Appunti Strutture Dati e Algoritmi

Elisa Stabilini

I Semestre A.A. 22/23

Contents

Chapter 1

Lezione 0: Introduzione al corso

Idea: per costruire algoritmi ho bisogno di *tecniche* per immaginare la struttura degli algoritmi. Lo scopo è quello di capire qual è la **tecnica di costruzione** che sta alle spalle di un dato algoritmo; quindi capire perché quell'algoritmo mi dà una soluzione.

1.1 Practical Info

Per il corso NON serve utilizzare un linguaggio specifico, il corso in sé non è avanzato nella programmazione (nella forma) ma nei contenuti, mi concentro sugli strumenti teorici per poter costruire algoritmi in contesti diversi.

1.1.1 Esame finale

Presentazione scientifica o seminario - presentazione di circa 35 minuti su un argomento che ha carattere algoritmico, argomento ancorato bene alle cose del corso e concordato preventivamente con i docenti (argomento a piacere). NB: la presentazione deve essere **auto-contenuta**.

Oltre alla presentazione, che sarà di carattere algoritmico bisognerà consegnare anche una implementazione software in un linguaggio a scelta (si possono fare anche confronti con altri algoritmi ecc.).

Oltre a questo ci sarà un breve orale sul corso di SDA.

1.2 Algoritmi

Definizione. Un algoritmo è una sequenza *finita* di passi, *effettivi* e *non ambigui* che determina un processo che *termina*.

Gli algoritmi hanno la richiesta di essere procedimenti che portino *sempre* a una conclusione, se non portasse a una conclusione e andasse in loop sarebbe semplicemente una tecnica di calcolo, può essere utile ma non è un algoritmo.

Ciascuna delle istruzioni che viene fornita deve essere

- **effettiva**: deve essere qualcosa di concreto che mi porta a un risultato.
- **non ambigua**: l'algoritmo deve essere ben specificato in ogni suo passo, ogni step deve essere assolutamente chiaro, non lascia "scelta" a me programmatore.

Questa sequenza di passi con queste caratteristiche prende dell'informazione in *input* e genera dell'informazione in *output*, quindi in un tempo finito avrò delle informazioni "nuove", quindi un algoritmo è qualcosa che genera una **corrispondenza** tra dati in ingresso e dati in uscita.

Gli algoritmi servono per risolvere problemi, in particolare i dati che fornisco in input sono i dati su cui il problema si sta interrogando, una *rappresentazione* che viene data in input darà un output. L'insieme di dati di output è la risposta (soluzione del problema) alla domanda che io ho fatto *relativamente ai dati che ho dato in input*

Problemi Solitamente ai problemi si dà un nome, si specifica una *istanza generica*, ovvero uno statement che mi dice di che cosa si occupa il problema, quali sono gli oggetti su cui mi sto ponendo delle domande (fissa gli oggetti su cui lavora l'algoritmo eg. dimensione matrice). Ovviamente un problema pone una **domanda**, la domanda a cui sto cercando di rispondere.

Esempio: Problema di ordinamento

- Istanza generica: insieme (generalmente array) di n numeri. NB: non sto specificando quali sono i numeri, sto solo dicendo che ho n numeri
- Output: permutazione degli n numeri che rispetti la condizione di ordinamento

Allora per questo problema di ordinamento scrivo un algoritmo che risolva il problema (*algoritmo di sorting*).

Di volta in volta poi viene fornita l'**istanza particolare** del problema che devo affrontare. Un algoritmo è sostanzialmente una **trasformazione che è coerente con la richiesta del mio problema**

1.2.1 Correttezza dell'algoritmo

Definizione. Un algoritmo si dice corretto se per ogni istanza specifica la risposta fornita è corretta, deve essere giusto per ogni possibile input.

Esistono degli strumenti che cercano di aiutarmi a dimostrare la correttezza o meno dell'algoritmo; tuttavia non di tutti gli algoritmi è possibile verificare la correttezza, quando non è possibile dimostrare la correttezza faccio solo dei "beta-testing", se avessi degli strumenti per verificare immediatamente la correttezza di un software non avrei bisogno del beta-testing.
Negli algoritmi di sorting la correttezza può essere verificata abbastanza facilmente.

Definizione. Un programma è semplicemente la formalizzazione di un algoritmo utilizzando un determinato linguaggio

Tutti gli algoritmi fanno capo ad una serie di tecniche conoscendo le quali possono essere costruiti algoritmi funzionanti in vari contesti.

Grafi Un grafo è un insieme di punti chiamati *vertici*, o nodi, del grafo collegati tra loro mediante *archi*. Un grafo è uno strumento di astrazione che si usa ovunque, può rappresentare qualunque cosa, ad esempio può rappresentare una rete stradale.

Esempio: voglio trovare il percorso *migliore* che colleghi due punti (due punti di una rete stradale quindi posso dire due punti di un grafo). In questo contesto migliore può significare varie cose, sono molte le funzioni costo disponibili che posso voler ottimizzare). L'algoritmo più semplice prevede che generi tutte le strade possibili e poi scelga il minimo, in questo caso però sono tante le cose che devo scartare. Algoritmi di questo tipo sono detti **esaustivi** perché considero tutte le possibilità (anche detti algoritmi *brute force*). Questo algoritmo è corretto ma è impraticabile.
In generale, se non ho una strategia quello che posso fare sempre è usare un algoritmo *brute force*.

Da un punto di vista combinatorio sono molti i problemi che hanno moltissimi tentativi che si dovrebbero fare per trovare la soluzione giusta, che hanno moltissime candidate soluzioni (moltissime nel senso che è impraticabile vederle tutte). Quindi servono proprio degli algoritmi sofisticati, gli algoritmi che vedremo sono quelli che evitano questa complessità ovvero evitano i metodi forza bruta. Esistono problemi che hanno tantissime soluzioni, per cui esplorarle tutte sarebbe impraticabile ma purtroppo finora non abbiamo algoritmi efficienti per la ricerca di una soluzione.

Esistono anche i problemi così detti **irresolubili** per i quali non sarà mai possibile scrivere programmi per le soluzioni (si può dimostrare che non si possono fare), esempio la correttezza, la terminazione di un programma non si possono testare.

1.3 Strutture dati

Il motivo per cui è importante studiare anche le strutture dati è che l'unione di algoritmi e di strutture dati costituisce i **programmi**. In particolare un punto fondamentale della costruzione di buoni algoritmi è la capacità di manipolare efficacemente l'informazione. Infatti gli algoritmi passano anche attraverso l'utilizzo di *strutture dati*, molto spesso la scelta di una determinata struttura dati influenza la *prestazione* di un algoritmo, alcuni algoritmi hanno veramente dei salti di prestazione cambiando la struttura dati. Quindi ho bisogno di buone strutture dati per il supporto e per l'elaborazione dei dati all'interno di un algoritmo.

Esempio numerico Suppongo di avere a disposizione due computer con le seguenti caratteristiche:

- Computer A (veloce): 10 miliardi di istruzioni/s (10 GHz)
- Computer B (lento): 10 milioni di istruzioni/s (10 MHz)

Ho a disposizione i seguenti due algoritmi caratterizzati dalla seguente complessità in tempo:

- Indetion sort $2n^2$
- Merge sort $5n \log n$

Combinando le prime due opzioni ottengo un tempo di 5.5 ore, mentre combinando le seconde due ottengo un tempo di 20 min.

Esempi di strutture dati

- Stack: è una pila, appoggio un dato sull'altro (memorizzare on top e poi leggere il top dello stack).
- Coda: se eccede la fine vado all'inizio (?)
- Alberi: simili a pile bidimensionali (per cui poi posso muovermi non solo in una direzione ma in due - due per ogni "cella").

Definizione. Strutture dati: metodo di organizzazione dei dati

Alberi Le strutture ad albero sono strutture di memorizzazione in cui ho uno spazio per il dato e due spazi occupati da vettori che puntano ad una nuova "sotto-struttura". Le strutture ad albero soddisfano la seguente proprietà: ciascun elemento è sicuramente più grande del figlio sinistro e più piccolo del figlio destro (importante per la ricerca ad alberi). Si può dimostrare che l'altezza dell'albero è logaritmica rispetto al tempo medio di ricerca.

Algoritmo di ricerca: Dato che ad ogni suddivisione il numero di oggetti tra cui devo cercare alla k-esima query dimezzando. L'algoritmo di ricerca (*bisezione*) avrà la seguente dipendenza dal numero di caratteri su cui deve eseguire la ricerca

$$\frac{n}{2^k} = 1 \rightarrow k = \log n \tag{1.1}$$

1.4 Perchè studiare gli *algoritmi*

- Devono essere *mantenibili* e *adattabili*: se l'ho scritto bene, la filosofia era buona allora posso "riciclarlo", non solo su un altro problema ma anche su un altro hardware che lavora in modo diverso
- Ho bisogno di avere strumenti che mi consentano di verificare la correttezza degli algoritmi e in teoria degli algoritmi vengono forniti strumenti per verificare la correttezza
- Al pari dell'avanzamento della tecnologia hardware è necessario l'avanzamento della tecnologia software, se il software è scadente non riuscirei comunque a lavorare bene.

Chapter 2

Lezione 1: Algoritmi: Progettazione e Strumenti di analisi

2.1 Un primo esempio ben noto: sorting

Gli aspetti più importanti nella costruzione degli algoritmi sono:

- progettazione
- valutazione correttezza
- valutazione dell'efficienza

Il motivo per cui il problema di sorting è l'esempio migliore su cui iniziare ad affrontare questi temi è che è un problema di cui ora si sa tutto, quanto bravi possiamo essere, esattamente la richiesta di risorse computazionali, è un problema semplice ma istruttivo. Vedremo per questo algoritmo due *tecniche* di programmazione diverse, che portano con se due modi diversi di valutare la *complessità* e la *correttezza* dell'algoritmo. Conoscere i problemi di sorting è utile anche perché ovunque le informazioni vengono immagazzinate in una struttura ordinata, perché ad esempio, in una struttura ordinata è più facile una ricerca di un oggetto; quindi l'ordinamento è fondamentale. Inoltre, quello dell'ordinamento è stato uno dei primi problemi perché originariamente i programmi erano realizzati su delle schede forate e c'era bisogno di sapere come ordinarle. Alla base dello studio del sorting si avevano quindi dei motivi molto precisi e pratici. Per avere un'idea di quanto fosse importante basti pensare che a metà degli anni '90 Knut ha scritto *The art of computer programming*, serie di libri sulla programmazione di cui uno interamente dedicato al sorting.

2.1.1 Il problema

Ho in ingresso una n-upla di *oggetti* ordinabili (quindi deve esserci presente anche una relazione di ordinamento **lineare**). Per semplicità di trattazione poniamo che questa sia una n-upla di interi. Voglio *ridisporli* in modo tale che sia in ordine crescente (anche in questo caso scelgo un ordine crescente per semplicità di trattazione e senza perdita di generalità), quindi l'output sarà un vettore di interi ordinato.

Suppongo che il numero di oggetti sia tale da **essere mantenuto nella memoria centrale**, allora posso immaginare che siano contenuti in un *array* di interi. Quindi il problema diventa quello dell'ordinamento di un array (se avessi bisogno di memoria di massa dovrei fare un ordinamento esterno che richiede un processo diverso).

2.1.2 Progettazione algoritmo

Pseudo-codice

Per prima cosa ho bisogno di un modo per *specificare gli algoritmi.* Mi serve uno strumento di programmazione di algoritmi che non sia un linguaggio naturale (troppo ambiguo) ma che non sia necessariamente rigoroso quanto un linguaggio con la sua sintassi. Quello che si utilizza è uno **pseudo-codice** una sorta di linguaggio in cui non ho particolari regole di sintassi (sintassi lasca) che però somigli ad un linguaggio di programmazione. Questo pseudo linguaggio mi consente di concentrarmi sul problema senza preoccuparmi di come lo dovrò scrivere, mi solleva da qualunque dettaglio legato alla sintassi e alla gestione della memoria. In uno psuedo-linguaggio scrivo gli array come

$$A[1..n], \tag{2.1}$$

, quindi A è un array di n variabili intere accessibili mediante il loro indirizzo (inizio l'array da 1 ma se serve posso anche farlo cominciare da 0). Sintassi per accedere all i-esima componente:

$$A[i], \tag{2.2}$$

per indica re solo una porzione di array, da i a j indico con

$$A[i..j]. \tag{2.3}$$

2.1.3 Un primo algoritmo: Insertion sort

Progettare un algoritmo significa immaginare nella testa una sequenza di passi che portino ad un risultato. In generale il primo algoritmo che si può immaginare per affrontare un problema di ordinamento è quello di prendere la testa del vettore, scorrere il secondo vettore che sto costruendo e inserire la carta nel punto adatto (dove devo definire che cos'è il punto adatto).

```
for j=2 → A.length do
    key ← A[j];
    i ← J-1;
    while i>0 and A[i] > key do
        A[i+1] ← A[i];
        i ← i-1;
    end
    A[i+1] ← key;
end
```

Algorithm 1: Insertion sort

L'ordinamento per inserzione si basa sull'idea di *cercare un punto di inserzione.* Se ho un blocco di oggetti prendo la prima carta, la seconda sarà da disporre ordinata rispetto alla prima, quindi devo *trovare il punto di inserimento della seconda*

carta (la posiziono a destra o a sinistra a seconda dell'ordinamento). Ovviamente quando trovo il punto di inserzione devo shiftare un pochino le altre carte, quelle che ho già. Quindi trovo un ordinamento cercando di volta in volta il punto esatto di *inserzione della carta*.
In generale, raccolta la j-esima carta, devo prendere le j-1 carte e farle passare valutando di volta in volta l'ordinamento tra j e i ($i \in (1, j-1)$).
Il vantaggio è che le $j-1$ carte che ho già sono ordinate. Questa osservazione aiuta nella progettazione di un algoritmo, infatti in linea di massima posso valutare delle **proprietà temporali**, **dinamiche** che mi aiutano a capire se l'algoritmo è corretto o meno (a seconda che queste si mantengano o meno nel tempo).

