Politecnico di Torino
Laurea Triennale in Ingegneria Informatica

appunti di
Algoritmi e programmazione

Autori principali: Luca Ghio
Docenti: Paolo Enrico Camurati, Gianpiero Cabodi
Anno accademico: 2011/2012
Versione: 1.0.2.1
Data: 10 luglio 2021

Ringraziamenti

Oltre agli autori precedentemente citati, quest'opera può includere contributi da opere correlate su WikiAppunti e su Wikibooks, perciò grazie anche a tutti gli utenti che hanno apportato contributi agli appunti *Algoritmi e programmazione* e al libro *Algoritmi*.

Informazioni su quest'opera

Quest'opera è pubblicata gratuitamente. Puoi scaricare l'ultima versione del documento PDF, insieme al codice sorgente LaTeX, da qui: http://lucaghio.epizy.com/redirs/5

Quest'opera non è stata controllata in alcun modo dai professori e quindi potrebbe contenere degli errori. Se ne trovi uno, sei invitato a correggerlo direttamente tu stesso realizzando un commit nel repository Git pubblico o modificando gli appunti *Algoritmi e programmazione* su WikiAppunti, oppure alternativamente puoi contattare l'autore principale inviando un messaggio di posta elettronica a luca.ghio@studenti.polito.it.

Licenza

Quest'opera è concessa sotto una licenza Creative Commons Attribuzione - Condividi allo stesso modo 4.0 Internazionale (anche le immagini, a meno che non specificato altrimenti, sono concesse sotto questa licenza).

Tu sei libero di:

- condividere: riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare questo materiale con qualsiasi mezzo e formato;
- modificare: remixare, trasformare il materiale e basarti su di esso per le tue opere;

per qualsiasi fine, anche commerciale, alle seguenti condizioni:

- **Attribuzione:** devi attribuire adeguatamente la paternità sul materiale, fornire un link alla licenza e indicare se sono state effettuate modifiche. Puoi realizzare questi termini in qualsiasi maniera ragionevolmente possibile, ma non in modo tale da suggerire che il licenziante avalli te o il modo in cui usi il materiale;
- **Condividi allo stesso modo:** se remixi, trasformi il materiale o ti basi su di esso, devi distribuire i tuoi contributi con la stessa licenza del materiale originario.

Indice

Capitolo 1

Gli algoritmi

L'**algoritmo** è una sequenza di istruzioni elementari per risolvere un problema:

- non ambigue: dev'essere chiara la semantica;
- numero finito di passi: ciò che non finisce non si può chiamare algoritmo.

1.1 Classificazione dei problemi

I problemi si dividono in due categorie:

- **problemi decisionali:** "Sì o no?";
- **problemi di ottimizzazione:** trovare la soluzione che minimizza il costo da pagare.

1.1.1 Problemi decisionali

I problemi decisionali si dividono in decidibili e indecidibili.

I **problemi indecidibili** sono quelli per cui non esiste alcun algoritmo che li risolve (ad es. problema di Turing, congettura di Goldbach).

Problemi decidibili

Un problema è **trattabile** se è noto un algoritmo polinomiale (n^c) che lo risolve in tempi ragionevoli.

Un problema è **non trattabile** se non potrà mai essere risolvibile tramite algoritmi polinomiali, ma potrebbe esserlo tramite un algoritmo complesso (ad es. esponenziale) non ancora noto.

Di un problema non polinomiale (NP) sono noti degli algoritmi esponenziali che lo risolvono, ma non si sa se esiste un algoritmo polinomiale che lo risolve. Se verrà trovato un algoritmo polinomiale, il problema NP diventerà di classe P:

$$P \subseteq NP$$

P è sottoinsieme improprio di NP, cioè non si può dire: $P \subset NP$

È possibile inviduare alcuni problemi NP che attraverso semplici trasformazioni si possono far diventare altri: questi fanno parte della **classe NP-completa**.

1.2 Ricerca di un elemento in un vettore

Ricerca sequenziale/lineare

- caso migliore (l'elemento viene trovato subito): 1 accesso;
- caso medio: $\frac{n}{2}$ accessi;
- caso peggiore (l'elemento non viene trovato): n accessi.

Ricerca binaria/dicotomica

- insieme ordinato: cerco di metà in metà
- numero massimo di accessi: $\log_2 n$

1.3 Introduzione agli algoritmi di ordinamento

algoritmo di ordinamento: si fa con le permutazioni.

Esempio CPU scheduling: si hanno n task ciascuno con una specifica durata, e bisogna eseguirli con il minimo tempo medio di attesa $\Rightarrow$ li si mette in ordine crescente di durata.

Approccio di un tipico algoritmo di ordinamento Il vettore dei dati da ordinare viene suddiviso in 2 sottovettori sinistro e destro. Il sottovettore sinistro è ordinato, il sottovettore destro non è ordinato. A ogni passo si sposta un elemento nel sottovettore sinistro mantenendo l'invarianza della proprietà di ordinamento, cioè tenendolo ordinato.

Un vettore contenente un solo dato è per definizione ordinato.

1.3.1 Insertion sort

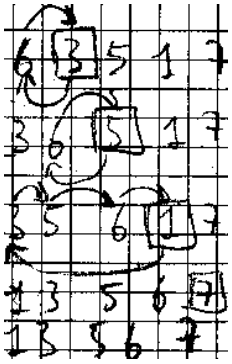


Figura 1.1: Insertion sort.

1. si scandisce il sottovettore ordinato (dall'indice $i+1$) da sinistra a destra, fino a quando non si trova una chiave minore dell'elemento i-esimo;
2. shift a destra degli elementi ordinati;
3. si inserisce l'elemento nella posizione ordinata.

Capitolo 2

L'analisi di complessità degli algoritmi

L'**analisi di complessità** è un metodo formale per prevedere le risorse richieste dall'algoritmo (cioè il tempo di esecuzione e la quantità di memoria utilizzata).

- Il tempo però non è un tempo fisico o misurato sperimentalmente su un processore, ma dipende dal numero di passi ⇒ l'analisi di complessità è **indipendente dalla macchina**. Si assume che la macchina lavori in modo sequenziale e non parallelo (architettura di Von Neumann). Un algoritmo meno complesso su una macchina più lenta può essere più veloce.
- L'analisi è **indipendente da quali dati in ingresso** ci sono ma è **dipendente** solo **da quanti dati in ingresso** ci sono, cioè più in generale dalla **dimensione** n del problema.

Output

- $S(n)$: memoria richiesta
- $T(n)$: tempo di esecuzione

Classificazione degli algoritmi per complessità (dal meno complesso al più complesso)

- costante: 1
- logaritmico: $\log n$
- lineare: n
- linearitmico: $n \log n$
- quadratico: n^2
- cubico: n^3
- esponenziale: 2^n

2.1 Analisi di complessità di caso peggiore

L'analisi di complessità è detta:

- **asintotica** se il numero di dati tende a infinito;
- **di caso peggiore** se il tempo stimato non può che essere maggiore o uguale al tempo effettivo su uno specifico dato.

Si stima il caso peggiore perché è quello più frequente generalmente, e perché il caso medio spesso è difficile da calcolare o coincide con il peggiore.

Ricerca sequenziale

$$T(n) \propto n$$

Ricerca dicotomica Bisogna considerare il caso peggiore: la chiave non si trova. Alla i-esima iterazione, la lunghezza del sottovettore è uguale a $\frac{1}{2^i}$. La terminazione avviene quando il vettore è lungo $1 = \frac{1}{2^i};\ i = \log_2 n$.

Insertion sort Nel caso peggiore, ad ogni iterazione del ciclo esterno si fanno $i-1$ scansioni del vettore, e all'ultima se ne fanno $n-1$:

$$T(n) = 1 + 2 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2} \rightarrow T(n) \propto n^2$$

2.2 Notazioni

2.2.1 O grande (O)

$$T(n) = O(g(n)) \Leftrightarrow \exists c > 0,\ n_0 > 0 : \forall n \geq n_0 \rightarrow 0 \leq T(n) \leq c \cdot g(n)$$

$g(n)$ è un **limite superiore lasco**:

- superiore: è una funzione maggiorante, a partire da un certo n_0;
- lasco: non scresce come $g(n)$, ma al più come $g(n)$.

Se $T(n)$ è un polinomio in n di grado $m \Rightarrow T(n) = O(n^m)$.

2.2.2 Omega grande (Ω)

$$T(n) = \Omega(g(n)) \Leftrightarrow \exists c > 0,\ n_0 > 0 : \forall n \geq n_0 \rightarrow 0 \leq c \cdot g(n) \leq T(n)$$

$g(n)$ è il **limite inferiore lasco** della complessità asintotica di caso peggiore ($\neq$ caso migliore). Se si dimostra che, per una certa operazione, la Ω è maggiore della complessità lineare, non si potrà mai trovare un algoritmo lineare che svolga quell'operazione.
$T(n)$ polinomio $\Rightarrow T(n) = \Omega(n^m)$

2.2.3 Theta grande (Θ)

$$T(n) = \Theta(g(n)) \Leftrightarrow \exists c_1,\ c_2\ n_0 > 0 : \forall n \geq n_0 \rightarrow 0 \leq c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$$

$g(n)$ è il **limite asintotico stretto** di $T(n)$: $T(n)$ cresce come $g(n)$.
È la combinazione dei due precedenti:

- $T(n)$ polinomio $\Rightarrow T(n) = \Theta(n^m)$;
- $T(n) = \Theta(g(n)) \leftrightarrow T(n) = O(g(n)) \wedge T(n) = \Omega(g(n))$

2.3 Online connectivity

L'**online connectivity** è un problema avente per input un insieme di coppie di interi p e q (per es. i nodi di una rete). $(p,\ q)$ definisce una relazione di connessione tra i nodi p e q.

Proprietà

- **commutativa:** se p è connesso con $q \Rightarrow q$ è connesso con p;
- **transitiva:** se p è connesso con $q \wedge q$ è connesso con $r \Rightarrow p$ è connesso con r.

Le coppie possono essere:

- mantenute: la connessione della coppia non è ancora nota (output: coppia (p, q));
- scartate: la coppia è già connessa, anche per la proprietà transitiva o commutativa (output: nullo).

Applicazioni

- reti di computer: ottimizzazione togliendo le connessioni ridondanti;
- reti elettriche;
- linguaggi di programmazione: variabili equivalenti.

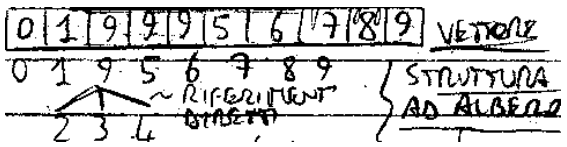
La struttura dati a **grafo** è costituita da 2 insiemi: nodi p e q e archi.
Per ottimizzare il grafo, si verifica di volta in volta se la connessione della coppia in input non è ancora nota o se è implicata già dalle precedenti. L'analisi di questo genere presuppone l'esistenza del grafo $\Rightarrow$ non è un'analisi di connettività online.
La connettività si dice **online** se si sta lavorando non su una struttura o su un modello pre-esistente, ma su un grafo che viene costruito arco per arco.

1. ipotesi: si conoscono i vertici, e ogni vertice è connesso solo a se stesso $\Rightarrow$ nessun arco;
2. **operazione find** (ricerca): a ogni coppia in input, verifica se p e q appartengono allo stesso insieme (2 find);
3. **operazione union** (unione): se p e q non erano connessi, unisci l'insieme a cui appartiene p (S_p) e l'insieme a cui appartiene q (S_q).

L'online connectivity può essere implementata tramite vettori:

- **quick-find:** find rapide, union lente;
- **quick-union:** find lente, union rapide.

2.3.1 Quick-find



A seconda del contenuto della cella del vettore:

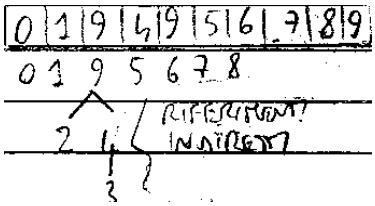
- se è il suo stesso indice i: il vertice i-esimo è connesso a se stesso;
- se è un altro indice: i due vertici appartengono allo stesso insieme.

