Fun with algorithms
PCTO di Sapienza Università di Roma Edizione 2023/2024

Tiziana Calamoneri – Angelo Monti
calamo@di.uniroma1.it, monti@di.uniroma1.it

Questa dispensa ha un duplice scopo.
Innanzitutto, offre un'introduzione breve ed informale ai concetti di algoritmo e di efficienza di una computazione: cercheremo di chiarire cosa sono gli algoritmi, perché è utile studiarli, qual è il loro ruolo nel panorama delle scienze informatiche. Descriveremo inoltre alcuni algoritmi basilari, già presentati durante l'incontro introduttivo, corredati da link a video che ne visualizzano il funzionamento.
Nella seconda parte, la dispensa descrive un problema ed una sua possibile soluzione algoritmica. Ogni gruppo di studenti dovrà proporne una visualizzazione realizzando un video.

Font ad alta leggibilità biancoenero®di biancoenero edizioni srl, disegnata da Umberto Mischi. Disponibile gratuitamente per chi ne fa un uso non commerciale. www.biancoeneroedizioni.com

1. Istruzioni per lo svolgimento e la consegna del progetto

- Il video deve essere completato e caricato su YouTube improrogabilmente entro il

  7 maggio 2024

- Contestualmente, occorre inviare una mail ai docenti con subject

  PCTO 2023 – Fun with Algorithms – *

  rimpiazzando "*" con l'elenco dei cognomi degli studenti del gruppo di lavoro (max 5 studenti per gruppo). Nel corpo della mail, riportare l'URL del video caricato su YouTube e il nome della scuola.

- Il video deve essere condiviso in modalità pubblica (in tal caso sarà visibile a tutti) o non in elenco (in tal caso potrà essere essere visto e condiviso solo da chi abbia il link).

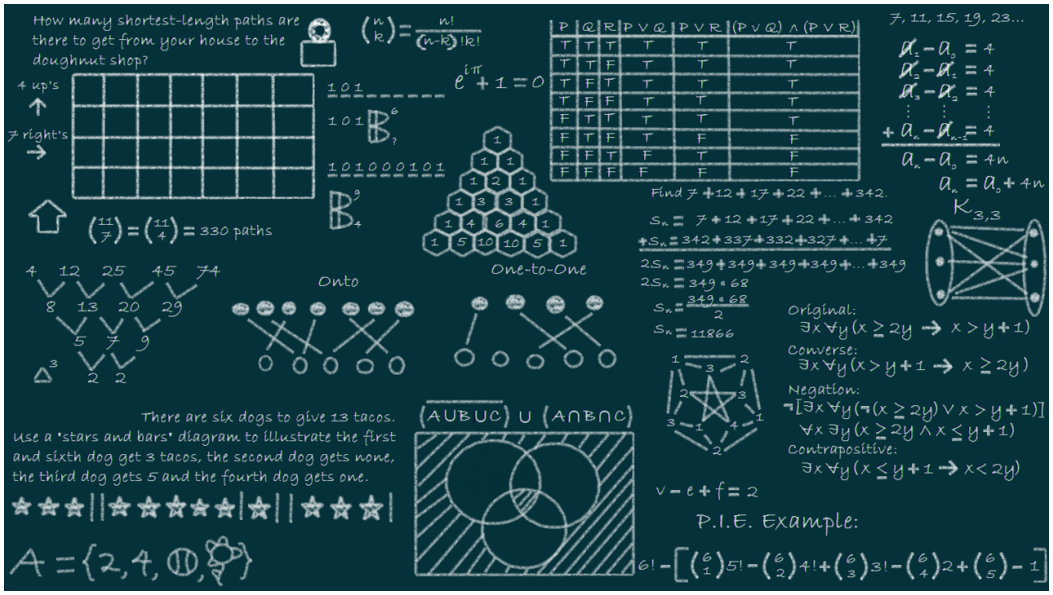
- Se per motivi di privacy non volete che il vostro video sia elencato nella pagina web del PCTO "Fun with Algorithms", specificatelo nella mail.

- La durata del video non deve superare i 5 minuti.

- Il numero di ore che vi verrà riconosciuto (massimo 40) sarà legato alla qualità del video prodotto, in termini di chiarezza della visualizzazione, originalità e precisione realizzativa.

Parte 1

Una breve introduzione agli algoritmi



2. Introduzione

Cosa è l'informatica? Senza pretendere di dare una risposta precisa ed esaustiva, limitiamoci ad osservare che i computer e molte altre tecnologie informatiche permeano la nostra vita quotidiana, sia in modo direttamente percepibile (ad esempio mentre navighiamo in Internet dal nostro PC) sia in modo invisibile (ad esempio quando usiamo tecnologie "embedded", ossia "immerse" dentro un'automobile, un elettrodomestico, un telefono). Non solo, apparecchiature da cui dipendono le vite di molte persone (come aeroplani di linea ed apparecchiature mediche) sono sempre più informatizzate, il che rende di cruciale importanza evitare errori e possibili malfunzionamenti. Senza l'informatica, la società contemporanea - così come la conosciamo - non esisterebbe.

Spesso - e a torto - l'informatica viene considerata una mera attività pratica, per svolgere la quale è sufficiente un approccio dilettantistico, senza una vera professionalità. Nulla di più inesatto! Si tratta di una disciplina scientifica di cui i calcolatori rappresentano solo uno strumento: l'informatico può anche lavorare solamente con carta e penna! Dal dizionario Devoto-Oli della lingua italiana:

> "L'informatica è la scienza che consente di ordinare, trattare e trasmettere l'informazione attraverso l'elaborazione elettronica..."

