?}{\leqslant} 1 \cdot c \log 1 = 0 \Rightarrow 1 \not\leqslant 0$$

  È falso, ma non è un problema, non a caso si chiama notazione asintotica: il valore di m lo possiamo scegliere noi.

- **caso base**: dimostriamo la disequazione per $T(2)$, $T(3)$

$$\begin{aligned} T(2) &= 2T(\lfloor \frac{2}{2} \rfloor) + 2 = 4 \leqslant 1 \cdot c \cdot 2 \log 2 \Leftrightarrow c \geqslant 2 \\ T(3) &= 2T(\lfloor \frac{3}{2} \rfloor) + 3 = 5 \leqslant 1 \cdot c \cdot 3 \log 3 \Leftrightarrow c \geqslant \frac{5}{3 \log 3} \\ T(4) &= 2T(\lfloor \frac{4}{2} \rfloor) + 4 = 2T(\lfloor 2 \rfloor) + 4 \end{aligned}$$

Non è necessario provare la terza disequazione, in quanto viene espressa in base ai casi base diversi da $T(1)$ che sono già stati dimostrati e quindi possono costituire la base per la nostra induzione.

Riassumendo:

Abbiamo provato che $T(n) \leqslant cn \log n$

- nel passo induttivo: $\forall c \geqslant 1$
- nel caso base: $\forall c \geqslant 2, c \geqslant \frac{5}{3 \log 3}$

Visto che sono tutte disequazioni con il segno $\geqslant$, è sufficiente utilizzare $c \geqslant \max\left\{1, 2, \frac{5}{3 \log 3}\right\}$

Questo vale per $n = 2$, $n = 3$, e per tutti i valori di n successivi, quindi $m = 2$.

Ultimo esercizio

$$T(n) = \begin{cases} 9T(\lfloor n/3 \rfloor) + n & n > 1 \\ 1 & n \leqslant 1 \end{cases}$$

Effettuiamo un tentantivo $\boxed{T(n) = \mathcal{O}(n^2)}$

Dobbiamo dimostrare che $\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn^2, \forall n \geqslant m$

- **ipotesi induttiva**: $\exists c > 0 : T(k) \leqslant ck^2, \forall k < n$
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 9T\left(\lfloor \tfrac{n}{3} \rfloor\right) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{3} \rfloor \\ &\leqslant 9c\left(\lfloor \tfrac{n}{3} \rfloor\right)^2 + n && \text{rimuovo l'intero inferiore} \\ &\leqslant 9c\left(\tfrac{n^2}{9}\right) + n && \text{semplifico il 9} \\ &= cn^2 + n && \text{obiettivo} \\ &= cn^2 + n \leqslant cn^2 \end{aligned}$$

L'ultima disequazione risulta falsa per un termine di ordine inferiore: proviamo quindi a modificare l'ipotesi induttiva e a ripetere il passo induttivo.

- **ipotesi induttiva più stretta** $\exists c > 0 : T(k) \leqslant c(k^2 - k), \forall k < n$
- **passo induttivo** dimostriamo la disequazione per $T(n)$:

$$\begin{aligned} T(n) &= 9T\left(\lfloor \tfrac{n}{3} \rfloor\right) + n \\ &\leqslant 9c\left(\lfloor \tfrac{n}{3} \rfloor^2 - \lfloor \tfrac{n}{3} \rfloor\right) + n && \text{sost. ip. ind. con } k = \lfloor \tfrac{n}{3} \rfloor \\ &\leqslant 9c\left(\left(\tfrac{n}{3}\right)^2 - \tfrac{n}{3}\right) + n && \text{rimuovo l'intero inferiore} \\ &\leqslant 9c\left(\tfrac{n^2}{9} - \tfrac{n}{3}\right) + n && \text{svolgo la potenza} \\ &\leqslant cn^2 - 3cn + n && \text{moltiplico} \\ &\leqslant cn^2 - 3cn + n \overset{?}{\leqslant} cn^2 - cn && \text{obiettivo} \\ &\Leftrightarrow 2cn \geqslant cn && \text{semplifico} \\ &\Leftrightarrow c \geqslant \tfrac{1}{2} \end{aligned}$$

- **caso base**

$$T(1) = 1 \leqslant c(1^2 - 1) = 0, falso$$
$$T(2) = 9T(0) + 2 = 11 \leqslant c(2^2 - 2) \Leftrightarrow c \geqslant {}^{11}/_{2}$$
$$T(3) = 9T(1) + 3 = 12 \leqslant c(3^2 - 3) \Leftrightarrow c \geqslant {}^{12}/_{6}$$
$$T(4) = 9T(1) + 4 = 13 \leqslant c(4^2 - 4) \Leftrightarrow c \geqslant {}^{13}/_{12}$$
$$T(5) = 9T(1) + 5 = 14 \leqslant c(5^2 - 5) \Leftrightarrow c \geqslant {}^{14}/_{20}$$
$$T(6) = 9T(2) + 6$$

  Non è necessario andare oltre poiché T(6) dipende da T(2) che è già stato dimostrato.

  Riassumendo i parametri scelti sono:

  - $c \geqslant \max\left\{\frac{1}{2}, \frac{11}{2}, \frac{12}{6}, \frac{13}{12}, \frac{14}{20}\right\}$
  - $m = 1$

  Nota che l'esempio combina le due difficoltà insieme, ma è stato creato artificiosamente: infatti se avessimo scelto come ipotesi più stretta $T(n) \leqslant cn^2 - bn$, il problema sui casi base non si sarebbe posto.

  Abbiamo quindi dimostrato che $T(n) \leqslant c(n^2 - n) \leqslant cn^2, \forall n \geqslant 1, \forall c \geqslant \frac{14}{20}$, ossia che $T(n) = \mathcal{O}(n)$

Riassumendo

Il metodo di sostituzione è composto da tre parti:

1. si *indovina* una possibile soluzione e si formula un'ipotesi induttiva;
2. si *sostituisce* nella ricorrenza le espressioni $T(\cdot)$, utilizzando l'ipotesi induttiva;
3. si *dimostra* che la soluzione è valida anche per il caso base.

Bisogna fare attenzione:

- ad ipotizzare soluzioni troppo "strette";
- ad alcuni casi particolari che richiedono astuzie matematiche;
- ai casi base in cui compare il logaritmo in quanto potrebbe complicare le cose.

3.4 Metodo dell'esperto (o delle ricorrenze comuni)

Esiste un'ampia classe di ricorrenze che possono essere risolte facilmente facendo ricorso ad alcuni teoremi, ognuno dei quali si occupa di una classe particolare di equazioni di ricorrenza.

Teorema 8 (ricorrenze lineari con partizione bilanciata)**.** *Siano a e b costanti intere tale che $a \geqslant 1$ e $b \geqslant 2$, e c, β costanti reali tali che $c > 0$ e $\beta \geqslant 0$. Sia $T(n)$ data dalla relazione di ricorrenza:*

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

Posto $\alpha = \frac{\log a}{\log b} = \log_b a$, allora:

$$T(n) = \begin{cases} \Theta(n^{\alpha}) & \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \alpha = \beta \\ \Theta(n^{\beta}) & \alpha < \beta \end{cases}$$

Commento Affrontiamo le equazioni di ricorrenza in cui la dimensione viene divisa in b parti, dove b dev'essere almeno pari a 2; l'algoritmo ricorsivo dev'essere richiamato a volte, dove a è almeno 1. Nella versione estesa, che vedremo fra poco, vedremo che i parametri a e b verranno "rilassati".

Prendiamo ad esempio il primo esercizio $T(n) = 4T(\frac{n}{2})$ e vediamo che si può risolvere semplicemente calcolando $\alpha = \log_b a = \log_2 4 = 2 > \beta = 1$, possiamo quindi concludere che $T(n) = \Theta(n^{\alpha}) = \Theta(n^2)$.

Dimostrazione del teorema delle ricorrenze lineari con partizione bilanciata. Assumiamo che n sia una potenza intera di b, ossia che $n = b^k$, $k = \log_b n$ poiché ci permetterà di semplificare i calcoli successivi ed è ininfluente sul risultato. Ad esempio supponiamo che l'input abbia dimensione $b^k + 1$ ($2^8 + 1 = 257$ bit), se estendiamo l'input fino ad una dimensione b^{k+1} ($2^8 + 1 = 512$ bit, facendo del *padding*, l'input sarebbe stato esteso al massimo di un fattore costante b (2 nel nostro caso), il che è ininfluente al fine della complessità computazionale.

Calcoliamo l'albero delle ricorrenze per la seguente equazione di ricorrenza:

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

livello	dim.	costo chiam.	no. chiamate	costo livello
0	b^k	$c\,b^{k\beta}$	1	$cb^{k\beta}$
1	b^{k-1}	$c\,b^{(k-1)\beta}$	a	$a \cdot c\,b^{(k-1)\beta}$
2	b^{k-2}	$c\,b^{(k-2)\beta}$	a^2	$a^2 \cdot c\,b^{(k-2)\beta}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	b^{k-i}	$c\,b^{(k-i)\beta}$	a^i	$a^i \cdot c\,b^{(k-i)\beta}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	b	$cb^{(k-(k-1))\beta} = c\,b^{\beta}$	a^{k-1}	$a^{k-1} \cdot c\,b^{\beta}$
k	1	d	a^k	$a^k \cdot d$

Sommando i costi totali del k-esimo livello e dei livelli fino al $k-1$, si ottiene:

$$
\begin{aligned}
T(n) &= da^k + \sum_{i=0}^{k-1} a^i \cdot cb^{(k-i)\beta} && \text{moltiplico} \\
&= da^k + \sum_{i=0}^{k-1} a^i \cdot cb^{k\beta} \cdot b^{-i\beta} && \text{porto fuori i termini non dipendenti da } i \\
&= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \frac{a^i}{b^{i\beta}} && \text{raccolgo } i \\
&= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{a}{b^{\beta}}\right)^i
\end{aligned}
$$

A questo punto ho una formula chiusa ma non ancora nella sua forma definitiva.

Facciamo alcune osservazioni.

- $a^k = a^{\log_b n} = a^{\frac{\log n}{\log b}} = 2^{\log_2 a \frac{\log n}{\log b}} = 2^{\log_2 n \frac{\log a}{\log b}} = n^{\frac{\log a}{\log b}} = n^{\alpha}$
- $\alpha = \frac{\log a}{\log b} \Leftrightarrow \alpha \log b = \log a \Leftrightarrow \log b^{\alpha} = \log a \Leftrightarrow a = b^{\alpha}$
- poniamo $q = \frac{a}{b^{\beta}} = \frac{b^{\alpha}}{b^{\beta}} = b^{\alpha-\beta}$

Grazie alle osservazioni appena fatto possiamo sostituire i parametri nell'equazione finale:

$$
\begin{aligned}
T(n) &= da^k + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{a}{b^{\beta}}\right)^i && \text{sostituisco: } \alpha^k \to n^{\alpha} \\
&= dn^{\alpha} + cb^{k\beta} \sum_{i=0}^{k-1} q^i && \text{sostituisco: } a \to b^{\alpha},\ q = \tfrac{b^{\alpha}}{b^{\beta}}
\end{aligned}
$$

Da qui si aprono tre possibilità, ossia che 1. $\alpha > \beta$ 2. $\alpha = \beta$ 3. $\alpha < \beta$ Studiamole una per una.

1. Caso $\boxed{\alpha > \beta}$, ne segue che $q = b^{\alpha-\beta} > 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i && \text{serie geometrica finita}\\
&= n^\alpha d + cb^{k\beta}\left[\frac{q^k-1}{q-1}\right] && \text{introduco la disequazione}\\
&\leqslant n^\alpha d + cb^{k\beta}\frac{q^k}{q-1} && \text{sostituisco: } q = \tfrac{a}{b^\beta} \Rightarrow q^k = \tfrac{a^k}{b^{k\beta}}\\
&= n^\alpha d + \frac{cb^{k\beta}a^k}{b^{k\beta}}\frac{1}{q-1} && \text{semplifico: } b^{k\beta}\\
&= n^\alpha d + \frac{ca^k}{q-1} && \text{sostituisco: } a^k = n^\alpha\\
&= n^\alpha d + \frac{cn^\alpha}{q-1} && \text{raccolgo per } n^\alpha\\
&= n^\alpha\left[d + \frac{c}{q-1}\right]
\end{aligned}
$$

   Visto che d, c e q sono tutti termini positivi e costanti possiamo concludere che n^α limita superiormente l'espressione e che quindi $T(n) = \mathcal{O}(n^\alpha)$. Infine per via della componente non ricorsiva dn^α, $T(n)$ è anche $\Omega(n^\alpha)$, possiamo concludere che $T(n) = \Theta(n^\alpha)$.

2. Caso $\boxed{\alpha = \beta}$, ne segue che $q = b^{\alpha-\beta} = 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\sum_{i=0}^{k-1} q^i && q^i = 1^i = 1\\
&= n^\alpha d + cn^\beta k && 1 + 2 + \cdots + k - 1 = k\\
&= n^\alpha d + cn^\alpha k && \text{sostituisco: } \beta = \alpha\\
&= n^\alpha(d + ck) && \text{raccolgo per } n^\alpha\\
&= n^\alpha(d + c\tfrac{\log n}{\log b}) && \text{sostituisco: } k = \log_b n
\end{aligned}
$$

   Visto che d, c e $\log b$ sono tutti termini positivi e costanti e che non abbiamo introdotto disequazioni, possiamo affermare che $T(n) = \Theta(n^\alpha \log n)$.

3. Caso $\boxed{\alpha < \beta}$, ne segue che $q = b^{\alpha-\beta} < 1$

$$
\begin{aligned}
T(n) &= dn^\alpha + cb^k\sum_{i=0}^{k-1} q^i && \text{serie geometrica finita}\\
&= dn^\alpha + cb^k\left[\frac{q^k-1}{q-1}\right] && \text{inversione, } 1-q>0\\
&= dn^\alpha + cb^k\left[\frac{1-q^k}{1-q}\right] && \text{introduco}\\
&\leqslant dn^\alpha + cb^k\left[\frac{1}{1-q}\right] && \text{la disequazione}\\
&= n^\alpha d + \frac{cn^\beta}{1-q} && \text{sostituisco: } b^k = n
\end{aligned}
$$

   $n^\alpha < n^\beta$ quindi considero il polinomio di grado maggiore, di conseguenza $T(n)$ è $\mathcal{O}(n^\beta)$. Poiché $T(n) = \Omega(n^\beta)$ per via del termine non ricorsivo, possiamo affermare che $T(n) = \Theta(n^\beta)$.

Fine dimostrazione. □

Teorema (ricorrenze lineari con partizione bilanciata estesa)**.** *Sia $a \geqslant 1$, $b > 1$, $f(n)$ asintoticamente positiva, e sia*

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ d & n \leqslant 1 \end{cases}$$

Sono dati tre casi:

1. $\exists \varepsilon > 0 : f(n) = \mathcal{O}(n^{\alpha-\varepsilon})$ *allora* $T(n) = \Theta(n^{\alpha})$*;*
2. $f(n) = \Theta(n^{\alpha})$ *allora* $T(n) = \Theta(f(n) \log n)$*;*
3. $\exists \varepsilon > 0 : f(n) = \mathcal{O}(n^{\alpha+\varepsilon}) \wedge$
   $\exists c : 0 < c < 1, \exists m > 0 :$
   $af(\frac{n}{b}) \leqslant cf(n), \forall n \geqslant m$ *allora* $T(n) = \Theta(f(n))$.

Non vedremo la dimostrazione poiché è troppo complessa.

Commento Il teorema precendente funzionava con n^{β}, aveva una serie di condizioni semplificative, ora non ci sono più. Nel secondo caso se $f(n) = n^{\beta}$ ritorniamo esattamente al secondo caso del teorema precendente, ma qui possiamo prendere in considerazione funzioni più complesse.

Esercizi

Le soluzioni sono indicate fra parentesi.

- $T(n) = 9T(\frac{n}{3}) + n$ [$\mathcal{O}(n^{2-\varepsilon})$, con $\varepsilon < 1$]
- $T(n) = T(\frac{2}{3}n) + 1$ [$\Theta(n^0)$]
- $T(n) = 3T(\frac{n}{4}) + n \log n$ [$c = 3/4$, $m = 1$]
- $T(n) = 2T(\frac{n}{2}) + n \log n$ [nessun caso applicabile]

Teorema (ricorrenze lineari di ordine costante)**.** *Siano $\{a_1, a_2, \ldots, a_n\}$ costanti intere non negative, con h costante positiva, c e β costanti reali tali che $c > 0$ e $\beta \geqslant 0$, e sia $T(n)$ definita dalla relazione di ricorrenza:*

$$T(n) = \begin{cases} \sum\limits_{1 \leqslant i \leqslant h} a_i T(n-1) + cn^{\beta} & n > m \\ \Theta(1) & n \leqslant m \leqslant h \end{cases}$$

Posto $a = \sum_{1 \leqslant i \leqslant h} a_i$, allora:

1. $T(n)$ *è* $\Theta(n^{\beta+1})$*, se* $a = 1$*;*
2. $T(n)$ *è* $\Theta(a^n n^{\beta})$*, se* $a \geqslant 2$.

Commento Questo teorema tratta ricorrenze lineari *di ordine costante* perché tutte le volte rimuoviamo dalla dimensione di input n una quantità costante.

Esercizi

Le soluzioni sono indicate fra parentesi.

- $T(n) = T(n-10) + n^2$ [costo polinomiale]
- $T(n) = T(n-2) - T(n-1) + 1$ [costo esponenziale]

3.5 Algoritmo della somma massimale di un sottovettore

Date le nostre nuove conoscenze possiamo calcolare con precisione la complessità delle varie versioni degli algoritmi proposti per la soluzione al problema della somma massimale di un sottovettore.

Complessità della prima versione

```
int maxsum1(int[] A, int n) {
    int maxSoFar = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum = sum + A[k];
            }
            maxSoFar = max(maxSoFar, sum);
        }
    }
    return maxSoFar;
}
```

La complessità dell'algoritmo può essere approssimata come segue (contando il numero di esecuzioni della riga più interna):

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$$

Vogliamo provare che $T(n) = \mathcal{O}(n^3)$.

Dimostrazione. **limite superiore**: $\exists c_2 > 0, \exists m \geqslant 0 : T(n) \leqslant c_2 n^3,\ \forall n \geqslant m$.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) && \text{spiegazione} \\
&\leqslant \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} n && \text{spiegazione} \\
&\leqslant \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n && \text{spiegazione} \\
&= \sum_{i=0}^{n-1} n^2 && \text{spiegazione} \\
&= n^3 \leqslant c_2 n^3
\end{aligned}$$

Questa disequazione è vera per $n \geqslant m = 0$ and $c_2 \geqslant 1$. □

Vogliamo provare che $T(n) = \Omega(n^3)$.

Dimostrazione. **limite inferiore**: $\exists c_1 > 0, \exists m \geqslant 0 : T(n) \leqslant c_1 n^3,\ \forall n \geqslant m$.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\
&\geqslant \sum_{i=0}^{n/2} \sum_{j=i}^{n+n/2-1} (j - i + 1) && \text{spiegazione} \\
&= \sum_{i=0}^{n/2} \sum_{j=i}^{n+n/2-1} n/2 && \text{spiegazione} \\
&= \sum_{i=0}^{n/2} n^2/4 \geqslant n^3/8 \geqslant c_1 n^3
\end{aligned}$$

Questa disequazione è vera per $n \geqslant m = 0$ and $c_1 \geqslant 8$. □

Complessità della seconda versione

```
int maxsum2(int[] A, int n) {
    int maxSoFar = 0;
    for (int i=0; i < n; i++) {
        int sum = 0;
        for (int j=i; j < n; j++) {
            sum = sum + A[j];
            maxSoFar = max(maxSoFar, sum);
        }
    }
    return maxSoFar;
}
```

La complessità di questo algoritmo può essere approssimata come segue (stiamo contando il numero di passi nel ciclo più interno):

$$T(n) = \sum_{i=0}^{n-1} n - i$$

Vogliamo provare che $T(n) = \Theta(n^2)$

Dimostrazione.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} n - i \\ &= \sum_{i=1}^{n} i \qquad \text{spiegazione} \\ &= \frac{n(n+1)}{2} = \Theta(n^2) \qquad \text{spiegazione} \end{aligned}$$

Questo non richiede ulteriori spiegazioni. □

Complessità della terza versione

```
int maxsum_rec(int[] A, int i, int j) {
    if (i == j)
      return max(0, A[i]);

    int m = (i + j) / 2;
    int maxs = maxsum_rec(A, i, m);
    int maxd = maxsum_rec(A, m + 1, j);
    int maxss = 0;
    int sum = 0;

    for (int k = m; k >= i; k--) {
      sum = sum + A[k];
      maxss = max(maxss, sum);
    }

    int maxdd = 0;
    sum = 0;
    for (int k = m + 1; k <= j; k++) {
      sum = sum + A[k];
      maxdd = max(maxdd, sum);
    }

    return max(max(maxs, maxd), maxss + maxdd);
}
```

Per questo, definiamo la equazione di ricorrenza:

$$T(n) = 2T(n/2) + n$$

Utilizzando il teorema, possiamo vedere che $\alpha = \log_2 2 = 1$ e $\beta = 1$, quindi $T(n) = \Theta(n \log n)$.

Complessità della quarta versione

```
int maxsum4(int A[], int n) {
    int maxSoFar = 0;
    int maxHere = 0;
    for (int i = 0; i < n; i++) {
        maxHere = max(maxHere + A[i], 0);
        maxSoFar = max(maxSoFar, maxHere);
    }

    return maxSoFar;
}
```

È facile vedere che la complessità di questa versione è $\Theta(n)$.

Capitolo 4

Analisi ammortizzata

Capitolo 5

Strutture dati

> *Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.*
>
> Steven S. Skiena, *The Algorithm Design Manual*

5.1 Strutture dati astratte

Alcune definizioni

Definizione 5.1.1 (Tipo di dato)**.** In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.

Definizione 5.1.2 (Tipo di dato astratto)**.** Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.

Definizione 5.1.3 (Tipi di dato primitivi)**.** Sono dei tipi di dati che vengono forniti direttamente dal linguaggio. Come ad esempio: int (`+,-,*,/, %`), boolean (`!, &&, ||`).

Ogni tipo di dato deve distinguere *specifica* ed *implementazione* di un tipo di dato astratto. La *specifica* è astratta, il "manuale d'uso" che nasconde i dettagli implementativi all'utente, mentre l'*implementazione* è la realizzazione vera e propria del tipo di dato.

Tabella 5.1: Differenza fra specifica ed implementazione

Specifica	Implementazione
Numeri reali	IEEE-754
Pile	Pile basate su vettori Pile basate su puntatori
Code	Code basate su vettori circolari Code basate su puntatori

Definizione 5.1.4 (Strutture di dati)**.** Le strutture di dati sono collezioni di dati, caratterizzate più dall'organizzazione della collezione piuttosto che dal tipo dei dati in esse contenute.

Le strutture dati sono un modo sistematico per organizzare i dati e su di esse sono definite un insieme di operatori che permettono di manipolare la struttura stessa. Le strutture dati possono essere caratterizzate in vari modi:

- *lineari*/*non lineari*: presentano (o meno) una sequenza al loro interno;
- *statiche*/*dinamiche*: possono variare (o meno) di dimensione o di contenuto;
- *omogenee*/*disomogenee*: si riferisce ai dati contenuti al loro interno.

Tabella 5.2: Implementazione delle strutture dati nei vari linguaggi.
Nota che Java distingue chiaramente la specifica dall'implementazione

Tipo	Java	C++	Python
Sequenze	`List`, `Queue`, `Deque`, `LinkedList`, `ArrayList`, `Stack`, `ArrayDeque`	`list`, `forward_list`, `vector`, `stack`, `queue`, `dequeue`	`list`, `tuple`
Insiemi	`Set`, `TreeSet`, `HashSet`, `LinkedHashSet`	`set`, `unordered_set`	`set`, `fronzenset`
Dizionari	`Map`, `HashTree`, `HashMap`, `LinkedHashMap`	`map`, `unordered_map`	`dict`
Alberi	-	-	-
Grafi	-	-	-

5.2 Sequenza

Una sequenza è una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove un valore può comparire più di una volta. L'ordine all'interno della sequenza è importante.

Le operazioni ammesse su una sequenza sono:

- L'aggiunta e la rimozione elementi, specificando la posizione (tipicamente un intero), l'elemento s_1 si trova in posizione pos_i ed esistono posizioni fittizie pos_0 e pos_{n+1};
- Accesso diretto alla testa e coda;
- Accesso sequenziale a tutti gli altri elementi.

Specifica SEQUENCE

Una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove lo stesso valore può comparire più volte.

Sequence

```
// INTERPRETARE
boolean isEmpty      // true se la sequenza è vuota
boolean finished     // true se p è uguale a pos_0 o a
 pos_{n+1}

// LEGGERE
Pos head             // posizione del primo elemento
Pos tail             // posizione dell'ultimo elemento

// ITERARE
Pos next             // posizione dell'elem. che segue p
Pos prev             // posizione dell'elem. che precede p

// MODIFICA

// inserisce l'elemento di tipo Item nella
   posizione p,
// ritorna la nuova posizione,
// che diviene il predecessore di p
Pos insert(Pos p, Item v)
// rimuove l'elemento contenuto nella pos. p,
// ritorna il successore di p
Pos remove(Pos p)

// legge l'elemento di tipo Item
// contenuto nella posizione p
read(Pos p)

// scrive l'elemento v di tipo Item
// nella posizione p
write(Pos p, Item v)
```

5.2.1 Implementazione delle sequenze

Di seguito vengono presentati alcuni esempi d'utilizzo dell'implementazione delle sequenze nei diversi linguaggi di programmazione utilizzati oggigiorno.

Codice 5.1: Implementazione delle liste in Java

```
List<String> lista = new LinkedList<String>();
lista.add("two");
lista.addFirst("one");
lista.addLast("three");

Result: [ "one", "two", "three" ]
```

Codice 5.2: Implementazione delle liste in C++

```
std::list<int> lista;
lista.push_front(2);
lista.push_front(1);
lista.push_back(3);

Result: [1,2,3]
```

Codice 5.3: Implementazione delle liste in Python

```
lista = ["one", "three"]
lista.insert(1, "two")

Result: [ 'one', 'two', 'three' ]
```

5.3 Insiemi

Un insieme è una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti. L'ordinamento fra elementi è dato dall'eventuale relazione d'ordine definita sul tipo degli elementi stessi.

Le operazioni ammesse su un'insieme sono:

- operazioni di base: come inserimento, cancellazione e verifica di contenimento;
- operazione di ordinamento: massimo, minimo;
- operazioni insiemistiche: unione, intersezione, differenza;
- iteratori: effettuare operazione per ogni elemento contenuto nell'insieme.

Struttura dati SET

Una struttura dati *dinamica*, *non lineare* che memorizza una *collezione non ordinata di elementi* senza valori ripetuti.

Set

```
// INTERPRETARE
int size                    // cardinalità dell'insieme
boolean contains            // true se x è contenuto
```

```
// OPERAZIONI DI BASE
// inserisce x nell'insieme, se assente
insert(ITEM k)
// rimuove x nell'insieme, se presente
remove(ITEM k)

// OPERAZIONI INSIEMISTICHE
static SET union(SET A, SET B)
static SET intersection(SET A, SET B)
static SET difference(SET A, SET B)
```

Codice 5.4: Implementazione degli insiemi in Java

```
List<String> lista = new LinkedList<String>();
Set<String> docenti = new TreeSet<>();
docenti.add("Alberto");
docenti.add("Cristian");
docenti.add("Alessio");

Result: { "Alberto", "Alessio", "Cristian" }
```

Codice 5.5: Implementazione degli insiemi in C++

```
std::set<std::string> frutta;
frutta.insert("mele");
frutta.insert("pere");
frutta.insert("banane");
frutta.insert("mele");
frutta.remove("mele")

Result: { "banane", "pere" }
```

Codice 5.6: Implementazione degli insiemi in Python

```
items = { "rock", "paper", "scissors", "rock" }
print(items)
print("Spock" in items)
print("lizard" not in items)

Result: { "rock", "paper", "scissors" }
False
True
```

5.4 Dizionari

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* $R : D \to C$, o associazione chiave-valore, dove:

- l'insieme D è il dominio (gli elementi sono detti *chiavi*);
- l'insieme C è il codominio (gli elementi sono detti *valori*).

Le operazioni ammesse sui dizionari sono:

- ottenere il valore associato ad una particolare chiave (se presente) o **nil** se assente;
- inserire una nuova associazione chiave-valore, cancellando eventuali associazioni precedenti per la stessa chiave;
- rimuovere un'associazione chiave-valore esistente.

Specifica dizionario

Un dizionario è una struttura dati che rappresenta il concetto matematico di *relazione univoca* o associazione chiave-valore.

DICTIONARY

ITEM lookup(ITEM k)	// restituisce il valore associato alla chiave k, **nil** altrimenti
ITEM insert(KEY k, ITEM v)	// associa il valore v alla chiave k
remove(KEY k)	// rimuove l'associazione della chiave k

Codice 5.7: Implementazione dei dizionari in Java

```
Map<String, String> capoluoghi = new HashMap<>();
capoluoghi.put("Toscana", "Firenze");
capoluoghi.put("Lombardia", "Milano");
capoluoghi.put("Sardegna", "Cagliari");
```

Codice 5.8: Implementazione dei dizionari in C++

```
std::map<std::string, int> wordcounts;
std::string s;

while (std::cin >> s && s != "end")
  ++wordcounts[s];
```

Codice 5.9: Implementazione dei dizionari in Python

```
v = {}
v[10] = 5
v["alberto"] = 42
v[10]+v["alberto"]

Result: 47
```

5.5 Alberi

Un albero ordinato è dato da un insieme finito di elementi detti nodi. Uno di questi nodi è designato come radice. I rimanenti nodi, se esistono sono partizionati in insiemi *ordinati* e *disgiunti*, anch'essi alberi ordinati.

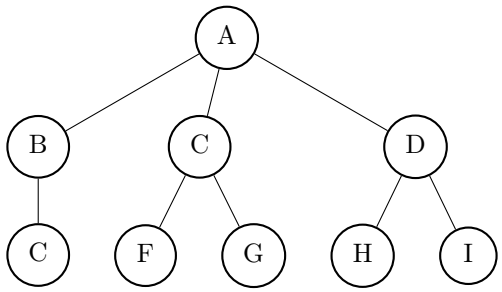


Figura 5.1: Un albero

Non vedremo implementazioni nei vari linguaggi in quanto non esiste una struttura dati definita riconosciuta universalmente.

5.6 Grafi

La struttura dati grafo è composta da:

- un insieme di elementi detti nodi o vertici;
- un insieme di coppie (ordinate oppure no) di nodi detti archi.

Tutte le operazioni su alberi e grafi ruotano attorno alla possibilità di effettuare visite su di essi, vedremo la specifica completa più avanti.

Nota. La scelta della struttura dati si riflette sull'efficienza e sulle operazioni ammesse.

5.7 Implementazione strutture dati elementari

5.7.1 Lista

Una lista è una sequenza di nodi, contenenti dati arbitrari e 1-2 puntatori all'elemento successivo e/o precedente.

La contiguità nella lista non implica che ci sia continuità nella memoria. Tutte le operazioni effettuate sulla lista hanno complessità $\mathcal{O}(1)$, ma per fare una ricerca dobbiamo spendere $\mathcal{O}(n)$.

Esistono diverse implmentazioni della lista, le quali possono essere:

- bidirezionale o monodirezionale;
- con sentinella o senza;
- circolare o non circolare.

Struttura dati lista bidirezionale con sentinella in pseudocodice

```
LIST                                  // bidirezionale con sentinella
LIST pred                             // predecessore
LIST succ                             // successore
LIST value                            // elemento

LIST List
    // la sentinella fa riferimento a sé stessa
    t.pred = t
    t.succ = t
    return t

POS head
    return succ

POS tail
    return pred

POS next
    return p.succ

POS prev
    return p.pred

boolean finished(POS p)
    return p = this

ITEM read(POS p)
    return p.value

write(POS p)
    return p.value

// posso fare queste operazioni essendo sicuro
// di avere sempre un predecessore
POS insert(POS p, ITEM v)
    LIST t = List t.value = v
    t.pred = p.pred
    p.pred.succ = t
    t.succ = p
    p.pred = t
    return p

POS remove(POS p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    LIST t = p.succ
    delete p
    return t
```

Il costo delle operazioni di lettura, scrittura, inserimento e rimozione per questa struttura è $\mathcal{O}(1)$.

Codice 5.10: Lista bidirezionale *senza* sentinella in Java

```
class Pos {
    Pos succ;    /** Prossimo elemento della lista */
    Pos pred;    /** Precedente elemento della lista */
    Object v;    /** Valore */

    Pos(Object v) {
        succ = pred = null;
        this.v = v;
    }
}

public class List {
    private Pos head;        /** Primo elemento della lista */
    private Pos tail;        /** Ultimo elemento della lista */

    public List() {
        head = tail = null;
    }

    public Pos head()                  { return head; }
    public Pos tail()                  { return tail; }
    public boolean finished(Pos pos)   { return pos == null; }
    public boolean isEmpty()           { return head == null; }
    public Object read(Pos p)          { return p.v; }
    public void write(Pos p, Object v) { p.v = v; }

    public Pos next(Pos pos) {
        return (pos != null ? pos.succ : null);
    }

    public Pos prev(Pos pos) {
        return (pos != null ? pos.pred : null);
    }

    public void remove(Pos pos) {
        if (pos.pred == null) // sto inserendo in testa
            head = pos.succ;
        else
            pos.pred.succ = pos.succ;

        if (pos.succ == null) // sto inserendo in coda
            tail = pos.pred;
        else
            pos.succ.pred = pos.pred;
    }

    public Pos insert(Pos pos, Object v) {
        Pos t = new Pos(v);

        if (head == null) {
            head = tail = t; // Inserisci in una lista vuota
        } else if (pos == null) {
            t.pred = tail; // Inserisci alla fine
            tail.succ = t;
            tail = t;
        } else {
            t.pred = pos.pred; // Iserimento davanti ad una posizione esistente
            if (t.pred != null)
                t.pred.succ = t;
            else
                head = t;

            t.succ = pos;
            pos.pred = t;
```

Figura 5.2: `xkcd no. 379`

5.7.2 Pila

La pila è una struttura dati *dinamica*, *lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per meno tempo nell'insieme" (con strategia LIFO, *Last-In-First-Out*).

Specifica STACK	
boolean isEmpty	`// restituisce vero se la pila è vuota`
push(ITEM v)	`// inserisce` v `in cima alla pila`
ITEM pop	`// estrae l'elemento in cima alla pila e lo restituisce al chiamante`
ITEM top	`// legge l'elemento in cima alla pila`

Ogni volta che viene effettuata una chiamata a funzione si usa implicitamente una pila, che memorizza tutti i record di attivazione delle chiamate effettuate. Sfrutteremo questo meccanismo implicito per visitare gli alberi, attraverso una visita in profondità.

Le pile possono essere implementate come:

- liste bidirezionali, dove il puntatore punta all'elemento top (non utilizzate);
- tramite vettore, dove la dimesione è limitata quindi si crea un *overhead* più basso.

Struttura dati pila basata su vettore in pseudocodice

```
ITEM[] A                                  // elementi
int n                                     // cursore
int m                                     // dimesione massima

// crea una pila vuota
STACK Stack(int dim)
    STACK t = new STACK
    t.A = new int[0...dim - 1]
    t.m = dim
    t.n = 0

    return t

// leggi l'elemento in cima alla pila
ITEM top
    precondition: n > 0

    return A[n]

// restituisce true se la pila è vuota
boolean isEmpty
    return n == 0

// estrae l'elemento in cima alla pila e lo
   restituisce al chiamante
ITEM pop
    precondition: n > 0

    ITEM t = A[n]
    n++

    return t

// inserisce v in cima alla pila
push(ITEM v)
    precondition: n < m

    n++
    A[n] = v
```

Codice 5.11: Pila basata su vettore circolare in Java

```
public class VectorStack implements Stack {

    /** Vector containing the elements */
    private Object[] A;

    /** Number of elements in the stack */
    private int n;

    public VectorStack(int dim) {
        n = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[n-1];
    }

    public Object pop() {
        if (n == 0)
            throw new IllegalStateException("Stack is empty");

        return A[--n];
    }

    public void push(Object o) {
        if (n == A.length)
            throw new IllegalStateException("Stack is full");

        A[n++] = o;
    }

}
```

5.7.3 Coda

La coda è una struttura dati *dinamica lineare* in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato, ed è quello che "è rimasto per più tempo nell'insieme" (con strategia, FIFO, *First-In-First-Out*).

Specifica QUEUE	
boolean isEmpty	`// restituisce vero se la coda è vuota`
ITEM enqueue(ITEM v)	`// inserisce` v `in fondo alla coda`
ITEM dequeue	`// estrae l'elemento in cima alla coda e lo restituisce al chiamante`
ITEM top	`// legge l'elemento in testa alla coda`

Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda. La politica FIFO è onesta (*fair*) rispetto l'ordine in cui i processi sono stati inseriti.

Le code possono essere implementate come:

- liste monodirezionali, dove sono presenti due puntatori: uno alla testa (*head*) per l'estrazione, ed uno alla coda per l'inserimento;
- vettori circolari, il quale ha una dimensione limitata e crea un *overhead* più basso.

Struttura dati coda basata su vettore circolare in pseudocodice

```
ITEM[] A                                  // elementi
int n                                     // dimensione attuale
int testa                                 // testa
int m                                     // dimesione massima

// crea una cosa vuota
QUEUE Queue(int dim)
    QUEUE t = new QUEUE
    t.A = new int[0...dim-1]
    t.m = dim
    t.testa = 0
    t.n = 0
    return t

// legge l'elemento in testa alla coda
ITEM top
    precondition: n > 0
    return A[testa]

// restituisce true de la coda è vuota
ITEM isEmpty
    return n == 0

// estrae l'elemento in testa alla coda e lo
   restituisce al chiamante
ITEM dequeue
    precondition: n > 0
    ITEM t = A[testa]
    testa = (testa + 1) mod m
    n++
    return t

// inserisce v in fondo alla coda
ITEM enqueue
    precondition: n < m
    A[(testa + n) mod m] = v
    n++
```

Codice 5.12: Coda basata su vettore in Java

```
public class VectorQueue implements Queue {

    /** Element vector */
    private Object[] A;

    /** Current number of elements in the queue */
    private int n;

    /** Top element of the queue */
    private int head;

    public VectorQueue(int dim) {
        n = 0;
        head = 0;
        A = new Object[dim];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public Object top() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        return A[head];
    }

    public Object dequeue() {
        if (n == 0)
            throw new IllegalStateException("Queue is empty");

        Object t = A[head];
        head = (head+1) % A.length;
        n = n-1;
        return t;
    }

    public void enqueue(Object v) {
        if (n == A.length)
            throw new IllegalStateException("Queue is full");

        A[(head+n) % A.length] = v;
        n = n+1;
    }
}
```

Capitolo 6

Alberi

6.1 Definizioni

Definizione 6.1.1 (albero radicato, *rooted tree*)**.** Un albero consiste di un insieme di nodi e un insieme di archi orientati che connettono coppie di nodi, con le seguenti proprietà:

- un nodo dell'albero è designato come nodo radice;
- ogni nodo n, a parte la radice, ha esattamente un arco entrante;
- esiste un cammino unico dalla radice ad ogni nodo;
- l'albero è connesso.

Definizione 6.1.2 (albero radicato, definizione ricorsiva)**.** Un albero è dato da:

- un insieme vuoto, oppure
- una radice e zero o più sottoalberi, ognuno dei quali è albero; la radice è connessa alla radice di ogni sottoalbero con un arco orientato.

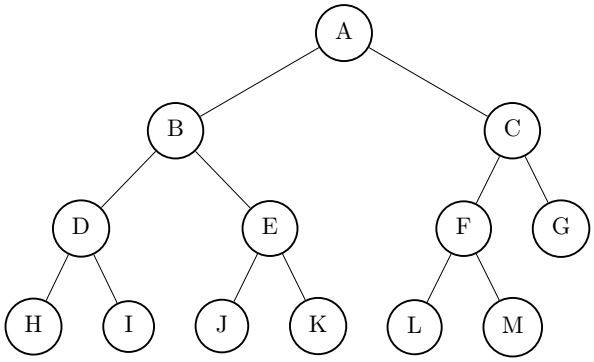
Definizione 6.1.3 (profondità, *depth*)**.** La lunghezza del cammino semplice dalla radice al nodo (misurato in archi).

Definizione 6.1.4 (livello, *level*)**.** L'insieme dei nodi alla stessa profondità.

Definizione 6.1.5 (altezza dell'albero, *height*)**.** La profondità massima delle sue foglie.

6.2 Terminologia

- A è la radice (*root*);
- B, C sono radici dei sottoalberi (*roots of their subtrees*);
- D, E sono fratelli (*siblings*);
- D, E sono figli (*children*) di B;
- B è il padre (*parent*) di D, E;
- H, I, J, K, L, M, G sono foglie (*leafs*);
- gli altri nodi sono nodi interni (*internal nodes*);
- E è lo zio (il fratello del padre) di I;
- B è il nonno di I, I è il nipote di B.



6.3 Alberi binari

Definizione 6.3.1 (Albero binario)**.** Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, che vengono identificati come figlio sinistro e figlio destro.

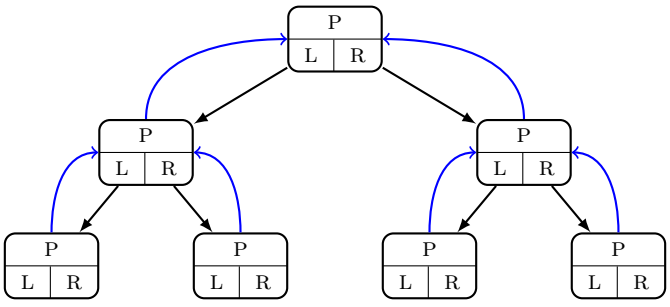
Nota. Due alberi T e U che hanno gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, sono distinti qualora un nodo u sia designato come figlio sinistro di un nodo v in T come figlio destro del medesimo nodo in U. In altre parole, anche se due alberi hanno lo stesso numero di nodi ed ognuno di questi nodi ha lo stesso numero di figli non è che detto che l'albero risultante sia identico.

Specifica albero binario

```
// GESTIONE ALBERO
Tree(ITEM v)  // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read  // legge il valore memorizzato nel nodo
write(ITEM v)  // modifica il valore memorizzato nel nodo
TREE parent  // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA

// restituiscono il figlio sinistro (destro) di questo nodo,
// restituisce nil se assente
TREE left
TREE right

// inserisce il sottoalbero radicato in t
// come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
insertRight(TREE t)

// distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo
deleteLeft
deleteRight
```

Nota. Le funzioni *senza parametri* sono indicate con un carattere senza grazie e privi di parentesi tonde vuote al fine di alleggerire la lettura del codice.

6.3.1 Memorizzazione di un albero binario



Vengono memorizzati i seguenti campi:

- *parent*: riferimento al nodo padre;
- *left*: riferimento al figlio sinistro;
- *right*: riferimento al figlio destro.

Uno qualunque di questi oggetti potrebbe essere pari a **nil**, stando ad indicare che non esiste nessun sottoalbero.

6.3.2 Implementazione

Algoritmo 6.3.1: Implementazione albero binario in pseudocodice

```
// crea un nuovo albero
// restituisce la radice dell'albero creato
TREE Tree(ITEM v)
    TREE t = new TREE
    t.parent ← nil
    t.left ← t.right ← nil
    t.value ← v
    return t
insertLeft(TREE t)
    if left ≠ nil then
        t.parent ← this
        left ← t
insertRight(TREE t)
    if right ≠ nil then
        t.parent ← this
        right ← t
// elimina ricorsivamente il sottoalbero sinistro
deleteLeft()
    if left ≠ nil then
        left.deleteLeft
        left.deleteRight
        left ← nil
// elimina ricorsivamente il sottoalbero destro
deleteRight()
    if right ≠ nil then
        right.deleteLeft
        right.deleteRight
        right ← nil
```

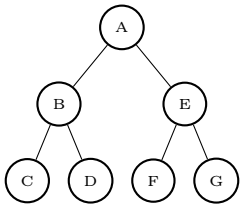
6.3.3 Visite

La visita di un albero (o la ricerca) è una strategia per passare attraverso (visitare) tutti i nodi di un albero. Si possono distinguere due tipi di visite:

1. visita in profondità: chiamata anche *Depth-First Search* (DFS), per visitare un albero visita ricorsivamente ognuno dei suoi sottoalberi; esistono tre varianti in base a quando il nodo viene visitato (pre, in o post-ordine); questa particolare visita sfrutta implicitamente il meccanismo di una pila (*stack*) tramite le chiamate ricorsive effettuate;
2. visita in ampiezza: chiamata anche *Breadth First Search* (BFS), per visitare un albero visita ogni livello, uno dopo l'altro partendo dalla radice; richiede esplicitamente l'utilizzo di una coda (*queue*).

Algoritmo 6.3.2: Schema per visita in profondità

```
dfs-schema(TREE t)
    if t ≠ nil then
        // pre-order visit
        stampa t
        dfs(t.left)
        // in-order visit
        stampa t
        dfs(t.right)
        // post-order visit
        stampa t
```



pre-visita	A B C D E F G
in-visita	C B D A F E G
post-visita	C D B F G E A

A seconda di dove scrivo il codice in questo schema ottengo una visita diversa.

6.3.4 Applicazioni

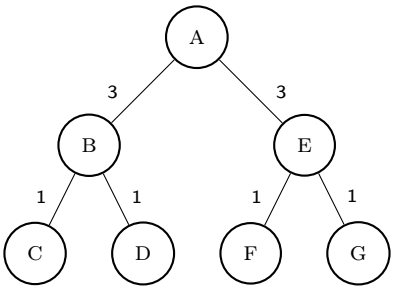
In genere post-visita e in-visita sono quelle più applicate, la pre-visita meno.

Visita in post-ordine

Una possibile applicazione della visita post-ordine è quella di effettuare un conteggio dei nodi presenti nell'albero.

Algoritmo 6.3.3: Conteggio dei nodi in un albero

```
count(TREE t)
    if t == nil then
        // è un albero vuoto
        return 0
    else
        // conto ricorsivamente i nodi
        C_ℓ = count(t.left)
        C_r = count(t.right)
        return C_ℓ + C_r + 1
```

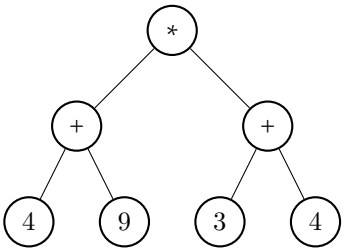


Visita in ordine (in-visita)

Una possibile applicazione della visita post-ordine è quella di stampare espressioni con operatori binari.

Algoritmo 6.3.4: Stampa espressioni con operatori binari

```
int stampaEspressioni(TREE t)
    if t.left == nil and t.right == nil then
        // siamo in una foglia
        stampa t.read
    else
        // sono su un nodo interno
        stampa "("
        stampaEspressioni(t.left)
        stampa t.read
        stampaEspressioni(t.right)
        stampa ")"
```



Stampa: `(( 4 + 9 ) * ( 3 + 4 ))`

Complessità di una visita

Il costo di una visita di un albero contenente n nodi è $\Theta(n)$, in quanto ogni nodo viene visitato al massimo una volta.

6.4 Alberi generici

Algoritmo 6.4.1: Specifica albero generico

```
// GESTIONE ALBERO
Tree(ITEM v) // costruisce un nuovo nodo, contenente v, senza figli o genitori
ITEM read // legge il valore memorizzato nel nodo
write(ITEM v) // modifica il valore memorizzato nel nodo
TREE parent // restituisce il padre, oppure nil se questo nodo è radice

// GESTIONE STRUTTURA
// restituiscono il primo figlio,
// oppure nil se questo nodo è una foglia
TREE leftmostChild

// restituisce il prossimo fratello,
// oppure nil se assente
TREE rightSibling

// inserisce il sottoalbero t
// come primo figlio di questo nodo
insertChild(TREE t)

// inserisce il sottoalbero t
// come prossimo fratello di questo nodo
insertSibling(TREE t)

// distuggi l'albero radicato
// identificato dal primo fratello
deleteChild

// distuggi l'albero radicato
// identificato dal primo figlio
deleteSibling
```

6.4.1 Visita in profondità

Un albero binario è anche un albero generale e lo visitiamo esattamente come lo visitavamo prima.

Algoritmo 6.4.2: Visita in profondità

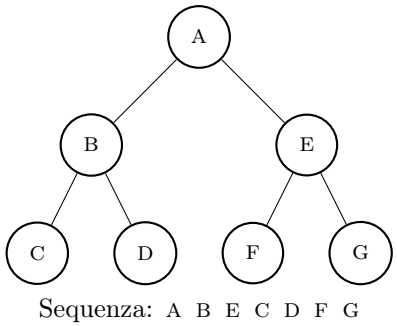
```
dfs(TREE t)
    if t ≠ nil then
        // pre-order visit
        stampa t
        dfs(t.left())
        // effettuo visita
        TREE u ← t.leftmostChild
        while u ≠ nil do
            dfs(u)
            u.rightSibling
        // post-order visit
        stampa t
```

6.4.2 Visita in ampiezza

Mentre nella visita in profondità il meccanismo della pila (*stack*) era implicito nelle chiamate ricorsive, in questo caso è necessario utilizzare *esplicitamente* una coda (*queue*). Un'altra differenza fra i due algoritmi è che quello in profondità è un algoritmo ricorsivo, l'altro è iterativo. Quando tutti i nodi di un livello vengono estratti dalla coda, la coda contiene solo ed unicamente i nodi del livello successivo.

Algoritmo 6.4.3: Visita in ampiezza

```
bfs(TREE t)
    QUEUE Q ← Queue
    Q.enqueue(t)  // inserisci la radice
    while not Q.isEmpty do
        // fintanto che la coda non è vuota
        // estraggo un nodo dalla coda
        TREE u ← Q.dequeue
        // visita per livelli del nodo u
        stampa u
        // fintanto che ho almeno un figlio
        u ← u.leftmostChild
        while u ≠ nil do
            // metto in coda il figlio
            Q.enqueue(u)
            // passo al figlio destro
            u ← u.rightSibling
```



Commento Mettiamo in coda tutti i nodi che vogliamo visitare passo passo. Qui la stampa è in pre-visita ma qui – a differenza dei grafi – non ha molta importanza se la visita la facciamo prima o dopo. Visito tutti i figli prima di passare al livello successivo.

6.5 Memorizzazione

Esistono diversi modi per memorizzare un albero, più o meno indicati a seconda del numero massimo e medio di figli presenti. Le realizzazioni possibili sono:

1. con vettore dei figli;
2. primo figlio, prossimo fratello;
3. con vettore dei padri

6.5.1 Realizzazione con vettore dei figli

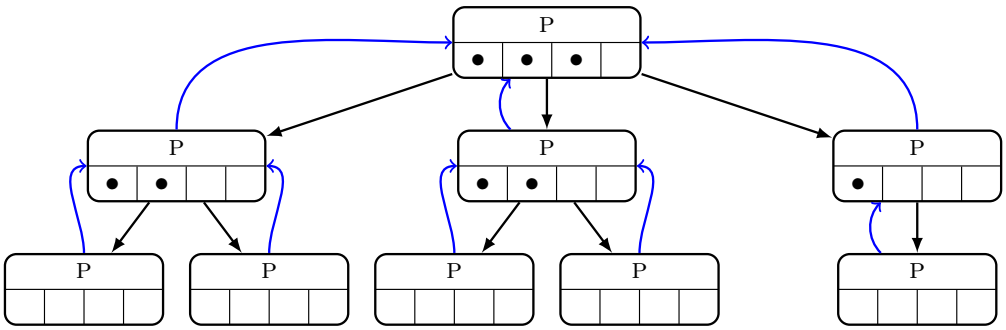


Figura 6.1: Realizzazione con vettore dei figli

Vengono memorizzati i seguenti campi:

- *parent* che è il riferimento al nodo padre;

- vettore dei figli il quale a seconda del numero dei figli può comportare una discreta quantità di spazio sprecato.

6.5.2 Realizzazione basata su primo figlio, prossimo fratello

Viene implementato come una lista di fratelli.