Complessità

- find: accesso a una cella di vettore tramite il suo indice: $O(1)$
- union: scansione del vettore per cambiare da p a q: $O(n)$

Complessità di caso peggiore totale: numero di coppie $\times$ dimensioni vettore (n)
riferimenti diretti (nessuna catena) $\Rightarrow$ si trova subito il rappresentante

2.3.2 Quick-union



C'è una catena di riferimenti di tipo indiretto. A seconda del contenuto della cella del vettore:

- se è uguale al suo indice i: fine catena;
- se è diverso dal suo indice: passa alla cella i-esima a cui punta: `id[id[...id[`i`]]...] = (id[`i`])*`

Complessità

- find: due elementi sono connessi se `(id[`i`])* = (id[`j`])*`: $T(n) = O(n)$
- union: la testa della prima catena punta alla testa della seconda, cioè `(id[`p`])* = (id[`q`])*` $\Rightarrow$ l'albero si frastaglia: $T(n) = O(1)$

Nella find si deve perciò percorrere tutta la catena fino alla testa, la union è immediata (unione solo delle teste).

Capitolo 3

Gli algoritmi di ordinamento

- **ordinamento interno:** dati memorizzati in memoria centrale (accesso diretto ai dati $\Rightarrow$ si ottimizza il numero di passi);
- **ordinamento esterno:** dati memorizzati in memoria di massa (tempi di attesa più lunghi $\Rightarrow$ si ottimizza il numero di accessi).
- **ordinamento in loco:** usa oltre al vettore da ordinare una quantità di memoria ausiliaria che non dipende dal numero di dati da ordinare.
- **ordinamento stabile:** l'ordine degli elementi ripetuti eventuali è uguale tra input e output.

3.1 Classificazione per complessità

- $O\left(n^2\right)$ (quadratico): semplici, iterativi, basati sul confronto (es. insertion sort, selection sort, bubble sort);
- $O\left(n\sqrt{n}\right)$ (es. shell sort);
- $O\left(n \cdot \log n\right)$ (linearitmico): ottimi ma più complessi perché ricorsivi (es. merge sort, quick sort, heap sort);
- $O\left(n\right)$ (lineare): basati sul calcolo, ma ipotesi restrittive sui dati, cioè possono essere usati solo con determinati dati (es. counting sort).

3.1.1 Limite inferiore della complessità asintotica nel caso peggiore per gli algoritmi di ordinamento basati sul confronto

Gli algoritmi basati sul confronto non possono avere una complessità inferiore a $n \log n$ (linearitmica), solo quelli basati sul calcolo:

$$T\left(n\right) = \Omega\left(n \cdot \log n\right)$$

Dimostrazione

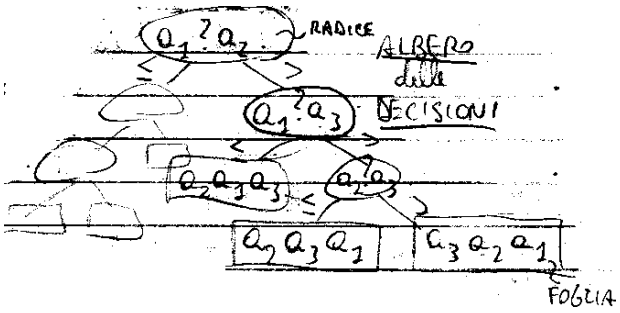


Figura 3.1: Caso di 3 interi (a_1, a_2, a_3).

Osservazioni:

- 3! permutazioni possibili
- 3 confronti nel caso peggiore

Tabella 3.1: Caso di n interi.

livello i	**numero di foglie** 2^i
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$

- osservazione: $n!$ permutazioni possibili $\leq$ numero di foglie;
- servono almeno 2^h foglie per accomodare tutte le $n!$ permutazioni possibili $\Rightarrow 2^h \geq n!$;
- tramite la relazione di Stirling $n! > \left(\frac{n}{e}\right)^n$:[1]

$$2^h > \left(\frac{n}{e}\right)^n;\ \log_2 2^h > \log\left(\frac{n}{e}\right)^n;\ h > n\left(\log_2 n - \log_2 e\right) \Rightarrow T(n) = \Omega\left(n \log n\right) \quad n \to +\infty$$

3.2 Bubble sort (o exchange sort)

Al 1° ciclo, confronta due elementi adiacenti, se necessario li scambia, quindi passa alla casella successiva $\Rightarrow$ si ottiene solo che l'ultimo elemento a destra è il massimo, ma gli altri non sono ordinati, cioè il valore massimo è galleggiato a destra.

Il vettore si divide idealmente in due sottovettori $\Rightarrow$ serve un doppio ciclo annidato:

- sottovettore destro: inizialmente vuoto, contiene via via i massimi ordinati;
- sottovettore sinistro: si svuota man mano.

Ottimizzazione: è possibile usare un flag per terminare prima il ciclo se non è stato effettuato alcuno scambio nel ciclo interno.

[1] La base del logaritmo scompare perché si sottointende la moltiplicazione per una costante per il cambio di base.

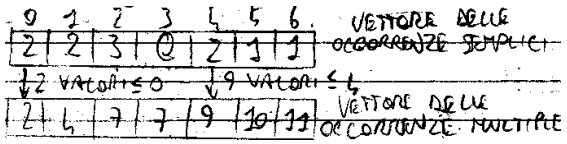
3.3 Selection sort

Al 1° ciclo, scandisce il vettore, trova la posizione del minimo, e lo scambia con il 1° elemento. Pertanto è il sottovettore sinistro che si riempie in modo ordinato, man mano che il ciclo viene incrementato ⇒ il selection sort e il bubble sort sono degli **algoritmi incrementali**. Il ciclo interno cerca il minimo nel sottovettore destro.

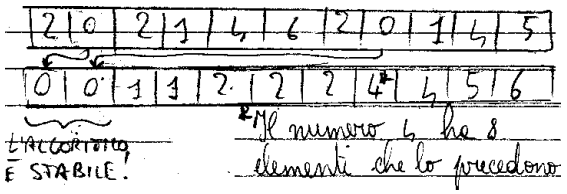
3.4 Counting sort

È un algoritmo stabile (va bene per le chiavi ripetute), ma non è un algoritmo in loco. Va a calcolare direttamente la posizione finale di un dato.

L'algoritmo crea il **vettore delle occorrenze semplici**, che contiene il numero di occorrenze nell'input del dato che corrisponde all'indice, e poi crea il **vettore delle occorrenze multiple**:



Leggendo il vettore in input da destra verso sinistra, pone ogni elemento nella posizione memorizzata nel vettore delle occorrenze multiple, quindi decrementa il valore posizione:



Utilizza 3 vettori: il vettore di partenza, il vettore risultato, e il vettore delle occorrenze. La dimensione del vettore delle occorrenze dipende dall'intervallo a cui possono appartenere i dati ⇒ non è un ordinamento in loco.

Viene implementato con 4 cicli `for` ⇒ $T(n) = O(n + k)$: si considera non più solo la dimensione dei dati, ma anche la loro complessità k, che cresce linearmente con n, ma la complessità k non può tendere a infinito. Supponendo $k \sim n$: $T(n) = O(n)$.

Capitolo 4

Grafi e alberi

4.1 Grafi

$G = (V, E)$:

- V = insieme finito dei vertici/nodi
- E = insieme finito di archi, che mette coppie di vertici in relazione (binaria)
- **grafo non orientato:** non esiste un verso di percorrenza
- **grafo orientato:** un arco può essere percorso in un solo verso

cappio arco che ritorna sul vertice di partenza (solo nei grafi orientati)

Due vertici si dicono tra loro **adiacenti** se sono connessi da un arco.

grado di un vertice numero di archi che insistono su quel vertice
per i grafi orientati: `in_degree` (entranti), `out_degree` (uscenti)

Esiste un **cammino** tra due vertici se esiste una sequenza di vertici per cui il primo vertice è uguale a uno dei due vertici e l'ultimo è uguale all'altro vertice, e se esiste una sequenza di k archi.

cammino semplice non ripassa mai su un nodo già passato ⇒ i vertici sono tutti distinti

ciclo o **cammino chiuso** il vertice di partenza e di arrivo coincidono
caso particolare: cappio = ciclo di lunghezza 1

grafo aciclico privo di cicli ⇒ se si ha bisogno che l'elemento precedente sia chiuso prima di passare al successivo, si deve verificare che non ci siano cicli che impediscono l'avvio

grafo connesso se si prende una qualsiasi coppia di vertici, esiste sempre un cammino che connette tale coppia (solo nei grafi non orientati)

componente connessa è il sottografo massimale[1] di un grafo per cui tutti i vertici sono mutualmente connessi (solo nei grafi non orientati)

Nei grafi orientati la raggiungibilità in un senso non implica la raggiungibilità nell'altro.

grafo fortemente connesso ogni coppia di vertici è raggiungibile in entrambi i sensi

[1] **massimale**: è il più grande dei sottoinsiemi per cui vale questa proprietà, cioè quello che comprende più vertici connessi possibile.

grafo completo esistono tutti gli archi per tutte le coppie di vertici:

- nei grafi non orientati: $|E| = \frac{(|V|-1)\cdot|V|}{2}$
- nei grafi orientati: $|E| = (|V|-1)\cdot|V|$

dove:

- $|E|$ è il numero di archi;
- $|V|$ è il numero di vertici;
- $|V|-1$ è il numero di archi per ogni vertice.

$|E|$ può crescere al più con il quadrato di $|V|$:

- nei **grafi densi**: $|E| \cong \left|V^2\right|$
- nei **grafi sparsi**: $|E| \ll \left|V^2\right|$

grafo pesato una funzione che riceve in input una coppia di nodi e restituisce il peso della coppia (generalmente un numero intero)

4.2 Alberi

albero non radicato grafo non orientato, connesso, aciclico (no nodi in particolare)

foresta insieme di alberi

Proprietà G albero non radicato:

- ogni coppia di nodi è connessa da un solo **cammino semplice**;
- G **connesso:** rimuovere qualsiasi arco comporta sconnettere il grafo;
- $|E| = |V| - 1$;
- G **aciclico:** aggiungere un arco comporta introdurre un ciclo.

4.2.1 Alberi radicati

Si individua il nodo radice $r \Rightarrow$ si stabiliscono delle relazioni padre-figlio tra le coppie di vertici (1 arco).

Se $y \in$ cammino da r a $x \Rightarrow y$ antenato, x discendente (più archi). L'antenato è **proprio** se $x \neq y$.

Casi particolari

- radice: no padre
- foglie: no figli

Proprietà

- **grado** = numero massimo di figli;
- **profondità** x = lunghezza del cammino da r (livello);
- **altezza** h = profondità massima.

Alberi binari

albero binario albero con grado 2 (ogni nodo può avere 0, 1 o 2 figli). Definizione risorsiva:

- insieme di nodi vuoto (foglia)
- radice, sottoalbero sinistro, sottoalbero destro

albero binario completo ogni nodo o è una foglia o ha 2 figli:

- numero di foglie: 2^h
- numero di nodi: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$[2]

albero binario bilanciato tutti i cammini radice-foglia hanno uguale lunghezza (non c'è una foglia più vicina di altre alla radice). Un albero completo è sempre bilanciato (ma non viceversa).

albero binario quasi bilanciato c'è al massimo uno scarto di una unità di lunghezza

[2]Si ricorda la serie geometrica: $\sum_{i=0}^{h} x^i = \frac{x^{h+1}-1}{x-1}$

Capitolo 5

La ricorsione

- **ricorsione diretta:** nella definizione di una procedura si richiama la procedura stessa
- **ricorsione indiretta:** nella definizione di una procedura si richiama una o più procedure che richiamano la procedura stessa

Si rompe ricorsivamente un problema in analoghi sottoproblemi più semplici, fino a quando non si arriva alla soluzione di un problema elementare (si veda il capitolo 6).