In questa definizione vogliamo evidenziare due parole chiave: informazione e scienza. In effetti, la pervasività dell'informatica deriva dal fatto che in qualunque tipo di attività umana è necessario gestire informazioni di un qualche tipo, che vanno memorizzate ed elaborate con precisione. Si tratta di attività rigorose, sistematiche e che richiedono un approccio scientifico.

3. Algoritmi, problemi computazionali ed efficienza

Cos'è un algoritmo?

La definizione di informatica proposta dall'ACM (Association for Computing Machinery), una delle principali organizzazioni scientifiche di informatici di tutto il mondo, è la seguente:

> "L'informatica è la scienza degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione ed applicazione."

Iniziamo dunque con l'introdurre un concetto fondamentale, centrale nell'informatica, quello di algoritmo.

Un algoritmo è "una sequenza di comandi elementari ed univoci che terminano in un tempo finito ed operano su strutture dati".

Un comando è elementare quando non può essere scomposto in comandi più semplici; è univoco quando può essere interpretato in un solo modo.

Ad esempio, domandiamoci se la seguente ricetta per fare un uovo al tegamino è un algoritmo:

1. Rompere un uovo in padella.
2. Cuocere l'uovo.

Il passo 1 non è né elementare né univoco in quanto, dopo aver rotto l'uovo (cosa che già è possibile fare in vari modi diversi), si deve separare il guscio dal resto. Il passo 2 non è elementare: l'attività di cottura si può scomporre nell'accensione del fornello, nell'aggiunta del condimento e del sale, prevede il controllo della cottura, ecc.

Se un algoritmo è veramente tale, e quindi è ben specificato, chi (o ciò che) lo esegue non ha bisogno di "pensare": deve solo eseguire con precisione i passi elencati, nella sequenza in cui appaiono.

E infatti un calcolatore esegue pedissequamente tutte le operazioni elencate negli algoritmi pensati (ossia progettati) da un essere umano. Se si verifica un errore e il risultato è sbagliato, l'errore non è del calcolatore, ma di chi ha progettato l'algoritmo!

Input e output. Un algoritmo è dunque una procedura di calcolo ben definita che prende un certo valore - o un insieme di valori - come input e genera un valore - o un insieme di valori - come output. In altre parole, è una sequenza di passi computazionali che trasforma l'input in output.

Il procedimento deve essere descritto in modo preciso e non ambiguo, in modo da poter essere eseguito "meccanicamente", cioè applicando le regole senza pensare al loro significato, e senza alcuno spazio per la creatività e l'interpretazione personale.

Programmi.

Vi sono molti modi per descrivere un algoritmo: l'algoritmo per la divisione in colonna imparato alle elementari è perfettamente specificato a parole. Nell'informatica, si usano i linguaggi di programmazione: un algoritmo, quando viene espresso in un linguaggio di programmazione, diventa un programma, eseguibile da un computer.

Problemi computazionali.

Possiamo considerare un algoritmo come uno strumento per risolvere un problema computazionale ben definito. La definizione del problema specifica in termini generali la relazione di input/output desiderata. L'algoritmo descrive una specifica procedura per ottenere tale relazione di input/output.
Ad esempio, supponiamo di dovere ordinare una sequenza di numeri in ordine non decrescente. Questo problema si presenta spesso nella pratica. Il problema dell'ordinamento può essere formalmente definito nel seguente modo:

<u>Input</u>: una sequenza di n numeri$\langle a_1, a_2, \ldots, a_n \rangle$
<u>Output</u>: una permutazione (riordinamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Per esempio, data la sequenza di input ⟨31, 41, 59, 26, 41, 58⟩, un algoritmo di ordinamento restituisce come output la sequenza ⟨26, 31, 41, 41, 58, 59⟩.

La sequenza di input è detta *istanza* del problema dell'ordinamento. In generale, l'istanza di un problema è costituita dall'input - che soddisfa eventuali vincoli imposti nella definizione del problema - richiesto per calcolare una soluzione del problema.

L'ordinamento è un'operazione fondamentale in informatica (molti algoritmi la usano come passo intermedio), e per questo sono stati sviluppati numerosissimi algoritmi di ordinamento, con diverse caratteristiche. La scelta dell'algoritmo più appropriato per una data applicazione dipende – fra l'altro – dal numero di elementi da ordinare, dal livello di ordinamento iniziale degli elementi, da eventuali vincoli sui valori degli elementi e dal tipo di unità di memorizzazione da utilizzare (memoria principale, dischi o nastri).

Tornando alle definizioni fondamentali, diremo che un *algoritmo* è *corretto* se, per ogni istanza di input, termina con l'output corretto. Diremo inoltre che un algoritmo corretto *risolve il problema* computazionale dato. Un algoritmo errato potrebbe non terminare affatto, su qualche istanza di input, o potrebbe terminare fornendo una soluzione errata, ossia diversa da quella desiderata.

Efficienza

Se i computer fossero infinitamente veloci e la memoria dei computer fosse gratuita, ci sarebbe ancora qualche motivo studiare gli algoritmi? La risposta è si, se non altro perché vorremmo ugualmente dimostrare che il nostro metodo di risoluzione termina e fornisce la soluzione esatta.

Se i computer fossero infinitamente veloci, qualsiasi metodo corretto per risolvere un problema funzionerebbe egregiamente. Probabilmente, vorremmo che la nostra implementazione rispettasse le buone norme dell'ingegneria del software (ovvero fosse ben progettata e documentata), ma il più delle volte adotteremmo il metodo più semplice da implementare. Ovviamente, i computer possono essere veloci, ma non infinitamente veloci! E la memoria può costare poco, ma non essere gratuita! Il tempo di elaborazione e lo spazio richiesto in memoria sono in pratica risorse limitate, che devono essere saggiamente utilizzate: gli algoritmi che sono efficienti in termini di tempo e spazio ci aiuteranno a tal scopo.