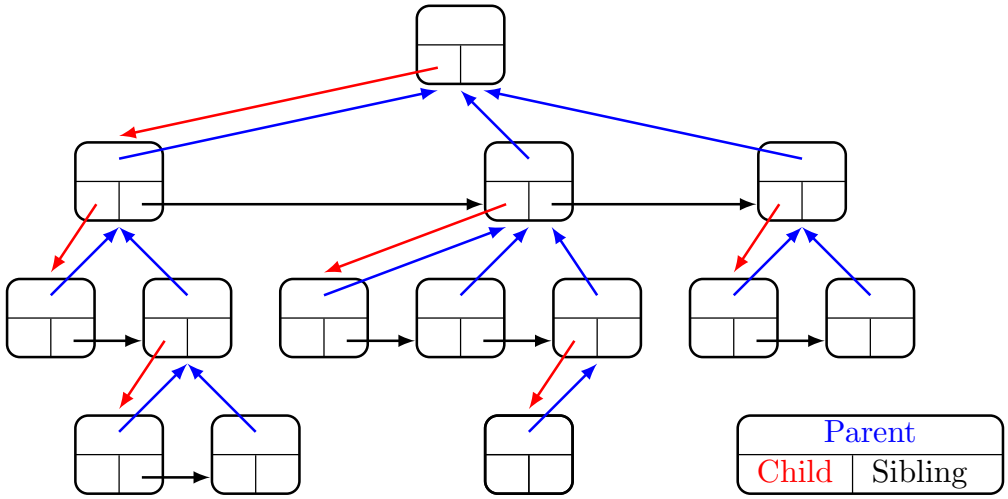


Figura 6.2: Realizzazione basata su primo figlio, prossimo fratello

La memorizzazione che viene utilizzata nel *file system* è esattamente questa.

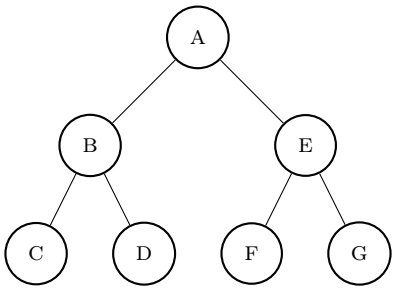
Algoritmo 6.5.1: Implementazione albero "primo figlio, prossimo fratello" in pseudocodice

```
TREE parent                                   // Riferimento al padre
TREE child                                    // Riferimento al primo figlio
TREE sibling                                  // Riferimento al prossimo fratello
ITEM value                                    // Valore memorizzato nel nodo

TREE Tree(ITEM v)
    TREE t = new TREE
    t.value ← v
    t.parent ← t.child ← t.sibling ← nil
    return t

insertChild(TREE t)
    t.parent ← self
    // inserisci t prima dell'attuale primo figlio
    t.sibling ← child
    child ← t

insertSibling(TREE t)
    t.parent ← parent
    // inserisci t prima dell'attuale prossimo
       fratello
    t.sibling ← sibling
    sibling ← t

deleteChild()
    TREE newChild ← child.rightSibling
    delete(child)
    child ← newChild

deleteSibling()
    TREE newBrother ← sibling.rightSibling
    delete(sibling)
    sibling ← newBrother

// metodo ausiliare
delete(TREE t)
    TREE u ← t.leftmostChild
    while u ≠ nil do
        TREE next ← u.rightSibling
        delete(u)
        u ← next
```

6.5.3 Realizzazione con vettore dei padri

Nella realizzazione con vettore dei padri, l'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre del vettore.

1	A	0
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3



Questa realizzazione può sembrare particolarmente assurda poiché dato un nodo non permette di stabilire direttamente quali sono i suoi figli, ma ci sono molti algoritmi che sono interessati solo ai padri. Questa è la rappresentazione più compatta che possiamo creare, vedremo la sua utilità quando andremo a studiare le visite sui grafi.

Capitolo 7

Alberi Binari di Ricerca

7.1 Introduzione

Facciamo un breve ripasso della struttura dati dizionario. La struttura dati dizionario è un insieme dinamico che implementa le seguenti funzionalità:

- ITEM lookup(ITEM v) permette di cercare per una certa chiave;
- insert(ITEM k, ITEM v) permette di associare una chiave ad un valore;
- remove(ITEM k) permette di rimuovere una certa associazione chiave-valore.

Tabella 7.1: Possibili implementazioni della struttura dati dizionario e relative complessità

Struttura dati	lookup	insert	remove
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Vettore non ordinato	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$

* assumendo che l'elemento sia già stato trovato, $\mathcal{O}(n)$ altrimenti.

Ora vedremo la struttura dati dizionario implementata come un albero binario di ricerca.

L'idea che ha portato allo sviluppo degli alberi binari di ricerca è stata quella di portare la ricerca binaria (o dicotomica) negli alberi, avendo quindi un meccanismo dinamico per la memorizzazione delle informazioni ma basandosi sul meccanismo della ricerca binaria per recuperarle.

Le associazioni chiave-valore vengono memorizzate in un albero binario. Ogni nodo contiene una coppia ($u.key$, $u.value$). Le chiavi devono appartenere ad un insieme *totalmente ordinato*, ossia dev'essere possibile stabilire, date due chiavi, una relazione di precendenza fra di loro.

Proprietà. *Le seguenti proprietà permettono di realizzare un algoritmo di ricerca dicotomica:*

1. *Le chiavi contenute nei nodi del sottoalbero sinistro di u sono minori di $u.key$;*
2. *Le chiavi contenute nei nodi del sottoalbero destro di u sono maggiori di $u.key$.*

Nota. Queste proprietà valgono per ogni nodo e riguardano l'intero sottoalbero.

Vedremo un algoritmo per verificare se un albero binario è un albero binario di ricerca più avanti (verifyABR), il quale controllerà se queste proprietà sono soddisfatte.

Specifica Alberi Binari di Ricerca (A B R)

```
// CONTENUTO DI UN NODO
TREE parent
TREE left
TREE right
ITEM key
ITEM value

// GETTERS
TREE parent
TREE left
TREE right
ITEM key
ITEM value

// ORDINAMENTO
TREE successorNode(TREE t)
TREE predecessorNode(TREE t)
TREE min
TREE max

// FUNZIONI DIZIONARIO
ITEM lookup(ITEM k)
insert(ITEM k, ITEM v)
remove(ITEM k)

// FUNZIONI INTERNE
ITEM lookupNode
insertNode(TREE T, ITEM k, ITEM v)
removeNode(TREE T, ITEM k)
```

Ricerca di un nodo

Algoritmo 7.1.1: Ricerca di un nodo in un dizionario realizzato tramite A B R

```
int lookup(ITEM k)
    TREE t ← lookupNode(tree, k)
    if t ≠ nil then
        return t.value
    else
        return nil

// RICERCA DI UN NODO, iterativa
TREE lookupNode(TREE T, ITEM k)
    TREE u ← T  // parto dalla radice
    while u ≠ nil and u.key ≠ k do
        u ← iif(k < u.key, u.left, u.right)

// RICERCA DI UN NODO, ricorsiva
TREE lookupNode(TREE T, ITEM k)
    if T == nil or T.key == k then
        return T
    else
        return lookupNode(iif(k < u.key, u.left, u.right), k)
```

Ricerca del minimo e del massimo

Algoritmo 7.1.2: Ricerca del minimo e del massimo in un dizionario realizzato tramite ABR

```
// RICERCA DEL MINIMO
TREE min(TREE T)
    TREE u = T  // parto dalla radice
    while u.left ≠ nil do
        u ← u.left
    return u
```

```
// RICERCA DEL MASSIMO
TREE max(TREE T)
    TREE u = T  // parto dalla radice
    while u.right ≠ nil do
        u ← u.right
    return u
```

Ricerca del predecessore, successore

Algoritmo 7.1.3: Ricerca del predecessore e del successore di un nodo in un dizionario realizzato tramite ABR

```
// RICERCA DEL PREDECESSORE
TREE predecessorNode(TREE t)
    if t == nil then
        return t
    if t.left ≠ nil then
(1)     return max(t.left)
    else
(2)     TREE p ← t.parent
        while p ≠ nil and t == p.left do
            t ← p  // padre
            p ← p.parent  // nonno
    return p
```

```
// RICERCA DEL SUCCESSORE
TREE successorNode(TREE t)
    if t == nil then
        return t
    if t.right ≠ nil then
(3)     return min(t.right)
    else
(4)     TREE p ← t.parent
        while p ≠ nil and t == p.right do
            t ← p
            p ← p.parent
    return p
```

(1) u ha figlio sinistro: il predecessore è il massimo del sottoalbero sinistro di u;

(2) u non ha figlio sinistro: risalendo attraverso i padri, il predecessore è il primo avo v tale per cui u sta nel sottoalbero destro di v;

(3) u ha figlio destro: il successore è il minimo del sottoalbero destro di u;

(4) u non ha figlio destro: risalendo attraverso i padri, il sucessore è il primo avo v tale per cui u sta nel sottoalbero sinistro di v.

Nota. Posso trovare **nil** se passo alla funzione successorNode il nodo massimo o alla funzione predecessorNode il minimo (usciranno dal ciclo restituendo p che sarà pari a **nil**).

Inserimento di un nodo

La funzione insertNode inserisce un'associazione chiave-valore (k, v) nell'albero T. Se la chiave è già presente, sostituisce il valore associato; altrimenti, viene inserita una nuova associazione. Se l'albero è vuoto ($T == \textbf{nil}$) restituisce il primo nodo dell'albero, altrimenti restituisce la radice di T inalterata.

La funzione ausiliaria link si occupa di inserire il nodo collegandolo al corretto genitore.

Algoritmo 7.1.4: Inserimento di un nodo in un DICTIONARY realizzato tramite ABR

```
// IMPLEMENTAZIONE DIZIONARIO
insert(ITEM k, ITEM v)
    tree ← insertNode(tree, k, v)

// INSERIMENTO DI UN NODO
TREE insertNode(TREE T, ITEM k, ITEM v)
    TREE p ← nil  // padre
    TREE u ← T  // parto dalla radice

    // cerco posizione inserimento
    while u ≠ nil and u.key ≠ k do
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    if u ≠ nil and u.key == k then
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    else
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)

        if p == nil then
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    return T

// collega un nodo padre p ad un nodo figlio u
link(TREE p, TREE u, ITEM x)
    if u ≠ nil then
        // il nodo è stato cancellato
        u.parent ← p  // registro il padre

    if p ≠ nil then
        // collego il nodo sul figlio corretto
        u ← iif(x < p.key, p.left, p.right)
```

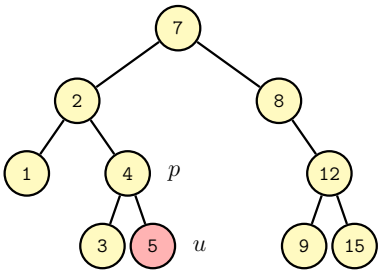
Rimozione di un nodo

Rimuove il nodo contenente la chiave k dall'albero T, restituisce la radice dell'albero (potenzialmente cambiata).

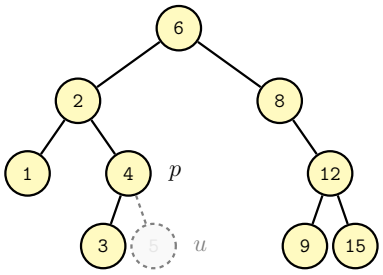
Algoritmo 7.1.5: Rimozione di un nodo in un DICTIONARY realizzato tramite ABR

```
// IMPLEMENTAZIONE DIZIONARIO
remove(ITEM k)
    TREE tree ← removeNode(tree, k)

// RIMOZIONE DI UN NODO
TREE removeNode(TREE T, ITEM k)
    // individuo il nodo da rimuovere
    TREE u ← lookupNode(T, k)

    // se il nodo da rimuovere è presente nell'albero...
    if u ≠ nil then
(1)     // ...e non ha figli
        if u.left == nil and u.right == nil then
            if u.parent ≠ nil then // se esiste il padre
                link(u.parent, nil, k) // rimuovo il puntatore al figlio

            // rimuovo direttamente il nodo
            delete u
(3)     // ...ed ha due figli
        if u.left ≠ nil and u.right ≠ nil then
            TREE s ← successorNode // individuo il successore

            link(s.parent, s.right, s.key) // collego il sottoalbero destro

            // copio il successore
            // nella posizione del nodo rimosso
            u.key ← s.key
            u.value ← s.value

            // rimuovo il successore
            delete s
(2)     // ...ed ha un solo figlio (sinistro)
        if u.left ≠ nil and u.right == nil then
            link(u.parent, u.left, k) // collega il figlio al padre

            if u.parent == nil then // se il padre non esiste
                T == u.right // il figlio diventa la radice
        // ...ed ha un solo figlio (sinistro)
        else
            link(u.parent, u.right, k) // collega il figlio al padre

            if u.parent == nil then // se il padre non esiste
                T == u.right // il figlio diventa la radice

    // restituisco la radice
    return T
```

(1) se il nodo da eliminare u non ha figli: lo si elimina semplicemente, in quanto togliere una foglia non altera le proprietà di ordinamento dell'albero;

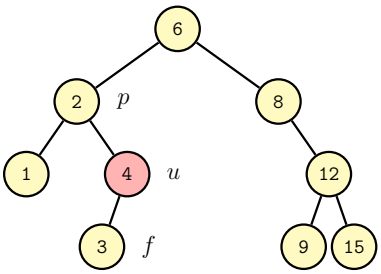


(a) Individuazione nodo foglia

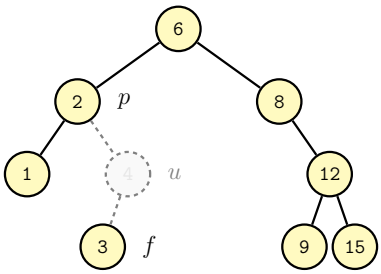


(b) Rimozione del nodo foglia

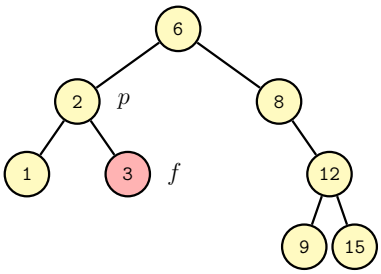
(2) se il nodo da eliminare ha un solo figlio f (destro o sinistro): si elimina u e si collega f all'ex-padre p di u in sostituzione di u (tramite la funzione link); le proprietà di ordinamento non vengono alterate in quanto tutti i nodi del sottoalbero destro di p sono maggiori di p stesso;



(a) Individuazione nodo u da eliminare

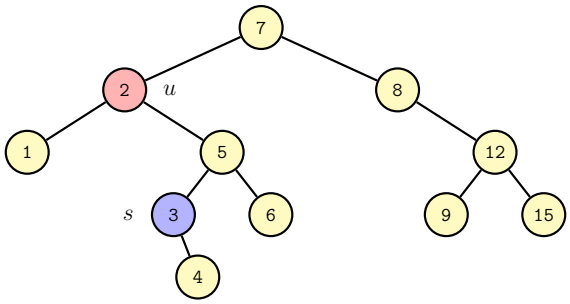


(b) Rimozione nodo u

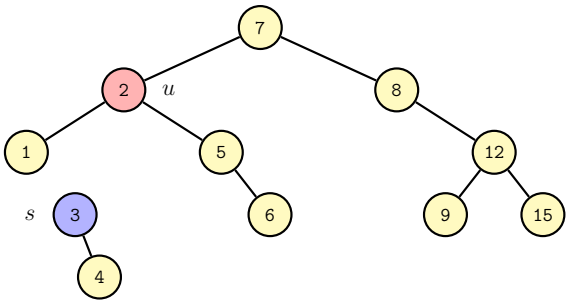


(c) Collegamento del sottoalbero f di u al padre p di u

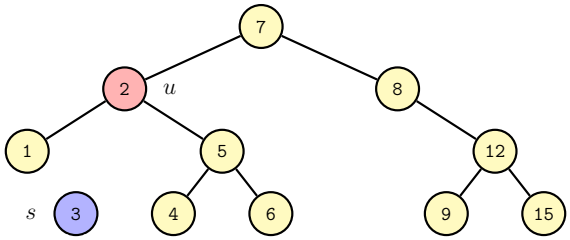
(3) se il nodo da eliminare u ha due figli: cerchiamo di ricadere nel caso (2); (a) individuiamo il successore (predecessore) s di u, il quale è il più piccolo valore maggiore di u (il più grande valore minore di u) e di conseguenza non ha figli sinistri (non ha figli destri); (b) si "stacca" il successore s; (c) si collega l'eventuale figlio destro di s al padre (tramite la funzione link) in quanto trovandosi nel sottoalbero sinistro del nonno vuol dire che sicuramente il suo valore non è maggiore del padre di s; (d) si copia s su u, si rimuove il nodo s, così facendo rispetto comunque l'ordine parziale.



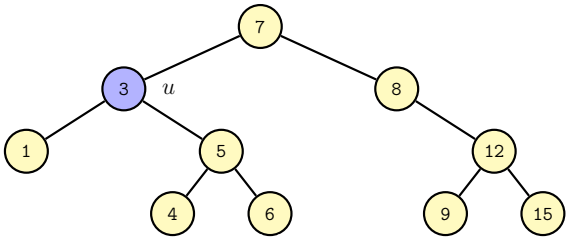
(a) Individuiamo il successore s di u



(b) Si "stacca" il successore s



(c) Si collega l'eventuale figlio destro di s al padre



(d) Si copia s su u, si rimuove il nodo s

Costo computazionale delle operazioni

Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia. Quindi se l'altezza dell'albero è definita come h, il tempo di ricerca ha complessità $\mathcal{O}(h)$.

Il caso pessimo è rappresentanto da un albero sbilanciato completamente a destra o completamente a sinistra. Questo caso può accadere quando si inseriscono ordinatamente i dati nell'albero. Questo caso, dove l'altezza $h = n$ porta ad una complessità $\mathcal{O}(n)$.

Mentre il caso ottimo è rappresentanto da un albero perfettamente bilanciato. Nell'esempio è mostrato un albero perfetto con $2^h - 1$ nodi, dove h è l'altezza. In questo caso la complessità è pari a $\mathcal{O}(\log n)$, ad esempio con $h = 2^3 - 1$ la complessità è $\mathcal{O}(\log h) = \mathcal{O}(\log 7) < 3$.

Ci domandiamo quindi quale sia l'altezza media di un albero binario di ricerca. Il caso "semplice" è quello di considerare che gli inserimenti avvengano in maniera statisticamente uniforme, è possibile dimostrare che l'altezza media è $\mathcal{O}(\log n)$, mentre il caso generale, ossia quello in cui avvengono sia inserimenti che cancellazioni è di difficile trattazione. Per evitare questa casistica si utilizzano varie tecniche per mantenere l'albero bilanciato. Per capire queste tecniche abbiamo prima bisogno di fissare un concetto.

Definizione 7.1.1 (Fattore di bilanciamento)**.** Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v.

Negli anni sono state usate diverse tecniche, ora in disuso:

- Alberi AVL (1962): $\beta(v) \leqslant 1$ per ogni nodo v, il bilanciamento dell'albero avveniva tramite rotazioni;
- B-Alberi (1972): $\beta(v) = 0$ per ogni nodo v, sono specializzati per strutture in memoria secondaria;
- Alberi 2-3 (1983): $\beta(v) = 0$ per ogni nodo v, in cui ogni nodo può avere o 2 o 3 figli, se ad un nodo viene aggiunto un ulteriore figlio, il ramo viene spezzato in due rami con 2 figli ciascuno, mentre se ad un ramo con 2 figli ne viene tolto uno allora l'unico figlio rimanente viene collegato al padre, questo potrebbe riportare il problema al primo caso; il bilanciamento viene ottenuto quindi tramite merge/split, il grado è variabile.

Nota (Meccanismo di rotazione)*.* Il meccanismo di rotazione ci permette di abbassare il fattore di sbilanciamento rispettando le proprietà di ordinamento parziale.

7.2 Alberi Binari di Ricerca bilanciati

Definizione 7.2.1 (Albero Red-Black)**.** Un albero red-black è un albero binario di ricerca in cui:

- ogni nodo è colorato di rosso o di nero;
- le chiavi vengono mantenute solo nei nodi interni dell'albero;
- le foglie sono costituite solo da nodi speciali **Nil**.

I nodi speciali **Nil** sono dei nodi sentinella il cui unico scopo è quello di evitare di trattare diversamente i puntatori ai nodi, dai puntatori **nil**; infatti al posto di un puntatore **nil** si usa un puntatore ad un nodo **Nil**; in memoria ne esiste solo uno per motivi di economia. I nodi con figli **Nil** sono le foglie nell'albero binario di ricerca corrispondente.

Un albero red-black deve rispettare i seguenti vincoli:

1. la radice è nera;
2. tutte le foglie sono nere;
3. entrambi i figli di un nodo rosso sono neri;
4. ogni cammino semplice da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u ha lo stesso numero di nodi neri.

Algoritmo 7.2.1: Specifica RED-BLACK TREE

```
// CONTENUTO DI UN NODO
TREE parent
TREE left
TREE right
int color  // RED o BLACK
ITEM key
ITEM value
// GETTERS
TREE parent
TREE left
TREE right
int color
ITEM key
ITEM value
```

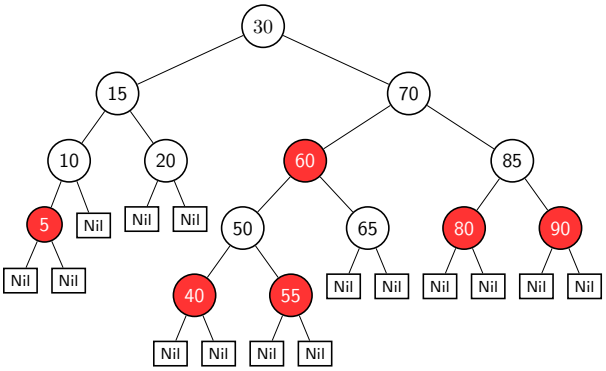
Proprietà (Altezza nera di un nodo v)**.** *L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v (escluso) ad ogni foglia (inclusa) del suo sottoalbero.*

Proprietà (Altezza nera di un albero Red-Black)**.** *L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice.*

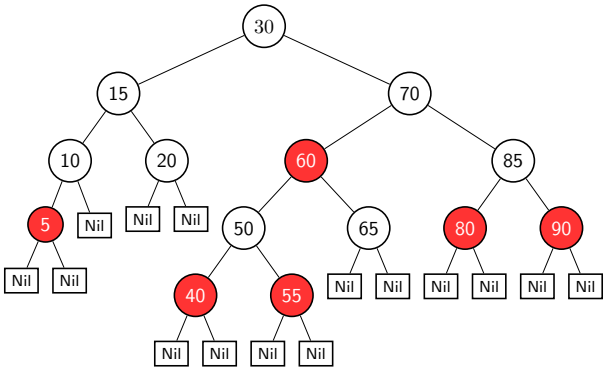
Entrambe le proprietà sono ben definite perché tutti i percorsi hanno lo stesso numero di nodi neri (per via della regola no. 4).

Esempi

Nelle suguenti figure i nodi neri sono segnati in bianco, mentre quelli rossi sono segnati.

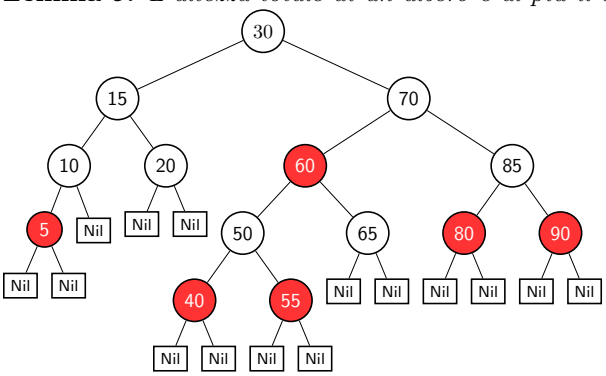


Es. 1 Entrambi i figli di un nodo rosso sono neri (3), ma un nodo nero può avere figli neri.

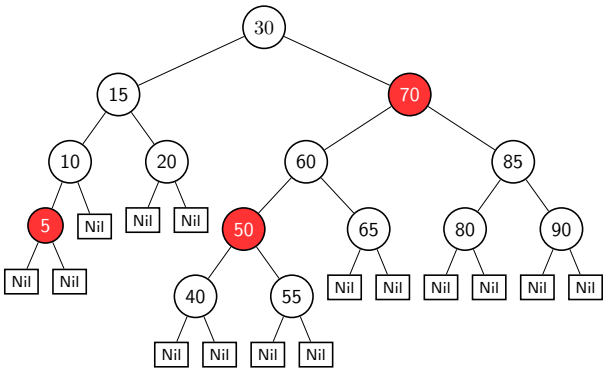


Es. 2 Ogni percorso da un nodo interno ad un nodo **Nil** ha lo stesso numero di nodi neri (4). L'altezza nera di quest'albero è 3

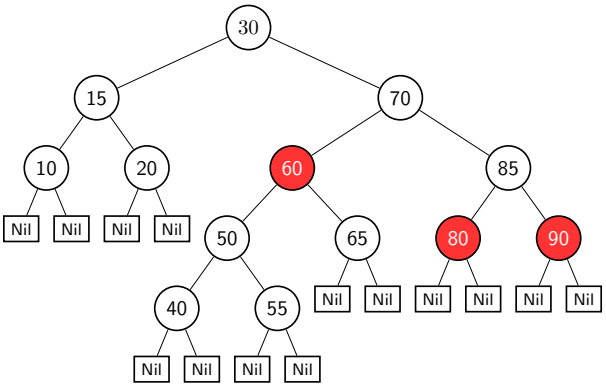
Lemma 9. *L'altezza totale di un albero è al più il doppio della sua altezza nera.*



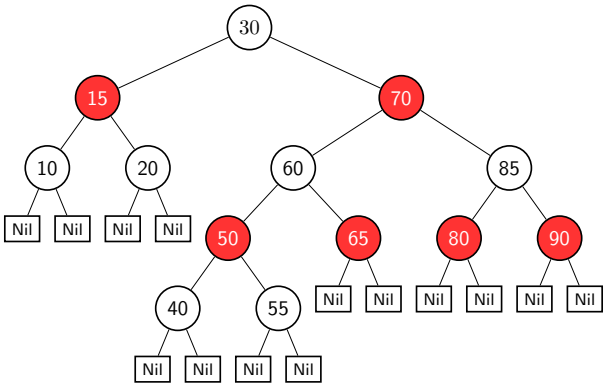
Es. 3 Più colorazioni sono possibili (versione 1). L'altezza di questo albero è 3



Es. 4 Più colorazioni sono possibili (versione 2). L'altezza di questo albero è 3



Es. 5 Cambiare colorazione può cambiare l'altezza nera. L'altezza di questo albero è 3



Es. 6 Cambiare colorazione può cambiare l'altezza nera. Stesso albero, l'altezza nera di questo albero è 2

7.2.1 Inserimento di un nodo

Durante la modifica di un albero Red-Black è possibile che le condizioni di bilanciamento risultino violate. Quando i vincoli Red-Black vengono violati si può agire in due modi:

- modificando i colori nella zona della violazione;
- operando dei bilanciamenti dell'albero tramite rotazioni (a destra o a sinistra)

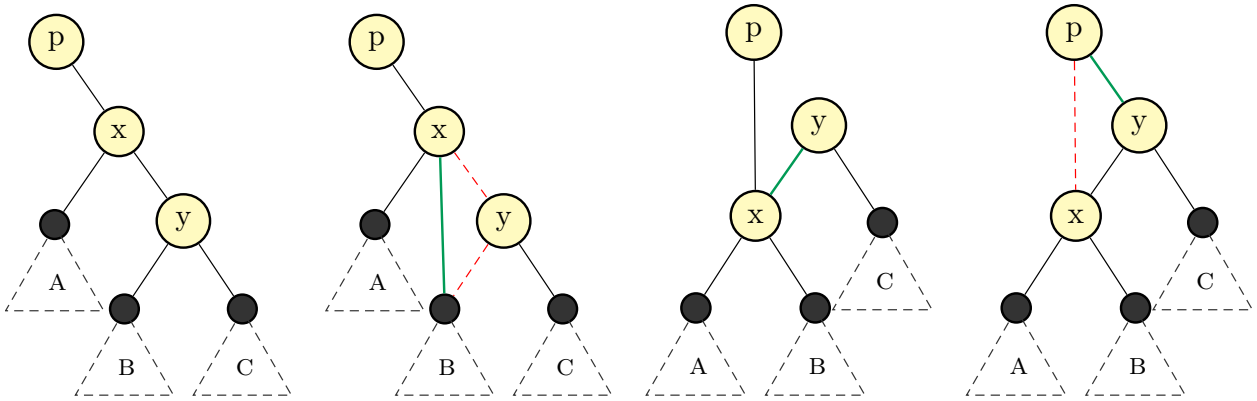
7.2.2 Bilanciamento dell'albero

Rotazione a sinistra

Algoritmo 7.2.2: Bilanciamento dell'albero tramite rotazione a sinistra

```
// effettua una rotazione verso sinistra
TREE rotateLeft(TREE x)
(1)   TREE y ← x.right
      TREE p ← x.parent
(2)   x.right ← y.left  // il sottoalbero B diventa figlio destro di x
      if y.left ≠ nil then
          y.left.parent ← x
(3)   y.left ← x  // x diventa figlio sinistro di y
      x.parent ← y
(4)   y.parent ← p  // y diventa figlio di p
      if p ≠ nil then
          if p.left == x then
              p.left ← y
          else
              p.right ← y
      return y
```

Nota. Il disegno differisce minimamente da quello che si trova sulle slide per motivi di comodità nel disegnarli con LaTeX



(1) Stato iniziale **(2)** Il sottoalbero B diventa figlio dx di x **(3)** x diventa figlio sx di y **(4)** y diventa figlio di p, il vecchio padre di x

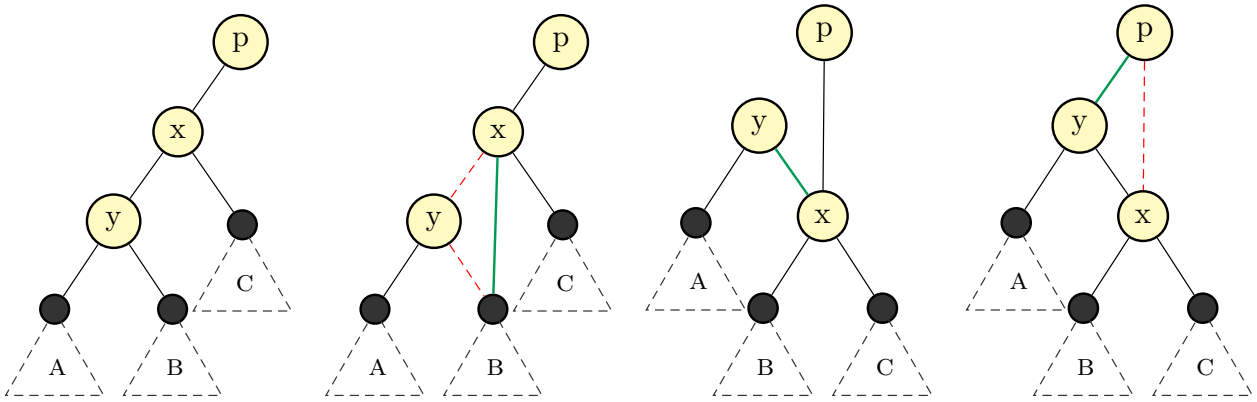
Figura 7.5: Esempio di rotazione a sinistra

Rotazione a destra

La rotazione a destra è simmetrica e viene spiegata ed illustrata per completezza.

Algoritmo 7.2.3: Bilanciamento dell'albero tramite rotazione a destra

```
// effettua una rotazione verso destra
TREE rotateRight(TREE x)
    // entrambi potrebbero essere nil
(1) TREE y ← x.left
    TREE p ← x.parent
(2) x.left ← y.right  // il sottoalbero B diventa figlio sinistro di x
    if y.right ≠ nil then
        y.right.parent ← x
(3) y.right ← x  // x diventa figlio destro di y
    x.parent ← y
(4) y.parent ← p  // y diventa figlio di p
    if p ≠ nil then
        if p.right == x then
            p.right ← y
        p.left ← y
    return y
```



(1) Stato iniziale

(2) Il sottoalbero B diventa figlio sx di x

(3) x diventa figlio dx di y

(4) y diventa figlio di p, il vecchio padre di x

Figura 7.6: Esempio di rotazione a destra

Inserimento di un nodo in un albero binario bilanciato

Per inserire un nodo in un albero Red-Black si usa la stessa procedura usata per gli alberi binari di ricerca e si colora il nuovo nodo di RED. Il vincolo che potremmo violare è il terzo, quello che prevede che entrambi i figli di un nodo rosso siano neri.

Algoritmo 7.2.4: Inserimento di un nodo in un RED-BLACK TREE

```
// Inserimento di un nodo in un albero Red-Black
TREE insertNode(TREE T, TREE k, ITEM x)
    TREE p ← nil  // riferimento al padre
    TREE u ← T  // riferimento alla radice

    // cerco posizione inserimento
    while u ≠ nil and u.key ≠ k do
        p ← u
        u ← iif(k < u.key, u.left, u.right)

    if u ≠ nil and u.key == k then
        // la chiave è già presente, aggiorno il valore
        u.value ← v
    else
        // la chiave non è presente
        // creo un nodo coppia chiave-valore
        TREE new ← Tree(k, v)

        // collego il nodo creato
        link(p, new, k)
        balanceInsert(new)

        if p == nil then
            T ← new  // primo nodo ad essere inserito

    // restituisco l'albero non modificato o il nuovo nodo
    return T
```

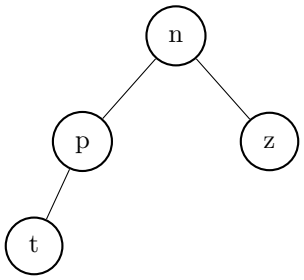
Nel caso in cui l'inserimento vìoli il terzo vincolo ci sposteremo verso l'alto lungo il percorso di inserimento; cercheremo di ripristinare il terzo vincolo; sposteremo le violazioni verso l'alto rispettando il quarto vincolo (mantenendo l'altezza nera dell'albero); al termine, coloreremo la radice di nero (onorando il primo vincolo).

Nota. Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi, altrimenti non sono necessarie.

Algoritmo 7.2.5: Bilanciamento dell'albero in seguito all'inserimento di un nodo rosso

```
balanceInsert(TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso
    // t == nil è la condizione di fino ciclo
    while t ≠ nil do
(0)     TREE p ← t.parent                                              // riferimento al padre
        TREE n ← iif(p ≠ nil, p.parent, nil)                           // riferimento al nonno
        TREE z ← iif(n == nil, nil, iif(n.left == p, n.right, n.left)) // riferimento allo zio
(1)     if p == nil then
            t.color ← BLACK
            t ← nil  // fine
(2)     else if p.color == BLACK then
            t ← nil  // fine
(3)     else if z.color == RED then
            p.color ← z.color ← BLACK
            n.color ← RED
            t ← n  // passo il problema al nonno
        else
(4a)        if (t == p.right) and (p == n.left) then
                rotateLeft(p)
                t ← p  // passo il problema al padre
(4b)        if (t == p.left) and (p == n.right) then
                rotateRight(p)
                t ← p  // passo il problema al padre
            else
(5a)            if (t == p.left) and (p == n.left) then
                    rotateRight(n)
(5b)            else if (t == p.right) and (p == n.right) then
                    rotateLeft(n)

                p.color ← BLACK
                n.color ← RED
                t ← nil  // fine
```

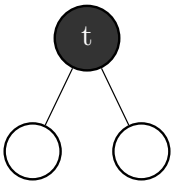
(0) dichiaro dei riferimento al padre, al nonno e allo zio



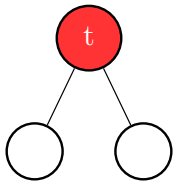
(1) il nuovo nodo t non ha padre; Questo può accadere in due casi:

- è il primo nodo ad essere inserito, oppure
- quando abbiamo spostato la violazione verso l'alto fino a raggiungere la radice

Ricoloriamo t di BLACK in quanto trovandosi sulla radice dell'albero non viola nessun vincolo.

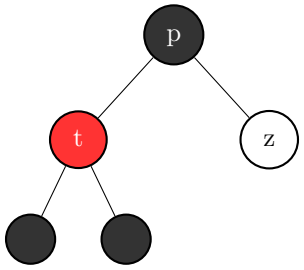


(a) Possibile violazione del primo vincolo

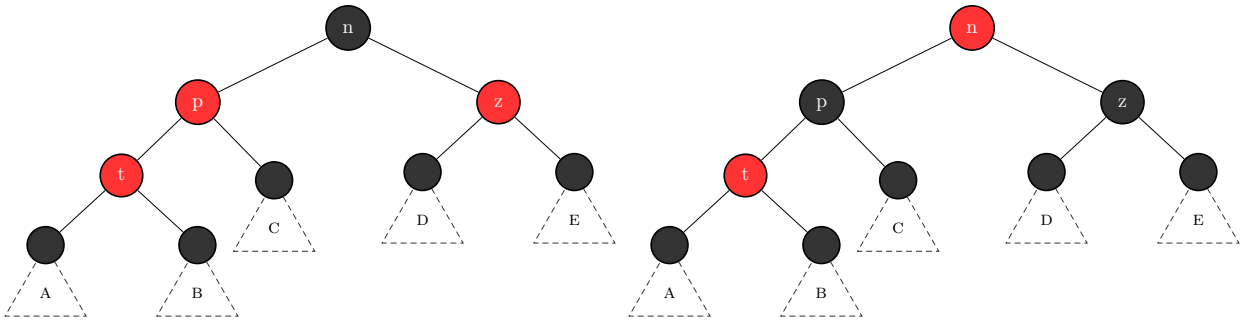


(b) Ricolorazione del nodo t

(2) il padre p di t è nero; anche in questo caso non abbiamo violato nessun vincolo perché avendo inserito un nodo rosso la lunghezza dei cammini neri non cambia e avendo inserito un nodo rosso figlio di un nodo nero non violiamo il terzo vincolo;



(3) il padre p e lo zio z sono rossi; Se z è rosso è possibile ricolorare di nero p e z, e di rosso n; poiché tutti i cammini che passano per z e p passano anche per n, l'altezza nera non è cambiata (non abbiamo violato il quarto vincolo); Abbiamo spostato così il problema verso l'alto, più precisamente sul nonno che potrebbe aver violato il primo o il terzo vincolo, ovvero che n può essere una radice rossa o che abbia un padre rosso. Per risolvere il problema poniamo $t \leftarrow n$ e continuiamo il ciclo.

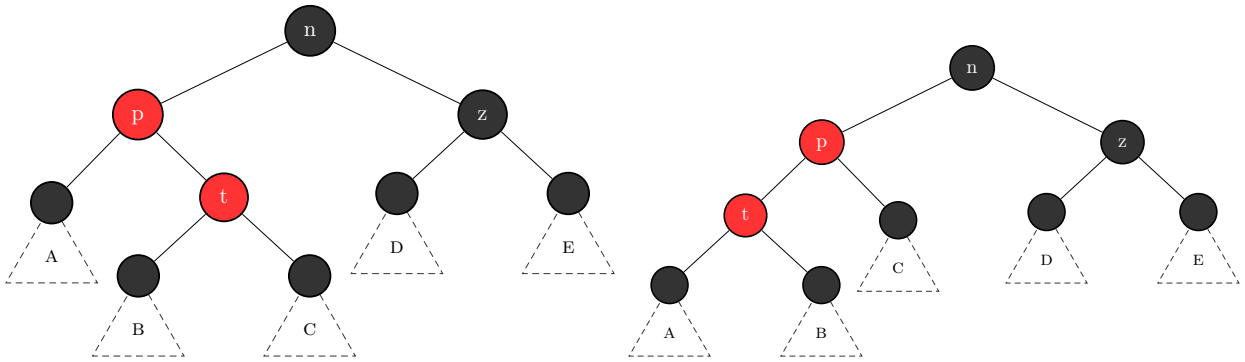


(a) Possibile violazione del primo e del terzo vincolo

(b) Ricolorazione del padre e dello zio di nero

`(4a)` il padre p è rosso e lo zio z è nero; si assuma che t sia figlio *destro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a sinistra a partire dal nodo p scambiamo i ruoli di t e d p ottenendo il caso `(5a)`, dove i nodi in conflitto sul terzo vincolo sono entrambi figli *sinistri* dei loro padri; i nodi coinvolti nel cambiamento sono p e t, entrambi rossi, quindi l'altezza nera non cambia; abbiamo spostato il problema al padre, quindi poniamo $t \leftarrow p$ e continuiamo il ciclo.

`(4b)` speculare al caso `(4a)` (ossia che che t sia figlio *sinistro* di p e che p sia figlio *destro* di n)



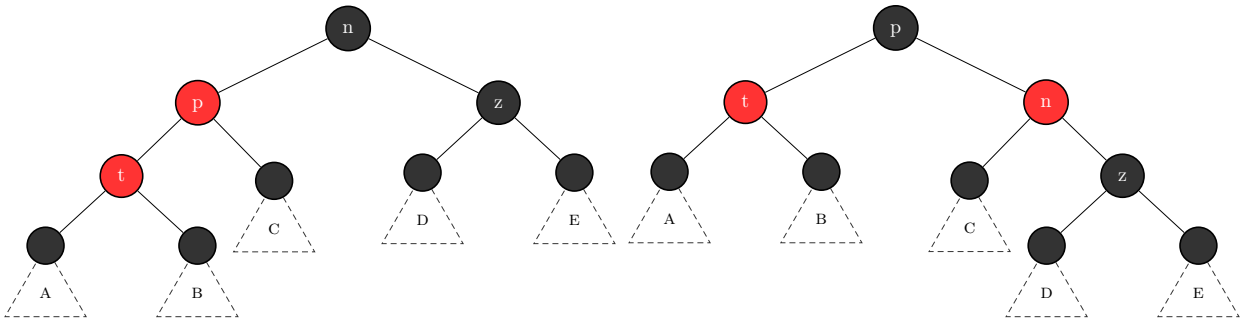
(a) Possibile violazione del primo e del terzo vincolo

(b) Ricolorazione del padre e dello zio di nero

Figura 7.9: Rappresentazione grafica del caso `(4a)`

`(5a)` anche in questo caso il padre p è rosso e lo zio z è nero; ma si assuma che t sia figlio *sinistro* di p e che p sia figlio *sinistro* di n; effettuando una rotazione a destra a partire dal nodo n ci porta ad una situazione in cui t e n sono figli di p; colorando p di nero ed n di rosso ci troviamo in una situazione in cui tutti i vincoli vengono rispettati (in particolare, l'altezza nera che passano per la radice è uguale a quella iniziale).

`(5b)` speculare al caso `(5a)` (ossia che che che t sia figlio *destro* di p e che p sia figlio *destro* di n)



(a) Possibile violazione del primo e del terzo vincolo

(b) Ricolorazione del padre e dello zio di nero

Figura 7.10: Rappresentazione grafica del caso `(5a)`

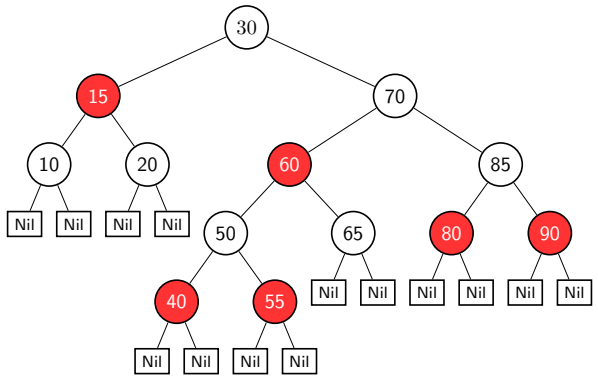
Complessità Ognuna di queste operazione avviene in tempo costante. Ogni volta il problema può salire di uno o due livelli, in quanto l'altezza dell'albero è limitata da $\log n$, il nostro algoritmo è limitato superiormente da $\log n$, ossia $\mathcal{O}(\log n)$.

Più precisamente:

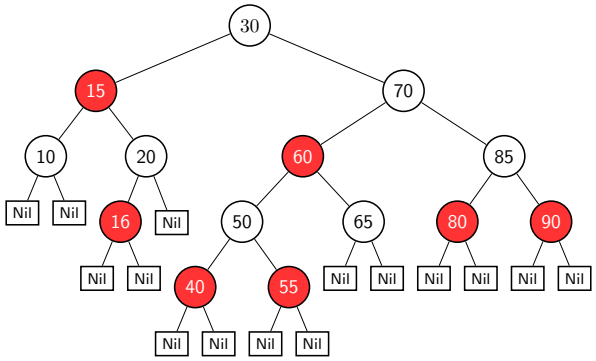
- $\mathcal{O}(\log n)$ per scendere fino al punto di inserimento;
- $\mathcal{O}(1)$ per effettuare l'inserimento;
- $\mathcal{O}(\log n)$ per risalire ed "aggiustare" (caso `3`)

Esempi di inserimento

Proviamo ad inserire il nodo 16 nell'albero red-black sottostante.

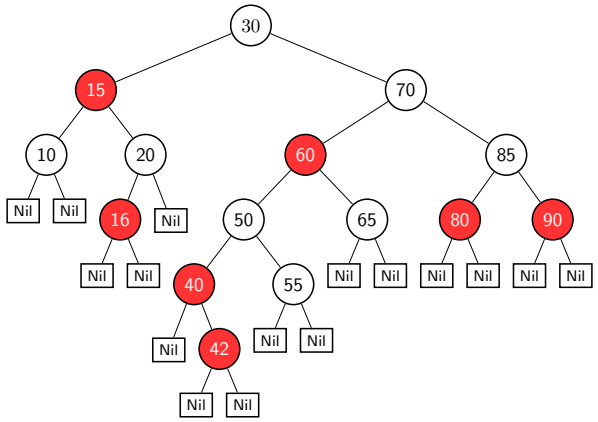


(a) Stato attuale

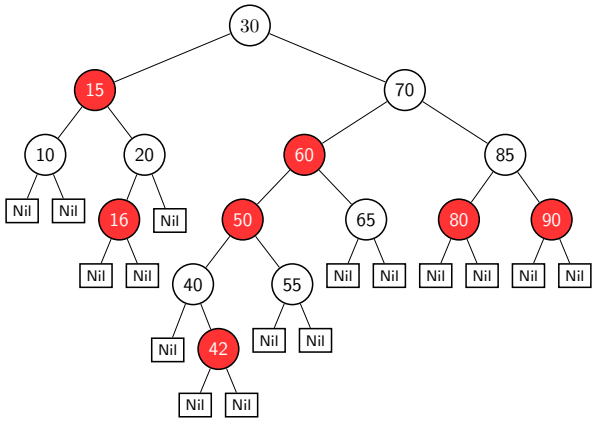


(b) Inserimento del nodo 16 andato a buon fine

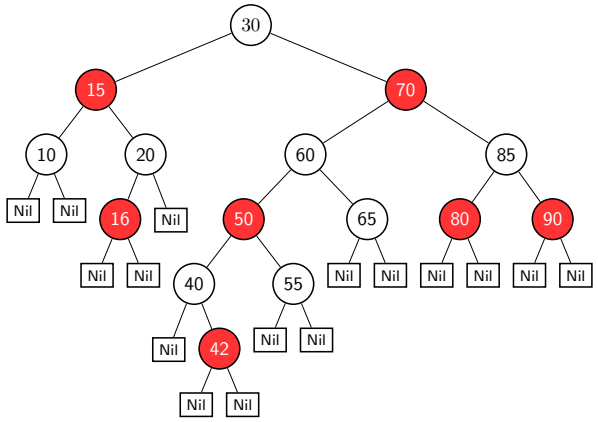
Non violiamo alcun vincolo in quanto il padre di 16 è nero e non abbiamo modificato l'altezza nera. Questo caso rappresenta il caso 2, l'inserimento del nodo 16 è quindi andato a buon fine. Alternativamente proviamo ad inserire il nodo 42 sempre nello stesso albero.



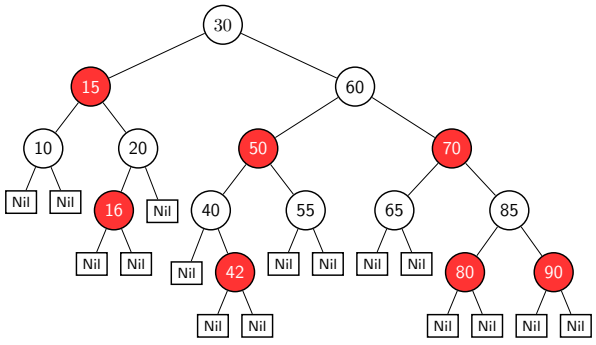
(a) Inserimento del nodo 42, violiamo il secondo vincolo, ci troviamo nel caso 3 (z rosso), quindi coloriamo di nero p e z e di rosso n, il problema si sposta ad n



(b) Violiamo il terzo vincolo, entrambi i nodi rossi sono figli *sinistri* quindi ci troviamo nel caso 5a, quindi coloriamo di nero p, di rosso n ...



(c) ... ed effettuiamo una rotazione a destra con perno n



(d) Abbiamo ripristinato il terzo vincolo, gli altri vincoli non sono mai stati violati, quindi abbiamo finito

Questo esempio ci mostra chiaramente come gli alberi Red-Black, attraverso il rispetto dei vincoli, tendano a mantenere bilanciato l'albero. Esiste una versione *top-down* dell'algoritmo di inserimento che scende fino interessato "aggiustando" l'albero man mano.

7.2.3 Rimozione di un nodo

Se il nodo rimosso è rosso l'altezza nera rimane invariata, non sono stati creati nodi rossi consecutivi e la radice resta nera. Il problema sorge quando si rimuovono nodi neri in quanto è possibile che siano stati violati il primo ed il terzo vincolo e sicuramente è stato violato il quarto vincolo in quanto cambia l'altezza nera. L'algoritmo balanceDelete(T, t) ripristina la proprietà Red-Black con rotazioni e cambiamenti di colore. Ci sono quattro casi possibili (e 4 simmetrici).

Algoritmo 7.2.6: Rimozione di un nodo in un RED-BLACK TREE

```
balanceDelete(TREE T, TREE t)
    t.color ← RED  // coloro il nodo da inserire di rosso
    while (t ≠ T) and (t.color == BLACK) do
        TREE p ← t.parent                                          // riferimento al padre
        if t == p.left then
            TREE f ← p.right                                       // riferimento al fratello
            TREE ns ← f.left                                       // riferimento al nipote sinistro
            TREE nd ← f.right                                      // riferimento al nipote destro
(1)         if f.color == RED then
                p.color ← RED
                f.color ← BLACK
                rotateLeft(p)
                // t viene lasciato inalterato, quindi si ricade nei casi 2, 3, 4
            else
(2)             if ns.color == nd.color == BLACK then
                    f.color == RED
                    t ← p  // passo il problema al padre
(3)             else if (ns.color == RED) and (nd.color == BLACK) then
                    ns.color ← BLACK
                    f.color ← RED
                    rotateRight(f)
                    // t viene lasciato inalterato, quindi si ricade nel caso 4
(4)             else if nd.color == RED then
                    f.color == p.color
                    p.color ← BLACK
                    nd.color ← BLACK
                    rotateLeft(p)
                    t ← T
        else
            // casi speculari
```

La cancellazione è concettualmente complicata, ma è efficiente.

(1) si passa ad uno dei casi 2, 3, 4;

(2) si torna ad uno degli altri casi, ma risalendo di un livello l'albero;

(3) si passa al caso 4;

(4) si termina.

È possibile visitare al massimo un numero $\mathcal{O}(\log n)$ di casi, ognuno dei quali è gestito in $\mathcal{O}(1)$.

Capitolo 8

Hashing

Capitolo 9

Insiemi

Possono essere implementati con molte delle strutture dati viste fin'ora. Ognuna delle quali rappresenta valntaggi e svantaggi.

9.1 Realizzazione con vettori booleani

L'insieme viene rappresentato attraverso un vettore booleano di m elementi. Il quale è notevolmente semplice da implementare ed estremamente efficiente verificare se un elemento appartiene all'insieme. Sfortunatamente la memoria occupata è $\mathcal{O}(m)$, indipendentemente dalle dimensioni effettive, inoltre alcune operazioni dipendono dalla memoria utilizzata per memorizzare questi oggetti, piuttosto che dal numero di oggetti effettivamente memorizzati, il che porta ad una complessità di queste operazione di $\mathcal{O}(m)$.

