Appunti Computability

Diego Oniarti

Anno 2024-2025

Contents

1 Macchina di Turing **2**

2 Multi-Tape Turing Machine **3**

3 Turing vs Universal Machines **4**
3.1 Random-access Machine 5

4 Uncomputable machines **5**
4.1 Halting Problem 6
4.1.1 rambling 6

5 Recap **6**
5.1 Insiemi 6
5.2 recursive 7
5.3 Recursively Enumerable 7
5.4 coRE 8

6 Ordering **8**

7 boh **8**

8 Busy beaver "game" **9**

9 2024-10-07 **9**

10 Proprietà di una TM **9**
10.1 Teorema di Rice 10

11 Random String **11**
11.1 Compression Algorithm 12
11.2 Kolmogorov 12
11.3 "Most strings are incompressible" 13
11.4 Kolmogorov complexity 13
11.5 La Kolmogorov Complexity È Uncomputable 14

12 Reduction **15**
12.1 Proprietà 15

13 Unione di $RE \setminus R$ **15**

14 Computational Complexity **17**
14.1 Deterministic Time 18
14.2 Polynomial Languages 18
14.3 Prime numbers 18
14.4 Integer Linear Programming 19
14.5 Certificazione 19

15 Non-deterministic Turing machines **20**
15.1 Independent Set 20
15.2 Reductions (polynomial) 21
15.3 k-CNF 21

16 Cook-Levin theorem **22**
16.1 NP-complete 22

17 2024-11-25 **23**
17.1 3-vertex coloring 23

18 Everything is in EXP **23**
18.1 Linguaggio EXP non NP 23

19 space complexity **24**

1 Macchina di Turing

Una macchina di Turing è rappresentata da:

- $\Sigma = \{\sigma_0, \ldots, \sigma_{m-1}\}$
- $Q = \{q_0, \ldots, q_{n-1}\}$
- $q_o \in Q$ stato iniziale
- $f : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$

La funzione di transizione può essere definita come una tabella

Q

	σ_0	$\cdots$	σ_n
q_0			
$\vdots$			
q_n			

Un altra rappresentazione per una macchina di Turing è quella della macchina a stati (come quelle viste in LFC). Gli stati corrispondono agli stati della macchina di Turing, mentre le transizioni contengono il carattere letto, quello da scrivere, e la transizione.

NB. Come definiamo la funzione di transizione non è importante. Per la definizione di una macchina di Turing basta che esista una funzione di transizione del tipo $f : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$

Non è necessario che la funzione di transizione sia totale.

Esempio Questa è la funzione di transizione per una macchina di Turing che inizia con un numero binario sul nastro e ci aggiunge 1.

	$\llcorner\!\lrcorner$	0	1
LSB	$\llcorner\!\lrcorner, carry, \leftarrow$	$0, LSB, \rightarrow$	$1, LSB, \rightarrow$
$carry$	$1, MSB, \leftarrow$	$1, MSB, \leftarrow$	$0, carry, \leftarrow$
MSB	$\llcorner\!\lrcorner, halt, \rightarrow$	$0, MSB, \leftarrow$	$1, MSB, \leftarrow$

Esempio Ideiamo una macchina di Turing che inizia con un numero sul nastro. La macchina deve:

- creare una copia del numero letto
- scrivere questa copia a destra del numero dato
- lasciare il numero intoccato
- lasciare un $\llcorner\!\lrcorner$ tra i due numeri

Una soluzione valida sarebbe questa, dove l'alfabeto è $\Sigma = \{\llcorner\!\lrcorner, 0, 1, \hat{0}, \hat{1}\}$ e la funzione di transizione è:

	$\llcorner\!\lrcorner$	0	1	$\hat{0}$	$\hat{1}$
$next$	$\llcorner\!\lrcorner, halt, \rightarrow$	$\hat{0}, first0, \rightarrow$	$\hat{1}, first1, \rightarrow$		
$first1$	$\llcorner\!\lrcorner, scom1, \rightarrow$	$0, first1, \rightarrow$	$1, scom1, \rightarrow$		
$scom1$	$1, left, \leftarrow$	$0, scom1, \rightarrow$	$1, scom1, \rightarrow$		
$first0$	$\llcorner\!\lrcorner, scom0, \rightarrow$	$0, first0, \rightarrow$	$1, scom0, \rightarrow$		
$scom0$	$0, left, \leftarrow$	$0, scom0, \rightarrow$	$1, scom0, \rightarrow$		
$left$	$\llcorner\!\lrcorner, left, \leftarrow$	$0, left, \leftarrow$	$1, left, \leftarrow$	$0, next, \rightarrow$	$1, next, \rightarrow$

2 Multi-Tape Turing Machine

Per convenzione immaginiamo una macchina con un tape di input, uno di output, e gli altri sono per utilizza arbitrario.
Ogni tape ha un puntatore suo, e può decidere di muoverlo o lasciarlo intoccato.
La funzione di transizione prende questa forma

$$f : \Sigma^t \times Q \to \Sigma^t \times Q \times \{\leftarrow, \downarrow, \rightarrow\}^t$$

dove l'apice t (numero di tape) indica che l'elemento è ripetuto t volte all'interno di una tupla.
La funzione quindi prende l'input di tutti i tape e lo stato corrente. Questo decide cosa scrivere in ogni tape, lo stato a cui muoversi, e la direzione in cui muovere ogni tape.

Multi-Tape vs Single Tape Una macchina di Turing con più tape può svolgere le stesse computazioni di una macchina a tape singolo. È solo più veloce a farlo.

3 Turing vs Universal Machines

Una differenza che rimane tra la macchina di Turing descritta e un computer come lo vediamo oggi è la seguente. Una macchina di Turing è hard coded per svolgere un singolo compito.