Non è una definizione circolare (all'infinito), ma c'è la condizione di terminazione (ricorsione finita).

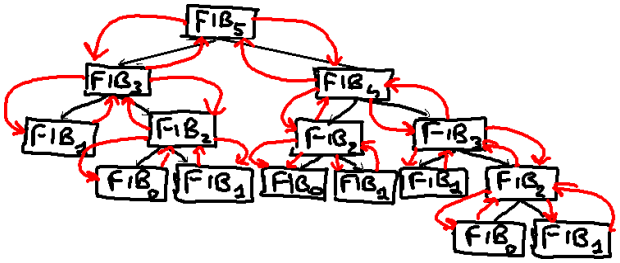
5.1 Esempi

5.1.1 Fattoriale

La definizione stessa di fattoriale è di tipo ricorsivo.

A ogni passo si genera un unico sottoproblema anziché due ⇒ non sarà un albero binario completo.

5.1.2 Numeri di Fibonacci



Un numero di Fibonacci FIB_{n+1} è la somma dei due che lo precedono nell'ordinamento: FIB_{n-1} e FIB_n.

La ratio aurea[1] è il rapporto tra un segmento maggiore a e uno minore b: $\varphi = \frac{a}{b}$. Il segmento maggiore a è il medio proporzionale tra il minore b e la somma dei due:

$$(a+b) : a = a : b;\ \frac{\varphi+1}{\varphi} = \varphi;\ \varphi^2 - \varphi - 1 = 0;\ \varphi = \frac{1+\sqrt{5}}{2} \vee \varphi' = \frac{1-\sqrt{5}}{2}$$

[1] Si veda la voce Sezione aurea su Wikipedia in italiano.

Ogni numero di Fibonacci ha la seguente relazione con la ratio aurea φ:

$$\text{FIB}_n = \frac{\varphi^n - \varphi'^n}{\sqrt{5}}$$

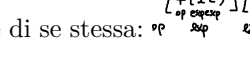
La ricorsione non procede in modo parallelo per livelli dell'albero, ma è un'analisi in profondità dell'**albero della ricorsione**, ovvero si segue ogni cammino fino alla foglia = soluzione elementare.

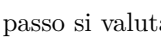
5.1.3 Algoritmo di Euclide

Dati m e n, permette di calcolare il Massimo Comun Divisore:

- se $m > n \Rightarrow \text{MCD}(m, n) = \text{MCD}(n, m \bmod n)$
- condizione di terminazione: se $n = 0$ ritorna m

5.1.4 Valutazione di espressione in forma prefissa[2]

Ponendo per semplicità di trattare solo con operandi di somma e prodotto di arità[3] 2, una espressione può essere definita ricorsivamente in funzione di se stessa:  , con condizione di terminazione: l'espressione è uno degli operandi.

Inserendo l'espressione in un vettore  , a ogni passo si valuta `a[i]`:

- se `a[i]` è un operatore, valuta l'espressione a destra dell'operatore;
- condizione di terminazione: se `a[i]` è un numero, ritornalo.

5.1.5 Ricerca binaria

Si considera a ogni passo della ricorsione un sottovettore di metà dimensione, anch'esso ordinato.

5.1.6 Ricerca del massimo di un vettore

A ogni passo, si suddivide il vettore in due sottovettori, si recupera il massimo di ciascun sottovettore, e si confrontano i risultati. Si termina quando il sottovettore ha un solo elemento.

5.1.7 Moltiplicazione rapida di 2 interi

1° modo) Seguo la definizione ricorsiva: $\begin{cases} x \cdot y = x + x \cdot (y-1) \\ x \cdot 1 = x \end{cases}$

2° modo) Si assume:

- si sa calcolare il prodotto solo di cifre decimali (come se si avessero solo le tavole pitagoriche[4]);
- per semplicità che x e y abbiano $x = 2^k$ cifre.

Si divide x in due sottovettori x_s e x_d di metà dimensione, e così pure y in y_s e $y_d \Rightarrow$ $\begin{cases} x = x_s \cdot 10^{\frac{n}{2}} + x_d \\ y = y_s \cdot 10^{\frac{n}{2}} + y_d \end{cases}$

$$x \cdot y = \left(x_s \cdot 10^{\frac{n}{2}} + x_d\right) \cdot \left(y_s \cdot 10^{\frac{n}{2}} + y_d\right) = x_s \cdot y_s \cdot 10^n + x_s \cdot y_d \cdot 10^{\frac{n}{2}} + x_d \cdot y_s \cdot 10^{\frac{n}{2}} + x_d \cdot y_d$$

[2] Per approfondire, si veda la voce Notazione polacca su Wikipedia in italiano.
[3] Si veda la voce Arietà su Wikipedia in italiano.
[4] Si veda la voce Tavola pitagorica su Wikipedia in italiano.

Si continua a suddividere ciascun sottovettore fino alla condizione di terminazione: i fattori hanno una sola cifra.

A ogni passo, si generano 4 sottoproblemi di dimensione metà ⇒ si richiama per 4 volte l'operazione di moltiplicazione.

5.1.8 Moltiplicazione rapida di 2 interi ottimizzata

Il numero di moltiplicazioni richiamate a ogni passo si riduce a 3:

$$x_s \cdot y_d + x_d \cdot y_s = x_s \cdot y_s + x_d \cdot y_d - (x_s - x_d) \cdot (y_s - y_d)$$

5.2 Liste

5.2.1 Definizioni

sequenza lineare insieme finito di elementi, ciascuno associato a un indice univoco, in cui conta la posizione reciproca tra di essi (cioè ogni elemento ha un successore e un predecessore)

1. **accesso diretto:** (implementazione tramite vettore) locazioni di memoria contigue accessibili tramite indice ⇒ complessità $O(1)$;
2. **accesso sequenziale:** l'accesso a un certo elemento necessita della scansione sequenziale della lista a partire dal primo elemento ⇒ complessità $O(n)$.

Una lista è una sequenza lineare ad accesso sequenziale. La lista è un concetto astratto, e si può implementare in C:

- tramite un vettore: lo posso usare anche per dati in locazioni di memoria non contigue;
- tramite un sistema a puntatori 

: in questo caso la lista si dice concatenata.

lista concatenata insieme di elementi dello stesso tipo (dati), memorizzati in modo non contiguo (nodi, implementati tramite struct), accessibili mediante riferimento (link ⇒ puntatori) al successore/precedente. La memoria viene allocata e liberata dinamicamente per ogni nodo (⇒ i dati possono teoricamente essere infiniti), ma l'accesso non è diretto.

5.2.2 Classificazione

- ordinata / non ordinata
- circolare / con testa e coda
- singolo-linkata / doppio-linkata (senza/con link a successore)

5.2.3 Ricorsione

Definizione ricorsiva Una lista è un elemento seguito da una lista. Funzioni:

- conteggio: a ogni passo: 1 + numero degli elementi della sottolista considerata a partire dal successore;
- attraversamento: elenca gli elementi in una lista con una strategia (ordine) predefinita di percorrimento;
- eliminazione di un elemento.

Non sempre la soluzione ricorsiva è più efficiente di quella iterativa.

Capitolo 6

Il paradigma "divide et impera"

1. **divide:** un problema di dimensione n viene ripartito in a sottoproblemi, ciascuno di dimensione $\frac{n}{b}$, tra loro indipendenti ($\Leftrightarrow$ ricorsione su ogni sottoproblema);
2. **impera:** quando si arriva alla soluzione elementare, avviene la risoluzione diretta ($\Leftrightarrow$ condizione di terminazione);
3. **combina:** le soluzioni parziali (elementari) vengono ricombinate per ottenere la soluzione del problema di partenza.

6.1 Equazioni alle ricorrenze

L'**equazione alle ricorrenze** permette di analizzare la complessità di ogni passo del procedimento divide et impera:

$$\begin{cases} T(n) = D(n) + a \cdot T\left(\frac{n}{b}\right) + C(n), & n > c \\ T(n) = \Theta(1), & n \leq c \end{cases}$$

1. divide: $D(n)$ (costo della divisione)
2. impera: $\Theta(1)$ (costo della soluzione elementare)
3. combina: $C(n)$ (costo della ricombinazione)

- a = numero di sottoproblemi che risulta dalla fase di divide
- $\frac{n}{b}$ = dimensione di ciascun sottoproblema

Eseguire l'**unfolding** significa sviluppare alcuni passi di un'equazione alle ricorrenze per trovare una forma analitica chiusa della ricorrenza stessa sotto forma di sommatoria. Serve per analizzare la complessità di una funzione ricorsiva.

6.1.1 Ricerca binaria

Equazione alle ricorrenze

$$\begin{cases} D(n) = \Theta(1) \\ C(n) = \Theta(1) \\ a = 1,\ b = 2 \end{cases} \rightarrow \begin{cases} T(n) = T\left(\frac{n}{2}\right) + 1\cancel{+1}, & n > 1 \\ T(1) = 1, & n = 1 \end{cases}$$

Unfolding

1. sviluppo: $T(n) = T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{4}\right) + 1 + 1 = \cdots = T\left(\frac{n}{2^i}\right) + \sum 1 = \cdots$
2. condizione di terminazione: $T\left(\frac{n}{2^i}\right) = T(1)$; $\frac{n}{2^i} = 1$; $i = \log_2 n$[1]
3. sommatoria: $T(n) = \sum_{i=0}^{\log_2 n} 1 = 1 + \log_2 n$
4. complessità asintotica: $T(n) = O(\log n) \Rightarrow$ algoritmo logaritmico

6.1.2 Ricerca del massimo di un vettore

Equazione alle ricorrenze

$$\begin{cases} D(n) = \Theta(1) \\ C(n) = \Theta(1) \\ a = 2,\ b = 2 \end{cases} \rightarrow \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + 1\cancel{+1}, & n > 1 \\ T(1) = 1, & n = 1 \end{cases}$$

- $a = 2$: il vettore viene suddiviso in due sottovettori
- $b = 2$: ciascun sottovettore è ampio la metà del vettore di partenza

Unfolding

- $T(n) = 1 + 2 + 4 + 8T\left(\frac{n}{8}\right) = 1 + 2 + 4 + \cdots = \sum_{i=0}^{\log_2 n} 2^i$
- progressione geometrica[2] $\Rightarrow T(n) = \frac{2^{\log_2 n + 1} - 1}{2 - 1} = 2^{\log_2 n + 1} - 1 = 2n - 1$
- $T(n) = O(n) \Rightarrow$ algoritmo lineare

6.1.3 Moltiplicazione rapida di 2 interi

$$x \cdot y = x_s \cdot y_s \cdot 10^n + x_s \cdot y_d \cdot 10^{\frac{n}{2}} + x_d \cdot y_s \cdot 10^{\frac{n}{2}} + x_d \cdot y_d$$

Equazione alle ricorrenze

$$\begin{cases} D(n) = \Theta(1) \\ C(n) = \Theta(n) \\ a = 4,\ b = 2 \end{cases} \rightarrow \begin{cases} T(n) = 4T\left(\frac{n}{2}\right) + n\cancel{+1}, & n > 1 \\ T(1) = 1, & n = 1 \end{cases}$$

- $C(n) = \Theta(n)$: per eseguire la somma binaria di due numeri di n cifre occorre scandire ciascuna cifra
- $a = 4$: i problemi ricorsivi sono le 4 operazioni di prodotto (le somme hanno complessità lineare ma non sono ricorsive)
- $b = 2$: x_s, x_d, y_s e y_d sono ampi la metà dei vettori di partenza

Unfolding

- $T(n) = n + 4 \cdot \frac{n}{2} + 4^2 \cdot \frac{n}{4} + 4^3 \cdot T\left(\frac{n}{8}\right) = n(1 + 2 + 4 + \cdots) = n \sum_{i=0}^{\log_2 n} 2^i$
- progressione geometrica $\Rightarrow T(n) = n \cdot \left(2^{\log_2 n + 1} - 1\right) = 2n^2 - n$
- $T(n) = O\left(n^2\right) \Rightarrow$ algoritmo quadratico

[1] Per semplicità si suppone n una potenza esatta di 2.
[2] Si veda la sezione Serie geometrica nella voce Progressione geometrica su Wikipedia in italiano.