NB: La correttezza deriva dall'analisi di proprietà che si creano mentre scrivo l'algoritmo

2.1.4 Correttezza

Vedere che l'algoritmo funziona su certe istanze particolari non è sufficiente a dimostrarne la correttezza.
`Insertion sort` è un ciclo che iterazione dopo interazione fa sì che *io mi stia costruendo la soluzione per segmenti successivi*, `insertion sort` è quindi un esempio di **algoritmo incrementale**, la soluzione si trova a partire dalla soluzione ottenuta al passo precedente.

Quello che si fa con algoritmi di questo tipo è cercare di *dimostrare delle proprietà interessanti* che hanno l'algoritmo e la struttura dati, dimostrando che queste proprietà sono vere per ogni iterazione dell'algoritmo, queste proprietà si definiscono **proprietà loop-invariant**; dimostrare questo mi consente di dimostrare la correttezza dell'algoritmo. Per algoritmi incrementali quello che interessa è spesso *formalizzare* bene la proprietà. All'iterazione J l'array

$$A[1, ..J] \tag{2.4}$$

è ordinato; se riesco a dimostrare ch questa proprietà è vera per ogni J allora il vettore finale sarà ordinato.
La dimostrazione di questa proprietà ricalca sostanzialmente il processo di dimostrazione induttiva.

Dimostrazione

Per il primo step, $J = 2$ il vettore $A[1]$ è per forza ordinato, suppongo allora che sia vera per J, allora, posso dimostrarlo per $J + 1$?

Cioè posso dimostrare che se $A[1..J-1]$ allora $A[1..J]$ è ordinato? La risposta è sì perché so che a destra e a sinistra del punto di inserzione il vettore era ordinato e l'inserimento *ha mantenuto l'ordine locale*.Ora devo capire come questa proprietà invariante si traduce alla fine del ciclo.
Si esce dal ciclo quando $J =$`A.length()+1`, l'ordine ha continuato a mantenersi, la loop-invariance mi dice che il vettore è ordinato. Quindi ho dimostrato la correttezza tramite una proprietà che continua a mantenersi passo dopo passo.

2.1.5 Analisi delle prestazioni

L'analisi delle prestazioni di un algoritmo (in informatica questa branca si chiama *teoria delle complessità*) è lo studio del costo computazionale dell'algoritmo.

Ciascun processo naturale che funziona consuma qualcosa, quando gli algoritmi lavorano utilizzano *risorse computazionali*. Esempi di risorse computazionali che utilizza un programma

- memoria: possibilmente vorrei minimizzarla
- tempo: principale risorsa computazionale, voglio algoritmi che impieghino il minor tempo possibile
- assorbimento di energia
- larghezza di banda (se è un programma che viene svolto in rete)
- numero di processori (se lavoro in parallelo)

Spesso queste risorse computazionali sono in competizione l'una con l'altra. In un corso classico di algoritmi quello che si vuole minimizzare (in termini di risorse computazionali) è il tempo; quindi devo prima di tutto cercare come definire il tempo all'interno dell'algoritmo e poi minimizzarlo.

Gli algoritmi che progetteremo qui sono per computer mono-processori con accesso ad una memoria centrale (macchina RAM), ovvero algoritmi per computer tradizionali.

Ottimizzazione tempo

Specificare il tempo in linea teorica richiede di determinare *quali operazioni ho a disposizioni sul mio processore*, quindi, per prima cosa devo concordare sul **set di istruzioni disponibili**, dopo di che devo concordare sul *costo di ciascuna istruzione*. Bisogna tenere conto anche del fatto che i processori tendono ad avere un set di istruzioni che siano il più ragionevoli possibile, il più minimale possibile. Quindi negli algoritmi potrò utilizzare **operazioni matematico - logiche** semplici, assumo di poter fare *scrittura* e *assegnazione* di variabili, di avere a disposizione **istruzioni condizionali** e possibilità di **dichiarare sub-routine**.

Per non rendere l'analisi dei tempi troppo complicata ma mantenerla significativa devo fare molte semplificazioni (eg. ogni somma coincide un clock del processore, in realtà non è sempre così perché dipende dalle dimensioni granulari del numero, da quanti bit di memoria occupa il mio numero). Qui però sto lavorando in pseudo codice quindi assumo che sia sempre 1 clock.

Complessità in tempo di Insertion-sort

- la misura del tempo deve tenere conto che l'algoritmo rimane lo stesso ma questo deve essere una *funzione della dimensione dell'input*
- a parità di dimensione dei dati di ingresso potrei avere tempi diversi (non è determinato solo dalla taglia)

Quindi posso pensare al tempo come una funzione che dipenda dalla taglia dell'input. Naturalmente la definizione di taglia dell'input varia da problema a problema; ad esempio nel caso di un problema di ordinamento la taglia dell'input coinciderà con la lunghezza dell'array, in un problema tipo TSM sarà il numero di città che devo moltiplicare ecc..

Chapter 3

Lezione 2: Complessità temporale $T(n)$

La tecnica che sta alla base dell'algoritmo di insertion-sort è una tecnica che è detta **incrementale**: la soluzione viene costruita man mano, passo dopo passo. Nello specifico iterazione dopo iterazione prendo una nuova componente dell'array che costruisce un nuovo segmento ordinato. Quindi incremento man mano lo spazio della soluzione che al termine è tutto l'array, quindi il segmento ordinato che ottengo al termine è una soluzione del problema originario. Con questo esempio di algoritmo abbiamo iniziato a vedere i due aspetti fondamentali della costruzione degli algoritmi:

- analisi della correttezza di un algoritmo: In questo caso l'algoritmo è costituito da cicli, la tecnica che viene utilizzata in questi casi è quella di trovare una proprietà che sia loop-invariant. Ovvero trovare una proprietà che viene conservata in ogni ciclo (e che quindi viene conservata anche nel tempo).
- costo computazionale di un algoritmo: In questo corso vediamo solo algoritmi per macchine sequenziali per cui la risorsa computazionale fondamentale è la risorsa tempo. In questo caso quindi devo misurare quanto tempo impiega l'algoritmo ad arrivare ad una soluzione; l'obiettivo naturalmente è quello di impiegarci il minor tempo possibile.

3.1 Come misurare il tempo sugli algoritmi

Si pone a questo punto il problema della misura del tempo richiesto da un algoritmo per giungere a termine, ovviamente potrei anche pensare di cronometrare il tempo impiegato dall'algoritmo a risolvere una determinata istanza, ma dopo un breve ragionamento ci si rende conto che questa soluzione risulta impraticabile per due motivi:

1. non è possibile misurare il tempo dell'algoritmo su tutti gli array, non posso fare una misura locale per ciascuna istanza, ho bisogno di una misura del tempo che sia più generica.
2. se misuro con un cronometro il tempo impiegato dall'algoritmo a risolvere una determinata istanza non misuro davvero il tempo dell'algoritmo ma misuro quello del processore (se cambio processore ma mantengo lo stesso algoritmo posso avere tempi diversi).

inoltre

Un'informazione intuitiva che avevamo ottenuto era che i tempi di esecuzione degli algoritmi crescono al crescere della taglia dell'input; la risorsa tempo quindi deve essere **funzione della taglia dell'input**, l'algoritmo non cambia ma il tempo aumenta all'aumentare della taglia dell'input. Devo anche considerare che la definizione di tempo possa fornirmi valori diversi anche a parità di taglia di input, a parità di taglia posso avere due o più tempi diversi. Quindi l'algoritmo sarà funzione della taglia dell'input e poi cercherò di capire come tenere conto del fatto che anche per input della stessa taglia posso avere tempi diversi.

Taglia dell'input La taglia dell'input è definita sostanzialmente come il numero dei dati che do come input all'algoritmo, prendo come riferimento la taglia più *naturale* da definire per ciascun problema (eg. numero di righe e colonne di una matrice, dimensione di un array, numero di punti in un grafo).

3.2 Tempo di un algoritmo

A questo punto definiamo una funzione che mi consenta di definire il tempo richiesto da un algoritmo per giungere a termine.

Complessità in tempo di un algoritmo Definisco il tempo come una funzione $T(n)$ dove n è la taglia dell'input. A ciascun algoritmo associo la sua funzione di complessità in tempo, che mi restituisce il tempo di funzionamento dell'algoritmo su un input di dimensione n. Dato che a parità di taglia di input posso avere valori di tempo diversi, intuitivamente **posso calcolarlo come il numero di istruzioni che vengono eseguite dall'algoritmo** (ovvero numero di operazioni che faccio seguendo lo pseudo-codice che rappresenta l'algoritmo); l'algoritmo infatti dovrà eseguire più o meno istruzioni a seconda della dimensione dell'input, e ovviamente a parità di numero di istruzioni il tempo di esecuzione può essere maggiore o inferiore a seconda di alcune caratteristiche dell'input.

3.2.1 Complessità in tempo di insertion-sort

NB: io misuro il numero di passi ma in realtà ogni passo corrisponde a un'operazione sul processore, la cui durata dipende dal tipo di processore e dal tipo di passo. In pratica per essere il più preciso possibile dovrei voler inserire anche il costo hardware (sempre in tempo) di ciascuna operazione nel conteggio del tempo. Assumo

```
for j=2 → A.length do
    key ← A[j];
    i ← J-1;
    while i>0 and A[i] > key do
        A[i+1] ← A[i];
        i ← i-1;
    end
    A[i+1] ← key;
end
```

Algorithm 2: Insertion sort

di avere n oggetti da ordinare e voglio capire quanto ci mette l'algoritmo. Valuto ciascuno degli step dell'algoritmo

1. assumo che ogni assegnamento costi `c1`, questo assegnamento viene fatto `n` volte.
2. assumo che ogni assegnamento costi `c2`, questo assegnamento viene fatto `n`-1 volte. Utilizzo una costante diversa perché questo è un assegnamento tecnicamente diverso dal primo
3. assumo che ogni assegnamento costi `c3`, questo assegnamento viene fatto `n`-1 volte, in questo caso il motivo per cui utilizzo un terzo costo computazionale è che questo assegnamento prevede anche una sottrazione.
4. Il costo del quarto step ha il costo della verifica di due booleane, `c4`. La domanda diventa, quante volte questa condizione deve essere verificata. Deve essere verificata $\sum_{J=2}^{n}(t_J)$
5. A questo punto, verificate le due condizioni booleane devo capire quante volte avviene il primo assegnamento all'interno del `while`, assegnamento che assumo abbia un costo `c5` e che venga effettuato $\sum_{J=2}^{n}(t_J - 1)$ volte.
6. Naturalmente il secondo assegnamento all'interno del `while` avviene sempre $\sum_{J=2}^{n}(t_J - 1)$ e ha un costo `c6`.
7. l'ultima istruzione avrà un costo `c7` e viene eseguita tante volte quanto il primo assegnamento quindi `n-1` volte.

.

