Elementi di Calcolo e Complessità

Federico Matteoni

A.A. 2019/20

Indice

Introduzione

Prof. Pierpaolo Degano `pierpaolo.degano@unipi.it`
Con Giulio Masetti `giulio.masetti@isti.snr.it`
Esame: compitini/scritto + orale

Capitolo 1

Calcolabilità

1.1 Teoria della Calcolabilità

Illustra **cosa può essere calcolato da un computer** senza limitazioni di risorse come spazio, tempo ed energia. Vale a dire:

- Quali sono i **problemi *solubili*** mediante una **procedura effettiva** (qualunque linguaggio su qualunque macchina)?
- Esistono **problemi *insolubili***? Sono interessanti, realistici, oppure puramente artificiali?
- Possiamo raggruppare i problemi in **classi**?
- Quali sono le **proprietà** delle classi dei problemi solubili?
- Quali sono le relazioni tra le classe dei problemi insolubili?

Astrazione Utilizzeremo **termini astratti per descrivere la possibilità di eseguire un programma ed avere un risultato**. Questa astrazione è un **modello** che non tiene conto di dettagli al momento irrilevanti.
Un po' come l'equazione per dire quanto ci mette il gesso a cadere che non tiene conto delle forze di attrito dell'aria.

Problema della Decisione Un problema è risolto se si conosce una **procedura**, che con un numero **finito** di operazioni, permette di decidere se una proposizione logica è vera o falsa.

1.2 Algoritmo

Algoritmo Insieme **finito** di istruzioni.

Istruzioni Elementi da un insieme di **cardinalità finita**, ognuna ha **effetto limitato** (localmente e "*poco*") sui dati, che devono essere **discreti**. Un'istruzione deve richiedere tempo finito per essere elaborata.

Computazione **Successione di istruzioni finite** (**passi discreti**) in cui **ogni passo dipende solo dai precedenti**. Ogni passo dipende da una **porzione finita dei dati** in modo **deterministico**.
Non c'è limite alla memoria necessaria al calcolo (è finita ma illimitata). Neanche il tempo è limitato (necessario al calcolo). Tanto tempo e tanta memoria quante ce ne servono.

Eccezioni Un'eccezione a questa definizione di algoritmo è costituita dalle macchine concorrenti/interattive, dove gli input variano nel tempo. Inoltre vi sono formalismi che tengono conto di algoritmi probabilistici e stocastici. Altre eccezioni sono gli algoritmi non deterministici, ma per ognuno di essi esiste un algoritmo deterministico equivalente.

In sintesi Un numero finito di istruzioni da un set finito di istruzioni possibili, un numero discreto di passi di tempo finito e ognuno dipendente solo dai precedenti, senza limite di passi o spazio durante l'esecuzione.

1.3 Macchina di Turing

Introdotta da **Alan Turing** nel 1936, confuta la speranza *"non ignorabimus"* di poter risolvere qualsiasi cosa con un programma.
Turing originariamente la presenta supponendo di aver un impiegato precisissimo ma stupido, con una pila di fogli di carta ed una penna, ed un foglio di carta con le istruzioni che esegue con estrema diligenza. Non capisce quello che fa, e si chiama "**computer**".

Struttura matematica Una Macchina di Turing (MdT) è una quadrupla:

$$\mathrm{M} = (\mathrm{Q}, \sum, \delta, \mathrm{q}_0)$$

Q = $\{q_i\}$ è l'**insieme finito degli stati** in cui si può trovare la macchina.
Indicheremo con lo stato speciale $h \notin$ Q la fine corretta della computazione.

$\Sigma = \{\sigma, \sigma' \dots\}$ è l'**insieme finito di simboli**.
Ci sono elementi che devono per forza esistere:

carattere **bianco**, vuoto

$\triangleright$ carattere di inizio della memoria, chiamato **respingente**, che funziona come un inizio file

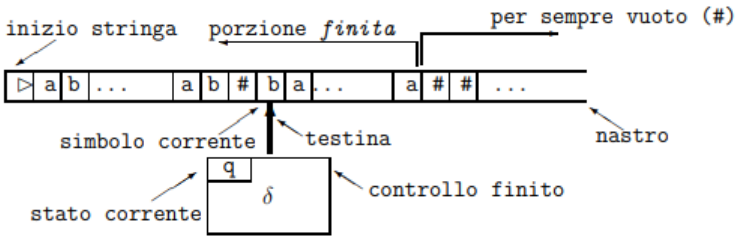
L, R, - $\notin \Sigma$ per indicare di spostare il cursore, rispettivamente, a sinistra, destra o rimanere fermo.

$\delta \subseteq (\mathrm{Q} \times \Sigma) \to (\mathrm{Q}' \bigcup \{h\}) \times \Sigma' \times \{\mathrm{L}, \mathrm{R}, -\}$ è **funzione di transizione**.
Mantiene determinismo perché funzione, ad un elemento associa un solo elemento (la transizione è univoca), e le transizioni finite perché prodotto cartesiano di insiemi finiti. Imponiamo la condizione di funzione cioè che
$\forall\,(q,\sigma) \to (q',\sigma',D'),(q,\sigma) \to (q'',\sigma'',D'') \Rightarrow q' = q'', \sigma' = \sigma',' D' = D''$
δ(q, $\triangleright$) = (q', $\triangleright$, R), cioè se sono a inizio file possono solo andare a destra.
Può essere vista come una relazione di transizione, $\delta \subseteq (Q \times \Sigma) \times (Q \bigcup \{h\}) \times \Sigma \times \{L, R, -\}$

$q_0 \in$ Q lo **stato iniziale**

Quindi una **configurazione** è una quadrupla $(q, u, \sigma, v) \in ((Q \cup \{h\}) \times \Sigma^* \times \Sigma \times \Sigma^F)$ con $\Sigma^F = \Sigma^* \cdot (\Sigma - \{\#\}) \cup \{\epsilon\}$
σ è il simbolo attualmente sotto la testina della MdT, e una notazione breve è $(q, u\underline{\sigma}v)$ o (q, w) quando la posizione non è importante.

Una MdT può essere così rappresentata



Mappatura a coda di rondine, bigezione tra (m, n) $\to$ k, cioè $N^2 \to N$.
Costruire un modello per il calcolo dopo aver posto delle condizioni affinché qualcosa si possa chiamare algoritmo.

1.3.1 Σ

$\Sigma^0 = \{\epsilon\}$, con ϵ = parola vuota, che non contiene caratteri

$\Sigma^{i+1} = \Sigma \cdot \Sigma^i = \{\sigma \cdot u \mid \sigma \in \Sigma \wedge u \in \Sigma^i\}$

$\Sigma^* = \bigcup_{i \in N} \sum^i$, insieme di tutte le possibili combinazioni di simboli

$\Sigma^f = \Sigma^* \cdot (\Sigma - \{\#\} \bigcup \{\epsilon\})$, cioè l'insieme di tutte le stringhe che terminano con un carattere non bianco ma che possono terminare con la stringa vuota

Esempio $\Sigma_B = \{0,1\} \longrightarrow \Sigma_B^* = \{\epsilon, 0, 1, 01, 10, 010, 110010, \dots\}$ tutti i numeri binari

1.3.2 Transizioni

La **situazione corrente** di una macchina di Turing può essere scritto come (q, u, σ, v) dove:

q è lo **stato attuale**, $q \in Q$

u è la **stringa a sinistra** del carattere corrente, $u \in \sum^*$

σ è il **carattere corrente**, $\sigma \in \sum$

v è il **resto della stringa** che termina con un carattere non nullo, $v \in \sum^f$

Può anche essere più comodamente espressa come $(q, u\,\underline{\sigma}\,v)$

1.3.3 Computazione

Una computazione è una transizione $(q, x) \longrightarrow (q', \omega)$. Una macchina di Turing parte **sempre** da $(q_0, \underline{\triangleright}x)$.
Ogni computazione può esprimere il numero di passi necessari, ad esempio $\gamma \longrightarrow^n \gamma'$.
$\forall$ *computazione* $\gamma \Rightarrow \gamma \longrightarrow^0 \gamma$. Inoltre se $\gamma \longrightarrow \gamma' \wedge \gamma' \longrightarrow^n \gamma''$ allora $\gamma \longrightarrow^{n+1} \gamma''$

Convergenza e divergenza Diremo che la computazione $(q_0, w) \rightarrow^n (q', w')$ *termina* (cioè **converge** $\downarrow$) $\Leftrightarrow q' = h$ e che *non termina* (cioè **diverge** $\uparrow$) $\Leftrightarrow \forall\, q', w' \mid (q_0, w) \rightarrow^n (q', w')\, \exists\, q'', w'' \mid (q', w') \rightarrow (q'', w'')$

Possiamo specificare la macchina M con cui eseguiamo la computazione con $\rightarrow^n_M$, usando M(w) per specificare che la macchina M inizia la computazione da $(q_0, \underline{\triangleright}w)$, cioè che applichiamo M a w.