Possiamo quindi formalizzare una macchina di Turing U che sia universale? sì.
L'input di U deve essere un encoding di una macchina m.

Prendiamo per esempio la macchina di Turing che aggiunge 1 ad un numero (vista in precedenza).

	$\sqcup$	0	1
0	$\sqcup, 1, \leftarrow$	$0, 0, \rightarrow$	$1, 0, \rightarrow$
1	$1, H$	$1, H$	$0, 1, \leftarrow$

Possiamo definire un'alfabeto che ci permetta di descrivere questa tabella sotto forma di stringa.

$$\Sigma = \{\sqcup, 1, 0, ', ', ; \}$$
$$\text{tabella} = \sqcup, 1, 0; 0, 0, 1; 1, 0, 1; 1, , ; 1, , ; 0, 1, 0$$

Prendiamo cone convenzione che ogni cella sia definita da 3 simboli separati da virgole. Un simbolo mancante è interpretato come l'*Halting State*.

La macchina universale U è composta da:

- Un tape $\lfloor m \rfloor$ che rappresenta la macchina m
- Un tape s che rappresenta lo stato di m
- Uno o più tape usati per l'esecuzione di m

nb. Come detto in Sec.2, questo può essere svolto anche con una macchina a tape singolo.

L'esecuzione di m utilizzando U richiede più *step* dell'esecuzione di m. Ma il numero di step di U scala linearmente con quello di m.

$$t(m, s) \leq 2|s| + 1$$
$$t(U, \lfloor m \rfloor s) \leq kt(m, s)$$

dove $t(a, b)$ è il tempo di esecuzione della macchina a sull'input b.

3.1 Random-access Machine

Un computer moderno può essere definito come una "*random access machine*" in quanto accede agli indirizzi di memoria in tempo costante, a differenza della macchina di Turing che deve scorrere il tape.
Questa è l'unica differenza tra i due tipi di macchine. Quella di Turing è "lenta".

Ogni altro aspetto di una CPU odierna può essere creato analogamente in una macchina di Turing (Pc, registri, memoria, etc..).

Nota sull'alfabeto

Abbiamo usato un alfabeto Σ di 5 caratteri per descrivere la Turing machine, ma potremmo rappresentare ogni simbolo con un numero binario a tre cifre. Questo ci permette di descrivere un programma come una stringa binaria.

4 Uncomputable machines

Possiamo rappresentare ogni possibile macchina di Turing e ogni può output in una tabella

$$TM \begin{cases} \overbrace{\quad\quad\quad\quad\quad\quad\quad\quad}^{\Sigma^*} \\ \begin{array}{c|ccccc} & \epsilon & 0 & 1 & 00 & \cdots \\ \hline \epsilon & & & & & \\ 0 & & & & & \\ 1 & & & & & \\ \vdots & & & & & \end{array} \end{cases}$$

$$UC : \Sigma^* \mapsto \{0, 1\}$$
$$UC(\alpha) = \begin{cases} 0 & m_\alpha(\alpha) = 1 \\ 1 & \text{altrimenti} \end{cases}$$
m_α macchina descritta dalla stringa α

Ora possiamo usare l'argomento della diagonale per creare una macchina UC che non sia computabile.

Thesis

$$\forall m \in TM \exists s \in \{0,1\}^* : \ m(s) \neq UC(s)$$

Sapevamo già che esistessero problemi non calcolabili. Questa è sono un'altra prova.

$$\begin{aligned} UC \in \{f : \Sigma^* \to \{0,1\}\} \quad & \text{uncountable} \\ \{TM\} \quad & \text{countable} \end{aligned}$$

Congettura di Goldbach. Ogni numero maggiore di due può essere espresso come la somma di due numeri primi.

4.1 Halting Problem

Esiste una macchina $H(\lfloor M \rfloor, \epsilon)$ che so comporti così?

$$H(\lfloor M \rfloor, \epsilon) = \begin{cases} 1 & M(\epsilon) \text{ halts} \\ 0 & M(\epsilon) \text{ does not halt} \end{cases}$$

No. Se esistesse esisterebbe anche la macchina H' con questo comportamento.

$$H'(\lfloor M \rfloor, \epsilon) = \begin{cases} 0 & H(\lfloor M \rfloor, \epsilon) == 1 \\ \infty & H(\lfloor M \rfloor, \epsilon) == 0 \end{cases}$$

Il comportamento di $H(H'(\lfloor M \rfloor, \epsilon))$ non può poi essere definito.

NB! Questa non è la dimostrazione usata dal prof. Per quella chiedi in giro.

4.1.1 rambling

Halt non è ricorsiva. Halt è ricorsivamente enumerabile? Si. Basta usare il metodo "parallelo" diagonale visto in precedenza. (avanzare tutti i casi di uno step alla volta) faccio un backup

5 Recap

5.1 Insiemi

Abbiamo due modi di definire un subset ti tutte le stringhe.

$$s \subseteq \Sigma^*$$
$$f : \Sigma^* \to \{0,1\}$$

5.2 recursive

Un set è *recursive* se e solo se

$$s \in R \iff \exists m\ TM\ s.t. \forall x \in \Sigma^*\ m(x) = \begin{cases} 0 & x \notin s \\ 1 & x \in s \end{cases}$$

5.3 Recursively Enumerable

Ci sono tre modi di definire RE.

- Un set è ricorsivamente enumerabile se:

  $$s \in RE \iff \exists m\ TM\ s.t. \forall x \in \Sigma^*\ m(x) = \begin{cases} 1 & x \in s \\ \text{anything else} & x \notin s \end{cases}$$

  Anything Else include anche il non haltare mai.