[1] La complessità in tempo dell'algoritmo è data dalla somma di tutti questi termini, moltiplicato ciascuno per il numero di volte in cui il passo viene effettuato. Quindi

$$T(n) = n{\cdot}c1+(n-1){\cdot}c2+(n-1){\cdot}c3+c4{\cdot}\sum_{J=2}^{n}(t_J)+c5{\cdot}\sum_{J=2}^{n}(t_J-1)+c6{\cdot}\sum_{J=2}^{n}(t_J-1)+c7{\cdot}(n-1) \tag{3.1}$$

Il tempo di esecuzione può variare a seconda della natura dei dati, pur avendo sempre n dati. In particolare se il vettore è ordinato allora $t_J = 1$

$$T(n) = \tag{3.2}$$
$$c1 \cdot n + c2 \cdot (n-1) + c3 \cdot (n-1) + c4 \cdot (n-1) + c7 \cdot (n-1) = \tag{3.3}$$
$$(c1 + c2 + c3 + c4 + c5)n - (c2 + c3 + c4 + c7) = \tag{3.4}$$
$$an + b \tag{3.5}$$

Quindi in caso di **vettore già ordinato** ho complessità **lineare** nel tempo. Se invece l'array è ordinato secondo un ordine inverso rispetto a quello richiesto la dipendenza del tempo dalla taglia dell'input è quadratica; infatti $t_J = J$ le somme mi danno ciascuna un termine del tipo $\frac{n(n+1)}{2}$

Quindi si vede che la taglia dell'input mi dà alcune informazioni sul tempo dell'algoritmo ma mi trovo con la situazione seguente

$$\begin{cases} an + b \text{ best case} \\ a'n^2 + b'n + c \text{ worst case} \end{cases} \tag{3.6}$$

[1] La t all'interno della sommatoria è una variabile generica non ha che fare con il tempo

L'approccio migliore da utilizzare quando bisogna definire la prestazione di un algoritmo è quello del caso peggiore. Quindi il modo in cui si risolve la questione del doppio valore a parità di taglia dell'input è definendo la complessità dell'algoritmo nel caso peggiore. Il tempo di esecuzione di un algoritmo si misura nel caso peggiore, quindi quando si parla della complessità di un algoritmo si parla - quasi sempre - di complessità - **worst case**.

Complessità worst-case è chiamata complessità worst case la complessità in tempo dell'algoritmo sull'input per cui, a parità di taglia, la performance dell'algoritmo è peggiore. A meno che non sia diversamente specificato la complessità che viene fornita è sempre la complessità worst case.

Il motivo per cui si dà la complessità worst case è che questa definisce un *upper-bound* onesto (ovvero un *upper-bound* **valido su tutti gli input**). Inoltre, mentre in alcuni casi il caso peggiore è da ricercare con cura, in altri casi il caso worst-case è abbastanza frequente (casi in cui worst case non è proprio improbabile). L'altro motivo per cui viene fornita generalmente la complessità worst-case è che worst-case è abbastanza simile all'*avarage* case, la complessità *avarage-case* è definita mediante una media su tutti i valori di T(n); tra l'altro, per fare questo, avrei bisogno di una distribuzione di probabilità su cui calcolare la media (su cui pesare i valori, che generalmente non ho. Inoltre, la worst case complexity mi dà funzioni molto più manipolabili da un punto di vista matematico rispetto all'*avarage* case

Quello che faremo ogni volta che faremo ogni volta che vedremo un algoritmo sarà quindi cercare di determinare il numero di passi nel caso peggiore.

Si osservi che non sono state specificate le costanti perché per n grandi il termine che dà il contributo principale è quello maggiore in n. Posso anche specificare i coefficienti del polinomio ma non è particolarmente utile. Quello che si dà generalmente nella valutazione in tempo della complessità degli algoritmi è **l'andamento per n grande**, non solo perché è quello che conta ma anche perché gli algoritmi vengono utilizzati su un numero grande di dati (non utilizzo un algoritmo per fare un "lavoro piccolo". Quello che si fa quindi è fornire **l'ordine di grandezza del comportamento dell'algoritmo**. In questo caso, per questo algoritmo di insertion-sort scriverei

$$\mathcal{O}(n^2) \tag{3.7}$$

In questo caso abbiamo visto in azione una tecnica incrementale, ho le istanze (gli input) e vado a cercare la soluzione finale andando a risolvere dei sotto-problemi, dei segmenti iniziali della mia istanza, in generale **il tasso di ampliamento della soluzione giova della soluzione parziale costruita fino a quel momento**. Gli algoritmi incrementali hanno una forma tipica in cui si ha un ciclo all'interno del quale di volta in volta si allarga il segmento di soluzione.

Proprio perché gli algoritmi incrementali hanno questa forma specifica, una tecnica che è possibile utilizzare per stabilire la correttezza di un algoritmo è quella *loop-invariant* (devo trovare la proprietà, che si conserva per ogni ciclo, che mi dice che la soluzione finale è quella corretta).

3.3 Conclusioni

La prestazione di un algoritmo viene misurata stimando la risorsa principale che viene consumata dall'algoritmo, quindi nel caso di *algoritmi seriali* come quelli che vediamo, si tratta della risorsa tempo. La complessità in tempo di una algoritmo per noi è una funzione $T(n)$

$$T : \mathbb{N} \to \mathbb{N} \tag{3.8}$$

T(n) : **fornisce l'ordine di grandezza del massimo numero di passi effettuati da un algoritmo**.

Per definire questa risorsa si utilizza sempre un'analisi worst case a meno che non venga specificato diversamente. Nel caso di `insertion-sort` abbiamo dato una definizione analitica della complessità in tempo, in generale però mi basta sapere come crescere al crescere di n la complessità in tempo, quindi uso solo l'ordine di grandezza (non faccio una analisi molto fine).

Chapter 4

Lezione 3: Divide et impera

4.1 Divide et impera - fine lezione 2

Altra tecnica di costruzione degli algoritmi (tecnica del tutto generale) che introduciamo sull'algoritmo di ordinamento è la tecnica `divide et impera`.

La tecnica divide et impera utilizza degli strumenti diversi per valutare sia la correttezza che la complessità (in tempo) di un algoritmo. Questo algoritmo, si vedrà, ha una complessità in tempo che è, anche se di poco, al di sotto di $\mathcal{O}(n^2)$ L'algoritmo che vediamo prende il nome di merge-sort (algoritmo per fusione, quello che faccio è *fondere delle soluzioni parziali*). La complessità in tempo è $n\log(n)$, ovviamente inferiore a alla complessità in tempo di `insertion sort`.

NB: Si noti, a partire dal caso particolare del confronto degli algoritmi `insertion sort` e `merge sort` l'importanza di effettuare la valutazione di complessità su un numero grande di dati (se la effettuassi su un numero piccolo di dati l'algoritmo più efficiente potrebbe sembrare diverso da quello che è in realtà.

Inoltre si può dimostrare che **non possono esistere algoritmi di ordinamento che ci possano impiegare meno di $n\log n$ passi (ovviamente valutazione worst case)**. Quindi, non solo ho un lower-bound sulla complessità in tempo di un algoritmo per effettuare una determinata operazione, ma ho anche un algoritmo che raggiunge il minimo possibile di complessità in tempo, quindi è uno dei migliori algoritmi possibili. Un algoritmo di questo tipo viene detto un algoritmo **ottimo**.

L'approccio di questo tipo è un approccio assolutamente naturale: prima di tutto individuo nel problema dei sotto-problemi. Ho un problema grande: allora cerco di spezzarlo in tanti problemi che sono un po' più semplici, visto che sono problemi più semplici allora questi problemi li so risolvere (ho già algoritmi efficienti per risolvere questi problemi piccoli). Le soluzioni parziali così ottenute possono essere combinate per risolvere il problema originario. Tra l'altro spesso i sotto-problemi che ho costruito sono "copie" del problema originale ma di taglia più piccolo (es. ricerca binaria), a volte questi sotto-problemi non sono altro che il problema originale ma segmentato, su delle taglie minori. Quando utilizzo un approccio del tipo divide et impera ho una sorta di albero fino a che non ho raggiunto la complessità *minima*, dopo di che posso ricombinare i risultati dei singoli problemi possono essere ricombinati a dare la soluzione di tutto il problema.

L'idea del divide et impera è che un problema visto nella sua interezza può essere complicato da affrontare ma *stemperando la complessità del problema in-*

iziale posso risolverlo. Molto spesso la suddivisione in sotto-problemi è di fatto una suddivisione dell'istanza su cui devo lavorare; quindi è come se avessi lo stesso problema ma su istanze più piccole. Ovviamente questo lo faccio pagando come prezzo la ricombinazione delle diverse soluzioni parziali.

Se i singoli pezzi sono ancora lo stesso problema iniziale allora posso portare agli estremi il ragionamento. Si dice che ho una **discesa ricorsiva**, per cui il problema è lo stesso del problema generale ma su istanze più piccole con la speranza che l'istanza più piccola diventi immediatamente risolubile (ovvero spero a un certo punto di ottenere istanze del problema generale ma per cui la soluzione è immediatamente evidente). Con questo approccio ricorsivo arrivo allo stesso problema ma su istanze più piccole. La divisione dall'istanza iniziale mi porta ad una dimensione dell'istanza per cui la soluzione è immediata, ciascuna di queste soluzioni ovviamente non è la soluzione del problema generale; però ricombinando le varie istanze ottengo la soluzione del problema generico.

Il vantaggio dell'utilizzare questo algoritmo deriva dal fatto che **gli algoritmi di ricombinazione delle soluzioni parziali sono più semplici della risoluzione del problema generale**, ovviamente però la difficoltà si sposta dal problema generale alla regola di ricombinazione delle soluzioni parziali.

4.2 Ricerca del massimo

Applico l'approccio `divide et impera` alla ricerca del valore massimo in un array

- input: array di numeri (in generale una sequenza numerica, non per forza un array, in questo caso assumo un array senza perdita di generalità)
- output: numero massimo della sequenza

Come primo approccio per affrontare questo problema posso pensare di ordinare la sequenza e poi prendere il primo elemento, ma ovviamente il costo dell'algoritmo è quello dell'ordinamento del vettore ($n \log n$).

In alternativa, in un primo momento posso pensare ad un **algoritmo incrementale** per la ricerca del massimo: inizio pensando che il primo elemento della sequenza sia il massimo, confrontando con tutti gli altri trovo il vero massimo.

```
k ← A[1];
for j=2 → A.length do
    if A[j]>k then
        k=A[j];
    end
end
```

Algorithm 3: Ricerca incrementale del massimo

4.2.1 Dimostrazione della correttezza

Esattamente come per l'`insertion sort`, questo algoritmo è costruito in modo incrementale: alla j-esima iterazione l'elemento che sale sul podio è il più grande tra tutti quelli visti finora. Gli algoritmi incrementali sono tendenzialmente dei cicli, per questo motivo, per questo tipo di algoritmi dimostrare la correttezza implica il trovare una proprietà loop-invariant (non varia con il tempo).

- è vero per il primo step dell'algoritmo (per $j = 2$ è vero che sul podio si trova il valore maggiore tra a1 e a1 - proprio perché ci si trova $a1$ -)
- assumo che sia vero al passo j, infatti sul podio si trova il più grande tra a_1 e a_j
- a questo punto se $a_{j+1} > a_j$ allora a_{j+1} è il numero più grande di tutti (se invece questa condizione non è soddisfatta a_j rimarrà sul podio ma in questo caso sarà il numero più grande di tutti

Quindi questo dimostra che è una algoritmo incrementale (incremento sempre di più la mia conoscenza sulla posizione del massimo) e ne dimostra la correttezza

4.2.2 Complessità in tempo dell'algoritmo $T(n)$

Dato che questo algoritmo cicla su tutti gli n elementi che gli ho passato come argomento allora $T(n) = \mathcal{O}(n)$. Questo è un **ottimo** algoritmo per trovare il massimo.

4.3 Ricerca del massimo - divide et impera

In un approccio divide et impera una questione fondamentale è il modo in cui vengono divise le istanze (in maniera simmetrica o no, quante parti lo divido). Una cosa assolutamente naturale da fare è dividere a metà l'istanza, dopo di che la stessa operazione può essere ripetuta. Al livello più avanzato di divisione la soluzione è auto-evidente. A questo punto comincia la fase di ricombinazione, la ricombinazione è sempre la stessa **routine** che agisce sempre su due numeri.

```
DEI-MAX(A,p,q);
if p=q then
| return A[p];
end
else
| r = (p+q)/2 ;
| a = DEI-MAX(A, p, r);
| b = DEI-MAX(A, r+1, q);
| m = max(a,b);
| return m;
end
```

Algorithm 4: Massimo DEI

```
max(a,b) if a>b then
| return a;
end
else
| return b;
end
```

Algorithm 5: Massimo

In questo caso l'argomento prende come argomento il vettore e il primo e l'ultimo elemento del vettore.

4.3.1 Ricorsione

La divisione in sotto-problemi giova del fatto che tutti gli strumenti hanno una procedura detta **ricorsione**. Si chiamano procedure ricorsive procedure che possono chiamare se stesse ma su numeri più piccoli (eg. calcolo del fattoriale - il punto auto-evidente del fattoriale è 0).

Viene richiamata su un array A e trova il massimo della porzione di array compresa tra p e q, se devo effettivamente trovare il massimo di un array di n elementi allora dovrò alla fine per forza richiamarla su 1-n.

In questo caso quello che fa l'algoritmo è continuare a scindere l'istanza iniziale fino a quando giungo a una condizione in cui la soluzione è auto-evidente. Quando p=q allora la lunghezza del segmento è un solo elemento dell'array (questa è la base della ricorsione, è il punto auto-evidente della ricorsione).