6.1.4 Moltiplicazione rapida di 2 interi ottimizzata

Equazione alle ricorrenze

$$a = 3 \rightarrow T(n) = 3T\left(\frac{n}{2}\right) + n$$

Unfolding

- $T(n) = n + 3 \cdot \frac{n}{2} + 3^2 \cdot \frac{n}{4} + 3^3 \cdot T\left(\frac{n}{8}\right) = n\left[1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots\right] = n\sum_{i=0}^{\log_2 n}\left(\frac{3}{2}\right)^i$
- progressione geometrica $\Rightarrow T(n) = 2n\left[\left(\frac{3}{2}\right)^{\log_2 n+1} - 1\right] = 3 \cdot 3^{\log_2 n} - 2n$
- per la proprietà dei logaritmi: $a^{log_b n} = n^{log_b a} \rightarrow T(n) = 3 \cdot n^{log_2 3} - \cancel{2n}$
- $T(n) = O\left(n^{\log_2 3}\right) \Rightarrow$ più efficiente della moltiplicazione non ottimizzata ($\log_2 3 < 2$)

6.1.5 Le Torri di Hanoi

Ho $k = 3$ pioli verticali, con $n = 3$ dischi forati in ordine decrescente di diametro sul primo piolo. Si vuole portarli tutti e 3 sul terzo piolo. Condizioni per lo spostamento:

1. si può spostare un solo piolo per volta;
2. sopra ogni disco solo dischi più piccoli.

Il problema si suddivide in 3 sottoproblemi:

1. problema da $n-1$: 000 $\rightarrow$ 011 (2 deposito) $\Rightarrow$ si suddivide a sua volta in 3 sottoproblemi elementari
2. problema da 1: 011 $\rightarrow$ 211 $\Rightarrow$ sottoproblema elementare
3. problema da $n-1$: 211 $\rightarrow$ 222 (0 deposito) $\Rightarrow$ si suddivide a sua volta in 3 sottoproblemi elementari

Equazione alle ricorrenze $T(n) = 2T(n-1) + 1$

1. dividi: $D(n) = \Theta(1)$ (considero $n-1$ dischi)
2. risolvi: $2T(n-1)$ (ho 2 sottoproblemi ciascuno da $n-1$)
3. impera: $\Theta(1)$ (termino quando sposto un disco solo)
4. combina: $C(n) = \Theta(1)$ (nessuna combinazione)

Unfolding

- $T(n) = 1 + 2 + 4 + 8T(n-3) = 2^0 + 2^1 + 2^2 + \cdots$
- $n - i = 1;\ i = n - 1 \rightarrow T(n) = \sum_{i=0}^{n-1} 2^i$
- progressione geometrica $\Rightarrow T(n) = 2^n - \cancel{1}$
- $T(n) = O(2^n) \Rightarrow$ algoritmo esponenziale (anche se decidibile)

6.1.6 Il righello

Disegnare le tacche di un righello in maniera ricorsiva, di differenti altezze, fino a quando si arriva all'altezza 0. Si disegna ricorsivamente a metà del sottointervallo la tacca, quindi si considerano i due sottointervalli sinistro e destro e si disegna la tacca di una unità di altezza inferiore.

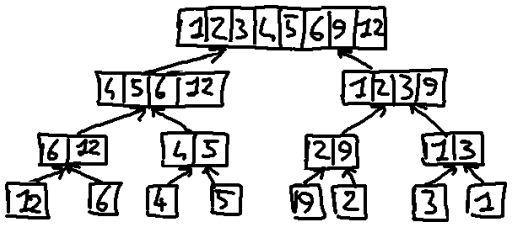
Backtrack Si esplora una scelta per volta, e se la scelta non va bene si ritorna indietro. (vd. filo di Arianna) Si termina quando tutte le scelte sono esaurite. Tramite il backtrack si può esplorare tutto lo spazio delle soluzioni.

6.2 Gli algoritmi ricorsivi di ordinamento

6.2.1 Merge sort (o ordinamento per fusione)

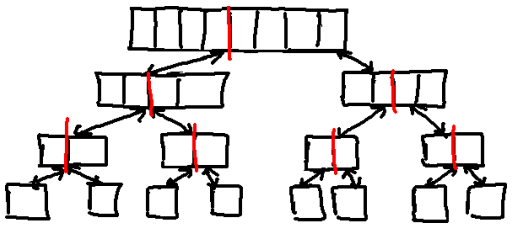
Passi

1. divide: si suddivide il vettore in due sottovettori
2. ricorsione:
   - si applica il merge sort sul sottovettore sinistro
   - si applica il merge sort sul sottovettore destro
   - condizione di terminazione: il sottovettore ha 1 cella
3. combina: si uniscono tutti i sottovettori ordinati, scandendo in parallelo i due sottovettori da sinistra verso destra e confrontando di volta in volta i valori scegliendo quello minore:



Analisi di complessità

Ragionamento intuitivo: ho $\log_2 n$ livelli di ricorsione e devo fare $\frac{n}{2} + \frac{n}{2} = n$ unioni $\Rightarrow n \log_2 n$ operazioni totali:



Equazione alle ricorrenze

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

1. dividi: $D(n) = \Theta(1)$ (calcola la metà di un vettore)
2. risolvi: $2T\left(\frac{n}{2}\right)$ (risolve 2 sottoproblemi di dimensione $\frac{n}{2}$ ciascuno)
3. terminazione: $\Theta(1)$ (semplice test)
4. combina: $C(n) = \Theta(n)$

Unfolding

$$T(n) = n + 2 \cdot \frac{n}{2} + 2^2 \cdot \frac{n}{4} + 2^3 T\left(\frac{n}{8}\right) = \sum_{i=0}^{\log_2 n} n = n \sum_{i=0}^{log_2 n} 1 = n(1 + \log_2 n) = n \log_2 n \cancel{+n}$$

$T(n) = O(n \log n) \Rightarrow$ algoritmo linearitmico (ottimo)

6.2.2 Quicksort

Passi

È quadratico ($O\left(n^2\right)$) $\Rightarrow$ secondo la teoria degli algoritmi non dovrebbe essere ottimo, però: sperimentalmente si dimostra che il caso peggiore ricorre molto raramente, a differenza del caso medio.

Partizione La divisione non avviene a metà, ma secondo una certa proprietà: sceglie arbitrariamente un elemento separatore (= **pivot**), quindi separa gli altri valori in sottovettore sinistro e destro a seconda se sono minori o maggiori del pivot.

Individua (tramite un ciclo ascendente su i e un ciclo discendente su j) una coppia di elementi di indici i e j che siano entrambi fuori posto (ovvero l'i-esimo è maggiore del pivot, e il j-esimo è minore del pivot), quindi li scambia.

Condizione di terminazione: `i == j` (si ferma quando tutte le coppie sono già a posto).

Costo della partizione: $T(n) = \Theta(n)$ (cioè la scansione di tutti gli n elementi).

Al termine della ricorsione è inutile una fase di ricombinazione, perché i dati si trovano già nella posizione corretta.

Analisi di complessità

Il quicksort non segue la formula generale del divide et impera.

Caso peggiore Il vettore si trova in ordine inverso $\Rightarrow$ la partizione genera un sottovettore da $n-1$ elementi e l'altro da 1 elemento.

Equazione alle ricorrenze

$$T(n) = T(1) + T(n-1) + n + 1 = T(n-1) + n \cancel{+2}$$

- $T(1)$ = costo per il sottovettore da 1 elemento
- $T(n-1)$ = costo per il sottovettore da $n-1$ elementi
- n = costo della partizione
- 1 = costo unitario della risoluzione (riguarda gli elementi al termine della ricorsione)

Unfolding

$$T(n) = n + T(n-1) = n + (n - 1 + T(n-2)) = \cdots = \frac{n}{2}(n+1)$$

Caso migliore Ogni sottovettore è esattamente la metà $\Rightarrow$ ricorda il merge sort (in questo caso è linearitmico).

Caso medio Partizione fortemente sbilanciata, senza arrivare al caso peggiore $\Rightarrow$ il quicksort è linearitmico.

È meglio cercare un pivot in modo da dividere sempre in maniera conveniente il vettore $\Rightarrow$ la complessità non cambia (neanche prendendo un pivot che separa sempre il vettore a metà), ma si possono ridurre le costanti.

Capitolo 7

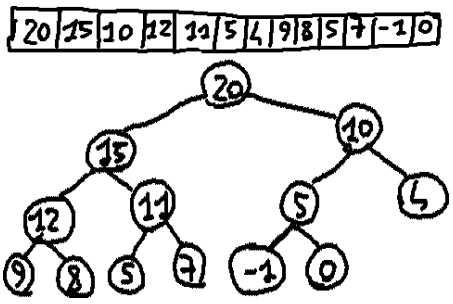
Gli heap

7.1 Definizioni

7.1.1 Coda a priorità

coda a priorità[1] struttura dati atta a mantenere un insieme S di dati aventi chiave e priorità
⇒ l'elemento inserito si posiziona nell'insieme a seconda della sua priorità, e l'elemento servito per primo è quello a massima priorità

7.1.2 Heap



heap[2] albero binario con:

- proprietà strutturale: tutti i livelli sono completi, tranne al più l'ultimo riempito da sinistra verso destra (albero quasi bilanciato);
- proprietà funzionale: la chiave contenuta nel nodo padre è $\geq$ le chiavi contenute in ciascuno dei suoi sottoalberi figli ⇒ la chiave massima si trova nella radice (si considera solo $>$ per escludere le chiavi duplicate).

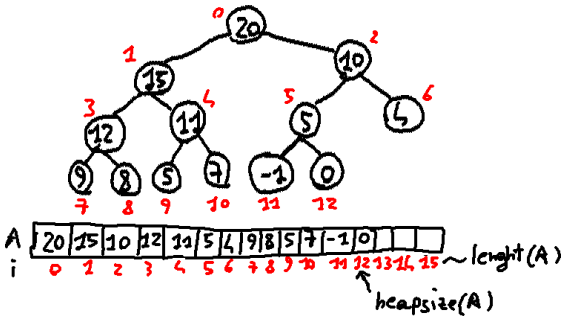
7.2 Implementazione

La coda a priorità si può implementare con l'heap, dove la priorità è in funzione del tempo di arrivo:

- coda FIFO[3] (**coda**): il primo che entra è il primo a uscire/essere servito (es. coda alla cassa);

[1] Si veda la voce Priority queue su Wikipedia in inglese.
[2] Si veda la voce Heap (data structure) su Wikipedia in inglese.
[3] Si veda la voce FIFO su Wikipedia in inglese.

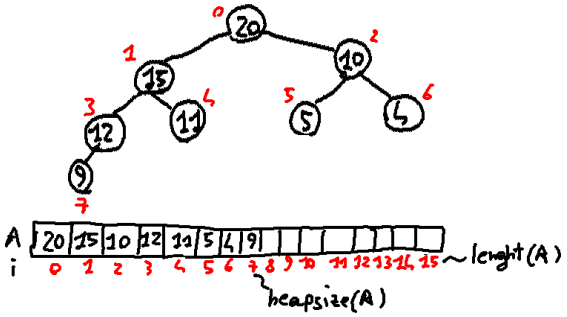
- coda LIFO[4] (**stack**): l'ultimo che entra è il primo a uscire/essere servito (es. pila di libri).



Si etichetta ogni nodo con un intero crescente. Ogni figlio sinistro ha indice $2i + 1$, ogni figlio destro ha indice $2i + 2$. Per riempire l'heap, si usa un vettore e si scandiscono le celle contigue entro l'heap-size (= numero di elementi nell'heap).

7.3 Procedure

Le operazioni per trovare l'indice del figlio conoscendo quello del padre (o viceversa) sono efficienti. Un albero completo ha $2^{h+1} - 1$ nodi $\Rightarrow$ per rappresentare l'heap seguente:



si dovrebbe sovradimensionare il vettore a 15 celle, sprecando circa metà delle celle $\Rightarrow$ è nel complesso ugualmente accettabile perché le operazioni sono comunque semplici.

