mlampis.dev

Teoria della Complessita'

Manuale per le ripetizioni di Informatica

Marco Lampis

6 dicembre 2022

Indice

0 Informazioni generali

Ciao! Sono **Marco**, sono uno studente magistrale in *Software Engineering* al Politecnico di Torino e mi sono laureato in Ingegneria Informatica all'università di Pisa. Nella vita sono un programmatore, uno smanettone e amante di videogiochi! Amo quello che studio, e per questo motivo fornisco ripetizioni di informatica con particolare attenzione a:

- programmazione (Java, C++, C, Python, C#, Javascript, PHP)
- algoritmi e strutture dati
- basi di Dati
- più o meno tutto quello che riguarda l'informatica!

Sia per studenti delle scuole superiori che per l'università.

0.1 Come svolgo le lezioni

La mia metodologia è innovativa e interattiva: utilizzo una piattaforma apposita per lavorare contemporaneamente sullo stesso file con gli studenti, come se fossimo accanto. In questo modo, le lezioni sono più coinvolgenti e divertenti, e gli studenti possono imparare in modo facile ed efficiente. Inoltre, le lezioni includono sia esercitazioni che approfondimenti teorici, aiuto compiti e revisione.

Per aiutare gli studenti a prepararsi al meglio e a esercitarsi anche a casa, ho creato un sito web dedicato con risorse utili come teoria, esercizi e slide. Su questo sito, gli studenti possono trovare tutto il materiale di cui hanno bisogno, e il materiale è sempre disponibile gratuitamente per i miei studenti. Inoltre, uso un iPad per prendere appunti durante le lezioni e poi condividerli con gli studenti, in modo che possano rivedere tutto ciò che è stato trattato.

Se volete migliorare le vostre conoscenze in informatica e avere un supporto personalizzato e professionale, non esitate a contattarmi. Sarò felice di fornirvi maggiori informazioni e di fissare una lezione con voi.

1 Introduzione

La teoria della calcolabilità, della computabilità, e della ricorsione cerca di comprendere quali funzioni possono essere calcolate tramite un procedimento automatico. In altre parole, essa cerca di determinare se una data funzione è teoricamente calcolabile a prescindere dal fatto che sia anche trattabile, cioè a prescindere dalla quantità di risorse che la sua esecuzione richiede in termini di tempo o di memoria, che a livello pratico potrebbero essere proibitive. Questa disciplina è comune sia alla matematica sia all'informatica.

Di conseguenza l'obiettivo principale è dare una definizione formale e matematicamente rigorosa dell'idea intuitiva di funzione calcolabile. Da una parte l'approccio è quello di approfondire il concetto di calcolabilità, cercando di individuare le categorie di problemi che sono teoricamente risolvibili, e dall'altra mappare questo concetto su ciò che è teoricamente calcolabile sui computer, sempre senza considerare le limitazioni imposte dai costi, dal tempo e dalla quantità di memoria impiegata.

Un altro importante aspetto è quello di definire matematicamente il concetto di algoritmo in modo che i programmi possano essere concretamente pensati in termini di oggetti matematici, più precisamente come funzioni che restituiscono un determinato risultato a partire da un certo insieme di dati in ingresso.

2 Classi di complessità

In informatica si dividono i problemi in più classi di complessità, in base alla loro difficoltà di risoluzione. Queste classi sono:

- **P**: i problemi risolvibili in tempo polinomiale con una macchina di Turing deterministica.
- **NP**: i problemi possono essere risolti in tempo polinomiale con una macchina di Turing non deterministica. Questa classe è nota anche come *Non-deterministic Polynomial Time*.
- **NP-completo**: i problemi NP-completi sono i problemi più difficili di **NP**. Ogni problema NP completo è tale per cui è possibile ricondurre un qualsiasi problema NP a quest'ultimo.
- **EXP**: i problemi risolvibili in tempo esponenziale (o peggiore) con una macchina di Turing deterministica.

2.1 Algoritmo deterministico

Un algoritmo deterministico è tale se, dato un particolare input, produrrà sempre lo stesso output, con la macchina sottostante che passa sempre attraverso la stessa sequenza di stati; il percorso di esecuzione dell'algoritmo è lo stesso in ogni esecuzione.