Algoritmi progettati per risolvere lo stesso problema spesso sono notevolmente diversi nella loro efficienza. Queste differenze possono essere molto più significative di quelle dovute all'hardware ed avere un impatto ben maggiore.

Consideriamo, ad esempio, il problema del calcolo della somma dei primi n interi.
Possiamo pensare ad un primo algoritmo fatto così:

```
Funzione Calcola_Somma_1(n: intero)
1     somma ← 0                          1 operazione
2     for i = 1 to n do
3           aggiungi i a somma           n operazioni
4     stampa somma                       1 operazione
```

Quindi, il costo computazionale di questo algoritmo è proporzionale al numero dato in input n.
Osserviamo, tuttavia, che lo stesso problema può essere risolto in modo ben più efficiente come segue:

```
Funzione Calcola_Somma_2(n: intero)
1       somma ← n (n+1)/2                1 operazione
2       stampa somma                     1 operazione
```

Il costo della funzione è, ovviamente, costante e non dipende da quanto è grande n; perciò, è decisamente migliore rispetto al precedente.

Algoritmi efficienti o computer veloci?

Come esempio concreto, mettiamo a confronto un computer veloce (computer V) che esegue insertion sort e un computer lento (computer L) che esegue merge sort. Entrambi devono ordinare un elenco di un milione (10^6) di numeri. Supponiamo che il computer V esegua un miliardo di istruzioni al secondo e il computer L esegua soltanto 10 milioni di istruzioni al secondo, ovvero il computer V è 100 volte più veloce del computer L in termini di potenza di calcolo. Per rendere la differenza ancora più evidente, supponiamo che il miglior programmatore del mondo abbia codificato insertion sort nel linguaggio macchina del computer V e che il codice risultante richieda $2 \cdot n^2$ istruzioni per ordinare n numeri (in questo caso, $c_1 = 2$). Merge sort, invece, è stato programmato per il computer L da un programmatore medio con un linguaggio di alto livello ed un compilatore inefficiente; il codice risultante richiede $50 \cdot n \log_2 n$ istruzioni ($c_2 = 50$). Per ordinare un milione di numeri, il computer V con insertion sort impiega:

$$\frac{2(10^6)^2 \, istruzioni}{10^9 \, istruzioni / \sec} = 2000 \sec$$

Il computer L con merge sort impiega invece:

$$\frac{50 \cdot 10^6 \log_2 10^6 \, istruzioni}{10^7 \, istruzioni / \sec} \approx 100 \sec$$

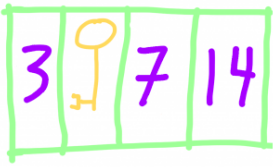
Utilizzando un algoritmo il cui tempo di esecuzione cresce più lentamente, perfino con un compilatore scadente, l'esecuzione sul computer L è 20 volte più veloce di quella sul computer V! Il vantaggio di merge sort è ancora più significativo se ordiniamo dieci milioni di numeri: insertion sort impiega più di 2 giorni, merge sort meno di 20 minuti. In generale, al crescere della dimensione del problema, il vantaggio relativo di merge sort aumenta considerevolmente.

L'esempio mostra che gli algoritmi, come l'hardware, sono una tecnologia. Le prestazioni globali di un sistema dipendono tanto dalla scelta di algoritmi efficienti quanto dalla scelta di un hardware veloce. Analogamente a quanto è avvenuto in altri rami dell'informatica, anche la progettazione di algoritmi efficienti ha avuto un rapido progresso.

Qualcuno potrebbe chiedersi se gli algoritmi siano davvero così importanti nei moderni calcolatori, alla luce di altre tecnologie apparentemente più avanzate. La risposta è ancora si! Sebbene ci siano applicazioni che non richiedano esplicitamente un contenuto algoritmico a livello dell'applicazione (per esempio, alcune semplici applicazioni web), molte richiedono una dose non indifferente di codice algoritmico. Per esempio, considerate un servizio web che determina come spostarsi da un sito all'altro. La sua implementazione fa affidamento su un hardware veloce, un'interfaccia grafica utente, una rete WAN e, possibilmente, anche su sistemi orientati agli oggetti. Ma richiede anche algoritmi per svolgere determinate operazioni che sono cruciali nell'applicazione, ad esempio trovare i percorsi più rapidi tra due luoghi, rappresentare le mappe ed interpolare gli indirizzi.

In sintesi, gli algoritmi sono il cuore delle principali tecnologie utilizzate nei moderni calcolatori. Grazie alle loro crescenti capacità di elaborazione e memorizzazione, i calcolatori vengono utilizzati per risolvere problemi sempre più complicati. Avere una solida base di conoscenza degli algoritmi e delle tecniche alla base della loro progettazione è una caratteristica che contraddistingue i programmatori esperti dai principianti: con i moderni calcolatori, potete svolgere alcuni compiti senza sapere molto di algoritmi, ma con una buona conoscenza degli algoritmi, potete fare molto, molto di più!

Un esempio: il problema della ricerca



Vedremo ora un altro problema in cui due diversi algoritmi risolutivi hanno una diversa efficienza, ma in questo esempio, rispetto al problema dell'ordinamento, succede qualcosa di nuovo: per poter utilizzare uno degli algoritmi, abbiamo bisogno che l'input rispetti una certa proprietà.