Struttura dati SET implementata come Vettore Booleano

```
boolean[] V
int size
int dim

SET Set(int m)
    SET t ← new SET
    t.size ← 0
    t.dim ← m
    t.V ← [false] * m
    return t

SET contains(int x)
    if 1 ⩽ x ⩽ dim then
        return V[x]
    else
        return false

int size
    return size

insert(int x)
    if 1 ⩽ x ⩽ dim then
        if not V[x] then
            size++
            V[x] ← true

remove(int x)
    if 1 ⩽ x ⩽ dim then
        if V[x] then
            size++
            V[x] ← false
```

```
SET union(SET A, SET B)
    // crea un insieme della capacità max
    SET C ← Set(max(A.dim, A.dim))

    // inserisci gli elementi di A
    from i ← 1 until A.dim do
        if A.contains(i) then
            C.insert(i)

    // inserisci gli elementi di B
    from i ← 1 until B.dim do
        if A.contains(i) then
            C.insert(i)

SET intersection(SET A, SET B)
    // crea un insieme della capacità min
    SET C ← Set(min(A.dim, A.dim))

    from i ← 1 until min(A.dim, A.dim) do
        // se è contenuto in entambi
        if A.contains(i) and B.contains(i) then
            C.insert(i) // aggiungilo

SET difference(SET A, SET B)
    SET C ← Set(A.dim)

    from i ← 1 until A.dim do
        // se è contenuto A e non in B
        if A.contains(i) and not B.contains(i)
         then
            C.insert(i) // aggiungilo
```

9.1.1 Implementazioni nei linguaggi di programmazione

Esistono alcune implementazioni nei linguaggi attualmente utilizzati. In Java esiste la struttura dati `BitSet` i cui meotodi sono illustrati nella tabella 9.1. Mentre in `C++ STL` esistono due implementazioni `std::bitset` e `vector<bool>`:

- `bitset` è una struttura dati con dimensione fissata nel template al momento della compilazione;
- `vector<bool>` è una specializzazione di `vector` per ottimizzare la memorizzazione, ha dimensione dinamica.

Tabella 9.1: Implementazione `java.util.BitSet`

Operazione	Metodo
contains	`boolean get(int i)`
size	`int cardinality()`
insert	`void set(int i)`
remove	`void clear(int i)`
union	`void and(BitSet set)`
intersection	`void or(BitSet set)`

9.2 Realizzazione con vettore non ordinato

Struttura dati SET implementata come vettore non ordinato

```
SET difference(SET A, SET B)
    SET C ← Set  // non ha bisogno della dimensione
    from s ∈ A do
        // se non è contenuto in B
        if not B.contains(s) then
            C.insert(s)  // aggiungilo
```

Costo delle operazioni Le operazioni di ricerca, inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di inserimento (assumendo che non esista l'elemento) costano $\mathcal{O}(1)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(nm)$.

9.3 Realizzazione vettore ordinato

Struttura dati SET implementata come vettore ordinato

```
SET intersection(LIST A, LIST B)
    LIST C ← Set  // non ha bisogno della dimensione
    // creo puntatori alle liste
    POS p ← A.head
    POS q ← B.head
    while not A.finished(p) and B.finished(q) do
        if A.read(p) == B.read(q) then // se gli elementi coincidono
            C.insert(C.tail, A.read(p))  // inseriscilo nell'intersezione
            // scorri i puntatori
            p ← A.next(p)
            q ← B.next(q)
        else if A.read(p) < B.read(q) then
            p ← A.next(p)  // scorro puntatore di A
        else
            q ← B.next(q)  // scorro puntatore di B
```

Costo delle operazioni Le operazioni di ricerca costano $\mathcal{O}(n)$ con le liste e $\mathcal{O}(\log n)$ con i vettori, le operazioni di inserimento e cancellazione costano $\mathcal{O}(n)$, le operazioni di unione, intersezione e differenza $\mathcal{O}(n)$.

9.4 Reality Check

In realtà si utilizzano strutture dati complesse che permettono di ottimizzare le performance. Se abbiamo bisogno dell'ordinamento si utilizzano alberi bilanciati, mentre se abbiamo bisogno di sapere smeplicemente se un elemento è contenuto o meno si utilizzano le tabelle hash.

Per le operazioni di ricerca, inserimento e cancellazione negli alberi bilanciati hanno una complessità di $\mathcal{O}(\log n)$, mentre nelle tabelle hash di $\mathcal{O}(1)$.

Le implementazioni più diffuse degli alberi bilanciati sono `TreeSet` in Java, `OrderedSet` in Python e `set` in `C++`, mentre le implementazioni più diffuse delle tabelle hash sono `HashSet` in Java, `set` in Python e `unordered_set` in `C++`.

Tabella 9.2: Implementazioni e relative complessità delle operazioni
$m \equiv$ dimensione del vettore o della tabella hash

	contains lookup	insert	remove	min max	foreach (memoria)	Ordine
Vettore booleano	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(m)$	Sì
Lista non ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	No
Lista ordinata	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Vettore ordinato	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	Sì
Alberi bilanciati	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	Sì
Hash (mem. interna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(n)$	No
Hash (mem. esterna)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m+n)$	$\mathcal{O}(m+n)$	No

Capitolo 10

Grafi

10.1 Introduzione

Un grafo non è altro che un insieme di entità collegate da un insieme di relazioni che possono essere interpretate in vari modi.

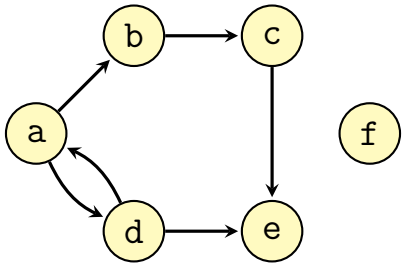
Il questa parte della lezione affronteremo i problemi che possono essere risolti tramite grafi non pesati, ossia:

- ricerca del cammino più breve, misurato in numero di archi (che differisce dal problema del cammino minimo definito su grafi pesati che cerca di stabilire quale sia il cammino meno costoso);
- componenti (fortemente) connesse;
- verifica ciclicità;
- ordinamento topologico.

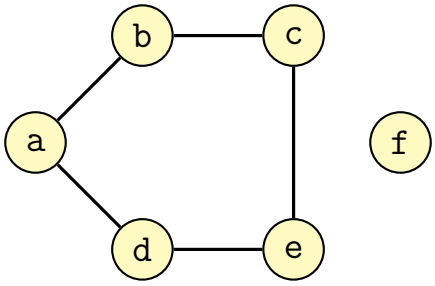
10.1.1 Definizioni

Definizione 10.1.1 (Grafo orientato, *directed*)**.** Un grafo orientato è una coppia $G = (V, E)$ dove V è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie ordinate (u, v) di nodi detti archi (edges).

Definizione 10.1.2 (Grafo non orientato, *undirected*)**.** Un grafo non orientato è una coppia $G = (V, E)$ dove è un insieme di nodi (node) o vertici (vertex), mentre E è un insieme di coppie non ordinate (u, v) dette archi (edges).



(a) Grafo diretto

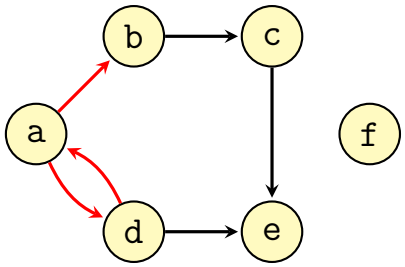


(b) Grafo indiretto

10.1.2 Terminologia

Proprietà (Adiacenza)**.** *Un vertice v è detto adiacente a u se esiste un arco (u, v).*

Proprietà (Incidenza)**.** *Un arco (u, v) è detto incidente da u a v.*



- (a, b) è incidente da a a b
- (a, d) è incidente da a a d
- (d, a) è incidente da d a a
- b è adiacente ad a
- d è adiacente ad a
- a è adiacente a d

Nota. In un grafo non orientato, la relazione di adiacenza è simmetrica.

10.1.3 Ragionamenti sulla complessità

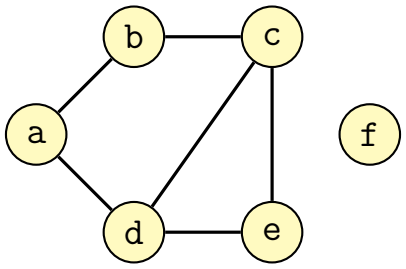
Definiamo il numero di nodi con $n = |V|$, ed il numero di archi con $m = |E|$. C'è una relazione precisa fra n ed m. In un grafo non orientato $m \leqslant \frac{n(n-1)}{2} = \mathcal{O}(n^2)$, mentre in un grafo orientato $m \leqslant n^2 - n = \mathcal{O}(n^2)$. Questi ordini di grandezza ci serviranno a valutare quale algoritmo utilizzare in base al numero di possibili archi. La complessità viene quindi espressa in termini sia di n che di m, ad esempio $\mathcal{O}(n + m)$.

10.1.4 Casi speciali

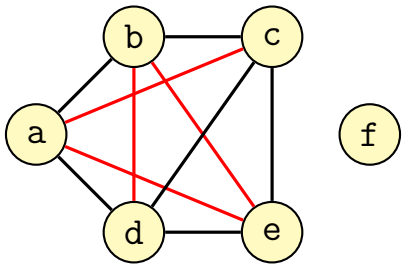
Completezza di un grafo

Un grafo con un arco fra tutte le coppie di nodi è detto *completo*. Informalmente (non c'è accordo sulla definizione) parleremo di:

- grafo *sparso* se ha "pochi archi"; ad esempio grafi con m pari a $\mathcal{O}(n)$ o $\mathcal{O}(n \log n)$ sono considerati tali;
- grafo *denso* se ha "tanti archi"; ad esempio grafi con m pari a $\Omega(n^2)$.



(a) Grafo sparso



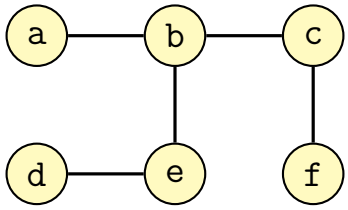
(b) Grafo denso

Alberi radicati

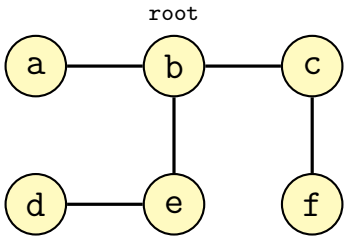
Definizione 10.1.3 (Albero non radicato, libero)**.** Un albero libero (free tree) è un grafo connesso con $m = n - 1$, dove non viene identificata una radice.

Definizione 10.1.4 (Albero radicato)**.** Un albero radicato (rooted tree) è un grafo connesso con $m = n - 1$ nel quale uno dei nodi è designato come radice.

Definizione 10.1.5 (Foresta)**.** Un insieme di alberi è un grafo detto foresta.



(a) Albero libero



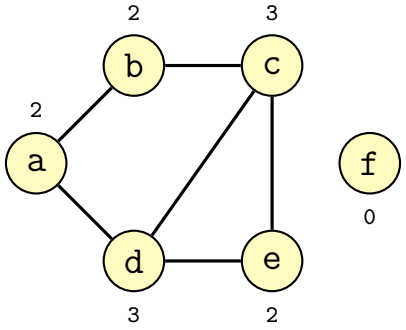
(b) Albero radicato

Definizioni

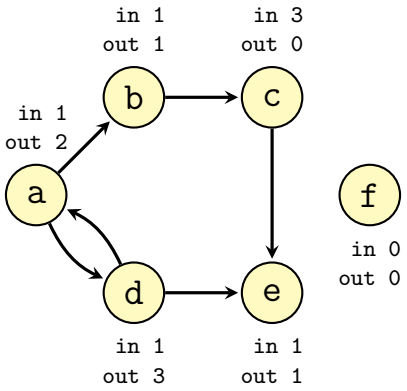
Definizione 10.1.6 (Grado, *degree*)**.** In un grafo non orientato il grado di un nodo è il numero di archi incidenti su di esso.

Definizione 10.1.7 (Grado entrante, *in-degree*)**.** In un grafo orientato il grado entrante di un nodo è il numero di archi entranti su di esso.

Definizione 10.1.8 (Grado uscente, *out-degree*)**.** In un grafo orientato il grado uscente di un nodo è il numero di archi uscenti da esso.



(a) Grado di un grafo indiretto



(b) Grado di un grafo diretto

Definizione 10.1.9 (Cammino, *path*)**.** In un grafo $G = (V, E)$, un cammino C di lunghezza k è una sequenza di nodi $u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leqslant i \leqslant k - 1$.

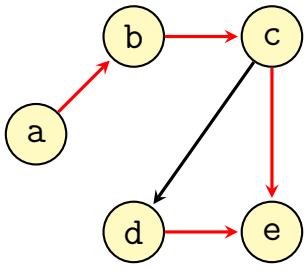


Figura 10.5: Cammino in un grafo diretto, a, b, c, e, d è un cammino di lunghezza 4

Nota. Un cammino è detto semplice se tutti i suoi nodi sono distinti.

10.1.5 Specifica

Nella versione più generale, il grafo è una struttura di dati dinamica che permette di aggiungere e rimuovere nodi e archi. La specifica che utilizzeremo non prevede la rimozione dei nodi dal grafo.

Algoritmo 10.1.1: Specifica della struttura dati GRAPH

```
Graph                                          // crea un grafo vuoto
SET V                                          // restituisce l'insieme di tutti i nodi
int size                                       // restituisce il numero di nodi
SET adj(NODE u)                                // restituisce l'insieme di nodi adiacenti a u
insertNode(NODE u)                             // aggiunge il nodo u al grafo
deleteNode(NODE u)                             // rimuove il nodo u dal grafo
insertEdge(NODE u, NODE v)                     // aggiunge l'arco (u,v) al grafo
deleteEdge(NODE u, NODE v)                     // rimuove l'arco (u,v) dal grafo
```

10.1.6 Memorizzazione

Esistono due diversi modi per memorizzare un grafo:

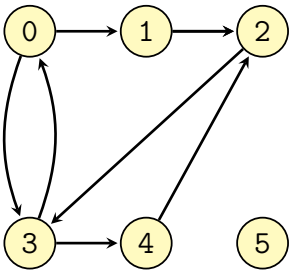
1. **matrici di adiacenza**: utilizza una matrice contenente un bit per indicare la presenza di ciascun arco: questo permette di controllare in tempo costante se un determinato arco è presente. La matrice di adiacenza viene ottenuta nel seguente modo:

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

2. **lista di adiacenza**: una lista delle adiacenze presenti fra i nodi.

Grafo orientato

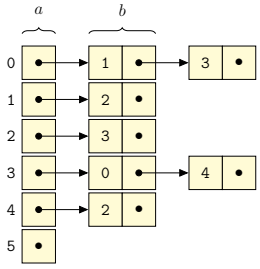
Memorizzare un grafo orientato attraverso matrice di adiacenza occupa uno spazio pari a n^2 bit, mentre se si utilizza una lista di adiacenza vengono occupati $an + bm$ bit.



(a) Grafo orientato

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	0	1	0	0
1	0	0	1	0	0	0
2	0	0	0	1	1	0
3	1	0	0	0	1	0
4	0	0	1	0	0	0
5	0	0	0	0	0	0

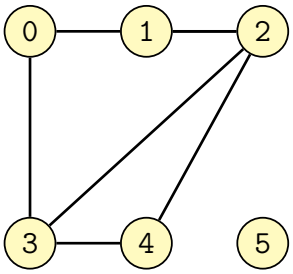
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo non orientato

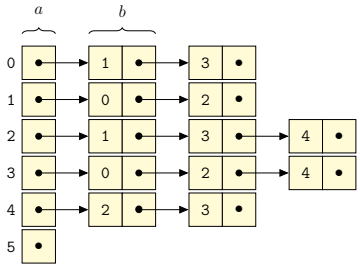
Se il grafo non è orientato ed utilizziamo una matrice di adiacenza per memorizzarlo, è sufficiente memorizzare solo la metà superiore, occupando uno spazio pari a $n(n-1)/2$ bit, mentre con lista di adiacenza dobbiamo raddoppiare i puntatori occupando $an + 2 \cdot bm$ bit.



(a) Grafo non orientato

	0	**1**	**2**	**3**	**4**	**5**
0		1	0	1	0	0
1			1	0	0	0
2				1	1	0
3					1	0
4						0
5						

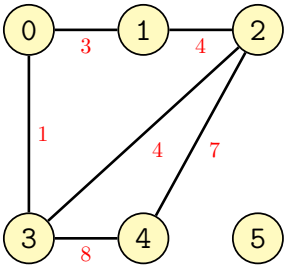
(b) Matrice di adiacenza



(c) Lista di adiacenza

Grafo pesato

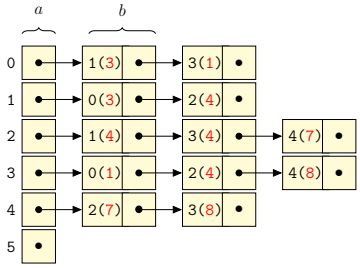
Infine se il grafo è pesato ed usiamo le matrici per rappresentarlo possiamo memorizzare il peso al posto del valore booleano, l'assenza dell'arco è quindi segnalata da un valore particolare (come -1 o ∞ in base alla procedura che vogliamo risolvere). Mentre se utilizziamo una lista di adiacenza memorizzeremo semplicemente il peso.



(a) Grafo pesato

	0	**1**	**2**	**3**	**4**	**5**
0		3	0	1	0	0
1			4	0	0	0
2				4	7	0
3					8	0
4						0
5						

(b) Matrice di adiacenza pesata



(c) Lista di adiacenza pesata

Dettagli sull'implementazione

Nel seguito, se non diversamente specificato, assumeremo che:

- l'implementazione sia basata su vettori di adiacenza (statici o dinamici);
- l'accesso alle informazioni abbia costo costante (ossia che la classe NODE sia equivalente a **int**);
- le operazioni per aggiungere nodi e archi abbiano costo ammortizzato $\mathcal{O}(1)$;
- dopo l'inizializzazione, il grafo sia statico.

Iterazione su nodi e archi

Negli algoritmi che seguiranno utilizzeremo questi schemi di codice per iterare su nodi ed archi.

Algoritmo 10.1.2: Schemi di iterazione per nodi ed archi

```
// iterare sui nodi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }

// iterare sugli archi
foreach u ∈ G.V do
    { Esegui operazioni sul nodo u }
    foreach v ∈ G.adj(u) do
        { Esegui operazioni sull'arco u, v }
```

Complessità dell'iterazione Quest'operazione costa $\mathcal{O}(m+n)$ con le liste di adiacenza, mentre $\mathcal{O}(n^2)$ con le matrici di adiacenza.

Riassumendo

Le matrici sono ideali per grafi *densi*, occupano uno spazio $\mathcal{O}(n^2)$ e iterare su tutti gli archi costa $\mathcal{O}(n^2)$, verificare se u è adiacente a v richiede tempo costante $\mathcal{O}(1)$.

Le liste di adiacenza sono ideali per grafi *sparsi*, occupano uno spazio $\mathcal{O}(n+m)$ e iterare su tutti gli archi costa $\mathcal{O}(n+m)$, verificare se u è adiacente a v richiede tempo lineare $\mathcal{O}(n)$.

10.2 Visite dei grafi

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ di partenza (che prende il nome di *radice* o di *sorgente*), si vuole visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da r.

In ampiezza La visita in ampiezza effettua una visita dei nodi per livelli: prima visita la radice, poi i nodi a distanza uno dalla radice, poi i nodi a distanza due e così via... Una possibile applicazione di questa visita è quella di calcolare i cammini più brevi da una singola sorgente.

In profondità La visita in profondità effettua una visita ricorsiva: per ogni nodo adiacente, si visita il nodo e tutti i suoi nodi adiacenti ricorsivamente. Delle possibili applicazioni sono l'ordinamento topologico, la verifica della ciclicità e le componenti connesse e fortemente connesse.

10.2.1 Visita in ampiezza

Un approccio ingenuo alla visita di un grafo potrebbe essere il seguente:

Algoritmo 10.2.1: Primo tentativo di visita di un grafo

```
visita(GRAPH G)
    foreach u ∈ G.V do
        { visita nodo u }
        foreach v ∈ G.adj(u) do
            { visita arco (u, v) }
```

Ma la struttura del grafo non viene presa in considerazione, poiché si itera su tutti i nodi e gli archi senza alcun criterio. Un possibile approcio potrebbe essere quello di sfruttare l'algoritmo delle visite sugli alberi

Algoritmo 10.2.2: Algoritmo adatto all'attraversamento degli alberi

```
BFSTraversal(GRAPH G, int r)
    QUEUE Q ← Queue
    Q.enqueue(r)
    while not Q.isEmpty do
        NODE u ← Q.dequeue
        { visita il nodo u }
        foreach u ∈ G.adj(u) do
            Q.enqueue(v)
```

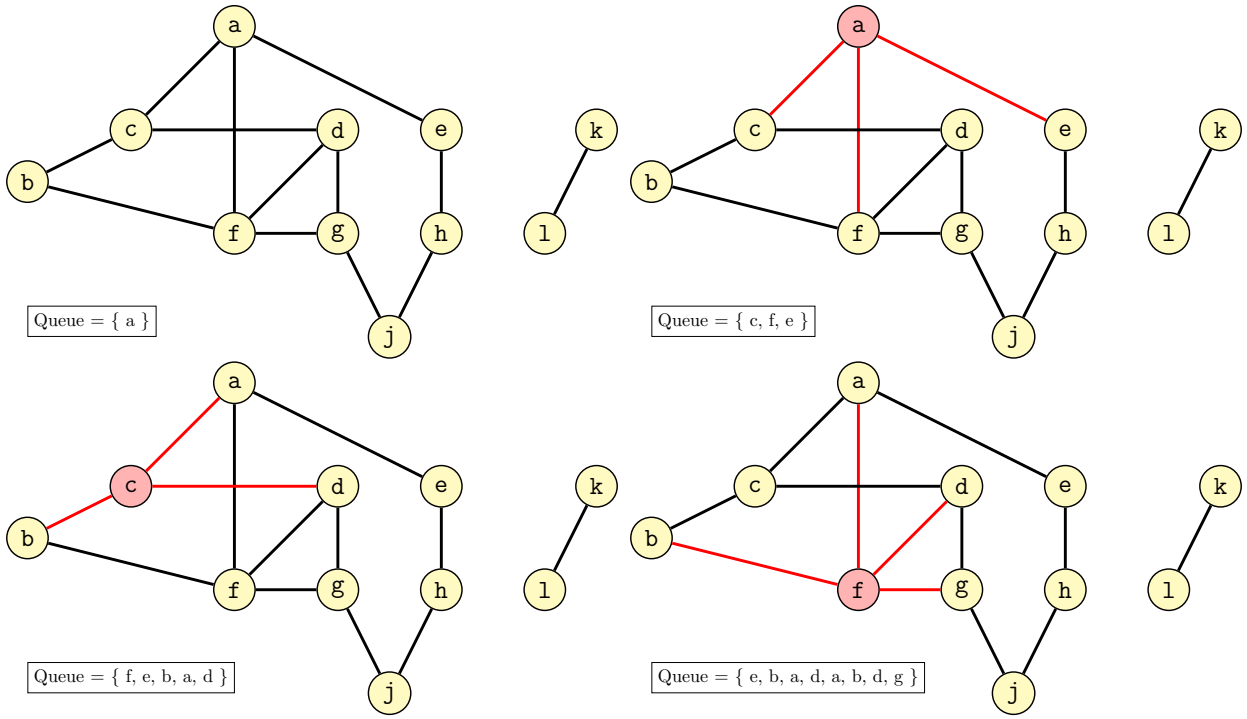


Figura 10.9: Esempio di attraversamento di un grafo tramite la procedura di visita di un albero

Dall'esempio possiamo notare come i nodi vengano reinseriti all'infinito all'interno della coda e questo non permette all'algoritmo di terminare. Negli alberi, data la loro struttura, abbiamo la sicurezza che non visiteremo mai lo stesso nodo più di una volta. Abbiamo bisogno di un meccanismo che ci permetta di evitare che questo avvenga. Possiamo farlo memorizzando i nodi che abbiamo già visitato.

Algoritmo 10.2.3: Algoritmo adatto all'attraversamento dei grafi

```
visita(GRAPH G, NODE r)
    SET S ← Set  // insieme generico, da specificare (STACK, QUEUE)
    S.insert(r)  // inserisco il nodo, da specificare

    // ho visitato il nodo
    { marca il nodo r come "scoperto" }

    // fintanto che l'insieme non è vuoto
    while S.size > 0 do
        // la politica di rimozione dipende dal problema da risolvere
        NODE u ← S.remove
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do
            { esamina l'arco (u, v) }
            if v non è già stato scoperto then
                // serve a non inserire il nodo più di una volta
                { marca il nodo v come "scoperto" }
                S.insert(v)  // inserisce il nodo nell'insieme, da specificare
```

Gli obiettivi della visita in ampiezza sono:

- visitare i nodi a distanze crescenti dalla sorgente: visitare quindi i nodi a distanza k prima di quelli a distanza $k+1$;
- calcolare il cammino più breve da r a tutti gli altri nodi, dove il cammino più breve è il percorso con il minor numero di archi;
- generare un albero in ampiezza (*breadth-first*): l'albero in ampiezza è un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Algoritmo 10.2.4: Procedura specializzata per la visita in ampiezza di un grafo

```
// visitare tutti i nodi a distanza k prima di visitare i nodi a distanza k+1
bfs(GRAPH G, NODE r)
    QUEUE S ← Queue // creo una pila
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    boolean[] visitato ← boolean[1...G.n] // della dimensione del no. di nodi
    foreach u ∈ G.V − {r} do visitato[u] ← false // devo ancora visitarli
    visitato[r] ← true // radice visitata

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue // rimuovo un nodo
        { esamina il nodo u }
        foreach v ∈ G.adj(u) do // per ciascun nodo adiacente "v"
            { esamina l'arco (u, v) }
            if not visitato[v] then // se non ho ancora visitato "v"
                visitato[v] ← true // marcalo come visitato
                S.enqueue // inseriscilo nella coda
```

10.2.2 Cammini più brevi

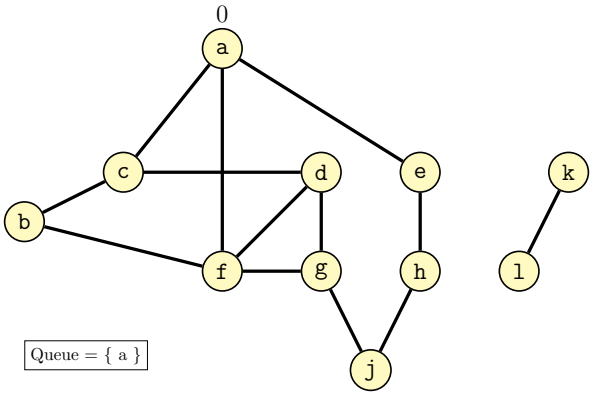
Vediamo ora un'applicazione della visita in ampiezza: la ricerca dei cammini più brevi e lo facciamo tramite un esempio particolare: calcolare il numero di erdös.

Paulo Erdös è stato uno dei matematici più prolifici al mondo (1500+ articoli e 500+ co-autori). Definiamo il numero di erdos nel seguente modo:

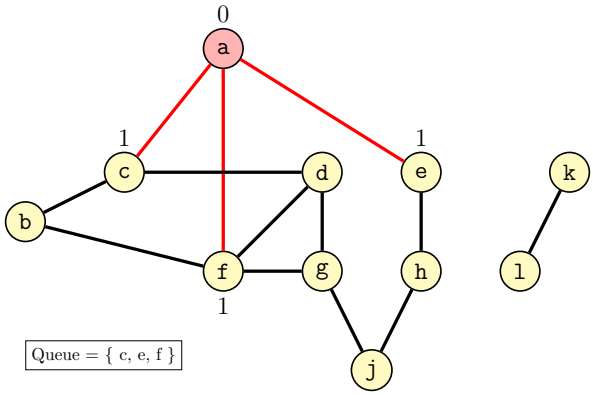
- Erdös ha valore $erdos = 0$;
- i co-autori di Erdös hanno $erdos = 1$;
- se X è co-autore di qualcuno con $erdos = k$ e non è co-autore con qualcuno con $erdos < k$, allora X ha $erdos = k + 1$;
- le persone non raggiunte da questa definizione hanno $erdos = +\infty$.

Algoritmo 10.2.5: Ricerca dei cammini minimi più brevi dalla radice

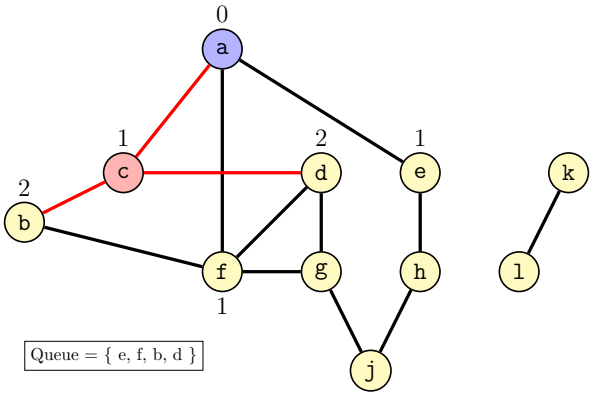
```
// il camminimo più breve fra due vertici viene memorizzato tramite il vettore dei padri p
erdos(GRAPH G, NODE r, int[] erdos, NODE parent)
    // struttura di supporto
    QUEUE S ← Queue // creo una pila
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    foreach u ∈ G.V − {r} do erdos[u] ← ∞ // nodi non ancora raggiunti
    erdos[r] ← true // erdõs ha distanza 0 da se stesso
    parent[r] ← nil // per la stampa del cammino

    // visita del grafo
    while not S.isEmpty do
        NODE u ← S.dequeue
        foreach u ∈ G.adj(u) do
            { esamina l'arco (u, v) }
            if erdos[v] == ∞ then
                // il nodo non è stato ancora stato scoperto
                erdos[v] ← erdos[u] + 1 // gli assegno un livello di erdõs+1
                parent[v] ← u // memorizzo il padre del nodo attuale nel v. dei padri
                S.enqueue(v) // è la prima volta che lo raggiungo quindi lo metto in coda
```



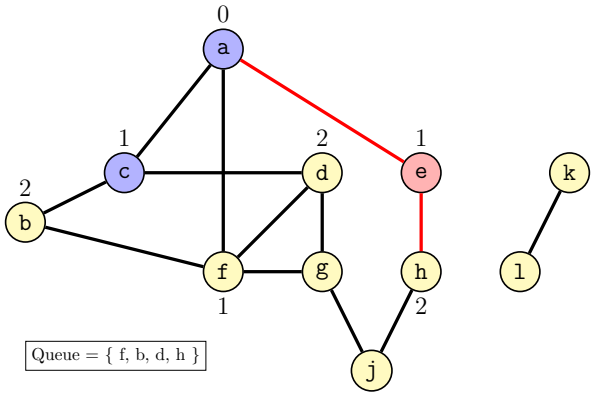
(a) Assegno al nodo a distanza 0, lo aggiungo alla coda



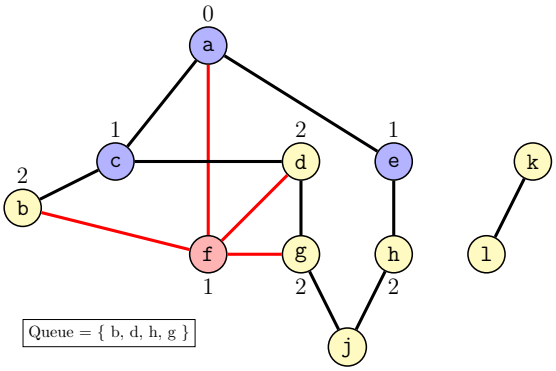
(b) Estraggo il nodo a, assegno ai suoi nodi adiacenti non ancora visitati (c, f, e) distanza $a + 1 = 1$ e li aggiungo alla coda



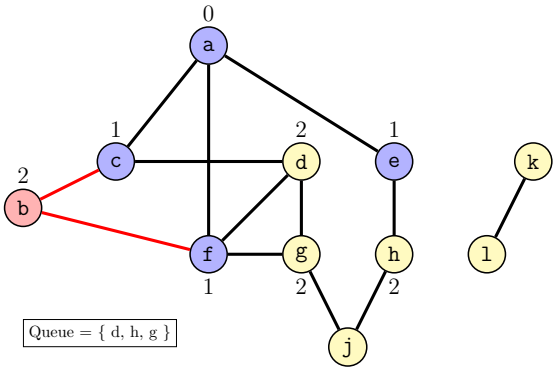
(c) Estraggo il nodo c, assegno ai suoi nodi adiacenti non ancora visitati (b, d) distanza $c + 1 = 2$ e li aggiungo alla coda



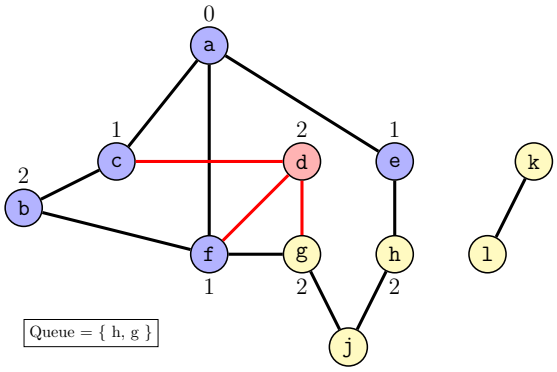
(d) Estraggo il nodo e, assegno ai suoi nodi adiacenti non ancora visitati (h) distanza $e + 1 = 2$ e lo aggiungo alla coda



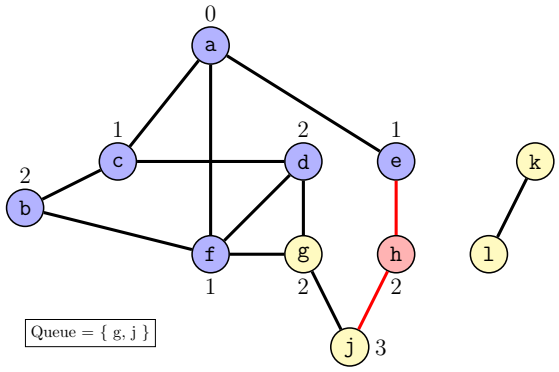
(e) Estraggo il nodo f, assegno ai suoi nodi adiacenti non ancora visitati (b, g) distanza $f + 1 = 2$ e li aggiungo alla coda



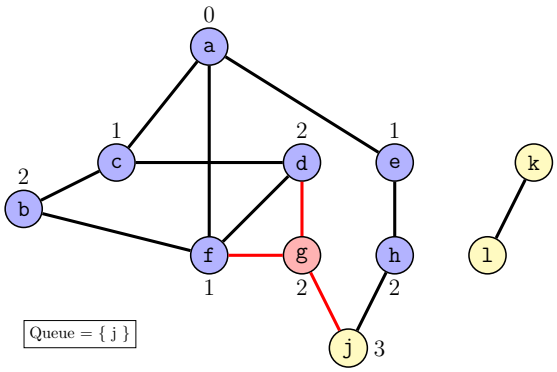
(f) Estraggo il nodo b, il quale non ha nodi adiacenti non ancora visitati



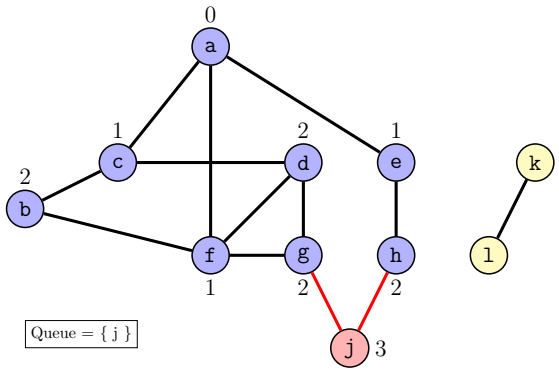
(g) Estraggo il nodo d, il quale non ha nodi adiacenti non ancora visitati



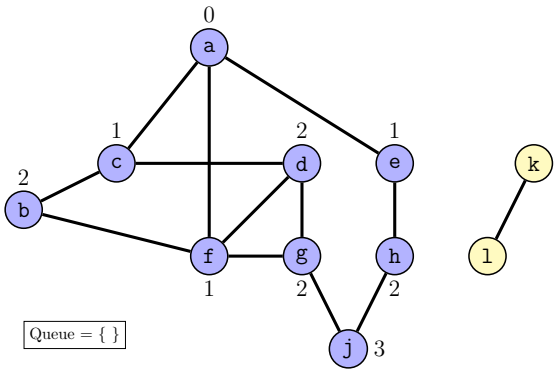
(h) Estraggo il nodo h, assegno al suo nodo adiacente non ancora visitato (j) distanza $h + 1 = 3$ e lo aggiungo alla coda



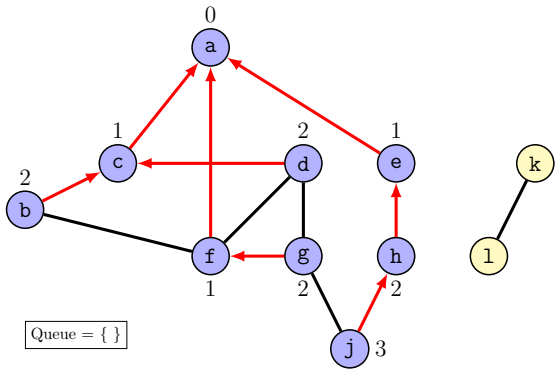
(i) Estraggo il nodo g, il quale non ha nodi adiacenti non ancora visitati



(j) Estraggo il nodo j, il quale non ha nodi adiacenti non ancora visitati



(k) Questo è l'albero risultante



(l) Rappresentazione dell'albero di copertura bfs, attraverso il vettore dei padri

Figura 10.10: Esempio di visita di un grafo tramite la procedura di visita in ampiezza

L'albero "di copertura" con radice r viene memorizzato tramite vettore dei padri (che nell'algoritmo di erdos abbiamo chiamato *parent*).

Algoritmo 10.2.6: Stampa del cammino

```
// stampa il cammino da r a s nell'ordine corretto
stampaCammino(NODE r, NODE s, NODE[] parent)
    // parent: vettore dei padri
    if r == s then // se è la radice
        stampa s // la stampo
    else if parent[s] == nil then
        stampa "nessun cammino da r a s"
    else
        // la chiamata ricorsiva prima della stampa per stampare in ordine
        stampaCammino(r, parent[s], p)
        stampa s
```

10.2.3 Complessità della visita in ampiezza

Ognuno degli n nodi viene inserito nella coda al massimo una volta, ed ogni volta che un nodo viene estratto tutti i suoi archi vengono analizzati una volta sola, per una complessità totale di $\mathcal{O}(m+n)$. Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} out_d(u)$$

dove $out_d(u)$ è il grado uscente (*out-degree*) del nodo u.

10.2.4 Visita in profondità

Molto spesso la visita in profondità è solo una parte di una soluzione ad un altro problema, viene utilizzata per esplorare un intero grafo, e non solo i nodi raggiungibili da una singola sorgente.

L'output dell'algoritmo non è più un singolo albero, ma una foresta *depth-first* (ossia un insieme di alberi *depth-first*).

La struttura dati utilizzata non è più una coda, bensì uno *stack* implicito (attraverso la ricorsione) o esplicito (vedremo un algoritmo nel seguito).

Algoritmo 10.2.7: Visita in profondità, ricorsiva con stack implicito

```
// genera un albero depth-first
dfs(GRAPH G, NODE u, boolean[] visitato)
    visitato[u] = true // ho visitato il nodo
(5) { esamina il nodo u (caso pre-visita) }
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) }
        if not visitato[v] then // se non l'ho ancora visitato
            // chiamata ricorsiva
            dfs(G, v, visitato) // lo visito ricorsivamente
(6) { esamina il nodo u (caso post-visita) }
```

Analisi della complessità Questo algoritmo ha la stessa complessità della visita in ampiezza: $\mathcal{O}(n+m)$ con il grafo implementato tramite liste di adiacenza, e $\mathcal{O}(n^2)$ con matrice di adiacenza.

Si presenta però un problema, mentre con gli alberi possiamo assumere che la loro profondità sia limitata, con i grafi non possiamo fare lo stesso discorso. Eseguire una visita in profondità basata su chiamate ricorsive

può essere rischioso (errore di stack overflow) in grafi molto grandi e connessi (in particolare se non sono orientati, in quanto la visita analizza tutti i nodi). È possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio. In tali casi, si preferisce utilizzare una visita in ampiezza, oppure in profondità ma basata su stack esplicito.

Algoritmo 10.2.8: Visita in profondità, iterativa con stack esplicito, visita in *pre*-ordine

```
// effettua una visita in profondità iterativa
dfs(GRAPH G, NODE r)
    STACK S ← Stack
    S.push(r) // inserisco la radice nella pila
    boolean[] visitato ← new boolean[1...G.size]
    foreach u ∈ G.V − {r} do visitato[u] ← false

    visitato[r] ← true // marco la radice come visitata
    while not S.isEmpty do
        NODE u ← S.pop // estraggo un nodo
        if not visitato[v] then // se non l'ho ancora visitato
            { esamina il nodo u in pre-ordine }
            visitato[v] ← true // lo segno come visitato
            foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
                { esamina l'arco (u, v) }
                S.push(v) // lo inserisco nella pila
```

La procedura si ottiene semplicemente sostituendo una pila alla coda utilizzata nella visita in ampiezza.

Un nodo può essere inserito nella pila più volte (tante volte quanti sono gli archi entranti in quel nodo, il numero di archi entranti in tutti i nodi è limitato superiormente dal numero di archi m, quindi $\mathcal{O}(m)$, in quanto il controllo se un nodo è già stato inserito viene fatto all'estrazione, non all'inserimento come avveniva in precedenza.

Complessità della visita in ampiezza con stack esplicito Inserimenti ed estrazioni sono pari al numero degli archi, quindi $\mathcal{O}(m)$, visitare gli archi costa $\mathcal{O}(m)$ e le visite dei nodi costano $\mathcal{O}(n)$, per una complessità risultante di $\mathcal{O}(m+n)$, invariata rispetto agli algoritmi precedenti.

Visita in post-ordine È possibile effettuare una visita in profondità con una procedura con stack esplicito e visita in *post*-ordine ma abbiamo bisogno di aggiungere due "flag": discovery e finish.

Quando un nodo viene scoperto viene inserito nello stack con il tag discovery; quando un nodo viene estratto dalla coda con tag discovery: viene re-inserito con il tag finish e tutti i suoi vicini vengono inseriti; Quando un nodo viene estratto dalla coda con tag finish, viene effettuata la post-visita. Non vedremo il codice poiché è complicato e i dettagli non sono interessanti.

10.2.5 Componenti connesse

Vogliamo identificare le componenti connesse di un grafo. Prima di tutto vogliamo capire se è connesso, dopodiché vogliamo sapere quante sono le sue componenti connesse. Il motivo per cui vogliamo farlo è che molti algoritmi che operano sui grafi iniziano decomponendo il grafo nelle sue componenti connesse per poi ri-comporre i risultati assieme. Sono definite due tipologie di problemi:

- componenti connesse (*Connected Components, CC*);
- componenti fortemente connesse (*Strongly Connected Components, SCC*)

Entrambi i problemi sono definiti su grafi non orientati.

Per capire meglio il problema abbiamo bisogno di qualche definizione:

Definizione 10.2.1 (Raggiungibilità di un nodo)**.** Un nodo v è raggiungibile da un nodo u se esiste almeno un cammino da u a v.

Definizione 10.2.2 (Grafo non orientato connesso)**.** Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 10.2.3 (Componente connessa)**.** Un grafo $G' = (V', E')$ è una componente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

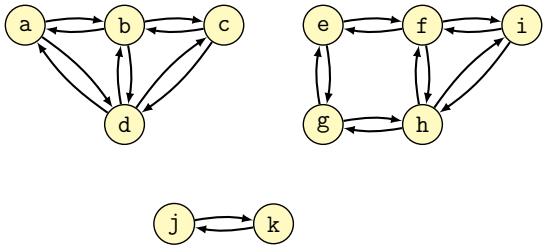
Definizione 10.2.4 (Sottografo)**.** G' è un sottografo di G $(G' \subseteq G) \Leftrightarrow V' \subseteq V$ e $E' \subseteq E$

Definizione 10.2.5 (Grafo massimale)**.** G' è massimale se e solo se non esiste nessun altro sottografo G'' di G tale che G'' è connesso e più grande di G' (ad esempio $G' \subseteq G'' \subseteq G$)

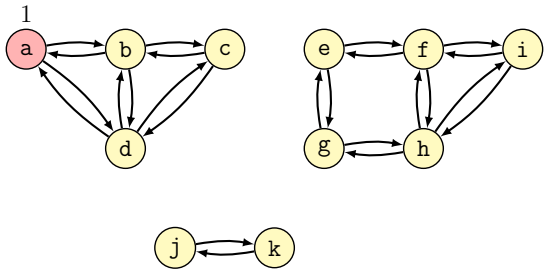
Vogliamo quindi verificare se un grafo è connesso oppure no, ed identificare le sue componenti connesse. Per farlo utilizzeremo di un vettore (che chiameremo id), il quale conterrà l'indentificatore alla quale la componente appartiene ($id[u]$ è l'indentificatore della c.c. a cui appartiene u). Un grafo risulta connesso se, al termine della visita in profondità, tutti i suoi nodi risultano marcati. Altrimenti la visita deve ricominciare da capo da un nodo non marcato, identificando una nuova componente.

Algoritmo 10.2.9: Identifica le componenti connesse di un grafo non orientato

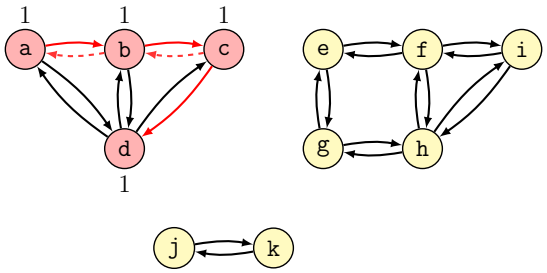
```
// parte iterativa
int[] cc(Graph G, Stack S)
    // creo un vettore della dimensione dei nodi del grafo
    int[] id ← new int[1...G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do id[u] ← 0
    int counter = 0 // contatore delle componenti connesse
    foreach u ∈ G.V do // per ogni nodo del grafo
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++ // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(Graph G, int counter, Node u, int[] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id) // lo visito ricorsivamente
```



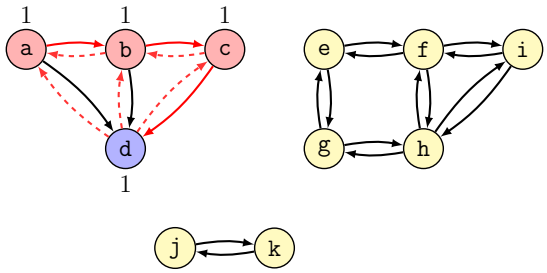
(a) Stato iniziale



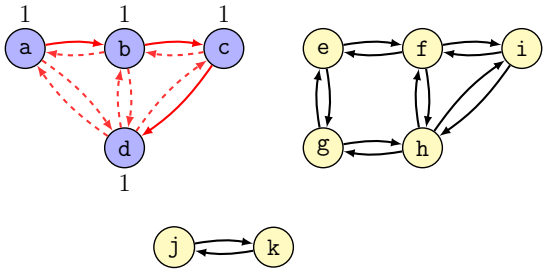
(b) Partiamo dal nodo a per comodità, gli assegniamo il valore 1



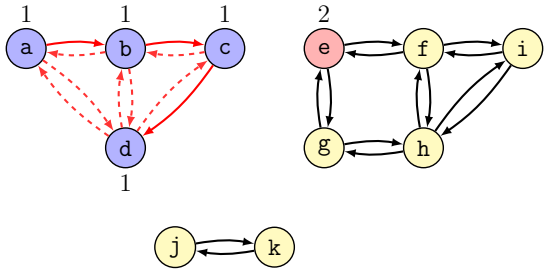
(c) Visito tutti i nodi adiacenti al nodo a



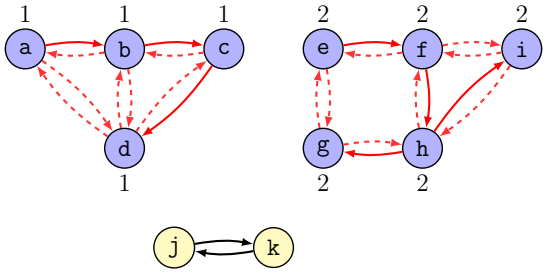
(d) Una volta arrivati al nodo d abbiamo già visitato tutti i suoi possibili vicini



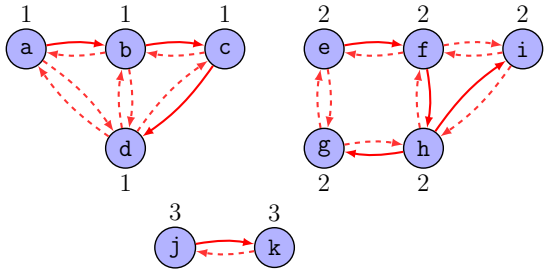
(e) Abbiamo completato la visita della prima componente connessa



(f) Abbiamo ricontrollato se potevamo ripartire da uno qualsiasi dei nodi della prima componente ma aveva già un id assegnato, partiamo da e per comodità



(g) Completo così anche la seconda componente...



(h) ...e la terza

Figura 10.11: Esempio di identificazione delle componenti connesse in un grafo non orientato

10.3 Verifica ciclicità

Grafi aciclici non orientati

Definizione 10.3.1 (ciclo, *cycle*)**.** In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, \dots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

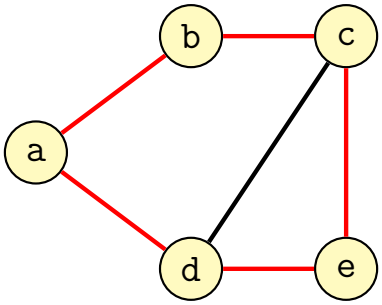


Figura 10.12: $k > 2$ esclude cicli banali composti da coppie di archi, i quali sono onnipresenti nei grafi non orientati. Questo grafo contine 3 cicli, uno di lunghezza 3, uno di lunghezza 4 ed uno di lunghezza 5.

Definizione 10.3.2 (Grafo aciclico)**.** Un grafo non orientato che non contiene cicli è detto aciclico.

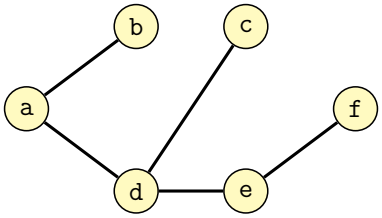


Figura 10.13: Un grafo aciclico.

Un grafo che contiene cicli è detto *ciclico*, altrimenti viene detto *aciclico*. Dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

Nota. Se è un grafo connesso ed è aciclico, allora è un albero.

L'idea è che se visito un nodo che ho già visitato, allora ho identificato un ciclo.

Algoritmo 10.3.1: Ricerca di un ciclo in un grafo non orientato

```
// restituisce true se trova un ciclo
boolean hasCycleRec(GRAPH G, NODE u, NODE p, boolean[] visited)
    // G:       grafo esplorato
    // u:       nodo da esaminare
    // p:       nodo da cui provengo (padre)
    // visited: vettore dei nodi visitati
    visited[u] ← true  // lo visito per la prima volta

    foreach v ∈ G.adj(u) − {p} do // visito tutti i suoi vicini
        // G.adj(u) − {p}:  non considero il nodo da cui provengo (è un grafo orientato)

        if visited[v] then // ho già visitato il nodo
            return true  // ho trovato un ciclo

            // altrimenti effettuo una visita ricorsiva sul nodo vicino v
        else if hasCycleRec(G, v, u, visited) then
            // se una qualsiasi delle sottochiamate ritorna vero, allora ho trovato un ciclo
            return true

    // non ho trovato alcun ciclo
    return false

// ricerca di un ciclo per grafi disconnessi
boolean hasCycle(GRAPH G)
    boolean[] visited ← new boolean[]1G.size  // creo il vettore
    foreach u ∈ G.V do // lo inizializzo
        visited[u] ← false

    foreach u ∈ G.V do // per ciascun nodo appartenente al grafo
        if not visited[u] then // il primo nodo non sarà stato visitato
            if hasCycleRec(G, v, null, visited) then
                // effettuo una visita ricorsiva sul nodo vicino v
                return true

    return false
```

La parte non ricorsiva dell'algoritmo ci permettere di cercare cicli in grafi non connessi, in quanto stiamo osservando grafi e non alberi (i quali sono connessi).