- $$\forall x \in \Sigma^* m(x) = \begin{cases} 1 & x \in s \\ \infty & x \notin s \end{cases}$$

- m scrive su un tape tutti e soli gli elementi di s.

Possiamo dimostrare che le 3 definizioni sono equivalenti.

- $2 \implies 1$: Triviale. $\infty \in$ Anything Else
- $1 \implies 2$: Assumendo di avere una macchina m_1, possiamo costruire una macchina m_2.

  $$m_2(x) = \begin{cases} 1 & m_1(x) = 1 \\ \infty & otherwise \end{cases}$$

  Questa tecnica di prendere una macchina e modificarla per crearne un'altra è chiamata **riduzione**.

- $2 \implies 3$: Assumiamo di avere m_2.

  $queue \leftarrow empty$
  $\forall x \in \Sigma^*$:
  $queue.push$ $(x, \text{init configuration of } m_2(x))$
  $\forall (y, \text{configuration of } m_2(y)) \in q$:
  if configuration is halted :
  output y
  remove from queue$(y, config)$
  $else$:
  advance configuration by one step

Diagonale. Questo è a tutti gli effetti un ennesimo utilizza del metodo diagonale.

5.4 coRE

Un set è Co recursively enumerable (coRE) se il suo complementare è ricorsivamente enumerabile.

$$s \in RE \qquad m(x) = \begin{cases} 1 & x \in S \\ \text{anything else} & x \notin S \end{cases}$$

$$s \in coRE \qquad m(x) = \begin{cases} 0 & x \notin S \\ \text{anything else} & x \in S \end{cases}$$

$$s \in coRE \qquad \overline{m(x)} = \begin{cases} 0 & x \notin S \\ \infty & x \in S \end{cases}$$

Set. Dato il powerset di Σ^* (tutte le stringhe), RE e $coRE$ sono due sottoinsiemi di $P(\Sigma^*)$. R (linguaggi ricorsivi) **è** l'intersezione di RE e $coRE$.

6 Ordering

Sia dato un linguaggio $L \subseteq \Sigma^*$ e un ordinamento $<$. Assumiamo che L sia ricorsivamente enumerabile ma non ricorsivo $L \in RE \setminus R$. Essendo L in RE, esiste una macchina m che produce tutti gli elementi di L. Possiamo provare che non esiste una macchina che li produce in ordine.

7 boh

$$HALT = \{(t, s) : m_t(s) \neq \infty\} \in RE \setminus R$$
$$HALT_\epsilon = \{t : m_t(\epsilon) \neq \infty\} \notin R$$

Ipotizziamo per assurdo che H_ϵ sia ricorsiva.

$$H_\epsilon : \Sigma^* \to \{0, 1\}$$
$$t \mapsto \begin{cases} 1 & m_t(s) \text{ halts} \\ 0 & \text{otherwise} \end{cases}$$

AO! mi sono distratto e non ho seguito. Però la prova funziona per riduzione e contraddizione. Crea una macchina H che scrive un input e

chiama H_ϵ mi pare

8 Busy beaver "game"

$$|\Sigma| = n$$
$$|Q| = m$$
$$halt \notin Q$$

Il numero di macchine possibili con questi parametri è $2n^2m(m+1)$. Si può vedere questo costruendo la tabella che definisce le transizioni della macchina.

Chiamiamo $\Sigma(m)$ Il numero massimo di 1 che una macchina con m stati può mettere sul tape.
Poi chiamiamo $S(m)$ il numero massimo di step che una macchina con m esegue prima di haltare.
Per entrambi consideriamo solo macchine che ricevono ϵ come input e haltano.

$S(m)$ non è computabile.

9 2024-10-07

$$(L, <) \subseteq (\Sigma^*, >)$$
$$L \in RE \setminus R$$
$$m_L$$

10 Proprietà di una TM

Una *Proprietà* di una Turing machine è una qualsiasi funzione binaria (decision function) sulla macchina.

$$HALT_\epsilon : TM \mapsto \{0, 1\}$$

Un esempio è Halt. Altri sono:

1. m has 10 states (Computable)
2. m decides prime numbers (Specification)
3. m recognizes halting TMs (Semantica)
4. m decides halting TMs (Computable perché è sempre *False*. Triviale)

Riconoscere vs Decidere. Una macchina di Turing *Riconosce* qualcosa se conferma qualcosa ("risponde si"). Ma non ha un comportamento stabilito in caso contrario

Una macchina *Decide* qualcosa se risponde "si" o "no" in maniera definitiva

Quindi una proprietà "P" **decide** un set di Turing Machines.

$$P : TM \mapsto \{0, 1\}$$

Triviale Una proprietà P è *triviale* se $P = \emptyset$ o $P = TM$

Specification Boh °¬°

Semantic Ogni macchina di Turing può essere vista come

$$m(s) = \begin{cases} 1 \\ \text{anything else} \\ \infty \end{cases}$$

Quindi possiamo dire che ogni macchina di Turing riconosce un linguaggio $L(m) = \{s \in \Sigma^* : m(s) = 1\}$

Una proprietà è *Semantica* se.

$$\forall m_1, m_2 \in TM. L(m_1) = L(m_2) \implies P(m_1) = P(m_2)$$

Se le due macchine compiono lo stesso lavoro (riconoscono lo stesso linguaggio): O entrambe hanno la proprietà, o nessuna delle due la ha.

$$P(m) = \text{"All strings recognized by } m \text{ have an even length"}$$

La macchina che riconosce solo la stringa vuota (ϵ) ha questa proprietà. Questo perché tutte le stringhe che vengono riconosciute da questa macchina (solo 1) hanno lunghezza 0, che è pari.