Esempio Macchina di Turing che esegue la semplice somma di due semplici numeri romani.

q	σ	$\delta(q, \sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	I	(q_0, I, R)
q_0	$+$	(q_1, I, R)
q_1	I	(q_1, I, R)
q_1	$\#$	$(q_2, \#, L)$
q_2	I	$(h, \#, -)$

$(q_0, \underline{\triangleright}II + III) \longrightarrow (q_0, \triangleright\underline{I}I + III) \longrightarrow (q_0, \triangleright I\underline{I} + III) \longrightarrow$
$(q_0, \triangleright II\ \underline{+}\ III) \longrightarrow (q_1, \triangleright III\underline{I}II) \longrightarrow (q_1, \triangleright IIII\underline{I}I) \longrightarrow$
$(q_1, \triangleright IIIII\underline{I}) \longrightarrow (q_1, \triangleright IIIIII\underline{\#}) \longrightarrow (q_2, \triangleright IIIIII\underline{I}) \longrightarrow$
$(h, \triangleright IIIII)$

Esempio Macchina di Turing che verifica se una stringa di lettere a, b è palindroma o no.

q	σ	$\delta(q, \sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	a	$(q_A, \triangleright, R)$
q_0	b	$(q_B, \triangleright, R)$
q_0	$\#$	$(h, \#, -)$
q_A	a/b	$(q_A, a/b, R)$
q_A	$\#$	$(q_{A'}, \#, L)$
$q_{A'}$	a	$(q_R, \#, L)$
q_B	a/b	$(q_B, a/b, R)$
q_B	$\#$	$(q_{B'}, \#, L)$
$q_{B'}$	a	$(q_R, \#, L)$
q_R	a/b	$(q_R, a/b, R)$
q_R	$\triangleright$	$(q_0, \triangleright, R)$

$(q_0, \underline{\triangleright}abba) \longrightarrow (q_0, \triangleright\underline{a}bba) \longrightarrow (q_A, \triangleright\triangleright\,\underline{b}ba) \longrightarrow$
$(q_A, \triangleright\triangleright\, b\underline{b}a) \longrightarrow (q_A, \triangleright\triangleright\, bb\underline{a}) \longrightarrow (q_A, \triangleright\triangleright\, bba\underline{\#}) \longrightarrow$
$(q_{A'}, \triangleright\triangleright\, bb\underline{a}) \longrightarrow (q_R, \triangleright\triangleright\, b\underline{b}) \longrightarrow (q_R, \triangleright\triangleright\,\underline{b}b) \longrightarrow (q_R, \triangleright\underline{\triangleright}bb)$
$\longrightarrow (q_0, \triangleright\triangleright\,\underline{b}b) \longrightarrow (q_B, \triangleright\triangleright\triangleright\underline{b}) \longrightarrow (q_B, \triangleright\triangleright\triangleright b\underline{\#}) \longrightarrow$
$(q_{B'}, \triangleright\triangleright\triangleright\underline{b}) \longrightarrow (q_R, \triangleright\triangleright\underline{\triangleright}) \longrightarrow (h, \triangleright\triangleright\triangleright)$

1.4 Linguaggi di Programmazione

Un primo formalismo di algoritmo, come abbiamo visto, è la **macchina di Turing**: attenendosi alle richieste di tempo e spazio arbitrariamente grandi ma finiti, risolve un **problema**.
Un secondo formalismo sono i **linguaggi di programmazione**.

1.4.1 Sintassi

Sintassi astratta Definiamo la **sintassi** dello scheletro di un semplice linguaggio di programmazione imperativo. Una **sintassi astratta** è una sintassi non concreta, cioè che non tiene conto di alcune cose come la procedenza tra gli operatori.
Sintassi

Expr $\to$ E ::= x | n | E + E | E $\cdot$ E | E - E

Bexpr $\to$ B ::= tt | ff | E $<$ E | $\neg$B | B $\vee$ B

Comm $\to$ C ::= `skip` | x = E | C;C | `if` B `then` C `else` C | `for` i = E `to` E `do` C | `while` B `do` C

Abbiamo una serie di insiemi da definire ulteriormente

x $\in$ Var, l'insieme delle **variabili**

n $\in N$, **numeri naturali**.

Abbiamo anche la **memoria** per poter **assegnare ad una variabile il suo significato**

σ : Var $\to_{fin} N$
Si dice "a dominio finito", indicata dal fin sotto la freccia, per indicare che il dominio Var ha cardinalità finita. Var dominio è quindi un sottoinsieme di Var insieme delle variabili che sarebbe infinito.
La memoria si può aggiornare, diventando $\sigma' = \sigma[\mathrm{x} \mapsto \mathrm{n}]$.
Ad esempio, $\sigma'(y) = \mathrm{n}$ se y = x, altrimenti $\sigma'(y) = \sigma(y)$

Chiameremo `WHILE` il linguaggio definito dalla grammatica BNF definita sopra, e `FOR` il linguaggio risultante dall'omissione del comando `while` B `do` C nella definizione.

1.4.2 Funzioni di Valutazione

Inoltre, per valutare le espressioni generate dalla grammatica, servono delle **funzioni di valutazione**. Esse **trovano il significato di ogni espressione**

Funzione di valutazione delle espressioni
$\mathscr{E}$: Expr $\times$ (Var $\to N$) $\to N$
La sua **semantica denotazionale** è la seguente

$\mathscr{E}[\mathrm{x}]_\sigma = \sigma(\mathrm{x})$ $\qquad$ $\mathscr{E}[\mathrm{E}_1 \pm \mathrm{E}_2]_\sigma = \mathscr{E}[\mathrm{E}_1]_\sigma \pm \mathscr{E}[\mathrm{E}_2]_\sigma$
$\mathscr{E}[\mathrm{n}]_\sigma = \mathrm{n}$

Importante notare come gli operatori $+$, $-$, $\cdot$ *dentro* le espressioni siano dei **semplici token denotazionali**, mentre sono gli operatori *valutati* ad eseguire il vero e proprio calcolo. Per chiarire questo aspetto, facciamo un esempio. Valutiamo con la nostra funzione $\mathscr{E}[\mathrm{E}_1 + \mathrm{E}_2]_\sigma = \mathscr{E}[\mathrm{E}_1]_\sigma$ *più* $\mathscr{E}[\mathrm{E}_2]_\sigma$. **Se non definiamo l'operatore** "*più*", ponendo $\sigma(\mathrm{x}) = 25$ la valutazione

$\mathscr{E}[3 + \mathrm{x}]_\sigma = \mathscr{E}[3]_\sigma$ *più* $\mathscr{E}[\mathrm{x}]_\sigma = 3$ *più* $25 = 42$

è corretta quanto

$\mathscr{E}[3 + \mathrm{x}]_\sigma = \mathscr{E}[3]_\sigma$ *più* $\mathscr{E}[\mathrm{x}]_\sigma = 3$ *più* $25 = 28$

Ovviamente utilizzeremo la valutazione specificata in precedenza e gli operatori aritmetici assumeranno il loro significato standard.
L'unica eccezione è l'operatore $-$, che nel nostro caso sarà il **meno limitato** dal simbolo $\dot{-}$, la cui unica differenza è che non può dare un risultato inferiore a 0. Ad esempio, $5 \dot{-} 7 = 0$

Funzione di valutazione di espressioni booleane
$\mathscr{B}$: Bexpr $\times$ (Var $\to N$) $\to$ {tt, ff}
La cui **semantica denotazionale** è la seguente

$$\mathscr{B}[\text{tt}]_\sigma = \text{tt} \qquad \mathscr{B}[\neg \text{B}]_\sigma = \neg \mathscr{B}[\text{B}]_\sigma$$
$$\mathscr{B}[\text{ff}]_\sigma = \text{ff}$$
$$\mathscr{B}[\text{E}_1 < \text{E}_2]_\sigma = \mathscr{E}[\text{E}_1]_\sigma < \mathscr{E}[\text{E}_2]_\sigma \qquad \mathscr{B}[\text{B}_1 \vee \text{B}_2]_\sigma = \mathscr{B}[\text{B}_1]_\sigma \vee \mathscr{B}[\text{B}_2]_\sigma$$

Anche qua vale il medesimo discorso sulla definizione sugli effettivi operatori.