Grafi aciclici orientati

Cosa succede se iniziamo a considerare i grafi orientati?

Definizione 10.3.3 (ciclo, *cycle*)**.** In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k \geqslant 2$ è una sequenza di nodi $u_0, u_1, \dots u_k$ tale che $(u_i, u_{i+1} \in E)$ (un cammino) per $0 \leqslant i \leqslant k-1$ e $u_0 = u_k$ (il primo e l'ultimo nodo coincidono, ossia è chiuso).

La definizione è identica a parte che ora consideriamo come cicli anche i cammini composti da due soli nodi ($k \geqslant 2$)

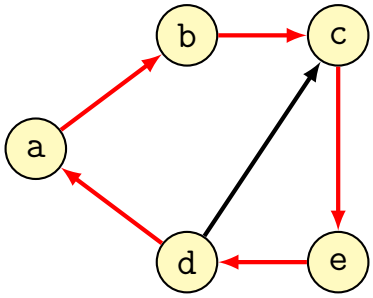
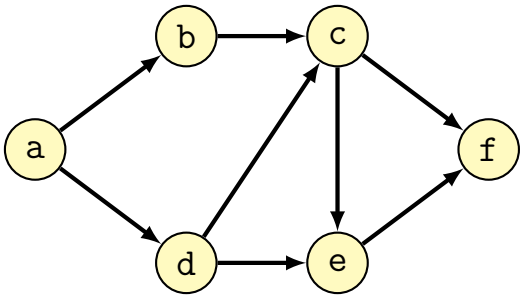


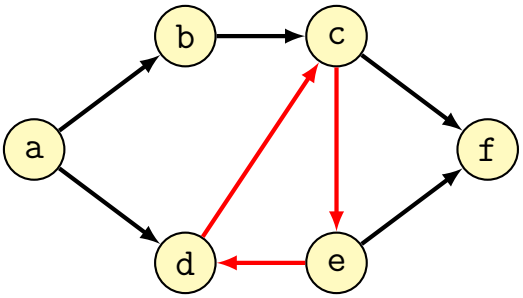
Figura 10.14: a, b, c, e, d, a è un cammino di lunghezza 5

Nota. Un ciclo è detto semplice se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)

Definizione 10.3.4 (Grafo diretto aciclico, *Directed Acyclic Graph*)**.** Un grafo orientato che non contiene cicli è detto DAG (Directed Acyclic Graph)



(a) Un grafo aciclico.



(b) Un grafo ciclico. Il ciclo è dato dai nodi c, e, f.

Figura 10.15: Esempio di grafo aciclico e ciclico

I *Directed Acyclic Graph*, DAG d'ora in poi, rappresentano una classe di problemi ben precisi e vedremo degli algoritmi che trattano nello specifico questi grafi. Il problema è il medesimo: dato un grafo non orientato, vogliamo poter essere in grado di determinare se contiene cicli o meno. Dobbiamo scrivere quindi un algoritmo che restituisca **true** se il grafo contiene un ciclo, **false** altrimenti.

L'algoritmo precedente riesce a risolvere anche queto problema? Riesci a pensare ad un grafo orientato per cui l'algoritmo appena visto non si comporta correttamente?

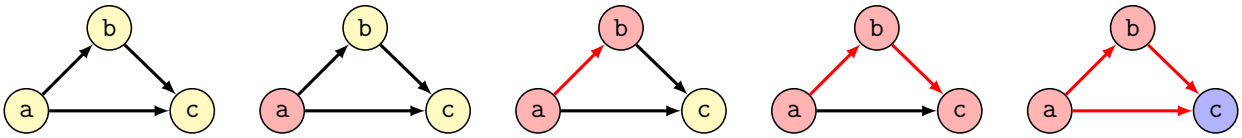
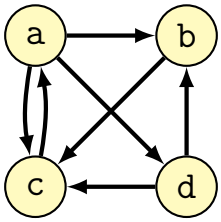


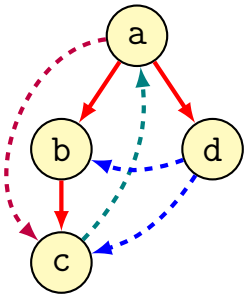
Figura 10.16: Se in un grafo esistono due cammini che portano allo stesso nodo l'algoritmo precedente dirà erroneamente che esiste un ciclo.

Classificazione degli archi

Ogni volta che si esamina un arco da un nodo marcato (visitato) ad un nodo non marcato (non visitato), tale arco viene detto arco dell'albero. Gli archi non inclusi nell'albero possono essere divisi in tre categorie:

- se u è un antenato di v in T, (u, v) è detto arco in avanti;
- se u è un discendente di v in T, (u, v) è detto arco all'indietro;
- altrimenti viene detto arco di attraversamento.





Algoritmo 10.3.2: Schema per visita dell'albero in profondità

```
// classifica i lati di un grafo
dfs-schema(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // time:    contatore
    // dt:      tempo di scoperta
    // ft:      tempo di fine
    esamina il nodo u (caso pre-visita)
    time++ // incremento il contatore
    dt[u] ← time // lo memorizzo nel vettore di scoperta
    // effettuo una visita in profondità
    foreach v ∈ G.adj(u) do
        { esamina l'arco (u, v) (qualsiasi) } // qui si sviluppa la logica dell'algoritmo
        if dt[v] == 0 then // non ho ancora esaminato il nodo
            { esamina l'arco (u, v) (albero) }
            dfs-schema(G, v, time, dt, ft) // effettuo la chiamata ricorsiva
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho trovato
               un arco all'indietro
            { esamina l'arco (u, v) (indietro) }
        else if dt[u] < dt[v] and ft[v] ≠ 0 then
            // se raggiungo un mio discendente e ho terminato la mia visita, allora ho trovato un arco in
               avanti
            { esamina l'arco (u, v) (avanti) }
        else
            // l'ultimo caso rimanente
            { esamina l'arco (u, v) (attraversamento) }
    { visita il nodo u (post-visita) }
    time++ // aggiorno il contatore
    ft[u] ← time // lo memorizzo nel vettore di fine
```

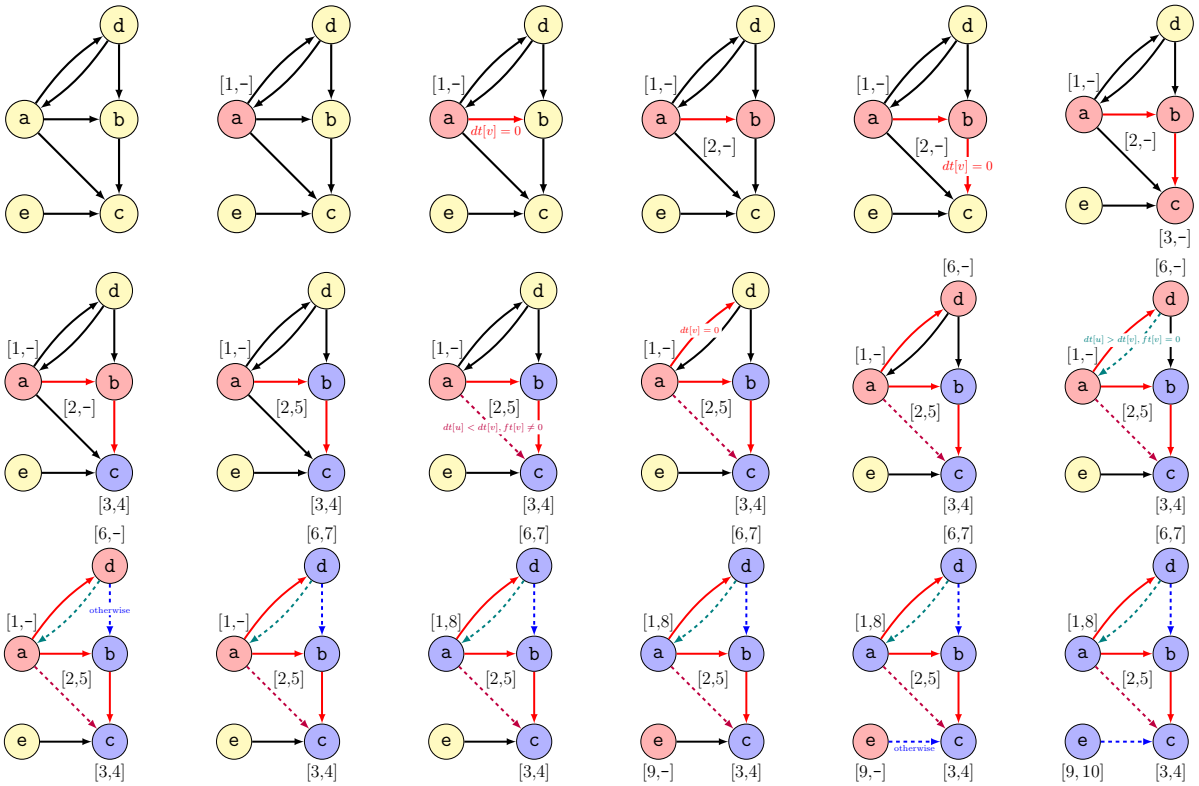


Figura 10.18: Visitati in ordine alfabetico per comodità

Osserviamo i numeri assegnati ai nodi; se li consideriamo come intervalli allora $a \subset b$ perché $[1,8] \subset [2,5]$. Osservando gli intervalli possiamo quindi dedurre le relazioni di discendenza fra i nodi.

Ma perché li classifichiamo? Perché possiamo dimostrare delle proprietà sul tipo di archi e usarle per costruire algoritmi migliori.

Teorema. *Data una visita in profondità di un grafo $G = (V, E)$, per ogni coppia di nodi $(u, v) \in V$, solo una delle condizioni seguenti è vera:*

- *Gli intervalli $[dt[u], ft[u]$ e $[dt[v], ft[v]]$ sono non-sovrapposti; u, v non sono discendenti l'uno dell'altro nella foresta* depth-first*;*
- *L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$; u è un discendente di v in un albero* depth-first*;*
- *L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$; v è un discendente di u in un albero* depth-first*.*

Teorema. *Un grafo orientato è aciclico se e solo se non esistono archi all'indietro nel grafo*

Dimostrazione. Abbiamo due casi:

1. se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e sia (u, v) un arco del ciclo. Il cammino che connette u a v verrà prima o poi visitato, e da v verrà scoperto l'arco all'indietro (v, u) (se esiste un ciclo prima o poi nella visita lo vado a toccare);
2. se esiste un arco all'indietro (u, v) dove v è un antenato di u, allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo.

□

Sfruttando questa dimostrazione possiamo quindi semplificare l'algoritmo precedente per la ricerca di un ciclo in un grafo aciclico diretto.

Algoritmo 10.3.3: Ricerca di un ciclo in un grafo aciclico diretto

```
// applicabile solo ai DAG, in quanto non hanno archi all'indietro
boolean hasCycle(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
    // u:  il primo nodo che viene visitato

    time++ // aumento il contatore
    dt[u] ← time // memorizzo il tempo di scoperta

    foreach v ∈ G.adj(u) do
        if dt[v] == 0 then // non ho ancora scoperto questo nodo
            // effettuo una visita ricorsiva
            if hasCycle(G, v, time, dt, ft) then
                return true

        // logica dell'algoritmo
        else if dt[u] > dt[v] and ft[v] == 0 then
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita, allora ho trovato
               un arco all'indietro (un ciclo)
            return true

    time++ // aumento il contatore
    ft[u] ← time // memorizzo il tempo di fine

    // non ho trovato un ciclo
    return false
```

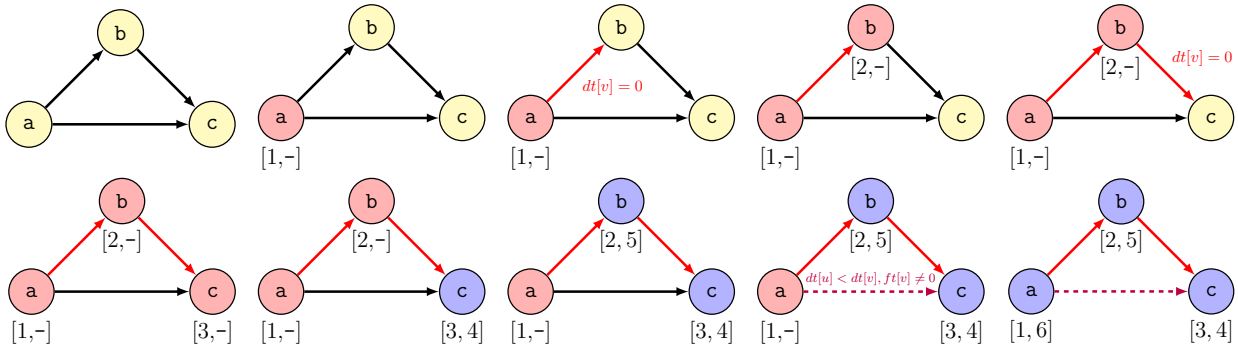


Figura 10.19: Questo è il particolare esempio sulla quale l'algoritmo precedente falliva

Nota. Se nella schema degli intervalli ci sono due intervalli sovrapposti, allora esiste un ciclo.

10.4 Ordinamento topologico

L'obiettivo è quello di scrivere un algoritmo che prende in input un DAG e che ne restituisca un possibile ordinamento topologico.

Definizione 10.4.1 (ordinamento topologico)**.** Dato un grafo diretto e aciclico (DAG) G, un ordinamento topologico di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

Nota. Se il grafo contiene un ciclo, non esiste un ordinamento topologico.

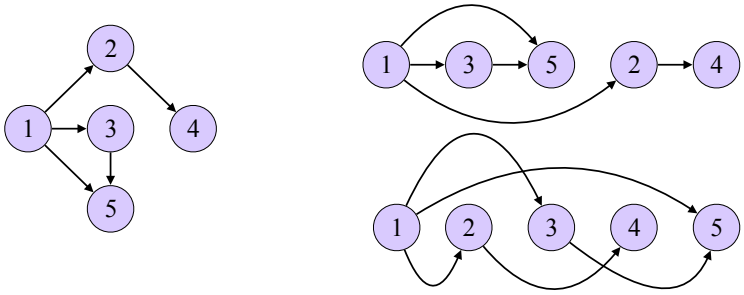


Figura 10.20: Possono esistere più ordinamenti topologici.

Un approccio banale potrebbe essere il seguente:

- trovo un nodo senza archi entranti;
- aggiungo questo nodo nell'ordinamento e lo rimuovo dal grafo insieme a tutti i suoi archi;
- ripeto questa procedura fino a quando tutti i nodi sono stati rimossi.

Si può fare meglio di così. Eseguiamo una visita in profondità nel quale l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista in *post*-ordine. E restituiamo la lista così ottenuta. Restituiamo in output la sequenza dei nodi ordinati per tempo decrescente di fine.

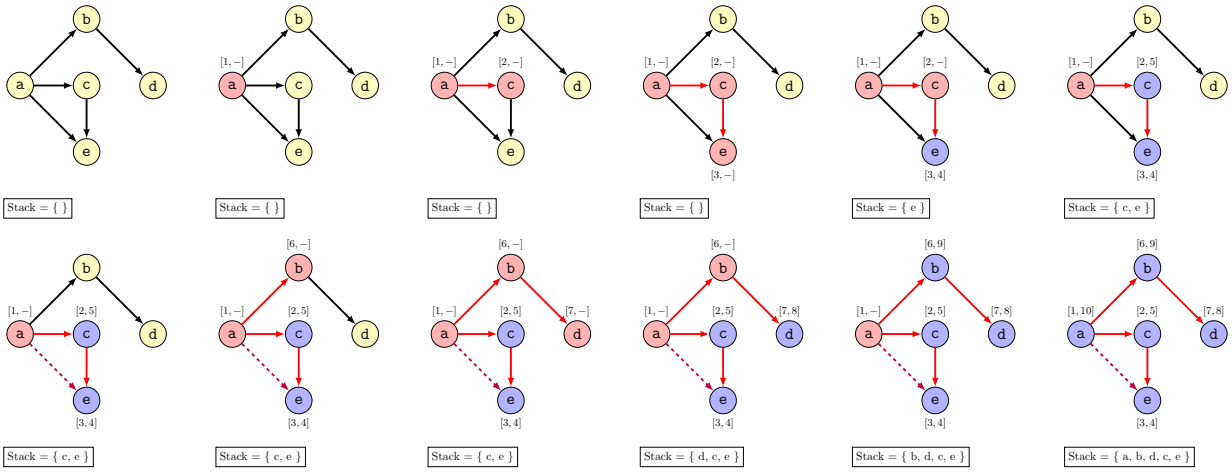


Figura 10.21: Esempio di ordinamento topologico

Algoritmo 10.4.1: Ordinamento topologico di un grafo orientato aciclico

```
// ritorna una pila in cui il primo elemento è il primo elemento dell'ordinamento
STACK topSort(GRAPH G)
    STACK S ← Stack
    boolean[] visited ← new boolean[1...G.size]
    foreach u ∈ G.V do
        visited[u] ← false
    foreach u ∈ G.V do // per ogni nodo del grafo
        if not visitato[u] then // se non l'ho visitato
            // effettua una chiamata ricorsiva
            ts-dfs(G, u, visitato, S)
    return S

// restituisce l'ordinamento topologico dei nodi di un DAG
int ts-dfs(GRAPH G, NODE u, boolean[] visitato, STACK S)
    visitato[u] ← true // imposta il nodo come visitato
    foreach v ∈ G.adj(u) do
        // è una grafo diretto aciclico quindi non ho bisogno di ricordarmi da dove sono venuto
        if not visitato[v] then
            // effettua una visita in profondità
            i ← ts-dfs(G, u, visitato, S)
    S.push(u) // aggiungi il nodo in testa alla pila
```

Quando termino tutte le chiamate ricorsive l'algoritmo restituisce un ordinamento topologico dei nodi del grafo dato in input; quando un nodo è "finito" tutti i suoi discendenti sono stati scoperti e aggiunti alla lista. Aggiungendolo in testa alla lista, il nodo si trova prima dei nodi a cui i suoi archi puntano, ossia i suoi discendenti.

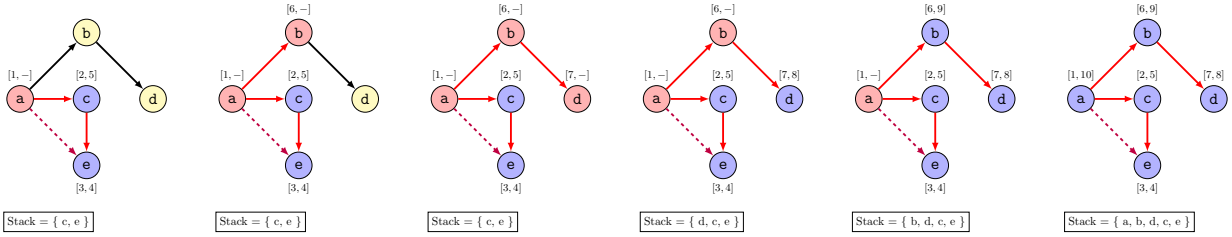


Figura 10.22: Esempio di ordinamento topologico alternativo al precendente, il quale dimostra informalmente che l'algoritmo funziona partendo da qualsiasi nodo

10.5 Componenti fortemente connesse

Definizione 10.5.1 (Grafo fortemente connesso)**.** Un grafo orientato $G = (V, E)$ è fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro suo nodo.

Definizione 10.5.2 (Componente fortemente connessa)**.** Un grafo $G' = (V', E')$ è una componente fortemente connessa di G se e solo se G' è un sottografo connesso e massimale di G.

Le definizioni di fortemente connessa è identica alla definizione di componente connessa, ma si opera su grafi orientati, mentre prima operavamo su grafi non orientati.

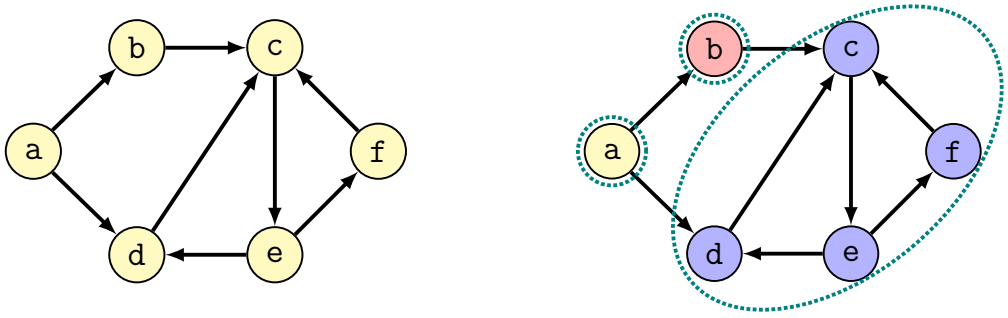


Figura 10.23: (a) e (b) sono componenti connesse massimali, (c, d, e, f) è una componente fortemente connessa.

Per definire le componenti fortemente connesse potremmo applicare l'algoritmo cc; purtroppo il risultato dipende dal nodo di partenza.

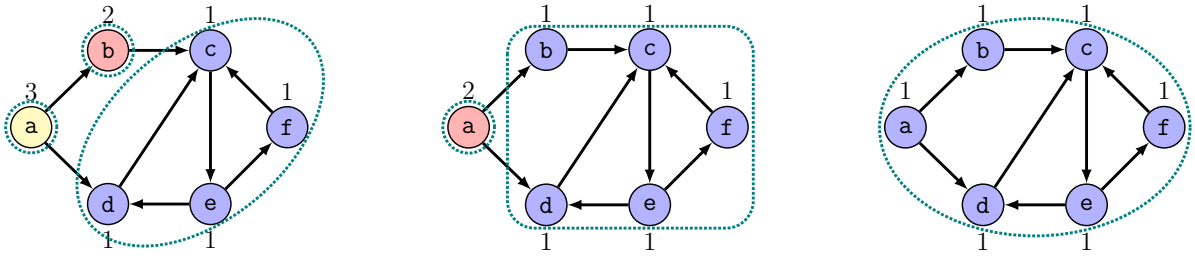


Figura 10.24: Il risultato dipende dal nodo di partenza.

10.5.1 L'algoritmo di Kosaraju

L'algoritmo di Kosaraju:

1. effettua una visita in profondità del grafo G
2. ne calcola il grafo trasposto G^T (ossia il grafo con la direzione degli archi invertiti)
3. esegue una visita in profondità sul grafo trasposto G^T utilizzando l'algoritmo cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita;

Le componenti connesse (e i relativi alberi *depth-first*) rappresentano le componenti fortemente connesse di G.

Algoritmo 10.5.1: Algoritmo di Kosaraju

```
// indentifica le componenti fortemente connesse
int[] scc(GRAPH G)
    STACK S ← topSort(G) // prima visita
    G^T ← transpose(G) // trasposizione del grafo
    return cc(G^T, S) // seconda visita
```

Restituisce un vettore di interi che associa ad ogni nodo l'id della sua componente fortemente connessa. Applicando l'algoritmo di ordinamento topologico *su un grafo generale*, siamo sicuri che:

- se un arco (u, v) non appartiene ad un ciclo, allora u viene lista prima di v nella sequenza ordinata;
- gli archi di un ciclo vengono listati in qualche ordine (che è ininfluente).

Utilizziamo quindi la procedura topSort per ottenere i nodi in ordine decrescente di tempo di fine.

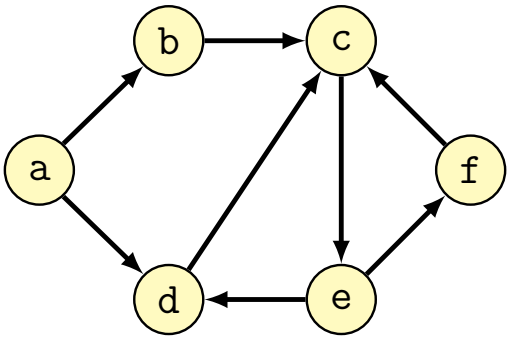
Nella topSort non calcoliamo nemmeno i tempi di fine, li utilizziamo semplicemente per dimostrare che i nodi vengono ordinati in ordine inverso di tempo di fine.

Definizione 10.5.3 (Grafo trasposto)**.** Dato un grafo orientato $G = (V, E)$, il grafo trasposto $G_t = (V, E_T)$ ha gli stessi nodi e gli archi orientati in senso opposto: $E_T = \{(u, v) \mid (v, u) \in E\}$

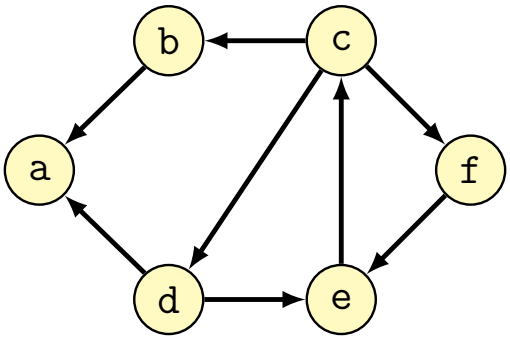
Algoritmo 10.5.2: Calcolo del grafo trasposto

```
// restituisce il grafo transposto
int[] transpose(GRAPH G)
    GRAPH G^T ← Graph // creo il grafo
    foreach u ∈ G.V do
        G^T.insertNode(u) // aggiungo gli stessi nodi
    foreach u ∈ G.V do
        foreach v ∈ G.adj(u) do
            G^T.insertEdge(v, u) // li aggiungo in ordine inverso
    // restituisco il grafo trasposto
    return G^T
```

Complessità Il costo computazionale totale ammonta a $\mathcal{O}(m+n)$, in quanto aggiungere i nodi costa $\mathcal{O}(n)$, gli archi $\mathcal{O}(m)$ ed ogni operazione costa $\mathcal{O}(1)$.



(a) Grafo originale



(b) Grafo trasposto

Algoritmo 10.5.3: Identificazione delle componenti connesse alternativa

```
// parte iterativa
int[] cc(Graph G, Stack S)
    // creo un vettore della dimensione dei nodi del grafo
    int[] id ← new int[1...G.size]
    // inizializzo il vettore
    foreach u ∈ G.V do
        id[u] ← 0
    // contatore delle componenti connesse
    int counter ← 0
    while not S.isEmpty do // fintanto che la pila non è vuota
        u ← S.pop
        if id[u] == 0 then // ho trovato una nuova componente connessa
            counter++ // aggiorno il contatore
            // effettuo una chiamata ricorsiva sul nodo scoperto
            ccdfs(G, counter, u, id)
    // restituisco l'identificativo della componente connessa
    return id
// visita ricorsiva di ciascuna componente
ccdfs(Graph G, int counter, Node u, int[] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u:       il nodo che sto visitando
    // id:      l'identificativo della componente
    // memorizzo l'identificativo della cc
    id[v] ← counter
    foreach v ∈ G.adj(u) do // per ciascun nodo adiacente
        if id[v] == 0 then // non è ancora stato visitato
            // v:  il nodo su cui vado ad operare
            ccdfs(G, counter, v, id) // lo visito ricorsivamente
```

Invece di esaminare i nodi in ordine arbitrario, questa versione di cc li esamina nell'ordine LIFO memorizzato nello stack.

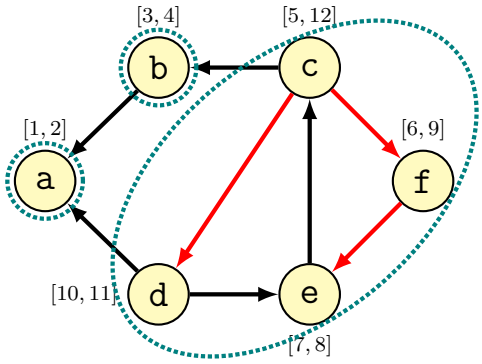


Figura 10.26: Identificazione delle componenti fortemente connesse; l'ordine in cui li visito è quello della pila `{ a, b, c, e, d, f }`

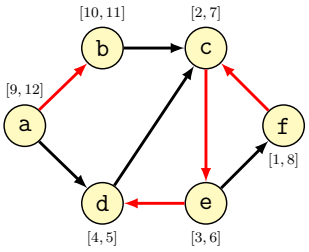
Algoritmo 10.5.4: Identificazione delle componenti fortemente connesse

```
// indentifica le componenti fortemente connesse
int[] scc(GRAPH G)
    STACK S ← topSort(G)  // prima visita
    G^T ← transpose(G)  // trasposizione del grafo
    return cc(G^T, S)  // seconda visita
```

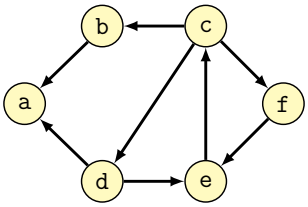
Complessità Ognuna delle fasi che compongono l'algoritmo:

1. visita in profondità della topSort;
2. la trasposizione del grafo di transpose;
3. la visita delle componenti connesse.

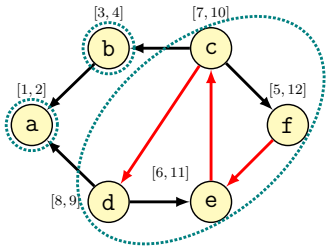
richiede un costo di $\mathcal{O}(m+n)$. Quindi la complessità è lineare nel numero di nodi e nel numero di archi, ossia $\mathcal{O}(m+n)$.



(a) La prima visita parte da f, la seconda dal nodo a



(b) Calcolo il grafo trasposto G^T



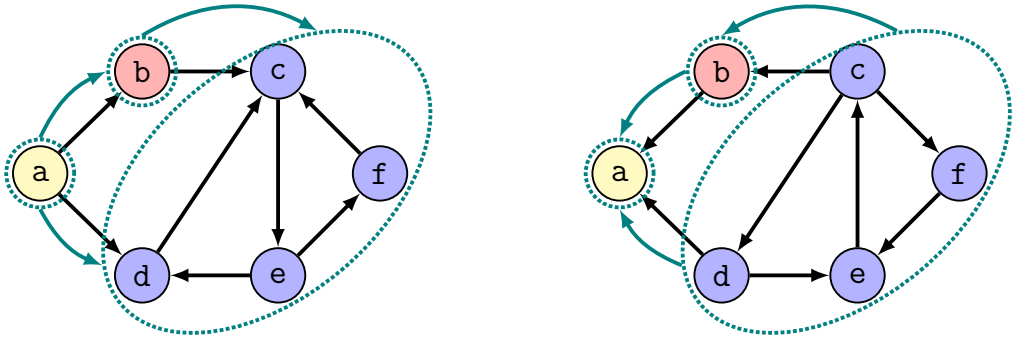
(c) Identificazione delle componenti fortemente connesse

Figura 10.27: Una seconda esecuzione dell'ordinamento topologico

10.5.2 Dimostrazione di correttezza

Definizione 10.5.4 (Grafo delle componenti)**.** Il grafo delle componenti si definisce come il grafo $C(G) = (V_c, E_c)$, dove:

- $V_c = \{C_1, C_2, \ldots, C_k\}$, dove C_i è la i-esima componente fortemente connessa del grafo G;
- $E_c = \{(C_i, C_j) \mid \exists (u_i, v_i) \in E : u_i \in C_i \wedge u_j \in C_j\}$



(a) Grafo delle componenti del grafo

(b) Grafo delle componenti del grafo trasposto

Figura 10.28

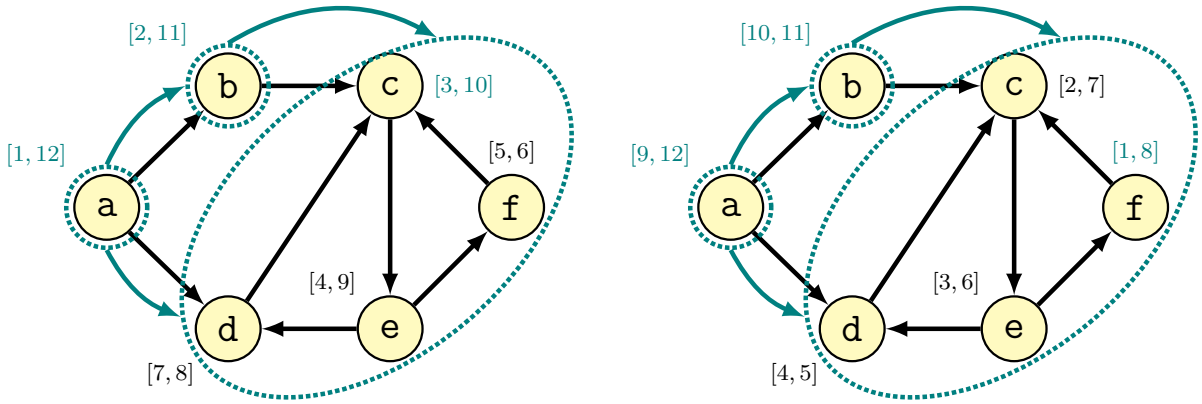
Quando si traspone un grafo fortemente connesso, l'insieme di nodi che compongono il grafo delle componenti connesse rimane lo stesso, mentre gli archi sono in direzione inversa.

Nota. Il grafo delle componenti è aciclico poiché se contenesse un ciclo, il ciclo stesso sarebbe una più grande componente fortemente connessa, il che è assurdo.

Se il grafo delle componenti è aciclico, allora posso farne un'ordinamento topologico. Possiamo definire quindi $dt(C) = \min\{dt(u) \mid u \in C\}$ e $ft(C) = \max\{ft(u) \mid u \in C\}$, i quali corrisponderanno ai tempo di inizio e di fine del primo nodo visitato in C.

Teorema. *Siano C e C' due distinte componenti fortemente connesse nel grafo orientato $G = (V, E)$. Se c'è un arco $(C, C') \in E_c$, allora $ft(C) > ft(C')$.*

La componente che finisce la visita per ultima è la componente dalla quale si possono raggiungere le altre componenti.



(a) Grafo delle componenti del grafo

(b) Grafo delle componenti del grafo trasposto

Corollario. Siano C_u e C_v due componenti fortemente connesse distinte del grafo orientato $G = (V, E)$. Se c'è un arco $(u, v) \in E_t$ tale che $u \in C_u$ e $v \in C_v$, allora $ft(C_u) < ft(C_v)$.

In generale:

$$(u,v) \in E_t \Rightarrow (v,u) \in E \Rightarrow (C_v, C_u) \in E_c \Rightarrow ft(C_v) > ft(C_u) \Rightarrow ft(C_u) < ft(C_v)$$

Nel nostro caso:

$$(b,a) \in E_t \Rightarrow (a,b) \in E \Rightarrow (C_a, C_b) \in E_c \Rightarrow \underset{12}{ft(C_a)} > \underset{11}{ft(C_b)} \Rightarrow \underset{11}{ft(C_b)} < \underset{12}{ft(C_a)}$$

Se la componente C_u e la componente C_v sono connesse da un arco $(u, v) \in E_t$, allora possiamo dedurne che $ft(C_u) < ft(C_v)$ (dal corollario) e che la visita di C_v inizierà prima della visita di C_u (dal teorema). Non esistendo cammini fra C_v e C_u in G_t (altrimenti il grafo sarebbe ciclico) la visita di C_v non raggiungerà C_u. In altre parole, l'algoritmo `cc` assegnerà correttamente gli indentificatori delle componenti ai nodi.

Algoritmo di Tarjan

L'algoritmo di Tarjan è preferito a quello di Kosaraju il quale, avendo comunque la medesima complessità computazionale ($\mathcal{O}(m+n)$), è preferito a quest'ultimo in quanto necessita di una sola visita e non richiede la trasposizione del grafo (al posto di una doppia visita e di memoria aggiuntiva).

Applicazioni

Gli algoritmi sulle componenti fortemente connesse possono essere utilizzati per risolvere il problema "2-satisfiability" (2-SAT), un problema di soddisfacibilità booleana con clausole composte da coppie di letterali.

Capitolo 11

Strutture dati speciali

Capitolo 12

Divide et Impera

12.1 Risoluzione di problemi

Dato un problema non esistono "ricette originali" per risolverlo in modo efficiente; tuttavia è possibile evidenziare quattro fasi:

1. **classificazione del problema**: è il primo passo verso la risoluzione;
2. **caratterizzazione della soluzione**: bisogna caratterizzare matematicamente la soluzione, evitando di escludere soluzioni banali;
3. **tecnica di progetto**: quando è possibile dividere il problema in più sottoproblemi di complessità minore allora la tecnica "divide et impera" potrebbe essere quella più appropriata (più avanti vedremo delle tecniche più interessanti quali: programmazione dinamica (Capitolo 13), algoritmi ingordi (Capitolo 14) e backtrack (Capitolo 16));
4. **utilizzo di strutture dati**: bisogna scegliere la struttura dati più adatta alla risoluzione del nostro particolare problema (spesso sarà una tabella hash o un albero binario di ricerca, più avanti vedremo delle strutture dati specializzate per risolvere problemi specifici, a differenza di quelle che abbiamo visto fin'ora che sono generiche).

Queste fasi non sono necessariamente sequenziali, l'ordine dipende da come stiamo affrontando il problema.

12.1.1 Classificazione dei problemi

Ma come possiamo classificare un problema? Le classi di problemi che affronteremo possono essere raggruppate in quattro macro-categorie:

- **problemi decisionali**: consistono nel determinare se il dato in ingresso soddisfa o meno una certa proprietà ed hanno una risposta binaria (si/no, true/false); come ad esempio stabilire se un grafo risulta connesso o meno. Su questo genere di problemi spesso non esistono delle tecniche standard e bisogna creare algoritmi ad-hoc;
- **problemi di ricerca**: consistono nel trovare nello spazio di soluzioni possibili una soluzione ammissibile che rispetti certi vincoli, come ad esempio la ricerca della posizione di una sottostringa in una stringa. In questi problemi la tecnica "divide et impera" può rincorrere in nostro aiuto;
- **problemi di ottimizzazione**: ad ogni soluzione è associata una funzione di costo e vogliamo trovare quella di costo minimo, come ad esempio il cammino (pesato) più breve fra due nodi. Questa classe di problemi può essere risolta tramite la programmazione dinamica o algoritmi ingordi;
- **problemi di approssimazione**: a volte, trovare la soluzione ottima è computazionalmente impossibile e ci si accontenta di una soluzione approssimata, in questo caso il costo rimane basso ma non sappiamo se è ottimale; un esempio di questo genere di problemi è quello del commesso viaggiatore.

12.1.2 Caratterizzazione della soluzione

È fondamentale definire bene il problema dal punto di vista matematico. La formulazione del problema può suggerire una prima idea, seppur banale, alla risoluzione dello stesso. Lo si può osservare nella formulazione del seguente problema: data una sequenza di n elementi, una permutazione ordinata è data dal minimo

seguito da una permutazione ordinata dei restanti $n-1$ elementi. Questa formulazione produce l'algoritmo selectionSort. La definizione matematica può suggerire una possibile tecnica, ad esempio:

- se troveremo una *sottostruttura ottima* allora potremmo applicare la programmazione dinamica (Capitolo 13);
- se troveremo la *proprietà greedy* allora potremmo applicare un algoritmo ingordo (Capitolo 14).

Tecniche di soluzione dei problemi

Come vengono affrontati i problemi dalle varie tecniche?

- nella tecnica divide-et-impera un problema viene suddiviso in sotto-problemi indipendenti, i quali vengono risolti ricorsivamente (avendo quindi un approccio dall'alto verso il basso, detto *top-down*); Abbiamo già visto diversi esempi dell'applicazione di questa tecnica, provate a pensare all'algoritmo mergeSort: ordinare due sottovettori sono due problemi indipendenti (ordinare il sottovettore di sinistra non richiede conoscere il contenuto del vettore di destra e viceversa);
- nella programmazione dinamica la soluzione viene costruita (dal basso verso l'altro, *bottom-up*) a partire da un insieme di sotto-problemi potenzialmente ripetuti.
- la tecnica della *memoization* (annotazione) è la versione *top-down* della programmazione dinamica.
- la tecnica *greedy* effettua sempre la scelta localmente ottima (necessita di una dimostrazione).
- il backtrack procede per "tentativi", tornando ogni tanto sui suoi passi;
- nella ricerca locale la soluzione ottima viene trovata "migliorando" via via soluzioni esistenti; Negli algoritmi probabilistici si dimostra che talvolta è meglio scegliere casualmente, ma in modo "gratuito", che con giudizio, ma in maniera costosa.

12.2 La tecnica del Dividi-et-Impera

La tecnica del Divide-et-Impera si suddivide in tre fasi principali:

- **Divide**: divide il problema in sotto-problemi più piccoli è indipendenti;
- **Impera**: risove i sottoproblemi ricorsivamente;
- (**Combina**): "unisce" le soluzioni dei sottoproblemi.

Sfortunatamente non esiste una ricetta unica per applicare questa tecnica: ad esempio l'algoritmo mergeSort ha una fase "divide" banale (basta calcolare il valore mediano) ma, allo stesso tempo una fase di unione delle soluzioni complessa, diversamente nel quickSort la fase "divide" è complessa ma non esiste una fase "combina". È quindi necessario fare uno sforzo creativo, in quanto la tecnica ci dà una modalità con cui ad arrivare alla soluzione, ma bisogna applicarla caso per caso.

Minimo divide-et-impera

La tecnica "divide-et-impera" non è un proiettile d'argento, a volte utilizzarla crea più danni di quanti ne risolva. Osserva questo esempio nella quale è presentato un algoritmo di ricerca del minimo con questa tecnica.

Algoritmo 12.2.1: Algoritmo di ricerca del minimo con tecnica divide-et-impera

```
minrec(int[] A, int i, int j)
    if i == j then
        return A[i]
    else
        m ← ⌊(i+j)/2⌋
        return min(minrec(A, i, m), minrec(A, m + 1, j))
```

Complessità L'algoritmo divide il vettore a metà, cerca il minimo nella metà di sinistra e nella metà di destra, il risultato è il minimo dei due minimi.

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

La complessità ammonta a $\alpha = 1, \beta = 0, T(n) = \Theta(n)$, non ne vale la pena, tanto vale fare la ricerca del minimo come abbiamo spiegato a lezione.

12.3 La torre di Hanoi

La torre di Hanoi è un gioco matematico che prevede tre pioli e n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è quello di impilare i dischi sul piolo di destra, senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta ed utilizzando il piolo centrale come appoggio. Questo problema può essere risolto tramite la tecnica "divide-et-impera".

Algoritmo 12.3.1: Versione ricorsiva della soluzione al problema della torre di Hanoi

```
hanoi(int n, int src, int dest, int middle)
    if n = 1 then
        stampa src → dest
    else
        hanoi(n − 1, src, middle, dest)  // sposta n − 1 dischi da src a middle
        stampa src → dest  // sposta 1 disco da src a dest
        hanoi(n − 1, middle, dest, src)  // sposta n − 1 dischi da middle a dest
```

Nella prima parte l'algoritmo sposta $n-1$ dischi da *src* a *middle* utilizzando *dest* come punto d'appoggio. Dopodiché sposta l'ultimo disco rimanente dalla *src* alla *dest*. Infine sposta $n-1$ dischi da *src* a *dest* utilizzando *src* come punto d'appoggio.

Complessità L'equazione di ricorrenza prodotta da questo algoritmo è $T = 2T(n-1) + 1 = \Theta(2^n)$. Si può dimostrare che questa soluzione è ottima (non si può fare meglio di così).

12.4 Algoritmo di ordinamento

L'algoritmo di ordinamento quickSort è basato sulla tecnica "divide et impera", nel caso medio ha una complessità di $\mathcal{O}(n \log n)$, mentre nel caso pessimo è di $\mathcal{O}(n^2)$. Fino a qualche anno fa era l'algoritmo di eccellenza per l'ordinamento. Infatti presenta molti aspetti a suo favore:

- il fattore costante del quickSort è migliore di quello del mergeSort;

- non utilizza memoria addizionale in quanto svolge i calcoli "in-memory" (a differenza di mergeSort che ha bisogno di un vettore di appoggio);
- esistono delle tecniche "euristiche" per evitare il caso pessimo.

Quindi spesso è preferito ad altri algoritmi. All'interno dell'ultimo capitolo riassumeremo tutti gli algoritmi di ordinamento visti fin'ora e ne vedremo di nuovi, tra questi anche gli algoritmi attualmente utilizzati negli attuali linguaggi di programmazione (c, java, python).

Spiegazione Sono dati in input un vettore $A[1 \dots n]$, gli indici *start*, *end* tali che $1 \leqslant start \leqslant end \leqslant n$, tali indici indicano quale parte del vettore stiamo ordinando, come avviene in mergeSort.

1. la parte del "divide" avviene nel seguente modo:
   - scegliamo un valore $p \in A[start \dots end]$ detto perno (*pivot*);
   - spostiamo gli elementi del vettore $A[start \dots end]$ in modo tale che:
     - $\forall i \in [start \dots j-1] : A[i] \leqslant p$;
     - $\forall i \in [j+1 \dots end] : A[i] \geqslant p$

     l'indice j viene calcolato per rispettare tale condizione;
   - il perno viene messo in posizione $A[j]$.
2. la parte "impera" ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente quickSort;
3. la parte "combina" non fa nulla.

Algoritmo 12.4.1: quickSort

```
quickSort(ITEM[] A, int primo, int ultimo)
    // su almeno due elementi
    if primo < ultimo then
        int j ← perno(A, primo, ultimo)  // logica dell'algoritmo

        // richiamo l'algoritmo su entrambi i sottovettori
        quickSort(A, primo, j − 1)
        quickSort(A, j + 1, ultimo)
```

Algoritmo 12.4.2: perno

```
// sposta gli elementi più piccoli a sinistra del perno, i più grandi a destra
int perno(ITEM[] A, int primo, int ultimo)
    ITEM x ← A[primo] // il perno è il primo elemento
    int j ← primo // il cursore parte dal primo elemento

    // spostamenti "in-place"
    from i ← primo until ultimo do
        if A[i] < x then // l'elemento è più piccolo del perno
            j++ // sposta il cursore j
            A[i] ↔ A[j] // scambia gli elementi: i ↔ j

    /* a questo punto tutti gli elementi posizionati prima della posizione j sono più piccoli del perno,
       rimane solo da riposizionare il perno nella sua posizione finale (è ordinato)                    */

    // riposiziono il perno
    A[primo] ← A[j]
    A[j] ← x

    // restituisco la posizione del perno
    return j
```

Complessità computazionale Il costo della funzione perno è $\Theta(n)$ (deve guardare $n-1$ valori ed effettuare i confronti). Il costo di quickSort dipende dal partizionamento:

- il partizionamento *peggiore* si verifica quando il perno è l'elemento minimo (o massimo), questo particolare caso accade quando il vettore è ordinato in ordine crescente (decrescente). La complessità risultante è $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.
- il partizionamento *migliore* avviene quando il vettore di dimensione n viene diviso in due sottoproblemi di dimensione $n/2$. La complessità risultante è $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$;
- il partizionamento nel *caso medio* è molto più vicino al caso ottimo che al caso peggiore, prendiamo ad esempio il partizionamento 9-a-1:

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

Prendiamo un altro esempio, il partizionamento 99-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{99n}{10}\right) + cn = \Theta(n \log n)$$

Nota. In questi esempi, il partizionamento ha proporzionalità limitata e i fattori moltiplicativi possono essere importanti.

Il costo computazionale dipende dall'ordine degli elementi e non dai loro valori. Dobbiamo quindi considerare tutte le possibili permutazioni, il che è difficile dal punto di vista analitico. Alcuni partizionamenti saranno parzialmente bilanciati, altri pessimi; in media questi si alterneranno nella sequenza di partizionamenti, ma quelli parzialmente bilanciati "dominano" quelli pessimi.

Moltiplicazione di catena di matrici

Viene fatto un accenno ad un argomento che verrà affrontato in modo più approfondito nel capitolo successivo.

12.5 Conclusioni

La tecnica divide-et-impera viene applicata quando i passi "divide" e "combina" sono semplici e i costi risultano migliori del corrispondente algoritmo iterativo (quindi, ad esempio, va bene per effettuare l'ordinamento, ma non per effettuare la ricerca del minimo). Ulteriori vantaggi dell'applicazione di questa tecnica sono:

- la facile parallelizzazione: la possibilità di dividere il problema in più sottoproblemi porta ad una naturale divisione dei compiti fra più processori;
- l'utilizzo ottimale della memoria *cache* (*cache oblivious*): tutti i dati con la quale stiamo lavorando sono colocalizzati nella memoria principale.

12.6 Applicazione della tecnica

Infine vediamo una prima applicazione della tecnica e ne valutiamo le prestazioni.

12.6.1 Gap

In un vettore V contenente $n \geqslant 2$ interi, un gap è un indice i, $1 < i \leqslant n$, tale che $V[i-1] < V[i]$.

- Dimostrare che se $n \geqslant 2$ e $V[1] < V[n]$, allora V contiene almeno un gap;
- Progetta un algoritmo che, dato un vettore V contentente $n \geqslant 2$ interi e tale che $V[1] < V[n]$ (la condizione sopra), restituisca la posizione di un gap nel vettore (questo algoritmo assume che il gap esista).

Dimostrazione per assurdo. Suppponiamo che non ci sia un gap nel vettore. Allora $V[1] \geqslant V[2] \geqslant V[3] \geqslant \cdots \geqslant V[n]$, che contraddice il fatto che $V[1] < V[n]$. □

Proviamo a riformulare la proprietà tenendo conto di due indici:

- sia V un vettore di dimensione n;
- siano i, j due indici tali che $1 \leqslant i < j \leqslant n$ e $V[i] < V[j]$.

In altre parole, ci sono più di due elementi nel sottovettore $V[i \dots j]$ e il primo elemento $V[i]$ è più piccolo dell'ultimo elemento $V[j]$.

Dimostrazione per induzione. Voglia provare per induzione sulla dimensione n del sottovettore che il sottovettore contiene un gap.

- **caso base**: $n = j - i + 1 = 2$, ad esempio $j = i + 1$: $V[i] < V[j]$ implica che $V[i] < V[j]$ implica che $V[i] < V[i+1]$, che è un gap;
- **ipotesi induttiva**: dato un qualunque (sotto)vettore $V[h \dots k]$ di dimensione $n' < n$, tale che $V[h] < V[k]$, allora $V[h \dots k]$ contiene un gap;
- **passo induttivo**: consideriamo un qualunque elemento m tale che $i < m < j$. Almeno uno dei due casi seguenti è vero:
  - se $V[m] < V[j]$, allora esiste un gap in $V[m \dots j]$, per ipotesi induttiva;
  - se $V[i] < V[m]$, allora esiste un gap in $V[i \dots m]$, per ipotesi induttiva.

□

Algoritmo 12.6.1: Algoritmo che ricerca un intervallo all'interno del vettore

```
// funzione wrapper
gap(int[] V, int n)
    // n:       dimensione del vettore
    return gapRec(V, 1, n)

gapRec(int[] V, int i, int j)
    if j == i + 1 then // ho due elementi
        return j  // ritorno il secondo elemento

    m = ⌊(i+j)/2⌋  // calcolo il mediano

    if V[m] < V[j] then
        return gapRec(V, m, j)  // a destra
    else
        return gapRec(V, i, m)  // a sinistra
```

Tabella 12.1: Valutazione delle prestazioni degli algoritmi scritti in python

n	Iterativa (ms)	Ricorsiva (μs)
10^3	0,06	2,05
10^4	0,61	2,78
10^5	6,11	3,36
10^6	62,44	4,01
10^7	621,69	4,87
10^8	6205,72	5,47

Parte II

Secondo semestre

Capitolo 13

Strutture dati speciali

Capitolo 14

Scelta della struttura dati

Nota. La prima cosa da fare quando si progetta un algoritmo è capire quale sia la struttura dati adatta a risolvere quel particolare problema.

14.1 Cammini minimi, sorgente singola

14.1.1 Problema dei cammini minimi

Definizione 14.1.1 (Costo del cammino)**.** Dato un cammino $p = \langle v_1, v_2, \ldots, v_k \rangle$ con $k > 1$, il *costo del cammino* (*weight of the path*) è dato da

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Ossia dalla somma dei singoli pesi dei lati che compongono il percorso.

Definizione del problema Dati in input un grafo orientato $G = (V, E)$, un nodo sorgente s ed una funzione di peso $w: E \to R$ (che associa ad ogni arco un numero reale che rappresenta il peso).