2.2 Algoritmo non deterministico

Un algoritmo non deterministico (in inglese Non-deterministic Algorithms) è un algoritmo che, anche per lo stesso input, può esibire comportamenti diversi su corse diverse. In altre parole, è un algoritmo in cui il risultato di ogni algoritmo non è definito in modo univoco e il risultato potrebbe essere casuale.

Una varietà di fattori può far sì che un algoritmo si comporti in modo non deterministico:

- Se utilizza uno stato esterno diverso dall'input (l'input dell'utente, un valore casuale o dati memorizzati nel disco).
- Se funziona in modo sensibile al tempo, ad esempio se ha più processori che scrivono sugli stessi dati contemporaneamente. In questo caso, l'ordine preciso in cui ciascun processore scrive i propri dati influirà sul risultato.

- Se un errore hardware fa sì che il suo stato cambi in modo imprevisto.

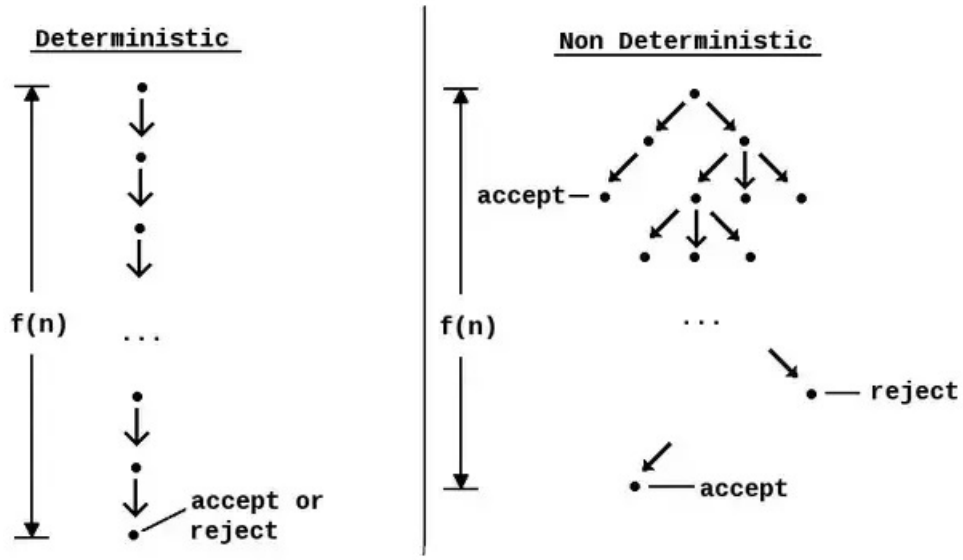

Figura 2.1: Deterministico e Non deterministico

2.3 NP

La classe NP è una classe di problemi che possono essere risolti in tempo polinomiale con una macchina di Turing non deterministica. Questa classe è nota anche come *Non-deterministic Polynomial Time*. La sua caratteristica è che un algoritmo appartenente a NP è verificabile in tempo polinomiale

Consiglio: per pensare a come risolvere in modo polinomiale un problema di questo tipo, ipotizza di avere un numero sufficiente di computer in grado di calcolare, contemporaneamente, la soluzione di un problema eseguendo ciascuna un percorso diverso. In questo modo, si può risolvere il problema in tempo polinomiale.

Attenzione: NP sta per **Non-Deterministic**, non per Non-Polynomial!

2.4 NP-completo

Nella teoria della complessità, un problema NP-completo o NPC (cioè un problema completo per la classe NP) è un problema decisionale che verifica le seguenti proprietà:

- è possibile **verificare** una soluzione in modo efficiente (in tempo polinomiale); la classe di problemi che verificano questa proprietà è indicata con NP;
- tutti i problemi della classe NP si riducono a questa tramite una *riduzione polinomiale*; questo significa che il problema è difficile almeno quanto tutti gli altri problemi della classe NP.

Se dunque venisse scoperto un modo per rendere polinomiale un problema **NP-completo**, ne seguirebbe che lo stesso metodo potrebbe essere applicato a tutti i problemi NP, e quindi anche a tutti i problemi P.