Il problema è quello di cercare un elemento in un elenco.

Ricerca sequenziale

Se l'elenco o sequenza di elementi in cui cercare non è ordinato (secondo il criterio di ricerca), si è costretti ad eseguire una ricerca sequenziale: a partire dal primo elemento della sequenza si esaminano uno dopo l'altro tutti gli elementi, finché o si trova l'elemento cercato o si arriva al termine della sequenza senza aver trovato ciò che si cerca.

Alcuni esempi più specifici sono i seguenti:

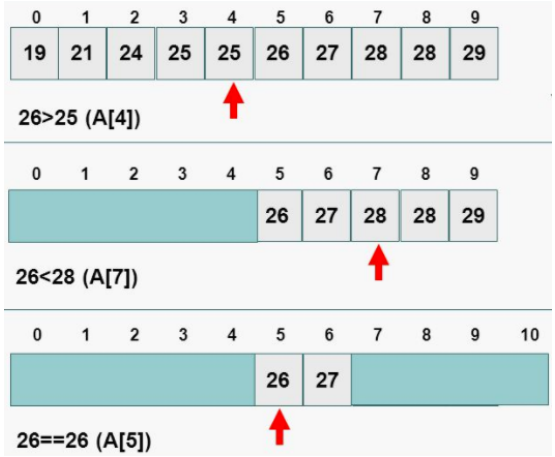
- cercare una parola in una lunga lista non ordinata di parole;
- cercare il prezzo di un prodotto in un lungo scontrino della spesa di un ipermercato;
- cercare in un elenco telefonico a chi corrisponda un dato numero (nell'elenco gli elementi sono ordinati per nome, non per numero).

Quanti passi si fanno in media, con la ricerca sequenziale, per trovare l'elemento cercato in una sequenza di n elementi? Se si è molto fortunati lo si trova subito; se si è sfortunati lo si trova in ultima posizione oppure non lo si trova, e in entrambi i casi si devono fare n passi.
In media si faranno quindi circa n/2 passi, cioè un numero di passi proporzionale ad n o, più precisamente, lineare in n. Quindi anche il tempo medio necessario per effettuare la ricerca è proporzionale a n. Se la lunghezza della lista raddoppia, il tempo raddoppia; se la lunghezza triplica, il tempo triplica, e così via. Ciò vale tanto per gli esseri umani che per i computer.

Ricerca binaria o dicotomica

Se la sequenza di parole è ordinata alfabeticamente e il tempo di accesso a un suo elemento è indipendente dalla posizione dell'elemento nella sequenza stessa (ed è piccolo), allora si può usare la ricerca binaria. Ad esempio, una persona può aprire un dizionario circa alla pagina voluta senza dover sfogliare una per una tutte le pagine precedenti.

Una possibile descrizione a parole, imprecisa, dell'algoritmo usato è la seguente:



Esempio di applicazione dell'algoritmo di ricerca binaria: il valore cercato è pari a 26

Precondizione: la sequenza (ad accesso diretto) è ordinata

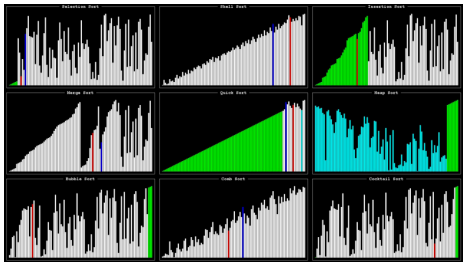
1. Si accede all'elemento centrale della sequenza e lo si confronta con l'elemento da cercare; a seconda del risultato del confronto ci si restringe a considerare solo la prima o la seconda metà;
2. si confronta l'elemento da cercare con l'elemento centrale della metà selezionata, e di conseguenza ci si restringe a una porzione di lunghezza dimezzata;
3. si ripete poi il procedimento.

Quanti passi occorrono per cercare un elemento per mezzo della ricerca binaria in un elenco di n elementi? Si osservi che, ad ogni passo, la porzione di elenco in cui cercare si riduce all'incirca alla metà. Quindi, se ad esempio è $n = 2^{10} = 1024$, al secondo passo la porzione in cui cercare è di circa 512, al terzo di circa 256, e così via. Nel caso peggiore dopo circa $10 = \log_2 2^{10}$ passi si trova l'elemento, o si stabilisce che esso non è presente.
Dietro alla grande velocità della ricerca binaria rispetto alla ricerca sequenziale, riscontrabile anche nella vita di tutti i giorni nelle ricerche

manuali o visive, c'è il diverso tasso di crescita delle funzioni logaritmiche rispetto alle funzioni lineari.

Concludiamo questo argomento con una curiosità. Abbiamo appena fornito prima un algoritmo che ha bisogno di un numero di operazioni lineare nella dimensione dell'insieme in cui cercare e poi uno, molto più efficiente, a cui servono solo un numero logaritmico di operazioni. Spostiamoci ora in un campo completamente diverso. Sia data una lunga sequenza di DNA, e ci chiediamo se un certo frammento sia presente in esso o no. Questo può essere fatto usando la così detta ibridizzazione, cioè il processo che unisce due eliche singole di DNA in un'unica elica doppia. E' noto infatti che frammenti di singole eliche si attraggono e formano un'elica doppia se essi sono uno il complementare dell'altro (cioè, molto semplicisticamente, se ad ogni nucleotide A corrisponde un nucleotide T e viceversa, e ad ogni nucleotide C corrisponde un nucleotide G e viceversa). Per questo scopo, i biologi usano le sonde, piccoli frammenti di eliche singole di DNA che hanno una sequenza nota e sono fluorescenti. L'avvenuta ibridizzazione di una sonda in un frammento di DNA sconosciuto evidenzia la presenza della sequenza complementare a quella della sonda nel frammento. Considerando l'ibridizzazione come un singolo passo computazionale su una sorta di ipotetica macchina molecolare, potremmo dire che questo è un algoritmo di ricerca che si può eseguire con costo costante...