1.4.3 Semantica Operazione Strutturale

Structural Operationale Semantics Metodo attraverso il quale viene fornita la semantica dei comandi. Parte da un **insieme di configurazioni** Γ

$$\Gamma = \{(\text{C}, \sigma) \mid \text{FV(C)} \subset \text{dom}(\sigma)\} \cup \{\sigma\}$$

dove FV(C) sono le **variabili del programma** e con FV(C) $\subset$ dom(σ) si richiede che tutte le variabili del programma abbiano un valore nella memoria fornita. Si fa l'unione con la sola memoria σ perché la situazione finale è (, σ) che, analogamente allo stato fittizio h nella macchina di Turing, segnala la fine dell'esecuzione. Inoltre si hanno le **transizioni** $\to$

$$\to \subset \Gamma \times \Gamma$$

Definiamo quindi un **insieme di transizioni (Γ, $\to$)** tramite delle **regole di inferenza** del tipo $\frac{premesse}{conclusione}$. In assenza di premesse, $-$, la regola di inferenza si dice **assioma**.

$$\frac{-}{(skip, \sigma) \to \sigma} \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_1, \sigma)}\ se\ \mathscr{B}[B]_\sigma = tt$$

$$\frac{-}{(x = E, \sigma) \to \sigma[x \mapsto n]}\ se\ \mathscr{E}[E]_\sigma = n \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_2, \sigma)}\ se\ \mathscr{B}[B]_\sigma = ff$$

$$\frac{(C_1, \sigma) \to (C_1', \sigma')}{(C_1; C_2, \sigma) \to (C_1'; C_2, \sigma')} \qquad \frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \to \sigma}\ se\ \mathscr{B}[E_2 < E_1]_\sigma = tt$$

$$\frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \to (i = n_1;\ C;\ for\ i = n_1 + 1\ to\ n_2\ do\ C, \sigma)}\ se\ \mathscr{B}[E_2 < E_1]_\sigma = ff \wedge [E_1]_\sigma = n_1 \wedge [E_2]_\sigma = n_2$$

$$\frac{-}{(while\ B\ do\ C, \sigma) \to (if\ B\ then\ C;\ while\ B\ do\ C, \sigma)}$$

1.5 Calcolabilità

1.5.1 T-Calcolabile

Dati Σ alfabeto della macchina, Σ_0 alfabeto di input e Σ_1 alfabeto di output, con $\#, \triangleright \notin \Sigma_0 \cup \Sigma_1 \subset \Sigma$

$$\text{M=(Q, } \Sigma, \delta, \text{ q}_0) \text{ calcola } f : \Sigma_0^* \longrightarrow \Sigma_1^* \Leftrightarrow (\forall w \in \Sigma_0^* \wedge f(w) = z \Rightarrow M(w) \to_{fin} (\text{h}, \triangleright z))$$

Si dice che la **funzione** f è **T-Calcolabile**.
Cioè, esiste una macchina di Turing che per ogni stringa finita in input arriva, con un numero finito di passi, all'arresto lasciando sul nastro la stringa di output corretta. Notare come non viene data nessuna interpretazione al risultato della f.

1.5.2 `while`-Calcolabile

$$\text{C calcola } f : \text{Var} \to N \Leftrightarrow (\forall\ \sigma : \text{Var} \to N \wedge f(x) = n \Rightarrow \text{C}(\sigma) \to_{fin} \sigma' \wedge \sigma'(x) = n)$$

Si dice che la funzione f è **`while`-Calcolabile**.
Cioè esiste un programma C che calcola il risultato corretto in un numero finito di passi.

Invariante Tutti i risultati visti fin'ora **sono invarianti rispetto al modello dei dati**, e questo vale anche per la T-Calcolabilità e la `while`-Calcolabilità.
In particolare, se ho i dati in un formato A allora posso codificarli nel formato B in cui opera la macchina, calcolare il risultato in formato B e decoficarlo nel formato A di partenza. Questo vale se **le codifiche sono funzioni biunivoche e "*facili*"**. Vedremo cosa significa essere "facili", ma per adesso basti pensare ad un numero finito di passi e che terminano sempre.

1.5.3 Esempio di codifica

	0	1	2	3	4	5
0	0	2	5	9	14	
1	1	4	8	13		
2	3	7	12			
3	6	11	...			
4	10	16				
5	15					

Codifica a coda di rondine

Codifica $(x,\, y) \mapsto \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$
Es. $(3,\, 1) \mapsto \frac{1}{2}(9 + 6 + 1 + 9 + 1) = \frac{26}{2} = 13$

Decodifica $n \mapsto (n - \frac{1}{2}k(k+1),\; k - (n - \frac{1}{2}k(k+1)))$
con $k = \lfloor \frac{1}{2}(\sqrt{1 + 8n} - 1) \rfloor$

Es $8 \mapsto (8 - 6,\, 6 - 8 + 3) = (2,\, 1)$
$k = \lfloor \frac{1}{2}\sqrt{1 + 8 \cdot 8} - 1 \rfloor = 3$
$\frac{k(k+1)}{2} = 6$

1.6 Notazione

Una **funzione** f è $\subset$ A $\times$ B, con A spazio di partenza e B codominio. Quindi $f(\mathrm{a}) = \mathrm{b}$ si può esprimere anche con $(\mathrm{a},\, \mathrm{b}) \in f$, con a $\in$ A e b $\in$ B.

$f(\mathrm{a}) = \mathrm{b} \wedge f(\mathrm{a}) = \mathrm{c} \Rightarrow \mathrm{b} = \mathrm{c}$

Considereremo **funzioni parziali**, cioè funzioni con A contenente punti dove f non è definita. Non è quindi detto che $\forall\ \mathrm{a} \in \mathrm{A}\ \exists\ \mathrm{b} \in \mathrm{B} \mid f(\mathrm{a}) = \mathrm{b}$

f **converge** su a, cioè $f(\mathrm{a})\downarrow \Leftrightarrow \exists\ \mathrm{b} \mid\ f(\mathrm{a}) = \mathrm{b}$

f **diverge** su a, cioè $f(\mathrm{a})\uparrow \Leftrightarrow \nexists\ \mathrm{b} \mid\ f(\mathrm{a}) = \mathrm{b}$

Dominio di f: $dom(f) = \{\mathrm{a} \mid f(\mathrm{a})\downarrow\}$

Immagine di f: $imm(f) = \{\mathrm{b} \mid \exists\ \mathrm{a} \in \mathrm{A} \Rightarrow f(\mathrm{a}) = \mathrm{b}\}$

Rapporto tra algoritmi A e funzioni f f è un **insieme potenzialmente infinito di coppie**, ma non posso assegnare due f diverse allo stesso insieme, mentre esistono tanti algoritmi diversi che calcolano la stessa funzione.
Ad esempio, $f = \emptyset$ è calcolata da `while(true) do skip` ma anche da `while(true) do skip;skip`.

1. Quali sono le funzioni calcolabili?
   Nelle ipotesi iniziali di definizione di algoritmo, per adesso conosciamo le T-Calcolabili e le `while`-Calcolabili.

2. Quali proprietà hanno?
   Posso combinarle?

3. Esistono funzioni non calcolabili?

4. Sono interessanti?
   Esistono a prescindere dalla macchina?

Algoritmi e calcolabilità Per ora abbiamo definito gli algoritmi in base al loro comportamento, sotto forma di **configurazioni che si susseguono** del tipo (istr. corrente + ..., memoria). Abbiamo anche diversi modi di affrontare la calcolabilità:

1. **Hardware**, con la macchina di Turing
   Questo è uno dei primi esempi di calcolo, è semplice da capire e si descrivono direttamente macchine che eseguono gli algoritmi. Uno dei primi approcci allo studio della complessità.
   Cambio programma $\rightarrow$ Cambio macchina

2. **Software**
   Ho l'interprete, cioè la semantica, fissi. Se cambio il programma non devo cambiare la macchina

   (a) Programmi `while`
   Base della programmazione iterativa, dalla semantica operazionale e anch'essi usati per lo studio della complessità

   (b) Funzioni ricorsive
   Base della programmazione funzionale

1.7 Funzioni ricorsive primitive

Per formalizzare i vari modi con cui possiamo esprimere le funzioni, usiamo quella che si chiama **λ-notazione**. Queste espressioni individuano gli argomenti all'interno di un'espressione che descrive una funzione, scritta seguendo un'opportuna sintassi.

$$\lambda < \textit{variabili} > . < \textit{espressione} >$$

Esempio $\lambda\, x,\, y.expr$
Gli **argomenti** dell'espressione *expr* sono x, y. Si dice anche che x, y **appaiono legate da** λ **in *expr***.
Invece un qualsiasi altro simbolo di variabile w in *expr* **non è da considerarsi argomento** dell'espressione, e viene definito **libero** in *expr*.
Altri **esempi** per evidenziare la **notazione**:

$\lambda\, y.x + y$

$\lambda\, x \lambda\, y.x + y$ che può essere riscritta come $\lambda\, x,\, y.x + y$ ed equivale a dire $somma(x, y) = x + y$ dando così il nome *somma* alla funzione.

$\lambda\, x_1, x_2, \ldots, x_n.expr$ riscritta come $\lambda\, \overrightarrow{x}.expr$

1.7.1 Classe C

La classe C delle **funzioni ricorsive primitive** è la **minima classe** di funzioni che obbediscono alle seguenti regole di inferenza, regole di sintassi per definire le funzioni.
Casi base

Zero: $\lambda \overrightarrow{x}.0$
Prende un vettore di argomenti e restituisce 0.

Successore: $\lambda x.x + 1$
Prende un valore e restituisce il suo successore.