Trovare un cammino da s ad u, per ogni nodo $u \in V$, il cui costo sia minimo, ovvero più piccolo o uguale al costo di qualunque altro cammino da s a u.

Nota. Non ci limitiamo a trovare un solo percorso, ma tutti i cammini da un nodo a tutti gli altri nodi.

Panoramica sul problema Per risolvere il problema del cammino minimo fra una coppia di vertici, si risolve il problema di cammini minimi da sorgente unica (si trovano tutti i cammini che partono da un nodo) e si estrae il cammino richiesto. Per quanto riguarda il *caso pessimo* non si conoscono algoritmi che abbiamo tempo di esecuzione migliore.

In alcuni casi gli archi possono avere peso negativo. Questo influisce sul problema (se è ben definito oppure no) e sulla soluzione (in assenza di archi negativi si (possono?) devono utilizzare tecniche diverse).

Nell'algoritmo di Dijkstra si suppone che tutti gli archi abbiano peso positivo, mentre nell'algoritmo di Bellman-Ford gli archi possono avere peso negativo, ma non possono esistere cicli di peso negativo.

Nota. In generale possiamo ammettere pesi negativi, ma non cicli negativi.

Considerazioni sui cicli Se esiste un ciclo di peso negativo raggiungibile dalla sorgente, non esistono cammini finiti di peso minimo; per qualunque cammino, basterà passare per un ciclo negativo più volte per ottenere un ciclo di costo inferiore.

Ovviamente, in un cammino minimo *non è possibile la presenza di un ciclo di peso positivo.* Mentre i cicli di peso nullo possono essere banalmente eliminati dal cammino minimo, in quanto inutili e ridondanti.

14.1.2 Sottostruttura ottima

Nota che due cammini minimi possono avere un tratto in comune, ma non possono convergere in un nodo comune B dopo aver percorso un tratto iniziale distinto.

(a) Due cammini minimi che hanno un tratto in comune a partire dal nodo B

(b) Due cammini che convergono in un nodo comune B dopo aver percorso un tratto iniziale distinto.

Figura 14.1: La figura 14.1a è una condizione ammissibile, mentre 14.1b non lo è.

Definizione 14.1.2 (Albero dei cammini minimi)**.** L'albero dei cammini minimi è un albero di copertura radicato in s avente un cammino da s a tutti i nodi raggiungibili da s.

Nota. Non confonderlo con gli alberi di copertura di peso minimo.

Soluzione ammissibile Una soluzione *ammissibile* può essere descritta da un *albero di copertura* T radicato in s e da un *vettore delle distanze* d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T.

(a) Soluzione ammissibile (albero di peso minimo)

(b) Soluzione ottima (albero dei cammini minimi)

Figura 14.2: Nota la differenza fra i due alberi: a sinistra l'albero di *peso minimo* che rappresenta una soluzione ammissibile, ma non ottima, mentre a destra l'albero dei cammini minimi, ossia l'insieme dei percorsi che minimizzano il peso fra il nodo sorgente e tutti gli altri nodi, il quale rappresenta la soluzione ottima.

Nota. In questo problema devo trovare i percorsi che minimizzano il peso fra un nodo e tutti gli altri nodi, e **non**, come sembra spontaneo fare, l'albero che ha complessivamente peso minimo.

Rappresentazione dell'albero Utilizziamo la rappresentazione basata su vettore dei padri, così come abbiamo fatto con le visite in ampiezza/profondità.

14.1.3 Teorema di Bellman

Teorema 10 (Teorema di Bellman)**.** *Una soluzione ammissibile T è (anche) ottima se e solo se:*

$$d[v] = d[u] + w(u,v) \text{ per ogni arco } (u,v) \in \boldsymbol{T}$$
$$d[v] \leqslant d[u] + w(u,v) \text{ per ogni arco } (u,v) \in \boldsymbol{E}$$

Dimostrazione per assurdo (parte 1). Sia T una soluzione ottima. Consideriamo un qualunque arco $(u,v) \in E$ e sia $w(u,w)$ la sua lunghezza.

Ovviamente se $(u,v) \in T$, allora $d[v] = d[u] + w(u,v)$. Invece se $(u,v) \notin T$, allora poiché T è ottimo, deve risultare $d[v] \leqslant d[u] + w(u,v)$, altrimenti esisterebbe nel grafo G un cammino da s a v più corto di quello in T, che è *assurdo* perché abbiamo ipotizzato che T fosse ottima. □

Dimostrazione per assurdo (parte 2). Supponiamo per assurdo che il cammino da s a u in T non sia ottimo. Allora esiste un cammino da s a u con distanza $d'[u] < d[u]$. Sia $d'[v]$ la distanza da s ad un generico

nodo v che appare in tale cammino. Poichè $d'[s] = d[s] = 0$, ma $d'[u] < d[u]$, esiste un arco (h,k) per cui $d'[h] \geqslant d'[h]$ e $d'[k] < d[k]$. Per costruzione $d'_h + w(h,k) = d'_k$. Per ipotesi $d_h + w(h,k) \geq d_k$. Combinando queste due relazioni, si ottiene:

$$d'_k = d'_h + w(h,k) \geqslant d_h + w(h,k) \geqslant d_k$$

che contraddice l'ipotesi. □

14.1.4 Verso un algoritmo

Algoritmo prototipo per il calcolo dei cammini minimi

```
// Algoritmo prototipo dei cammini minimi
(int, int) CamminiMinimi(GRAPH G, NODE s)
    // Inizializza T ad una foresta di copertura composta da nodi isolati
    // Inizializza d con una sovrastima della distanza (d[s] = 0, d[x] = +∞)
    while ∃(u,v): d[u] + G.w(u,v) < d[v] do
        // Esiste un arco che mi permette di migliorare la stima
        d[v] = d[u] + w(u,v) // Aggiorno la distanza
        // Sostituisci il padre di v in T con u
    return (T, d)
```

Commento L'algoritmo prende in input un grafo e il nodo sorgente. I pesi vengono estratti dalla struttura dati GRAPH. Inizializziamo d con una sovrastima della distanza; $d[s] = 0$ sta a significare che la sorgente ha distanza da sè stessa pari a 0 (caso base) e con $d[x] = +\infty$ indico che la distanza di tutti gli altri nodi, fintanto che non è nota, è pari a $+\infty$.

Nota. Se al termine dell'esecuzione dell'algoritmo qualche nodo mantiene una distanza infinita, allora esso non è raggiungibile dalla sorgente.

Algoritmo generico per il calcolo dei cammini minimi

```
(int, int) CamminiMinimi(GRAPH G, NODE s)
    // Inizializzazione dei vettori
    int[] d ← new int[1...G.n]                          // distanze dalla sorgente
    int[] T ← new int[1...G.n]                          // vettore dei padri
    boolean[] b ← new boolean[1...G.n]                  // per sapere in tempo costante se u ∈ S

    // Inizializzo tutti i nodi tranne la sorgente
    foreach u ∈ G.V − {s} do
        T[u] ← nil  // non hanno padri
        d[u] ← +∞  // non li ho ancora raggiunti
        b[u] ← false  // non appartengono ancora all'insieme

    // Inizializzo la sorgente
    T[s] ← nil // non ha padre
    d[s] ← 0 // per convenzione
    b[s] ← true // appartiene all'insieme
(7) STRUTTURADATI S ← StrutturaDati
    S.aggiungi(s)

    while not S.isEmpty do
(8)     int u ← S.estrai  // estraggo un nodo
        b[]u ← false  // non è più contenuto nella struttura dati

        foreach v ∈ G.adj(u) do // per tutti i vicini
            if d[u] + G.w(u,v) < d[v] then // se migliora la stima
                if not b[v] then // se non fa già parte dell'insieme
(9)                 S.aggiungi(v) // aggiungilo
                    b[v] ← true // fa parte dell'insieme
                else
(10)                // Azione da intraprendere nel caso v sia già presente in S

                // aggiorno i vettori
                T[v] ← u
                d[v] ← d[u] + G.w(u,v)

    return (T, d)
```

Commento **boolean**[] ci permette di sapere in tempo costante se un certo nodo appartiene ad una struttura dati oppure no, non sarà necessario quando implementeremo realmente il codice.

14.2 Algoritmo di Dijkstra

Il seguente algoritmo è stato sviluppato da Edsger W. Dijkstra nel 1956, pubblicato nel 1959. Nella versione originale veniva utilizzato per trovare la distanza minima fra due nodi sfruttando il concetto di coda con priorità. Tieni conto però che le code di priorità basate sugli heap binari sono state proposte nel '64, infatti l'algoritmo che di solito viene considerato di Dijkstra è in realtà la versione modificata di Johnson.

Implementazione L'algoritmo utilizza una coda con priorità basata su vettore.

Algoritmo 14.2.1: Algoritmo di Dijkstra

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
(1)  PRIORITYQUEUE S ← PriorityQueue  // O(n)·1
     S.inserisci(s, 0)
     while not S.isEmpty do // O(n)
(2)      // O(n) vettore ordinato / O(log n) heap binario
         int u ← S.deleteMin
         b[u] ← false
         foreach v ∈ G.adj(u) do
             if d[u] + G.w(u,v) < d[v] then
                 if not b[v] then
                     // O(1)·n vettore ordinato / O(log n)·n heap binario
(3)                  S.inserisci(v, d[u] + G.w(u,v))
                     b[v] ← true
                 else
                     // O(1)·m vettore ordinato / O(log n)·m heap binario
(4)                  S.decrease(v, d[u] + G.w(u,v))
                 // aggiorno i vettori
                 T[v] ← u
                 d[v] ← d[u] + G.w(u,v)
     return (T, d)
```

Analisi della complessità

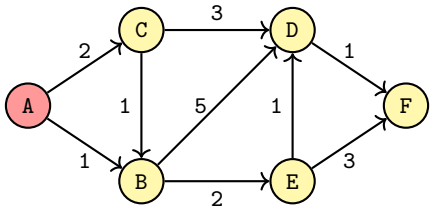
① Viene creato un vettore di dimensione n. Ogni elemento u-esimo rappresenta il nodo u. Le priorità (distanze) vengono inizializzate ad $+\infty$. La priorità di s è posta uguale a 0. Per un costo di $O(n)$;

② Si ricerca il minimo all'interno del vettore, una volta trovato si "cancella" la sua priorità. Per un costo complessivo di $O(n)$ (viene svolto all'interno di un ciclo);

③ Si registra la priorità nella posizione corrispondente all'indice v. Per un costo di $O(1)$;

④ Si aggiorna la priorità nella posizione corrispondente all'indice v. Per un costo di $O(1)$.

Esempio di esecuzione



		A	B	C	D	E	F
A	0	**0**	~~∅~~	~~∅~~	~~∅~~	~~∅~~	~~∅~~
B	∞	1	**1**	~~1~~	~~1~~	~~1~~	~~1~~
C	∞	2	2	**2**	~~2~~	~~2~~	~~2~~
D	∞	∞	6	5	4	**4**	4
E	∞	∞	3	3	**3**	~~3~~	~~3~~
F	∞	∞	∞	∞	6	5	**5**

- ogni colonna contiene lo stato del vettore d all'inizio di ogni ripetizione del ciclo while **not** S.isEmpty
- ogni riga v rappresenta l'evoluzione dello stato dell'elemento $d[v]$;

- la legenda delle colonne rappresenta il nodo che viene estratto.

Correttezza Tutte le volte che estraiamo un nodo, quel nodo ha una distanza (priorità) positiva ed estraiamo nodi a distanza progressivamente crescenti. Se estraggo un nodo dalla coda tutti gli altri nodi hanno distanze più grandi. Tutte le volte che estraggo un nodo la sua distanza non può più essere modificata. Ed è questo il motivo per cui l'algoritmo di Dijkstra funziona (bene) solo con pesi positivi.

Nota. L'algoritmo di Dijkstra funziona correttamente solo con pesi positivi.

14.2.1 Correttezza per pesi positivi

Ogni nodo viene estratto una e una sola volta. Al momento dell'estrazione la sua distanza è minima.

Dimostrazione per induzione sul numero k di nodi estratti. Per $k = 0$ (caso base) è vero poiché $d[s] = 0$ e non ci sono lunghezze negative. Supponiamo che sia vero per i primi $k-1$ nodi (ipotesi induttiva). Quando viene estratto il k-esimo nodo u, la sua distanza $d[u]$ dipende dai $k-1$ nodi già estratti (passo induttivo). Non può quindi dipendere dai nodi ancora da estrarre, che hanno distanza $\geqslant d[u]$. Di conseguenza $d[u]$ è minimo e u non verrà più re-inserito, perché non ci sono distanze negative. □

14.3 Algoritmo di Johnson

Analisi della complessità Con l'introduzione dell'heap binario nel '64 le operazioni che prima venivano svolte con complessità $\mathcal{O}(n)$ sul vettore ordinato ora hanno complessità $\mathcal{O}(\log n)$. Di conseguenza la complessità totale dell'algoritmo scende da $\mathcal{O}(n^2)$ a $\mathcal{O}(m \log n)$.

Per *grafi densi* non conviene utilizzare uno heap binario in quanto $m = \Theta(n^2)$ e di conseguenza l'algoritmo avrebbe una complessità di $\mathcal{O}(n^2 \log n)$ si preferisce quindi la versione con vettore ordinato per una complessità di $\mathcal{O}(n^2)$, mentre per *grafi sparsi* $m = \Theta(n)$ e l'algoritmo $\mathcal{O}(n \log n)$ che è migliore di $\mathcal{O}(m \log n)$.

14.4 Algoritmo di Fredman-Tarjan

Analisi della complessità Sfruttando un heap di fibonacci l'operazione di decrease ha costo ammortizzato costante; così facendo hanno abbassato la complessità a $\mathcal{O}(m + n \log n)$. Per *grafi sparsi* produce un miglioramento nella complessità.

14.5 Algoritmo di Bellman-Ford-Moore

Nota. É computazionalmente più pesante dell'algoritmo di Dijkstra ma può lavorare anche con archi di peso negativo.

Implementazione Utilizza una coda senza priorità.

Algoritmo 14.5.1: Algoritmo di Bellman-Ford-Moore

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
(1)  QUEUE S ← Queue
     S.enqueue(s)  // metto in coda il nodo sorgente
     while not S.isEmpty do // O(n)
(2)      int u ← S.dequeue  // O(1 · n)
         b[u] ← false
         foreach v ∈ G.adj(u) do
             if d[u] + G.w(u,v) < d[v] then
                 if not b[v] then
(3)                  // lo metto in cosa quando c'è un miglioramento
                     S.enqueue(v)  // O(m · n)
                     b[v] ← true
     return (T, d)
```

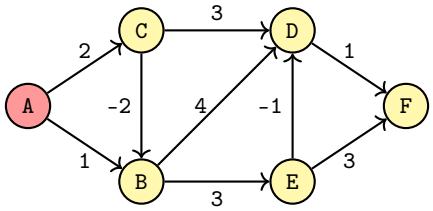
① Viene creata una coda di dimensione n. Per un costo computazionale di $O(n)$;

② Viene estratto il prossimo elemento della coda. Per un costo computazionale di $O(1)$;

③ Si inserisce l'indice v in coda. Per un costo computazionale di $O(1)$.

L'inserimento in coda può essere fatto più di una volta durante il ciclo di esecuzione dell'algoritmo, al contrario di quel che accade nell'algoritmo di Dijkstra. Il passo ④ non è necessario in quanto non c'è una priorità da aggiornare.

Esempio di esecuzione



Come leggere la tabella sottostante

- la prima riga contiene l'elemento estratto dalla coda;
- l'ultima riga contiene lo stato della coda;
- i vettori delle distanze sono rappresentate dalle colonne.

nodo estratto:			A	B	C	D	E	B	F	D	E	D	F
vettori	A	0	0	0	0	0	0	0	0	0	0	0	0
	B	∞	1	1	0	0	0	0	0	0	0	0	0
	C	∞	2	2	2	2	2	2	2	2	2	2	2
	D	∞	5	5	5	3	3	3	3	3	2	2	2
	E	∞	∞	4	4	4	4	3	3	3	3	3	3
	F	∞	∞	∞	∞	6	5	5	5	4	4	3	3
queue:	S		A	BC	CDE	DEB	EBF	BFD	FDE	DE	E	D	F	

Dimostrazione di correttezza Per dimostrare la correttezza dell'algoritmo dobbiamo dare la definizione di "passata". Una passata si definisce ricorsivamente come segue: per $k = 0$, la zeresima passata consiste nell'estrazione del nodo s dalla coda S, per $k > 0$ la k-esima passata consiste nell'estrazione di tutti i nodi presenti in S al termine della passata $k-1$-esima. Viene data solo un'intuizione, non viene fatta una dimostrazione formale. Al termine della passata k, i vettori T e d descrivono i cammini minimi di lunghezza al più k. Al termine della passata $n-1$, i vettori T e d descrivono i cammini (di lunghezza al più $n-1$).

Analisi della complessità L'inserimento in coda dei nodi avviene solo una volta per un costo di $\mathcal{O}(1)$. Un nodo può essere estratto e reinserito al massimo $n-1$ volte, per un costo di $\mathcal{O}(n^2)$. Quando avviene un miglioramento la distanza viene aggiornata, per un costo i $\mathcal{O}(n\dot{m})$. Il costo complessivo dell'algoritmo risulta quindi $\mathcal{O}(n\dot{m})$.

Cammini minimi su DAG I cammini minimi su DAG sono sempre ben definiti; anche in presenza di pesi negativi, in quanto non esistono cicli (nè tantomeno quelli negativi). É possibile rilassare gli archi *in ordine topologico*, *una volta sola*. Non essendoci cicli, non c'è modo di tornare su un nodo già visitato ed abbassare il valore della sua distanza (il suo campo d).

Si utilizza quindi l'ordine topologico.

Algoritmo 14.5.2: Algoritmo di Bellman-Ford-Moore applicato su DAG

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
    int[] d = new int[1...G.n]                          // d[u] è la distanza da s a u
    int[] T = new int[1...G.n]                          // T[u] è il padre da u nell'albero T

    // Inizializzo i vettori
    foreach u ∈ G.V − {s} do
        T[u] = nil
        d[u] = +∞

    // Inizializzo la sorgente
    T[s] = nil
    d[u] = 0

    // Effettuo l'ordinamento topologico dei nodi nel DAG
    STACK S = topSort

    // fintanto che la pila non è vuota
    while not S.isEmpty do
        u = S.pop // estraggo un nodo
        foreach v ∈ G.adj(v) do // per ogni nodo adiacente
            if d[u] + G.w(u,v) < d[v] then // se il peso è migliore di quello presente
                // aggiorno il peso
                T[v] = u
                d[v] = d[u] + G.w(u,v)

    // restituisco il vettore dei padri e il vettore delle distanze
    return (T, d)
```

14.5.1 Riassumendo

Tabella 14.1: Quale complessità preferire?

Algoritmo	Complessità	Input
Dijkstra	$\mathcal{O}(n^2)$	Pesi positivi, grafi denso
Johnson	$\mathcal{O}(m \log n)$	Pesi positivi, grafi sparso
Fredman-Tarjan	$\mathcal{O}(m + n \log n)$	Pesi positivi, grafi denso, dimensioni molto grandi
Bellman-Ford	$\mathcal{O}(m \cdot n)$ $\mathcal{O}(m+n)$	Pesi negativi DAG
BFS	$\mathcal{O}(m+n)$	Senza pesi

14.5.2 Cammini minimi, sorgente multipla

Vogliamo cercare i cammini minimi fra tutti i nodi.

Tabella 14.2: Quale complessità preferire?

Algoritmo	Complessità	Input
Pesi positivi, grafo denso	$\mathcal{O}(n \cdot n^2)$	Applicazione ripetuta (n) dell'algoritmo di Dijkstra
Pesi positivi, grafo sparso	$O(n \cdot (m \log n))$	Applicazione ripetuta dell'algoritmo di Johnson
Pesi negativi	$O(n \cdot nm)$	Applicazione ripetuta di Bellman-Ford (sconsigliata)
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di **Floyd e Warshall**
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di **Johnson per sorgente multipla**

L'algoritmo di Bellman-Ford è sconsigliato per grafi densi perché può arrivare ad avere una complessità di $\mathcal{O}(n^4)$, mentre l'algoritmo di Floyd e Warshall ha una complessità di $\mathcal{O}(n^3)$ indipendentemente dalla forma del grafo.

14.6 Algoritmo di Floyd-Warshall

Utilizza la programmazione dinamica. Ci riesce ridefinendo la definione del costo di cammino in modo tale che possa essere calcolato in modo ricorsivo.

Definizione 14.6.1 (Cammini minimi k-vincolati)**.** Sia k un valore in $\{0, \dots, n\}$. Diciamo che un cammino p^k_{xy} è un cammino minimo k-vincolato fra x ed y se esso ha il costo minimo fra tutti i cammini fra x e y che non passano per nessun vertice in $v_{k+1}, \dots, v_n$ (x e y sono esclusi dal vincolo).

Nota. Assumiamo, come abbiamo sempre fatto, che esista un ordinamento fra i nodi del grafo $v_1, v_2, \dots, v_n$.

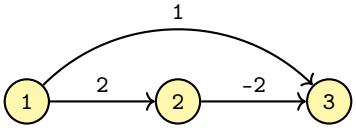
Definizione 14.6.2 (Distanza k-vincolata)**.** Denotiamo con $d^k[x][y]$ il costo totale del cammino minimo k-vincolato fra x e y, se esiste.

$$d^k[x][y] = \begin{cases} w(p^k_{xy}) & \text{se esiste } p^k_{xy} \\ +\infty & \text{altrimenti} \end{cases}$$

La formulazione ricorsiva è la seguente:

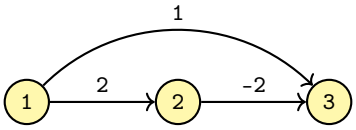
$$d^k[x][y] = \begin{cases} d[x][y] & k = 0 \\ d^{k-1}[x][y] \vee d^{k-1}[x][k] \wedge d^{k-1}[k][y] & k > 0 \end{cases}$$

Ad esempio:



$$\begin{aligned} d^0[1][3] &= 1 \\ d^1[1][3] &= 1 \\ d^2[1][3] &= \min(d^1[1][3], d^1[1][2] + d^2[2][3]) \\ &= \min(1, +2 - 2) \\ &= \min(1, 0) = 0 \end{aligned}$$

Oltre a definire la matrice d, calcoliamo una matrice T dove $T[x][y]$ rappresenta il predecessore di y nel cammino più breve da x a y. Ad esempio:



$T[1][2] = 1$
$T[2][3] = 2$
$T[1][3] = 2$

Algoritmo 14.6.1: Algoritmo di Floyd-Warshall

```
(int[], int[]) CamminiMinimi(GRAPH G, NODE s)
    // Credo le matrici
    int[][] d = new int[1...n][1...n]                    // matriche delle distanze
    int[][] T = new int[1...n][1...n]                    // matriche dei padri (predecessori)
    // Inizializzo i vettori
    foreach u, v ∈ G.V do
        d[u][v] = +∞
        T[u][v] = nil
    // Inserisco i valori iniziali
    foreach u ∈ G.V do
        foreach v ∈ G.adj(u) do
            d[u][v] = G.w(u, v)
            T[u][v] = u
    // Aggiorno le distanze
    from k ← 1 until G.n do
        foreach u ∈ G.V do
            foreach v ∈ G.adj(u) do
                if d[u][k] + d[k][v] < d[u][v] then
                    d[u][v] = d[u][k] + d[k][v]
                    T[u][v] = T[k][v]
    return d
```

14.7 Algoritmo di Warshall

Definizione 14.7.1 (Chiusura transitiva)**.** La chiusura transitiva $G^* = (V, E^*)$ di un grafo $G = (V, E)$ è il grafo orientato tale che $(u, v) \in E^*$ se e solo esiste un cammino da u a v in G.

Supponendo di avere il grafo G rappresentato da una matrice di adiacenza M, la matrice M^n rappresenta la matrice di adiacenza di G^*.

La formulazione ricorsiva è la seguente:

$$M^k[x][y] = \begin{cases} M[x][y] & k = 0 \\ M^{k-1}[x][y] \vee M^{k-1}[x][k] \wedge M^{k-1}[k][y] & k > 0 \end{cases}$$

Conclusioni

Abbiamo visto una panoramica dei più importanti algoritmi per la ricerca dei cammini minimi. Esistono anche altri algoritmi, in particolare l'algortimo A^* utilizza euristiche per velocizzare la ricerca.

Capitolo 15

Divide et Impera

15.1 Risoluzione di problemi

Dato un problema non esistono "ricette originali" per risolverlo in modo efficiente; tuttavia è possibile evidenziare quattro fasi:

1. **classificazione del problema**: è il primo passo verso la risoluzione;
2. **caratterizzazione della soluzione**: bisogna caratterizzare matematicamente la soluzione, evitando di escludere soluzioni banali;
3. **tecnica di progetto**: quando è possibile dividere il problema in più sottoproblemi di complessità minore allora la tecnica "divide et impera" potrebbe essere quella più appropriata (più avanti vedremo delle tecniche più interessanti quali: programmazione dinamica (Capitolo 13), algoritmi ingordi (Capitolo 14) e backtrack (Capitolo 16));
4. **utilizzo di strutture dati**: bisogna scegliere la struttura dati più adatta alla risoluzione del nostro particolare problema (spesso sarà una tabella hash o un albero binario di ricerca, più avanti vedremo delle strutture dati specializzate per risolvere problemi specifici, a differenza di quelle che abbiamo visto fin'ora che sono generiche).

Queste fasi non sono necessariamente sequenziali, l'ordine dipende da come stiamo affrontando il problema.

15.1.1 Classificazione dei problemi

Ma come possiamo classificare un problema? Le classi di problemi che affronteremo possono essere raggruppate in quattro macro-categorie:

- **problemi decisionali**: consistono nel determinare se il dato in ingresso soddisfa o meno una certa proprietà ed hanno una risposta binaria (si/no, true/false); come ad esempio stabilire se un grafo risulta connesso o meno. Su questo genere di problemi spesso non esistono delle tecniche standard e bisogna creare algoritmi ad-hoc;
- **problemi di ricerca**: consistono nel trovare nello spazio di soluzioni possibili una soluzione ammissibile che rispetti certi vincoli, come ad esempio la ricerca della posizione di una sottostringa in una stringa. In questi problemi la tecnica "divide et impera" può rincorrere in nostro aiuto;
- **problemi di ottimizzazione**: ad ogni soluzione è associata una funzione di costo e vogliamo trovare quella di costo minimo, come ad esempio il cammino (pesato) più breve fra due nodi. Questa classe di problemi può essere risolta tramite la programmazione dinamica o algoritmi ingordi;
- **problemi di approssimazione**: a volte, trovare la soluzione ottima è computazionalmente impossibile e ci si accontenta di una soluzione approssimata, in questo caso il costo rimane basso ma non sappiamo se è ottimale; un esempio di questo genere di problemi è quello del commesso viaggiatore.

15.1.2 Caratterizzazione della soluzione

È fondamentale definire bene il problema dal punto di vista matematico. La formulazione del problema può suggerire una prima idea, seppur banale, alla risoluzione dello stesso. Lo si può osservare nella formulazione del seguente problema: data una sequenza di n elementi, una permutazione ordinata è data dal minimo

seguito da una permutazione ordinata dei restanti $n-1$ elementi. Questa formulazione produce l'algoritmo selectionSort. La definizione matematica può suggerire una possibile tecnica, ad esempio:

- se troveremo una *sottostruttura ottima* allora potremmo applicare la programmazione dinamica (Capitolo 13);
- se troveremo la *proprietà greedy* allora potremmo applicare un algoritmo ingordo (Capitolo 14).

Tecniche di soluzione dei problemi

Come vengono affrontati i problemi dalle varie tecniche?

- nella tecnica divide-et-impera un problema viene suddiviso in sotto-problemi indipendenti, i quali vengono risolti ricorsivamente (avendo quindi un approccio dall'alto verso il basso, detto *top-down*); Abbiamo già visto diversi esempi dell'applicazione di questa tecnica, provate a pensare all'algoritmo mergeSort: ordinare due sottovettori sono due problemi indipendenti (ordinare il sottovettore di sinistra non richiede conoscere il contenuto del vettore di destra e viceversa);
- nella programmazione dinamica la soluzione viene costruita (dal basso verso l'altro, *bottom-up*) a partire da un insieme di sotto-problemi potenzialmente ripetuti.
- la tecnica della *memoization* (annotazione) è la versione *top-down* della programmazione dinamica.
- la tecnica *greedy* effettua sempre la scelta localmente ottima (necessita di una dimostrazione).
- il backtrack procede per "tentativi", tornando ogni tanto sui suoi passi;
- nella ricerca locale la soluzione ottima viene trovata "migliorando" via via soluzioni esistenti; Negli algoritmi probabilistici si dimostra che talvolta è meglio scegliere casualmente, ma in modo "gratuito", che con giudizio, ma in maniera costosa.

15.2 La tecnica del Dividi-et-Impera

La tecnica del Divide-et-Impera si suddivide in tre fasi principali:

- **Divide**: divide il problema in sotto-problemi più piccoli è indipendenti;
- **Impera**: risove i sottoproblemi ricorsivamente;
- (**Combina**): "unisce" le soluzioni dei sottoproblemi.

Sfortunatamente non esiste una ricetta unica per applicare questa tecnica: ad esempio l'algoritmo mergeSort ha una fase "divide" banale (basta calcolare il valore mediano) ma, allo stesso tempo una fase di unione delle soluzioni complessa, diversamente nel quickSort la fase "divide" è complessa ma non esiste una fase "combina". È quindi necessario fare uno sforzo creativo, in quanto la tecnica ci dà una modalità con cui ad arrivare alla soluzione, ma bisogna applicarla caso per caso.

Minimo divide-et-impera

La tecnica "divide-et-impera" non è un proiettile d'argento, a volte utilizzarla crea più danni di quanti ne risolva. Osserva questo esempio nella quale è presentato un algoritmo di ricerca del minimo con questa tecnica.

Algoritmo 15.2.1: Algoritmo di ricerca del minimo con tecnica divide-et-impera

```
minrec(int[] A, int i, int j)
    if i == j then
        return A[i]
    else
        m ← ⌊(i+j)/2⌋
        return min(minrec(A, i, m), minrec(A, m + 1, j))
```

Complessità L'algoritmo divide il vettore a metà, cerca il minimo nella metà di sinistra e nella metà di destra, il risultato è il minimo dei due minimi.

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

La complessità ammonta a $\alpha = 1, \beta = 0, T(n) = \Theta(n)$, non ne vale la pena, tanto vale fare la ricerca del minimo come abbiamo spiegato a lezione.

15.3 La torre di Hanoi

La torre di Hanoi è un gioco matematico che prevede tre pioli e n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è quello di impilare i dischi sul piolo di destra, senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta ed utilizzando il piolo centrale come appoggio. Questo problema può essere risolto tramite la tecnica "divide-et-impera".

Algoritmo 15.3.1: Versione ricorsiva della soluzione al problema della torre di Hanoi

```
hanoi(int n, int src, int dest, int middle)
    if n = 1 then
        stampa src → dest
    else
        hanoi(n − 1, src, middle, dest)  // sposta n − 1 dischi da src a middle
        stampa src → dest  // sposta 1 disco da src a dest
        hanoi(n − 1, middle, dest, src)  // sposta n − 1 dischi da middle a dest
```

Nella prima parte l'algoritmo sposta $n-1$ dischi da *src* a *middle* utilizzando *dest* come punto d'appoggio. Dopodiché sposta l'ultimo disco rimanente dalla *src* alla *dest*. Infine sposta $n-1$ dischi da *src* a *dest* utilizzando *src* come punto d'appoggio.

Complessità L'equazione di ricorrenza prodotta da questo algoritmo è $T = 2T(n-1) + 1 = \Theta(2^n)$. Si può dimostrare che questa soluzione è ottima (non si può fare meglio di così).

15.4 Algoritmo di ordinamento

L'algoritmo di ordinamento quickSort è basato sulla tecnica "divide et impera", nel caso medio ha una complessità di $\mathcal{O}(n \log n)$, mentre nel caso pessimo è di $\mathcal{O}(n^2)$. Fino a qualche anno fa era l'algoritmo di eccellenza per l'ordinamento. Infatti presenta molti aspetti a suo favore:

- il fattore costante del quickSort è migliore di quello del mergeSort;

- non utilizza memoria addizionale in quanto svolge i calcoli "in-memory" (a differenza di mergeSort che ha bisogno di un vettore di appoggio);
- esistono delle tecniche "euristiche" per evitare il caso pessimo.

Quindi spesso è preferito ad altri algoritmi. All'interno dell'ultimo capitolo riassumeremo tutti gli algoritmi di ordinamento visti fin'ora e ne vedremo di nuovi, tra questi anche gli algoritmi attualmente utilizzati negli attuali linguaggi di programmazione (c, java, python).

Spiegazione Sono dati in input un vettore $A[1 \dots n]$, gli indici $start$, end tali che $1 \leqslant start \leqslant end \leqslant n$, tali indici indicano quale parte del vettore stiamo ordinando, come avviene in mergeSort.

1. la parte del "divide" avviene nel seguente modo:
   - scegliamo un valore $p \in A[start \dots end]$ detto perno ($pivot$);
   - spostiamo gli elementi del vettore $A[start \dots end]$ in modo tale che:
     - $\forall i \in [start \dots j-1] : A[i] \leqslant p$;
     - $\forall i \in [j+1 \dots end] : A[i] \geqslant p$

     l'indice j viene calcolato per rispettare tale condizione;
   - il perno viene messo in posizione $A[j]$.
2. la parte "impera" ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente quickSort;
3. la parte "combina" non fa nulla.

Algoritmo 15.4.1: quickSort

```
quickSort(ITEM[] A, int primo, int ultimo)
    // su almeno due elementi
    if primo < ultimo then
        int j ← perno(A, primo, ultimo)  // logica dell'algoritmo

        // richiamo l'algoritmo su entrambi i sottovettori
        quickSort(A, primo, j − 1)
        quickSort(A, j + 1, ultimo)
```

Algoritmo 15.4.2: perno

```
// sposta gli elementi più piccoli a sinistra del perno, i più grandi a destra
int perno(ITEM[] A, int primo, int ultimo)
    ITEM x ← A[primo] // il perno è il primo elemento
    int j ← primo // il cursore parte dal primo elemento

    // spostamenti "in-place"
    from i ← primo until ultimo do
        if A[i] < x then // l'elemento è più piccolo del perno
            j++ // sposta il cursore j
            A[i] ↔ A[j] // scambia gli elementi: i ↔ j

    /* a questo punto tutti gli elementi posizionati prima della posizione j sono più piccoli del perno,
       rimane solo da riposizionare il perno nella sua posizione finale (è ordinato)                    */

    // riposiziono il perno
    A[primo] ← A[j]
    A[j] ← x

    // restituisco la posizione del perno
    return j
```

Complessità computazionale Il costo della funzione perno è $\Theta(n)$ (deve guardare $n-1$ valori ed effettuare i confronti). Il costo di quickSort dipende dal partizionamento:

- il partizionamento *peggiore* si verifica quando il perno è l'elemento minimo (o massimo), questo particolare caso accade quando il vettore è ordinato in ordine crescente (decrescente). La complessità risultante è $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.
- il partizionamento *migliore* avviene quando il vettore di dimensione n viene diviso in due sottoproblemi di dimensione $n/2$. La complessità risultante è $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$;
- il partizionamento nel *caso medio* è molto più vicino al caso ottimo che al caso peggiore, prendiamo ad esempio il partizionamento 9-a-1:

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

  Prendiamo un altro esempio, il partizionamento 99-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{99n}{10}\right) + cn = \Theta(n \log n)$$

  Nota. In questi esempi, il partizionamento ha proporzionalità limitata e i fattori moltiplicativi possono essere importanti.

  Il costo computazionale dipende dall'ordine degli elementi e non dai loro valori. Dobbiamo quindi considerare tutte le possibili permutazioni, il che è difficile dal punto di vista analitico. Alcuni partizionamenti saranno parzialmente bilanciati, altri pessimi; in media questi si alterneranno nella sequenza di partizionamenti, ma quelli parzialmente bilanciati "dominano" quelli pessimi.

Moltiplicazione di catena di matrici

Viene fatto un accenno ad un argomento che verrà affrontato in modo più approfondito nel capitolo successivo.

15.5 Conclusioni

La tecnica divide-et-impera viene applicata quando i passi "divide" e "combina" sono semplici e i costi risultano migliori del corrispondente algoritmo iterativo (quindi, ad esempio, va bene per effettuare l'ordinamento, ma non per effettuare la ricerca del minimo). Ulteriori vantaggi dell'applicazione di questa tecnica sono:

- la facile parallelizzazione: la possibilità di dividere il problema in più sottoproblemi porta ad una naturale divisione dei compiti fra più processori;
- l'utilizzo ottimale della memoria *cache* (*cache oblivious*): tutti i dati con la quale stiamo lavorando sono colocalizzati nella memoria principale.

15.6 Applicazione della tecnica

Infine vediamo una prima applicazione della tecnica e ne valutiamo le prestazioni.

15.6.1 Gap

In un vettore V contenente $n \geqslant 2$ interi, un gap è un indice i, $1 < i \leqslant n$, tale che $V[i-1] < V[i]$.

- Dimostrare che se $n \geqslant 2$ e $V[1] < V[n]$, allora V contiene almeno un gap;
- Progetta un algoritmo che, dato un vettore V contentente $n \geqslant 2$ interi e tale che $V[1] < V[n]$ (la condizione sopra), restituisca la posizione di un gap nel vettore (questo algoritmo assume che il gap esista).

Dimostrazione per assurdo. Suppponiamo che non ci sia un gap nel vettore. Allora $V[1] \geqslant V[2] \geqslant V[3] \geqslant \cdots \geqslant V[n]$, che contraddice il fatto che $V[1] < V[n]$. □

Proviamo a riformulare la proprietà tenendo conto di due indici:

- sia V un vettore di dimensione n;
- siano i, j due indici tali che $1 \leqslant i < j \leqslant n$ e $V[i] < V[j]$.

In altre parole, ci sono più di due elementi nel sottovettore $V[i \dots j]$ e il primo elemento $V[i]$ è più piccolo dell'ultimo elemento $V[j]$.

Dimostrazione per induzione. Voglia provare per induzione sulla dimensione n del sottovettore che il sottovettore contiene un gap.

- **caso base**: $n = j - i + 1 = 2$, ad esempio $j = i + 1$: $V[i] < V[j]$ implica che $V[i] < V[j]$ implica che $V[i] < V[i+1]$, che è un gap;
- **ipotesi induttiva**: dato un qualunque (sotto)vettore $V[h \dots k]$ di dimensione $n' < n$, tale che $V[h] < V[k]$, allora $V[h \dots k]$ contiene un gap;
- **passo induttivo**: consideriamo un qualunque elemento m tale che $i < m < j$. Almeno uno dei due casi seguenti è vero:
  - se $V[m] < V[j]$, allora esiste un gap in $V[m \dots j]$, per ipotesi induttiva;
  - se $V[i] < V[m]$, allora esiste un gap in $V[i \dots m]$, per ipotesi induttiva.

□

Algoritmo 15.6.1: Algoritmo che ricerca un intervallo all'interno del vettore

```
// funzione wrapper
gap(int[] V, int n)
    // n:       dimensione del vettore
    return gapRec(V, 1, n)

gapRec(int[] V, int i, int j)
    if j == i + 1 then // ho due elementi
        return j  // ritorno il secondo elemento

    m = ⌊(i+j)/2⌋  // calcolo il mediano

    if V[m] < V[j] then
        return gapRec(V, m, j)  // a destra
    else
        return gapRec(V, i, m)  // a sinistra
```

Tabella 15.1: Valutazione delle prestazioni degli algoritmi scritti in python

n	Iterativa (ms)	Ricorsiva (μs)
10^3	0,06	2,05
10^4	0,61	2,78
10^5	6,11	3,36
10^6	62,44	4,01
10^7	621,69	4,87
10^8	6205,72	5,47

Capitolo 16

Programmazione Dinamica

> *Those who cannot remember the past are condemned to repeat it*
>
> George Santayana, *1905*

Introduzione

La programmazione dinamica è un metodo per spezzare ricorsivamente un problema in sottoproblemi, i quali vengono risolti una sola volta e la loro soluzione viene memorizzata in una tabella. Nel caso il sottoproblema debba essere risolto nuovamente, si recupera la soluzione dalla tabella. La tabella è facilmente indirizzabile: la sua consultazione costa $\mathcal{O}(1)$.

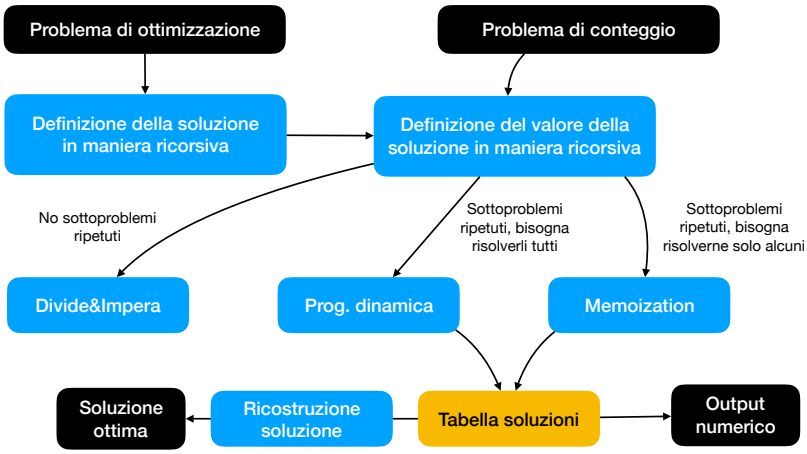


Figura 16.1: Approccio generale ad un problema

La programmazione dinamica nella storia

Il termine *Dynamic Programming* è stato coniato da Richard Bellman agli inizi degli anni '50, nell'ambito dell'ottimizzazione matematica. Inizialmente, si riferiva al processo di risolvere un problema compiendo le migliori decisioni una dopo l'altra. "*Dynamic*" doveva dare un senso "temporale", mentre "*Programming*" si riferiva all'idea di creare "programmazioni ottime", per esempio nella logistica. È possibile approfondire all'indirizzo `https://en.wikipedia.org/wiki/Dynamic_programming#History`.

La *programmazione dinamica* è caratterizzata da 4 fasi principali:

1. caratterizzare *matematicamente* la struttura di una soluzione ottima;
2. definire ricorsivamente il valore di una soluzione ottima;
3. calcolare il valore di una soluzione ottima con approccio *bottom-up* ed utilizzare una tabella per memorizzare la soluzione dei sottoproblemi ed utilizzarla per evitare di ripetere i calcoli più volte;
4. ricostruire la soluzione ottima.

16.1 Domino

Definizione del problema Il gioco del domino è basato su tessere di dimensione 2×1. Scrivere un algoritmo efficiente che prenda in input un intero n e restituisca il numero di possibili disposizioni di n tessere in un rettangolo $2 \times n$.

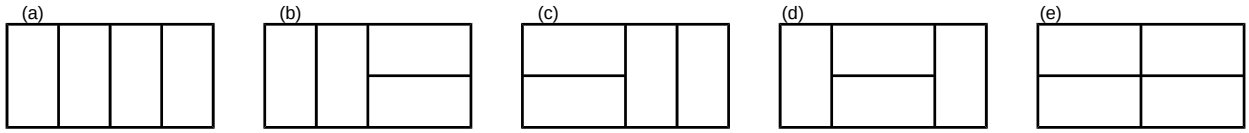


Figura 16.2: Rappresentazione delle cinque disposizioni possibili con cui è possibile riempire un rettangolo 2×4.

Definizione ricorsiva

Definiamo una formula ricorsiva $DP[n]$ che ci permetta di calcolare il numero di disposizioni possibili quando si hanno n tessere. Con nessuna tessera ($n = 0$) esiste una sola disposizione possibile. Avendo a disposizione una tessera ($n = 1$) è possibile disporla solo verticalmente.

Se posiziono una tessera in verticale, risolverò il problema di dimensione $n-1$, mentre se la posiziono in orizzontale ne devo mettere due, risolvendo così il problema di dimensione $n-2$. Queste ultime due possibilità si sommano insieme (conteggio).

$$DP(n) = \begin{cases} 1 & n \leqslant 1 \\ DP[n-2] + DP[n-1] & n > 1 \end{cases}$$

La serie generata è una successione di Fibonacci. $DP[n]$ infatti è pari al $n+1$-esimo numero della serie.

Algoritmo ricorsivo

L'algoritmo che viene scaturito dalla definizione ricorsiva del problema è il seguente:

Algoritmo 16.1.1: Algoritmo *ricorsivo* che risolve il problema Domino

```
int domino1(int n)
    if n ⩽ 1 then
        return 1
    else
        return domino1(n − 1) + domino1(n − 2)
```

Analisi della complessità L'equazione di ricorrenza associata a domino1 è la seguente:

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

È una ricorrenza lineare di ordine costante, per calcolare la sua complessità applichiamo quindi il *master theorem*: i fattori moltiplicativi di $T(n-1)$ e $T(n-2)$ sono $a_1 = 1$ e $a_2 = 1$ rispettivamente, possiamo raccogliergli in $a = a_1 + a_2 = 2$, il fattore β risulta pari a 0, in quanto $n^0 = 1$. Possiamo quindi concludere che la complessità dell'algoritmo è $\Theta(a^n \cdot n^\beta) = \Theta(2^n)$.

L'albero di ricorsione generato dall'algoritmo è il seguente:

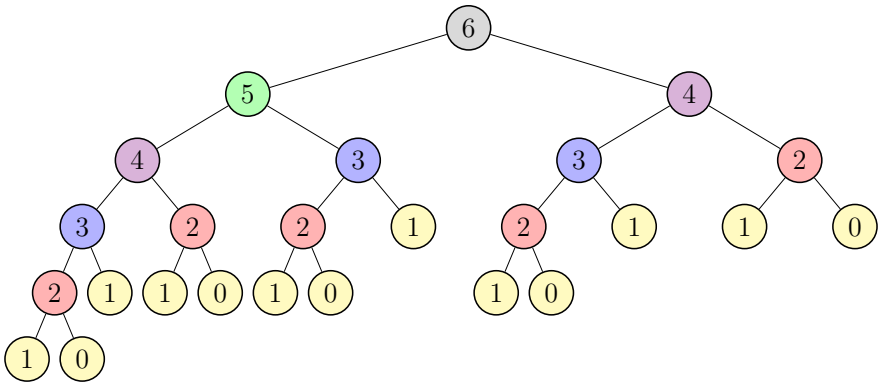


Figura 16.3: Albero di ricorsione per domino1. Possiamo notare che molti sottoproblemi vengono ripetuti.

Come evitare di risolvere un problema più di una volta

Dall'albero di ricorsione possiamo notare che molti sottoproblemi vengono ripetuti. Per evitare che questo avvenga memorizziamo il risultato ottenuto risolvendo un particolare problema in una **tabella DP**, la quale sarà un vettore, una matrice o dizionario dipendentemente dalle nostre esigenze. La tabella conterrà un elemento per ogni sottoproblema che dobbiamo risolvere. Memorizzeremo i casi base nelle relative posizioni. Dopodiché l'iterazione sarà *bottom-up*: partiremo dai casi base e andremo verso problemi via via sempre più grandi fino a risolvere il problema originale.

Algoritmo 16.1.2: Algoritmo *iterativo* che risolve il problema Domino

```
int domino2(int n)
    DP ← new int[0...n] // inizializzo la "tabella" DP
    DP[0] ← DP[1] ← 1 // inserisco i casi base

    // computo i valori successivi sulla base dei valori precedenti
    from i ← 2 until n do
        DP[i] ← DP[i − 1] + DP[i − 2]

    return DP[n] // restituisco il valore n-esimo richiesto
```

Analisi della complessità La complessità in tempo è $T(n) = \Theta(n)$, quella in spazio è $S(n) = \Theta(n)$. È possibile migliorare l'algoritmo riducendo lo spazio utilizzato.

Tabella 16.1: I casi base vengono inseriti manualemente nelle relative posizioni.
Ogni valore i-esimo successivo viene computato sulla base dei suoi valori precedenti $i-1$ e $1-2$.

n	0	1	2	3	4	5	6	7
$DP[n]$	*1*	*1*	2	3	5	8	13	21

Algoritmo 16.1.3: Algoritmo *iterativo che ottimizza lo spazio utilizzato* che risolve il problema Domino

```
int domino3(int n)
    int DP_0 ← 1
    int DP_1 ← 1
    int DP_2 ← 1
    from i ← 2 until n do
        DP_0 ← DP_1
        DP_1 ← DP_2
        DP_2 ← DP_0 + DP_1
    return DP_2
```

Analisi della complessità Questa implementazione ha costo costante nello spazio $S(n) = \Theta(1)$.

n	0	1	2	3	4	5	6	7
$DP[n]_0$	–	–	1	1	2	3	5	8
$DP[n]_1$	*1*	*1*	1	2	3	5	8	13
$DP[n]_2$	*1*	*1*	2	3	5	8	13	21

Sotto il modello di costo logaritmico, le tre versioni hanno le complessità mostrate nella tabella.

Tabella 16.2: Confronto delle varie versioni della funzione di fibonacci

Funzione	Tipologia	$T(n)$	$S(n)$
domino1	ricorsiva	$\mathcal{O}(n2^n)$	$\mathcal{O}(n^2)$
domino2	iterativa	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
domino3	finale	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

Si può fare meglio di così utilizzando l'esponenziazione di matrici basata su quadrati (per approfondimenti consultare `https://brilliant.org/wiki/fast-fibonacci-transform/`).

16.2 Hateville

Descrizione del problema Hateville è un villaggio particolare, composto da n case, numerate da 1 a n lungo una singola strada. Ad Hateville ognuno odia i propri vicini della porta accanto, da entrambi i lati. Quindi il vicino i odia i vicini $i-1$ e $i+1$ (se esistenti). Hateville vuole organizzare una sagra e vi ha affidato il compito di raccogliere i fondi. Ogni abitante i ha intenzione di donare una quantità $D[i]$, ma non intende partecipare ad una raccolta fondi a cui partecipano uno o entrambi i propri vicini.

Dobbiamo scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta.

Consegne del problema I problemi che possono esserci posti sono due:

1. scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta;
2. scrivere un algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1, \ldots, n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ è massimale.

se risolviamo il primo siamo ad un passo dalla soluzione del secondo, mentre se risolviamo il primo abbiamo risolto necessariamente il primo.

Esempi di esecuzione Con un vettore di donazioni $D = [4, 3, 6]$ la raccolta massima è 10, dato dall'insieme di indici $\{1, 3\}$. Mentre con un vettore di donazioni $D = [10, 5, 5, 10]$ la raccolta massima è 20, dato dall'insieme di indici $\{1, 4\}$.

La domanda che dobbiamo porci è la seguente: è possibile ridefinire una formula ricorsiva che ci permetta di calcolare il sottoinsieme di case che, se selezionate, dà origine alla maggior quantità di donazioni?

16.2.1 Definizione ricorsiva

Ridefiniamo il problema caratterizzandolo matematicamente.

Definiamo $HV(i)$ uno dei possibili insiemi di indici da selezionare per ottenere una donazione ottimale delle prime i case di Hateville, numerate $1, \ldots, n$. $HV(n)$ diventa quindi la soluzione del problema originale.

Passo ricorsivo

Andiamo per passi. Consideriamo il vicino i-esimo.

- cosa succede se **non accetto** la sua donazione? Lo scarto. Proviamo ad esprimerlo in funzione dei problemi precedenti (dopotutto il problema viene risolto in maniera ricorsiva):

$$HV(i) = HV(i-1)$$

- cosa succede se **accetto** la sua donazione? Accetto la donazione i-esima e scarto i vicini. In simboli:

$$HV(i) = \{i\} \cup HV(i-2)$$

- a questo punto come faccio a **decidere** quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:

$$HV(i) = highest(HV(i-1), \{i\} \cup HV(i-2))$$

La funzione *highest* restituisce l'insieme di valore massimo.

Sottostruttura ottima

Quando voglio provare che una soluzione di programmazione dinamica è corretta devo riuscire a dimostrare che le mie possibili scelte sono quelle giuste. Per dimostrarlo utilizziamo il teorema di sottostruttura ottima, che dice sostanzialmente che il modo in cui ho applicato la ricorsione è corretto.

Proviamo quindi a dimostrare le scelte fatte. Il problema dato dalle prime i case è indicato con $HV_p(i)$ e una (ce ne può essere più di una) soluzione ottima per questo problema è indicato con $HV_s(i)$.

Se $i \notin HV_s i$ allora $HV_s(i) = HV_s(i-1)$ (ossia se l'i-esimo elemento non appartiene alla soluzione del problema $HV_s i$ allora abbiamo scartare quella donazione $HV_s(i-1)$), altrimenti se l'abbiamo accettata e quindi $i \in HV_s i$ allora $HV_s(i) = HV_s(i-2) \cup \{i\}$.

Se abbiamo la soluzione ottima, allora possiamo dimostrare che abbiamo la soluzione ottima per i rispettivi sottoproblemi (da qui sottostruttura).

Nota. Nella maggior parte dei casi non è necessaria una dimostrazione della soluzione, ma basta un'intuizione.

16.2.2 Dimostrazione sottostruttura ottima

Ricordiamo che indichiamo con $HV_p(i)$ il problema dato dalle prime i case e con $HV_s(i)$ una delle possibili soluzioni ottime per questo problema. Indichiamo inoltre con $|HV_s(i)|$ l'ammontare di donazioni per la soluzione ottima $HV_s(i)$.

Nota. Dobbiamo dimostrare separatamente il caso in cui prendiamo la donazione i-esima e il caso in cui non la prendiamo, in quanto sono eventi mutualmente esclusivi.

Entrambe le dimostrazioni procedono per assurdo.

Dimostrazione caso 1: $\boxed{i \notin HV_s(i)}$. Se non abbiamo preso la donazione i-esima vogliamo dimostrare che la soluzione ottima $HV_s(i)$ è una soluzione ottima anche per il problema precedente $HV_p(i-1)$.

Se così non fosse esisterebbe una soluzione (migliore) $HV_s^{'}(i-1)$ per il problema $HV_p(i-1)$ tale che l'ammontare delle donazioni del problema precedente sarebbe maggiore (in simboli $|HV_s^{'}(i-1)| > |HV_s(i)|$). Ma allora $HV_s^{'}(i-1)$ sarebbe una soluzione per $HV_p(i)$, che è assurdo. Quindi $HV_s(i)$ è una soluzione ottima anche per $HV_p(i-1)$. □

Dimostrazione caso 2: $\boxed{i \in HV_s(i)}$. Se abbiamo preso l'i-esima donazione vogliamo dimostrare che $i-1$ non appartiene alla soluzione ottima (in simboli $i-1 \notin HV_s(i)$), altrimenti non sarebbe una soluzione ammissibile. Quindi, se dalla soluzione ottima $HV_s(i)$ togliessimo la donazione i-esima ($HV_s(i) - \{i\}$) questa dovrebbe essere una soluzione ottima per $HV_p(i-2)$.

Se così non fosse, esisterebbe una soluzione $HV_s^{'}(i-2)$ per il problema $HV_p(i-2)$ tale che il suo guadagno sarebbe maggiore (in simboli $|HV_s^{'}(i-2)| > |HV_s(i) - \{i\}|$). Ma allora $HV_s^{'}(i-2) \cup \{i\}$ sarebbe una soluzione per $HV_p(i)$, che è assurdo. Quindi $i-1 \notin HV_s(i)$. □

16.2.3 Completare la ricorsione

Ragioniamo sui casi base. Se ho 0 case il mio guadagno è zero: $HV(0) = \varnothing$; se ho una casa prendo semplicemente la sua donazione: $HV(1) = \{1\}$.

Possiamo quindi scrivere la formula per calcolare la somma massima date i case:

$$HV(i) = \begin{cases} 0 & i = 0 \\ \{1\} & i = 1 \\ highest(HV(i-1), HV(i-2) \cup \{i\}) & i \geqslant 2 \end{cases}$$

Non vale la pena scrivere un algoritmo ricorsivo, basato su divide-et-impera, per risolvere il problema di Hateville poiché si risolverebbero molti sottoproblemi più volte.

16.2.4 Memorizzare una tabella

Facciamo qualche esempio di esecuzione. Nel primo il vettore delle donazioni è $D = [10, 5, 5, 8, 4, 7, 12]$, mentre nel secondo è $D = [10, 1, 1, 10, 1, 1, 10]$. Convincersi che gli insiemi risultanti sono corretti.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
HV	$\varnothing$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,4\}$	$\{1,3,5\}$	$\{1,4,6\}$	$\{1,3,5,7\}$

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
HV	$\varnothing$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,4\}$	$\{1,4\}$	$\{1,4,6\}$	$\{1,4,7\}$

A questo punto dobbiamo risolvere ancora due problemi:

1. dobbiamo definire la funzione *highest* (ma è banale);
2. dobbiamo memorizzare gli insiemi nella tabella.

Memorizzare gli insiemi delle scelte nella tabella è costoso, quindi lo non faremo. Costruiremo invece il valore della soluzione. Così facendo eviteremo di memorizzare gli insiemi nella tabella e potremmo comunque ricostruire la soluzione a posteriori.

Tabella di programmazione dinamica

Indichiamo con $DP[i]$ il **valore** della massima quantità di donazioni che possiamo ottenere dalle prime i case di Hateville, e con $DP[n]$ il valore della soluzione ottima. Possiamo quindi riempire la tabella di programmazione dinamica nel seguente modo:

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ \mathsf{max}(DP[i-1], DP[i-2] + DP[i]) & i \geqslant 2 \end{cases}$$

Nel caso avessi 0 donatori allora la somma delle donazioni sarà 0. Mentre nel caso abbia un solo donatore ($i = 1$) allora prenderò la sua donazione ($D[1]$). Infine nel caso avessi 2 o più donatori allora prendere una scelta: o scarterò quella donazione, quindi non considererò più quell'indice ($DP[i-1]$), o la accetter ($+DP[i]$) e dovrò scartare a priori la scelta del suo vicino ($DP[i-2]$). La sceltà è rappresentata dalla funzione max che selezionerà quale fra le due scelte sarà quella più conveniente.

Nota. Non memorizziamo più insiemi, ma valori. Infatti non effettuiamo più l'unione di insiemi ma la somma fra i valori contenuti all'interno della tabella.

Dall'equazione di ricorrenza possiamo scrivere in modo naturale un algoritmo iterativo che risolve questo particolare problema. Nel caso volessimo implementare un algoritmo ricorsivo allora dovremmo utilizzare la tecnica della *memoization* che vedremo più avanti.

Algoritmo 16.2.1: Algoritmo iterativo che risolve il problema Hateville

```
int hateville(int[] D, int n)
    // creo la tabella, un vettore in questo caso
    int[] DP ← new int[0...n]

    // inserisco i casi base
    DP[0] ← 0 // nessun donatore
    DP[1] ← D[1] // un solo donatore

    // calcolo il valore n-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i-1], DP[i-2] + D[i])

    // restituisco il valore n-esimo
    return DP[n]
```

Stiamo calcolando la soluzione per ogni possibile sottoproblema $(n+1)$ qual è il valore massimo della soluzione. Questa soluzione ha complessità $\Theta(n)$ in quanto dobbiamo fare $\Theta(n)$ per ottenere il risultato.

Soluzione con linguaggi di programmazione

Vediamo un paio di implementazioni con "veri" linguaggi di programmazione. Gli indici differiscono dalla notazione matematica.