10.1 Teorema di Rice

Se una proprietà è sia *semantica* che *non triviale* allora è **undecidable**

Prova per assurdo Sia P semantica e non triviale. Deve esserci almeno una macchina m_p per cui la proprietà sia vera (altrimenti sarebbe triviale)

$$m_p \in TM \ s.t. \ P(m_p) = 1$$

Without loss of generality: $L(m) = \emptyset \implies P(m) = 0$ Le macchine che riconoscono l'insieme vuoto non hanno la proprietà P.

Supponiamo per assurdo che P sia decidibile. Quindi

$$\exists \mathcal{P} \in TM \ s.t. \ \forall m : \mathcal{P}(m) = P(m)$$

Esiste una macchina $\mathcal{P}$ che decide la proprietà P.

Abbiamo poi la macchina $HALT : TM \times \Sigma^* \mapsto \{0, 1\}$ che decide se una certa macchina halta con un certo input.

Prendiamo poi una macchina qualsiasi $n \in TM$. Ovviamente possiamo ottenere $\mathcal{P}(\lfloor n \rfloor)$. La macchina prende un input t e:

Algorithm 1: n

save t on a separate tape;
Put s on the input tape;
run $m(s)$;
restore original input t;
run $m_p(t)$

$$P(m_{ms}) = \begin{cases} 0 & m(s) = \infty \\ m_p(t) & m(s) \neq \infty \end{cases}$$

$$m(s) = \infty \implies L(n_{ms}) = \emptyset$$
$$m(s) \neq \infty \implies L(n_{ms}) = L(m_p) \implies P(n_{ms} = P(m_p) = 1$$

Quindi questa macchina risolverebbe l'halting problem. Questo è ovviamente assurdo e prova la tesi.

11 Random String

Qual'è la definizione di una stringa random? Potremmo dire che una stringa è casuale se ogni simbolo ha la stessa probabilità di apparire in ogni posizione. Ma data una determinata stringa, è possibile determinare se sia stata generata tramite un processo casuale o con intento?

Per esempio $S_0 = 000000$ non sembra casuale, ma potrebbe esserlo. $S_1 = 01101001010010$ invece sembra più "casuale" ma potrebbe essere generato secondo una regola precisa.

S_0 è facilmente comprimibile. Possiamo dire "è composta da 9 0" (*Run-length encoding*). Arrivano quindi in considerazione i concetti di entropia e quantità di informazione.

Diciamo quindi che una stringa è *casuale* quando non può essere compressa (o non può essere compressa molto). Questo è ovviamente soggetto a eccezioni, come il caso che si tiri 100 volte testa con una moneta.
Questa definizione però si basa pesantemente sulla definizione di un "algoritmo di compressione".

Useremo intercambiabilmente il termine *incomprimibile* e *casuale*.
Assumeremo anche di trattare solo algoritmo lossless.

11.1 Compression Algorithm

Un algoritmo di compressione prende una stringa s e genera una descrizione di s più corta di s stesso. Questo implica ovviamente l'esistenza di un processo di *decompressione* simmetrico a quello di compressione.

$$
\begin{aligned}
&S \in \Sigma^* \\
&|s| = l \\
&\exists t \in \Sigma^* \ s.t \ |t| < l \\
&\exists \underbrace{m}_{unzip} .m(\underbrace{t}_{zipped}) = \underbrace{s}_{original}
\end{aligned}
$$

Ci stiamo affidando a una macchina di decompressione m, ma cosa sappiamo riguardo all'efficenza di m?

11.2 Kolmogorov

Consideriamo la dimensione non solo di t ma anche della macchina m che decomprime t. Questo equivale a avere un "self expanding executable file".

Immagina di scaricare un file compresso in un formato sconosciuto e di dover anche scaricare un programma per la decompressione del suddetto file.
La dimensione del programma di estrazione dovrebbe essere inclusa nella dimensione totale del download.

Per definire la dimensione di m fixiamo una macchina universale U.

$$
\begin{aligned}
&S \in \Sigma^* \\
&|s| = l \\
&\underbrace{(m,t)}_{description \ of \ s} \qquad s.t \ U(m,t) = s
\end{aligned}
$$

Ottimalità ovviamente a noi interessa la coppia (m,t) più piccola possibile (quella "ottimale") che riproduca s alla fine.
Un caso triviale sarebbe la macchina che copia un input e l'input originale. Questo ovviamente non è ottimale.

11.3 "Most strings are incompressible"

$$\Sigma = \{0, 1\}$$
$$S = \Sigma^{100}$$
$$|S| = |\Sigma^{100}| = 2^{100}$$

Quante stringhe $s \in \Sigma^{100}$ sono comprimibili almeno del 10%? ($C(s) = t \quad |t| \leq 90$)

Perché esista un algoritmo di compressione deve esistere alche un algoritmo di decompressione.

$$t \in \bigcup_{i=0}^{90} \Sigma^i = T$$
$$|T| = \left| \bigcup_{i=0}^{90} \Sigma^i \right| = \sum_{i=0}^{90} 2^i = 2^{91} - 1$$
$$\frac{|T|}{|S|} = \frac{1}{512}$$

Quindi una stringa su 512 di tutte quelle di lunghezza 100 o meno è comprimibile di almeno il 10%. Questo a prescindere dall'algoritmo di compressione.

Quindi "most strings are random".

Fortunatamente, le stringhe che ci interessano di solito non sono random, quindi sono comprimibili.