Base della ricorsione: La base della ricorsione è quell'argomento della ricorsione che ferma la ricerca ricorsiva, è il punto in cui la soluzione è quando è auto-evidente.

NB: Quelle rappresentate all'interno dell'algoritmo non sono parentesi quadre ma sto prendendo la parte intera (okay il segmento non è 1 quindi cerco il massimo su ciascuna delle due metà.

4.3.2 Calcolo della correttezza

In questo caso non ho il loop, quindi non posso usare la tecnica del loop-invariant. Quello che posso fare è dimostrare la correttezza dell'algoritmo per **ricorsione strutturale, induzione**. Per prima cosa dimostrare che è vero per n=1, assumo che sia vera per j per poi dimostrare che è vera per $j+1$. Se $n = 1$ allora la soluzione è evidente, ho un solo numero, per forza questo numero è il massimo, a questo punto assumo che $a =$ `DEI.MAX(A,p,q)` sia vera (quindi assumo che il numero a così definito sia il massimo sugli $n-1$ elementi del vettore che sto considerando), per ipotesi induttiva. A questo punto il fatto che $a =$`DEI.MAX(A,1,n)` si vera è immediato perché da una parte avrò il massimo della prima metà e poi il massimo della seconda e il massimo tra questi due massimi è sicuramente il massimo assoluto.

4.3.3 Complessità in tempo

Dimostrare la complessità in tempo per un approccio divide et impera è un po' più complicato rispetto al caso di un approccio incrementale.

In questo caso ho una equazione di ricorrenza nella funzione incognita $T(n)$, la funzione che cerco va dai naturali ai naturali. Dato che la divisione per 2 nei naturali non viene sempre bene, faccio la seguente assunzione: la dimensione degli array è sempre una potenza di due, ovvero assumo di poter far tornare sempre le mie divisioni. Se così non fosse, non avrei una vera equazione di ricorrenza perché la nuova equazione non è più quella di prima ma avrei anche dei numeri con la virgola (e si traduce in una equazione con una forma diversa e matematicamente più complicata), se n è potenza di due allora la parte superiore e parte inferiori sono uguali e quindi torno al caso generale.

Fatta questa assunzione, per arrivare alla soluzione quello che si usa è il risultato della serie geometrica

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1} \tag{4.1}$$

In particolare se $|x| < 1$ allora $\sum_{i=0}^{\infty} x^i = \frac{1}{x-1}$

Se vado avanti fino a $\log n$ volte ottengo una soluzione che deve essere soddisfatta da quella incognita. Quindi la parte destra che ho ricostruito passo dopo passo mi basta prendere k e sostituirci klogn. In questo modo ottengo una serie geometrica, il risultato della somma geometrica di prima mi come risultato

$$T(n) = 7n - 5 \rightarrow T(n) = \mathcal{O}(n) \tag{4.2}$$

A questo punto controllo che $T(n)$ soddisfi le richieste che ho impostato in precedenza.

Spazio per i conti

Considerazioni

Stimare la complessità in tempo di un divide et impera richiede tendenzialmente di risolvere una equazione di ricorrenza; esistono varie tecniche per risolvere questo tipo di equazioni, quella che abbiamo utilizzato in questo caso prende il nome di soluzione *per sostituzioni successive*.

In questo caso abbiamo fatto tanto rumore per niente nel senso che la complessità in tempo che ho ottenuto è pure peggiore di quella che avrei trovato normalmente (con il primo algoritmo che ho pensato).

D'altra parte però se utilizzo l'approccio divide et impera per ordinare ottengo un algoritmo che ha una complessità in tempo di $n \log n$. Questo algoritmo prende il nome di **merge sort**, questo ordina array in maniera ricorsiva - ovvero usando divide et impera).

4.4 Divide et impera per ordinamento - merge sort

Immagino di dover utilizzare un array di un numeri. Cerco una fusione di array ordinati in array ordinati, indipendentemente da quanto sono lunghi, quindi ordino un array a partire da due array che sono già ordinati, non riordino una cosa disordinata. è a questo che mi è servita la discesa iterativa, è questa la fusione che avviene, la funzione merge non è una funzione casuale ma è una funzione che lavora su segmenti già ordinati

Chapter 5

Lezione 4: Merge sort

`Merge sort` è un algoritmo di ordinamento molto interessante perché è l'algoritmo ottimo, quindi `merge-sort` è una *tecnica* che può portare a buoni argomenti e case-study. Merge sort lavora con la tecnica divide et impera che ha una ricerca ricorsiva per individuare (al termine della ricorsione) dei casi in cui la risposta è auto-evidente. Questa è la prima parte dell'algoritmo di merge-sort.

A questo punto, una volta terminata la routine di divide et impera, prende avvio la seconda fase dell'algoritmo. Ricordiamo che quello che voglio fare è riordinare un vettore costituito da *una potenza di due* elementi (scelgo una potenza di due senza perdita di generalità solo per semplificare i calcoli nella dimostrazione di correttezza e nel calcolo della complessità in tempo dell'algoritmo). Una volta che riduco il problema di ordinamento a un punto in cui la soluzione è auto-evidente (ho un vettore di un solo elemento) il problema che si pone è quello di ordinare queste diverse soluzioni auto-evidenti, infatti, nessuna di queste soluzioni auto-evidenti è soluzione del problema iniziale (è **parte** della soluzione del problema iniziale). Ho quindi bisogno di una **routine di ricombinazione** che mi dia soluzioni sempre più ampie della stessa istanza. Quindi ho una operazione di **fusione** di *una sequenza ordinata* con un'*altra sequenza ordinata*. [1]

5.1 Routine di merge

Per prima cosa mi fisso sulle due teste dei vettori, in questo modo il minimo tra i due elementi è il minimo totale delle due liste; prendo questo minimo e lo metto da parte. Fatto questo rimangono una testa (quella di prima che non ho toccato) e quella nuova del vettore da cui ho tolto il primo (per cui questa nuova testa è l'elemento successivo). Questa routine di `merge` funziona su *segmenti* dell'array originale.

Scriviamo ora uno pseudo-codice per la routine di `merge`. Per prima cosa ho bisogno di 2 array di appoggio che contengano le due sotto-sequenze (array dichiarati localmente). A questo punto

1. confronto le teste dei due array (`L` e `R`)

2. la minore la metto all'inizio del segmento che devo ordinare per fusione (segmento che originariamente era costituito da due parti localmente ordinate)

[1] L'operazione di fusione funziona bene solo perché sto lavorando con segmenti ordinati, se non fossero tali allora l'operazione di fusione funzionerebbe molto male.

3. avanza la testa del vettore da cui ho preso il numero: l'indice della prima testa viene aggiornato

4. aggiorno l'indice di `A` che deve ospitare il nuovo minimo tra i numeri rimanenti

5. Gli array che accolgono la prima e la seconda sotto-sequenza ordinata sono terminati da un simbolo convenzionale (indicato come infinito) ma ognuna di queste liste viene chiuso da un simbolo che è più alto di tutti perché in questo modo **se una sequenza viene esaurita allora si passa alla seconda e si continua a svuotare la seconda**. Non è necessario che la sentinella sia la stessa per entrambi gli array.

```
MERGE(A,p,q,r);
n_1 ← (q-p+1);
n_2 ← (n-q);
L[1,...,n_1 +1];
R[1,...,n_2 +1];
for i←1 to n_1 do
 | L[i] ← A[p+i-1]
end
for j←1 to n_2 do
 | R[j] ← A[p+j] ;
end
L[n_1 + 1] ← ∞;
R[n_2 + 1] ← ∞;
i←1;
j←1;
for k←p to r do
 | if  L[i] ≤ R[j] then
 |  | A[k] ← L[i];
 |  | i ← i+1;
 | end
 | else
 |  | A[k] ← R[j];
 |  | j←j+1;
 | end
end
```

Algorithm 6: Ordinamento Merge

Con i primi due cicli `for` sto riempiendo o vettori, subito dopo aggiungo in coda ai due vettori le due sentinelle (che in questo caso indico con ∞). Dopo di che ho lo sviluppo formale dell'algoritmo ch ho descritto nell'elenco puntato sopra.

5.1.1 C(n): complessità in tempo di `merge` per creare una sequenza ordinata di n elementi

Ho il primo salvataggio degli n elementi (fino a che devo assegnare i e j pari a 1) dopo di che spenderò una altro numero lineari di passi.

5.1.2 Correttezza di merge

La correttezza di questa seconda routine può essere studiata tramite un invariante di ciclo. Ad esempio posso dimostrare che ad ogni iterazione del `for` nell'array finale ci sia sempre una sequenza ordinata. Infatti la parte "inferiore dell'algoritmo" è dimostrato per induzione.

5.2 Algoritmo finale

Quello che ho appena visto è solo l'algoritmo di ricombinazione. Devo ora considerare l'intero algoritmo. Se $p \geq r$ significa che non c'è niente da ordinare (sono alla dimensione dell'array oppure ho superato la dimensione dell'array). Se non è così significa che c'è qualcosa da ordinare, quindi c'è un segmento con dei dati, allora prendi il punto medio, ordina la prima parte, ordina la seconda parte (sempre chiamando ricorsivamente a se stessa e concludendo la discesa ricorsiva). Al termine della discesa ricorsiva puoi ricombinare.

```
MERGE-SORT(A,p,r);
if p<r then
    q← [(p + r)/2];
    MERGE-SORT(A,p,q);
    MERGE-SORT(A,q+1,r);
    MERGE(A,p,q,r);
end
```

Anche in questo caso devo dimostrare la correttezza e calcolare la complessità dell'algoritmo.

5.2.1 Correttezza di `merge-sort`

Prima di tutto dimostro che è corretta su array di dimensione 1. Suppongo per ipotesi induttiva che questo algoritmo sia corretto per vettori di n/2 allora, dato che ho dimostrato prima la correttezza locale di `merge` sicuramente è vero per n.

5.2.2 Complessità dell'algoritmo

Guarda slide 4-5-6 lezione 5.

Chapter 6

Lezione 5: Matrici e simbologia

6.1 Prodotto di matrici

6.2 Complessità in tempo dell'algoritmo

Ho la traduzione informatica della sommatoria Dato che la complessità in tempo dell'algoritmo è il numero di operazioni che faccio per moltiplicare due matrici NxN, questo che è l'algoritmo ovvio ha un tempo N^3, che non è nemmeno così male considerato che è un tempo polinomiale. Il problema è che quando aumento le dimensioni delle matrici ho bisogno di un tempo che vada almeno come N^2.

Diminuzione complessità Ora il problema diventa quello di cercare di abbassare la complessità in tempo dell'algoritmo, quindi in sostanza abbassare il grado di T, si può osservare che posso abbassare questo tempo ma non riuscirò mai a scendere al di sotto di N^2. Non posso scendere sotto il quadrato perché devo almeno leggere tutti gli elementi di una matrice che sono esattamente N^2. Ovviamente si sta parlando di matrici generali (ovviamente mi metto nel caso peggiore), sicuramente ho un algoritmo che va come il cubo e al limite posso ridurlo al quadrato. Se voglio cercare di abbassare la complessità in tempo dell'algoritmo quindi devo cercare una complessità tra il cubo e il quadrato. Per molti anni si è creduto il tempo cubico non fosse migliorabile, in realtà si è riusciti ad abbassare il limite con un algoritmo ricorsivo di tipo "divide et impera" che rimanda il problema del calcolo del prodotto a matrici di dimensioni sempre inferiori.

Il primo approccio può essere quello di dividere le due matrici originali in blocchi (rimando il problema del calcolo del prodotto delle due matrici) a matrici che a questo punto hanno dimensione sostanzialmente $\frac{N}{2} \times \frac{N}{2}$. Per farlo chiamo A1 il blocco in alto a sinistra della matrice A e poi procedo da sinistra a destra con la numerazione, lo stesso faccio con la matrice B. A questo punto anche la matrice risultato me la immagino divisa a blocchi. L'algoritmo di merge è quello che reinserisce ciascun blocco nella sua posizione originaria, quello che ho fatto ora è stato semplicemente rimandare il problema ad 8 sotto-problemi. A sua volta la matrice A11verrà suddivisa quindi rimanderò il problema nella moltiplicazione di due matrici N/4. Continuerò a rimandare il problema quindi ad utilizzare una divisione fino a quando la matrice non sarà di dimensione 1x1.