Proiezione/Identità: $\lambda \overrightarrow{x}.x_i$
$\overrightarrow{x} = x_1, \ldots, x_n,\ 1 \leq i \leq n$

Casi iterativi

Composizione
$g_1, \ldots, g_n \in C$ con k argomenti ("a k posti") e
$h \in C$ a n posti
$\Rightarrow \lambda x_1, \ldots, x_k.h(g_1(\overrightarrow{x}), \ldots, g_n(\overrightarrow{x})) \in C$

Ricorsione primitiva
$h \in C$ a $n + 1$ posti e
$g \in C$ a $n - 1$ posti

$$\Rightarrow \begin{cases} f(0, x_2, \ldots, x_n) & = g(x_2, \ldots, x_n) & \textit{Passo finale} \\ f(x_1 + 1, x_2, \ldots, x_n) & = h(x_1, f(x_1, x_2, \ldots, x_n), x_2, \ldots, x_n) & \textit{Iterazione} \end{cases}$$

Questo rappresenta un **ciclo for**: x_1, la prima variabile di f, è il contatore da decrementare, g è il passo finale e h rappresenta i passi interni del ciclo. La terminazione è garantita, e si possono definire le primitive matematiche base con le funzioni ricorsive primitive.
$f \in C \Leftrightarrow$ esiste una successione $f_0, \ldots, f_n = f \mid \forall f_1$ è ottenuto con i casi base oppure f_i è ottenuto con i casi iterativi da f_j con $j < i$

Esempio Esempio di funzioni ricorsive
$f_1 = \lambda x.x$
$f_2 = \lambda x.x + 1$
$f_3 = \lambda x_1, x_2, x_3.x_2$
$f_4 = f_2(f_3(x_1, x_2, x_3))$
$\begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) \end{cases}$

Proviamo a calcolare $f_5(2, 3) =$

Regola di valutazione interna-sinistra: valuto per primo quello che sta dentro i parametri partendo da sinistra.
$f_5(2,3) =$
$= f_5(1+1,3) =$
$= f_4(1, f_5(1,3), 3) =$
$= f_4(1, \overline{f_4(0, f_5(0,3), 3)}, 3) =$
$= f_4(1, f_4(0, \overline{f_1(3)}, 3), 3) =$
$= f_4(1, \overline{f_4(0,3,3)}, 3) =$
$= f_4(1, \overline{f_2(f_3(0,3,3))}, 3) =$
$= f_4(1, \overline{f_2(3)}, 3) =$
$= \overline{f_4(1,4,3)} =$
$= \overline{f_2(f_3(1,4,3))} =$
$= \overline{f_2(4)} =$
$= \overline{5}$

Vediamo cosa succede con una **regola di valutazione esterna**:
$f_5(2,3) =$
$= \overline{f_4(1, f_5(1,3), 3)} =$
$= \overline{f_2(f_3(1, f_5(1,3), 3))} =$
$= \overline{f_3(1, f_5(1,3), 3)} + 1 =$
$= \overline{f_5(1,3)} + 1 =$
$= \overline{f_4(0, f_5(0,3), 3)} + 1 =$
$= \overline{f_2(f_3(0, f_5(0,3), 3)} + 1 =$
$= \overline{f_3(0, f_5(0,3), 3)} + 1 + 1 =$
$= \overline{f_5(0,3)} + 2 =$
$= \overline{f_1(3)} + 2 =$
$= 3 + 2 =$
$= 5$

Meno Limitato Non ritorna mai numeri negativi, ma 0.
$f_7(x, y) = y$

$f_8(x, y) = x$

$\begin{cases} pred(0) = 0 \\ pred(x+1) = f_8(x, pred(x)) \end{cases}$

$f_9(x, y, z) = pred(f_3(x, y, z))$

$\begin{cases} f_{10}(0, y) = f_1(y) \\ f_{10}(x+1, y) = f_9(x, f_{10}(x, y), y) \end{cases}$

$\Rightarrow x \dot{-} y = f_{10}(f_7(x, y), f_8(x, y))$

Somma Non è altro che generalizzazione del successore, applico il successore tante volte quante servono.

$\begin{cases} 0 + y = y \\ (x+1) + y = (x + y) + 1 \end{cases}$

Prodotto Sfrutto la somma

$\begin{cases} 0 * y = 0 \\ (x+1) * y = (x * y) + y \end{cases}$

Potenza Generalizza il prodotto

$\begin{cases} x^0 = 1 \\ x^{y+1} = (x^y) * x \end{cases}$

C'è un modo per generalizzare la potenza? $\Rightarrow$ **Ackerman**.

Relazione Diciamo che la relazione $R(x_1, \ldots, x_n) \subset N^n$ è **ricorsiva primitiva** se lo è la sua **funzione caratteristica** χ_R definita come
$\chi_R(x_1, \ldots, x_n) = \begin{cases} 1 & se\ (x_1, \ldots, x_n) \in R \\ 0 & se\ (x_1, \ldots, x_n) \notin R \end{cases}$

Quindi se χ_R è ricorsiva primitiva allora anche R è ricorsiva primitiva. Come esempio, vediamo l'uguaglianza:

$$\chi_=(0,0) = 1 \quad \chi_=(x+1, y+1) = \chi_=(x, y) \quad \chi_=(0, y+1) = 0 \quad \chi_=(x+1, 0) = 0$$

Esempio $P = \{\ n \in N \mid n$ è un numero primo$\}$ è ricorsiva primitiva. Questo per il teorema di fattorizzazione unica.
$\forall x \in N\ \exists$ numero finito di esponenti $x_1 \neq 0 \mid x = p_0^{x_0} \cdot p_1^{x_1} \cdot \ldots \cdot p_n^{x_n}$
Come trovare tali esponenti con f ricorsiva primitiva.
M = (Q, Σ, δ, q_0)
Q = {q_0, …, q_k), Σ = {σ_0, …, σ_n)

1.7.2 Numerazione di Godel

Kurt Gödel, rappresentare algoritmi come numeri: **Gödelizzazione**, cioè data una MdT M trovo i che è il suo **numero di Gödel**.
La funzione caratteristica dell'insieme dei numeri primi è ricorsiva primitiva. Gli esponenti della fattorizzazione in numeri primi di qualsiasi numero naturale sono dati da funzioni ricorsive primitive.
Ne deriva che ogni sequenza di numeri naturali può essere codificata come numero naturale, usando i numeri come esponenti della fattorizzazione in numeri primi. **Questa codifica è iniettiva ma non suriettiva**, e una codifica effettiva deve essere anche suriettiva. Gödel fornì una migliore codifica che era anche suriettiva, da cui deriva che le funzioni di codifica e decodifica sono ricorsive primitive. Tutte le MdT, programmi `for`/`while`, funzioni ricorsive primitive e μ-ricorsive possono essere effettivamente codificate e decodificate come numeri naturali, usando i simboli sintattici come numeri naturali distinti. Le computazioni (intese come successioni di configurazioni) possono anch'esse essere codificate e decodificate.

Data M = (Q, $\sum$, δ, q_0) MdT con Q = $\{q_0, \ldots, q_n\}$ e $\sum = \{\sigma_0, \ldots, \sigma_m\}$, ogni quintupla $(q_i, \sigma_j, q_k, \sigma_l, D) \in \delta$ è codificata come $p_0^{i+1} \cdot p_1^{j+1} \cdot p_2^{k+1} \cdot p_3^{l+1} \cdot p_4^{m_D}$ con $p_0 < \ldots < p_4$ sono numeri primi. Se consideriamo i simboli di direzione come simboli veri e propri, allora L = σ_{m+2}, R = σ_{m+3}, — = σ_{m+4} e poniamo $m_D \in \{m_L = m+2, m_R = m+3, m_- = m+4\}$

Grazie al teorema della fattorizzazione unica, ad ogni quintupla è associato un solo intero e viceversa, che rende la funzione iniettiva.
Possiamo ordinare l'insieme delle quintuple che caratterizza M, ottenendo una successione di quintuple codificate con un numero: abbiamo una successione $a_0 a_1 \ldots a_n$ di numeri naturali tali che $a_i \neq a_j$ se $i \neq j$. Codificando questa successione otteniamo un altro numero, i detto **indice associato alla MdT M**.

La funzione non è suriettiva, ma il problema può essere aggirato: decodificando i, se ottengo una MdT bene, altrimenti i non è un indice e lo butto, oppure restituisco un valore speciale.

Tutte le funzioni definibili mediante gli schemi di ricorsione primitiva sono totali, e data una funzione ricorsiva primitiva si può scrivere un programma `FOR` che dati gli stessi argomenti produce lo stesso risultato, e viceversa.
Abbiamo trovato il formalismo definitivo, che esprime tutte le funzioni calcolabili? **No**, perché esiste la funzione di Ackermann: totale, con una definizione accettabile ma che non è definibile mediante gli schemi di ricorsione primitiva.