11.4 Kolmogorov complexity

$$D_{s,u} = \{(m, t) \in \Sigma^* \ s.t \ U(m, t) = s\}$$
$$K_U(s) = \min_{(m,t) \in D_{s,u}} (|m| + |t|)$$

Chiamiamo la *Complessità di Kolmogorov* di una stringa s (con rispetto a una macchina universale U) la lunghezza minima di m, t tali che $U(m, t) = s$.
Questa definizione di *complessità* ovviamente dipende dalla macchina universale U. Se volessimo prendere anche U in considerazione?

Supponiamo di avere due macchine universali U e V. Possiamo prendere una descrizione in U e usarla in V aggiungendo dell'overhead.

$$V(u, (m, t)) = U(m, t) = s$$

Una caratteristica particolare è che l'overhead è sempre lo stesso indipendentemente da m e t. È sempre la stessa u.

$$\forall U, V \in UTM \exists C_{U,V} = |u_v|^*$$
$$K_V(s) \leq K_U(s) + C_{U,V}$$
$$K_V = O(K_U)$$

Le due rappresentazioni sono asintoticamente equivalenti.

11.5 La Kolmogorov Complexity È Uncomputable

Prendiamo una macchina universale U. K_U **non** è computable.

Prendiamo la frase

"The smallest number that cannot be defined with less than thirteen words"

Questa frase è un paradosso, perché se trovassimo questo numero la frase gli si applicherebbe (ed essendo una frase di 12 parole lo renderebbe definibile con meno di 13 parole).
Ora dobbiamo trovare una maniera formale di dimostrare la con computabilità della complessità di Kolmogorov utilizzando questo paradosso.

- number *to* string
- defined *to* Kolmogorov complexity
- words *to* symbols

$$K_U(s) \text{The minimum \# of symbols that define } s \text{ (wrt } U)$$

Prova per assurdo Supponiamo di avere una macchina $m_{K,U} \in TM$ tale che $\forall s \in \Sigma^* m_{K,U}(s) = K_U(s)$

Algorithm 2: m

```
forall s ∈ Σ* do
    if K_U(s) > ||⌊m_{k,u}⌋|| + 1000000* then
        output s and HALT;
    end
end
```

$$||\lfloor m \rfloor|| \leq ||\lfloor m_{ku} \rfloor|| + 1000000$$

*representation of u in v

*Costante di overhead per l'esecuzione dell'algoritmo stesso

12 Reduction

Supponiamo di avere due linguaggi $L_1, L_2 \subseteq \Sigma^*$, e una funzione $f : \Sigma^* \to \Sigma^*$.
Supponiamo poi che la funzione f mappi elementi di L_1 in L_2
$(x \in L_1 \iff f(x) \in L_2)$.
Se sappiamo che L_2 è ricorsiva, possiamo dire nulla su L_1? Si, sappiamo che L_1 è recursive, in quanto possiamo costruire $m_1(x) = m_2(f(x))$.
Chiamiamo una funzione f una *"riduzione"* se si comporta un questo modo $(x \in L_1 \iff f(x) \in L_2)$. E *"Turing riduzione"* se f è computabile con una macchina di Turing.
Due linguaggi sono *"riducibili"* o *"Turing riducibili"* se esiste una f da un linguaggio all'altro.

12.1 Proprietà

Se L_1 può essere ridotto a L_2:

- L_2 recursive $\implies$ L_1 recursive
- L_1~~recursive~~ $\implies$ L_2 ~~recursive~~
- L_2 RE $\implies$ L_1 RE

13 Unione di $RE \setminus R$

Supponiamo di avere due linguaggi $L_1, L_2 \in RE \setminus R$.
Sappiamo $L_{12} = L_1 \cup L_2 \in RE$. Basta runnare m_1 e m_2 assieme e se una delle due halta haltiamo m_{12}.
Ma possiamo dire se L_12 in RE o in $RE \setminus R$? No. Non possiamo imporre alcun constrain sul fatto che sia *recursive*.

Proof. Questo è provato definendo E l'insieme di tutte le stringhe rappresentanti TM con un numero pari di stati. Poi definiamo O l'insieme di tutte le macchine con numero dispari di stati.
Ora $L_1 = E \cup HALT_\epsilon$ e $L_2 = O \cup HALT_\epsilon$.

Esercizio

$$L \in \Sigma^* \text{ recursive}$$

$$TM\ m \text{ Partially decides } L$$

$$\forall x \in \Sigma^* m(x) = \infty \vee m(x) = \begin{cases} 0 & x \notin L \\ 1 & x \in L \end{cases}$$

x	ϵ	0	1	00	01
$m(x)$	0	∞	1	0	∞

Possiamo dire che una macchina m *decide parzialmente* L se: "Quando halta da la risposta giusta". Trivialmente la macchina che non halta mai decide parzialmente tutti i linguaggi.

domanda La proprietà "partially decides L" è decidibile?
Iniziamo vedendo se possiamo usare il teorema di Rice.

La proprietà è triviale? No, possiamo facilmente creare una macchina che non soddisfa la proprietà.

La proprietà è semantica? No. Quindi non possiamo usare Rice.
Dimostrazione che la proprietà non è semantica.

$$L = \{0, 01\}$$

x	ϵ	0	1	00	01
$m_1(x)$	∞	∞	1	∞	∞
$m_2(x)$	0	0	1	0	0

Le due macchine riconoscono lo stesso linguaggio $L' = \{1\}$. Ma la prima ha la proprietà, mentre la seconda no.□

Dimostrazione che P non è decidibile Questo può essere dimostrato con la stessa prova che abbiamo usato per dimostrare il teorema di Rice. Ovvero riducendo la macchina H che risolve l'halting problem a quella che decide la proprietà P.
$(m, t) \mapsto n_{m,t}(x)$

Algorithm 3: n

move x on aux tape;
replace it with t;
run m_m(t);
restore original input x;
run $m_A(x)$

esercizio

$\Sigma = \{\llcorner, 0, 1\}$
$Q = \{q_0, q_1, \ldots, q_n\}$
q_0 initial state
$f : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$

Vediamo come possiamo "modificare" una TM.