A questo punto posso scrivere l'algoritmo ricorsivo per questo compito: per prima cosa mi creo una matrice di appoggio, se la dimensione della matrice è 1x1 allora calcola il risultato, altrimenti le partiziono, la stessa cosa devo fare sulla matrice partizionata. Per partizionare la matrice si propone una routine che abbia degli indici, dopo di che seleziono gli elementi che corrispondono ad un certo numero di indici.

```
DEI-MULT(A,B) n ← A.row C ← n × n matrix if n==1 then
| C_11 = a_11 · b11
end
else
| Partition(A,B,C) C_11 ← DEI-MULT(A_11, B_11) +
|  DEI-MULT(A_12, B_21) C_12 ← DEI-MULT(A_11, B_12) +
|  DEI-MULT(A_12, B_22) C_21 ← DEI-MULT(A_21, B_11) +
|  DEI-MULT(A_22, B_21) C_22 ← DEI-MULT(A_21, B_12) +
|  DEI-MULT(A_22, B_22)
end
return C
```

Algorithm 7: DEI per moltiplicazione di matrici

Complessità in tempo Il T_n in questo caso dipende dalla complessità dei sotto problemi, anche in questo caso devo scrivere l'equazione di ricorrenza per questo schema. IL primo step sarà ovviamente porre $T_n = 1$ se $n = 1$. A questo punto dovrò usare $\frac{n}{2}$, devo tenere conto anche della somma (devo sommare componente per componente). Quindi la complessità in tempo dell'algoritmo è peggiore di quella che volevo, è ancora una volta $\mathcal{O}(n^3)$. Questo mostra che ci sono casi in cui non posso applicare divide et impera in maniera banale suddividendo sempre a metà il problema, ci sono casi in cui ho bisogno di prestare attenzione al tipo di divisione che faccio e devo sceglierla in maniera tale da dover fare poi meno iterazioni ed eventualmente meno operazioni possibili.

Questa è proprio l'idea che sta alla base della costruzione di quell'algoritmo che mi dà una complessità n^2. La suddivisione delle matrici che fa è particolare. Intanto le matrici di appoggio sono 7, ha organizzato in modo diverso le moltiplicazioni, in maniera tale da dover fare solamente 7 prodotti su matrici $\frac{n}{2} \times \frac{n}{2}$, anziché 8. Infatti tutte le matrici che mi servono le posso ottenere come somma dei prodotti queste 7 matrici. Questo algoritmo è ancora un algoritmo ricorsivo, l'algoritmo è sempre divide et impera, però porta al richiamo della stessa routine 7 volte anziché 8. Quindi anche l'equazione di ricorrenza è molto simile di quella di prima m la soluzione di questa equazione è $n^{\log 7}$ che è minore di n^3. In sostanza, quello che dice l'algoritmo è che il calcolo del prodotto riga colonna tra matrici, organizzando meglio le cose può essere fatto in meno di n^3 step. Solamente gli algoritmi sono adattativi, quindi in alcuni casi specifici funziona meglio in altri peggio e a seconda del tipo di matrice che ho posso utilizzare un algoritmo diverso.

Chapter 7

Strumenti matematici per l'analisi dei dati

Si introducono ora alcuni concetti matematici per l'analisi degli algoritmi e più in generale utili per il **conteggio di strutture discrete** infatti siamo nell'ambito di funzioni f, g.

$$f, g : \mathbb{N} \to \mathbb{N} \tag{7.1}$$

Si introducono strumenti matematici finalizzati allo studio e alla soluzione di equazioni di ricorrenza, equazioni fondamentali per l'analisi degli algoritmi.

7.1 Notazioni matematiche

Tasso di crescita delle funzioni Non mi interessa una conoscenza precisa del tempo impiegato da un algoritmo ma mi interessa il tasso di crescita. Per tasso di crescita di una funzione si intende *come va una funzione per N grande* (non potrebbe essere diversamente perché lavorando con i naturali l'unico punto di accumulazione possibile è l'infinito).

$$f_n = \mathcal{O}(g_n) \text{ if } \exists k > 0 \text{ t.c for } n(k) > k,\ f_n < g_n \tag{7.2}$$

In generale quando sto confrontando queste due funzioni, intuitivamente significa che *da un certo punto in poi la crescita di f è dominata dalla crescita di g*, non so esattamente come vada. Si noti che n dipende da k perché a seconda di qual è il punto che prendo dipende dal valore della costante.

$$f_n = \Theta(g_n) \text{ if } \exists c_1, c_2 > 0 k > 0 \text{ t.c for } n(k) > k, \text{ , } g_n - c_1 < f_n < g_n + c_2 \tag{7.3}$$

In sostanza la Θ è una sorta di miglioramento di $\mathcal{O}$, in questo caso significa che la crescita di f sta proprio in una striscia creata intorno a g aumentata di una costante e diminuita di una costane.

$$f_n = \Omega(g_n) \text{ if } \exists k > 0 \text{ t.c for } n(k) > k, \text{ , } f_n > g_n \tag{7.4}$$

L'ultimo simbolo di Landau spesso utilizzato serve per stabilire un lower-bound sulla crescita di una funzione, la funzione cresce almeno tanto quanto g_n.

$$f_n = o(g_n) \text{ if } \lim_{n\to\infty} \frac{f_n}{g_n} = 0 \tag{7.5}$$

Questo significa che il numeratore è molto più piccolo del denominatore, per quanto riguarda il contrario si utilizza invece θ:

$$f_n = \theta(g_n) \text{ if } \lim_{n\to\infty} \frac{f_n}{g_n} = \infty \tag{7.6}$$

7.2 Altri metodi per la risoluzione di equazioni di ricorrenza

Le equazioni di ricorrenza sono fondamentali per l'informatica, in alcuni casi la sostituzione non funziona così bene ma magari ci sono altri modi per stimare il tasso di crescita di una funzione (se non sono in grado di stimare esattamente il risultato). Questi metodi sono utili quando il metodo ovvio di sostituzione successiva non funziona.

Considero un esempio di equazione (ovviamente, come per le equazioni differenziali in realtà bisogna essere molto creativi).

Primo caso: sostituzione Quando ci sono pavimenti di questo tipo o c'è un soffitto, le sostituzioni successive diventano un po' laboriose, allora quello che si può fare è un'ipotesi sull'andamento del tasso di crescita di queste funzioni. So che è una funzione che da un certo punto in poi si comporterà in un certo modo. Se anche stimassi correttamente questo c non avrei esattamente la soluzione dell'equazione ricorsiva ma sarei per lo meno in grado di descrivere l'andamento. Il primo step del metodo di sostituzione è **cercare di indovinare quanto può essere la crescita**, a questo punto devo verificare che la soluzione sia coerente con l'equazione. Per verificare la **coerenza** della soluzione proposta utilizzo l'**induzione**, suppongo che quanto detto sia vera e poi dimostro che è vera per un valore più grande (ovviamente lo dimostro sfruttando l'equazione di ricorrenza). Dimostro, grazie all'equazione di ricorrenza vera per N. A questo punto Tn non so dire esattamente cosa sia ma se assumo che sia più piccolo di $c \cdot n \log n$ non vado contro la mia equazione di ricorrenza. Osservo che per n=1 non va benissimo però diciamo che non mi interessa più di tanto perché tanto una valutazione del tasso di crescita mi serve per n grande. A questo punto posso provare a fare alcuni tentativi per capire come farlo valere "quasi sempre", in questo caso prendo n=2 e vedo come regolare la costante.

La cosa fondamentale è stata usare l'induzione per dimostrare che il mio tentativo è coerente con quello che dice l'equazione di ricorrenza, dopo di che per i valori piccoli posso anche cercare di aggiustare le costanti.
Il problema di questo modo di risoluzione delle equazioni di ricorrenza è che non so esattamente come fare a fare la prima stima del tasso di crescita, in questo caso mi è andata bene ma non è scontato trovare il tentativo funzionante.
Come stimare il tasso di crescita di una funzione guardando l'equazione per n grande, alcune equazioni che ad un primo sguardo possono sembrare complicate in realtà possono avere un andamento più semplice.

I buoni tentativi si possono fare dicendo "okay più o meno mi ricorda quella che ho visto in questo caso" oppure potrebbe essere utile cercare di restringere il campo di variabilità del tasso di crescita della funzione. Quindi posso ad esempio delimitare la ricerca sia sulla parte superiore che sulla parte inferiore.

Secondo caso: cambio di variabili Una seconda possibilità è quella di ridursi da un'equazione di ricorrenza di un tipo ad una equazione di ricorrenza un po'

più maneggevole tramite un **cambiamento di variabile**. NOTA: quando in informatica non è diversamente specificato la base di un logaritmo è sempre base 2.

A questo punto posso osservare ad esempio che in questo caso quello che ottengo è una funzione non una composizione di funzioni (della nuova variabile si intende). In questo modo posso fare uno studio di funzione e risolvere una equazione più semplice. A questo punto, trovata la soluzione nella variabile m, faccio la soluzione inversa per tornare ad una soluzione espressa nella variabile originaria. L'idea è che le sostituzioni che faccio devono portarmi ad una equazione che conosco.

Terzo caso: master theorem È un teorema fondamentale nella risoluzione di molte, non tutte, equazioni di ricorrenza. Soprattutto aiuta nella risoluzione di equazioni di ricorrenza derivanti da algoritmi con approccio divide et impera. Enunciato: il master theorem afferma che è possibile capire l'ordine di grandezza della soluzione esaminando i coefficienti in gioco. Il master theorem vale anche quando ho in gioco parte superiore o parte inferiore e simili. Mi da delle condizioni per capire la soluzione (guardare slide per vedere enunciato). Dammi a, b, n dell'equazione di ricorrenza. Mi dà un ordine del tasso di crescita della soluzione di una equazione di ricorrenza.

Il teorema principale (in inglese master theorem), noto anche come teorema dell'esperto o teorema del maestro, è un teorema inerente all'analisi degli algoritmi che fornisce una soluzione asintotica ad una famiglia di relazioni di ricorrenza. È stato inizialmente esposto da Jon Bentley, Dorothea Haken, e James B. Saxe nel 1980, dove fu descritto come un metodo unificato per una famiglia di ricorrenze. Il nome di questo teorema è stato popolarizzato dal famoso manuale Introduzione agli algoritmi di Cormen, Leiserson, Rivest e Stein. Non fornisce una soluzione a tutte le possibili relazioni di ricorrenza, ed una sua generalizzazione è il teorema di Akra-Bazzi.

Chapter 8

Lezione 6: Programmazione dinamica

La programmazione dinamica è una nuova tecnica di disegno e progetto di algoritmi. La programmazione dinamica viene in aiuto in molti casi, soprattutto nei casi in cui l'approccio divide et impera non va più bene. Effettivamente molti problemi affrontati con "divide et impera" non forniscono una soluzione efficiente. Questa tecnica è un ulteriore strumento di costruzione di algoritmi per quando le altre tecniche non funzionano. Ovviamente anche in questo caso non è detto che funzioni automaticamente, dipende dal problema che devo affrontare.

8.1 Possibili problemi con "divide et impera"

1 problema quando uso "divide et impera" prendo l'istanza iniziale del problema e la suddivido in sotto-istanze. La suddivisione in sotto-istanze però deve essere **bilanciata**. La misura della sotto-istanza deve essere molto più ridotta rispetto all'istanza iniziale (eg. negli algoritmi visti la mia discesa ricorsiva riduce con **velocità esponenziale** la dimensione). Le stesse tecniche di analisi della complessità mi consentono di vedere questo, ad esempio se ottengo una equazione di ricorrenza non simile a quella di "merge-sort" può essere che il tempo sia assolutamente insoddisfacente.

2 problema Posso anche dividere bene il problema, però può anche essere che io vada a suddividere in sotto problemi che non sono disgiunti (ho un determinato sotto-problema che devo risolvere in punti diversi dell'algoritmo). Altra condizione di cui ho bisogno quindi è che la suddivisione mi porti a sotto-problemi che siano **disgiunti**, non connessi. Ovvero che gli stessi problemi si presentano in punti diversi del mio algoritmo.

8.1.1 alcolo dei numeri di Fibonacci

La sequenza di Fibonacci viene introdotta per risolvere molti problemi "naturali".

$$\mathcal{F}(n)_{n\geq 0} = 0, 1, ... \tag{8.1}$$

La formula che, fissato n, mi restituisce l'n-esimo numero della sequenza di

Fibonacci

$$\mathcal{F}(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{\big)} n + \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right]\phi = \lim_{n\to\infty} = \frac{F(n)}{F(n-1)} \tag{8.2}$$

Questo parametro ϕ è anche chiamato radice aurea. La sequenza di Fibonacci può essere calcolata con un algoritmo del tipo divide et impera (posso utilizzare una routine ricorsiva)

Il problema è calcolare la complessità in tempo di questo algoritmo. Calcolando la complessità in tempo della soluzione di questo problema con una tecnica di tipo divide et impera si osserva che la performance è pessima. Infatti quando costruisco l'algoritmo osservo che la discesa è lineare in r. Inoltre si osserva che i sotto-problemi non sono per nulla disgiunti (devo calcolare lo stesso numero della sequenza un sacco di volte).