1.7.3 Funzione di Ackermann

La funzione di Ackermann **non è definibile** mediante gli schemi di ricorsione primitiva definiti in precedenze, ma è totale ed ha una definizione intuitivamente accettabilissima.

$$A(0, 0, y) = y$$

$$A(0, x+1, y) = A(0, x, y) + 1$$

$$A(1, 0, y) = 0$$

$$A(z+2, 0, y) = 1$$

$$A(z+1, x+1, y) = A(z, A(z+1, x, y), y)$$ **doppia ricorsione**

La **doppia ricorsione** presente non è un problema: tutti i valori su cui si ricorre decrescono, quindi i valori di $A(z, x, y)$ sono definiti in termini di un numero finito di valori della funzione A. Quindi intuitivamente A è calcolabile.
Inoltre **cresce più rapidamente di ogni funzione ricorsiva primitiva** ma **non è ricorsiva primitiva**.
Ma cosa calcola? Una sorta di esponenziale generalizzato, infatti:

$$A(0, x, y) = y + x$$

$$A(1, x, y) = y * x$$

$$A(2, x, y) = y^x$$

$$A(3, x, y) = y^{y^{\cdots^{y}}\}\ x \text{ volte}}$$

1.7.4 Realizzazione

Con il linguaggio-`while` e il linguaggio-`for` posso riprodurre i casi base della ricorsione. In particolare, per ogni programma `for` esiste una funzione ricorsiva primitiva e viceversa.

Un programma che calcola lo 0 è un programma che legge gli ingressi e scrive 0 in uscita.

Il successore lo realizzo con un assegnamento uscita = ingresso +1

La proiezione consiste nel leggere in memoria la variabile x_i cercata e metterla in uscita

Realizzo h tale che $h(g_1(x,y,z), g_2(x,y,z))$, con programma p_1 associato a g_1, p_2 associato a g_2 e p_3 associato a h.
Il programma che realizza la composizione sarà quindi $p_1;p_2;p_3$.

Per la ricorsione primitiva $\begin{cases} f(0,y) = g(y) \to p_1 \\ f(x+1,y) = h(x, f(x,y), y) \to p_2 \end{cases}$ che dopo qualche passaggio abbiamo visto che $f(x+1,y) = h(x, f(x,y), y) = h(x, h(x+1, f(x+1,y)), y)$. Associando p_1 a g e p_2 a h, lo realizzo con il programma-`for`

```
t1 = g(y);
for (i = 1 to x + 1):
        t1 = g(i, t1, y);
end
```

Per la **funzione caratteristica** $\chi_I(n) = \begin{cases} 1 & n \in I \\ 0 & altrimenti \end{cases}$

1.8 Diagonalizzazione

Esiste un formalismo che esprime tutte e sole le funzioni totali calcolabili? No
Dobbiamo necessariamente avere a che fare con funzioni parziali, ma perché "no"?
Qualunque formalismo o esprime solo funzioni totali ma non tutte, oppure esprime anche funzioni parziali. La dimostrazione è fondamentale per la teoria della calcolabilità: prende il nome di **diagonalizzazione**.

Dimostazione Fissato il formalismo delle funzioni ricorsive primitive, posso prendere l'algoritmo di Gödel per numerarle: avrò le funzioni f_0, f_1, ..., f_n, ... così numerate.
Definisco $g(n) = f_n(n) + 1$ (*diagonalizzazione* viene da usare lo stesso indice per indice e parametro): effettivamente calcolabile perché basta prendere l'algoritmo n-esimo, applicarlo a n e sommare 1 al risultato. Inoltre, g è totale.
Se g è una ricorsiva primitiva, allora è numerabile: diciamo che g ha come numero i, cioè $f_i(n) = g(n) = f_n(n) + 1$
Se diagonalizzo avrò $f_i(i) = g(i) = f_i(i) + 1$ ma **non può succedere** che $f_i(i) = f_i(i) + 1$
$\Rightarrow g$ non è ricorsiva primitiva.

Se io prendo le funzioni parziali, posso applicare lo stesso ragionamento?
$\phi(x) = \psi_x(x) + 1$
Diciamo come prima che $\phi(x)$ ha indice i, quindi $\psi_i(x) = \phi(x) = \psi_x(x) + 1$
Se $\psi_i(x)$ diverge, allora $\psi_x(x)$ diverge e anche $\psi_x(x) + 1$ diverge, quindi sono uguali. Non si applica il ragionamento della diagonalizzazione nel caso delle funzioni parziali.
Quindi il discorso **si applica ad ogni formalismo che definisca solo funzioni totali**.

1.9 μ-ricorsive

Minima classe $\mathscr{R}$ che, allo schema fino alla ricorsione primitiva, aggiunge:

Minimizzazione $\phi(\overrightarrow{x}, y) \in \mathscr{R}$
$\psi(\overrightarrow{x}) = \mu y.[\phi(\overrightarrow{x}, y) = 0 \wedge \forall z \leq y \mid \phi(\overrightarrow{x}, z) \downarrow]$

$\mu x.[I]$ è il minimo elemento di I insieme.

Cosa significa? Data una funzione $\phi \in \mathscr{R}$ (che ovviamente può essere una ricorsiva primitiva), la vado a calcolare sugli argomenti $\overrightarrow{x}$ della ψ e su una certa y. Se vale 0, il risultato è y, altrimenti **deve** convergere e vado avanti incrementando y di 1 e ricalcolando fino a che non trovo un risultato pari a 0.
Quindi le μ-ricorsive definiscono anche funzioni non totali, al contrario delle ricorsive primitive che definiscono solo funzioni totali.
La μ corrisponde ad un **ciclo while**: inizia con $y = 0$ e lo incrementa fino a che ϕ non è uguale a 0 (guardia del loop), ritornando y. Se viene piazzato un limite superiore a y, allora ψ potrebbe essere ricorsiva primitiva e se lo è anche la ϕ allora la condizione di convergenza è garantita.
La diagonalizzazione non può essere applicata direttamente alle funzioni μ-ricorsive, perché $\phi_n(n)$ può divergere. Non tutte le funzioni μ-ricorsive possono essere estese a funzioni totali: **una funzione è μ-calcolabile se può essere ridefinita come μ-ricorsiva**.

Esempio $\phi(x, y) = 42$ è costante, quindi ricorsiva primitiva, quindi anche μ-ricorsiva.
$\psi(x) = \mu y.[\phi(x, y) = 42]$ ovunque indefinita perché non tornerà mai 0 quindi $\nexists\, y$.

Quindi **terminazione e non terminazione sono cruciali**.
Se la definisco per casi? Ad esempio $f(x) = \begin{cases} \mu y.[y < g(x) \mid h(x, y) = 0] & \text{se} \exists \text{ tale } y \\ 0 & \text{altrimenti} \end{cases}$ con g, h ricorsive primitive.
f è ricorsiva primitiva, perché composizione di ricorsive primitive, ed è anche totale, perché converge sempre.
Quindi **se pongo dei limiti al numero di tentativi**, dato da $y < g(x)$, si ricade nelle ricorsive primitive e non ci sono problemi di parzialità.

1.9.1 Notazione

Per ragioni storiche, una relazione $I \subset N^n$ è **ricorsiva** (sinonimo di totale) $\Leftrightarrow$ la sua funzione caratteristica χ_I è **calcolabile totale**.
Inoltre, come già detto, I è ricorsiva primitiva $\Leftrightarrow$ χ_I è ricorsiva primitiva.

1.10 Tesi di Church-Turing

Le funzioni intuitivamente calcolabili sono tutte e sole le T-calcolabili.
In realtà è un'ipotesi, ma è così forte che viene presa come tesi. Ci permette di non considerare il formalismo con cui formalizziamo gli algoritmi, poiché tutti i formalismi rappresentano la stessa *classe di elementi*. Ci limiteremo a dire algoritmo, Macchina di Turing... indifferentemente, poiché grazie a questa tesi possiamo dire che un algoritmo è equivalente qualsiasi sia il linguaggio in cui è scritto.
Questa tesi postula che la nozione di calcolabilità *intuitiva* è robusta, ma cade se si rilascia anche una sola delle ipotesi fatte sulla natura degli algoritmi.

Da qui in avanti parleremo di **funzioni calcolabili** senza specificare il formalismo con cui le definiamo.
Quante sono? Ce ne sono di non calcolabili? Se ce ne sono, sono interessanti?

1.10.1 Risultati

Indichiamo con ϕ_i è la **funzione calcolata dall'i-esimo algoritmo** M_i

ϕ_i è funzione $\rightarrow$ **semantica**

M_i è algoritmo $\rightarrow$ **sintassi**

Può succedere per $i \neq j$ che $\phi_i = \phi_j$ ma $\mathrm{M}_i \neq \mathrm{M}_j$ (ad esempio `while(true) do skip` e `while(true) do skip;skip`).

$$\text{T-calcolabili} = \texttt{while}\text{-calcolabili} = \mu\text{-calcolabili}$$

D'ora in avanti parliamo solo di funzioni calcolabili, quindi ϕ_i *è* calcolabile.