1. Se la posizione iniziale della TM fosse sconosciuta (quindi la posizione dell'input) possiamo creare una macchina che si posiziona all'inizio dell'input? Si. Dovrebbe fare la spola a destra e sinistra spostando dei marker fino a che non trova l'input.

2. Ipotizziamo che la macchina abbia delle celle "difettose". $\Sigma' = \Sigma \cup \{x\}$. Possiamo modificare Q e f per ignorare le celle difettose?

Per ogni stato q_n creiamo 4 stati $q_{nl1}, q_{nl2}, q_{nr1}, q_{nr2}$

	x	$\sqcup$	0	1
q_{nl1}	$x, q_{nl1}, \leftarrow$	$\sqcup, q_{nl2}, \rightarrow$	$0, q_{nl2}, \rightarrow$	$0, q_{nl2}, \rightarrow$
q_{nl2}	$x, q_n, \leftarrow$	$\sqcup, q_n \leftarrow$	$0, q_n \leftarrow$	$1, q_n \leftarrow$
q_{nr1}	$x, q_{nr1}, \rightarrow$	$\sqcup, q_{nr2}, \leftarrow$	$0, q_{nr2}, \leftarrow$	$0, q_{nr2}, \leftarrow$
q_{nr2}	$x, q_n, \rightarrow$	$\sqcup, q_n \rightarrow$	$0, q_n \rightarrow$	$1, q_n \rightarrow$

Dopodiché, nella funzione di transizione originale, sostituiamo ogni regola del tipo $(\sigma, q_n, \leftarrow)$ con $(\sigma, q_{nl1}, \leftarrow)$ e ogni regola del tipo $(\sigma, q_n, \rightarrow)$ con $(\sigma, q_{nr1}, \rightarrow)$

14 Computational Complexity

Da ora in poi prenderemo in considerazione solo linguaggi ricorsivi. Quindi c'è sempre una macchina di Turing che li riconosce.

$$L \text{ is recursive}$$
$$\exists m : L(m) = L \wedge \forall s \in \Sigma^* m(s) < \infty$$

Data una stringa $s \in \Sigma^*$, $t_m : \Sigma^* \mapsto \mathbb{N}$ è una funzione che mappa s al *numero di step* svolti da $m(s)$ prima di interrompersi.
$T_m : \mathbb{N} \mapsto \mathbb{N} : n \mapsto \max_{|s|=n} t_m(s)$ ci dice il numero di step in relazione alla *dimensione* dell'input.

es. Prendiamo la funzione f che decide se un numero è pari o no

$$\Sigma = \{\sqcup, 0, 1\}$$
$$f : \mathbb{N} \mapsto \{0, 1\}$$
$$n \mapsto \begin{cases} 1 & even \\ 0 & odd \end{cases}$$

Assumiamo che i numeri siano rappresentati in binario e che m si comporti così:

1. scorri a destra fino a fine del numero (n steps)
2. fai un controllo sull'ultimo bit (1 step)
3. scorri a sinistra cancellando il numero (n steps)

4. scrivi l'output (1 step)

$$T_m(\underbrace{\qquad n \qquad}_{\text{size of representation of } m}) = 2n + 2 = O(n)$$

14.1 Deterministic Time

$$DTIME(f) = \{L \subseteq \Sigma^* : \exists m.L(m) = L \wedge T_m(n) = O(f(n))\}$$

14.2 Polynomial Languages

$$\mathbb{P} = \bigcup_{k=0}^{\infty} DTIME(n^k)$$

Input types Se il problema che stiamo affrontando non prende espressamente stringhe come input dobbiamo escogitare un encoding per l'input in maniera che una Turing machine possa lavorarci.
Nel caso di un grafo possiamo decidere di usare una matrice di adiacenza.

$$A \in \{0,1\}^{n \times n}$$

$$A_{ij} = \begin{cases} 1 & \{i,j\} \in E \\ 0 & else \end{cases}$$

input size $= O(n^2)$

14.3 Prime numbers

Algorithm 4: is_prime

```
function is_prime(n ∈ ℕ) {;
for i = 2 ... √n do
    if n%i == 0 then
        reject;
    end
end
accept;
}
```

La dimensione dell'input è $|s| = O(\log n)$. Quindi $n = O(2^{|s|})$. $\sqrt{n} = O(2^{|s|/2})$. I numeri sono esponenzialmente proporzionali al numero di bit usati per rappresentarli. Quindi questo algoritmo per decidere se un numero è primo non è polinomiale.
Ci sono algoritmi più sofisticati che possono decidere se un numero è primo in tempo polinomiale con rispetto alla quantità di bit usati.

Satisfiability Date n variabili booleane $x_1, \ldots, x_n$, una congiunzione di disgiunzioni $(x \wedge x \wedge \cdots \wedge x) \vee (x \wedge x \wedge \cdots \wedge x)$

- term(atom): a variable x or its negation
- clause: $c_j = \vee i = 1^{n_j} t_{ij}$
- formula: $\wedge_{j=1}^{m} c_j$

Una formula è *soddisfacibile* se esiste una combinazione di variabili tale per cui risulta vera.
Il problema SAT è trovare questa combinazioni (almeno una di queste combinazioni).
Non abbiamo un algoritmo che risolva questo problema in tempo polinomiale.
Quindi non possiamo dire se SAT sia in $\mathbb{P}$ o meno.