La programmazione dinamica è sostanzialmente una programmazione del tipo divide et impera ma con possibilità di memorizzazione dei risultati, questo perché mi sono accorta che sono in un problema in cui ci sono elementi che devo ricalcolare molte volte. Gli algoritmi di programmazione dinamica **memorizzano risultati intermedi** perché in questo modo, quando alcuni problemi intermedi si ripresentano io ne conosco già la soluzione, non devo ricalcolarla.

```
v[0,...,n] ;
v[0] → 0 ;
v[1] → 1 ;
for k=2 to n do
    v[k] → v[k-1] + v[k-2] ;
    return v[n] ;
end
```

Algorithm 8: FIB(n)

I passi che impiego per portare a termine la routine sono n quindi $T(n) = \mathcal{O}(n)$, quello che succede in questo modo però è che occupo uno spazio $\mathcal{S}(n) = \mathcal{O}(n)$.

Naturalmente posso ottimizzare questo algoritmo per occupare meno spazio, ad esempio:

```
if n==0 then
    return 0;
end
else
    a := 0;
    b := 1;
    for k=2 to n do
        t = a+b;
        a = b;
        b = t;
    end
    return b;
end
```

Algorithm 9: FIB(n)

8.2 Problemi di Ottimizzazione

La programmazione dinamica è molto utile nell'affrontare problemi di ottimizzazione.

1. ho una istanza $x \in \mathcal{D}$ che appartiene a un dominio D.
2. ho uno spazio di soluzioni SOL(x) = possibili soluzioni per x
3. devo associare a ciascuna soluzione anche un costo $C : SOL \to \mathbb{R}$
4. output: il problema di ottimizzazione vuole una soluzione che "minimizzi" la funzione costo. Dato un input ho tanti possibili output e voglio prendere quella soluzione che mi minimizza tutto.

8.2.1 Problema del cammino minimo sui grafi

Un grafo è una rete fatta da un insieme di nodi connessa da archi o lati, nello specifico in questo caso ciascun arco è anche pesato (per cui percorrere un ramo ha un certo costo). Un grafo pesato come questo può essere una istanza del mio problema, la funzione costo che devo minimizzare è il costo del cammino. Posso ad esempio decidere che in questo caso il costo di un cammino è la somma (a volte, in un contesto probabilistico o quantistico conviene prendere come costo totale il prodotto dei costi anziché la somma dei costi). L'output che richiedo è il cammino di costo minimo da 1 a 4. Per rimappare il caso particolare nel caso del lucido precedente le soluzioni possibili in questo caso sono tutte quelle che soddisfano i *constraints* (quindi in questo caso sono tutte quelle che vanno da 1 a 4). Il mio algoritmo mi dovrà restituire *una soluzione*, non è detto che ci sia una soluzione che abbia costo minimo.

8.3 Problema del prodotto tra matrici

L'input di questo problema sono N matrici (posso assumere n matrici di interi), quello che voglio fare è calcolare il prodotto iterato di queste matrici. Le matrici sono oggetti bidimensionali, le posso moltiplicare solo se hanno dimensioni opportune (prodotto riga-colonna). Quindi l'input è una sequenza di N matrici con dimensioni opportune (per controllare le dimensioni posso fare come nella slide). Supponendo che le matrici siano moltiplicabili quello che mi chiedo è quale sia il minimo numero di prodotti che posso operare per ottenere il risultato (quello che voglio fare è trovare il risultato della moltiplicazione facendo il numero minimo di prodotti).

Il prodotto di matrici non è commutativo ma è associativo, quindi l'idea è che ho tante strategie di moltiplicazione che ho a disposizione proprio per via del fatto che il problema gode della proprietà associativa. A seconda del modo in cui devo associare le matrici può variare molto il tempo di calcolo di cui ho bisogno.

se moltiplico a× b e b×c allora impiego a×b×c (quindi la mia operazione elementare ha il costo di questo prodotto delle dimensioni).

Ha senso cercare di capire quale sia la strategia di raccoglimento delle matrici. Quello che vorrei è un algoritmo che mi dica quante operazioni al minimo devo fare (quello che voglio fare

Posso utilizzare un approccio esaustivo (quindi provare tutte le combinazioni possibili e scegliere il modo che comporta il minor numero di moltiplicazioni). La

correttezza di questo algoritmo esaustivo è ovvia, il problema però è la complessità, quindi mi chiedo quanti sono i modi di moltiplicare n matrici. Per contare le possibili strategie di associazione. Quello che vedo è che ad ogni tipo di moltiplicazione posso associare una parentesizazione (ovviamente intendo una buona parentesizazione, cioè non ci sono parentesi che restano aperte e via dicendo. quindi ho tanti modi possibili da provare tanti modi quanti sono le parentesizazione di n matrici. Quindi devo esplorare tutte le possibilità di ordinare in maniera buona le matrici.

La domanda iniziale, quanto tempo ci impiego è tante volte quante buone parantesizazioni posso scrivere, questo. Il numero è CN l'n-esimo numero di Catalan, che mi dice quante buone parantesizazioni ci sono per una lista di matrici. Quello che si osserva è che questo numero va come a^n, generalmente impraticabile.

In questo caso ho un'altra possibilità che è l'approccio dinamico. Usa la programmazione dinamica per utilizzare in tempo polinomiale, un buon tempo, qual è il numero minimo di moltiplicazioni che devo fare per calcolare il prodotto tra N matrici.

La programmazione dinamica entra in gioco quando ho provato l'approccio divide et impera ma mi rendo conto che sto calcolando più volte la soluzione di un medesimo problema

Notazione: Indico con `m[i,j]` il minimo numero di moltiplicazioni relativamente al calcolo $M_i \cdot M_{i+1} \cdot ... \cdot M_j$ Ogni strategia arriverà alla fine con due matrici da moltiplicare

Ovviamente sto ragionando ad alto livello perché sto mano a mano in realtà rimandando il problema. Un approccio di questo tipo mi fa subito venire in mente un approccio di tipo "divide et impera"

Il calcolo del numero minimo di operazioni per effettuare il prodotto di una serie di matrici va come n^3, come sempre quando si parla di programmazione dinamica si pone il problema di quanto spazio serve per l'algoritmo (il tempo è quasi ottimale, il costo spaziale è quadratico).

NB: *Per la fine di questa lezione guarda le ultime slide della lezione 7*

Chapter 9

Lezione 7: Teoria dei grafi

La teoria dei grafi si applica ovunque, non solo all'informatica ma anche a molti rami della matematica, a partire dalla topologia, possono essere modellati tramite dei grafi. Saper modellare i grafi e saper dimostrare proprietà sui grafi è fondamentale per potersi immaginare soluzioni algoritmiche a determinati problemi.

9.1 Storia della teoria dei grafi

Si dice che l'introduzione della teoria dei grafi sia avvenuta per la risoluzione di un problema reale, si parla comunque del 1700. Il problema in questione era il problema dei così-detti ponti di Koninsberg (affaccia sul Baltico alla foce di un fiume) e quindi è composta anche da isole, la domanda è se sia possibile fare una passeggiata per Koninsberg (passando una sola volta lungo ciascuno di questi ponti - 7 nello specifico). La risposta a questa domanda è ovviamente negativa. Il primo a rispondere a questa domanda fu Eulero nel 1736 à8anno di pubblicazione di un suo articolo in cui si rispondeva alla domanda in questione), che per affrontare il problema introdusse un novo tipo di rappresentazione.

Il passaggio fondamentale fatto da Eulero è il passaggio di astrazione, è stato quello di concentrarsi sul nucleo del problema, estraendo l'intima essenza del problema, ha distillato esattamente gli elementi che servono per risolvere il problema. Questa nuova rappresentazione ha consentito di generalizzare il problema e renderlo adattabile ad altri problemi, ha introdotto la struttura di punti ed archi che oggi chiamiamo appunto *grafo*.

9.2 Tipologie di grafo e relative proprietà

9.2.1 Grafi diretti (o orientati o di-grafi)

Dal punto di vista formale un grafo diretto G è rappresentato da una coppia di insiemi G$\{V, E\}$. V è un insieme, generalmente si tratta di un insieme finito ma non è obbligatorio, mentre E è un sottoinsieme di $V \times V$ (quindi in sostanza un insieme di **coppie ordinate** che indica i collegamenti, gli archi che collegano i diversi vertici. Dato che questo sottoinsieme proviene da un prodotto cartesiano la coppia è ordinata, quindi conta l'ordine in cui scrivo gli elementi). V è chiamato insieme dei vertici, E è chiamato insieme degli archi. Dal punto di vista della rappresentazione grafica, ogni elemento di V è rappresentato da un punto, l'elemento di E è rappresentato da un collegamento tra i due punti (su ciascun arco è riportata

una freccia che indica la direzione). Si noti anche che la notazione che prevede le parentesi tonde è tipica di una coppia ordinata. Il primo elemento di una coppia è il *nodo iniziale* di un arco, il secondo elemento è il *nodo terminale* dell'arco.

Nodi adiacenti o raggiungibili Dato un nodo v, w è detto adiacente a v se esiste un collegamento che va da v a w (si noti che se il collegamento andasse in direzione opposta allora v sarebbe adiacente a w, ma non sarebbe vero che w è adiacente a v). La definizione di adiacenza quindi non è biunivoca. In particolare si chiama **insieme di adiacenza** di un nodo v, l'insieme dei punti raggiungibili da v tramite un passo, ovvero quelli che hanno v come **primo elemento della coppia ordinata**.

Grado di un nodo

- Grado di entrata di un nodo: numero di archi entranti nel nodo
- Grado di uscita di un nodo: numero di archi uscenti dal nodo

9.2.2 Grafi non orientati

I grafi non orientati vengono genericamente indicati con il termine grafo, la definizione è sostanzialmente la stessa G$\{V, E\}$, con l'unica differenza che in questo caso E non è più sottoinsieme del prodotto cartesiano di V con se stesso ma è semplicemente un insieme di due elementi, entrambi contenuti in V. Graficamente sono rappresentati da nodi e archi che in questo caso non hanno indicazione del senso di percorrenza perché posso percorrerlo in qualunque direzione. Quando è chiaro che un grafo non è orientato, per alleggerire la notazione spesso si utilizzano comunque le parentesi tonde.

Insieme di adiacenza In questo caso l'insieme di adiacenza di un nodo è costituito da tutte quelle coppie che contengono il punto in questione. In questo caso la definizione di insieme di adiacenza è biunivoca. Se v è adiacente a w vale anche il viceversa.

Grado di un nodo In questo caso non si parla più di grado di entrata o di uscita di un nodo perché gli archi non sono più direzionati, si parla semplicemente di grado di un nodo e si indica con questo termine il **numero di nodi a cui è collegato**.

9.3 Cammini o cicli

Sia dato un grafo G, un cammino è una sequenza di archi che insistono su nodi adiacenti. Sequenza di archi che sono **ben congiunti** nei punti terminali. Ovviamente quando si è in presenza di un grafo diretto allora si parla di *cammino diretto*. Cammini: insiemi di archi **in E** che si congiungono **propriamente** (propriamente perché il termine assume significati diversi a seconda che il grafo sia diretto o no). Un cammino può essere indicato come sequenza di archi (sequenza di coppie eventualmente ordinate) o sequenza di nodi.

Si dice **lunghezza di un cammino** il numero di archi da cui è costituito (per ora assumiamo che tutti gli archi abbiano lo stesso peso, tutti i passi valgano 1).

Cammino semplice: Un cammino che non contiene archi *o* vertici ripetuti.

Ciclo o circuito: Cammino semplice in cui il nodo iniziale è uguale al nodo finale.

9.4 Rappresentazione di un grafo

Voglio una rappresentazione di grafo che sia direttamente implementabile. Un modo per rappresentare un grafo è grazie alla sua **Matrice di adiacenza** M_G. M_G è una matrice booleana che ha tante righe e colonne quanti sono i vertici del grafo. Ad esempio se un grafo ha N vertici allora M_G è una matrice $N \times N$, inoltre M_G è sempre una matrice booleana (ha come entries solo 0 e 1). Si ha che m_{ij} è non nullo solo se esiste un collegamento tra il nodo i e il nodo j.

Ovviamente devo stare attenta a riempire la matrice con i collegamenti corretti in un grafo diretto, **se il grafo non è diretto allora la matrice è simmetrica**[1].

Si chiama **cappio o self loop** un arco che connette un nodo a se stesso, i cappi sono possibili solamente sui grafi **diretti**. Quindi per un grafo **indiretto la traccia della matrice è sempre nulla**.

9.4.1 Vantaggi di rappresentazione mediante matrice di adiacenza

La matrice di adiacenza è un modo informaticamente conveniente di rappresentare un grafo (un array di array). Il vantaggio di avere una rappresentazione informatica di questo tipo è che una matrice (perché in generale lo sono gli array) facilmente gestibili. Questa rappresentazione è molto utile anche se il grafico è dinamico (a volte perdo dei collegamenti, altre volte li attivo). Per avere un grafo dinamico una matrice è comodissima perchè mi basta fare l'accesso all'entry corretta e cambiare il valore di attivazione da 0 a 1 o viceversa.