2.5 EXP

La classe di complessità **EXP** si riferisce ai problemi che possono essere risolti in **tempo esponenziale**. In altre parole, un problema appartenente alla classe di complessità EXP può essere risolto utilizzando un algoritmo che impiega un tempo di esecuzione che cresce esponenzialmente con la dimensione dell'input. Ad esempio, un algoritmo appartenente alla classe di complessità EXP potrebbe impiegare 2^n unità di tempo per risolvere un problema di dimensione n, dove n è il numero di elementi dell'input.

Questa include molti dei problemi NP-completi, ovvero problemi che non possono essere risolti in tempo polinomiale ma che possono essere risolti in modo efficiente utilizzando algoritmi appartenenti alla classe NP. Tuttavia, non tutti i problemi NP-completi appartengono alla classe di complessità EXP, poiché esistono algoritmi di approssimazione che possono risolvere alcuni problemi NP-completi in tempo polinomiale.

In generale, la classe di complessità EXP rappresenta un limite superiore per la risoluzione efficiente dei problemi di ottimizzazione, poiché problemi appartenenti a questa classe possono richiedere tempi di esecuzione molto lunghi per input di dimensioni moderate. Tuttavia, grazie ai progressi della ricerca in campo informatico, nuovi algoritmi e metodi di risoluzione vengono continuamente sviluppati che possono risolvere problemi appartenenti alla classe di complessità EXP in modo più efficiente.

2.5.1 Soluzioni imperfette

Fino ad ora, tutti gli algoritmi conosciuti per problemi NP-Completi necessitano di un tempo superpolinomiale nella dimensione dell'input. L'esistenza di algoritmi più veloci è sconosciuta. Quindi, per risolvere un problema NP-Completo di dimensioni non banali, generalmente vengono utilizzati i seguenti approcci:

- algoritmo di approssimazione: un algoritmo che trova velocemente una soluzione sub-ottimale che si trova in un intorno (noto) di quella ottimale.
- algoritmo probabilistico: un algoritmo il cui tempo medio di esecuzione per una distribuzione data del problema è provata essere buona—idealmente, uno che assegna una bassa probabilità a input "difficili".
- casi speciali: un algoritmo veloce se l'istanza del problema appartiene all'insieme di alcuni casi speciali. La parametrizzazione della complessità può essere vista come una generalizzazione di questo approccio.
- euristica: un algoritmo che funziona "ragionevolmente bene" in molti casi, ma per cui non c'è prova che sia sempre veloce e che dia buoni risultati.

2.6 Domande di esame

Di seguito sono riportati alcuni esempi di domande di esame che possono essere utili per prepararsi all'esame di **Computabilità e Complessità**.

2.6.1 Dimostrare che un algoritmo appartenente a P è anche appartenente a NP

In generale, un problema appartiene a P se può essere risolto in tempo polinomiale, mentre un problema appartiene a NP se può essere verificato in tempo polinomiale.

Per dimostrare che NP include (non siamo ancora in grado di dire se strettamente o meno) P, dobbiamo dimostrare che ogni problema di P può essere verificato in tempo polinomiale. Questo è possibile in quanto ogni problema di P può essere risolto in tempo polinomiale da una macchina di Turing deterministica e dunque anche verificato.

2.6.2 Perchè la scomposizione fattoriale è un problema appartenente a NP?

La scomposizione fattoriale è un problema appartenente a NP perché può essere verificata in tempo polinomiale. In altre parole, esiste un algoritmo che può verificare se una data scomposizione di un numero intero in fattori primi è corretta in un numero di passi che cresce in modo polinomiale rispetto alla dimensione dell'input.

Per fare un esempio concreto, supponiamo di avere un numero intero n e di voler verificare se può essere scritto come prodotto di fattori primi. Un modo per farlo potrebbe essere quello di cercare tutti i fattori primi del numero n, **uno per uno**, e verificare se il prodotto di questi fattori è uguale

a n. In questo caso, il tempo di esecuzione dell'algoritmo cresce in modo polinomiale rispetto alla dimensione dell'input, poiché il numero di fattori primi che occorre considerare è proporzionale alla logaritmo del numero n. Pertanto, la scomposizione fattoriale appartiene a NP, **non** avendo un modo per eseguire il calcolo in tempo polinomiale su una macchina deterministica.