7.3.1 Insert (**complessità:** $T(n) = O(\log n)$)

Aggiunge una chiave in base alla sua priorità:

- se il livello inferiore è già pieno, aggiunge un nuovo livello;
- se il livello inferiore non è ancora pieno, aggiunge una foglia partendo da sinistra verso destra.

Per rispettare la proprietà funzionale dell'heap, confronta la chiave nuova con la chiave del padre: se la chiave del padre è inferiore, sposta il padre al livello inferiore e ripete l'operazione con il "nonno", arrivando al più alla radice.

Complessità È un albero quasi bilanciato $\Rightarrow$ la lunghezza del cammino-foglia è logaritmica nel numero di dati. Il caso peggiore è il percorrimento di un cammino radice-foglia, cioè il numero dei confronti massimi che eseguo è limitato dalla lunghezza massima del cammino radice-foglia $\Rightarrow$ complessità: $T(n) = O(\log n)$.

[4] Si veda la voce LIFO su Wikipedia in inglese.

7.3.2 Heapify (complessità: $T(n) = O(\log n)$)

Procedura di servizio che trasforma in un heap una terna (sottoalbero sinistro; radice; sottoalbero destro), con la condizione che i due sottoalberi sinistro e destro soddisfino già le proprietà dell'heap (caso particolare: foglia).

Passi

- individua il massimo nella terna;
- se il massimo è in uno dei sottoalberi, scambia la radice del sottoalbero con la radice;
- scende ricorsivamente sul sottoalbero con cui è avvenuto lo scambio.

Caso peggiore cammino radice-foglia

Condizione di terminazione la ricorsione arriva ad una foglia

7.3.3 BuildHeap (complessità: $T(n) = O(n)$)

Un heap si può costruire in due modi:

1. si può inserire una serie di dati tramite operazioni di Insert consecutive in una struttura vuota;
2. tramite la procedura BuildHeap: partendo da un vettore di dati, lo interpreta come un albero binario, quindi applica operazioni di Heapify considerando terne dal basso verso l'alto e da destra verso sinistra $\Rightarrow$ ciclo for discendente dall'indice dell'ultimo padre all'indice 0 della radice. Complessità: nonostante si effettuino $\sim n$ operazioni di Heapify di costo $\log n$, si dimostra che la complessità non è linearitmica ma è: $T(n) = O(n)$

7.3.4 HeapSort (complessità: $T(n) = O(n \log n) \Rightarrow$ algoritmo ottimo)

Passi BuildHeap $\rightarrow$ scambio ultima_foglia-radice, ponendo quindi l'elemento massimo in fondo al vettore $\rightarrow$ ora lavora su un heap di dimensione $heapsize - 1$, che corrisponde alla parte sinistra non ancora ordinata del vettore $\rightarrow$ applica l'Heapify sull'heap rimanente (solo il primo elemento è fuori posto $\Rightarrow$ evita la BuildHeap) $\rightarrow$ scambia ultima_foglia_corrente-radice $\rightarrow$ ...

7.3.5 Altre

- maximum: ritorna l'elemento a priorità massima (complessità: $T(n) = \Theta(1)$)
- extract_max: estrae l'elemento a priorità massima (complessità: $T(n) = O(\log n)$)

Capitolo 8

Le tabelle di simboli

tabella di simboli (o dizionario) struttura dati formata da **record**, ovvero struct con dati aggregati di cui uno è la chiave

Si può estrarre la chiave dal record e stabilire se una chiave è minore o uguale a un'altra determinando una relazione d'ordine.

8.1 Algoritmi di ricerca

Gli **algoritmi di ricerca**, cioè le implementazioni delle tabelle di simboli, possono avere strutture lineari, ad albero o a tabella di hash.

8.1.1 Strutture lineari

- array: ordinati o non ordinati;
- liste: ordinate o non ordinate;
- tabelle ad accesso diretto: l'insieme K contenente tutti i dati memorizzati è un sottoinsieme dell'insieme universo U:
  - vantaggio: ogni dato in un suo sottoinsieme K ha un indice $\Rightarrow$ l'accesso è diretto (ricerca indicizzata per chiave);
  - svantaggio: il vettore dovrà avere un numero di celle pari alla cardinalità[1] di U $\Rightarrow$ rimangono delle celle vuote $\Rightarrow$ spreco di memoria lineare nel numero di dati, soprattutto se la cardinalità di K è molto minore di quella di U.

8.1.2 Strutture ad albero

- alberi binari di ricerca (BST): efficienti;
- alberi bilanciati: (es. RB-tree) garantiscono che le operazioni siano sempre di complessità logaritmica e non degenerino in lineari.

8.2 Operazioni

- data una chiave $\Rightarrow$ inserimento di un nuovo elemento
- data una chiave $\Rightarrow$ ricerca di un elemento[2]

[1] Si veda la voce Cardinalità su Wikipedia in italiano.
[2] Si distinguono le ricerche con successo (**hit**) e quelle con insuccesso (**miss**).

- data una chiave ⇒ cancellazione di elemento specificato
- dato un numero k ⇒ selezione del k-esimo elemento più piccolo
- ordinamento della tabella di simboli in base alla chiave
- unione di due tabelle di simboli

8.2.1 Complessità di caso peggiore

La complessità dell'inserimento di un nuovo elemento all'interno di una lista o di un array varia a seconda se gli elementi sono ordinati:

- non ordinati: l'inserimento ha un costo unitario (per es. inserimento sempre in testa);
- ordinati: l'inserimento ha un costo lineare, perché bisogna inserire il nuovo elemento mantenendo l'ordinamento:
  - array: bisogna spostare tutti gli elementi di una cella per far spazio al nuovo elemento;
  - liste: l'inserimento implica una ricerca per scansione.

La ricerca binaria si deve effettuare su un array ordinato ⇒ può essere inefficiente se l'array varia molto e bisogna ordinarlo spesso.

8.2.2 Complessità di caso medio

Nel caso medio:

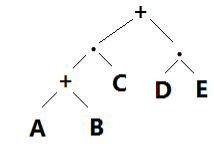
- le operazioni sulle strutture lineari sono quasi sempre lineari nel numero dei dati;
- le operazioni sulle strutture ad albero sono quasi sempre logaritmiche nel numero dei dati (nel caso peggiore sono lineari);
- le operazioni sulle tabelle di hash hanno costo unitario, ma senza spreco di memoria come nelle tabelle ad accesso diretto.

Capitolo 9

Alberi binari di ricerca (BST)

9.1 Operazioni su alberi binari

9.1.1 Attraversamento di un albero binario



+·+ABC·DE (pre order)
A+B·C+D·E (in order)
AB+C·DE·+ (post order)

Strategie di visita degli alberi binari (Root = radice, L = sottoalbero sinistro, R = sottoalbero destro)

- **pre order:** Root, L, R
- **in order:** L, Root, R
- **post order:** L, R, Root

L'operazione di attraversamento ha complessità lineare.

9.1.2 Calcolo ricorsivo di parametri

- numero di nodi: 1 Root + ricorsione(L) + ricorsione(R)
- altezza: 1 + altezza del sottoalbero maggiore (calcolata in modo ricorsivo)

9.2 Alberi binari di ricerca (BST)

albero binario di ricerca albero binario in cui, per ogni radice, si trovano nodi le chi chiavi sono minori o uguali nel sottoalbero sinistro e nodi le cui chiavi sono maggiori o uguali in quello destro ⇒ la radice è l'elemento di separazione tra dati (chiavi) minori a sinistra e maggiori a destra.

9.2.1 Operazioni di base

Search

Ricerca una chiave, determinando a ogni radice il sottoalbero in cui cercare con un semplice confronto.

Casi di terminazione

- search hit: la chiave è stata trovata in una radice;
- search miss: la chiave non è stata trovata e si è giunti a un albero vuoto.

Min/Max

Ricerca il valore minimo/massimo, percorrendo tutti i sottoalberi sinistri/destri.

Sort

Per ottenere l'ordinamento crescente delle chiavi basta visitare il BST in order. Per fare questo è necessario visitare tutti i nodi dell'albero: l'operazione ha quindi complessità lineare nel numero di nodi.

Successor

1° modo) Si ordinano i valori nell'albero con l'operazione Sort.

2° modo) Il successore è l'elemento che tra le chiavi più grandi ha la chiave più piccola $\Rightarrow$ è il minimo del sottoalbero destro. Se il sottoalbero destro è vuoto, si cerca il primo antenato che abbia come figlio sinistro il nodo stesso o un suo antenato.

Predecessor

1° modo) Si ordinano i valori nell'albero con l'operazione Sort.

2° modo) Il predecessore è l'elemento che tra le chiavi più piccole ha la chiave più grande $\Rightarrow$ è il massimo del sottoalbero sinistro. Se il sottoalbero sinistro è vuoto, si cerca il primo antenato che abbia come figlio destro il nodo stesso o un suo antenato.

Insert

Inserisce un nuovo elemento mantenendo le proprietà del BST. L'inserzione avviene sempre nelle foglie.

Se il BST è vuoto, crea un nuovo albero, altrimenti:

- inserzione ricorsiva: considera ricorsivamente terne L-Root-R, e a ogni passo effettua un confronto;
- inserzione iterativa: prima ricerca (search) la posizione in cui la chiave si dovrebbe trovare, quindi la inserisce in quella posizione.

Select

Dato un valore intero k, estrae/sceglie la $k+1$-esima chiave più piccola nel BST (con k che parte da 0). Dopo aver associato a ogni nodo il numero delle chiavi contenute nei sottoalberi radicati in esso,[1] a ogni terna L-Root-R determina se la chiave che si sta cercando può essere contenuta nel sottoalbero L in base al suo numero di chiavi associato t (per il sottoalbero sinistro vuoto: $t = 0$):

[1] Si calcola in questo modo:

- ogni foglia è costituita da 1 chiave;

- se $t = k$: termina l'operazione di select e ritorna Root;
- se $t > k$: scende nel sottoalbero L;
- se $t < k$: scende nel sottoalbero R, ricercando la chiave di ordine $k = k - t - 1$.

Complessità

Tutte queste operazioni, tranne la visita (sopra denominata "sort"), sui BST sono di complessità lineare[2] rispetto all'altezza dell'albero: $T(n) = O(h) \Rightarrow$ rispetto a n. Il BST è vantaggioso tanto più l'albero è bilanciato:

- caso migliore: $h = \log_2 n \Rightarrow T(n) = O(\log n)$ (albero completamente bilanciato)
- caso medio: $T(n) \simeq O(\log n)$
- caso peggiore: $h = n \Rightarrow T(n) = O(n)$ (albero completamente sbilanciato)

Effettuando tante operazioni su un BST bilanciato, il BST si potrebbe però sbilanciare:

1ª soluzione) ogni tanto si ribilancia il BST $\Rightarrow$ poco efficace, perché si aggiunge la complessità dell'operazione di ribilanciamento;

2ª soluzione) si costruisce un albero bilanciato per costruzione, applicando dei vincoli.

9.2.2 Operazioni di servizio

Rotate a destra/sinistra

Si scambia il rapporto padre-figlio tra due nodi x e y, posizionando opportunamente i sottoalberi di partenza dei nodi attraverso semplici spostamenti dei puntatori ai sottoalberi.

9.2.3 Operazioni avanzate

Inserimento alla radice

Inserisce la chiave con l'operazione Insert, quindi effettua delle rotazioni per spostare progressivamente la nuova chiave dal basso verso la radice.

Partition

Riorganizza l'albero intorno a una certa chiave di ordine k (Select), portandola alla radice (Rotate). Se applicata intorno alla chiave mediana, spesso permette di ribilanciare un BST.