```java
public int hateville(int[] D, int n) {
    int[] DP = new int[n+1];
    DP[0] = 0;
    DP[1] = D[0]; // l'indice parte da 0
    for (int i=2; i <= n; i++) {
        DP[i] = max(DP[i-1], DP[i-2] + D[i-1]); // devo prendere la donazione i-1
    }

    return DP[n];
}
```

Codice 16.1: Implementazione della soluzione in Java

```python
def hateville(D):
    DP = [ 0, D[0] ] # l'indice parte da 0

    for i in range(1,len(D)): # scrittura più elegante
        DP.append( max(DP[-1], DP[-2] + D[i]) )

    return DP[-1]
```

Codice 16.2: Implementazione della soluzione in Python

16.2.5 Ricostruire la soluzione originale

Questi sono i possibili risultati che possiamo ottenere applicando l'algoritmo.

i	0	1	2	3	4	5	6	7
D		10	5	5	8	4	7	12
DP	0	10	10	15	18	19	25	31

i	0	1	2	3	4	5	6	7
D		10	1	1	10	1	1	10
DP	0	10	10	11	20	20	21	30

A questo punto abbiamo il valore della soluzione massimale, ma non abbiamo la soluzione, ossia l'insieme degli indici.

Per riscostruire la soluzione guardiamo l'elemento i-esimo presente nella tabella nella posizione $DP[i]$, se la casa i-esima non è stata selezionata allora il valore di $DP[i]$ deriva da $DP[i-1]$, altrimenti (se la casa è stata selezionata) il suo valore deriva da $DP[i-2]+D[i]$. Se entrambe le equazioni sono vere, una vale l'altra.

Utilizziamo quindi questa informazione per ricostruire la soluzione in modo ricorsivo: per ricostruire la soluzione fino ad i, calcoliamo i valori fino a $i-2$ e aggiungiamo i (se la casa è stata selezionata), altrimenti li calcoliamo fino a $i-1$ senza aggiungere nulla.

Algoritmo 16.2.2: Ricostruire la soluzione generale di Hateville

```
int hateville(int[] D, int n)
    // creo la tabella
    int[] DP ← new int[0...n]
    // inserisco i casi base
    DP[0] ← 0
    DP[1] ← DP[1]
    // calcolo il valore i-esimo
    from i ← 2 until n do
        DP[i] ← max(DP[i-1], DP[i-2]+D[i])
    // restituisco il valore n-esimo
    return solution(DP, D, n)

// ricostruisce l'insieme degli indici dato il valore massimale
int solution(int[] DP, int[] D, int i)
    // i:       indice di scorrimento
    if i==0 then // nessun donatore
        return ∅
    else if i==1 then// un solo donatore
        return {1}
    else if DP[i]==DP[i-1] then // se non c'è variazione fra valori consecutivi

        return solution(DP, D, i-1) // scarto l'indice
    else // c'è variazione fra valori consecutivi
        SET sol = solution(DP, D, i-2) // chiamo ricorsivamente l'algoritmo sull'indice i-2
        sol.insert(i) // inserisco l'indice nell'insieme
        // restituisco l'insieme degli indici
        return sol
```

Effettuo prima la chiamata ricorsiva e poi l'inserimento dell'indice all'interno dell'insieme così alla fine dell'esecuzione gli indici saranno nell'ordine corretto.

Analisi della complessità La complessità computazionale di solution è $T(n) = \Theta(n)$, quella spaziale è $S(n) = \Theta(n)$.

Nota. Non è possibile migliorare la complessità spaziale di hateville poiché è necessario ricostruire la soluzione.

16.3 Zaino

Definizione informale del problema Dato un insieme di oggetti, ognuno caratterizzato da un *peso* ed un *profitto*, e uno "zaino" con un limite di capacità, individuare un sottoinsieme di oggetti il cui peso sia inferiore alla capacità dello zaino e in cui il valore totale degli oggetti sia massimale, ossia il più alto o uguale al valore di qualunque altro sottoinsieme di oggetti.

Definizione formale del problema Dati un vettore w, dove $w[i]$ è il **peso** (*weight*) dell'oggetto i-esimo, un vettore p, dove $p[i]$ è il **profitto** (*profit*) dell'oggetto i-esimo, e la **capacità** C dello zaino. Bisogna trovare un insieme $S \subseteq \{1, \ldots, n\}$ tale che:

- il **valore totale** deve essere minore o uguale alla capacità;
$$w(S) = \sum_{i \in S} w[i] \leqslant C$$
- il **profitto totale** deve essere massimizzato.
$$p(S) = \sum_{i \in S} p[i]$$

Esempi di esecuzione Con un vettore dei pesi $w = [10, 4, 8]$ ed un vettore dei profitti $p = [20, 6, 12]$ ed una capacità $C = 12$. L'insieme degli indici da restituire è $S = \{1\}$. Mentre con un vettore dei pesi $w = [10, 4, 8]$ ed un vettore dei profitti $p = [20, 7, 15]$ ed una capacità sempre pari a $C = 12$. L'insieme degli indici da restituire è $S = \{2, 3\}$.

Definizione matematica del valore della soluzione

Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $i \leqslant n$ oggetti contenuti in uno zaino di capacità $c \leqslant C$. Il massimo profitto ottenibile dal problema originale è rappresentato da $DP[n][C]$.

Passo ricorsivo

Andiamo per passi. Consideriamo l'oggetto i-esimo.

- cosa succede se **non prendo** quell'oggetto? La capacità non cambia e non c'è profitto. Avanziamo con l'indice $(i-1)$.
$$DP[i][c] = DP[i-1][c]$$
- cosa succede se **prendo** quell'oggetto? Sottraiamo il peso dalla capacità $(c - w[i])$ e aggiungiamo il relativo profitto $(+p[i])$. Avanziamo con l'indice $(i-1)$.
$$DP[i][c] = DP[i-1][c - w[i]] + p[i]$$
- a questo punto come faccio a **decidere** quale delle due opzioni scegliere? Semplicemente prendo quello che mi dà un guadagno maggiore. In simboli:
$$DP[i][c] = \mathsf{max}(DP[i-1][c - w[i]] + p[i], DP[i-1][c])$$

Completare la ricorsione

Ragioniamo sui casi base. Se non abbiamo più oggetti o se abbiamo finito la capacità dello zaino allora il nostro guadagno sarà 0. E nel caso prendessimo un oggetto la nostra capacità diventasse negativa? In quel caso mettiamo come valore convenzionale $-\infty$.

Possiamo quindi scrivere la formula per calcolare il massimo guadagno dati i oggetti ed una capacità c:

$$DP[i][c] = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \mathsf{max}(DP[i-1][c - w[i]] + p[i], DP[i-1][c])) & \text{altrimenti} \end{cases}$$

16.3.1 Algoritmo iterativo

Dalla formula possiamo ricavare il seguente algoritmo iterativo.

Algoritmo 16.3.1: Algoritmo *iterativo* per la soluzione al problema dello zaino

```
int knapsack(int[] w, int[] p, int n, int C)
    // w:       vettore dei pesi
    // p:       vettore dei profitti
    // n:       numero di oggetti
    // C:       capacità massima dello zaino
    // creo la tabella di programmazione dinamica
    DP ← new int[0...n][0...C]
    // la inizializzo
    from i ← 0 until n do
        DP[i][0] = 0 // capacità nulla
    from c ← 0 until C do
        DP[0][c] = 0 // nessun oggetto
    // calcolo caso per caso
    from i ← 1 until n do
        from c ← 1 until C do
            if w[i] ⩽ c then // se la capacità residua è sufficiente
                DP[i][c] = max(DP[i−1][c−w[i]]+p[i], DP[i−1][c]) // decido se prenderlo o scartarlo
            else
                DP[i][c] = DP[i − 1][c] // lo scarto
    // restituisco il profitto massimo
    return DP[n][C]
```

Esempio di esecuzione Con un vettore dei pesi $w = [4, 2, 3, 4]$ e un vettore dei profitti $p = [10, 7, 8, 6]$ ed una capacità di $C = 9$, la tabella di programmazione generata è la seguente:

i \ c	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	0	0	0	10	10	10	10	10	10
2	*0*	0	7	7	10	10	17	17	17	17
3	*0*	0	7	8	10	15	17	18	18	25
4	*0*	0	7	8	10	15	17	18	18	25

I valori in **grassetto** indicato numero di oggetti (i) e la capacità (c) dello zaino, mentre i valori in *corsivo* sono i valori inizializzati della tabella.

La tabella viene riempita per righe. Per capire come funziona l'algoritmo prendiamo in considerazione la seconda riga che corrisponde alla possibilità di prendere $i = 2$ oggetti: una volta raggiunta la capacità di $c = 2$ possiamo prendere l'oggetto con peso 7, una volta raggiunta la capacità di $c = 4$ devo scegliere il massimo profitto prendere l'oggetto 10 ($DP[i-1][c-w[i]]+p[i]$) e 7, ossia ignorarlo $DP[i-1][c]$. Quando raggiungo la capacità di $c = 6$ posso prendere entrambi gli oggetti per un profitto totale di 17.

16.3.2 Algoritmo ricorsivo

Complessità La complessità di knapsack è $\Theta(nC)$. Osserviamo che C è parte dell'input (non è la dimensione del problema). Applicchiamo quindi il criterio di costo logaritmico: per rappresentare C sono necessari $k = \log_2 C$ bit, quindi la complessità è $T(n) = \mathcal{O}(n2^k)$. knapsack è un algoritmo esponenziale.

Algoritmo 16.3.2: Algoritmo *ricorsivo* per la soluzione al problema dello zaino

```
// metodo wrapper
int knapsack(int[] w, int[] p, int n, int C)
    return knapsackRec(w, p, n, C)

int knapsackRec(int[] w, int[] p, int n, int c)
    // c:       capacità residua
    if c < 0 then
        return −∞
    else if i == 0 or c == 0 then
        return 0
    else
        int nottaken ← knapsackRec(w, p, i − 1, c)
        int taken ← knapsackRec(w, p, i − 1, c − w[i]) +p[i]
        return max(nottaken, taken)
```

Analisi della complessità La funzione ricorsiva scaturita dalla funzione knapsack è la seguente:

$$T(n) = \begin{cases} 1 & n \leqslant 1 \\ 2T(n-1)+1 & n > 1 \end{cases} = \mathcal{O}(2^n)$$

Non si può fare meglio di così.

16.3.3 Memoization

Nota. Non tutti gli elementi della matrice sono necessari alla risoluzione del nostro problema.

Prendiamo l'esempio precedente.

i \ c	0	1	2	3	4	5	6	7	8	9
0	***0***	***0***	***0***	***0***	***0***	***0***	***0***	***0***	*0*	***0***
1	***0***	0	**0**	**0**	**10**	**10**	**10**	**10**	10	**10**
2	*0*	0	**7**	7	10	**10**	**17**	17	17	**17**
3	*0*	0	7	8	10	**15**	17	18	18	**25**
4	*0*	0	7	8	10	15	17	18	18	**25**

I valori segnati in **rosso** sono gli unici necessari alla computazione della soluzione.

La *memoization* (annotazione) è una tecnica che fonde l'approccio di *memorizzazione* della programmazione dinamica con l'approccio *top-down* di divide-et-impera.

Quando devo risolvere un sottoproblema prima controllo se l'ho già risolto, guardando nella cella corrispondente della tabella dove memorizzo i risultati, altrimenti lo calcolo al momento (*on-the-fly*) chiamando

ricorsivamente i sottoproblemi e scrivendo i rispettivi risultati nella tabella. In questo modo mi assicuro di fare il calcolo una volta sola ed evito di calcolare valori che non verranno mai usati.

Per indicare che il problema non è ancora stato risolto la inizializzo ad un valore speciale (-1).

Algoritmo 16.3.3: Zaino con memoization

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    DP ← new int[1...n][1...C]
    from i ← 1 until n do
        from c ← 1 until C do
            DP[i][c] = −1 // non ho ancora risolto questo sotto-problema
    return knapsackRec(w, p, n, C, DP)
int knapsackRec(int[] w, int[] p, int n, int c, int[][] DP)
    if c < 0 then
        return −∞
    else if i == 0 or c == 0 then
        return 0
    else
        if DP[i][c] < 0 then
            int nottaken ← knapsackRec(w, p, i − 1, c, DP)
            int taken ← knapsackRec(w, p, i − 1, c − w[i], DP) +p[i]
            DP[i][c] ← max(nottaken, taken)
        return DP[i][c]
```

La tabella viene inizializzata esternamente, nella funzione *wrapper*, con un valore che non viene usato durante la procedura (-1 nel caso vengano usati *solo* valori positivi, $-\infty$ nel caso vengano usati sia valori positivi che valori negativi).

```python
def knapsackRec(w, p, i, c, DP):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        if DP[i][c] < 0:
            nottaken = knapsackRec(w, p, i-1, c, DP)
            taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
            DP[i][c] = max(nottaken, taken)
            return DP[i][c]

def knapsack(w,p,C):
    n = len(w)
    DP = [[-1]*(C+1) for i in range(n+1)]
    return knapsackRec(w,p,n,C,DP)
```

i \ c	0	1	2	3	4	5	6	7	8	9
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
1	*0*	−1	**0**	**0**	**10**	**10**	**10**	**10**	−1	**10**
2	*0*	−1	**7**	−1	−1	**10**	**17**	−1	−1	**17**
3	*0*	−1	−1	−1	−1	**15**	17	−1	−1	**25**
4	*0*	−1	−1	−1	−1	−1	−1	−1	−1	**25**

Scelta della struttura dati È meglio scegliere una tabella o un dizionario?

Il costo di inizializzazione della tabella è pari a $\mathcal{O}(nC)$. Applicata in questo modo, non c'è alcun vantaggio nell'utilizzare la tecnica di *memoization*. Permette tuttavia di tradurre in fretta le espressioni ricorsive.

Se invece di usare una tabella utilizzassimo un dizionario non dovremmo pagare il costo d'inizializzazione. Il costo di esecuzione è pari a $\mathcal{O}(min(2^n, nC))$.

```python
def knapsack(w,p,C):
    n = len(w)
    DP = {} # Hash-table dictionary
    return knapsackRec(w,p,n,C,DP)

def knapsackRec(w, p, i, c, DP):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        if not (i,c) in DP: # la soluzione non è contenuta nell'insieme delle soluzioni
        nottaken = knapsackRec(w, p, i-1, c, DP)
        taken = knapsackRec(w, p, i-1, c-w[i-1], DP) + p[i-1]
        DP[i,c] = max(nottaken, taken)
        return DP[i,c]
```

Codice 16.3: Implementazione di zaino con dizionario in Python

```python
from functools import wraps

def memo(func):
    cache = {}
    @wraps(func)
    def wrap(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrap

@memo
def knapsackRec(w, p, i, c):
    if c < 0:
        return -math.inf
    elif i == 0 or c == 0:
        return 0
    else:
        nottaken = knapsackRec(w, p, i-1, c)
        taken = knapsackRec(w, p, i-1, c-w[i-1]) + p[i-1]
        return max(nottaken, taken)
```

```
def knapsack(w, p, C):
    n = len(w)
    return knapsackRec(w, p, n, C)
```

Codice 16.4: Implementazione di zaino con dizionario in Python con annotazione automatica

Nota. La ricostruzione della soluzione in Python è lasciata come esercizio.

16.4 Zaino senza limiti

Con *senza limiti* ci si riferisce alla scelta degli oggetti.

Definizione del problema Dato uno zaino di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $i \leqslant n$ oggetti contenuti in uno zaino di capacità $c \leqslant C$, *senza porre limiti al numero di volte che un oggetto può essere selezionato*.

Definizione matematica del valore della soluzione

Come modifichiamo la formula ricorsiva?

$$DP[i][c] = \begin{cases} 0 & i = 0 \textbf{ or } c = 0 \\ -\infty & c < 0 \\ \mathsf{max}(DP[i-1][c-w[i]] + p[i], DP[i-1][c]) & \text{altrimenti} \end{cases}$$

In un caso come questo, è possibile semplificare la formula riducendo lo spazio occupato.

Variante della soluzione

Dato uno zaino senza limiti di scelta di capacità C e n oggetti caratterizzati da peso w e profitto p, definiamo $DP[c]$ come il massimo profitto che può essere ottenuto da tali oggetti in uno zaino di capacità $c \leqslant C$.

$$DP[c] = \begin{cases} 0 & c = 0 \\ max_{w[i] \leqslant c}\{DP[c-w[i]] + p[i]\} & c > 0 \end{cases}$$

Algoritmo 16.4.1: Zaino senza limiti con *memoization*

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    int[] DP ← new int[0...C]
    from i ← 0 until C do
        DP[i] = −1
    return knapsackRec(w, p, n, C, DP)
    return DP[C]

int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)
    if c == 0 then
        return 0
    if DP[c] < 0 then
        DP[c] ← 0
        from i ← 1 until n do
            if w[i] ⩽ c then
                int val ← knapsackRec(w, p, n, c − w[i], DP) + p[i]
                DP[c] ← max(DP[c], val)
    return DP[c]
```

Complessità Non è detto che tutti gli elementi debbano essere riempiti. La complessità in spazio è pari a $\Theta(C)$. Questo approccio rende più difficile ricostruire la soluzione. Possiamo ispezionare tutti gli elementi per capire da dove deriva il massimo. Conviene tuttavia memorizzare l'indice da cui deriva il massimo.

Algoritmo 16.4.2: Zaino senza limiti con *memoization* e ricostruzione della soluzione

```
// funzione wrapper
int knapsack(int[] w, int[] p, int n, int C)
    int[] DP ← new int[0...C]
    int[] pos ← new int[0...C] // vettori delle posizioni
    from i ← 0 until C do
        DP[i] = −1
        pos[i] = −1 // lo inizializzo
    knapsackRec(w, p, n, C, DP, pos)
    return solution(w, C, pos)

// calcolo dei valori
int knapsackRec(int[] w, int[] p, int n, int c, int[] DP, int[] pos)
    if c == 0 then
        return 0
    if DP[c] < 0 then
        DP[c] ← 0
        from i ← 1 until n do
            if w[i] ⩽ c then
                int val ← knapsackRec(w, p, n, c − w[i], DP, pos) + p[i]
                if val ⩾ DP[c] then
                    DP[c] ← val
                    pos[c] ← i
    return DP[c]

// ricostruzione della soluzione
solution(int[] w, int c, int[] pos)
    if c == 0 or pos[c] < 0 then
        return List()
    else
        List L ← solution(w, c − w[pos[c]], pos)
        L.insert(L.head(), pos[c])
        return L
```

16.5 Greedy

> *The point is, ladies and gentleman, that greed, for lack of a better word, is good. Greed is right, greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed, in all of its forms; greed for life, for money, for love, for knowledge has marked the upward surge of mankind.*

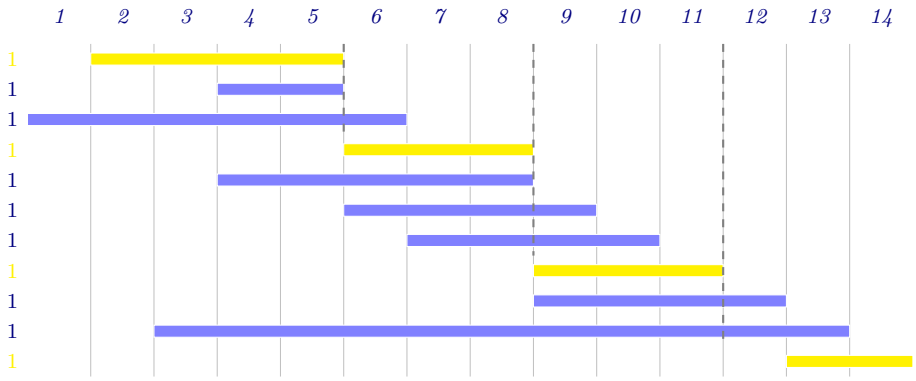


Figura 16.4: Una delle possibili soluzioni ingorde al problema dell'insieme indipendente di intervalli. Quali altre possibili soluzioni di cardinalità massima sono presenti? Nota che tutti gli intervalli hanno lo stesso peso.

”

Gordon Gekko, *Wall Street*

Nota. Non è difficile scrivere algoritmi ingordi, la parte difficile sta nel dimostrare che restituiscano la soluzione ottimale.

Molti dei problemi che vedremo li abbiamo già visti, ma in questo capitolo tratteremo dei loro casi particolari. Capita spesso che per risolvere casi generali ci si debba avvalere della programmazione dinamica, mentre per casi particolari sia meglio usare algoritmi ingordi.

16.6 Introduzione

Sia la programmazione dinamica che gli algoritmi greedy cercano di risolvere problemi di ottimizzazione. Gli algoritmi che li risolvono devono prendere una serie di decisioni e differiscono fra di loro da *come* queste decisioni vengono prese. Nella programmazione dinamica valutiamo tutte le possibili decisioni evitando di rivalutare decisioni (percorsi) già intraprese. Negli algoritmi ingordi, invece, selezioniamo *una sola* fra le possibili decisioni. Quale? Quella che sembra ottima (ovvero *localmente ottima*). È però necessario dimostrare che si ottiene un ottimo globale (ossia una soluzione *globalmente ottima*).

Nota. Questo approccio riduce la complessità di dover valutare tutte le possibilità, ma necessita di essere dimostrato.

Quando applicarla È consigliato applicare la tecnica greedy quando fra tutte le scelte possibili ne può essere individuata una che porta sicuramente alla soluzione ottima. Deve comunque rimanere valida (come nella programmazione dinamica) la proprietà di *sottostruttura ottima* ovvero che quando viene effettuata una scelta resti un sottoproblema con la stessa struttura del problema principale.

Nota. Non tutti i problemi hanno una soluzione greedy.

16.7 Insieme indipendente di intervalli non pesati

Definizione del problema Dati in input un insieme di intervalli della retta reale $S = \{1, 2, \dots n\}$. Trovare un *insieme indipendente massimale*, ovvero un sottoinsieme di cardinalità massima formato da intervalli disgiunti tra loro.

Nota (Proprietà degli intervalli). Gli intervalli sono chiusi a sinistra e aperti a destra.

Abbiamo già risolto questo problema (con la programmazione dinamica), ma il fatto che tutti i pesi siano pari a 1 porta ad una semplificazione del problema.

Nota. Per questo particolare problema non esiste un insieme di cardinalità 5 (ossia un insieme contenente cinque elementi).

Come affrontare il problema

1. *individuare* la sottostruttura ottima;
2. *scrivere* una definizione ricorsiva per la dimensione della soluzione ottima;
3. *cercare* una possibile scelta ingorda;
4. *dimostrare* che la scelta presa porti alla soluzione ottima;
5. *scrivere* un algoritmo ricorsivo (spesso sono iterativi) che effettui sempre la scelta ingorda.

16.7.1 Individuazione sottostruttura ottima

Assumiamo che gli intervalli siano ordinati per tempo di fine ($b_1 \leqslant b_2 \leqslant \dots b_n$).

Definiamo il sottoproblema $S[i,j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j, ovvero $S[i,j] = \{k \mid b_i \leqslant a_k < b_k \leqslant a_j\}$.

Per scopi implementativi (fungeranno da sentinelle) aggiungiamo due intervalli fittizi che singoleggiano $\pm\infty$, ovvero l'intervallo zeresimo indicato da b_0 ($b_0 = -\infty$) e l'intervallo successivo indicato da $n+1$ ($n+1 = +\infty$). In questi termini il problema iniziale corrisponde al problema $S[0, n+1]$.

Nota. L'idea è definire problemi a mano a mano sempre più piccoli. E definirli in base agli indici degli intervalli rimanenti.

Teorema. *Supponiamo che $A[i,j]$ sia una (poiché non è unica) soluzione ottimale di $S[i,j]$ e sia k un (qualunque) intervallo che appartiene a $A[i,j]$; suddividiamo quindi il problema $S[i,j]$ in due sottoproblemi:*

- *$S[i,k]$: gli intervalli di $S[i,j]$ che finiscono prima di k;*
- *$S[k,j]$: gli intervalli di $S[i,j]$ che iniziano dopo di k*

$A[i,j]$ contiene le soluzioni ottimali di $S[i,k]$ e $S[k,j]$ quindi:

- *$A[i,j] \cap S[i,k]$ è la soluzione ottimale di $S[i,k]$*
- *$A[i,j] \cap S[k,j]$ è la soluzione ottimale di $S[k,j]$*

Dimostrazione per assurdo. Se esistesse una soluzione migliore al problema $S[i,j]$, diciamo $A'[i,k]$ e la sostituissi ad $A[i,k]$ ottenerrei una soluzione migliore anche al mio problema originale, ma questo non è possibile poiché se ottenessi una soluzione migliore allora $A[i,k]$ non sarebbe una soluzione ottima, il che è assurdo. □

16.7.2 Definizione ricorsiva del costo della soluzione

Partendo dalla definizione ricorsiva della soluzione

$$A[i,j] = A[i,k] \cup \{k\} \cup A[k,j]$$

non possiamo determinare k a priori quindi dobbiamo necessariamente provare tutti i valori.

L'equazione di ricorrenza che si ottiene è la seguente

$$DP[i,j] = \begin{cases} 0 & S[i,j] = \varnothing \\ \max\limits_{k \in S[i,j]} \{\underbrace{DP[i,k]}_{\text{prima di } k} + \underbrace{DP[k,j]}_{\text{dopo } k} + 1\} & \text{altrimenti} \end{cases}$$

dove $DP[i,j]$ è la dimensione del più grande sottoinsieme $A[i,j] \subseteq S[i,j]$ di intervalli indipendenti.

1. se l'insieme di intervalli dati in input è vuoto ($S[i,j] = \varnothing$), allora la dimensione dell'insieme è pari a 0 (caso base);
2. altrimenti, se l'insieme di intervalli di partenza non è vuoto, consideriamo l'insieme di cardinalità massima. L'insieme viene calcolato scegliendo l'intervallo k-esimo fra tutti gli intervalli k appartenenti all'insieme di partenza ($k \in S[i,j]$), sommando quindi 1 alla soluzione finale, e chiamando ricorsivamente il problema sugli intervalli rimanenti.

16.7.3 Verso una soluzione ingorda

La definizione precedente ci permette di scrivere un algoritmo basato su programmazione dinamica o su memoization di complessità $\mathcal{O}(n^3)$: bisogna necessariamente risolvere tutti i problemi con $i < j$, con costo $\mathcal{O}(n)$ nel caso pessimo.

Nella risoluzione del problema di intervalli *pesati* abbiamo visto un algoritmo di complessità $\mathcal{O}(n \log n)$, il quale è applicabile anche nella risoluzione di questo problema. Questa complessità era data dall'ordinamento del vettore che avveniva prima che i dati venissero processati indipendentemente dall'ordinamento iniziale. Tuttavia è possibile migliorare il nostro algoritmo cercando una soluzione ingorda al nostro problema evitando di valutare tutte le possibili soluzioni. Infatti non è necessario analizzare tutti i possibili valori di k.

Teorema (scelta ingorda)**.** *Sia $S[i,j]$ un sottoproblema non vuoto, ed indentifichiamo m l'intervallo di $S[i,j]$ con il minor tempo di fine, allora:*

1. *il sottoproblema $S[i,m]$ è vuoto;*
2. *m è compreso in qualche soluzione ottima di $S[i,j]$.*

16.7.4 Dimostrazione che è una soluzione ottima

Dimostrazione. Fai riferimento alla spiegazione su youtube qui non riportata (importante). □

Conseguenze Non è più necessario analizzare tutti i possibili valori di k in quanto faccio una una scelta "ingorda", ma sicura: seleziono l'attività m con il minor tempo di fine. A questo punto non è più necessario analizzare due sottoproblemi; eliminando tutte le attività che non sono compatibili con la scelta ingorda mi resta solo il sottoproblema $S[m,j]$ da risolvere.

16.7.5 Scrittura dell'algoritmo

Algoritmo 16.7.1: Insieme indipendente di intervalli disgiunti

```
SET independentSet(int[] a, int[] b)
    { ordina a e b in modo che b[1] ⩽ b[2] ⩽ ··· ⩽ b[n] }
    // O(n) se già ordinati, O(n log n) altrimenti
    SET S ← Set
    S.insert (1) // inserisco il primo intervallo
    int ultimo ← 1 // ultimo intervallo inserito
    from i ← 2 until n do
        if a[i] ⩾ b[ultimo] then
            // gli intervalli sono indipendenti
            S.insert (i) // lo inserisco
            ultimo ← i // lo rendo l'ultimo inserito
    return S
```

Commento Divido il resto per il taglio della moneta più grande trovando il numero di monete massimo di quel taglio, dopodiché tolgo il valore della somma quelle monete dal resto.

Nota. Questo algoritmo non è applicabile al problema dell'insieme indipendente di intervalli *pesati*.

Ricapitolando Abbiamo cercato di risolvere il problema della selezione delle attività tramite programmazione dinamica, prima individuando una sottostruttura ottima, poi scrivendo una definizione ricorsiva per la dimensione della soluzione ottima.

Abbiamo poi dimostrato la proprietà della scelta greedy: dimostrando che per ogni sottoproblema, esiste almeno una soluzione ottima che contiene la scelta greedy. Infine abbiamo scritto un algoritmo iterativo che effettua sempre la scelta ingorda.

16.8 Resto

Definizione del problema Dati in input un insieme di "tagli" di monete, memorizzati in un vettore di interi positivi $t[1 \dots n]$ ed un intero R rappresentante il resto che dobbiamo restituire. Trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli di cui sopra, assumendo di avere un numero illimitato di monete per ogni taglio.

Più formalmente bisogna trovare un vettore x di interi non negativi tale che

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \qquad\qquad m = \sum_{i=1}^{n} x[i] \quad \text{è minimo}$$

16.8.1 Individuazione sottostruttura ottima

Sia $S(i)$ il problema di dare il resto pari ad i. Sia $A(i)$ una soluzione ottima del problema $S(i)$, rappresentata da un multi-insieme (un insieme nel quale lo stesso indice può comparire più di una volta); sia $j \in A(i)$ (j un possibile taglio). Allora, $S(i - t[j])$ è un sottoproblema di $S(i)$, la cui soluzione ottima è data da $A(i) - \{j\}$.

Quest'idea si traduce nella seguente definizione ricorsiva.

16.8.2 Definizione ricorsiva del costo della soluzione

Utilizziamo la tabella $DP[0 \dots R]$ per memorizzare le soluzioni e in $DP[i]$ memorizziamo il minimo numero di monete per risolvere il problema $S[i]$.

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min\limits_{1 \leqslant j \leqslant n} \{DP[i - t[j]] \text{ t.c. } t[j] \leqslant i\} + 1 & i > 0 \end{cases}$$

Il numero minimo di monete è 0 se non dobbiamo dare nessun resto (caso base), altrimenti cerchiamo fra tutti i possibili tagli quello che minimizza il numero di monete restituite.

16.8.3 Algoritmo basato su programmazione dinamica

Algoritmo 16.8.1: Programmazione dinamica applicata al problema del resto

```
restoDP(int[] t, int n, int R)
    // t:       tagli disponibili
    // n:       numero di monete
    // R:       il resto da dare

    DP ← new int[0...R]
    S ← new int[0...R]
    DP[0] ← 0 // caso base

    // Riempire la tabella DP
    from i ← 1 until R do
        DP[i] ← +∞

        from j ← 1 until n do // Riempio la tabella
            if t[j] ⩽ i and DP[i − t[j]] + 1 < DP[i] then
                // aggiorno il valore
                DP[i] ← DP[i − t[j]] + 1 // registro il valore
                S[i] ← j // la moneta da utilizzare per risolvere il problema quando il taglio è i

    while R > 0 do // ho resto da dare
        stampa t[S[R]] // stampo la moneta
        R ← R − t[S[R]] // decremento il resto
```

Commento $t[j] \leqslant i$ sta a significare che il taglio delle monete che possiamo scegliere dev'essere più piccolo del resto che dobbiamo dare.

Complessità $\mathcal{O}(nR)$ dato dai cicli innestati. Una soluzione dipendente dal resto R da dare non è ottimale, si può fare meglio di così.

Nota. Quest'algoritmo rappresenta la soluzione generale al problema e, a differenza dell'algoritmo ingordo (che vedremo più avanti), funziona per qualsiasi insieme di tagli di monete.

16.8.4 Scelta ingorda

Seleziona la moneta j più grande tale per cui $t[j] \leqslant R$, per poi risolvere il problema $S(R - t[j])$.

16.8.5 Scrittura dell'algoritmo

Algoritmo 16.8.2: Approccio ingordo al problema del resto

```
restoGreedy(int[] t, int n, int R, int[] x)
    { Ordina le monete in modo decrescente }
    // O(n) se già ordinato, O(n log n) altrimenti
    from i ← 1 until n do // O(n)
        // il numero di monete di taglio massimo
        x[i] ← ⌊R / t[i]⌋
        // calcolo il resto rimanente
        R ← R − x[i] · t[i]
    return R
```

Complessità Questo algoritmo ha complessità lineare, ossia $\mathcal{O}(n)$.

16.8.6 Dimostrazione che è una soluzione ottima

Nota. La seguente dimostrazione si riferiscere ai tagli $\bigcirc_1 = 50$, $\bigcirc_2 = 10$, $\bigcirc_3 = 5$, $\bigcirc_4 = 1$.

Sia x una qualunque soluzione ottima; quindi il resto R è esprimibile tramite una certa somma dei nostri possibili tagli, dove il numero dei tagli m è minimo:

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \qquad\qquad m = \sum_{i=1}^{n} x[i]$$

Sia m_k *la somma delle monete di taglio inferiore* a $t[k]$:

$$m_k = \sum_{i=k+1}^{4} x[i] \cdot t[i]$$

Se dimostriamo che la somma delle monete di taglio inferiore a k è minore del valore del taglio k-esimo ($\forall k : m_k < t[k]$), allora la soluzione (ottima) è proprio quella calcolata dall'algoritmo.

m_* denota l'insieme di monete con tagli inferiore a $\bigcirc_*$ centesimi, ad esempio m_2 denota l'insieme di monete con tagli inferiore a ($\bigcirc_2 =$) 10 centesimi. Mentre $x[*]$ simboleggia il *numero* di monete di quel taglio presenti nel resto della soluzione ottima, ad esempio $x[3]$ simboleggia il *numero* di monete di $\bigcirc_3 = 5$ centesimi.

$$\begin{aligned}
m_4 &= 0 && < 1 = t[4] \\
m_3 &= 1 \cdot x[4] && < 5 = t[3] \\
m_2 &= 5 \cdot x[3] + m_3 && \leqslant 5 + m_3 < 5 + 5 = 10 = t[2] \\
m_1 &= 10 \cdot x[2] + m_2 && \leqslant 40 + m_2 < 40 + 10 = 50 = t[1]
\end{aligned}$$

Nota. Fare riferimento alla spiegazione su youtube qui non riportata.

16.9 Approccio ingordo

Cercheremo di risolvere i successivi problemi avendo direttamente un approccio ingordo, senza passare prima per la programmazione dinamica. Come fare? bisogna:

- *evidenziare* i passi di decisione, *trasformando* il problema di ottimizzazione in un problema di "scelte" successive;
- *evidenziare* una possibile scelta ingorda, *dimostrando* che tale scelta rispetta il principio di scelta ingorda;
- *evidenziare* la sottostruttura ottima, *dimostrando* che la soluzione ottima del problema "residuo" dopo la scelta ingorda può essere unito a tale scelta;
- *scrivendo* un algoritmo top-down (anche iterativo), in alcuni casi sarà necessario pre-processare l'input.

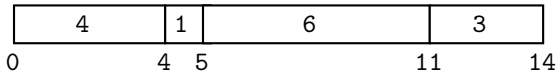
16.10 Scheduling

Definizione del problema Supponiamo di avere un processore e n job da eseguire su di esso, ognuno caratterizzato da un tempo di esecuzione $t[i]$ noto a priori. Trovare una sequenza di esecuzione (permutazione) che minimizzi il *tempo di completamento medio*.

Dato un vettore $A[1 \dots n]$ contenente una permutazione di $\{1, \dots n\}$, il *tempo di completamento* dell'h-esimo job nella permutazione è:

$$T_A(h) = \sum_{i=1}^{h} t[A[i]]$$

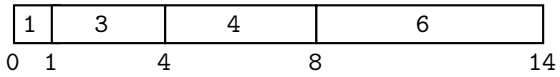
Ad esempio



dove il tempo di completamento medio è pari a

$$\frac{4 + 5 + 11 + 14}{4} = \frac{34}{4} = 8.5$$

Applicando l'algoritmo *Shortest Job First*:



dove il tempo di complemento è pari a

$$\frac{1 + 4 + 8 + 14}{4} = \frac{27}{4} = 6.75 < 8.5$$

Teorema (Scelta greedy)**.** *Esiste una soluzione ottima A in cui il job con minor tempo di fine m si trova in prima posizione ($A[1] = m$).*

Teorema (Sottostruttura ottima)**.** *Sia A una soluzione ottima di un problema con n job, in cui il job con minor tempo di fine m si trova in prima posizione. La permutazione dei seguenti $n - 1$ job in A è una soluzione ottima al sottoproblema in cui il job m non viene considerato.*

16.10.1 Dimostrazione

Fai riferimento alla spiegazione grafica su youtube qui non riportata.

16.11 Zaino frazionario

Riproponiamo un problema visto in precedenza, ma stavolta anzichè avere la limitazione di poter prenderere o non prendere un oggetto (chiamato problema dello zaino `0/1`), possiamo prenderne anche frazioni di esso (zaino *reale*).

Un approccio ingordo è quello di ordinare gli oggetti in ordine di *profitto specifico decrescente* (profitto su volume) e prendere quante più frazioni possibili degli elementi fino a riempire l'intera capacità dello zaino.

Algoritmo 16.11.1: Approccio ingordo al problema del resto

```
zaino(float[] p, float[] v, float C, int n, float[] x)
    { ordina p e v in modo che p[1]/w[1] ⩾ p[2]/w[2] ⩾ ··· ⩾ p[n]/w[n] }
    // O(n) se già ordinato, O(n log n) altrimenti
    from i ← 1 until n do
        x[i] ← min(C/w[i], 1) // ne prendo solo una frazione?
        C ← C − x[i] · w[i] // aggiorno la capacità residua
```

$x[i]$ rappresenta la frazione dell'oggetto i-esimo che deve essere presa.

16.11.1 Correttezza dell'algoritmo

Dimostrazione informale:

1. assumiamo che gli oggetti siano ordinati per profitto specifico decrescente;
2. sia x una soluzione ottima;
3. supponiamo che $x[1] < \min\left(\frac{C}{w[i]}, 1\right) < 1$;
4. allora possiamo costruire una nuova soluzione in cui $x'[1] = \min\left(\frac{C}{w[i]}, 1\right)$ e la proporzione di uno o più oggetti è ridotta di conseguenza;
5. otteniamo così una soluzione x' di profitto uguale o superiore, visto che il profitto specifico del primo oggetto è massimo.

16.12 Compressione di Huffman

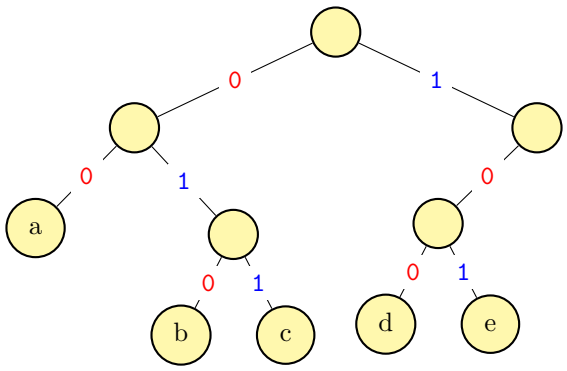
Per rappresentare in modo efficiente i dati (minimizzare lo spazio su disco occupato, minimizzare il tempo di trasferimento su disco...) abbiamo bisogno di adottare una qualche tecnica di compressione. Una fra le tante possibili è la *codifica dei caratteri*.

Nota. La codifica dei caratteri ha una complessità dipendente in percentuale dalle dimensioni del file di origine.

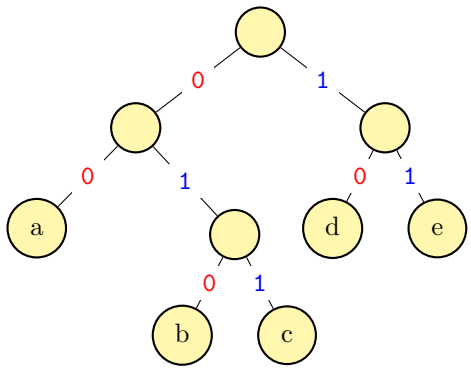
Nota. Un codice non deve essere mai prefisso di un altro codice, in quanto è una condizione necessaria per distinguerli.

16.12.1 Rappresentazione ad albero per la decodifica

Rappresenteremo il nostro risultato tramite un albero. La rappresentazione sulla sinistra non è la migliore possibile. Ed è possibile ottimizzare l'albero comprimendo i cammini per i caratteri `d` e `e` (come puoi vedere nella figura a destra). Modificando di conseguenza anche la codifica dei caratteri ed ottenendo una codifica ottimizzata.



a	b	c	d	e
00	010	011	100	111



a	b	c	d	e
00	010	011	10	11

Definizione formale del problema Dati in input un file F composto da caratteri di un certo alfabeto (che chiameremo Σ). Dobbiamo cercare di rappresentare il nostro file con il minor numero di bit possibili. Supponiamo che l'albero T rappresenti la nostra codifica. Per ogni carattere appartente all'alfabeto ($c \in \Sigma$), definiamo come $d_T(c)$ la profondità della foglia che rappresenta il carattere c. Quindi il codice per rappresentare c richiederà $d_T(c)$ bit. Infine se $f[c]$ è il numero di occorrenze di c in F, allora la *dimensione della codifica* è

$$C[F,T] = \sum_{c \in \Sigma} f[c] \cdot d_T(c)$$

- $f[c]$: frequenza del carattere c nell'alfabeto Σ;
- $d_T(c)$: profondità del nodo c nell'albero T, ovvero i bit necessari per la codifica di c.

Quindi una codifica C è data da un particolare albero T dove viene rappresentata e da un particolare file F.

16.12.2 Principio dell'algoritmo di Huffman

Vogliamo minimizzare la lunghezza della codifica per caratteri che compaiono più frequentemente: assegnando ai caratteri con frequenza minore codici corrispondenti a percorsi più lunghi all'interno dell'albero e a quelli di frequenza maggiore un percorso più corto.

Nota. Una certa codifica è progettata per un file specifico.

16.12.3 Algoritmo

Algoritmo 16.12.1: Approccio ingordo al problema del resto

```
TREE huffman(int[] c, int[] f, int n)
    // c[]1n:  caratteri dell'alfabeto
    // f[1...n]:  frequenze dei caratteri
    // n:  dimensione dell'alfabeto
    PRIORITYQUEUE Q ← MinPriorityQueue
    from i ← 1 until n do //                                   O(n)
        Q.inserisci(f[i], Tree(f[i], c[i])) //                 O(log n)
    from i ← 1 until n − 1 do // n:  radice                    O(n)
        // estraggo i 2 caratteri meno frequenti
        z1 ← Q.deleteMin
        z2 ← Q.deleteMin
        // Creo un nuovo nodo
        z ← Tree(z1.f + z2.f, nil)
        z.left ← z1
        z.right ← z2
        // Lo inserisco nella coda
        Q.inserisci(z.f, z)
    return Q.deleteMin
```

Siccome ogni volta devo estrarre i due elementi più piccoli faccio affidamento ad una MinPriorityQueue.

1. inserisco tutte le lettere con la priorità data dalla loro frequenza e con un nodo contenente sia la frequenza che la lettera associata. Dopodiché estraggo i due elementi più piccoli, credo un nuovo nodo contenente i due elementi appena estratti e restituisco il nodo così fatto al chiamante.

Funzionamento dell'algoritmo

1. rimuovo i due nodi con frequenza minore;
2. creo un nodo un'etichetta nulla e con frequenza pari alla somma delle frequenze dei nodi eliminati;
3. collego i due nodi con il nodo creato;
4. aggiungo il nodo così creato alla lista, mantenendo l'ordine;
5. si termina quando reasta un solo nodo nella lista;
6. al termine, si etichettano gli archi dell'albero con bit `0`, `1`.

Complessità Effettuo n volte operazioni che costano $\log n$, come le operazioni di inserisci o di deleteMin per una complessità totale di $\mathcal{O}(n \log n)$.

16.12.4 Dimostrazione di correttezza

Teorema. *L'output dell'algoritmo Huffman per un dato file è un codice a prefisso ottimo.*

Scegliere i due elementi con la frequenza più bassa conduce sempre ad una soluzione ottimale. Dato un problema sull'alfabeto Σ. È possibile costruire un sottoproblema con un alfabeto più piccolo in cui togliamo due caratteri e ne aggiungiamo uno fittizio.

16.12.5 Scelta ingorda

`MATERIALE MANCANTE`

16.13 Albero di copertura minimi

Problema Dato un grafo pesato, determinare come interconnettere tutti i suoi nodi minimizzando il costo del peso associato ai suoi archi (agli archi che andiamo a scegliere). Questo problema prende vari nomi, come albero di copertura minimo o albero di connessione minimo, in inglese *Minimum Spanning Tree*.

Risorse If you are curious about who is the cookie monster, then watch the MIT lecture on Minimum Spanning Tree of Professor Erik Demaine.

Definizione del problema Dati in input un grafo non orientato e connesso $G = (V, E)$ ed una funzione di peso (che determina il costo di connessione) $w: V \times V \to \mathbb{R}$ (data una coppia di nodi restituisce un numero reale che rappresenta il peso).

Nota. Poiché G non è orientato possiamo dire che $w(u, v) = w(v, u)$.

Un albero di copertura di G è un sottografo $T = (V, E_T)$ tale che T è un albero non radicato, i lati dell'albero siano un sottoinsieme di quelli del grafo ($E_T \subseteq E$) e che contenga tutti i vertici di G. Il problema consiste quindi nel trovare l'albero di copertura il cui *peso totale* sia minimo rispetto a ogni altro albero di copertura, dove il *peso totale* è dato da:

$$w(T) = \sum_{e \in T} w(e) = \sum_{[u,v] \in E_T} w(u, v)$$

Nota. Non è detto che l'albero di copertura minimo sia univoco.

16.13.1 Algoritmo generico

L'idea è di accrescere un sottoinsieme A di archi in modo tale che venga sempre rispettata la seguente invariante: A è un sottoinsieme di qualche albero di connessione minimo.

Teorema 11. *Un arco $[u, v]$ è detto sicuro per A se $A \cup \{[u, v]\}$ è ancora un sottoinsieme di qualche albero di connessione minimo.*

Algoritmo 16.13.1: Algoritmo generico per generare un albero di copertura minimo

