Algoritmi e complessità

Indice

1. Lezione 01 [26/09]

1.1. Notazione

1.1.1. Insiemi numerici

Useremo i principali **insiemi numerici** come $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ e, ogni tanto, le loro versioni con soli elementi positivi $\mathbb{N}^+, \mathbb{Z}^+, \mathbb{Q}^+, \mathbb{R}^+$.

1.1.2. Monoide libero

Un **magma** è una struttura algebrica $(A, \cdot)$ formata da un insieme e un'operazione. Se essa è:

- dotata di $\cdot$ **associativa** allora è detta **semigruppo**;
- dotata di un elemento $\overline{e} \in A$ tale che

  $$\forall x \in A \quad x \cdot e = e \cdot x = x$$

  allora è detta **monoide**; l'elemento $\overline{e}$ è chiamato **elemento neutro** e in un monoide esso è unico; alcuni monoidi importanti sono $(\mathbb{N}, +)$ oppure $(\mathbb{N}, *)$
- dotata di $\cdot$ **commutativa** allora si aggiunge **abeliano** alla sua definizione

Un **monoide libero** è un monoide i cui elementi sono generati da una base. Vediamo un importante monoide libero che useremo spesso durante il corso.

Partiamo da un **alfabeto** Σ, ovvero un insieme finito non vuoto di lettere/simboli. Definiamo Σ^* come l'insieme di tutte le sequenze di lettere dell'alfabeto Σ; queste sequenze sono dette parole/stringhe e una generica parola è $w \in \Sigma^*$ nella forma $w = w_0 ... w_{n-1} \mid n \geq 0 \land w_i \in \Sigma$. Usiamo $n \geq 0$ perché esiste anche la **parola vuota** ε. L'insieme Σ^* è numerabile.

Data una parola $w \in \Sigma^*$ indichiamo con $|w|$ il numero di simboli di w. La parola vuota è tale che $|\varepsilon| = 0$.

Un'operazione che possiamo definire sulle parole è la **concatenazione**: l'operazione è

$$\cdot : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$$

ed è tale che, date

$$x = x_0 ... x_{n-1} \quad y = y_0 ... y_{m-1} \mid x, y \in \Sigma^*$$

posso calcolare $z = x \cdot y$ come

$$z = x_0 ... x_{n-1} y_0 ... y_{m-1}.$$

Dato il magma $(\Sigma^*, \cdot)$, esso è:

- semigruppo perché $\cdot$ associativa;
- non abeliano perché $\cdot$ non commutativa (lo sarebbe se $\Sigma = \{x\}$);
- dotato di neutro $e = \varepsilon$.

Ma allora $(\Sigma^* \cdot)$ è un monoide. Esso è anche un monoide libero su Σ.

1.1.3. Funzioni

Chiamiamo

$$B^A = \{f \mid f : A \longrightarrow B\}$$

l'insieme di tutte funzioni da A in B; usiamo questa notazione perché la cardinalità di questo insieme, se A e B sono finiti, ha cardinalità $|B|^{|A|}$.

Spesso useremo un numero K come "insieme": questo va inteso come l'insieme formato da K termini, ovvero l'insieme $\{0, 1, ..., k-1\}$. Ad esempio, $0 = \varnothing$, $1 = \{0\}$, $2 = \{0, 1\}$, eccetera.

Date queste due definizioni, vediamo qualche insieme particolare.

Indichiamo con 2^A l'insieme

$$\{f \mid f : A \longrightarrow \{0, 1\}\},$$

ovvero l'insieme delle funzioni che classificano gli elementi di un A in un dato sottoinsieme di A, cioè ogni funzione determina un certo sottoinsieme. Possiamo quindi dire che

$$2^A \simeq \{X \mid X \text{ sottoinsieme di } A\}.$$

Questo insieme si chiama anche **insieme delle parti**, si indica con $\mathcal{P}(A)$ e ha cardinalità $2^{|A|}$ se A è finito.

Indichiamo con A^2 l'insieme

$$\{f \mid f : \{0, 1\} \longrightarrow A\}$$

l'insieme che rappresenta il **prodotto cartesiano**: infatti,

$$A^2 \simeq A \times A.$$

Indichiamo con 2^* l'insieme delle stringhe binarie, ma allora l'insieme 2^{2^*} è la famiglia di tutti i linguaggi binari, ad esempio $\varnothing$, 2^*, $\{\varepsilon, 0, 00, 000, ...\}$, eccetera.

1.2. Algoritmi 101

In questo corso vedremo una serie di algoritmi che useremo per risolvere dei problemi, ma cos'è un problema?

Un problema Π è formato da:

- un insieme di input possibili $I_\Pi \subseteq 2^*$;
- un insieme di output possibili $O_\Pi \subseteq 2^*$;
- una funzione $\text{Sol}_\Pi : I_\Pi \longrightarrow 2^{O_\Pi}/\{\varnothing\}$; usiamo l'insieme delle parti come codominio perché potrei avere più risposte corrette per lo stesso problema.