9.4.2 Contro di rappresentazione di matrice di adiacenza

Un contro dell'utilizzo di questa rappresentazione è che occupa troppo spazio (N^2) byte, è vantaggioso quando ho molti archi - la riempio quasi tutta - è svantaggioso quando invece il mio grafo ha pochi archi. Posso cercare di essere un po' più precisa per capire quando serve o meno. Posso cercare ad esempio di definire una soglia.

Dato un grafo di N vertici, posso avere al massimo $\frac{N(N-1)}{2}$ archi, è comunque dell'ordine di N^2. Nel caso di un di-grafo (grafico diretto) invece il numero massimo di collegamenti è esattamente N^2. Quindi ho sempre un'ordine di grandezza di N^2 oggetti; quando ho tanti archi quindi la matrice di adiacenza è una buona rappresentazione. Spreco spazio quando ho $o(N^2)$ archi, ad esempio quando ho un numero lineare di archi. Quando ho "così pochi" archi si parla di grafo sparso.

In questi casi la matrice di adiacenza non è conveniente da un punto di vista di spazio, da un punto di vista del tempo di può invece valutare, devo comunque mettere in conto almeno un tempo di inizializzazione di N^2, è comunque un buon tempo perché non ci sono molti algoritmi con questo tempo.

[1] Si noti che non è vero il viceversa, data una matrice di adiacenza simmetrica, non è detto che il grafo sia non orientato

9.4.3 Liste di adiacenza

Quando comunque la matrice di adiacenza è sovradimensionata rispetto alla variabile spaio allora si utilizza una rappresentazione in **liste di adiacenza**. Ad ogni componente metto la lista dei nodi adiacenti al nodo in questione (eg. componente 1 inserisco la lista di nodi adiacenti). Il lato positivo è che queste posso riempirle e farle crescere a piacere quindi non spreco spaio. In pratica rappresento i vertici e per ogni vertice rappresento la lista di adiacenze. Ad esempio se ho un grafo sparso, che so che continuerà a rimanere sparso mi conviene usare una lista di adiacenza.

Vantaggi lista di adiacenza Il vantaggio di una lista di adiacenze è che risparmio spazio, nel caso peggiore la dimensione però è sempre quadratica. Inoltre con una lista di adiacenza risparmio anche un po' sul tempo di inizializzazione.

Svantaggi lista di adiacenza In questo caso l'accesso non è assolutamente immediato, per arrivare a un collegamento devo passare tutta la lista di adiacenza. Fondamentalmente per questo devo stare attenta quando uso una lista di adiacenza.

Spesso la scelta tra matrice di adiacenza e lista di adiacenza dipende dal tipo di algoritmo con cui devo utilizzarla. Nella maggior parte dei casi si utilizzano le liste di adiacenza, più che altro perchè spesso i grafi che si devono rappresentare sono così grandi che la matrice di adiacenza non potrebbe nemmeno essere contenuta nella memoria. La rappresentazione con matrice di adiacenza è utile perché, a differenza della lista di adiacenza mi permette di risolvere una serie di problemi diversi sui grafi ma sempre con la stessa filosofia.

9.5 Problemi sui grafi

9.5.1 Raggiungibilità

Un problema tipico che viene affrontato con i gravi è quello della raggiungibilità. Ho un grafo e ho due nodi, s, p. La domanda è esiste un *cammino* da s a p (NB: non arco) che collega due vertici assegnati. Risolvere questo problema su un grafo significa in realtà risolvere una serie di problemi, può essere ad esempio "Esiste una dinamica di un sistema che lo porti da questa configurazione a quest'altra configurazione?" "Esiste una strada?".

Con la matrice di adiacenza questo problema si risolve elegantemente. Quello che faccio è **sfruttare la possibilità di moltiplicare tra se stesse matrici di adiacenza**. Quello che faccio però è cambiare la struttura algebrica su cui faccio questo prodotto, se normalmente lo mappo sul campo dei reali, in questo caso posso mappare sulla **struttura algebrica del semi-anello booleano**.

Semi-anello boolenao: caratterizzato dai seguenti elementi e dalle seguenti operazioni: $(\{0,1\}, \vee, \wedge, 0, 1)$.

A questo punto ho un oggetto booleano (matrice di adiacenza) e ho anche un modo di operare su di essa. A questo punto devo capire che cosa sia M_G^k ovvero M_G moltiplicata k volte per se stessa (dove la moltiplicazione è la moltiplicazione *and* booleana). Dato che sto lavorando sul semi-anello booleano quello che succede è che le operazioni mappano un elemento di questo semi-anello in un altro elemento

di questo semi-anello. Quindi devo chiedermi ora il significato della potenza k-esima della matrice M_G. Questa matrice k-esima mi dice a quali punti posso arrivare in k passi, l'elemento ij-esimo di M_G^k sarà 1 se e solo se esiste un cammino che va da i a j in k passi. è in sostanza una sorta di generalizzazione della matrice di adiacenza, la matrice di adiacenza mi parla di passi lunghi 1, la sua potenza k-esima mi parla di un cammino lungo k.

La dimostrazione si basa sull'induzione (dato che k varia sui naturali). La base di induzione (k=1) è vera perché coincide sostanzialmente con la definizione di matrice di adiacenza

Suppongo che sia vero per (k-1), questo è vero se e solo se $M_G^{(k-1)}$ contiene 1 se e solo se collega due nodi con un cammino di lunghezza k-1.

Dimostrazione finale: calcolo $M_G^k = M_G^{(K-1)} \times M_G$, guardo com'è fatto l'elemento ij della potenza k-esima in funzione delle altre due matrici.

NB: guardare la dimostrazione epr induzione sulle slide

Un esempio di algoritmo per studiare la raggiungibilità di un nodo è il seguente

```
set M_g;
k = M_g;
for i=1 to n do
    if k_st == 1 then
        return 1;
    end
    else
        k = k · M_G;
    end
end
return 0;
```

Algorithm 10: REACH(G,s,t)

Ll'obiettivo di questo algoritmo è quello dato un grafo G e due nodi, s e t, capire se c'è un percorso che li unisce In questo algoritmo utilizzo n come limite massimo per le iterazioni del ciclo `for` perché se dovessi considerare cammini con più di n passi su un algoritmo di n nodi, di fatto da n in avanti avrei semplicemente percorsi che connettono sempre gli stessi nodi ma che contengono anche cicli.

La complessità in tempo di questo algoritmo è $\mathcal{O}(n^4)$, è un tempo che si può migliorare, infatti esistono algoritmi che sono leggermente più veloci, anche se non di molto.

9.5.2 Problema di conteggio dei cammini

Ho sempre il mio grafo rappresentato dalla matrice, mi chiedo quanti cammini di lunghezza k congiungono due nodi. O viceversa (quello che affronto in questo caso). Fissato k, per ogni coppia di partenza e arrivo voglio sapere quanti cammini di lunghezza k collegano partenza e arrivo.

In questo caso per affrontare il problema sfrutto ancora la matrice di adiacenza M_G ma sul semi-anello $(\mathbb{N}, \cdot, +, 0, 1)$. Su tale semi-anello il prodotto di matrici avviene al solito come

$$A \cdot B = C \tag{9.1}$$

con

$$C_{ij} = \sum_{K=1}^{n} A_{ik} \cdot B_{kj} \tag{9.2}$$

Per affrontare il problema di conteggio si utilizza il seguente teorema: $(M_G^k)_{ij}$ contiene il numero di cammini di lunghezza k da `i` a `j` in G (G che ha n nodi). La dimostrazione di questo teorema è sulle slide della lezione 8.

Un esempio di algoritmo per il conteggio dei percorsi è il seguente

set M_G;
k=I;
for *1:=1 to k* **do**
  k = $k \cdot M_G$;
end
return k;

Algorithm 11: COUNT-PATH(G,k

L'algoritmo pernde come argomento il grafo G e k. In questo caso la complessità in tempo dell'algoritmo è $t(n) = \mathcal{O}(k \cdot n^3)$.

Quindi con la matrice di adiacenza ho risolto due problemi abbastanza diversi ma sempre con lo stesso algoritmo, sempre facendo dei prodotti tra matrici di adiacenza, il vantaggio delle matrici di adiacenza è che ho risolto, sostanzialmente con lo stesso approccio problemi tra loro molto diversi.

Osservazioni conclusive sull'uso della matrice di adiacenza Uno dei vantaggi dell'utilizzo della matrice di adiacenza è che ci da lo stesso approccio matriciale per rivolvere problemi diversi. E' sufficiente cambiare la struttura algebrica su cui sto lavorando; in oltre **si possono risolvere molti problemi scegliendo appropriatamente il semi-anello** su cui lavorare. Un altro vantaggio della rappresentazione matriciale è che gli algoritmi di soluzione del problema sono polinomiali (e in generale è probabile che si migliori). Inoltre, soprattutto, **un approccio matriciale porta a costruire algoitmi paralleli altamente efficienti**

9.6 Semi-anello booleano

Il semi-anello booleano è una struttura algebrica che insiste sul campo $\{0, 1\}$, le operazioni possibili sono $\vee, \wedge$, rispettivamente *or* e *and*. In pratica ho mappato l'operazione su un'algebra diversa.

Chapter 10

Lezione 8: Cammini minimi

Quando si parla di minimizzare dei cammini si parla di minimizzare il costo di un cammino. Quello che faccio è utilizzare un grafo a cui associo una funzione costo, generalmente a ciascun arco viene associato un costo (posso associarlo anche ad un nodo ma è meno usuale). Consideriamo costo un valore positivo (se ho anche valori negativi allora si tratta in realtà di un altro tipo di algoritmo) I costi, da un punto di vista di modellazione, la mia funzione costo può in realtà rappresentare molte cose diverse (eg. distanza tra due città, congestione di un tratto di rete) a seconda della situazione che ho di fronte il costo rappresenterà qualcosa di diverso. Il vantaggio è che lavoro con strumenti assolutamente matematici, indico con w_j il peso dell'arco j.

Naturalmente posso estendere la nozione di funzione costo anche ad un cammino, il costo di un cammino sarà data dalla somma di tutti i pesi. A volte però si utilizza anche come costo di un cammino il *prodotto* dei pesi. Quindi ho un **grafo pesato**, graficamente lo posso rappresentare come un grafo dove la funzione costo appare come etichetta sugli archi.

Matrice dei pesi è sostanzialmente uguale ad una matrice di adiacenza ma ha come entrate il peso di un determinato arco. In generale rimanere nello stesso posto ha peso nullo (ovvero il peso di un cappio è nullo) inoltre si ha la convenzione per cui **due nodi che non sono adiacenti hanno un costo infinito**. Le strutture su cui si lavora quindi sono grafi, spesso con questa rappresentazione.

10.1 Shortest path

Esistono diverse varianti di questo problema. Ora immaginiamo di avere un digrafo pesato (con un insieme di vertici - che immagino indicizzati), ho un insieme di archi e la funzione peso. Quello che voglio calcolare è il **cammino minimo tra ogni coppia di punti**, è una ricerca del cammino minimo **per ogni coppia di nodi**. Per semplicità di notazione ci concentriamo sull'individuare *il costo del cammino minimo*, non esattamente il cammino minimo, il nostro algoritmo con delle semplici modifiche consente anche di restituire il cammino minimo.

Per impostare il problema utilizziamo una notazione matriciale, l'idea è quella di partire dalla matrice dei pesi che lentamente modificherò fino ad arrivare a C_N. In sostanza costruirò una successione di matrici dei pesi fino ad arrivare a C_N

Significato del *valore della componente* c_{ij}^k è il costo del cammino minimo che collega i nodi i e j passando per nodi il cui indice è minore o uguale a k. Ho fissato

inizialmente un ordinamento dei vertici, quello che mostriamo è come calcolare la matrice C^k.

Su questa matrice dei pesi opererò facendo **operazioni sul semi-anello tropicale** definito come $(\mathbb{R}, min+, \infty, 0)$

A questo punto mi chiedo come calcolare c_{ij}^k conoscendo c_{ij}^{k-1}, ho a disposizione, per arrivare a questo risultato tutti i cammini minimi che passano per nodi fino a $k-1$. Un cammino che sfrutta il nodo k è un cammino è un cammino che arriva fino a k e poi va a j (senza ciclare su k). Il costo del cammino minimo quindi può

- contenere il nodo k (in questo caso cerco il minimo da v_k a v_j e lo attacco al minimo fino a k)
- non contenere il nodo k

Quello che succede è che o aggiungendo il nodo k mi si apre una nuova possibilità e quindi trovo un nuovo cammino minimo oppure il minimo è uno dei precedenti che era già contenuto nella matrice precedente. Quindi in pratica quello che devo fare è riempire step by step le matrici dei minimi fino ad arrivare al k fissato che ho chiesto. Tutto quello che devo fare è un prodotto di matrici.