```
SET mst-generico(GRAPH G, int[] w)
    SET A ← ∅ // parto da un insieme vuoto
    while A non forma un albero di copertura do
        Trova un arco sicuro [u, v]
        A ← A ∪ {[u, v]} // lo aggiungo all'albero
    return A
```

16.13.2 Dimostrazione

La dimostrazione banale è lasciata come esercizio al lettore. Scherzo, scherzo! (ma non la riporto comunque)

16.13.3 Algoritmo di Kruskal

Idea L'idea è quella di ingrandire sottoinsieme disgiunti (qualche idea sulla struttura dati da utilizzare?) di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero complessivo. Si individua quindi un arco sicuro scegliendo un arco *di peso minimo* fra tutti gli archi che connettono due alberi distinti (distinti da componenti connesse) della foresta.

L'algoritmo è ingordo perchè ad ogni passo si aggiunge alla foresta un arco con il peso minore.

Implementazione Per l'implementazione si usa una struttura Merge-Find Set (Mfset).

L'input non è un grafo G, ma un vettore di archi (EDGE[ ]) perché abbiamo bisogno di ordinare gli archi in base la peso. La rappresentazione dei grafi (per liste di adiacenza) non permette di ordinare gli archi in base al peso. La trasformazione non viene rappresentata nel seguente algoritmo.

Algoritmo 16.13.2: Algoritmo di Kruskal per la generazione di un MST

```
SET kruskal(EDGE[] A, int n, int m)
    // EDGE[]: vettore di archi
(1) SET T ← Set // insieme (inizialmente vuoto) che conterrà gli archi dell'albero minimo
    MFSET M ← Mfset(n) // insieme disgiunto grande

    // ordino per peso crescente
(2) { ordina A[1][m] in modo che A[1].peso ⩽ ··· ⩽ A[m].peso }
    int c ← 0 // quanti archi ho aggiunto
    int i ← 1 // quale arco sto guardando
(3) while c < n − 1 and i ⩽ m do // Termina quando l'albero è costruito
        // c < n − 1: ho raggiunto tutti gli archi necessari per fare un albero
        // i ⩽ m: ho esaurito tutti gli archi da guardare (per controllo)
        if M.find(A[i].u) ≠ M.find(A[i].v) then // non fanno parte dello stesso albero
            M.merge(A[i].u, A[i].v) // unisco gli insiemi disgiunti
            T.insert(A[i]) // inserisco l'arco all'albero
            c ← c + 1 // ho aggiunto un altro arco
        i ← i + 1 // guardo il prossimo arco
    return T // Ritorna l'albero di copertura minimo
```

Complessità Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati Merge-Find Set. Utilizziamo la versione con *euristica sul rango più compressione dei cammini* che rende tutte le operazioni con costo ammortizzato costante pari a $\mathcal{O}(1)$.

① l'inizializzazione richiede $\mathcal{O}(n)$ in quanto devono essere creati tutti gli alberi separati;

② l'ordinamento degli archi (m) richiede $\mathcal{O}(m \log n)$, siccome il numero degli archi è limitato superiormente da n^2 posso scrivere $\mathcal{O}(m \log n^2)$, per le proprietà dei logaritmi $\mathcal{O}(m \log n)$;

③ nel caso peggiore vengono eseguite $\mathcal{O}(m)$ operazioni sulla foresta di insiemi disgiunti (due find ed una merge), con tempo ammortizzato $\mathcal{O}(1)$. Per un totale di $\mathcal{O}(n + m \log n + m) = O(m \log n)$.

16.14 Algoritmo di Prim

Idea A differenza dell'algoritmo di Kruskal che formava molti alberi (rappresentati da insiemi disgiunti) che uniti successivamente, l'algoritmo di Prim procede mantenendo in A un singolo albero.

Nota. Durante l'esecuzione dell'algoritmo esiste un solo albero che rappresenta il quale rappresenta un sottoinsieme di un albero di copertura minimo del grafo totale.

L'albero parte da un vertice arbitrario r (la radice) e cresce fino a quando non ricopre tutti i vertici. Ad ogni passo viene aggiunto un arco leggero che collega un vertice in V_A con un vertice in $V - V_A$, dove V_A è l'insieme di nodi raggiunti da archi in A.

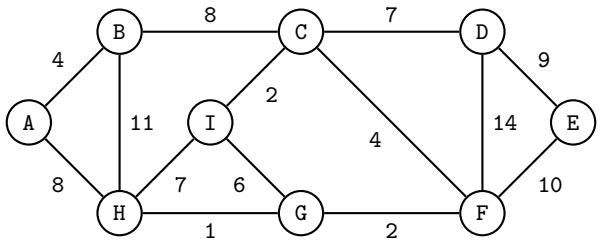
Implementazione Abbiamo bisogno di una struttura dati per i nodi non ancora presenti nell'albero. Durante l'esecuzione, i vertici che devono essere inseriti si trovano in una coda con min-priorità Q ordinata in base alla seguente definizione di priorità: "la priorità del nodo v è il peso minimo di un arco che collega v ad un vertice nell'albero, o $+\infty$ se tale arco non esiste".

L'albero è memorizzato tramite un *vettore dei padri*, in cui A è mantenuto implicitamente, infatti è definito $A = \{[v, p[v]] \mid v \in V - Q - \{r\}\}$ (Q archi non ancora raggiunti, $\{r\}$ radice).

Algoritmo 16.14.1: Algoritmo di Prim per la generazione di un MST

```
prim(Graph G, Node r, int[] p)
    // r:  nodo dalla quale parto
    // p:  vettore dei padri

    PriorityQueue Q ← MinPriorityQueue
    PriorityItem[] Pos ← new PriorityItem[1...G.n]

    // inserisco i nodi nella coda, memorizzando la loro posizione
(1) foreach u ∈ G.V() − {r} do
        Pos[u] ← Q.inserisci(u, +∞)

    // Inserisco il "nodo di partenza"
    Pos[r] ← Q.inserisci(r, 0)
    p[r] ← 0  // convenzione per indicare che non ha padre
(2) while not Q.isEmpty do // non ci sono più nodi
        Node u ← Q.deleteMin  // cancello e restituisco il nodo
        Pos[u] ← nil // non considero più quel nodo

        // per ciascun nodo adiacente a quello considerato
(3)     foreach v ∈ G.adj(u) do
            if Pos[v] ≠ nil and w(u,v) < Pos[v].priority then
                // Pos[v] ≠ nil:  è già stato visitato
                // w(u,v) < Pos[v].priority:
                Q.decrease(Pos [v], w(u,v))  // commento
                p[v] ← u  // commento
```

Esempio Fai riferimento alla spiegazione grafica, qui viene riportato solo lo schema sulla quale si basa.



Analisi della complessità $\mathcal{O}(m \log n)$

L'efficienza dell'algoritmo di Prim dipende dalla coda con priorità. Può essere implementato tramite heap binario oppure con un vettore non ordinato.

heap binario Nel caso si utilizzi un heap binario allora:

① l'inizializzazione costa $\mathcal{O}(m \log n)$;

② il ciclo principale viene eseguito $n - 1$ per una complessità di $\mathcal{O}(n)$ volte dove ogni operazione di deleteMin costa $\mathcal{O}(\log n)$;

③ il ciclo interno viene eseguito $\mathcal{O}(m)$ volte dove ogni operazione di decrease costa $\mathcal{O}(\log n)$.

Per un totale di $\mathcal{O}(n + m \log n + n \log n) = \mathcal{O}(m \log n)$.

Nota. L'algoritmo risulta asintoticamente uguale a quello di Kruskal.

vettore non ordinato Nel caso si utilizzi vettore un non ordinato:

① l'inizializzazione costa $\mathcal{O}(n)$;

② il ciclo principale viene eseguito $n - 1$ per una complessità di $\mathcal{O}(n)$ volte dove ogni operazione di deleteMin costa $\mathcal{O}(n)$;

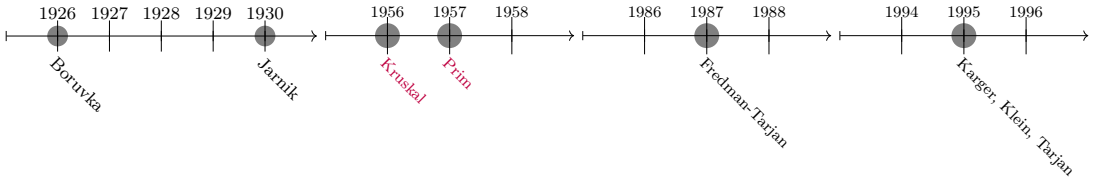
③ il ciclo interno viene eseguito $\mathcal{O}(m)$ volte dove ogni operazione di decrease costa $\mathcal{O}(1)$.

Per un totale di $\mathcal{O}(n + n^2 + m \cdot 1) = \mathcal{O}(n^2)$.

Nota. Cambiando la struttura dati cambia la complessità dell'algoritmo.

In definitiva se il grafo è *sparso* conviene utilizzare l'heap binario ($\mathcal{O}(m \log n)$), mentre se il grafo è *denso* (o addirittura completo) conviene utilizzare il vettore non ordinato ($\mathcal{O}(n^2)$).

16.14.1 Cenni storici



L'algoritmo di Fredman-Tarjan sfrutta un heap di Fibonacci che abbassa di molto la complessità dell'algoritmo ma ha dei costi associati molto grandi e quindi non viene utilizzato.

Nel '95 Karger, Klein e Tarjan hanno ideato un algoritmo probabilistico che risolve il problema in $\mathcal{O}(m + n)$ (molto spesso andare "a caso" conviene).

Se questo problema si possa risolvere in tempo lineare in modo deterministico è una questione ancora aperta.

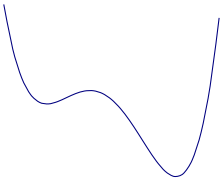
16.14.2 Conclusioni

Gli algoritmi ingordi sono semplici da programmare e molto efficienti. Inoltre quando è possibile dimostrare la proprietà di scelta ingorda, danno la soluzione ottima. Una soluzione sub-ottima è comunque accettabile.

Capitolo 17

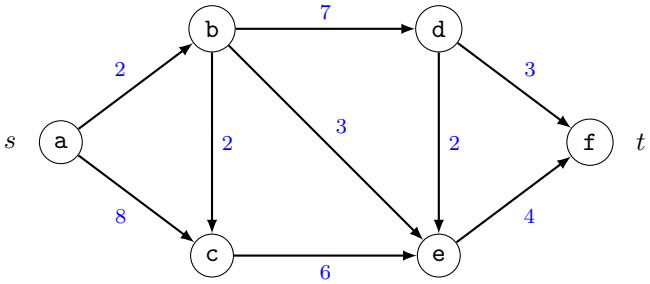
Ricerca locale

E' una tecnica di programmazione. L'idea generale è la seguente: se si conosce una soluzione ammissibile (non necessariamente ottima) ad un problema di ottimizzazione, si può cercare di trovare una soluzione migliore nelle "vicinanze" di quella precedente. Si continua in questo modo fino a quando non si è più in grado di migliorare



17.1 Problema di flusso massimo

Prima di definire il problema abbiamo bisogno di definire alcune entità e le corrispondenti proprietà.



Definizione 17.1.1 (rete di flusso)**.** Una rete di flusso $G = (V, E, s, t, c)$ è data da:

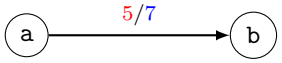
- un grafo orientato $G = (V, E)$;
- un nodo $s \in V$ detto sorgente;
- un nodo $t \in V$ detto pozzo;
- una funzione di capacità $c: V \times V \to \mathbb{R}^{\geq 0}$, tale per cui $(u, v) \notin R \Rightarrow c(u, v) = 0$.

Definizione 17.1.2. Un flusso in G è una funzione $f: V \times V \to \mathbb{R}$ (nota che può essere anche negativo) che gode delle seguenti proprietà:

- Vincolo sulla capacità;
- Antisimmetria;
- Conservazione del flusso.

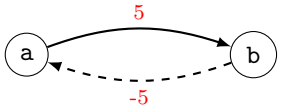
Vincolo sulla capacità Il flusso non deve eccedere la capacità sull'arco, ovvero

$$\forall u, v \in V : f(u, v) \leqslant c(u, v)$$



Antisimmetria Il flusso che passa dal nodo u al nodo v, è pari a meno il flusso dal nodo v al nodo u, ovvero

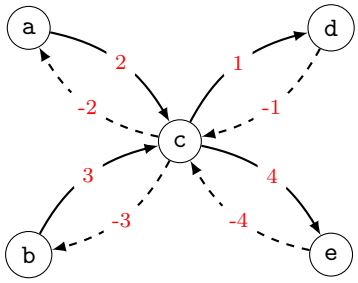
$$\forall u, v \in V : f(u, v) = -f(v, u)$$



Conservazione del flusso Per ogni nodo, la somma dei flussi entranti deve essere uguale alla somma dei flussi uscenti.

$$\forall u \in V - \{s,t\}: \sum_{v \in V} f(u,v) = 0$$

La sommatoria dei flussi che partono da u a qualunque altro nodo, ad esclusione della sorgente e del pozzo, deve essere 0.



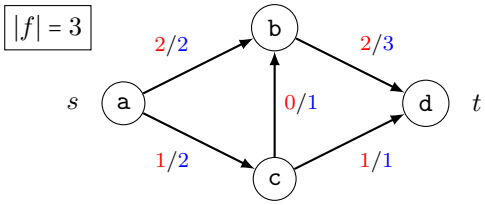
Definizione 17.1.3 (valore del flusso)**.** Il valore di un flusso f è definito come la quantità di flusso uscente da s.

$$|f| = \sum_{(s,v) \in E} f(s,v)$$

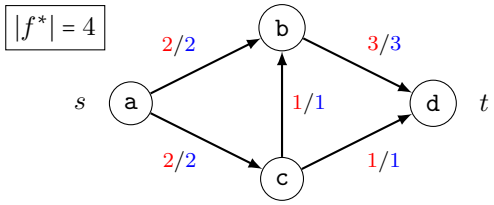
17.1.1 Problema del flusso massimo

Definizione 17.1.4 (Flusso massimo)**.** Data una rete $G = (V, E, s, t, c)$, trovare un flusso che abbia valore massimo fra tutti i flussi associabili alla rete.

$$|f^*| = \max\{\,|f|\,\}$$



(a) Il valore del flusso in questo caso è pari a 3 ma non è massimo.



(b) Il valore del flusso massimo per questa rete è 4.

Nota. Il nostro obiettivo è quindi quello di trovare il flusso massimo.

17.2 Metodo delle reti residue

Utilizziamo quindi il *metodo delle reti residue*. Il metodo consiste nel partire da un flusso "corrente" f, inizialmente nullo e 1) si "sottrae" il flusso attuale dalla rete iniziale, ottenendo così una rete residua; 2) si cerca un flusso g all'interno della rete residua; 3) si somma g ad f., ripetendo le azioni precedenti fino a quando non è più possibile trovare un flusso positivo g.

Se il procedimento viene svolto correttamente, è possibile dimostrare che questo approccio restituisce un flusso massimo.

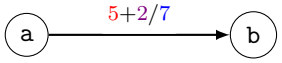
Nota. È un algoritmo di ricerca locale, senza una dimostrazione non è possibile dire che la soluzione sia ottima.

Definizione 17.2.1 (Flusso nullo)**.** Definiamo flusso nullo la funzione $f_0: V \times V \to R^{\leq 0}$ tale che $f(u,v) = 0$ per ogni $u, v \in V$.

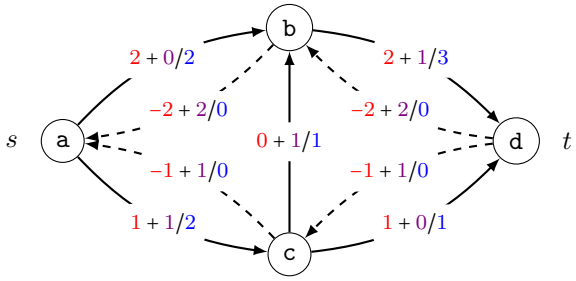
Definizione 17.2.2 (Somma di flussi)**.** Per ogni coppia di flussi f_1 e f_2 in G, definiamo il flusso somma $g = f_1 + f_2$ come un flusso tale per cui $g(u,v) = f_1(u,v) + f_2(u,v)$

Nota. È possibile invalidare il vincolo sulla capacità.

Definizione 17.2.3 (Capacità residua)**.** Definiamo capacità residua di un flusso f una funzione $c_f: V \times V \to R^{\geq 0}$ tale che $c_f(u,v) = c(u,v) - f(u,v)$
Per la definizione di capacità residua, si creano degli archi all'indietro: $c_f(b,a) = c(b,a) - f(b,a) = 0 - (-5) = 5$



Definizione 17.2.4 (Reti residue)**.** Data una rete di flusso $G = (V, E, s, t, c)$ e un flusso f su G, possiamo costruire una rete residua $G_f = (V, E_f, s, t, c_f)$, tale per cui $(u,v) \in E_f$ se e solo se $c_f(u,v) > 0$.



17.2.1 Algoritmo

Algoritmo 17.2.1: Schema generale

```
int[][] maxFlow(Graph G, Node s, Node t, int[][] c)
    f ← f_0 // inizializza un flusso nullo        r ← c // capacità iniziale
    // Fintanto che non rimane un flusso nullo
    repeat
        g ← trova un flusso r tale che |g| > 0, altrimenti f_0
        f += g // necessita di una dimostrazione
        r ← rete di flusso residua del flusso f in G
    until g == f_0
    return f // il flusso
```

Dimostrazione di correttezza

Lemma 12. *Se f è un flusso in G e g è un flusso in G_f, allora $f + g$ è un flusso in G.*

Vincolo sulla capacità

Dimostrazione.

$$
\begin{aligned}
g(u,v) &\leqslant c_f(u,v) && \text{sostituisco } c_f \\
g(u,v) &\leqslant c(u,v) - f(u,v) && \text{aggiungo termine uguale} \\
f(u,v) + g(u,v) &\leqslant c(u,v) - f(u,v) + f(u,v) && \text{raccolgo e semplifico} \\
(f+g)(u,v) &\leqslant c_f(u,v)
\end{aligned}
$$

□

Vincolo sulla capacità `MATERIALE MANCANTE`

Esempio Fare riferimento alla spiegazione grafica (minuto 35:27) qui non riportata (riferimento trasparenze 00 - 00.

Algoritmo 17.2.2: Algoritmo di Ford-Fulkerson

```
int nomeFunzione(parameters)
    return 0
```

Ricerca del cammino Manca del testo. Notare che le due cose non si escludono a vicenda. Il costo della visita è $\mathcal{O}(V + E)$.

Codice 17.1: inserire didascalia

```java
/**
 * Compute the max-flow using the Ford-Fulkerson algorithm
 * @param C the capacity matrix
 * @param s the source node
 * @param t the sink node
 * @return the flow matrix
 */
private static int[][] flow(int[][] C, int s, int t) {
    // Create an empty flow
    int[][] F = new int[C.length][C.length];

    // Visited array to perform DFS, initially empty
    boolean[] visited = new boolean[C.length];

    // Repeat until there is no path
    while (dfs(C, F, s, t, visited, Integer.MAX_VALUE) > 0) {
        Arrays.fill(visited, false);
    }
    return F;
}
```

Codice 17.2: inserire didascalia

```java
/**
 * Performs a DFS starting from node i and trying to reach node t.
 * @param C the capacity matrix; if capacity[x][y]>0, there is a edge from x to y
 * @param F the flow matrix to be computed
 * @param i the current node,
 * @param t the sink node
 * @param visited the boolean set containing the nodes that have been visited
 * @param min the smallest capacity found during the visit.
 * @returns the value of the additional flow found during the DFS
 */
private static int dfs(int[][] C, int[][] F, int i, int t, boolean[] visited, int min) {
    // If sink has been reached, terminate
    if (i == t) return min;

    visited[i] = true;
    for (int j = 0; j < C.length; j++) {
        if (C[i][j] > 0 && !visited[j]) {
            // Non-visited neighbor
            int val = dfs(C, F, j, t, visited, Math.min(min, C[i][j]));
            if (val > 0) {
                C[i][j] = C[i][j] - val;
                C[j][i] = C[j][i] + val;

                F[i][j] = F[i][j] + val;
                F[j][i] = F[j][i] - val;

                return val;
            }
        }
    }
```

```
    // The sink has not been found
    return 0;
}
```

Complessità Se le capacità della rete sono intere, l'algoritmo di Ford e Fulkerson ha complessità $\mathcal{O}((V+E)|f^*|)$ (il costo della visita in ampiezza) o complessità $\mathcal{O}(V^2|f^*|)$ nel caso pessimo (se fatto su una matrice).

L'algoritmo di Edmonds e Karp ha complessità $\Omega(V \cdot E^2)$ nel caso pessimo.

Entrambi i limiti superiori sono validi, si prende quindi il più basso fra i due.

Nota. Il cambio di notazione (ovvero segnalare i nodi con V (*vertex*) anziché con n, e i lati con E (*edges*) anziché con m) è necessario per non fare confusione con gli esercizi.

Lemma 13. *Il numero totale di aumenti di flusso eseguiti dall'algoritmo di Edmonds e Karp è $\mathcal{O}(V \cdot E)$.*

17.2.2 Dimostrazione complessità

`MATERIALE MANCANTE`

17.2.3 Dimostrazione correttezza

Per fare la dimostrazione di correttezza dobbiamo (re)introdurre delle definizioni.

Definizione 17.2.5 (Taglio)**.** Un taglio (S,T) della rete di flusso $G = (V, E, s, t, c)$ è una partizione V in S e $T = V - S$ tale che $s \in S$, $t \in T$.

Mettere schema a destra.

Definizione 17.2.6 (Capacità di un taglio)**.** La capacità $c(S,T)$ attraverso il taglio S,T è pari a

$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$

Mettere schema a destra.

Definizione 17.2.7 (Flusso di un taglio)**.** Se f è un flusso in G, il flusso netto $F(S,T)$ attraverso (S,T) è pari a

$$F(S,T) = \sum_{u \in S, v \in T} f(u,v)$$

Mettere schema a destra.

Lemma 14 (Valore del flusso di un taglio)**.** *Dato un flusso f e un taglio (S,T), la quantità di flusso $F(S,T)$ che attraversa il taglio è uguale a $|f|$.*

$$\begin{aligned} F(S,T) &= \sum_{u \in S, v \in T} f(u,v) && \text{(} V = T - S \text{)} \\ &= \sum_{u \in S, v \in V} f(u,v) - \sum_{u,v \in S} f(u,v) && \text{(conservazione del flusso e antisimmetria)} \\ &= \underbrace{\sum_{u \in S - \{s\}, v \in V} f(u,v)}_{0} + \sum_{v \in V} f(s,v) + \underbrace{\sum_{u,v \in S} f(u,v)}_{0} && \text{(per definizione)} \\ &= \sum_{v \in V} f(s,v) = f \end{aligned}$$

Un taglio definisce un limite superiorie al flusso massimo che può essere presente nella rete.

Lemma 15 (Capacità del taglio)**.** *Il flusso massimo è limitato superiormente dalla capacità del taglio minimo (ovvero il taglio la cui capacità è minotr fra tutti i taglio).*

`MATERIALE MANCANTE`

La somma dei flussi dev'essere necessariamente minore della somma delle capacità.

`MATERIALE MANCANTE`

Quindi, il valore del flisso è limitato superiormente dalla capacità di tutti i possibili tagli.

Teorema 16 (Teorema del flusso massimo)**.** *Le seguenti affermazioni sono equivalenti:*

- *a*

- b
- c

DA FINIRE

Capitolo 18

Backtrack

Problemi tipici Problemi tipici sono:

- *elencare tutte le possibili soluzioni* (enumerazione), come ad esempio elencare tutte le possibili permutazioni di un insieme;
- *costruire almeno una soluzione del problema*, in questo caso si utilizza l'algoritmo di enumerazione fermandosi alla prima soluzione dispoonibile;
- *contare le soluzioni*, in alcuni casi è possibile contare in modo analitico, in altri casi si costruiscoo le soluzioni e si contano;
- *trovare le soluzioni ottimali*, si enumerano tutte le soluzioni che vengono valutate tramite una funzione di costo. In questo caso si utilizzano anche altre tecniche di programmazione (come la programmazione dinamica, o la tecnica greedy).

18.1 Eumerazione

Per costruire tutte le soluzioni, si utilizza un approccio di "forza bruta" (*brute force*). In alcuni casi è l'unica strada percorribile. Fortunatamente i processori moderni rendono affrontabili problemi considerati di dimensioni medio-piccole. Inoltre, a volte lo spazio delle soluzioni non deve essere analizzato interamente.

Idea (backtracking). "Prova a fare qualcosa, e se non va bene, disfalo e prova qualcos'altro"

Il backtracking è una tecnica algoritmica che, come altre, deve essere personalizzata per ogni applicazione individuale. Dobbiamo quindi trovare un metodo sistematico per iterare su tutte le possibili istanze di uno spazio di ricerca.

Organizzazione del problema Una soluzione viene rappresentata come il vettore $S[1][n]$; il contenuto degli elementi $S[i]$ è una *sequenza di scelte* (possibili) C dipendenti dal problema.

Ad esempio preso un insieme C che rappresenta un insieme generico possiamo avere possibili *permutazioni* o *sottoinsiemi*. Se C è un insieme di mosse ottenerremo una *sequenza di mosse*; se nell'insieme C sono contenuti archi di un grafo, allora otterremo possibili percorsi del grafo; e così via.

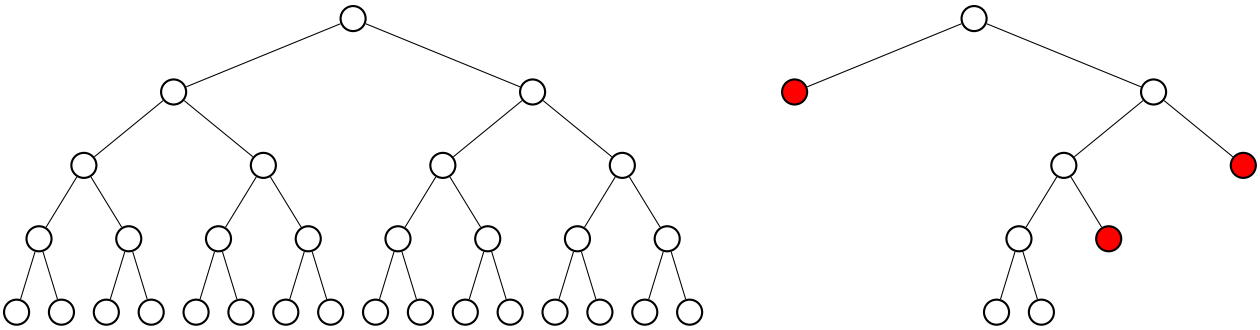
Scherma di risoluzione generale Ad ogni passo, partiamo da una soluzione generale $S[1][k]$ in cui $k \geqslant 0$ scelte sono state prese (ovvero abbiamo effettuato k scelte, dove k può essere anche nullo).

Se la sequenza di scelte che abbiamo effettuato ($S[1][k]$) costituiscono una soluzione ammissibile allora la "processiamo". Può essere quindi stampata, contata, valutata, oppure si può decidere di terminare elencando tutte le possibili soluzioni.

Indipendentemente dalla precedente se $S[1][k]$ non rappresenta una soluzione completa, allora proviamo, se è possibile, ad *estendere* $S[1][k]$ con una delle possibili scelte in una soluzione $S[1][k+1]$; altrimenti "cancelliamo" l'ultima scelta effettuata $S[k]$ tornando sui nostri passi (facendo quindi *backtracking*) e ripartendo dalla soluzione precedente $S[1][k-1]$.

Rappresentazione del problema Il nostro problema viene rappresentato da un *albero di decisione* nel quale:

- lo *spazio di ricerca* viene rappresentato dall'albero stesso;
- la *soluzione parziale vuota* (ossia quella dove non abbiamo preso nessuna decisione) è la radice;



(a) Lo spazio di ricerca di un determinato problema è rappresentato da un albero delle decisioni

(b) *pruning* dei rami

Figura 18.1: Un albero di decisione ed il suo corrispondente albero potato dei rami che non portano a soluzioni ammissibili.

- le *soluzioni parziali* sono rappresentate dai nodi interni;
- le *soluzioni ammissibili* vengono rappresentate dalle foglie.

Pruning I "rami" dell'albero che sicuramente non portano a soluzioni ammissibili possono essere "potati" (*pruned*). La valutazione della "potatura" viene effettuata nelle soluzioni parziali radici del sottoalbero da potare.

Nota. Il pruning riduce drasticamente lo spazio delle soluzioni del problema.

Approcci Ci sono due possibili approcci alla tecnica del backtracking:

1. la prima consiste nello sviluppo di un algoritmo *ricorsivo* che lavora tramite una visita in profondità nell'albero delle scelte;
2. la seconda consiste nello sviluppo di un algoritmo *iterativo* ed utilizza un approccio ingordo (*greedy*), eventualmente tornando sui propri passi (effetuando quindi *backtracking*).

Sia il problema dell'inviluppo convesso, che quello della corrispondenza fra stringhe (*string matching*) utilizzano questo approccio.

In base alle i scelte già effettuate decido quali sono le mie future scelte.

Algoritmo 18.1.1: Schema generale per il problema dell'enumerazione

```
boolean enumerazione(ITEM[] S, int n, int i, ...)
    // S:       vettore contenente le soluzioni parziali S[1][i]
    // n:       il numero massimo di scelte possibili
    // i:       indice corrente
    // ...:     parametri addizionali dipendenti dal problema

    SET C = scelte(S, n, i, ...) // Determina C in funzione di S[1][i-1]
    // C:       l'insieme dei possibili candidati per estendere la soluzione
    foreach c ∈ C do // per ogni possibile scelta fra quelle possibili
        S[i] ← c  // registro la scelta

        if isAdmissible(S, n, i) then
            // S[1][i] è una soluzione ammissibile
            if processSolution(S, n, i, ...) then
                // vogliamo bloccare l'esecuzione alla prima soluzione possibile
                return true

        // non decido di fermarmi alla prima soluzione ammissibile
        if enumerazione(S, n, i+1, ...) then
            // effettuo la i+1-esima scelta
            return true
    // non ho trovato la soluzione cercata
    return false
```

18.1.1 Problemi che trattano di sottoinsiemi

Definizione del problema Elencare tutti i sottoinsiemi dell'insieme $\{1, \ldots, n\}$

Algoritmo 18.1.2: Primo tentativo

```
subSets(int[] S, int n, int i)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        if i == n then
            processSolution(S, n)
        subSets(S, n, i + 1)
```

Algoritmo 18.1.3: Versione "più pulita"

```
subSets(int[] S, int n, int i)
    foreach c ∈ {0,1} do
        S[i] ← c
        if i == n then
            processSolution(S, n)
        else
            subSets(S, n, i + 1)
```

Fare riferimento alla spiegazione grafica qui non riportata.

Considerazioni Non c'è pruning. Tutto lo spazio possibile viene esplorato. Ma questo avviene per definizione. Questo porta ad una complessità di $\mathcal{O}(n \cdot 2^n)$. È possbile pensare ad una soluzione iterativa, ad-hoc? (che non utilizza la tecnica del backtracking)

Algoritmo 18.1.4: Versione "più pulita"

```
subSets(int[] S, int n, int i)
    from j = 0 until 2^n − 1 do // O(n^2)
        stampa "{"
        from i = 0 until n − 1 do // O(n)
            if (j and 2^i) ≠ 0 then
                stampa i
        stampa "}"
```

Definizione del problema Elencare tutti i sottoinsiemi di dimensione k di un insieme $\{1, \dots, n\}$

Algoritmo 18.1.5: Tentativo 1

```
subSets(int[] S, int n, int k, int i)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        if i == n then
            int count ← 0
            from j ← 1 until n do
                count += S[j]
            if count == k then
                // ho finito
                processSolution(S, n)
    subSets(S, n, k, i+1)
```

Algoritmo 18.1.6: Tentativo 2

```
subSets([...], int count)
    SET C ← iif(i ⩽ n, {0,1}, ∅)
    foreach c ∈ C do
        S[i] ← c
        count += S[i]
        // logica dell'algoritmo
        if i == n and count == k then
            processSolution(S, n)
        subSets(S, n, k, i+1, count)
        count -= S[i]
```

Tengo conto del contatore nelle chiamate successive e gli sottraggo i valori aggiunti nelle chiamate di *backtracking*.

Algoritmo 18.1.7: Elencare tutti i sottoinsiemi di dimensione k di un insieme $\{1, \dots, n\}$

```
subSets(int[] S, int n, int k, int i, int count)
    // n:       numeri di elementi
    // k:       numeri di elementi considerati
    // i:       la scelta che ho già fatto
    // se ho già raggiunto k o non ci sono abbastanza bit per arrivare a k non ho più bisogno di visitare il
       sotto-albero delle scelte relativo)
    SET C ← iif(count < k and count + (n − i + 1) ⩾ k, {0,1}, ∅)
    // count < k:  ho ancora la possibilità di accendere un bit
    // count + (n − i + 1) ⩾ k:  ho ancora abbastanza bit da accendere per raggiungere k
    foreach c ∈ C do
        S[i] ← c
        // i bit scelti non cambiano nelle chiamate successive
        count += S[i]  // sommo i bit a 1
        if count == k then
            processSolution(S, i)
        else
            subSets(S, n, k, i+1, count)
        // quando effettuo il backtrack devo tornare allo stato precedente
        count -= S[i]  // sottraggo i bit a 1
```

Considerazioni Abbiamo imparato che "specializzando" l'algoritmo generico, possiamo ottenere una versione più efficiente. Tuttavia abbiamo ottenuto solo un miglioramento parziale (verso $n/2$).

Nota. È difficile ottenere la stessa efficienza con un algoritmo iterativo.

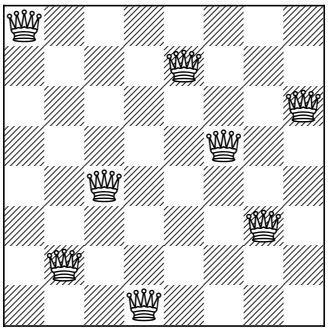
18.2 Permutazioni

Definizione del problema Stampa tutte le permutazioni di un insieme A. L'insieme dei candidati dipende dalla soluzione parziale *corrente*.

Algoritmo 18.2.1

```
permutazioni(SET A, int n, ITEM [] S, int i)
    // A:      insieme dalla quale prendo gli oggetti
    // n:      numero di permutazioni
    foreach c ∈ A do
        S[i] ← c  // scelgo l'oggetto
        A.remove(c)  // lo tolgo dall'insieme
        if A.isEmpty then
            processSolution(S, n)
        else
            // l'insieme A ha un elemento in meno
            permutazioni(A, n, S, i + 1)
        A.insert(c)
```

18.3 Problema delle otto regine

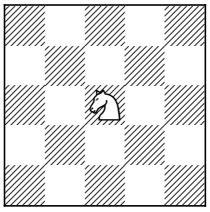


Definizione del problema Il problema consiste nello posizionare n regine in una scacchiera $n \times n$, in modo tale che nessuna regina ne "minacci" un'altra.
Idea. Ogni riga ed ogni colonna deve contenere esattamente una regina. Rappresentiamo la scacchiera con un vettore $S[1][n]$, effettuiamo *pruning* eliminando le diagonali.
Il numero di soluzioni ammonta a $n! = 8! = 40\,320$ della quale solo $15\,720$ vengono esplorate.

Esiste un algoritmo *probabilistico* che risolve il problema in tempo lineare, tuttavia non garantisce che la terminazione sia sempre corretta. Quindi viene fatto eseguire più volte fintanto che restituisce una soluzione corretta. Il meccanismo consiste nel partire da una soluzione "ragionevolmente buona" e di muovere il pezzo con il più grande numero di conflitti nella posizione, all'interno della stessa colonna, in cui ne genera il numero minore. La soluzione iniziale è scelta in modo "casuale" (ne parleremo più approfonditamente nel prossimo capitolo). L'algoritmo si ferma quando non ci sono più pezzi da muovere.

18.4 Giro di cavallo

Definizione del problema Lo scopo è trovare un "giro di cavallo", ovvero un percorso di mosse valide del cavallo in modo che ogni casella venga visitata al più una volta.



18.4.1 Algoritmo risolutivo

Algoritmo 18.4.1: Problema del giro di cavallo

```
cavallo(int[][] S, int i, int x, int y)
    SET C ← mosse(S,x,y)
    foreach c ∈ C do
        S[x,y] ← i  // ho raggiunto la posizione (x,y) all'i-esimo passo
        if i == 64 then
            processSolution(S)
            return true
        else if cavallo(S, i+1, x+m_x[c], y+m_y[c]) then
            // effettuo l'i-esima mossa
            return true
        S[x,y] ← 0
    return false
// trova le possibili mosse
mosse(int[][] S, int x, int y)
    SET C ← Set
    // fra tutte le possibili mosse
    from i ← 1 until 8 do
        // calcola la nuova posizione
        n_x ← x + m_x[i]
        n_y ← y + m_y[i]
        if 1 ⩽ n_x ⩽ 8 and 1 ⩽ n_y ⩽ 8 and S[n_x,n_y] ← 0 then
            // la posizione è libera
            C.insert(i)
    return C
```

Le prime due condizioni ($1 \leqslant n_x \leqslant 8$ **and** $1 \leqslant n_y \leqslant 8$ controllano se rientro nei limiti della scacchiera, mentre la terza ($S[n_x, n_y] \leftarrow 0$) controlla se la posizione non è stata toccata.

18.5 Sudoku

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

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

Figura 18.2: Il gioco del sudoku

18.5.1 Algoritmo risolutivo

Algoritmo 18.5.1: Sudoku

```
boolean sudoku(int[][] S, int i)
    int x ← i mod 9
    int y ← ⌊i/9⌋
    SET C ← Set

    // il numero è già stato scelto
    if S[x,y] ≠ 0 then
        S.insert(S[x,y])
    else
        // inseriamo un numero
        from c ← 1 until 9 do
            if verifica(S, x, y, c) then
                C.insert(c)

    int old ← S[x,y]
    foreach c ∈ C do
        S[x,y] ← c

        // arriviamo all'ultima casella
        if i == 80 then
            processSolution(S,n)
            return true

        if sudoku(S, i + 1) then
            return true

    S[x,y] ← old
    return false
```

```
// se posso inserire un numero in quella cella
verifica(int[][] S, int x, int y, int c)
    from j ← 0 until 8 do
        // controllo sulla colonna
        if S[x,j] == c then
            return false

        // controllo sulla riga
        if S[j,y] == c then
            return false

    int b_x ← ⌊x/3⌋
    int b_y ← ⌊y/3⌋
    from i_x ← 0 until 2 do
        from i_y ← 0 until 2 do
            // controllo sottotabella
            if S[b_x · 3 + i_x, b_y · 3 + i_y] = c then
                return false
```

18.6 Inviluppo convesso

Definizione 18.6.1 (poligono convesso)**.** Un poligono nel piano è convesso se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso, incluso il bordo.

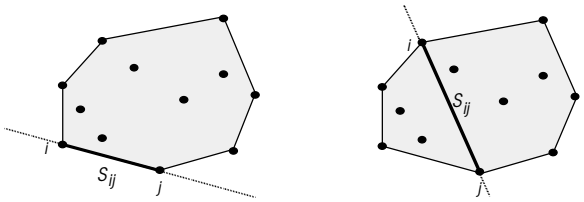
Definizione del problema Dati n punti $p_1, \ldots, p_n$ nel piano, con $n \geqslant 3$, l'inviluppo convesso (*convex null*) è il più piccolo poligono convesso che li contiene tutti.

18.6.1 Algoritmo inefficiente

Un poligono può essere rappresentato per mezzo dei suoi spigoli. Si consideri la retta che passa per una coppia di punti (p_i, p_j), che divide il piano in due semipiani chiusi. Se tutti i rimanenti $n-2$ punti stanno dalla *stessa parte*, allora lo spigolo S_{ij} fa parte dell'inviluppo convesso.

Algoritmo 18.6.1: inserire didascalia

Data una retta definita dai punti p_1 e p_2, determinare se due punti p e q stanno nello stesso semipiano definito dalla retta.



boolean stessaparte(POINT p_1, POINT p_2, POINT p, POINT q)

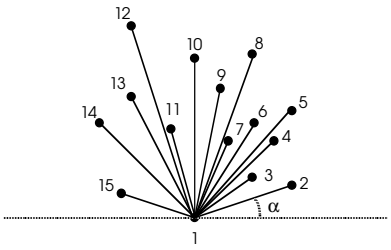
```
float dx ← p2.x − p1.x
float dy ← p2.y − p1.y
float dx1 ← p.x − p1.x
float dy1 ← p.y − p1.y
float dx2 ← q.x − p2.x
float dy2 ← q.y − p2.y
// se ⩾ 0 allora sono dalla stessa parte
return ((dx · dy1 − dy · dx1) · (dx · dy2 − dy · dx2) ⩾ 0)
```

Complessità Prendiamo tutte le coppie di punti (n^2) e controlliamo se tutti gli altri punti ($n-2$) stanno "dall'altra parte". La complessità ammonta a $\mathcal{O}(n^2 \cdot (n-2) = \mathcal{O}(n^3)$

18.6.2 Algoritmo di Graham

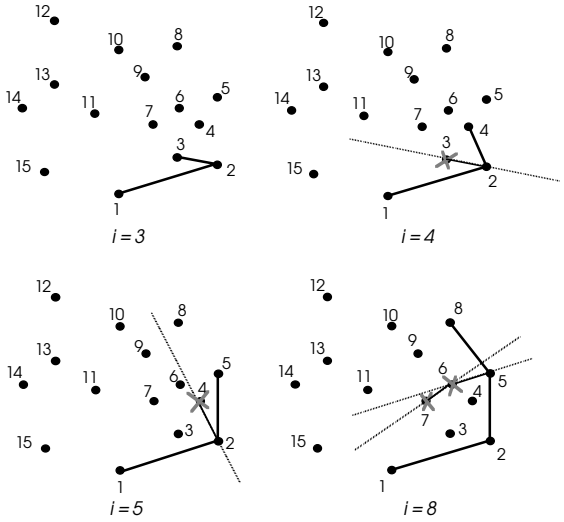
Prima fase

- il punto con ordinata minima fa parte dell'inviluppo convesso;
- si ordinano i punti in base all'angolo formato dalla retta passante per il punto con ordinata minima e la retta orizzontale.



Seconda fase

- inserisci p_1, p_2 nell'inviluppo corrente;
- per tutti i punti $p_i = 3, \ldots, n$:
  1. siano p_h e p_j, con $h < j = i - 1$, gli ultimi due vertici dell'inviluppo corrente;
  2. scandisci a "ritroso" i punti nell'inviluppo "corrente" ed elimina p_j se stessaparte(p_j,p_h,p_1,p_i) == **false**;
  3. termina tale scansione se p_j non deve essere eliminato;
  4. aggiungi p_i all'inviluppo "corrente".



Algoritmo 18.6.2: Algoritmo di Graham

```
STACK graham(POINT p, int n)
    int min = 1
    // trovo il minimo
    from i ← 2 until n do
        if p[i].y < p[min].y then
            min ← i
    p[1] ↔ p[min]
    { riordina p[2,...,n] in base all'angolo formato rispetto all'asse orizzontale quando sono connessi con
      p[1]}
    { elimina eventuali punti "allineati" tranne i più lontani da p1, aggiornando n }
    STACK S ← Stack
    // inserisci p1, p2 nell'inviluppo corrente
    S.push(p1)
    S.push(p2)
    // per tutti gli altri punti
    from i ← 3 until n do
        // escludo tutti i punti all'interno dell'inviluppo
        while not stessaparte(S.top, S.top2, p1, pi) do // top2 restituisce il secondo elemento
            S.pop
        S.push(pi)
```

Complessità L'algoritmo di Graham ha complessità $\mathcal{O}(n \log n)$ in quanto è dominato dall'ordinamento dei punti.

18.7 Algoritmi probabilistici

Idea. Se non sapete cosa fare fate una scelta casuale.

Nota. Il calcolo delle probabilità è applicato ai dati di output, non ai dati di input.

18.8 Algoritmi Montecarlo

Idea. Sono algoritmi in cui la correttezza è probabilistica.

`MATERIALE MANCANTE`

18.8.1 Test di primalità

Teorema (Piccolo teorema di fermat)**.** *Per ogni numero primo n e ogni $b \in [2, \dots, n-1]$:* $b^{n-1} \bmod n = 1$.

Algoritmo 18.8.1: Test di primalità

```
isPrime(int n)
    b ← random(2, n − 1)
    if b^{n−1} mod n ≠ 1 then
        // non può essere un numero primo
        return false
    // è un numero primo
    return true
```

18.9 Algoritmi Las Vegas

Idea. Sono algoritmi corretti, in cui il funzionamento è probabilistico.

`MATERIALE MANCANTE`

18.9.1 Statistiche d'ordine

`MATERIALE MANCANTE`

Algoritmo 18.9.1: inserisci didascalia

```
heapSelect(ITEM[] A, int n, int k)
    heapBuild(A)
    for i = 1 until k − 1 do
        deleteMin(A, n)
    // restituisce il k-esimo valore
    return deleteMin(A, n)
```

Analisi della complessità Non va bene per $k = n/2$, perchè viene $\mathcal{O}(n + n/2 \log n) = \mathcal{O}(n \log n)$

Cambiamo tattica. Utilizziamo una tecnica dividi-et-impera simile al quickSort. Ma, a differenza di quest'ultimo, non è necessario cercare in entrambe le partizioni.

Algoritmo 18.9.2: inserisci didascalia

```
selection(ITEM[] A, int start, int end, int k)
    if start == end then
        // caso base: ho un solo elemento
        return A[start]
    else
        // calcolo indici
        int j = perno(A, start, end)  // perno
        int q = j − start + 1  // mediana
        if k == q then
            // ho trovato il mio elemento
            return A[j]
        else if k < q then
            // cerco a sinistra
            return selection(A, start, j − 1, k)
        else
            // cerco a destra
            return selection(A, j + 1, end, k − q)
```

Analisi della complessità Effettuiamo un'analisi simile a quella fatta per il quickSort.

Caso pessimo Il caso pessimo è lo stesso del quickSort (che è il caso in cui il vettore sia già ordinato). In quanto dividiamo il vettore con 0 elementi a sinistra ed n elementi a destra della partizione.

$$T(n) = \begin{cases} 1 & n \leqslant 1 \\ T(n-1) + n & n > 1 \end{cases}$$
$$T(n) = \mathcal{O}(n^2)$$

Quindi nel caso pessimo questo algoritmo risulta peggiore di ordinare il vettore ($\mathcal{O}(n \log n)$) e di andare a fare la ricerca.

Caso ottimo Nel caso ottimo ogni volta divido esattamente a metà

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

Caso medio Assumiamo che perno restituisca con la stessa probabilità una qualsiasi posizione j del vettore A:

$$T(n) = n - 1 + \frac{1}{n} \sum_{q=1}^{n} T\left(\max\{q-1, n-q\}\right) \qquad \text{per } n > 1$$
$$\leqslant n - 1 + \frac{2}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} T(q)$$

$\frac{1}{n}$ la media su tutti i possibili valori di q $\sum_{q=1}^{n}$ rappresenta la dimensionalità di j, $q-1$ gli elementi a sinistra, $n-q$ a destra rispettivamente.

$\exists c > 0, \exists m \geqslant 0 : T(n) \leqslant cn, \forall n \geqslant m$

$$
\begin{aligned}
T(n) &\leqslant n - 1 + \frac{2c}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} q && \text{divido la sommatoria a metà}\\
&\leqslant n + \frac{2c}{n}\left(\sum_{q=1}^{n-1} q - \sum_{q=1}^{\lfloor n/2 \rfloor - 1} q\right) && \text{algebra}\\
&= n + \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor(\lfloor n/2 \rfloor - 1)}{2}\right) && \text{semplifico}\\
&\leqslant n + \frac{c}{n}\big(n(n-1) - (n/2+1)(n/2)\big) && \text{eseguo i calcoli}\\
&= n + c(n-1) - (c/2)(n/2+1) && \text{eseguo i calcoli}\\
&= n + cn - c - cn/4 - c/2 \\
&= cn\left(\frac{1}{c} + \frac{3}{4} - \frac{3}{2n}\right) \leqslant cn\left(\frac{1}{c} + \frac{3}{4}\right) \leqslant cn && \text{la mia disequazione risulta vera}
\end{aligned}
$$

Siamo partiti dall'assunzione che j assuma equiprobabilisticamente tutti i valori compresi fra 1 ed n. Se questa assunzione non fosse vera allora i nostri calcoli non avrebbero alcun fondamento. Forziamo quindi l'assunzione iniziale scegliendo un valore casuale come perno ($A[\mathsf{random}(start,\ end)]$) a differenza di prima dove prendevamo il primo valore ($A[start]$). Questo accorgimento vale anche per quickSort. La complessità nel caso medio diventa quindi $\mathcal{O}(n)$.

18.9.2 Selezione deterministica

Supponiamo di avere un algoritmo "black box" che mi ritorni un valore che dista al più $\frac{3}{10}n$ dal mediano nell'ordinamento.

Implementazione L'implementazione dell'algoritmo consiste nei seguenti passaggi:

- suddividi i valori in gruppi di 5. Chiameremo l'i-esimo gruppo S_i, con $i \in [1, \lceil n/5 \rceil]$
- trova il mediano M_i di ogni gruppo S_i
- tramite una chiamata ricorsiva, trova il mediano m dei mediani $[M_1, M_2, \ldots, M_{\lceil n/5 \rceil}]$
- usa m come pivot e richiama l'algoritmo ricorsivamente sul vettore opportuno, come nella selection randomizzata
- quando la dimensione scende sotto una certa dimensione, possiamo utilizzare un algoritmo di ordinamento per trovare il mediano

Algoritmo 18.9.3: inserisci didascalia

```
ITEM[] select(ITEM[] A, int primo, int ultimo, int k)
    // se la dimensione è inferiore ad una soglia (10), ordina il vettore e
    // restituisci il k-esimo elemento di A[primo][ultimo]
    if ultimo − primo + 1 ⩽ 10 then
        insertionSort(A, primo, ultimo) // Versione con indici inizio/fine
        return A[primo + k − 1]

    // divide A in ⌈n/5⌉ sottovettori di dim. 5 e ne calcola la mediana
    M ← new int[1 . . . ⌈n/5⌉]
(4) from i ← 1 until ⌈n/5⌉ do
        M[i] ← median5(A, primo + (i − 1) · 5, ultimo)

    // individua la mediana delle mediane e usala come perno
(5) ITEM m ← select(M, 1, ⌈n/5⌉, ⌈(⌈n/5⌉)/2⌉)
    int j ← perno(A, primo, ultimo, m) // Versione con m in input

    // calcola l'indice q di m in [primo...ultimo]
    int q ← j − primo + 1

    // confronta q con l'indice cercato e ritorna il valore conseguente
    if q == k then
        return m
    else if q < k then
(6)     return select(A, primo, q − 1, k)
    else
(3)     return select(A, q + 1, ultimo, k − q)