Delete

- nodo senza figli (foglia): si può cancellare subito la foglia;
- nodo con 1 figlio: basta connettere il figlio del nodo con il padre del nodo;
- nodo con 2 figli: bisogna sostituirlo o con il suo predecessore o con il suo successore:
  - Precedessor/Successor: si ricerca il predecessore/successore in uno dei sottoalberi;
  - Partition: lo si fa galleggiare fino alla radice;
  - lo si connette con l'altro sottoalbero.

- procedendo dal basso verso l'alto (bottom-up), si somma a ogni passo le dimensioni dei sottoalberi e si aggiunge 1 per la radice.

[2] Ad esempio, nel caso peggiore i confronti dell'operazione di search vengono fatti fino in fondo all'albero.

Capitolo 10

Tipologie di problemi

- **problemi di ricerca:** ricerca di una soluzione valida (= che soddisfa certi criteri), se esistente (ho o non ho trovato la soluzione?):
  - es.: ciclo hamiltoniano:[1] dato un grafo non orientato, esiste un cammino semplice[2] chiuso (ciclico) che collega tutti i vertici?
- **problemi di ottimizzazione:** ricerca della soluzione ottima (= che minimizzi un determinato costo o massimizzi un determinato vantaggio):
  - es.: cammini minimi: data una rete di città modellata come un grafo, a ogni arco è associata una funzione distanza (peso). Si cerca di minimizzare la funzione distanza. Se il cammino minimo non esiste, è come se ci fosse un cammino di peso infinito;
- **problemi ibridi** (ricerca + ottimizzazione)**:** ricerca della soluzione valida minima. Non si conoscono tuttora algoritmi polinomiali per risolvere questo tipo di problemi ⇒ algoritmi euristici:
  - es.: commesso viaggiatore:[3] ricerca del cammino semplice chiuso minimo.

S spazio delle soluzioni (= insieme di tutte le possibilità) $\rightarrow$ V soluzioni valide $\rightarrow$ M soluzioni migliori

- problemi di ricerca: $S \subset V \Rightarrow$ verificare che $V \neq \varnothing$ e trovare una soluzione valida qualsiasi;
- problemi di ottimizzazione: $S = V$ (tutte le soluzioni sono valide) $\Rightarrow$ trovare la soluzione ottima;
- problemi ibridi: $S \subset V \Rightarrow$ trovare la soluzione di minimo costo/massimo vantaggio all'interno di S.

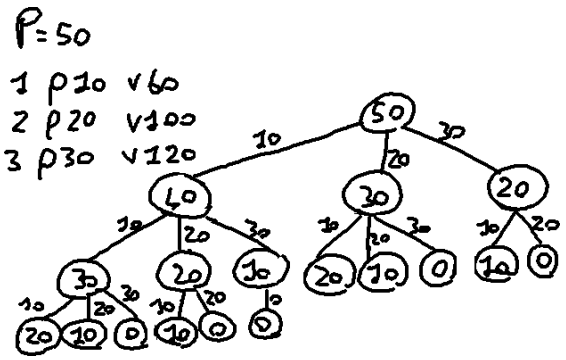
Trovare la soluzione ottima significa esaminare tutto lo spazio delle soluzioni ⇒ non è sempre possibile:

- **soluzione ottima globalmente:** massimo assoluto della funzione di ottimizzazione nell'intero dominio;
- **soluzione ottima localmente:** massimo relativo della funzione di ottimizzazione in un sottoinsieme del dominio ⇒ meno costo.

[1] Si veda la voce Ciclo hamiltoniano su Wikipedia in italiano.
[2] cammino semplice = che non passa mai due volte per lo stesso nodo (in questo caso vertice).
[3] Si veda la voce Problema del commesso viaggiatore su Wikipedia in italiano.

10.1 Soluzione ottima globalmente

10.1.1 Il problema dello zaino discreto (approccio divide et impera di tipo ricorsivo)



Un ladro con uno zaino piccolo di volume P deve rubare N oggetti appartenenti a un insieme S; l'oggetto j-esimo, di peso p_j e valore v_j, vale $x_j = 1$ se viene preso oppure $x_j = 0$ se lasciato. Il ladro deve minimizzare il volume occupato ($\sum_{j \in S} p_j x_j \leq P$) e massimizzare il valore complessivo ($\sum_{j \in S} v_j x_j = \text{MAX}$), verificando ogni volta la capacità residua disponibile, cioè il volume ancora libero. Il problema si dice **discreto** perché ogni oggetto è preso o lasciato, non si può spezzare in più parti. Le soluzioni possono essere enumerate esaustivamente tramite un albero, in modo da poterle esplorare tutte ricorsivamente. Si arriva a una foglia quando non è più possibile andare avanti (non è detto che lo zaino è pieno). L'obiettivo è trovare la foglia per cui il valore degli oggetti è massimo, tenendo presente che ci possono essere eventualmente più foglie con lo stesso valore. Questo approccio ha un costo elevato, ma la soluzione è ottima globalmente.

10.2 Il paradigma greedy

Si segue un solo cammino, scegliendo quella che appare di volta in volta la soluzione migliore, secondo una **funzione di appetibilità** (= che misura la convenienza locale) ⇒ non è detto che si trovi la soluzione ottima globalmente, però si riduce il costo. Nell'algoritmo greedy non c'è il backtrack, ossia non si ritorna mai sui passi precedenti. Il procedimento termina quando la configurazione non permette più l'esecuzione di alcuna scelta.

Tipi di appetibilità

- **appetibilità fisse:** le appetibilità sono costanti ⇒ una volta ordinate in un vettore, a ogni passo si sceglie l'appetibilità massima;
- **appetibilità modificabili:** a ogni passo si ricalcolano le appetibilità ⇒ le code a priorità permettono di mantenere l'ordinamento.

10.2.1 Esempi

Il cambiamonete

Il problema consiste nel trovare il resto con numero minimo di monete. A ogni passo, si sceglie la moneta di valore più grande (cioè avente appetibilità massima) tra quelle compatibili sul resto, quindi si considera il resto residuo. Le appetibilità sono fisse, perché dipendono unicamente dal sistema di monetazione corrente ⇒ se la monetazione è particolare, la soluzione data dall'algoritmo greedy può però essere non ottima.

Il problema dello zaino discreto (approccio greedy)

Si estrae un cammino dallo spazio di ricerca, cioè a ogni passo si segue un solo cammino. Questo approccio ha un costo inferiore, ma non è detto che si riesca a trovare la strada ottima globalmente.

Si cerca l'oggetto ad appetibilità massima (= massimo valore), con la limitazione che sia compatibile con la funzione volume.

Il problema dello zaino continuo

$0 \leq x_j \leq 1$ è una variabile reale che misura quanta parte di un oggetto (es. polvere d'oro) è stata presa.

Per ogni oggetto non si considera il valore complessivo, ma si considera il sottolinea|valore specifico: $\frac{v_j}{p_j} \Rightarrow$ soluzione migliore.

10.3 Codici

Come effettuare codifiche (da simbolo a parola di codice) e decodifiche (da parola di codice a simbolo)? Una serie di **parole di codice** si dice **codice**.

- **simboli isofrequenti:** la probabilità di trovare simboli è uniforme (c'è la stessa probabilità di trovare un simbolo piuttosto che un altro)
- **simboli non isofrequenti:** es. lingue (più vocaliche o più consonantiche)

Le parole di codice possono essere:

- a lunghezza fissa: dati $\operatorname{card}(S)$ simboli, sono possibili $n!$ parole di codice costituite da $n = \sup(\log_2 \operatorname{card}(S)) \in \mathbb{N}$ bit. La decodifica di un codice è più facile, perché basta suddividerlo nella unità di lunghezza;
- a lunghezza variabile: ogni parola di codice ha una lunghezza variabile $\Rightarrow$ la decodifica è difficoltosa, però si può usare per la compressione delle informazioni perché, assegnando meno bit ai simboli più frequenti, si può limitare il numero di bit necessari.

stringa di ordine k (**k-prefisso**) stringa contenente i primi k bit

Nel caso a lunghezza variabile, è necessario costruire un **codice (libero da) prefisso** per evitare le ambiguità, in modo che nessuna parola di codice sia a sua volta prefisso di un'altra parola di codice.

10.3.1 Codici di Huffman

I **codici di Huffman** sono buoni codici a lunghezza variabile. Come costruire un codice di Huffman tramite un albero binario:

1. i simboli si ordinano per frequenza crescente in una coda a priorità;
2. la coppia di simboli a frequenza meno elevata (costituita dal primo minimo e dal secondo minimo) si fonde in un **aggregato**;
3. si associa a un simbolo il bit 0 e all'altro simbolo il bit 1;
4. si assegna all'aggregato una frequenza pari alla somma delle loro frequenze;
5. l'aggregato si reinserisce nella coda a priorità;
6. si riparte al punto 1, considerando l'aggregato come un semplice simbolo ma con la nuova frequenza.

Condizione di terminazione: la coda a priorità è vuota (a ogni passo da due simboli si passa a un solo simbolo). È un algoritmo greedy perché ogni volta si scelgono le appetibilità massime (in questo caso: le frequenze minime).

Il costo è linearitmico perché si usano le operazioni sulle code a priorità: $T(n) = O(n \log n)$

Percorrendo questo albero binario, si possono effettuare le operazioni di codifica/decodifica.

Capitolo 11

Le tabelle di hash

La **tabella di hash** è un tipo di dato astratto che utilizza uno spazio di memoria congruente con il numero di dati, in cui si garantiscono operazioni prossime al costo unitario, ovvero con un tempo medio di accesso in ricerche/cancellazioni/inserzioni di tipo unitario.

- tabelle ad accesso diretto: la chiave è l'indice del vettore ⇒ complessità lineare, ma: non sono utilizzabili quando il numero delle chiavi è troppo grande e non si può allocare in memoria;
- strutture ad albero: complessità logaritmica, ma si spera che l'albero non degeneri;
- tabelle di hash: c'è una funzione intermedia detta **funzione di hash** che manipola la chiave e la trasforma in un indice.

Il numero $|K|$ delle chiavi contenute in una tabella di hash di dimensione M è molto minore della cardinalità dell'universo delle chiavi: $|K| \ll |U|$.

Se ogni chiave nell'universo ha un indirizzo compreso tra 0 e $m-1$, la funzione di hash $h : U \longrightarrow \{0, \cdots, m-1\}$ associa alla chiave k l'indice $h(k)$ della tabella T.

La tabella di hash deve garantire una buona distribuzione: se le chiavi sono equiprobabili allora i valori $h(k)$ devono essere equiprobabili, cioè la distribuzione deve essere **semplice uniforme** e non si deve polarizzare su determinate chiavi.

11.1 Tabelle di hash per valori numerici

11.1.1 Metodo moltiplicativo (chiavi k in virgola mobile)

Per una chiave k compresa nell'intervallo $[s, t]$:

$$h(k) = \sup\left(M\frac{k-s}{t-s}\right)$$

1. normalizzazione: si divide $k-s$ (offset tra la chiave k e l'estremo inferiore dell'intervallo) e $t-s$ (ampiezza dell'intervallo) ⇒ si ottiene un valore compreso tra 0 e 1;
2. si moltiplica per il numero M per trovare un valore compreso nell'intervallo tra 0 e $m-1$;
3. si prende l'intero superiore, poiché $h(k)$ è un numero intero.

11.1.2 Metodo modulare (chiavi k intere)

Si applica la funzione `mod()` a una chiave `k` intera:

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

Per limitare le collisioni, si prende per M un numero primo. Esempi antitetici di tabelle di hash sono la funzione $\mod 2^n$: concentra tutti i numeri in valori 0 e 1 $\Rightarrow$ tantissime collisioni perché si limita ai primi n bit più significativi.

11.1.3 Metodo moltiplicativo-modulare (chiavi k intere)

Data una costante A (per esempio: $A = \frac{\sqrt{5}-1}{2}$):

$$h(k) = \inf(k \cdot A) \bmod M$$

11.2 Tabelle di hash per stringhe

11.2.1 Metodo modulare per stringhe corte

Pensando ogni carattere come numero, la stringa è la rappresentazione compatta di un polinomio. Valutando il polinomio in un punto (di solito $\text{card}(U)$), si ottiene il valore numerico intero $k \Rightarrow$ si applica il metodo modulare per chiavi intere.