Se anziché fare questa operazione per ogni coppia di nodi fisso il nodo di partenza ottengo ovviamente un algoritmo molto migliore.

10.2 Algoritmo di Dijkstra

`https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra`

10.2.1 Complessità in tempo dell'algoritmo di Dijkstra

Impiego un tempo dell'ordine di N^2, quindi il vantaggio è che a questo punto tanto vale dargli come input una matrice di adiacenza (anziché una lista). Tempo quadratico TOP.

10.2.2 Correttezza dell'algoritmo di Dijkstra

Ora devo mostrare che l'algoritmo è corretto, la dimostrazione si può fare per induzione sull'insieme s, posso ragionare sulla cardinalità di S. Mostro che **certe proprietà si mantengono al variare della cardinalità di S**

- $\forall v \in S$ contiene il costo del cammino più corto da v_0 a v e il cammino giace in S
- $\forall v \notin S$ il costo minimo tutto il cammino è in S tranne l'ultimo passo

Si dimostra per induzione che queste due proprietà sono sempre vere. Se reisco a dimostrare queste prorprietà ad ogni passo di ampliamento di S allora alla fine ho che per ogni v S contiene la lunghezza del cammino minimo da v0 a v.

Chapter 11

Lezione 9: Tecniche greedy per la soluzione di problemi

Le tecniche greedy sono tecniche sono utilizzate prevalentemente per risolvere **problemi di ottimizzazione**. L'idea di un algoritmo greedy è che ad ogni passo dell'algoritmo, tra le tante scelte, sceglie sempre quella che *localmente* sembra la migliore, senza considerare l'andamento generale del procedimento.

In inglese greedy significa (avido ingordo, poco lungimirante), è proprio così che funziona l'approccio greedy alla soluzione di un problema. Idea: ad ogni istante massimizzo una scelta locale, faccio la scelta che mi sembra migliore **in quel momento**, quindi **ottimizza localmente la tua scelta**. Un procedimento di questo tipo non è detto che porti ad una soluzione che sia ottima (non è detto che massimizzi la situazione generale, per questo è greedy: perché mi concentro su quello che accade in questo momento. è una filosofia facilissima, devo semplicemente prendere l'ottimo e di volta in volta andare avanti. Quindi sicuramente facile da capire e anche veloce, il calcolo di come procedere è rapido, non ho molte cose da fare. Sicuramente i pregi sono **velocità e semplicità**, un approccio di questo tipo ha anche dei difetti: in primis non è detto che funzioni (può portarmi ad una soluzione finale diversa quella che mi aspettavo). In generale non è vero che un approccio di questo tipo ci porti alla migliore soluzione.

Quindi, abbiamo visto che `GREEDY` non funziona sempre, però d'altra parte la filosofia su cui si basa è smeplice e veloce, quindi ci si può concentrare sul cercare di capire la risposta ad alcune domande:

- Quando `GREEDY` funziona?
- Se `GREEDY` non funziona, quanto distante può andare dall'ottimo?
- Qual è la complessità in tempo di un algoritmo `GREEDY`

L'algoritmo in certi casi non funziona però è indubbio che sia comunque un approccio facile (quello che posso valutare è, ad esmepio di quanto sbaglia, oppure posso cercare di capire a quali problemi può avere senso applicarlo). La domanda è quali sono le situazioni in cui greedy funziona? Quali sono i problemi in cui questo algoritmo funziona? Voglio stimare quanto lontano sono andata dalla soluzione ottima? Mi da magari non la soluzione ottima ma comunque qualcosa di ragionevole. A prescinedere da tutto, posso constatare che effettivamente un algoritmo veloce? Come valuto la complessità in tempo dell'algoritmo.

Individuare i casi in cui l'algoritmo funziona Per studiare queste proprietà è utile introdurre un framework matematico con valenza del tutto generale ma particolarmente importante in questo caso. Introduco l'ambiente algebrico in cui posso studiare gli algoritmi `GREEDY`.

Dato E, indico con 2^E l'insieme di tutti i possibili sotto insieme di E (l'insieme delle parti, o insieme potenza) di E. Suppongo di avere E di cardinalità finita, voglio sapere quanti elementi stanno nell'insieme potenza. Se E è di cardinalità N, l'insieme potenza ha cardinalità 2^N. Considero quindi delle particolari famiglie dell'insieme delle parti. $\mathcal{F}$ è un insieme di sottoinsiemi di E che deve essere un'**ideale d'ordine** rispetto alla inclusione ovvero. Se $A \in \mathcal{F}, B \in A \to B \in F$. Allora $\langle E, \mathcal{F} \rangle$ prende il nome **di sistema di indipendenza**.

In un sistema di indipendenza si modellano molti problemi di ottimizzazione. Posso definire quindi una funzione peso su E (la mia funzione costo).

$$\omega : E \to \mathbb{R}^+ \tag{11.1}$$

Quindi posso conoscere anche il costo dei sottoinsiemi, ad esempio, sia A un sottoinsieme di E, allora

$$\omega(A) = \sum_{x \in A} \omega(x) \tag{11.2}$$

Esempio di problema di ottimizzazione: ricerca dell'insieme più pesante in $\mathcal{F}$ A questo punto voglio trovare l'insieme più pesante in $\mathcal{F}$, quello che potrei fare è un algoritmo esaustivo, per ognuno di questi calcolo il peso. Avendo tutti i pesi prendo il maggiore e di conseguenza dò in output quell'insieme . quindi provo il peso di tutti gli elementi di $\mathcal{F}$, quindi devo fare almeno cardinalità di $\mathcal{F}$ di prove, quindi il tempo è esponenziale. Quindi in questo caso l'approccio esaustivo è impraticabile. Provo quindi a costruire un algoritmo `GREEDY` per la risoluzione del problema

```
S = ∅;
Q = E;
while Q ≠ ∅ do
    choose m (elemento più PESANTE IN Q) if S ∪m ∈ F then
        S = S ∪m ;
    end
    Q = Q - m;
end
return S ;
```

Algorithm 12: GREEDY(E, $\mathcal{F}, \omega$)

In sostanza greedy prende l'elemento più pesante e via dicendo, di volta in volta controllando che il sottoinsieme finale sia in $\mathcal{F}$.

Per prima cosa mi chiedo se l'algoritmo è corretto (minima richiesta per correttezza è che sia almeno una possibile soluzione, in questo caso sì perché $S \in \mathcal{F}$ - controllo ad ogni interazione - quindi sicuramente è soluzione. Poi mi interessa sapere la complessità in tempo. Sicuramente in queste iterazioni passo N oggetti, però in ciascuna di queste iterazioni faccio qualcosa di un attimo più complicato. Compio N volte l'iterazione, però all'interno della quale ho la ricerca id un massimo. Inoltre devo soddisfare la condizione che s dia in $\mathcal{F}$. Quindi come passo preliminare posso ordinare Q (spostare gli elementi in un array e a quel punto posso fare davvero n passi.

NB: per complessità in tempo vedere slide della lezione

11.1 Quando GREEDY funziona

Per rispondere a questa domanda devo, per prima cosa caratterizzare la classe dei sistemi di indipendenza su cui GREEDY ha successo sempre.

Matroidi Questo è uno di quei casi in cui matematica e informatica si vede che hanno una le radici nell'altra. Un matroide è un sistema di indipendenza con una particolarità in più, deve valere la proprietà di scambio: uno dei due insiemi di $\mathcal{F}$ ha un elemento in più dell'altro, cardinalità dell'altro +1. Se aggiungo al secondo insieme questo +1 ottengo un nuovo elemento di $\mathcal{F}$. Un sistema di indipendenza in cui è sempre vera questa proprietà di scambio si chiama matroide. Ovvero, un sistema di indipendenza $\langle E, \mathcal{B}, \rangle$ è un matroide se, dati $A, B \in \mathcal{F}$ con $|B| = |A| + 1$ esiste almeno un elemento $b \in B - A$ tale che $A \cup b \in \mathcal{F}$

I matroidi sono gli insemi di indipendenza su cui gli algoritmi greedy hanno successo.

Il teorema di Rado fornisce una famiglia di situazioni su cui posso far funzionare l'algoritmo greedy. consideriamo un sistema di indipendenza. Le affermazioni A e B sono equivalenti. **Per ogni** funzione peso greedy funziona sul matroide.

NB: questo non significa che greedy funzioni solo su un matroide. Magari ci sono alcuni casi in cui sto lavorando, non su un matroide ma greedy mi dà la soluzione ottima. Se riesco a dimostrare che questo è un matroide allora mi dà l'ottimalità altrimenti no.

11.1.1 Teorema di Rado

Dato un sistema di indipendenza $\langle E, \mathcal{F} \rangle$ le due affermazioni sono equivalenti:

- per ogni funzione peso $\omega : E \to \mathbb{R}^+$ GREEDY ha successo su $\langle E, \mathcal{F}, \omega \rangle$
- $\langle E, \mathcal{F} \rangle$ è un matroide

Dimostrare che non B implica non A (uguale a A implica B). Dimostro che se B non è un matroide allora riesco ad esibire una funzione peso w per cui greedy non funziona su quella struttura. A questo punto posso regolare alfa tc w(b) maggiore di w(s) S sarà l'ottimo?

Chapter 12

Lezione 10: Tecnica GREEDY per il minimum spanning tree

12.1 Alberi

Dato un grafo non orientato $G = (V, E)$, G è **connesso** *se e solo se* qualunque coppia di nodi in G è collegata da un cammino. Quindi è connesso se non ci sono nodi isolati.

Inoltre G è **privo di cicli** se non ci sono cicli.

Definizione di Alberi $G = (V, E)$ non orientato è un **albero** se è connesso e privo di cicli.

Un albero con radice è un albero particolare in cui individuo un nodo particolare chiamato radice.

12.1.1 Alcune proprietà degli alberi

1. Un albero di n nodi possiede *n-1* archi
2. Tra ogni coppia di nodi un albero esiste un cammino
3. Se ad un albero aggiungo un arco creo un ciclo
4. Se ad un albero tolgo un arco lo disconnetto

Foresta: Una foresta è una collezione di alberi

Definizioni equivalenti di foresta

1. Una foresta è un grafo non orientato aciclico
2. Una foresta è una collezione di archi che non formano cicli

12.1.2 Alberi di copertura

Dato G=(V,E) non orientato, definiamo **sottografo** un qualsiasi G'=(V,E') con $E' \subseteq E$.

Da qui in avanti, quando si parlerà di grafi si parlerà solamente di grafi connessi. Dato G=(V,E) un albero di copertura (o di sostegno, in inglese **spanning tree** è un sottografo di G che sia un albero).

Un albero di copertura è il modo *più economico in termini di archi* per garantire comunque la connessione tra ogni coppia di nodi nel grafo G (più economico proprio per le proprietà degli alberi).

12.2 Il problema del minimum spanning tree

Input: G=(V,E) non diretto, $\omega E \to \mathbb{R}^+$ funzione peso
Output: Minimo spanning Tree pre G
Costo: dato l'albero T, $\omega(T) = \sum_{l \in T} \omega(l)$
Questo problema ha moltissime ricadue pratiche eg. costruzione di reti di telecomunicazione, di approvvigionamento e distribuzione, costruzione di reti di smistamento, attenzione all'affidabilità e alla valutazione dei guasti.

12.2.1 Tecniche GREEDY per minimum spanning

Sia G=(V,E); $\omega : E \to \mathbb{R}^+$; —V—=n.
. Step:

1. Un algoritmo GREEDY passo dopo passo sceglie l'arco **e** di costo minimo rimasto.

2. Consideralo col set S di archi che hai collezionato fino a questo momento:

   (a) Se $S \cup e$ non crea cicli allora lo tengo, quindi lo aggiungo al set S

   (b) Se $S \cup e$ crea cicli allora elimino quell'elemento che avevo cercato di aggiungere e S rimane quello precedente

3. Continua finché non avrai collezionato n-1 archi in S

A questo punto l'output dell'algoritmo è S, il minimum spanning tree.

```
S = {};
SORT(E);
for i=1 to m do
    if S∪ E[i] non contiene cicli then
        S = S ∪ E[i] ;
    end
end
return S;
```

Algorithm 13: Kruskal(G=(V,E), ω

nota: m è la cardinalità di E

Correttezza dell'algoritmo Per la dimostrazione della correttezza si usa il teorema di Rado, infatti, dato $G = (V, E)$ considera il sistema di indipendenza (E,F) con F = sottoinsiemi di E che non formano cicli. Si può dimostrare che Eì(E,F) è un matroide, di conseguenza, per il teorema di Rado, l'algoritmo di Kruskal è corretto per ogni funzione peso.

Complessità in tempo dell'algoritmo Per prima cosa devo tenere conto di sort che ha complessità in tempo $m \log m$, dopo di che esistono strutture dati per cui il test sui cicli può essere controllato in tempo $\log m$.

Quindi la complessità dell'algoritmo è $\mathcal{O}(m \log m)$