14.4 Integer Linear Programming

14.5 Certificazione

Clique, *D ILP*, e *SAT* non hanno algoritmi polinomiali che trovino una soluzione. Ma, data una soluzione, verificare che essa sia valida o meno è molto semplice (polinomiale). Quindi la difficoltà sta nel fare questa semplice prova su un numero smisurato di possibili soluzioni.

Problemi difficili da controllare. Non tutti i problemi condividono questa proprietà. Prendiamo l'esempio di un gioco da tavolo (come scacchi) e di voler decidere se la board è a favore di un giocatore o dell'altro. L'input del problema è limitato, essendo ristretto alla rappresentazione della board, ma ogni decisione del giocatore vincente non può essere realisticamente fatta senza simulare ogni possibile evoluzione del gioco. Questo è ovviamente computazionalmente complesso (esponenziale), a discapito dell'input limitato.

Definiamo più formalmente questa proprietà.

$$\forall x \in L$$
$$\exists M \text{ verifying machine}$$
$$\exists \underbrace{m}_{\text{certificate}} \in \Sigma^* : M(x, m) = 1$$
$$\exists p, q \text{ polinomiali}. |m| < p(|x|) \wedge t_m(x, m) \leq q(|x|)$$

Più formalmente Diciamo L possa essere SAT, Clique, o ILP.
Assumiamo poi

$$\exists m : x \in L \iff m(x) = 1 \text{ and } \forall k : T_m \neq O(n^k)$$
$$\exists m : x \in L \iff \exists c \in \Sigma^* : m(x, c) = 1 \wedge T_m(|x| + |c|) \leq q(|x| + |c|)$$

Con q, p polinomiali e $|c| \leq p(|x|)$

15 Non-deterministic Turing machines

Come ogni Turing machine, abbiamo un alfabeto Σ e un set di stati Q. La funzione di transizione è ciò che varia.
Una scelta implementatici può essere quella di definire più funzioni di transizione $f_1, \dots, f_n$.

$$f_1 : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$$
$$\vdots$$
$$f_n : \Sigma \times Q \to \Sigma \times Q \times \{\leftarrow, \rightarrow\}$$

In ogni stato della TM abbiamo tutti gli archi in uscita dettati dalle varie funzioni di transizione.
Un'altra opzione è quella di tenere una singola funzione di transizione ma modificarne il codominio

$$f_1 : \Sigma \times Q \to \mathcal{P}(\Sigma \times Q \times \{\leftarrow, \rightarrow\})$$

Prendiamo una macchina NTM N. Come possiamo dire che N *decide* L?
Innanzitutto, la macchina deve haltare. Quindi tutte le possibili branch devono arrivare a uno stato finale (anche diversi stati finali). Questo esclude la presenza di cicli nel grafo.
Se tutti gli stati finali accettano o rigettano l'input, la macchina *riconosce* L.
Se gli stati finali non concordano, ma almeno uno accetta, la macchina *decide* L
Per una macchina non deterministica prendiamo il tempo di computazione $t_n(x)$ come la profondità dell'albero di computazione di n sull'input x. Quindi prendiamo sempre il worst case scenario.

$$T_N(n) = \max_{s \in \Sigma^n} t_n(s)$$

def.

$$\mathbb{NP} = \{L \subseteq \Sigma : \exists m \ DTM \ \exists p, q \ poly \ s.t \forall x \in \Sigma^* \cdots\}$$
$$\mathbb{NP} = C = \bigcup_k \text{NTIME}(n^k)$$

15.1 Independent Set

$$IS = \{(G = (V, E), k) : G \text{ undirected}, k \in \mathbb{N}, 1 \leq k \leq |\mathbb{N}| \\ \exists V' \subseteq V \text{ s.t. } |V'| = k \text{ s.t. } \forall i, j \in V' \{i, j\} \notin E\}$$

Più intuitivamente, un *independent set* è un sottoinsieme dei vertici di un grafo tale che non esista alcun arco che colleghi direttamente i nodi.
La grandezza del set è dettata da un numero k dato. IS è $\mathbb{NP}$. Questo può essere facilmente dimostrato dal fatto che IS è equivalente a Clique sul grafo complementare a G.

15.2 Reductions (polynomial)

Ricordiamo che una *reduction function* è una funzione (o una macchina di Touring) che mappa tutte e solo le stringhe di un linguaggio L a stringhe di un linguaggio L'. Se definiamo una funzione di riduzione **polinomiale** e la usiamo per ridurre un problema p ad un altro problema $p' \in \mathbb{NP}$, possiamo dire che p sia in $\mathbb{NP}$. Una riduzione di questo tipo si chiama *Polynomial Reduction*

$$indset <_p clique$$
$$clique <_p indset$$

Quindi possiamo dire che i due problemi sono equivalenti

> $sat <_p ilp$**.** La dimostrazione consiste solo nel fornire una funzione di riduzione. È lunga da spiegare.

15.3 k-CNF

Possiamo specializzare il problema CNF limitando il numero di termini che possono essere in ogni clausola.
Per esempio in $3-cnf$ ogni clausola non può avere più di 3 termini.