Un altro esempio: il problema dell'ordinamento



Il problema dell'ordinamento degli elementi di un insieme è un problema molto ricorrente in informatica poiché ha un'importanza fondamentale per le applicazioni: lo si ritrova molto frequentemente come sottoproblema nell'ambito dei problemi reali. In effetti si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di algoritmi di ordinamento.
Un algoritmo di ordinamento è un algoritmo capace di ordinare gli elementi di un insieme sulla base di una certa relazione d'ordine, definita sull'insieme stesso (maggiore/minore, precede/segue, ecc.).
Tutti gli algoritmi che vedremo hanno alcuni aspetti in comune. Si basano su due tipi fondamentali di operazioni: il confronto fra due elementi e lo scambio di due elementi. Questa ipotesi di lavoro è necessaria per non costruire soluzioni irrealistiche (ad esempio, definendo una operazione che in un sol passo ordina un numero di elementi non limitato da una costante). Inoltre, per semplicità di trattazione, si suppone che gli n elementi da ordinare siano numeri interi e siano contenuti in un vettore i cui indici vanno da 1 a n.
Tuttavia, nei problemi reali, i dati da ordinare sono ben più complessi: in generale essi sono strutturati in record, cioè in gruppi di informazioni non sempre omogenee relative allo stesso soggetto (ad esempio, l'archivio studenti ha moltissimi record, ciascuno contenente molte informazioni relative ad un singolo studente), e si vuole ordinarli rispetto ad una di tali informazioni (ad esempio il cognome, oppure il numero di matricola, ecc.).

Algoritmi semplici

Di algoritmi di ordinamento ne esitono diversi e di diversi tipi. In un'ottica di progetto di algoritmi efficienti, illustreremo dapprima algoritmi di ordinamento concettualmente semplici ma non molto efficienti, per renderci poi conto del fatto che –volendo migliorare l'efficienza- sarà necessario mettere in campo idee meno banali.

L'algoritmo <u>insertion sort</u> (ordinamento per inserzione) si può intuitivamente assimilare al modo in cui una persona ordina un mazzo di carte. Partendo da un mazzo vuoto inseriamo una carta alla volta,

cercando il punto giusto dove inserirla. Alla fine il mazzo di carte risulta ordinato.
Nel caso dell'algoritmo insertion sort gli elementi da ordinare sono inizialmente contenuti nel vettore quindi non si parte da un insieme vuoto ma si deve trovare la posizione giusta di ogni elemento (è come se si volessero ordinare le carte tenendole in mano, anziché appoggiandole e prendendole poi una per una). Ciò si effettua "estraendo" l'elemento così da liberare la sua posizione corrente, spostando poi verso destra tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso ed, infine, inserendo l'elemento nella posizione che si è liberata. Alla fine del procedimento il vettore risulterà ordinato.
Il numero di operazioni svolte da questo algoritmo per ordinare n elementi varia a seconda dell'input: esso è infatti lineare in n nel caso migliore, quando il numero di spostamenti da effettuare è minimo e cioè quando il vettore è già ordinate; è invece quadratico in n quando il numero di spostamenti da effettuare è massimo e cioè quando il vettore in partenza è ordinato all'incontrario.
In casi come questo, poiché vogliamo avere una stima di quanto tempo serve per eseguire l'algoritmo in questione, ci concentriamo sul caso peggiore, certi che l'algoritmo non potrà impiegare un tempo maggiore.

L'algoritmo <u>selection sort</u> (ordinamento per selezione) funziona nel seguente modo: cerca il minimo dell'intero vettore e lo mette in prima posizione (con uno scambio, fatto anche se inutile); quindi cerca il nuovo minimo nel vettore restante, cioè nelle posizioni dalla seconda all'ultima incluse, e lo mette in seconda posizione (con uno scambio), poi cerca il minimo nelle posizioni dalla terza all'ultima incluse e lo mette in terza posizione, e così via.
Rispetto all'insertion sort, la principale differenza è che, per sistemare definitivamente un singolo elemento, si fa un solo scambio. Quindi, se lo scambio è una operazione costosa, selection sort è meglio di insertion sort.
Il costo di questo algoritmo non dipende in alcun modo dalla distribuzione iniziale dei dati nel vettore, cioè resta la stessa (quadratic in n) sia che il vettore sia inizialmente ordinato o no.

L'algoritmo <u>bubble sort</u> (ordinamento a bolle) ha un funzionamento molto semplice: l'algoritmo ispeziona una dopo l'altra ogni coppia di elementi adiacenti e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati. L'algoritmo prosegue fino a che non vi sono più coppie di elementi adiacenti nell'ordine sbagliato.
Anche in questo caso, il numero di operazioni che vengono eseguite dall'algoritmo è quadratico in n.