Le operazioni per valutare il polinomio non sono però efficienti $\Rightarrow$ questo metodo si può usare per stringhe corte.

11.2.2 Metodo modulare per stringhe lunghe

La valutazione del polinomio è effettuata tramite il **metodo di Horner**, che considera ricorsivamente polinomi di grado 1:

$$p_n(x) = \{[(a_n x + a_{n-1})\, x + a_{n-2}]\, x + \cdots + a_1\}\, x + a_0$$

La base x del polinomio può essere un numero primo o un numero pseudocasuale `a = (a * b) mod (M - 1)` (**hash universale**). Per calcolare la chiave k, per ogni polinomio di primo grado valutato si deve fare a ogni passo la funzione `mod M`.

11.3 Gestione delle collisioni

È inevitabile il fenomeno della **collisione**, ovvero quando chiavi diverse corrispondono allo stesso valore di indice; si può ridurre con buone funzioni di hash.

Si ha una collisione se:

$$k_i \neq k_j \longrightarrow h(k_i) = h(k_j)$$

11.3.1 Linear chaining

Ogni elemento della tabella contiene un puntatore alla testa di una lista concatenata che contiene tutte le chiavi collidenti.

Operazioni sulla lista

- inserzione: non si hanno esigenze di ordinamento $\Rightarrow$ l'inserzione meno costosa è quella in testa: $T(n) = O(1)$;
- ricerca: se $N = |K|$ = numero delle chiavi, M = dimensione della tabella di hash:
  - caso peggiore: tutte le chiavi si concentrano in una sola lista $\Rightarrow$ operazione lineare nella dimensione della lista, ma: caso poco frequente: $\Theta(N)$;
  - caso medio: le chiavi sono tutte uniformemente distribuite $\Rightarrow$ la dimensione media delle liste è uguale al fattore di carico $\alpha = \frac{N}{M}$: $T(n) = O(1+\alpha)$ (un buon valore di α è 0,1);

- cancellazione:
  - se la lista è doppio-linkata: $T(n) = O(1)$;
  - altrimenti, il costo è legato alla ricerca dell'elemento.

11.3.2 Open addressing

Ogni cella può contenere al massimo un solo elemento ($\alpha \leq 1$), e in caso di collisione il valore da inserire va inserito in un'altra cella, verificando che sia vuota (**probing** = ricerca di una cella vuota). Le $M-1$ celle rimanenti vengono sondate nell'ordine di una delle $(M-1)!$ permutazioni stabilita dalla funzione di hash, in funzione anche del tentativo t: $h(k,t) : U \times \{0, \cdots, M-1\} \longrightarrow \{0, \cdots, M-1\}$.

Linear probing

- inserzione: si sonda la cella di indice successivo e si inserisce nella prima casella vuota;
- ricerca: si passa di volta in volta alla cella di indice successivo, ricordando che al termine della tabella si deve passare alla prima cella (`mod M` a ogni incremento). È una gestione implicita delle liste all'interno di un vettore senza ricorrere al concetto di puntatore;
- cancellazione: bisogna distinguere le celle vuote dalle celle svuotate dalla cancellazione, perché altrimenti la ricerca successiva si fermerebbe alla cella svuotata.

Sperimentalmente, tentativi in media di probing per la ricerca:

- search miss: $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
- search hit: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

Se $\alpha \sim 0^+ \Rightarrow$ search miss $\approx 1 \Rightarrow$ efficiente.

Double hashing

L'indice della cella successiva in caso di probing con insuccesso viene calcolato da un'altra funzione di hash h_2.

Sperimentalmente, tentativi in media di probing per la ricerca:

- search miss: $\frac{1}{1-\alpha}$
- search hit: $\frac{1}{\alpha} \log \frac{1}{1-\alpha}$

Capitolo 12

L'ADT grafo non orientato

12.1 Matrice di adiacenza

La matrice di adiacenza A è una matrice quadrata $|V| \times |V|$, in cui ogni cella di indice ij contiene 1 o 0 a seconda se l'elemento i è connesso o no all'elemento j.

Vantaggi/svantaggi

- svantaggio: se il grafo non è orientato, la matrice risulta simmetrica rispetto alla diagonale principale;
- vantaggio: se il grafo è pesato, il valore di peso si può inserire direttamente nella matrice al posto di 1 $\Rightarrow$ un numero diverso da 0 determina sia l'esistenza dell'arco, sia il valore di peso;
- la complessità spaziale è: $S(n) = \Theta\left(|V|^2\right) \Rightarrow$ vantaggioso per grafi densi;
- vantaggio: per verificare una connessione basta accedere ad una cella della matrice $\Rightarrow$ costo unitario.

12.2 Lista di adiacenza

Ogni cella di indice i di un vettore contiene una lista di tutti gli elementi adiacenti all'elemento i-esimo.

Vantaggi/svantaggi

- svantaggio: in un grafo pesato, si deve memorizzare anche il peso sotto forma di denominatore;
- svantaggio: gli elementi complessivi nelle liste sono $2|E|$;
- $S(n) = O(\max(|V|, |E|)) = O(|V| + |E|) \Rightarrow$ è vantaggioso per i grafi sparsi;
- svantaggio: l'accesso topologico è meno efficiente perché non ha costo unitario.

12.3 Generazione di grafi casuali

1. Il grafo non è modello della realtà, ma viene generato da casuali coppie di vertici $\Rightarrow$ eventuali archi duplicati e cappi da eliminare.

2. Calcolo probabilistico che nasce da un grafo completo: Tra tutti i possibili $\frac{V(V-1)}{2}$ archi del grafo completo, si considerano solo gli archi di probabilità inferiore a un valore soglia di probabilità specificato:

$$E = p\frac{V(V-1)}{2} \Rightarrow p = 2\frac{E}{V(V-1)} \in [0,1]$$

Vantaggioso perché non si considerano duplicati e cappi, e se vengono richiesti E archi si ottengono in media E archi.

12.4 Applicazioni

I grafi si possono usare per:

- mappe: ricerca del cammino minimo tra due città;
- ipertesti: documenti = nodi, collegamenti = archi;
- circuiti.
- scheduling: ordinamento per tempi di esecuzione dei compiti vincolati (es. analisi I - analisi II - metodi matematici);
- matching: esecuzione di compiti in parallelo su più risorse;
- reti: mantenimento delle condizioni affinché una rete rimanga connessa.

Si considerano solo problemi trattabili, che abbiano cioè complessità polinomiale.

12.5 Problemi non trattabili

12.5.1 Problema della colorabilità

In una cartina geografica, quanti colori servono al minimo affinché ogni Stato sia colorato con colori diversi da quelli degli Stati adiacenti ad esso?

12.5.2 Ricerca di un cammino semplice

Esiste un cammino semplice che connette due nodi? Partendo da un nodo, passo da nodi adiacenti non ancora visitati finché non arrivo all'altro nodo. È un algoritmo di tipo ricorsivo, perché quando si arriva con insuccesso a un punto finale di un cammino semplice, si ritorna a esplorare le altre possibilità. Con un vettore si tiene conto dei nodi già visitati.

12.5.3 Ricerca di un cammino di Hamilton

Esiste un cammino semplice che connette due nodi e che tocca tutti i nodi del grafo una sola volta?

L'algoritmo di ricerca opera in maniera analoga al cammino semplice, aggiungendo il vincolo che il cammino da ricercare deve avere lunghezza $|V| - 1$. Durante il backtrack nel caso di un cammino non nella lunghezza desiderata, bisogna resettare il vettore che tiene conto dei nodi già visitati. La verifica di un cammino di Hamilton è polinomiale, ma la ricerca è di complessità esponenziale per colpa del backtrack.

12.5.4 Ricerca di un cammino di Eulero

Königsberg è una città costruita lungo un fiume, in cui vi è una serie di isole collegate alle due rive da ponti. Le isole/rive sono i nodi del grafo, e i ponti sono gli archi ⇒ **multigrafo**: ci sono archi duplicati in un grafo non orientato, con informazioni diverse non ridondanti. Si vogliono attraversare tutti i ponti una sola volta.

Il cammino di Eulero è il cammino, non necessariamente semplice, che attraversa tutti gli archi una sola volta. chiuso ⇒ ciclo di Eulero

Lemmi

- Un grafo non orientato ha un ciclo di Eulero se e solo se è connesso e tutti i suoi vertici sono di grado pari.
- Un grafo non orientato ha un cammino di Eulero se e solo se è connesso e se esattamente due vertici hanno grado dispari.

Capitolo 13

Gli algoritmi di visita dei grafi

La visita di un grafo consiste nell'elencare tutti i vertici del grafo a partire da un vertice dato secondo una determinata strategia, elencando le informazioni memorizzate nel grafo.

13.1 Visita in profondità (DFS)

Si segue un cammino finché esso non si interrompe per qualche ragione, e si ritorna sulle scelte fatte. Tutti i vertici vengono visitati indipendentemente dal fatto che siano tutti raggiungibili o meno dal vertice di partenza. Si esplora una foresta di alberi estratta dal grafo di partenza: ogni nodo del grafo appartiene a uno e un solo albero della foresta; solo gli archi di tipo **Tree** appartengono alla foresta, mentre gli altri archi si dicono Backward o, nel caso del grafo orientato, Forward e Cross.

13.1.1 Tempo

Si usa una linea del tempo discreta: a intervalli regolari esistono istanti in corrispondenza biunivoca con numeri naturali, e il tempo esiste soltanto in quegli istanti discretizzati. Sotto certe condizioni, il tempo passa discretamente dall'istante t all'istante $t + 1$. Ogni vertice si etichetta con un **tempo di scoperta** (il vertice è stato incontrato per la prima volta) e un **tempo di fine elaborazione** (il nodo non ha più informazioni da fornire).

13.1.2 Passi

1. si parte da un vertice, e si verifica se esistono vertici adiacenti (ovvero se è connesso ad altri vertici da archi);
2. i vertici adiacenti possono essere o ancora da scoprire, o già scoperti, o già terminati come elaborazione;
3. secondo un criterio fisso, si sceglie il nodo su cui scendere (ad esempio: nodo più a sinistra, ordine lessicografico) tra quelli non ancora scoperti;
4. si opera ricorsivamente sui nodi figlio, fino a quando non esistono più vertici adiacenti $\Rightarrow$ sono stati visitati tutti i nodi.

Per un nodo la fine elaborazione corrisponde all'uscita del nodo dalla ricorsione.

Convenzione colori: non ancora scoperto = bianco | scoperto ma non completato = grigio | scoperto e completato = nero

13.1.3 Classificazione degli archi

Dopo aver individuato gli archi di tipo Tree e riorganizzato il grafo di partenza come foresta di alberi di visita in profondità, vi si aggiungono gli archi rimanenti e li si classifica confrontando i tempi di scoperta e di fine elaborazione dei nodi su cui ciascun arco insiste:

- **Backward**: connette un nodo u a un suo antenato v (tempo di scoperta $v < u$ e tempo di fine elaborazione $v > u$);
- **Forward** (solo nei grafi orientati): connette un nodo a un suo discendente (tempo di scoperta $v > u$ e tempo di fine elaborazione $v < u$);
- **Cross**: archi rimanenti (solo nei grafi orientati).

13.1.4 Analisi di complessità

Lista delle adiacenze

$$T(n) = \Theta(|V| + |E|)$$

- inizializzazione: legata al numero dei vertici ($\Theta(|V|)$)
- visita ricorsiva: legata al numero di archi ($\Theta(|E|)$)

Matrice delle adiacenze

$$T(n) = \Theta\left(|V|^2\right)$$

13.2 Visita in ampiezza (BFS)

La visita in ampiezza opera in parallelo su tutti i nodi correnti. Non necessariamente vengono visitati tutti i vertici del grafo, ma solo i vertici raggiungibili dal vertice di partenza; non c'è più una foresta di alberi, ma c'è un unico albero. Si applica a un grafo non pesato, che è equivalente a un grafo pesato in cui ogni arco ha egual peso.