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee x_4) \wedge \cdots \wedge (\dots)$$

Altre formulazioni del problema impongono che le clausole abbaino esattamente k termini. Ma sono equivalenti.
È palese che $3-cnf \subseteq cnf$. Ma possiamo dimostrare che $3SAT <_p SAT$

$$f : \Sigma^* \to \Sigma^*$$
$$F \mapsto \begin{cases} F & F \text{ is } 3-cnf \\ x_1 \vee \neg x_1 & F \text{ not in } 3CNF \end{cases}$$

Questa funzione riduce ogni problema 3-sat in un problema sat equivalente.
È poi possibile ridurre un problema sat qualsiasi in un problema 3-sat equivalente. Per fare questo basta dividere tutte le clausole con più di 3 termini in più clausole a 3 termini.

$$c_i = (t_{i1} \vee t_{i2} \vee \cdots \vee t_{in} \vee)$$
$$if n_i > 3 :$$
$$c'_i = (t_{i1} \vee t_{i2} \vee x'_1) \wedge (\neg x'_1 \vee t_{i3} \vee \cdots \vee t_{in})$$

Aggiungiamo nuovi termini e ripetiamo questo processo iterativamente fino a che non abbiamo un problema 3-sat.
Quindi possiamo dire che sat e 3-sat sono perfettamente equivalenti.

3sat to indset riduciamo il problema 3 sat a uno di independent set costruendo un grafo specializzato a partire dalla formula 3-dnf del problema sat.

> **nb.** Il problema independent set deve essere del tipo m-independent set, dove m è il numero di clausole nel problema 3-sat.

16 Cook-Levin theorem

Ogni linguaggio (o problema) in $\mathbb{NP}$ può essere ridotto a SAT (e quindi anche a 3-SAT).

Recap di cosa voglia dire NP Un problema L è NP se possiamo definire una macchina non deterministica n tale che $n(x) = 1 \iff x \in L$. Il tempo di esecuzione per questa macchina deve essere dominato da una funzione polinomiale sulla dimensione dell'input.
Possiamo vedere le ramificazioni della macchina non deterministica come un albero, e sogni divergenza come un Problema sat.
Prendiamo una funzione di transizione f per una macchina di Turing.
Possiamo tradurre questa macchina in un circuito binario.

$$
\begin{array}{c}
(q_1, \dots, q_k) \in \{0,1\}^k \\
Q \quad k \geq \lceil \log_2(|Q| + 1) \rceil \\
\Sigma = \{0,1\} \\
\delta = \left| \begin{array}{l} 0 \mapsto \leftarrow \\ 1 \mapsto \rightarrow \end{array} \right.
\end{array}
$$

Questo circuito può essere rappresentato da una truth-table. Se abbiamo due funzioni di transizione, possiamo aggiungere un bit extra a ogni tabella per differenziare le due.

Facciamola corta Ogni problema in NP può essere ridotto a 3-SAT

16.1 NP-complete

Un linguaggio è NP-completo se ogni problema in NP può essere ridotto polinomialmente a esso.
Siccome abbiamo dimostrato che 3-SAT riduce a INDSET e che 3-SAT è NP-complete, possiamo dire che INDSET è NP-complete.

17 2024-11-25

Abbiamo constatato che NP contiene P e NP-complete.

2-sat Un problema 2 sat è come 3-sat ma ogni clausola ha 2 termini e basta. Quindi, sapendo che $a \implies b$ è equivalente a $\neg a \vee b$, possiamo riscrivere ogni clausola come un'implicazione (con le opportune negazioni).
Possiamo poi costruire un grafo orientato dove i nodi sono i termini e le loro negazioni, e gli archi sono le implicazioni. È dimostrato che la formula originale è satisfiable se e solo se non co sono path tra un termine e la sua negazione.

17.1 3-vertex coloring

Dato un grafo non orientato, possiamo assegnare una label a ogni nodo in modo che due nodi con la stessa label non si tocchino?
Questo problema è NP-complete, in quanto 3-sat può essere ridotto a 3-vertex coloring.

18 Everything is in EXP

Prendiamo un linguaggio L in NP qualsiasi. Quindi
$x \in L \iff \mathcal{N}(x) = 1 \quad t_{\mathcal{N}}(x) \leq p(|x|)$ Possiamo simulare la macchina non deterministica $\mathcal{N}$ con una macchina deterministica? NO.
Questo è facilmente dimostrabile partendo dalla rappresentazione della macchina non deterministica come un albero binario, dove ogni bivio è una divergenza nel non determinismo. I possibili outcome di questa macchina (quindi le foglie) sono $2^{p(|x|)}$.
Quindi per esplorarli tutti in maniera deterministica ci servirebbero $q(|x|) \cdot 2^{p(|x|)}$ step. Questa è una funzione di complessità esponenziale quindi la macchina deterministica *non può* essere polinomiale.
La classe $EXP = \bigcup_{k=0}^{\infty} DTIME(2^{n^k})$ contiene NP, $coNP$, e P

18.1 Linguaggio EXP non NP

Esistono linguaggi L in EXP e non in NP? Pensiamo di si, ma non lo sappiamo.
Questo però non ci ferma dal definire la sottoclasse di EXP contenente i linguaggi $EXP - complete$. Ovvero tutti i linguaggi a cui ogni problema in EXP può essere ridotto polinomialmente.
C'è il problema *restricted halt* che è $EXP - complete$, ma non sappiamo dimostrare che sia fuori da NP.

19 space complexity

Non ha senso parlare di macchine con una complessità di tempo inferiore a quella polinomiale, perché anche la macchina migliore deve almeno leggere l'input.
Se parliamo dello spazio occupato in memoria invece, possiamo. Da ora in poi assumiamo ogni macchina abbia 2 nastri, uno di input e uno di lavoro. La complessità viene calcolata solo in base alla dimensione necessaria del nastro di lavoro. Facciamo anche l'assunzione che il nastro di input sia di sola lettura.

$$\mathbb{P} \subseteq \mathbb{P}space$$
$$\mathbb{NP} \subseteq \mathbb{P}space$$
$$\mathbb{P}space \subseteq \mathbb{EXP}$$