Alcune visualizzazioni:

Insertion Sort:
https://www.youtube.com/watch?v=ROalU379l3U
https://www.youtube.com/watch?v=DFG-XuyPYUQ
https://www.youtube.com/watch?v=OGzPmgsI-pQ
https://www.youtube.com/watch?v=kPRAOW1kECg (0:10)

Selection Sort:
https://www.youtube.com/watch?v=Ns4TPTC8whw
https://www.youtube.com/watch?v=f8hXR_Hvybo
https://www.youtube.com/watch?v=xWBP4lzkoyM
https://www.youtube.com/watch?v=kPRAOW1kECg (0:00)

Bubble Sort:
https://www.youtube.com/watch?v=lyZQPjUT5B4
https://www.youtube.com/watch?v=8Kp-8OGwphY
https://www.youtube.com/watch?v=nmhjrI-aW5o
https://www.youtube.com/watch?v=kPRAOW1kECg (4:00)

Un algoritmo efficiente: il merge sort

L'algoritmo mergesort (ordinamento per fusione) è un algoritmo ricorsivo che adotta una tecnica algoritmica detta divide et impera. Essa può essere descritta come segue:

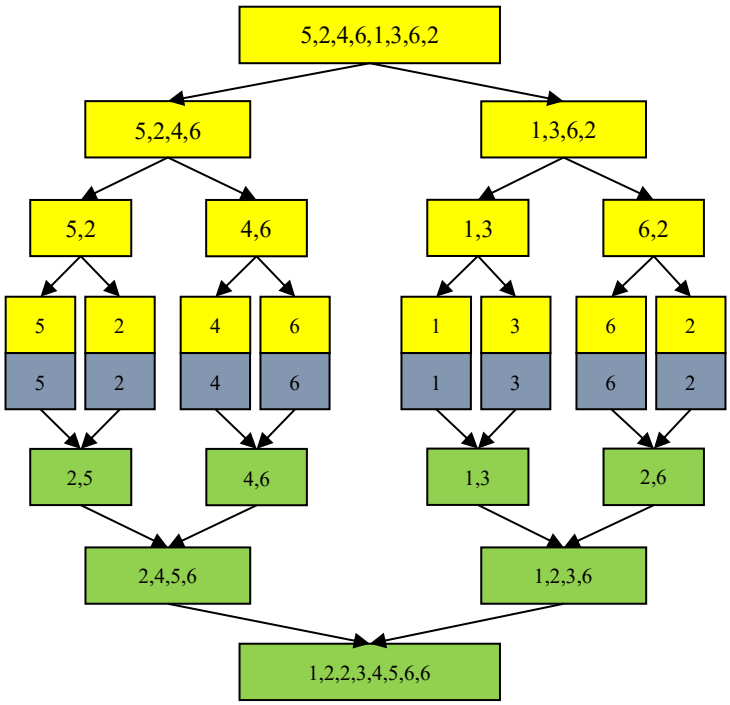
- il problema complessivo si suddivide in sottoproblemi di dimensione inferiore (divide);
- i sottoproblemi si risolvono ricorsivamente (impera);
- le soluzioni dei sottoproblemi si compongono per ottenere la soluzione al problema complessivo (combina).

Intuitivamente il mergesort funziona in questo modo:

- divide: la sequenza di n elementi viene divisa in due sottosequenze di n/2 elementi ciascuna;
- impera: le due sottosequenze di n/2 elementi vengono ordinate ricorsivamente;
- passo base: la ricorsione termina quando la sottosequenza è costituita di un solo elemento, per cui è già ordinata;

- combina: le due sottosequenze – ormai ordinate – di n/2 elementi ciascuna vengono "fuse" in un'unica sequenza ordinata di n elementi.

Vediamone il funzionamento (facendo riferimento alla figura che segue) su un problema costituito di 8 elementi, dando per scontato (solo per il momento) il funzionamento della funzione di fusione.



Si osservi che le caselle gialle del disegno corrispondono alle operazioni di suddivisione del vettore in sottovettori, cosa che avviene solo tramite le varie chiamate ricorsive. Le caselle azzurre del disegno invece indicano i casi base e quelle verdi corrispondono alle chiamate della funzione di fusione, le quali restituiscono porzioni di vettore ordinate sempre più grandi.

Illustriamo ora il funzionamento della fase di fusione. Essa sfrutta il fatto che le due sottosequenze sono ordinate, cosa che rende facile sfruttare una proprietà molto importante:

- il minimo della sequenza complessiva non può che essere il più piccolo fra i minimi delle due sottosequenze (se essi sono uguali, scegliere l'uno o l'altro non fa differenza);
- dopo aver eliminato da una delle due sottosequenze tale minimo, la proprietà rimane: il prossimo minimo non può che essere il più piccolo fra i minimi delle due parti rimanenti delle due sottosequenze.

L'ordinamento delle due porzioni di vettore permette di scandire ciascun elemento delle sequenze una volta sola, garantendo un numero di operazioni lineare.
E' possibile dimostrare che il numero di operazioni dell'algoritmo di Merge Sort è proporzionale ad n log n; tale algoritmo è perciò molto più efficiente di quelli semplici descritti precedentemente.

Alcune visualizzazioni di Merge Sort:

https://www.youtube.com/watch?v=XaqR3G_NVoo
https://www.youtube.com/watch?v=EeQ8pwjQxTM
https://www.youtube.com/watch?v=JSceec-wEyw

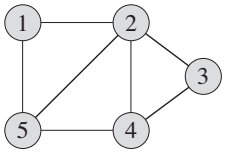
Alcune visualizzazioni in cui vari algoritmi di ordinamento sono messi a confronto:

https://www.youtube.com/watch?v=kPRA0W1kECg
https://www.youtube.com/watch?v=BeoCbJPuvSE

Grafi

In questo paragrafo vengono date alcune definizioni di base riguardanti i grafi, che dovrebbero essere sufficienti per consentire di comprendere le proposte di progetto che li riguardano.

Un grafo G = (V, E) è costituito da una coppia di insiemi, l'insieme finito V di nodi, o vertici, e l'insieme finito E di coppie non ordinate di nodi, dette archi o spigoli.