A ciascun vertice corrente si associa la distanza minima dal vertice di partenza, ma i pesi sono tutti uguali $\Rightarrow$ è un caso particolare della ricerca dei cammini minimi.

13.2.1 Passi

Bisogna ricondurre il lavoro in parallelo in un modello seriale. A ogni passo, la stima della distanza minima viene ricalcolata.

1. In primo luogo, si assume per ogni nodo la condizione "non esiste cammino" $\Rightarrow$ si associa la massima distanza concepibile $+\infty$.
2. Si scende sui vertici adiacenti v e w, si riesegue la stima della distanza, quindi li si inserisce in una coda FIFO.
3. Si riesegue il passo 2. per i vertici adiacenti del nodo v, che nella coda verranno inseriti dopo w, e per quelli di w, inseriti alla fine $\Rightarrow$ verranno processati prima i nodi adiacenti di w, poi i nodi adiacenti di v, quindi w e infine v, come se l'albero risultante venisse esplorato da destra verso sinistra livello per livello.

13.2.2 Analisi di complessità

Lista delle adiacenze

$$T(n) = \Theta(|V| + |E|)$$

Matrice delle adiacenze

$$T(n) = \Theta\left(|V|^2\right)$$

Capitolo 14

Le applicazioni degli algoritmi di visita dei grafi

14.1 Grafo aciclico

Un grafo è ciclico se ha almeno un arco di tipo Backward.

14.2 Componenti connesse **(per grafi non orientati)**

Individuando le componenti connesse (= insieme massimale di tutti i vertici raggiungibili) degli alberi della visita in profondità, si può verificare se esiste almeno un cammino tra due nodi dati.

14.3 Connettività **(per grafi che rappresentano una rete)**

14.3.1 Bridge **(= arco la cui rimozione disconnette il grafo)**

Un arco Tree è un bridge se non esiste nessun arco Backward che collega un qualsiasi discendente a un qualsiasi antenato nell'albero della visita in profondità. Solo gli archi Tree possono essere dei bridge, perché l'esistenza di un arco Backward implica l'esistenza di un altro cammino che collega i due nodi.

14.3.2 Punti di articolazione **(= vertice la cui rimozione disconnette il grafo)**

Si distinguono due casi:

- vertice radice del grafo: è un punto di articolazione se ha almeno due figli nell'albero della visita in profondità;
- altro vertice: è un punto di articolazione se almeno un sottoalbero figlio di quel nodo non ha un arco Backward che lo collega a un nodo antenato.

14.4 DAG

I DAG sono grafi orientati privi di cicli che rappresentano dei modelli di ordine parziale, dove gli archi sono dei vincoli di precedenza di esecuzione dei compiti: un compito è eseguibile solamente dopo il completamento dei precedenti, e così via ⇒ scheduling: azioni da svolgere in un certo ordine.

L'ordinamento topologico di un DAG è la riscrittura del DAG in una linea di vertici con il vincolo che tutti gli archi vadano da sinistra a destra o viceversa (inverso). Per effettuare l'ordinamento topologico, si ordinano i nodi per tempi di fine elaborazione di una visita in

profondità, quindi si disegnano tutti gli archi secondo una visita topologica diretta/inversa del grafo.

14.5 Componenti fortemente connesse (per grafi orientati)

grafo trasposto grafo con gli stessi vertici ma con gli archi di direzione inversa

Per trovare le componenti fortemente connesse si usa l'**algoritmo di Kosaraju**:

1. si traspone il grafo;
2. si esegue la visita in profondità sul grafo trasposto, calcolando i tempi di scoperta e di fine elaborazione;
3. eseguendo la visita in profondità sul grafo originale per quei tempi di fine elaborazione decrescenti, si trovano man mano le componenti fortemente connesse;
4. s'interpreta il grafo in classi di equivalenza (= componenti fortemente connesse) secondo la proprietà di mutua raggiungibilità, cioè si ricostruisce un grafo semplificato considerando ogni componente fortemente connessa come un unico nodo rappresentativo e considerando solo gli archi che connettono componenti fortemente connesse.

Capitolo 15

Gli alberi ricoprenti minimi

Il grafo ricoprente minimo è un sottoinsieme non unico di un generico grafo avente stessi vertici e un sottoinsieme di archi, in cui tutti i vertici vengono coperti in modo che la somma dei costi degli archi utilizzati sia la minore possibile. Un grafo ricoprente minimo è sempre aciclico (viene scelto solo uno dei cammini che compongono il ciclo) $\Rightarrow$ è un **albero ricoprente minimo**.

Si può rappresentare come lista di archi, come lista di adiacenza o come vettore dei padri.

Vettore dei padri Anziché per ogni radice puntare ai due figli come nel BST, per ogni figlio si punta al padre (`st` è il vettore dei padri) perché il padre è uno, i figli sono in varie quantità.

Gli algoritmi di Kruskal e Prim, partendo da un sottoinsieme di albero ricoprente minimo avente nessun arco, costruiscono arco per arco l'albero ricoprente minimo seguendo la proprietà di **invarianza**, cioè garantendo che l'arco via via inserito sia sicuro e mantenga la condizione di albero ricoprente minimo. Nonostante questi algoritmi siano di tipo greedy, la soluzione trovata è comunque quella ottima globalmente perché l'arco è garantito essere sicuro.

Il **taglio** è la partizione del grafo V in due insiemi S e $V - S$. La differenza tra gli algoritmi di Kruskal e di Prim è la scelta del taglio.

Dato un taglio che **rispetti** l'insieme degli archi corrente (cioè nessun arco attraversa il taglio), ogni **arco sicuro** deve essere l'**arco leggero** (cioè quello di peso minimo) che attraversa il taglio (cioè collega un nodo di S con un nodo di $V - S$).

Un arco sicuro collega due alberi non ancora connessi.

15.1 Algoritmo di Kruskal

A ogni passo si prendono gli archi sicuri tra quelli rimasti nell'intero grafo.

All'inizio, ciascun nodo è un albero composto da un nodo.

Quando gli archi presi sono multipli, nell'algoritmo corrispondono a iterazioni multiple (non vengono presi insieme).

La complessità $T(n) = |E| \log |E|$ è data sia se ordino prima tutti gli archi per peso decrescente, sia se ogni volta cerco l'arco di peso minimo.

15.2 Algoritmo di Prim

Partendo da un nodo, a ogni passo si prende il solo arco sicuro tra quelli che partono dal nodo corrente (in caso di due archi sicuri di egual peso, è arbitraria la scelta di uno di essi).

Man mano che si aggiunge un arco sicuro, nel vettore dei padri si memorizza come padre il nodo da cui l'arco sicuro parte.

La complessità è: $T(n) = O(|E| \log |V|)$

Il caso peggiore si verifica se il numero di archi $|E|$ è molto maggiore del numero di vertici $|V|$.

Capitolo 16

I cammini minimi

Si considerano grafi orientati e pesati. Il **peso** w **di un cammino** è la sommatoria dei pesi associati agli archi che compongono il cammino. Il **peso minimo** σ **di un cammino** tra u e v è uguale al minimo peso di tutti i cammini tra u e v se almeno uno esiste, altrimenti il peso è $+\infty$. Più cammini possono essere tutti di peso minimo, ma il peso minimo è univoco.

16.1 Applicazioni

Si applica per esempio a reti di città con pesi corrispondenti alle distanze.

Casi

- da sorgente singola: a partire da una città si calcolano i cammini minimi verso tutte le altre destinazioni;
- con destinazione singola: viceversa $\Rightarrow$ basta risolvere con lo stesso algoritmo lavorando sul grafo trasposto;
- tra tutte le coppie di vertici: si calcolano i cammini minimi tra tutte le coppie di vertici (matrice triangolare) $\Rightarrow$ basta iterare per tutti i vertici l'algoritmo con una sorgente singola.

16.2 Grafi con archi a peso negativo

Si distinguono due casi:

1. nessun arco (di peso negativo) appartiene a un ciclo con somma dei pesi negativa:
   - algoritmo di Dijkstra: non garantisce la soluzione ottima (algoritmo greedy);
   - algoritmo di Bellman-Ford: garantisce comunque la soluzione ottima;
2. esiste almeno un arco (di peso negativo) appartenente a un ciclo con somma dei pesi negativa:[1]
   - algoritmo di Dijkstra: il risultato non ha senso perché non ha senso il problema, e non rileva neanche il ciclo a peso negativo;
   - algoritmo di Bellman-Ford: è in grado di rilevare il ciclo a peso negativo.

[1] In realtà non ha neanche senso parlare di ricerca di cammini minimi perché percorrendo il ciclo si arriva a una stima $-\infty$.

16.3 Rappresentazione

- vettore dei predecessori: per il nodo i-esimo riporta l'indice del padre se esiste, altrimenti -1;
- sottografo dei predecessori $G_\pi\left(V_\pi, E_\pi\right)$: V_π contiene tutti i vertici per cui esiste il padre + il vertice di partenza, E_π contiene gli archi che connettono tutte le coppie di vertici padre-figlio di V_π tranne il vertice di partenza;
- albero dei cammini minimi $G' = (V', E')$: V' contiene tutti i vertici raggiungibili dalla radice s dell'albero $\Rightarrow$ siccome è un albero, esiste un solo cammino semplice che connette ogni vertice con la radice:
  - se il grafo non è pesato (o di pesi unitari), basta fare una visita in ampiezza;
  - se il grafo è pesato, durante la visita del grafo si utilizza una coda a priorità per rappresentare i pesi.

16.4 Relaxation

16.4.1 Fondamenti teorici

Posti due vertici v_i e v_j connessi da uno o più cammini per cui esiste un cammino minimo, si può prendere un qualsiasi sottocammino di quel cammino minimo. Se quel sottocammino non fosse ottimo, esisterebbe un altro sottocammino ottimo che abbasserebbe la stima del cammino complessivo $\Rightarrow$ non è possibile $\Rightarrow$ ogni sottocammino di un cammino minimo è minimo.

Un cammino ottimo da v_i a v_j è esprimibile attraverso il sottocammino minimo da v_i a v_{j-1} + l'arco da v_{j-1} a v_j (non si trattano multigrafi $\Rightarrow$ non si hanno più archi tra due stessi vertici).

16.4.2 Procedimento

Usando un vettore dei pesi `wt`, inizialmente la radice s dista 0 da se stessa, e tutti i vertici distano $+\infty$. A ogni passo si effettua la **relaxation**: si confronta con la stima precedente il peso del sottocammino minimo fino a $j-1$ + il peso dell'arco $\Rightarrow$ se la stima è migliore si prende la nuova distanza minima stimata, fino a quando non si raggiunge la distanza minima definitiva, poiché una relaxation effettuata su un cammino già minimo non ha più effetto.

L'algoritmo di Dijkstra applica una sola relaxation per ogni arco, e usa la coda a priorità; l'algoritmo di Bellman-Ford applica tante relaxation per arco quanti sono i vertici, e prima deve ordinare tutti i vertici.

16.5 Algoritmo di Dijkstra

A ogni passo si considerano in maniera greedy i vertici non ancora stimati (= cioè appartenenti a $V - S$) raggiungibili dal vertice corrente, su cui poter applicare la relaxation. Usa una coda a priorità, dove la priorità corrisponde alla distanza minima correntemente stimata, in cui ogni vertice viene estratto una sola volta e non ristimando mai più di una volta una stima. Condizione di terminazione: la coda a priorità è vuota.

Complessità

$$T(n) = O\left(\left(|V| + |E|\right) \log |V|\right)$$

Se esiste un arco negativo, esso consentirebbe di ristimare la stima su un vertice già precedentemente stimato, ma quel vertice non entrerà mai più nella coda a priorità.

16.6 Algoritmo di Bellman-Ford

1. si ordinano gli archi attraverso un criterio di ordinamento stabilito;
2. per ogni vertice si applicano $|V| - 1$ relaxation;
3. alla fine si verifica che per tutte le $|V|$-esime relaxation non migliorino alcuna stima, perché altrimenti esiste almeno un ciclo negativo.