Tempo di calcolo $\exists$? una funzione calcolabile totale $t(i, n)$ che maggiora il tempo di calcolo di M_i(n)? No. Vediamo come dimostrarlo, introducendo una **diagonalizzazione**.
$t(i, n) = \begin{cases} k & \text{se } M_i(n) \downarrow \text{ in meno di } i \text{ passi} \\ 0 & \text{altrimenti} \end{cases}$
Sia T_i la misura **esatta** del tempo di calcolo di M_i. T_i(n) $\leq t(i, n)$ è calcolabile totale e $T_x(x)$ **tempo di calcolo**

effettivo, $t(x,x)$ **tempo di calcolo stimato**

$$\psi(x) = \begin{cases} \phi_x(x)+1 & \text{se } T_x(x) \leq t(x,x) \\ 0 & \text{altrimenti} \end{cases}$$

Quindi anche ψ è calcolabile totale. Applico Church-Turing, quindi $\phi_i(i) = \psi(i) = \begin{cases} \phi_i(i)+1 & \text{se } T_i(i) \leq t(i,i) \\ 0 & \text{altrimenti} \end{cases}$

Ma siccome ϕ_i è calcolabile totale, non può succedere che, quando termina, sia $\phi_i(i) = \phi_i(i)+1$, **è un assurdo**. Quindi $t(i,n)$ non è calcolabile totale, di conseguenza **non c'è modo di stimare il tempo di calcolo**.
Per lo stesso motivo non possiamo imporre limiti allo spazio.

Spazio di calcolo $\exists$? una funzione calcolabile totale che dato M_i, x dice quante celle di memoria uno specifico calcolatore C userà per calcolare $\mathrm{M}_i(x)$?

$$h(i,x) = \begin{cases} 1 & \text{se } M_i(x) \uparrow \text{(su } C\text{)} \\ 0 & \text{altrimenti} \end{cases}$$

Sia n la cardinalità di Σ, $m-1$ cardinalità di Q e k il numero di celle di C.
Posso quindi scrivere n^k **stringhe diverse**: il cursore può stare su k posizioni diverse e la macchina in m stati diversi.
Il **numero massimo di configurazioni diverse** (stato con posizione del cursore e stringa su nastro) è $l = n^k \cdot k \cdot m$, con n^k possibilità di nastro scritto, k possibili posizioni del cursore e m stati ($m-1+1$ per lo stato h di **halt**)
Dopo l passi, quindi, la configurazione si ripete necessariamente. Si può dire che **la macchina è in loop**.
Siccome la macchina attraversa un **numero finito di configurazioni superiormente limitato da** l, se la macchina non si è arrestata prima di l passi allora troverò per forza una configurazione già vista in precedenza, che mi porterà in una configurazione già vista e così via. **Quindi non terminerà mai**.
Ho dimostrato che h è calcolabile totale, ma posso scrivere quindi t come nella dimostrazione precedente, ma giungo all'assurdo già visto. **Quindi non posso mettere un limite al nastro.**

I seguenti risultati sono **invarianti rispetto all'enumerazione scelta**.

1.10.2 Teorema 1: Le Funzioni Calcolabili sono tante quante i numeri naturali

Le f calcolabili sono $\#N$. Anche le f calcolabili totali sono $\#N$.
Esistono funzioni *non* calcolabili, molte di più di quelle calcolabili.

Dimostrazione Sono almeno $\#N$ perché posso costruire una macchina che per qualsiasi input lascia un numero naturale sul nastro, quindi di queste ce ne sono almeno quanti sono i numeri naturali. Non sono più di $\#N$ perché le MdT si possono enumerare (es. Gödel).

Quindi indichiamo con ϕ_i la funzione (in generale, parziale) calcolata dall'algoritmo M_i e i **indice della macchina**. Come detto prima, può darsi che per $i \neq j$ sia $\phi_i = \phi_j$ ma sicuramente $\mathrm{M}_i \neq \mathrm{M}_j$.

1.10.3 Teorema 2: Ogni funzione calcolabile ϕ_i ha infiniti (numerabili) indici

Anche detto **padding lemma**.
Non solo, posso costruire un insieme infinito di indici A_i tale che $\forall\, j \in \mathrm{A}_i \Rightarrow \phi_j = \phi_i$ mediante una funzione ricorsiva primitiva.

Dimostrazione Sia M_i un programma `P`. Prendo `P;skip`, poi `P;skip;skip`... metto tanti `;skip` quanti voglio. Posso generare un numero infinito di programmi che calcolano tutti la stessa funzione.

1.10.4 Teorema 3: Forma Normale

Prendendo un qualsiasi algoritmo, posso riscriverlo in una **forma canonica/normale**, che non è per forza migliore ma è una forma specifica e da noi **privilegiata**.
$\exists$ un predicato $T(i,x,y)$ e una funzione $U(y)$ ricorsive primitive calcolabili totali tali che $\forall$ funzione calcolabile i, $x.\phi_i(x) = \mu y.[U(T(i,x,y))]$

Dimostrazione Devo costruire il predicato T e la funzione U.
$T(i,x,y)$ è detto **predicato di Kleene** ed è **vero** $\Leftrightarrow$ y **è la codifica di una computazione di** $\mathrm{M}_i(x)$ **terminante**.
Per calcolare, T prende i e recupera M_i. Comincia a scandire i valori y, li decodifica e, uno alla volta dato x ingresso, controlla se il risultato è una computazione terminante. Definisce U in modo che $U(y) = z$, con z risultato della computazione (ciò che rimane sul nastro della MdT corrispondente, cioè la macchina terminante $\mathrm{M}_i(x) = c_0 c_1 \dots c_n$

con $c_n = (h, \triangleright z \underline{\#}))$.
C'è un solo y nelle macchine deterministiche, se c'è, quindi T e U sono calcolabili per la tesi di Church-Turing. Inoltre il procedimento termina sempre, quindi T e U sono totali.
Se considerassimo le MdT non deterministiche, potrebbero esserci più computazioni che terminano in h, per cui μ (l'operatore di minimizzazione) darebbe come risultato il minimo intero che ne codifica una.

Un immediato **corollario** del teorema di forma normale è che tutte le funzioni T-calcolabili sono anche μ-calcolabili. Non solo, ma μy corrisponde al `while`, e T, U a due programmi `for`. Quindi ogni funzione calcolabile può essere ottenuta da due programmi scritti con il linguaggio `for` ed una sola applicazione del linguaggio `while`.

1.10.5 Teorema 4: Teorema di enumerazione

$\exists\, z \mid \forall\, i,\, x \;\; \phi_z(i,x) = \phi_i(x)$. Quindi z è **MdT universale**. z interprete, i è programma.
Fra tutti gli algoritmi, ce ne sono infiniti numerabili in grado di eseguire tutti gli altri algoritmi.

Dimostrazione $\phi_i(x) = \mu y.U(T(i,x,y))$ per il terzo teorema. Quindi $\phi_i(x)$ è un algoritmo, avrà un indice per Church-Turing che indicheremo con z.
Diciamo $\phi_i(x) = \phi_z(i,x) = \mu y.U(T(i,x,y))$ (senza y come argomento perché quantificata, non libera ma legata, una sorta di *variabile di lavoro*). Applico la transitività dell'uguaglianza e ho finito, $\phi_i(x) = \phi_z(i,x)$.
Più intuitivamente: M_z recupera la descrizione di M_i e la applica a x.

1.11 Macchina di Turing Universale

Ottimo modello per le macchine Von Neumann.

Due insiemi Utili per la rappresentazione
$Q_* = \{q_0, q_1, \ldots\} \not\ni h$, insieme numerabile: **non è l'insieme degli stati** della MdTU ma è un insieme ausiliario.
$\Sigma_* = \{\sigma_0, \sigma_1, \ldots\} \not\ni L, R, -$ insieme di simboli.
Ogni MdT M_k ha l'insieme degli stati Q_k e quello dei simboli Σ_k inclusi rispettivamente in Q_* e Σ_*

Codifica K $K : Q_* \cup \{h\} \cup \Sigma_* \cup \{L, R, -\} \to \{\,|\,\}^*$
Ognuno degli elementi q_i, σ_i ecc. viene codificato in questo modo.
$h \mapsto |$
$q_i \mapsto |^{i+2}$, con la barra verticale ripetuta $i+2$ volte
$L \mapsto |$, $R \mapsto ||$, $- \mapsto |||$
$\sigma_i \mapsto |^{i+4}$
C'è un problema: $|||||$ è uno stato ($\in Q_*$) o un simbolo ($\in \Sigma_*$)? Vedremo come disambiguare.
K non è biunivoca, ma lo è quando sappiamo se la stringa da decodificare è uno stato, un simbolo o una direzione: separereremo la codifica degli stati da quella dei simboli e delle direzioni da una marca c **non appartenente all'alfabeto della macchina da decodificare**.
Q_* contiene **tutti i possibili stati delle MdT**, e Σ_* contiene **tutti i possibili simboli delle MdT**. Quanto visto fin'ora è ausiliario alla costruzione della MdTU.