Come evidenziato in figura, di solito i vertici si rappresentano come punti (o piccoli cerchi) del piano mentre gli archi come linee continue che collegano le rappresentazioni dei loro due estremi.
Due nodi collegati da un arco sono detti adiacenti; ad esempio, nella figura precedente, 2 e 5 sono vertici adiacenti. Si dice che un arco è incidente ai suoi due estremi.
Il numero di archi incidenti in un certo nodo v è detto grado di v (degree(v)).
Usualmente, gli interi n ed m indicano rispettivamente il numero di nodi e di archi di cui è composto il grafo. E' facile vedere che il massimo numero di archi è n(n - 1)/2: basta considerare il grafo complete, cioè quello in cui tutti i possibili archi sono presenti.
Un grafo è aciclico se non è possibile trovare una sequenza di archi distinti che, a partire da un qualche nodo, riconduca nello stesso. Nella figura precedente, ad esempio, il grafo rappresentato non è aciclico, perché la sequenza di archi orientati (1,2), (2,5), (5,1) rappresentano un ciclo (in realtà ce ne sono anche altri, ma basta scovarne uno per affermare che il grafo non è aciclico).
Dato un grafo G=(V, E), se consideriamo solo alcuni suoi nodi ed alcuni suoi archi (ad esempio l'insieme $V' \subseteq V$ e l'insieme $E' \subseteq E$), allora il grafo G'=(V', E') si dirà sottografo di G.
Ad esempio, possiamo scegliere V'={1,2,5} ed E'={(1,2),(1,5),(2,5)}, oppure V''={1,2,5} ed E''={(1,2), (1,5)}, oppure V'''={1,2,3,4,5} ed E'''={(1,2),(2,5),(4,5),(2,3)}, o molte altre coppie di insiemi. Si osservi che G'=(V',E') contiene un ciclo mentre G''=(V'',E'') è aciclico; G'''=(V''',E''')

contiene tutti i nodi del grafo (cioè V'''=V) e solo un sottinsieme degli archi; in questo caso si parla di sottografo ricoprente.

Parte 2

Proposta di progetto

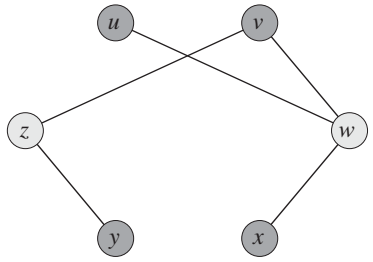
"It was bound to happen—they're beginning to think like binary computers."

La copertura tramite vertici

Una *copertura tramite vertici* di un grafo G=(V,E) è un sottinsieme di vertici C tale che, comunque si scelga un arco di G, almeno uno dei suoi estremi si trova in C.
Informalmente, ogni vertice in C "copre" gli archi ad esso incidenti, e C è una copertura di vertici se tutti gli archi di G risultano coperti.

Questo problema ha numerose applicazioni. Pensate ad un museo, in cui si vogliano controllare con delle guardie tutte le gallerie. Se le sale da cui partono le gallerie rappresentano i vertici di un grafo, e le gallerie gli archi, trovare una copertura tramite vertici equivale a trovare un posizionamento di guardie affinché tutte le gallerie siano sorvegliate.

La *taglia* di una copertura tramite vertici è il numero di vertici che la compongono. Ad esempio, la copertura tramite vertici in figura (che contiene vertici z e w) ha cardinalità 2.



Il *problema della copertura tramite vertici* consiste nel determinare una copertura di taglia minima.
Questo è un problema difficile da risolvere: si dice che il problema è *intrattabile*, cioè non si sa risolvere in tempo polinomialmente proporzionale alla dimensione del grafo in input, e si pensa che sia difficile che ci si possa mai riuscire.

Tuttavia, è molto semplice determinare una copertura tramite vertici la cui cardinalità non sia quella minima.

Di seguito si riportano alcuni algoritmi.

Algoritmo goloso "semplice"

Si considerano in sequenza i vertici del grafo; se il vertice è estremo di almeno un arco non lo si inserisce nella soluzione; poi si elimina (virtualmente) dal grafo il vertice appena inserito e tutti gli archi ad esso incidenti (che sono stati coperti); infine si passa a considerare il vertice successivo della sequenza.
Alla fine, il grafo rimane vuoto, perciò tutti gli archi saranno necessariamente coperti; segue che la soluzione trovata è una copertura tramite vertici corretta, seppure non necessariamente di taglia minima.

Facendo qualche esempio, sembra che si possa facilmente diminuire la taglia della soluzione scegliendo prima i vertici che possono coprire più archi, anziché considerarli per indice crescente. Il prossimo algoritmo va in questa direzione.

Algoritmo goloso "più intelligente"

Si considerano i vertici del grafo e si ordinano rispetto al loro grado, mettendo prima i vertici con grado più alto e dopo quelli con grado più basso.
Si prende il primo vertice (con grado massimo), si inserisce nella soluzione; poi si elimina (virtualmente) dal grafo il vertice appena inserito e tutti gli archi ad esso incidenti (che sono stati coperti); si aggiornano poi i gradi dei vertici rimasti nel grafo (che, in generale sono cambiati, visto che abbiamo rimosso archi); infine, si passa a considerare il nuovo vertice di grado più alto (che non può appartenere alla soluzione, visto che i nodi inseriti nella soluzione sono già stati eliminati dal grafo).