2.6.3 Quale è la complessità di verificare se un elemento è presente all'interno di un vettore? Come cambia se è ordinato o meno?

La complessità di verifica se un elemento è all'interno di un vettore non ordinato dipende dall'algoritmo utilizzato per effettuare la verifica. Un modo semplice per farlo potrebbe essere quello di scorrere tutti gli elementi del vettore uno per uno e verificare se l'elemento cercato è presente. In questo caso, la complessità dell'algoritmo è lineare ovvero O(n), poiché il numero di passi necessari per trovare l'elemento cercato è proporzionale alla dimensione del vettore.

Tuttavia, esistono altri algoritmi che possono essere utilizzati per effettuare la verifica in modo più efficiente. Ad esempio, se il vettore è costituito da numeri interi, è possibile utilizzare un algoritmo di ricerca binaria per trovare l'elemento cercato in tempo logaritmico. In questo caso, la complessità dell'algoritmo sarebbe pari a O(log n), dove n è la dimensione del vettore. Questo significa che il numero di passi necessari per trovare l'elemento cercato cresce in modo logaritmico rispetto alla dimensione del vettore, rendendo l'algoritmo più efficiente rispetto al primo approccio.

Se si dovesse però considerare anche il tempo di ordinamento del vettore, tale approccio risulterebbe sconveniente in quanto richiederebbe O(n log n) per ordinare a cui si somma O(log n) per la ricerca binaria con un totale di O(n log n + log n) = O(n log n).

2.6.4 Quale è la complessità di verificare se, dato un insieme finito di numeri, esista un sottoinsieme non vuoto a somma negativa?"

La complessità di verifica che esista un sottoinsieme non vuoto a somma negativa dipende dall'algoritmo utilizzato per effettuare la verifica. Un modo semplice per farlo potrebbe essere quello di scorrere tutti i sottoinsiemi del vettore uno per uno e verificare se la somma degli elementi di ciascun sottoinsieme è negativa. In questo caso, la complessità dell'algoritmo sarebbe esponenziale, poiché il numero di sottoinsiemi da considerare cresce esponenzialmente con la dimensione del vettore.

Tuttavia, esistono altri algoritmi che possono essere utilizzati per effettuare la verifica in modo più efficiente. Ad esempio, si può utilizzare un algoritmo di ricerca dicotomica per trovare un sottoinsieme a somma negativa in tempo polinomiale. In questo caso, la complessità dell'algoritmo sarebbe pari a O(n log n), dove n è la dimensione del vettore. Questo significa che il numero di passi necessari

per trovare il sottoinsieme cercato cresce in modo polinomiale rispetto alla dimensione del vettore, rendendo l'algoritmo più efficiente rispetto al primo approccio.

In generale, quindi, la complessità di verifica che esista un sottoinsieme non vuoto a somma negativa dipende dall'algoritmo utilizzato e dalle caratteristiche del vettore stesso.

2.6.5 Quale è la complessità di verifica, dato un insieme finito di stringhe, quali siano tutti gli anagrammi che iniziano con A?

la complessità di un algoritmo per verificare gli anagrammi può essere O(n^2) nella peggiore delle ipotesi, dove n è il numero di stringhe nell'insieme. In questo caso, l'algoritmo potrebbe essere migliorato utilizzando una mappa di frequenza delle lettere per verificare se una stringa è un anagramma di un'altra, il che ridurrebbe la complessità dell'algoritmo a O(n * m), dove m è il numero medio di lettere in una stringa.

2.7 Bibliografia

- What are the differences between NP, NP-Complete and NP-Hard?
- Differenza tra algoritmo deterministico e non deterministico in informatica
- NP-completo

3 Problema dell'arresto

Nella teoria della computabilità, il **problema dell'arresto** (in inglese *halting problem*) si pone il problema di determinare, dato un *programma* e un *input*, se questo terminerà in un tempo finito o continuerà per sempre. E' stato Alan Turing, nel 1936, a dimostrare che non è possibile avere un algoritmo generale per risolvere questo problema.