// calcola la mediana fra 5 elementi
int median5
    // Restituisco la mediana
    return m
```

Analisi della complessità

④ il calcolo dei mediani $M[]$ richiede al più $6\lceil n/5 \rceil$ confronti;

⑤ la prima chiamata ricorsiva dell'algoritmo select viene effettuata su $\lceil n/5 \rceil$ elementi;

⑥ la seconda chiamata ricorsiva dell'algoritmo select viene effettuata al massimo su $7\frac{n}{10}$ elementi (esattamente $n - 3\lceil \frac{\lceil n/5 \rceil}{2} \rceil$).

L'algoritmo select esegue nel caso pessimo $O(n)$ confronti:

$$T(n) = T\left(\tfrac{n}{5}\right) + T\left(7\tfrac{n}{10}\right) + \tfrac{11}{5}n = \mathcal{O}(n)$$

Capitolo 19

Problemi intrattabili
Teoria dell'NP-completezza

Introduzione

Con l'eccezione della sezione su backtrack, finora abbiamo considerato solo problemi con soluzioni in tempo polinomoniale. Il tempo di esecuzione per queste soluzioni è $\mathcal{O}(n^k)$ per qualche k.

Esistono problemi che per essere risolti hanno bisogno di un tempo almeno esponenziale (che indicheremo con $\mathbb{EXPTIME}$). Valutare una posizione di scacchi, dama, go; Oppure risolvere il problema delle Torri di Hanoi sono esempi di problemi con tempo almeno esponenziale. Esistono inoltre problemi per i quali non esiste alcuna soluzione. Come ad esempio il problema della terminazione (*halting problem*).

In questa lezione discuteremo di una classe di problemi per cui non è chiaro se esista un algoritmo polinomoniale oppure no; questi problemi sono tutti nella stessa barca: potrebbero essere tutti risolti in tempo polinomoniale, oppure nessuno di essi.

I problemi per il millennio (*Millennium problems*) sono sette problemi posti all'attenzione dei matematici dell'Istituto Clay. Per chi risolve uno di questi problemi c'è in palio un milione di dollari. Ad oggi, solo un problema è stato risolto (la Congettura di Poincarè).

Problema astratto Un problema astratto è una relazione binaria $R \subseteq I \times S$ tra un insieme I di istanze del problema e un insieme S di soluzioni.

Ad esempio prendiamo in considerazione il problema del cammino minimo (SHORTEST-PATH). Una sua istanza è una quadrupla (V, E, u, v), e una sua soluzione è una sequenza ordinata di vertici $v_1, \ldots, v_k$.

Nota. Possono esistere più soluzioni associate alla stessa istanza.

Tipologie di problemi

Esistono diverse tipologie di problemi: problemi di **ottimizzazione**, di **ricerca** o di **decisione**.

Nei **problemi di ottimizzazione** data un'istanza, dobbiamo trovare la "migliore" soluzione secondo criteri specifici. Ad esempio nel problema del cammino minimo dato un grafo G e due nodi u, v, dobbiamo trovare il cammino più breve fra essi.

Nei **problemi di ricerca** data un'istanza, dobbiamo trovare una possibile soluzione fra quelle esistenti. Ad esempio nel problema della ricerca di un cammino dato un grafo G e due nodi u, v, dobbiamo trovare un cammino fra essi.

Nei **problemi di decisione** data un'istanza, cerchiamo semplicemente di verificare se soddisfa o meno una data proprietà. La relazione in questo caso è una funzione $R : I \to \{0, 1\}$. Prendiamo di nuovo in considerazione il problema del percorso minimo ma stavolta ci poniamo una domanda. Dati un grafo G, due nodi u, v e un valore k, esiste un cammino tra u e v di lunghezza minore o uguale a k?

Equivalenza fra problemi di ottimizzazione e decisione In quanto la versione decisionale dei problemi è più semplice da trattare matematicamente, ragioneremo in termini di problemi di decisione e non di ottimizzazione.

Difatti se posso risolvere efficientemente la versione di ottimizzazione, allora posso risolvere efficientemente la versione di decisione. É vero anche l'opposto. Ossia, se *non* posso risolvere efficientemente la versione di decisione, allora *non* posso risolvere efficientemente la versione di ottimizzazione.

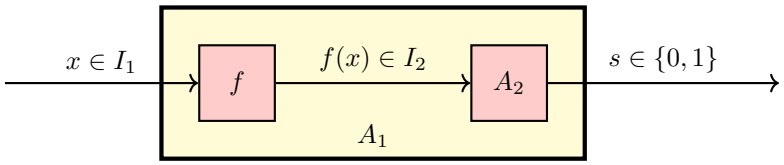
Ad esempio, riconducendoci sempre al problema del cammino più breve, se conosco il cammino più breve, posso rispondere alla domanda decisionale per qualunque valore di k.

19.1 Riduzioni

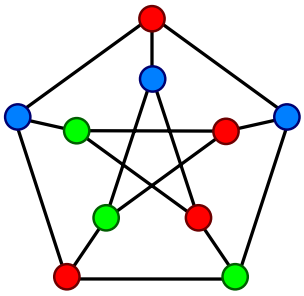
19.1.1 Riduzione polinomiale

Dati due problemi decisionali $R_1 \subseteq I_1 \times \{0,1\}$ e $R_2 \subseteq I_2 \times \{0,1\}$, R_1 è riducibile polinomialmente a R_2 (si scrive matematicamente $R_1 \leqslant_p R_2$) se esiste una funzione $f : I_1 \to I_2$ con le seguenti proprietà:

- f è calcolabile in tempo polinomiale;
- per ogni istanza x del problema R_1 e ogni soluzione $s \in \{0,1\}$, $(x,s) \in R_1 \Leftrightarrow (f(x),s) \in R_2$.



19.1.2 Colorazione di grafi



Dati un grafo non orientato $G = (V,E)$ e un insieme di colori C, una colorazione dei vertici è un assegnamento $f : V \to C$ che "colora" ogni nodo con uno dei valori in C, tale per cui nessuna coppia di nodi adiacenti ha lo stesso colore.
Possiamo formulare il problema sia sotto forma di problema di ottimizzazione, che decisionale. Nella forma decisionale dobbiamo determinare *se esiste*, dato un grafo non orientato $G = (V,E)$ e un valore k, una colorazione G con k colori; mentre nella forma di ottimizzazione dobbiamo restituire la colorazione che necessita del numero minimo di colori.

19.1.3 Sudoku

Il problema generale del Sudoku richiede di inserire dei numeri fra 1 e n^2 in una matrice di $n^2 \times n^2$ elementi suddivisa in $n \times n$ sottomatrici di dimensione $n \times n$, in modo tale che nessun numero compaia più di una volta in ogni riga, colonna e sottomatrice.

Esistono due versioni di questo problema, entrambe espresse in forma decisionale.

Le prima versione è quella classica: data una matrice $n^2 \times n^2$ elementi, determinare se esiste un modo per assegnare i numeri in modo da rispettare le regole del Sudoku.

Mentre nella seconda abbiamo qualche numero già presente: data una matrice $n^2 \times n^2$ elementi con alcuni numeri già presenti nella matrice, determinare se esiste un modo per assegnare i numeri in modo da rispettare le regole del Sudoku.

Riduzione da problema particolare a problema generale

Riduzione in tempo polinomiale $V = \{(x,y) : 1 \leqslant x \leqslant n^2, 1 \leqslant y \leqslant n^2\}$

$[(x,y),(x',y')] \in E \Leftrightarrow$

- $x = x'$; oppure,
- $y = y'$; oppure,
- $(\lceil x/n \rceil = \lceil x'/n \rceil) \wedge ((\lceil y/n \rceil = \lceil y'/n \rceil))$

$C = \{1, \ldots, n\}$.

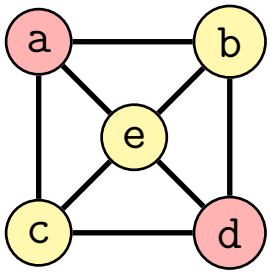
Se abbiamo una soluzione per la colorazione, allora abbiamo una soluzione algoritmica per il Sudoku. Scriveremo quindi che SUDOKU $\leqslant_p$ GRAPH-COLORING.

Troviamo diverse applicazioni. Come ad esempio nell'assegnamento delle radio frequenze in un insieme di torri cellulari. O nell'allocazione degli esami universitari.

19.1.4 Insieme indipendente

Sia dato un grafo non orientato $G = (V, E)$; un insieme $S \subseteq V$ è un insieme indipendente se e solo se nessun arco unisce due nodi in S. Matematicamente scriveremo

$$\forall (x, y) \in E : x \notin S \vee y \notin S$$



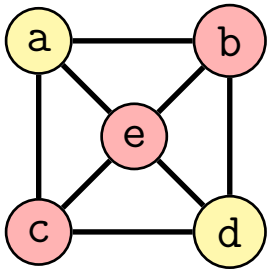
Possiamo formulare il problema sia sotto forma di problema di ottimizzazione, che decisionale. Nella forma decisionale dobbiamo determinare *se esiste*, dato un grafo non orientato $G = (V, E)$ e un valore k, un insieme indipendente di dimensione k; mentre nella forma di ottimizzazione dobbiamo restituire il più grande insieme indipendente presente nel grafo.

Packing problem Questo è un esempio di packing problem, un problema in cui si cerca di selezionare il maggior numero di oggetti, la cui scelta è però soggetta a vincoli di esclusione. I problema dello zaino ne è l'esempio prìncipe.

19.1.5 Copertura di vertici

Dato un grafo non orientato $G = (V, E)$, un insieme $S \subseteq V$ è una copertura di vertici se e solo se ogni arco ha almeno un estremo in S. Matematicamente scriveremo

$$\forall (x, y) \in E : x \in S \vee y \in S$$



Possiamo formulare il problema sia sotto forma di problema di ottimizzazione, che decisionale. Nella forma decisionale dobbiamo determinare *se esiste*, dato un grafo non orientato $G = (V, E)$ e un valore k, una copertura di vertici di dimensione al massimo k; mentre nella forma di ottimizzazione dobbiamo restituire la copertura dei vertici di dimensione minima.

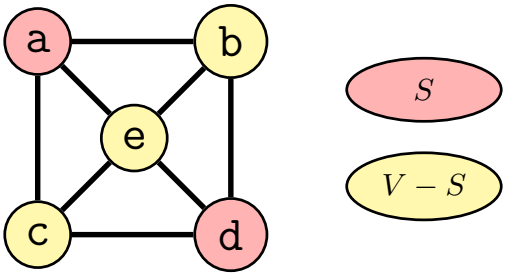
Covering problem Questo è un esempio di covering problem, un problema in cui si cerca di ottenere il più piccolo insieme in grado di coprire un insieme arbitrario di oggetti con il più piccolo sottoinsieme di questi oggetti.

19.1.6 Riduzione per problemi duali

Se $S \subseteq V$ è un insieme indipendente, allora $V - S$ è una copertura di vertici

Se S è un insieme indipendente:

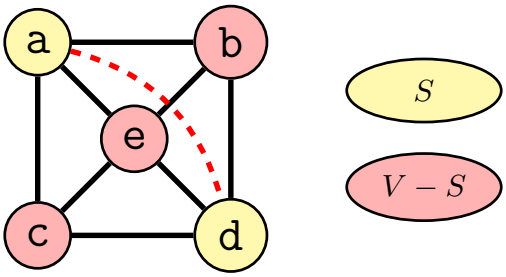
- ogni arco (x, y) non può avere entrambi gli estremi in S;
- quindi almeno uno dei due deve essere in $V - S$.



Se $V - S$ è una copertura di vertici, allora $S \subseteq V$ è un insieme indipendente

Supponiamo per assurdo che S non sia un insieme indipendente:

- allora esiste un arco (x, y) che unisce due nodi in S;
- nessuno dei due estremi sta in $V - S$;
- questo implica che $V - S$ non è una copertura di vertici, il che è assurdo.



19.1.7 Equivalenza dei problemi

Se il problema della copertura di vertici è riducibile al problema dell'insieme indipendente.

$$\text{VERTEX-COVER} \leqslant_p \text{INDIPENDENT-SET}$$

E il problema dell'insieme indipendente è riducibile al problema della copertura di vertici.

$$\text{INDIPENDENT-SET} \leqslant_p \text{VERTEX-COVER}$$

Abbiamo dimostrato che i due problemi sono equivalenti.

19.1.8 Soddisfacibilità di formule booleane

Data un'espressione in forma normale congiuntiva, il problema della soddisfacibilità consiste nel decidere se esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera.

Ad esempio se prendiamo l'espressione $(x \vee \bar{y} \vee z) \wedge (\bar{x} \vee w) \wedge y$ e assegniamo ad ogni variabile il valore di verità **true**, allora possiamo riscrivere l'espressione come $(\textbf{true} \vee \textbf{false} \vee \textbf{false}) \wedge (\textbf{false} \vee \textbf{true}) \wedge \textbf{true}$, semplificando ulteriormente otteniamo $\textbf{true} \wedge \textbf{true} \wedge \textbf{true}$, ed infine **true**. Possiamo dedurne quindi che la formula booleana iniziale è soddisfacibile, in quanto esiste almeno un'interpretazione delle variabili che la rende vera.

Riduciamo il problema generale ad una forma che include solo tre letterali. Chiameremo questo problema 3-SAT. Data una espressione in forma normale congiuntiva in cui le clausole hanno esattamente 3 letterali, il problema della soddisfacibilità consiste nel decidere se esiste una assegnazione di valori di verità alle variabili che rende l'espressione vera.

Vogliamo dimostrare che il problema 3-SATè riducibile polinomialmente a quello dell'insieme indipendente. Ossia

$$\text{3-SAT} \leqslant_p \text{INDIPENDENT-SET}$$

Riduzione tramite "gadget"

Data una formula 3-SAT, costruiamo un grafo nel modo seguente:

- per ogni clausola, aggiungiamo un terzetto di nodi, collegati fra di loro da archi;
- per ogni letterale che compare in modo normale e in modo negato, aggiungere un arco fra di essi (che chiameremo *arco di conflitto*).

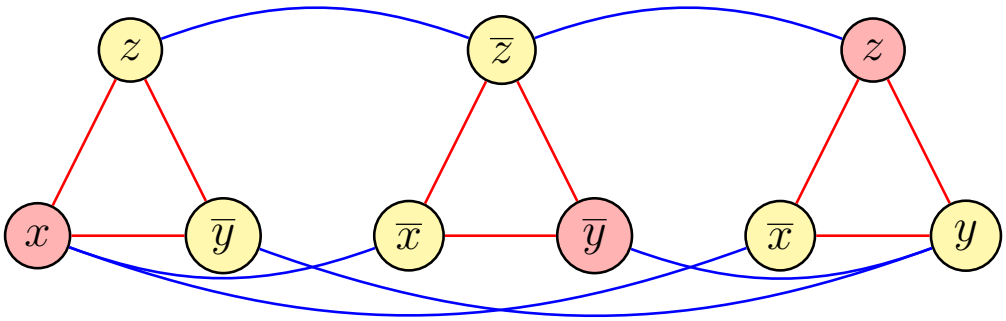


Figura 19.1: La formula 3-SAT è soddisfacibile se e solo è possibile trovare un insieme indipendente di dimensione esattamente k.

Riduzione da problema particolare a problema generale

Ovviamente il problema di soddisfacibilità delle formule booleane con sole tre variabili è riducibile polinomialmente al problema più generale. Scriviamo quindi che 3-SAT $\leqslant_p$ SAT.

Possiamo dimostrare l'inverso? Ossia che SAT $\leqslant_p$ 3-SAT?

É possibile trasformare una formula SAT in una formula 3-SAT usando due semplici trucchi:

- se la clausola è più lunga di tre elementi, si introduce una nuova variabile e si divide la clausola in due:

$$(a \vee b \vee c \vee d) \equiv (a \vee b \vee z) \wedge (\bar{z} \vee c \vee d)$$

- se la clausola è più corta di tre elementi, si fa "*padding*":

$$(a \vee b) \equiv (a \vee a \vee b)$$

19.1.9 Proprietà transitiva della riduzione polinomiale

É facile intuire che la nozione di riducibilità polinomiale gode della proprietà transitiva.

$$\text{SAT} \leqslant_p \text{INDIPENDENT-SET} \leqslant_p \text{VERTEX-COVER}$$

19.2 Classi $\mathbb{P}$, $\mathbb{PSPACE}$

Algoritmo Dati un problema di decisione R e un algoritmo A (scritto in un modello di calcolo Turing-equivalente) che lavora in tempo $f_t(n)$ e spazio $f_s(n)$, diciamo che A risolve R se A restituisce s su un'istanza x se e solo se $(x, s) \in R$.

Classi di complessità Data una qualunque funzione $f(n)$, chiamiamo:

- $\mathbb{TIME}(f(n))$ l'insieme dei problemi decisionali risolvibili da un algoritmo che lavora in tempo $\mathcal{O}(f(n))$;
- $\mathbb{SPACE}(f(n))$ gli insiemi dei problemi decisionali risolvibili da un algoritmo che lavora in spazio $\mathcal{O}(f(n))$

La classe $\mathbb{P}$ è la classe dei problemi decisionali risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathbb{P} = \bigcup_{c=0}^{\infty} \mathbb{TIME}(n^c)$$

La classe $\mathbb{PSPACE}$ è la classe dei problemi decisionali risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso:

$$\mathbb{PSPACE} = \bigcup_{c=0}^{\infty} \mathbb{SPACE}(n^c)$$

Nota. $\mathbb{P} \subseteq \mathbb{PSPACE}$.

19.3 Classe $\mathbb{NP}$

19.3.1 Certificato

Dato un problema decisionale R e un'istanza di input x tale che $(x, \textbf{true}) \in R$, un certificato è un insieme di informazioni che permette di provare che $(x, \textbf{true}) \in R$.

Un'assegnamento di verità alle variabili della formula è un certificato per il problema di soddisfacibilità delle formule booleane. Un'associazione nodo-colore $f : V \to \{1, \ldots, k\}$ è un certificato per il problema di colorazione dei grafi. Un sottoinsieme di V di k elementi è un certificato per il problema degli insiemi indipendenti. Tutti questi "certificati" hanno dimensione polinomiale nella dimensione dell'input.

I certificati possono essere verificati in tempo polinomiale:

- in SAT si calcola il valore di verità della formula a partire dall'assegnamento di verità delle variabili in tempo $\mathcal{O}(n)$;
- in GRAPH-COLORING si verifica che nodi adiacenti non abbiano lo stesso colore in tempo $\mathcal{O}(m+n)$;
- In INDIPENDENT-SET si verifica che nodi in S non abbiano nodi adiacenti in $V-S$ in tempo $\mathcal{O}(m+n)$.

Nota. La classe $\mathbb{NP}$ è l'insieme di tutti i problemi che ammettono un certificato verificabile in tempo polinomiale.

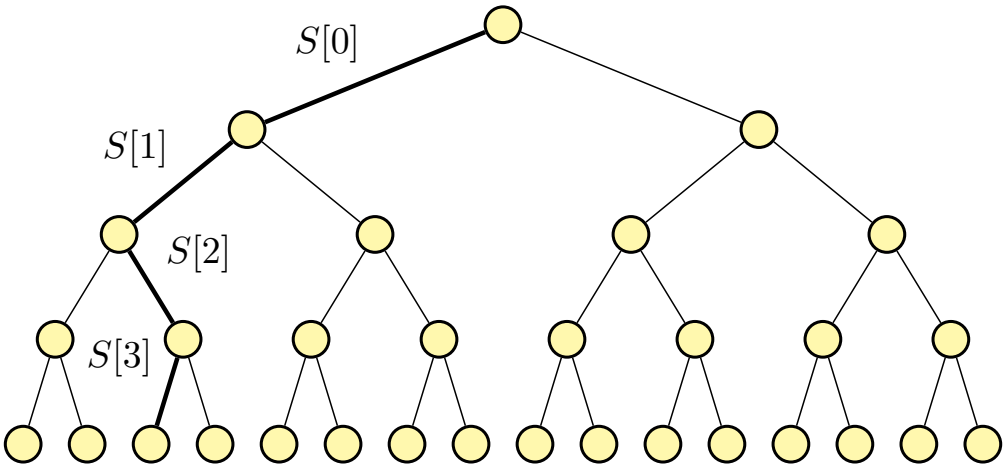
Certificati non polinomiali Esistono dei certificati che non possono essere verificati in tempo polinomiale. Ad esempio il problema *Quantified Boolean Formula (QBF)* è una generalizzazione del problema SAT nel quale ad ogni variabile possono essere applicati quantificatori universali ed esistenziali. Un esempio è la formula $\forall x \exists y \exists z((x \vee z) \wedge y)$. Si ritiene che un certificato del genere non esista.

19.3.2 Definizione basata su non determinismo

$\mathbb{NP}$ è l'insieme di problemi decisionali che possono essere risolti da una Macchina di Turing non deterministica in tempo polinomiale.

In maniera *molto* informale: dato uno stato ed un elemento di input, una macchina non deterministica può andare in un insieme finito di altri stati. Esistono due interpretazioni per la macchina non deterministica. Può essere una macchina che "azzecca" sempre la scelta giusta, oppure può essere una macchina non deterministica che si divide in un insieme finito di copie, una per scelta possibile.

Prendiamo ad esempio il problema SAT con quattro variabili.



19.3.3 Relazioni fra problemi

Lemma 17. *Se $R_1 \leqslant_p R_2$ e $R_2 \in \mathbb{P}$, allora anche R_1 è contenuto in $\mathbb{P}$.*

Dimostrazione. Sia $T_f(n) = \mathcal{O}(n^{k_f})$ il tempo necessario per trasformare un input di R_1, in input di R_2, tramite una funzione f. Sia $T_2(n) = \mathcal{O}(n^{k_2})$ il tempo necessario per risolvere R_2. Qual è la complessità di R_1?

La funzione f può prendere un input di dimensione n e trasformarlo in un input di dimensione $\mathcal{O}(n^{k_f})$ per R_2. Il tempo per risolvere R_1 sarà quindi $T_1(n) = \mathcal{O}(n^{k_f} n^{k_2})$.

Possiamo dedurre che $T_1(n)$ è polinomiale. □

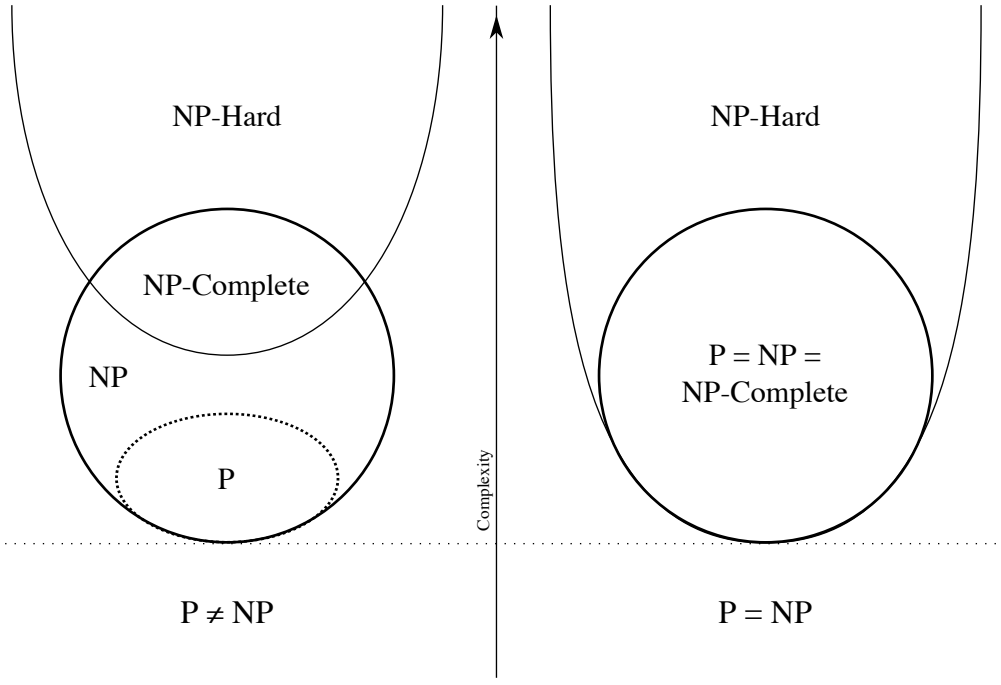
19.4 Problemi $\mathbb{NP}$-completi

Introduciamo alcune definizioni.

Problema $\mathbb{NP}$-arduo ($\mathbb{NP}$-hard) Un problema decisionale R si dice $\mathbb{NP}$-arduo se ogni problema $Q \in \mathbb{NP}$ è riducibile polinomialmente a R ($Q \leqslant_p R$).

Problema $\mathbb{NP}$-completo ($\mathbb{NP}$-complete) Un problema decisionale R si dice $\mathbb{NP}$-completo se appartiene alla classe $\mathbb{NP}$ ed è $\mathbb{NP}$-arduo.

Nota. Se un qualunque problema decisionale $\mathbb{NP}$-completo appartenesse a $\mathbb{P}$, allora risulterebbe $\mathbb{P} = \mathbb{NP}$.



Dimostrare che un problema è contenuto in $\mathbb{NP}$ è semplice. Dimostrare che un problema è $\mathbb{NP}$-completo richiede una dimostrazione difficile, apparentemente impossibile: tutti i problemi in $\mathbb{NP}$ sono riducibili polinomialmente a tale problema, anche quelli che non conosciamo!

Nel 1973 Leonid Levin ha dimostrato in maniera indipendente il seguente teorema.

Teorema (Teorema di Cook-Levin)**.** *SAT è $\mathbb{NP}$-completo.*

La dimostrazione del Teorema è complessa, è basata sugli stati della macchina di Turing.

Problemi introdotti oggi

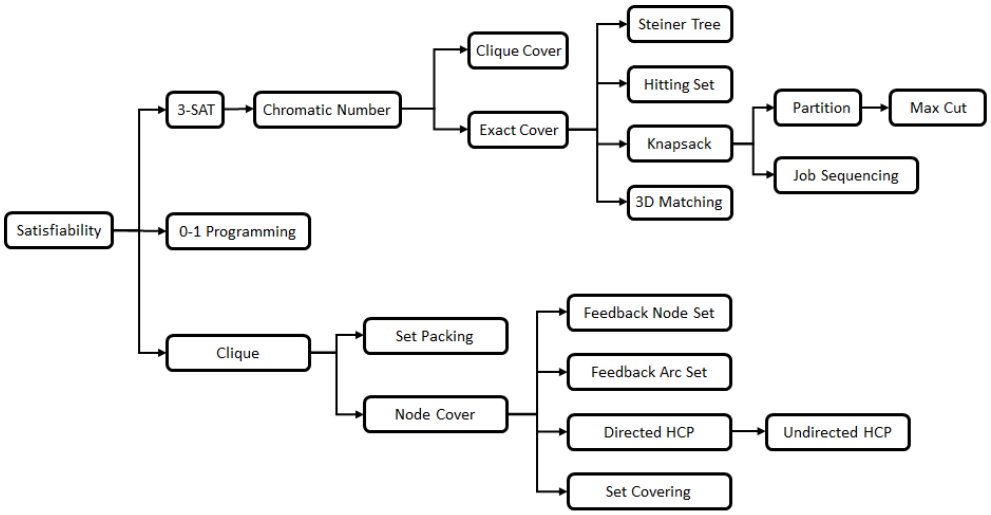
Partendo dalle riduzioni viste oggi e utilizzando il Teorema di Cook-Levin, otteniamo:

$$\text{SAT} \leqslant_p \text{3-SAT} \leqslant_p \text{INDIPENDENT-SET} \leqslant_p \text{VERTEX-COVER} \leqslant_p \text{SAT}$$

In altre parole, 3-SAT, INDIPENDENT-SET, VERTEX-COVER sono $\mathbb{NP}$-completi.

In "*Reducibility Among Combinatorial Problems*" Richard Karp ha stilato una lista di 21 problemi $\mathbb{NP}$-completi.

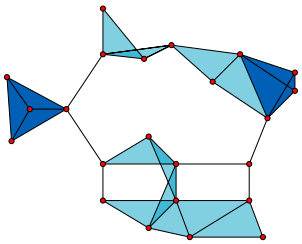
I 21 problemi $\mathbb{NP}$-completi di Karp

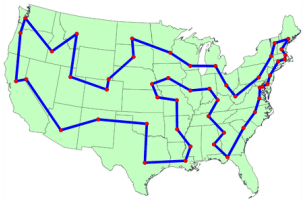


19.4.1 Problemi $\mathbb{NP}$-Completi "Classici"

Cricca (CLIQUE)

Dati un grafo non orientato ed un intero k, esiste un sottoinsieme di almeno k nodi tutti mutualmente adiacenti?
Questo problema trova applicazioni in bioinformatica, ingegneria elettronica e chimica.





Commesso viaggiatore (*Traveling salesperson*, TSP)

Date n città, le distanze tra esse, ed un intero k, è possibile partire da una città attraversare ogni città esattamente una volta tornando alla città di partenza, percorrendo una distanza non superiore a k?

Programmazione lineare `0/1`

Data una matrice A di elementi interi e di dimensione $m \times n$, ed un vettore b di m elementi interi, esiste un vettore x di n elementi `0/1` tale che $Ax \leqslant b$?

Ad esempio

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 &\geqslant 2 \\ x_1 - x_2 - x_3 + x_4 &\geqslant 0 \\ x_1 \quad\quad + x_3 + x_4 &\geqslant 1 \end{aligned}$$

Il sistema è verificato per $x_1 = x_2 = 1$ ed $x_3 = x_4 = 0$.

Copertura esatta di insiemi

Dato un insieme X e una collezione $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$ di sottoinsiemi di X, esiste una sottocollezione $\mathcal{Z} \subseteq \mathcal{Y}$ che partizioni X?

Supponiamo che ad esempio l'insieme X sia $\{1, 2, 3, 4, 5, 6, 7\}$ e la collezione $\mathcal{Y} = \{A, B, C, D, E, F\}$ sia così composta:

$$A = \{1, 4, 7\} \qquad B = \{1, 4\} \qquad C = \{4, 5, 7\} \qquad D = \{3, 5, 6\} \qquad E = \{2, 3, 6, 7\} \qquad F = \{2, 7\}$$

La sottocollezione che partiziona X è $\mathcal{Z} = \{B, D, F\}$.

Partizione (PARTITION)

Dato un vettore A contenente n interi positivi, esiste un sottoinsieme $S \subseteq \{1 \ldots n\}$ tale che $\sum_{i \in S} A[i] = \sum_{i \in S} A[i]$?

Somma di sottoinsieme (SUBSET-SUM)

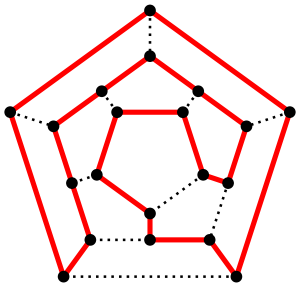
Dati un vettore A contenente n interi positivi ed un intero positivo k, esiste un sottoinsieme $S \subseteq \{1 \ldots n\}$ tale che $\sum_{i \in S} a[i] = k$?

Zaino

Dati un intero positivo C (la capacità dello zaino) e un insieme di n oggetti, tali che l'oggetto i è caratterizzato da un "profitto" $p[i] \in Z^+$ e da un peso $w(i) \in Z^+$. Esiste un sottoinsieme $S \subseteq \{1, \ldots, n\}$ tale che il peso totale $w(S) = \sum_{i \in S} w[i] \leqslant C$ e il profitto totale $p(S) = \sum_{i \in S} p[i]$ è maggiore o uguale a k?

Circuito hamiltoniano

Dato un grafo non orientato G, esiste un circuito che attraversi ogni nodo una e una sola volta?



La complessità si nasconde dove non te l'aspetti

Circuito hamiltoniano Dato un grafo non orientato G, esiste un circuito che attraversi ogni *nodo* una e una sola volta? É un problema ℕℙ-completo.

Circuito euleriano Dato un grafo non orientato G, esiste un circuito che attraversi ogni *arco* una e una sola volta? É un problema in ℙ.

Cammini massimi Dato un grafo $G = (V, E)$ e una funzione di peso w sugli archi, trovare il cammino con il peso *massimo*. É un problema ℕℙ-completo.

Cammini minimi Dato un grafo $G = (V, E)$ e una funzione di peso w sugli archi, trovare il cammino con peso *minimo*. É un problema in ℙ.

19.4.2 Problemi aperti

Isomorfismo fra grafi Il problema dell'isomorfismo fra grafi richiede di determinare se due grafi finiti sono isomorfi. Non sappiamo ancora se il problema sia $\mathbb{NP}$-completo o sia in $\mathbb{P}$. Il dibattito è ancora in corso.

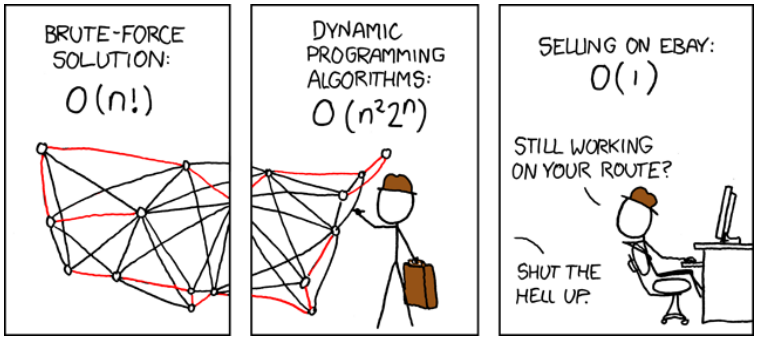
Primalità Dato un numero n, determinare se n è primo. Questo problema è incluso in $\mathbb{P}$.

Fattorizzazione Dato un numero n, individuare i fattori primi che lo compongono. Questo problema è sicuramente contenuto in $\mathbb{NP}$, si presume che non sia contenuto nè in $\mathbb{P}$, nè che sia $\mathbb{NP}$-completo.

Spunti di lettura

Potete approfondire l'argomento nei seguenti testi:

- Jeff Erickson, Algorithm. Cap. 12, NP-Hardness http://jeffe.cs.illinois.edu/teaching/algorithms/;
- Sanjeev Arora, Boaz Barak. Computational complexity: a modern approach. Cambridge University Press, 2009. http://theory.cs.princeton.edu/complexity/.



Capitolo 20

Soluzioni per problemi intrattabili

20.1 Algoritmi pseudo-polinomiali

Non si può avere tutto dalla vita; bisogna rinunciare a qualcosa:

- **generalità**: algoritmi pseudo-polinomiali che funzionano per solo alcuni casi particolari dell'input;
- **ottimalità**: algoritmi di approssimazione, che garantiscono di ottenere soluzioni "vicine" alla soluzione ottimale;
- **formalità**: algoritmi euristici, di solito basati su tecniche *greedy* o di ricerca locale, che forniscano sperimentalmente risultati buoni;
- **efficienza**: algoritmi esponenziali branch-&-bound, che limitano lo spazio di ricerca con un'accurata potatura.

Per il rilassamento di ognuno di questi vincoli possiamo definire un tecnica.

20.1.1 Somma di sottoinsiemi SUBSET-SUM

Definizione del problema Dati un insieme $A = \{a_1, a_2, \ldots, a_n\}$ di interi positivi e un intero positivo k, esiste un sottoinsieme S di indici in $\{1, \ldots, n\}$ tale che $\sum_{i \in S} A_i = k$?

Utilizzando *backtracking*, abbiamo risolto la versione di ricerca di questo problema. Quella appena enunciata è la versione decisionale. Per semplificare il confronto, ci concentriamo sulla seconda.

Somma di sottoinsiemi risolto tramite programmazione dinamica

Definiamo una tabella booleana $DP[0 \ldots n][0 \ldots k]$. $DP[i][r]$ è uguale a **true** se esiste un sottoinsieme dei primi i valori memorizzati in A la cui somma è pari a r, **false** altrimenti. Il problema generale è definito da $DP[n][k]$.

Definizione dell'equazione di ricorrenza Analizziamo caso per caso. Prima i casi base:

① se devo ottenere un valore uguale a 0 ($r = 0$), sono sicuro che non prendendo nessun oggetto posso ottenere quel valore, quindi restituiremo **true**;

② nel caso in cui voglia ottenere un valore ($r > 0$), ma non ho più alcun oggetto da sommare ($i = 0$), allora non mi sarà possibile soddisfare la richiesta, restituirò **false**.

Consideriamo ora i casi in cui vogliamo ottenere un valore ($r > 0$) e abbiamo ancora oggetti a nostra disposizione ($i > 0$).

③ il valore dell'oggetto considerato è troppo grande rispetto al valore r che voglio ottenere ($A[i] > r$), quindi escludo quell'oggetto ($i - 1$) e lascio inalterato il valore r;

④ posso prendere anche l'ultimo oggetto in quanto il suo valore non eccede r ($A[i] \leqslant r$), quindi considero la possibilità di non prenderlo e di lasciare inalterato il valore di r ($DP[i-1][r]$), oppure lo prendo e sottraggo il suo valore a quello che voglio ottenere ($DP[i-1][r - A[i]]$).

Siamo quindi riusciti a definire l'equazione di ricorrenza come segue

$$DP[i][r] = \begin{cases} \textbf{true} & r = 0 \\ \textbf{false} & r > 0 \wedge i = 0 \\ DP[i-1][r] & r > 0 \wedge i > 0 \wedge A[i] > r \\ DP[i-1][r] \textbf{ or } DP[i-1][r - A[i]] & r > 0 \wedge i > 0 \wedge A[i] \leqslant r \end{cases}$$

L'algoritmo sfrutta l'equazione di ricorrenza.

Algoritmo 20.1.1: Somma di sottoinsiemi risolto tramite programmazione dinamica

```
boolean subSetSum(int[] A, int n, int k)
    boolean[][] DP ← new boolean[0...n][0...k] = {false}
    // CASI BASE
    // se il valore da ottenere è 0, non ho bisogno di selezionare nessun indice
    for i ← 0 until n do // primacolonna
        DP[i][0] ← true // r = 0
    // se non ho nessun intero positivo da selezionare non mi è possibile arrivare al valore richiesto
    for r ← 1 until k do // primariga
        DP[0][r] ← false // r > 0 ∧ i = 0
    // CASO RICORSIVO
    from i ← 1 until n do
        from r ← 1 until A[i] − 1 do // commento
            DP[i][r] ← DP[i − 1][r] // A[i] > r
        from r ← A[i] until k do // commento
            DP[i][r] ← DP[i − 1][r] or DP[i − 1][r − A[i]] // A[i] ⩽ r
    // restituisco il valore in posizione k-esima
    return DP[n][k]
```

Complessità Essendo un problema decisionale, è possibile semplificare e utilizzare spazio $\Theta(k)$, invece che $\Theta(nk)$, in quanto avrò n righe e k colonne.

Esempio di esecuzione Ad esempio prendendo l'insieme $A = [5, 9, 10]$ di interi positivi e l'intero positivo $k = 24$ proviamo a riempire la tabella di programmazione dinamica. Dove le righe rappresentano il numero di oggetti presi e le colonne il valore di k desiderato.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	0	0	1

Analisi della complessità La complessità dell'algoritmo è $\Theta(nk)$, ma la complessità dei dati in ingresso è $\mathcal{O}(n \log k)$, in quanto i valori più grandi del nostro obiettivo possono essere esclusi. Se k è $\mathcal{O}(n^c)$ con c costante, allora subSetSum ha complessità polinomiale $\mathcal{O}(n^{c+1})$. Ma se k è $\mathcal{O}(2^n)$, allora subSetSum ha complessità superpolinomiale $\mathcal{O}(n \cdot 2^n)$.

Osservazione. La complessità di subSetSum dipende quindi dai valori contenuti nell'insieme e non soltanto dalla cardinalità dei dati in ingresso (n).

Somma di sottoinsiemi risolto tramite backtracking

Algoritmo 20.1.2: Somma di sottoinsiemi risolto tramite backtracking

```
boolean ssRec(Int[] A, int i, int r)
    if r == 0 then
        return true
    else if i == 0 then
        return false
    else if A[i] > r then
        return ssRec(A, i − 1, r)
    else
        return ssRec(A, i − 1, r) or ssRec(A, i − 1, r − A[i])
    return res
```

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Appendice A

Algoritmi di ordinamento

Introduzione

Durante il corso abbiamo visto alcuni algoritmi di ordinamento, le loro complessità sono riassunte nella tabella A.1.

Tabella A.1: Complessità degli algoritmi di ordinamento

Algoritmo	Caso ottimo	Caso pessimo	Tecnica utilizzata
SelectionSort	$\Omega(n)$	$\mathcal{O}(n)$	-
InsertionSort	$\Omega(n)$	$\mathcal{O}(n^2)$	-
ShellSort	$\Omega(n)$	$\mathcal{O}(n^{3/2})$	ricerca locale
MergeSort	$\Theta(n \log n)$		dividi-et-impera
HeapSort	$\Theta(n \log n)$		-
QuickSort	$\Omega(n \log n)$	$\mathcal{O}(n^2)$	-

shellSort ha una complessità "migliore" di insertionSort $n^{3/2} = n\sqrt{n}$). Sappiamo inoltre che quickSort nel caso medio ha una complessità di $n \log n$.

Possiamo arrivare alla conclusione che tutti questi algoritmi sono *basati su confronti*: le decisioni sull'ordinamento vengono prese in base al confronto (<, =, >) fra due valori. Esistono altri modi per ordinare gli oggetti, ma gli algoritmi generali devono essere necessariamente basati sui confronti.

Gli algoritmi migliori che abbiamo visto finora hanno una complessità di $\mathcal{O}(n \log n)$, insertionSort e shellSort sono più veloci ($\Omega(n)$ solo in casi speciali, è quindi lecito chiederci se sia possibile fare meglio di così.

Dimostrazione del limite inferiore del problema dell'ordinamento

È possibile dimostrare che qualunque algoritmo di ordinamento *basato sui confronti* ha una complessità di $\Omega(n \log n)$.

Partiamo facendo alcune assunzioni. Consideriamo un qualunque algoritmo basato su confronti. Assumiamo che tutti i valori siano distinti (non abbiamo perdita di generalità). L'algoritmo può essere quindi rappresentato tramite un *albero di decisione*, ossia un albero binario che rappresenta i confronti con gli elementi.

Gli alberi di decisione hanno due proprietà:

1. il *cammino radice-foglia* rappresenta la sequenza di confronti eseguiti dall'algoritmo corrispondente;
2. l'*altezza dell'albero* rappresenta il numero di confronti eseguiti dall'algoritmo corrispondente nel caso pessimo.

Si considerino tutti gli alberi di decisione ottenibili da algoritmi di ordinamento basati su confronti.

Lemma 18. *Un albero di decisione per l'ordinamento di n elementi contiene* ***almeno*** $n!$ *foglie.*

Questo accade perché le foglie rappresentano il modo in cui dobbiamo scambiare i valori e siccome $n!$ è una qualsiasi delle permutazioni in cui riceviamo il nostro input, l'albero di decisione corrispondente deve avere una foglia per ogni suo elemento. Potrebbe averne anche più di una foglia, in quanto l'algoritmo potrebbe non essere efficiente ed effettuare più ordinamenti di quanto ne siano necessari, ma deve averne *almeno* $n!$.

Ora possiamo restringerci al caso dell'albero binario in quanto abbiamo valori distinti e lo possiamo fare senza perdere generalità.

Lemma 19. *Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli (< e >) e sia k il numero delle sue foglie. Allora l'altezza dell'albero è* ***almeno*** $\log k$ *ovvero* $\Omega(n \log n)$.

L'altezza sarà $\log k$ solo se l'albero è perfettamente bilanciato.

Teorema 20. *Il numero di confronti necessari per ordinare n elementi nel caso peggiore è* $\Omega(n \log n)$.

Abbiamo $n!$ foglie, quindi l'altezza dell'albero è $\sim \log n!$ e abbiamo che $\sim n \log n$.

A.1 Algortimi non basati sui confronti

A.1.1 Spaghetti Sort

L'algoritmo spaghettiSort è caratterizzato da 5 fasi:

1. prendi n spaghetti;
2. taglia lo spaghetto i-esimo in modo proporzionale all'i-esimo valore da ordinare;
3. con la mano, afferra gli n spaghetti e appoggiali verticalmente sul tavolo;
4. prendi il più lungo, misuralo e metti il valore corrispondente in fondo al vettore da ordinare;
5. ripeti il passo 4 fino a quando non hai terminato gli spaghetti.

Nel modo in cui è stato descritto questo algoritmo è infinitamente parallelo: qualsiasi sia il numero di spaghetti potremmo trovare lo spaghetto più lungo in $\mathcal{O}(1)$. Quindi questo algoritmo ha una complessità di $\mathcal{O}(n)$.

A.1.2 Counting Sort

Assumiamo che i numeri da ordinare siano compresi in un intervallo $[1 \dots k]$ (questo algoritmo non funziona con le stringhe).

Come funziona Costruisce un vettore di appoggio $B[1 \dots k]$ che conta il numero di volte che un valore compreso in $[1 \dots k]$ compare in A. Ricolloca i valori così ottenuti nel vettore da ordinare A.

Nota. L'intervallo non deve necessariamente iniziare in 1 e finire in k; qualunque intervallo di cui conosciamo gli estremi può essere utilizzato nel Counting Sort.

Algoritmo A.1.1: Algoritmo di ordinamento Counting Sort

```
countingSort(ITEM[] A, int n, int k)
    int[] B ← new int[1...k] // creo il vettore d'appoggio
    for i ← 1 until k do // O(k)
        B[i] = 0 // azzero il vettore d'appoggio
    for j ← 1 until n do // O(n)
        B[A[j]] = B[A[j]] + 1 // conto quante volte incontro quel valore
    // vado a ricollocare i vari valori man mano che li incontro
    j = 1 // partendo dalla prima posizione
    from i = 1 until k do // scorro tutto il vettore
        while B[i] > 0 do // per tutte le occorrenze di un determinato valore
            A[j] ← i // lo inserisco nel vettore
            j++ // scorro il cursore
            B[i] ← B[i] − 1 // diminuisco le occorrenze
```

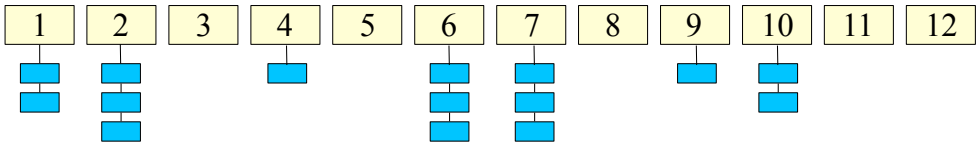
Complessità Inizializzare il vettore d'appoggio costa $\mathcal{O}(k)$. Contare le occorrenze di un valore costa $\mathcal{O}(n)$, in quanto fare riferimento ad una casella di memoria dato un valore costa $\mathcal{O}(1)$ e lo faccio n volte. L'algoritmo ha una complessità totale di $\mathcal{O}(n+k)$. Se k è $\mathcal{O}(n)$, allora la sua complessità è $\mathcal{O}(n)$.

Questo algoritmo non è basato sui confronti. Abbiamo quindi cambiato le condizioni di base e la dimostrazione che abbiamo visto per il limite inferiore non vale. Ad esempio se k è $\mathcal{O}(n^3)$ questo algoritmo è peggiore di tutti quelli visti finora.

A.1.3 Pigeonhole Sort

Se dobbiamo ordinare chiavi numeriche basse e valori associati, allora possiamo usare pigeonholeSort. Sfruttiamo lo stesso meccanismo di countingSort, ma anziché contare le occorrenze le inseriamo all'interno di una lista concatenata. L'inserimento di un elemento in una lista ha costo $\mathcal{O}(1)$, quindi l'algoritmo ha $\mathcal{O}(n)$ a patto che sia possibile limitare l'input a k elementi (possibilmente piccolo). Il costo complessivo risulta quindi $\mathcal{O}(n+k)$ e vale considerazione precedente.so

Nota. pigeonholeSort è un'estensione del countingSort che permette di ordinare in tempo lineare coppie (chiave, valore), invece che singoli interi. Le chiavi devono essere comprese fra 1 e k (con k possibilmente piccolo).



A.1.4 Bucket Sort

Se i valori in input sono:

1. valori reali uniformemente distribuiti nell'intervallo $[0, 1)$;
2. oppure è possibile normalizzarli nell'intervallo $[0, 1)$ in tempo lineare, in quanto sono uniformemente distribuiti, allora

è possibile usare bucketSort, ossia una versione di pigeonholeSort dove utilizziamo dei concetti probabilistici (l'ipotesi di uniformità) per sapere quanti elementi ci aspettiamo che siano contenuti in ogni lista.

Come funziona Divide l'intervallo in n sottovettori di dimensione $1/n$, detti *bucket*, e poi distribuisce gli n numeri nei *bucket*. I valori così inseriti possono essere riordinati tramite insertionSort.

La complessità attesa è di $\mathcal{O}(n)$, molto conveniente, ma l'ipotesi sui valori dell'input è molto stringente (ossia che siano uniformente distribuiti).

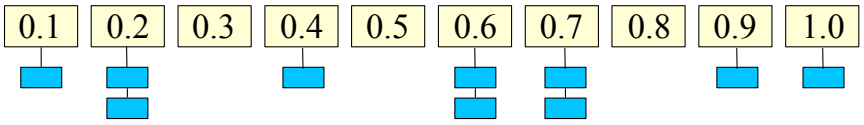


Figura A.1: Il primo *bucket* conterrà gli elementi con valore contenuto nell'intervallo $[0, 0.1)$, il secondo con valori contenuti nell'intervallo $[0.1, 0.2)$ e così via. Per l'ipotesi di uniformità, il numero atteso di valori nei bucket è 1.

Proprietà degli algoritmi di ordinamento

Definizione (stabilità)**.** Un algoritmo di ordinamento è detto **stabile** se preserva l'ordine iniziale tra due elementi con la stessa chiave.

Algoritmi stabili insertionSort, mergeSort, pigeonholeSort sono algoritmi stabili, mentre heapsort (non c'è modo di renderlo stabile) e quickSort (esistono delle versioni complicate che lo rendono tale) non lo sono. È possibile rendere stabile mergeSort avendo l'accortezza di inserire ordinatamente i valori nel vettore al momento dell'unione dei sottovettori.

Nota. Qualunque algoritmo di ordinamento può essere reso stabile. Basta associare al valore da ordinare l'indice in cui si trova all'inizio dell'ordinamento. Ordinando quindi prima per valore e poi per posizione.

A.2 Rissunto algoritmi di ordinamento

insertionSort ha una compleesità di $\Omega(n)$ nel caso ottimo e di $\mathcal{O}(n^2)$ nel caso pessimo. È stabile, sul posto, iterativo. Adatto per piccoli valori (non utilizzare quickSort) e sequenze quasi ordinate.

mergeSort ha una complessità di $\Theta(n \log n)$, è stabile ma richiede $\mathcal{O}(n)$ spazio aggiuntivo, è ricorsivo (richiede $\mathcal{O}(\log n)$ spazio nello *stack*). Buona performance in *cache*, buona parallelizzazione.

heapsort ha una complessità di $\Theta(n \log n)$, non stabile, sul posto, iterativo. Cattiva performance in cache, cattiva parallelizzazione viene quindi preferito in sistemi embedded dove parallelizzazione e memoria aggiuntiva non vengono considerati.

quickSort $\mathcal{O}(n \log n)$ in media, $\mathcal{O}(n^2)$ nel caso peggiore, non stabile, ricorsivo (richiede $\mathcal{O}(\log n)$ spazio nello *stack*). Buona performance in cache, buona parallelizzazione, buoni fattori moltiplicativi. È una delle scelte migliori grazie al meccanismo di randomizzazione permette nel caso medio di avere una complessità di $n \log n$.

countingSort $\Theta(n + k)$, richiede $\mathcal{O}(k)$ memoria aggiuntiva, iterativo. Molto veloce quando $k = \mathcal{O}(n)$.

pigeonholeSort $\Theta(n + k)$, stabile, richiede $\mathcal{O}(n + k)$ memoria aggiuntiva, iterativo. Molto veloce quando $k = \mathcal{O}(n)$.

countingSort e pigeonholeSort vengono utilizzati quando sappiamo qualcosa sui dati in input.

bucketSort $\mathcal{O}(n)$ nel caso i valori siano distribuiti uniformemente, stabile, richiede $\mathcal{O}(n)$ spazio aggiuntivo.

shellSort $\mathcal{O}(n\sqrt{n})$, stabile, adatto per piccoli valori, sequenze quasi ordinate. È una versione migliorata di insertionSort.