Se in un problema mi viene chiesto di "decidere qualcosa", siamo davanti ad un **problema di decisione**: questi problemi sono particolari perché hanno $O_\Pi = \{0, 1\}$ e hanno **una sola risposta possibile**, vero o falso, cioè non posso avere un sottoinsieme di risposte possibili.

Un algoritmo per Boldi è una **Macchina di Turing**. Sappiamo già come è fatta, ovvero:

- nastro bidirezionale infinito con input e blank:
- testina di lettura/scrittura two-way;
- controllo a stati finiti;
- programma/tabella che permette l'evoluzione della computazione.

Perché usiamo una MdT quando abbiamo a disposizione una macchina a registri (RAM, WHILE, lambda-calcolo)?

La **tesi di Church-Turing** afferma un risultato molto importante che però possiamo dare in più "salse":

- tutte le macchine create e che saranno create sono equivalenti, ovvero quello che fai con una macchina lo fai anche con l'altra;
- nessuna definizione di algoritmo può essere diversa da una macchina di Turing;
- la famiglia dei problemi di decisione che si possono risolvere è uguale per tutte le macchine;

- i linguaggi di programmazione sono Turing-completi, ovvero se ipotizziamo una memoria infinita allora è come avere una MdT.

Anche un computer quantistico è una MdT, come calcolo almeno, perché in tempo si ha la quantum supremacy.

Un **algoritmo** A per Π è una MdT tale che

$$x \in I_\Pi \rightsquigarrow \boxed{\text{A}} \rightsquigarrow y \in O_\Pi$$

tale che $y \in \text{Sol}_\Pi(x)$, ovvero quello che mi restituisce l'algoritmo è sempre la risposta corretta.

Ma tutti i problemi sono risolvibili? No, grazie Mereghetti.

Questo lo vediamo con le cardinalità:

- i problemi di decisione sono i problemi dell'insieme 2^{2^*}, ovvero data una stringa binaria (il nostro input) devo dire se essa sta o meno nell'insieme; questo insieme è tale che
$$\left|2^{2^*}\right| \approxeq \left|2^{\mathbb{N}}\right| \approxeq |\mathbb{R}|;$$
- i programmi non sono così tanti: visto che i programmi sono stringhe, e visto che Σ^* è numerabile, le stringhe su un linguaggio sono tali che $2^* \sim \mathbb{N}$.

Si dimostra che $\mathbb{N} \nsim \mathbb{R}$, quindi sicuramente esistono dei problemi che non sono risolvibili.

Una volta che abbiamo ristretto il nostro studio ai solo problemi risolvibili (noi considereremo solo quelli) possiamo chiederci quanto efficientemente lo riusciamo a fare: questa branca di studio è detta **teoria della complessità**.

In questo ambito vogliamo vedere quante risorse spendiamo durante l'esecuzione dell'algoritmo o del programma.

Abbiamo in realtà due diverse teorie della complessità: algoritmica e strutturale.

La **teoria della complessità algoritmica** ci chiede di:

- stabilire se un problema Π è risolubile;
- se sì, con che costo rispetto a qualche risorsa.

Le risorse che possiamo studiare sono:

- tempo come numero di passi o tempo cronometrato;
- spazio;
- numero di CPU nel punto di carico massimo;
- somma dei tempi delle CPU;
- energia dissipata.

Noi useremo quasi sempre il **tempo**. Definiamo

$$T_A : I_\Pi \longrightarrow \mathbb{N}$$

funzione che ci dice, per ogni input, quanto ci mette l'algoritmo A a terminare su quell'input.

Questo approccio però non è molto comodo. Andiamo a raccogliere per lunghezza e definiamo

$$t_A : \mathbb{N} \longrightarrow \mathbb{N}$$

tale che

$$t_A(n) = \max\{T_A(x) \mid x \in I_\Pi \wedge |x| = n\}$$

che va ad applicare quella che è la filosofia **worst case**. In poche parole, andiamo a raccogliere gli input con la stessa lunghezza e prendiamo, per ciascuna categoria, il numero di passi massimo che è stato rilevato. Anche questa soluzione però non è bellissima: è una soluzione del tipo "STA ANDANDO TUTTO MALEEEEE" (grande cit.).

Abbiamo altre soluzioni? Sì, ma non sono il massimo:

- la soluzione **best case** è troppo sbilanciata verso il "sta andando tutto bene";
- la soluzione **average case** è complicata perché serve una distribuzione di probabilità.

A questo punto teniamo l'approccio worst case perché rispetto agli altri due non va a rendere complicati i conti. Inoltre, prendere il massimo ci dà la certezza di non fare peggio di quel valore.

Useremo inoltre la **complessità asintotica**, ovvero per n molto grandi vogliamo vedere il comportamento dei vari algoritmi, perché "con i dati piccoli sono bravi tutti".

Il simbolo per eccellenza è l'O-grande: se un algoritmo ha complessità $O(f(n))$ vuol dire che $f(n)$ domina il tempo t_A del nostro algoritmo.