3.1 Dimostrazione

Si supponga per assurdo che esista un algoritmo, che, preso in ingresso un qualsiasi algoritmo `A` avente un input `I` ed è in grado di stabilire se a termina in tempo finito (restituendo **`true`**) o se non termina (restituendo **`false`**).

```
// halts() restituisce true se il suo input termina, false altrimenti
boolean halts(programma, input):
    if(programma termina):
        return true
    else:
        return false

// algoritmo P1
boolean P1(a, d):
    return halts(a(d));
```

Visto che per `P1` sia il parametro `A` che il parametro `I` sono solo delle sequenze di simboli *(non conoscendo differenza tra programmi e dati)*, possiamo richiamare `P1` fornendo `A` sia come algoritmo che come input: `P1(A, A)`. Prendiamo ora un'altra funzione `P2` a cui viene passato `A` come argomento e tale che:

- se `P1(A, A)` termina, restituisce **`true`**
- se `P1(A, A)` non termina, prosegue all'infinito (**`while(true);`**)

```
boolean P2(A):
  // se P1 termina, prosegue all'infinito
  if P1(A,A) while(true);

  // se non termina, restituisce true
```

```
6     else
7       return true;
```

Se passiamo a P2 come parametro il risultato stesso di P2, dunque P2(P2), ricadiamo nei seguenti casi:

- se P2 termina, P2(P2) non termina perché entra nel **while(true);**
- se P2 non termina, P2(P2) termina perché entra nel **else**

Ricadiamo allora in un assurdo, perché per avere come risultato **true** è necessario che P2 non termini, ma allo stesso tempo se sta fornendo un risultato è terminato!

Nota: Un algoritmo per assurdo ha una logica simile alla frase *"io mento"*, in quanto si basa sul presupposto che se l'affermazione è vera, per assurdo, è anche vera.

Nota: Per un computer non è possibile stabilire la differenza tra un dato e un algoritmo. Quando eseguiamo i programmi non si fa altro che cercare di "eseguire" dei dati, ma non sono intrinsecamente diversi.

4 SAT

Il problema della soddisfacibilità booleana è il problema della decisione che, data una formula in logica booleana, determina se esiste un insieme di variabili tali da rendere l'esito della formula vera.

Questo problema è molto importante nella teoria della complessità. È stato portato alla luce dal teorema di Cook, che è alla base della teoria della NP-completezza e del problema P = NP.

Il problema SAT è un problema NP-completo, il che significa che può essere risolto in modo efficiente (anche se non sempre perfetto) solo se si utilizzano algoritmi appartenenti alla classe NP. Inoltre, poiché il problema SAT è NP-completo, la sua soluzione può essere utilizzata per risolvere un ampio numero di altri problemi di ottimizzazione in modo efficiente.

4.1 Esempi

Esempi di problemi SAT includono:

- Determinare se una serie di clausole logiche può essere soddisfatta utilizzando un insieme di variabili booleane
- Determinare se un grafo può essere colorato con un numero limitato di colori in modo tale che nessun vertice adiacente abbia lo stesso colore
- Determinare se un insieme di numeri interi può essere suddiviso in sottoinsiemi che siano tutti la somma di uno stesso numero (ad esempio, suddividere un insieme di numeri interi in sottoinsiemi tutti la somma di 10)

Mentre una formulazione più rigorosa di quali combinazioni rendono vere le seguenti espressioni:

- `X and Y`, sia `X` che `Y` devono essere vere
- `X or Y`, basta che sia **`true`** una delle due variabili
- `X and not X`, non può essere vero
- `X or not X`, è sempre vero

5 Problema delle somme parziali

Il **problema delle somme parziali** è un problema di ottimizzazione che si riferisce alla possibilità di trovare una sottosequenza di una sequenza di numeri interi in modo tale che la somma dei numeri in questa sottosequenza sia massimale (o un numero scelto). Ad esempio, consideriamo la seguente sequenza di numeri:

```
4, −5, 8, 11, −3, 7, −2, 3, −20, 9
```

Una possibile soluzione al problema delle somme parziali per questa sequenza di numeri sarebbe la sottosequenza:

```
8, 11, −3, 7, −2, 3, 9
```

La sequenza di sopra massimizza il risultato, che ha una somma parziale di 25.