Costruzione Prendo una MdT qualunque $M = (Q, \Sigma, \delta, s)$. Ordiniamo gli stati ed i simboli.
$Q = \{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$, finito perché gli stati sono finiti.
$\Sigma = \{\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_l}\}$.
Con $i_1 < i_2 < \ldots < i_k$ e $j_1 < j_2 < \ldots < j_l$. Con questo abbiamo supposto Q e Σ **totalmente ordinate**.
Considero questo alfabeto: $\{\,|, c, d, \#, \triangleright\}$ con $|, c, d \notin Q_* \cup \Sigma_*$.
Adesso possiamo codificare le quintuple $\in \delta$. $\delta(q_{i_p}, \sigma_{j_q}) = (q, \sigma, D)$ con D un certo simbolo di direzione $(L, R, -)$.
Come codificarla? Attraverso la seguente stringa

$$s_{p,q} = cK(q_{i_p})cK(\sigma_{j_q})cK(q)cK(\sigma)cK(D)c$$

con c che funge da **separatore**, **tra la codifica di uno stato e la codifica di un simbolo**.
$s_{p,q} = cK(q_{i_p})cK(\sigma_{j_q})cdcdcd$ se $\delta(q_{i_p}, \sigma_{i_q})$ è indefinita

Esempio $\overline{M} = \{\{q_2\}, \{\sigma_1, \sigma_3, \sigma_5\}, \delta, q_2\}$, con $i_1 = 2$, $j_1 = 1$, $j_2 = 3$, $j_3 = 5$

$q_2 \to q_{i_1}$

$\sigma_1 \to \sigma_{j_1}$

$\sigma_3 \to \sigma_{j_2}$

$\sigma_5 \to \sigma_{j_3}$

$\delta(q_2, \sigma_1) = (h, \sigma_5, -) \mapsto s_{1,1} = c|^4c|^5c|c|^9c|^3c$ perché $K(q_2) = |^{2+2}$, $K(\sigma_1) = |^{1+4}$, $K(h) = |$, $K(\sigma_5) = |^{5+4}$, $K(-) = |||$
$\delta(q_2, \sigma_3) = (q_2, \sigma_1, R) \mapsto s_{1,2} = c|^4c|^7c|^4c|^5c|^2c$
$\delta(q_2, \sigma_5)$ non definita $\mapsto s_{1,3} = c|^4c|^9cdcdcdc$

Codifica della macchina ρ Possiamo ora definire una funzione di "*codifica*" (in realtà iniettiva, perché in fase di decodifica possiamo non trovare una M corrispondente) della macchina in esame. (s senza indici è lo stato iniziale)
$\rho(M) = cK(s)cs_{1,1}s_{1_2} \dots s_{1,l}c \dots cs_{k,1}s_{k_2} \dots s_{k,l}c$
Praticamente si mettono "in coda" tutte le codifiche dei δ(stato,simbolo) $\to s_{stato,simbolo}$, separando con c il "cambio" dello stato.
$\rho(M) = c$ <codifica di $s > c$ <codifiche dello stato 1 per simbolo> $c \dots c$ <codifiche dello stato k per simbolo> c

$\rho(M(w)) = \rho(M)\tau(w)$ con $\tau(\sigma'_0, \dots, \sigma'_n) = cK(\sigma'_0)cK(\sigma'_1)c \dots cK(\sigma'_n)c$

Esempio $\rho(\overline{M}) = c|^4c\ c|^4c|^5c|c|^9c|^3c\ c|^4c|^7c|^4c|^5c|^2c\ c|^4c|^9cdcdcdc\ c$

Dato che $\exists\ z \mid \forall\ i, x$ si ha $\phi_z(i, x) = \phi_i(x)$, allora vogliamo che la MdTU si comporta esattamente come $\overline{M}$ quando codifica $\overline{M}$. Quindi

$(s, \underline{\triangleright} w) \longrightarrow_n^M (h, u\underline{a}v) \Rightarrow (s_U, \underline{\triangleright}\rho(M)\tau(w)) \longrightarrow_m^U (h, \tau(uav)\underline{\#})$

$(s_U, \underline{\triangleright}\rho(M)\tau(w)) \longrightarrow_n^U (h, u'a'v')$ dovrebbe succedere che $a' = \#$ e $v' = \epsilon$, ma anche che $u' = \tau(uav)$. Se succede $\Rightarrow (s, \underline{\triangleright} w) \longrightarrow_m^M (h, uav)$

1.12 Teoremi

1.12.1 Teorema del parametro (s–m–n)

$\exists\ s$ calcolabile, totale, iniettiva $\mid \forall\ i,\ x\ \lambda y.\phi_i(x, y) = \phi_{s(i,x)}(y)$
Quindi considerato x come parametro, la funzione può essere calcolata da un'altra macchina di indice dipendente da i e x. Ottimo strumento per dimostrare diversi risultati.
Intuitivamente, il programma $P_{s(i,x)}$ opera su y soltanto mentre P_i opera su x e su y. Quindi x è un **parametro** di P_i. Da $P_i(x, y) \to j = s(i, x) \mid P_j(y) = P_i(x, y)$

Al caso generale, $\forall\ m, n > 0 \quad \exists\ s_n^m$ calcolabile, totale e iniettiva con $m + 1$ argomenti $\mid$
$\forall\ i, x_1, \dots, x_m \quad \lambda(y_1, \dots, y_n).\phi_i^{(m+n)}(x_1, \dots, x_m, y_1, \dots, y_n) = \phi^{(n)}_{s_n^m(i,x_1,\dots,x_m)}(y_1, \dots, y_m)$

Dimostrazione Dato i trova M_i, attraverso una funzione ricorsiva primitiva grazie al teorema di codifica.
Scrivi x sul nastro di M_i cioè prepara la configurazione iniziale $(q_0, \underline{\triangleright} x)$.
Questo è un algoritmo, quindi ha indice per C-T. Diciamo che l'indice è $n = s(i, x)$. s è calcolabile totale, e $\lambda y.\phi_i(x, y) = \phi_{s(i,x)}(y)$ è vera.
Per l'iniettività? Posso costruire s' con $\phi_{s(i,x)} = \phi_{s'(i,x)}$ in modo che $s'(i, x)$ generi indici in maniera strettamente crescente, cioè tali che $s'(i_0, x_0) > s'(i_1, x_1)$ se la codifica di (i_0, x_0) è maggiore di quella di (i_1, x_1).

1.12.2 Teorema di Espressività

Un formalismo F è Turing-Equivalente, cioè **calcola tutte e sole le funzioni T-calcolabili** $\Leftrightarrow$

- Ha un algoritmo universale (Teorema di Enumerazione)
- Vale il Teorema del Parametro

Supponiamo un programma che calcola prodotto con somme successive

```
P := 0;
while y > 0
P := P + w;
y := y - 1;
end
```

Abbiamo sicuramente bisogno della semantica del linguaggio, perché definisce perfettamente l'interprete.
Primo passo dell'interprete: scopre `;` e legge ciò che sta prima e dopo.
`P` non è parametro, quindi scrive sul nastro d'uscita `P := 0;`. Valuta il `while`. `y` è parametro, lo legge e verifica che è maggiore di 0, quindi legge

```
P := P + w;
y := y - 1;
while ...
```

Per valutare dove va il while deve valutare il resto, per valutare il resto (le prime due istr) deve valutare la prima. Scrive in uscita `P := P + w;` e rimane

```
y = y - 1;
while ...
```

Quindi $\sigma(y) = 2$ adesso vale $\sigma'(y) = 1$ quindi diventa

```
while y > 0
P := P + w;
y := y - 1;
```

che valutata la guardia diventa

```
P := P + w;
y := y - 1;
while ...
```

Scrive sul nastro `P := P + w;`...
Sul nastro c'è `P := 0; P := P + w; P := P + w;`, che è il programma specializzato.

1.12.3 Teorema di Ricorsione/Kleene 2

Operazioni che si possono fare sugli algoritmi che ci lasciano all'interno delle funzioni calcolabili
$\forall\ f$ calcolabile totale $\exists\ n \mid \phi_n = \phi_{f(n)}$ (n si dice **punto fisso**)
Posso trasformare il mio programma in maniera da ottenerne uno perfettamente equivalente, come fanno i compilatori: f trasforma gli indici, cioè dato n (quindi P_n programma) lo trasforma in $f(n)$ (cioè $P_{f(n)}$ programma). Considerando il punto fisso, la trasformazione di f non cambia la funzione calcolata, cioè P_n e $P_{f(n)}$ sono equivalenti con la stessa semantica (qua l'accezione di punto fisso è leggermente diversa, intesa come "punto fisso" rispetto alla semantica).
Questo teorema fornisce la "base" della **semantica denotazionale**.