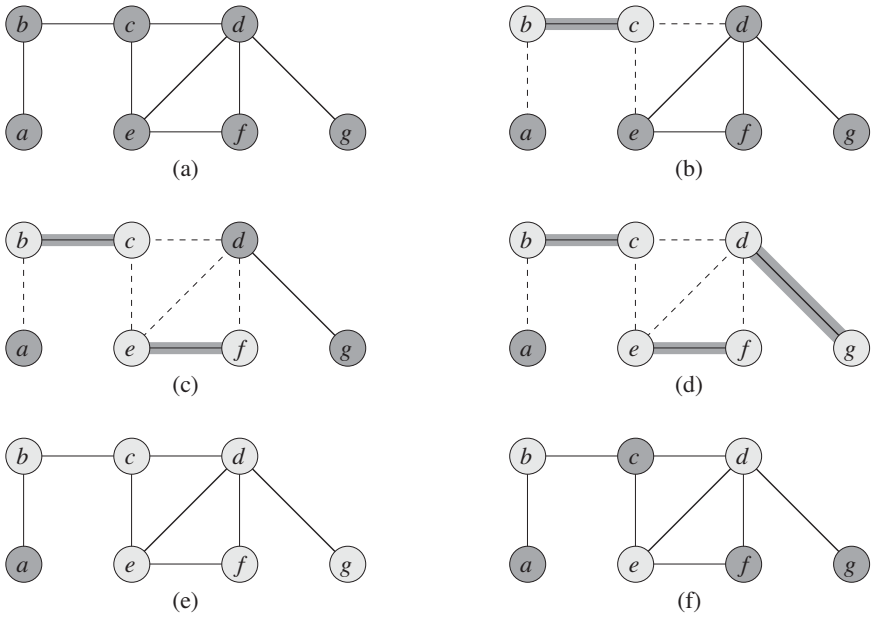
Anche in questo caso, alla fine, il grafo rimane vuoto, perciò tutti gli archi saranno necessariamente coperti, cioè la soluzione trovata è una copertura tramite vertici corretta, ma non abbiamo garanzie sulla taglia della soluzione.

Vediamo ora un algoritmo che garantisce che la taglia della soluzione prodotta sia al più il doppio di quella minima.

Algoritmo goloso 2-approssimante

Si osservi dapprima che per ogni arco (u, v), almeno uno tra i vertici u e v deve essere nella copertura di vertici. È dunque possibile scegliere un qualunque arco (u, v) ed aggiungere entrambi gli estremi all'insieme: in questo modo si copre non solo (u, v), ma anche tutti gli archi incidenti ai vertici u e v. Si può procedere eliminando (virtualmente) dal grafo i vertici inseriti nella soluzione e tutti gli archi ad essi incidenti, ed iterare il ragionamento finché il grafo non diventa vuoto.

Chiariamo meglio l'algoritmo su un esempio.



All'inizio (figura parte (a)) nessun vertice è nella copertura. Poi scegliamo un arco a caso, ad esempio (b, c), - figura parte (b) – ed inseriamo i suoi estremi nella copertura; al contempo, consideriamo come coperti tutti gli archi incidenti ai due nodi considerati. Iteriamo il procedimento scegliendo un nuovo arco non coperto, ad esempio (e, f) – figura parte (c), e di nuovo consideriamo come coperti tutti archi incidenti a questi due vertici. A questo punto, un solo arco (d, g) risulta non coperto, e possiamo quindi inserire i suoi estremi nella copertura – figura parte (d).
In figura parte (e) è mostrata la copertura di vertici finale; si osservi che essa non è ottima, in quanto ne esistono di taglia minore, come mostrato in figura parte (f).

È possibile dimostrare che questo algoritmo garantisce che la taglia della soluzione ottenuta sia al più il doppio di quella ottima; si dice quindi che l'algoritmo è *2-approssimante* (il 2 sta proprio ad indicare il fatto che la cardinalità è al più doppia di quella dell'ottimo).

Algoritmo goloso "casuale"

Si considerano gli archi del grafo uno per uno; per ciascun arco si sceglie a caso **uno** dei due vertici suoi estremi e lo si inserisce nella soluzione; poi si elimina (virtualmente) dal grafo il vertice appena inserito e tutti gli archi ad esso incidenti (che sono stati coperti); infine si passa a considerare un altro arco (che è necessariamente non coperto, visto che gli archi coperti sono stati eliminati dal grafo).
Alla fine, il grafo rimane vuoto, perciò tutti gli archi saranno necessariamente coperti; segue che la soluzione trovata è una copertura tramite vertici corretta, seppure non necessariamente di taglia minima.

Facendo qualche esempio, non è difficile provare che scegliere di selezionare un solo estremo dell'arco (anziché due come invece si faceva nell'algoritmo precedente) fa perdere la proprietà di avere un algoritmo 2-approssimante in quanto la taglia della soluzione prodotta da questo algoritmo può risultare arbitrariamente maggiore di quella della soluzione ottima.

Proposta di visualizzazione

Se vi sono chiari gli algoritmi con cui si possono determinare delle coperture di vertici di un grafo, sceglietene uno: non necessariamente quello migliore, ma quello che vi piace di più.
Poi, procedete così:

1. pensate ad una situazione reale che si può modellare come un grafo e, su di essa, un problema che si può interpretare come una copertura di vertici (come, ad esempio, la sorveglianza delle gallerie di un museo, che abbiamo descritto prima); avrete così il vostro grafo in input;
2. eseguite su questo grafo l'algoritmo che avete scelto al punto 1.;
3. provate a descrivere la situazione in modo visuale:
4. create un video in cui siano descritte tutte queste scelte, evidenziando, in particolare, come vengono scelti i vertici della

copertura, e come questa soluzione possa essere interpretata nella vostra situazione reale.