Il problema delle somme parziali può essere risolto in modo efficiente utilizzando un algoritmo di **programmazione dinamica**. Tuttavia, poiché il problema delle somme parziali è un problema NP-completo, non esiste un algoritmo noto per risolverlo in tempo polinomiale per una sequenza di numeri di lunghezza arbitraria.

6 Problema dello zaino

Il **problema dello zaino**, o in inglese *Knapsack problem*, è un problema di *ottimizzazione combinatoria* che si pone il seguente problema:

> data una lista di oggetti, ognuno con un certo valore e un certo peso, e una capacità massima di peso, trovare la combinazione di oggetti che massimizza il valore senza superare la capacità massima di peso.

Questo problema è detto np-completo, in quanto:

- non è risolvibile in tempo polinomiale
- è possibile trovare una soluzione ottima in tempo esponenziale
- dato un input, è possibile verificarlo in tempo polinomiale

Il tempo di trovare la risposta è pari a `O(2^n)`, dove `n` è il numero di oggetti, ovvero quello di valutare tutti i sottoinsiemi.

Il problema dello zaino può essere risolto in modo efficiente utilizzando un algoritmo di **programmazione dinamica**. Tuttavia, poiché il problema dello zaino è un problema **NP-completo**, non esiste un algoritmo noto per risolverlo in tempo polinomiale per un numero arbitrario di oggetti e di capacità dello zaino.

6.1 Esempio

Ad esempio, immaginiamo di avere a disposizione uno zaino con una capacità di *20 chilogrammi* e di dover scegliere tra i seguenti oggetti:

oggetto	peso	valore
Libro	1kg	10 euro
Lampada	3kg	20 euro
Computer	5kg	100 euro

oggetto	peso	valore
Materasso gonfiabile	8kg	40 euro

Una possibile soluzione al problema dello zaino per questa situazione sarebbe scegliere di inserire nel nostro zaino il libro, la lampada e il computer portatile, poiché questi oggetti hanno un valore totale di `130` euro e un peso totale di `9 kg`, che è inferiore alla capacità massima dello zaino.

7 Problema del commesso viaggiatore

Il **problema del commesso viaggiatore**, o dall'inglese **TSP** *(Traveling Salesman Problem)* è il seguente: dato un insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza.

Non esistono algoritmi efficienti per la risoluzione del TSP, l'unico metodo di risoluzione è rappresentato dall'**enumerazione totale**, ovvero nell'elaborazione di tutti i possibili cammini sul grafo per la successiva scelta di quello migliore. Per tale motivo la versione decisionale del problema è **np-completo**.

Il problema del commesso viaggiatore può essere risolto in modo efficiente utilizzando un algoritmo di **programmazione dinamica** o un algoritmo di **ricerca esaustiva**, come l'algoritmo di **Branch and Bound**. Tuttavia, poiché il problema del commesso viaggiatore è un problema NP-completo, non esiste un algoritmo noto per risolverlo in tempo polinomiale per un numero arbitrario di città da visitare.

7.1 Esempio

Immaginiamo che un commesso viaggiatore debba visitare le seguenti città:

- New York
- Chicago
- Los Angeles
- Miami
- Dallas

e che la distanza tra ogni coppia di città sia la seguente:

Città	New York	Chicago	Los Angeles	Miami	Dallas
New York	0	1,000	2,000	500	1,200

Città	New York	Chicago	Los Angeles	Miami	Dallas
Chicago	1,000	0	1,800	600	1,400
Los Angeles	2,000	1,800	0	1,500	1,600
Miami	500	600	1,500	0	1,100
Dallas	1,200	1,400	1,600	1,100	0

In questo caso, una possibile soluzione al problema del commesso viaggiatore sarebbe il seguente percorso:

New York -> Chicago -> Los Angeles -> Miami -> Dallas -> New York

che ha una distanza totale di 6,300 km.

_

7.2 Bibliografia

- Problema del commesso viaggiatore