Dimostrazione Definiamo la seguente funzione calcolabile "diagonale":

$$\psi(u,z) = \phi_{d(u)}(z) = \begin{cases} \phi_{\phi_u(u)}(z) & \text{se } \phi_u(u)\downarrow \\ \textit{indefinita} & \textit{altrimenti} \end{cases} \quad (1)$$

ψ è calcolabile, quindi per C-T ha un indice $\psi(u,z) = \phi_{s(i,u)}(z) = \phi_i(u,z)$ e posso applicare s–m–n ottenendo $\psi(u,z) = \phi_{d(u)}(z)$ (con $d(u) = \lambda u.s(i,u)$ calcolabile totale e iniettiva e indipendente da f).
Ma f è calcolabile totale, quindi anche $f(d(u))$ è calcolabile totale. Quindi ha indice per C-T

$$f(d(u)) = \phi_v(u) \quad (2)$$

Quindi $\phi_{d(v)} = \phi_{\phi_v(v)}$ e poniamo

$$d(v) = n \quad (3)$$

e d è iniettiva quindi

$$\phi_n =_3 \phi_{d(v)} =_1 \phi_{\phi_v(v)} =_2 \phi_{f(d(v))} =_3 \phi_{f(n)}$$

Quindi n è punto fisso.

Dom	f	**Imm**
N	$\lambda x.2x$	$\{2n \mid n \in N\}$
$\{2n \mid n \in N\}$	$\lambda x.\frac{x}{2}$	N

I è ricorsiva primitiva $\Leftrightarrow \chi_i(x) = 1$ se $x \in I$, 0 altriementi è calcolabile totale
A $= \{(i, x, k) \mid \exists\, y, n$ con $(y, n, x < k) \wedge \phi_i(x) \downarrow$ in n passi$\}$. Limita spazio e tempo
B $= \{(i, x, k, z) \mid \exists\, n < k \wedge \phi_i(x) = z$ in n passi$\}$. Anche qua limito il numero di passi, ma non la memoria
$f(i, x)$ rimpiazza k, con f calcolabile totale

1.12.4 Ricorsivamente Enumerabile

I è **ricorsivamente enumerabile** $\Leftrightarrow$ è **dominio di una funzione calcolabile**, cioè $\exists\, i \mid I = dom(\phi_i)$

I è ricorsivo $\Rightarrow$ I è ricorsivamente enumerabile
Ricordiamo: I è ricorsivo $\Leftrightarrow$ la sua funzione caratteristica è calcolabile totale.

I, $\overline{I}$ sono ricorsivamente enumerabili, con $\overline{I}$ insieme degli elementi $\notin I$, $\Leftrightarrow$ I, $\overline{I}$ sono ricorsivi

Dim: se I è ricorsivamente enumerabile, allora esiste ϕ_i con domino I, e $\overline{I}$ ricorsivamente enumerabile allora esiste $\phi_{\overline{i}}$.
Per sapere se x sta o non sta in I, faccio un passo di calcolo nella macchina i: se termina allora $x \in I$, altrimenti faccio un passo nella macchina $\overline{i}$ e se termina $x \in \overline{I} \Rightarrow x \notin I$.
Se non terminano, ripeto: faccio un passo in i... finché una delle due macchine non termina.

Ricorsivamente enumerabile perché vogliamo enumerare i suoi elementi, tirandone fuori uno alla volta.

Teorema di equivalenza fra caratterizzazioni I è ricorsivamente enumerabile $\Leftrightarrow I = \emptyset \vee I = imm(f)$, con f calcolabile totale.

Dim Il primo caso è banale (la funzione caratteristica dà sempre 0, ricorsiva)
$I \neq \emptyset$, $I = dom(\phi_i)$ costruisco la f
Indice di riga tiene traccia del numero di passi
Indice di colonna tiene traccia dell'argomento

	0	1	2	3	4	5
1	0	2	5	9	14	
2	1	4	8	13	18	
3	3	7	12	17		
4	6	11	16			
5	10	15				

Passi:

1. Calcola ϕ_i con la tabella sopra
   1 passo su 0, 2 passi su 0... finché non si arresta.
   Trovo (m, n) per cui $\phi_i(< m, n >) \downarrow$ con $< m, n >$ codifica di (m, n). Chiamo $< m, n >= \overline{n} \in I$ perché appartiene al dominio.

2. Calcola $\phi_i(n)$ per m passi. Termina? Allora $< m, n > \in I$ e $f(< m, n >) = n$.
   Se non termina, $f(< m, n >) = \overline{n}$.
   Per esempio $\phi_i(2)$ per 4 passi. Converge? Se non converge, allora $f(< 4, 2 >) = \overline{n}$. Se converge, pongo $f(< 4, 2 >) = 2$

1.13 K e Riduzioni

1.13.1 Insieme K

K $= \{x \mid \phi_x(x) \downarrow\}$ insieme degli algoritmi applicati a sé stessi e convergenti.

K ricorsivamente enumerabile? Si.
Prendo l'x-esima macchina, la applico a x e converge $\Rightarrow x \in K$. Quindi K è dominio di una funzione.

K ricorsivo? No.
Dimostrazione Per assurdo, **K** ricorsivo. Allora $\chi_K(x) = \begin{cases} 1 & se\ x \in K \\ 0 & altrimenti \end{cases}$ è calcolabile totale.
Prendiamo $f(x) = \begin{cases} \phi_x(x) + 1 & se\ \chi_K(x) = 1 \\ 0 & altrimenti \end{cases}$ Poiché χ è calcolabile totale, anche f lo è.
Proviamo quindi a cercare il numero di f, ma **non lo troviamo**. Perché poniamo i come indice di f, allora $\phi_i(i) = f(i) = \phi_i(i) + 1$ se $i \in K$, ma non può essere. Se $i \notin K$ non può essere $\phi_i(i) = f(i) = 0$ perché $\phi_i(i) \uparrow$

Osservazione Abbiamo appena visto che **non esiste un algoritmo per decidere se** $x \in K$, che quindi **è un problema insolubile** (ma semi-decidibile).
Inoltre $\overline{K}$ **non è ricorsivamente enumerabile**, quindi esistono problemi ancora più difficili di K. Infatti, se $\overline{K}$ fosse ricorsivamente enumerabile, sia K che $\overline{K}$ sarebbero ricorsivi (perché K è ricorsivamente enumerabile). Abbiamo stabilito che

Ricorsivi $\subsetneq$ **ricorsivi enumerabili** $\subsetneq$ non **ricorsivi enumerabili**

e che $\overline{K} \in$ *co-RE*, cioè la classe dei problemi i cui complementari sono ricorsivamente enumerabili ma non ricorsivi.

Bootstrapping Cross-compiler: un compilatore scritto in un certo linguaggio L, che compila $L \to A$, con A altro linguaggio. Se L non gira su una determinata macchina, posso applicarlo a sé stesso: produrrà qualcosa scritto in A che prende L e produce A.

$$C_L^{L\to A}(C_L^{L\to A}) = C_A^{L\to A}$$

Questo assomiglia alla nostra K.
$K_0 = \{(x, y) \mid \phi_y(x) \downarrow\}$: scrivo un programma che prende un altro programma in input e testa per verificare se termina su un input. Questo è il **problema della fermata** (**halting**), che detto in altri termini: dato x e y, il programma $P_y(x)$ termina?
K_0 non è ricorsivo. Perché $x \in K \Leftrightarrow (x, x) \in K_0$, quindi se K_0 fosse ricorsivo lo sarebbe anche K.

1.13.2 Riduzioni

Riduzione Una riduzione è una particolare funzione che trasforma un problema (ovvero un insieme o una classe) A in un altro problema B preservando inalterata la caratteristica principale.

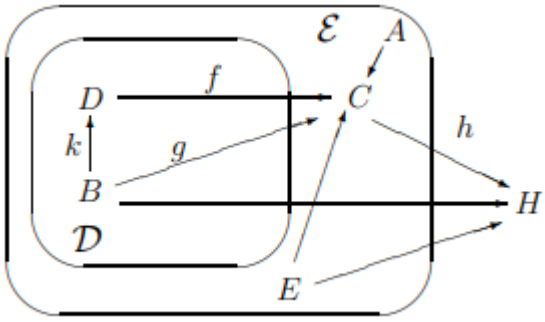
f è una **riduzione** da A a B, si scrive $A \leq_f B \Leftrightarrow (\forall x \in A \Leftrightarrow f(x) \in B)$

Proprietà: $A \leq_f B \Leftrightarrow \overline{A} \leq_f \overline{B}$

Famiglia di riduzioni $A \leq_F B \Leftrightarrow \exists\, f \in F \mid A \leq_f B$. Di queste ci interessano quelle che mantengono determinate proprietà.

Classi di problemi $\mathscr{D}, \mathscr{E}$ **classi di problemi** $\mathscr{D} \subseteq \mathscr{E} (\subseteq$ ricorsivi), allora $\leq_F$ **classifica** $\mathscr{D}$, $\mathscr{E} \Leftrightarrow \forall A, B, C$ problemi:

1. $A \leq_F A$
   Cioè l'identità $\in F$
2. $A \leq_F B$, $B \leq_F C \Rightarrow A \leq_F C$
   Cioè $f, g \in F \Rightarrow f(g) \in F$, f chiusa rispetto alla composizione.
3. $A \leq_F B$, $B \in \mathscr{D} \Rightarrow A \in \mathscr{D}$
   Preordine parziale, perché non vale la simmetria.
   Potrebbe $A \leq_F B$, $B \leq_F A$ ma non coincidere
   D è **ideale**, o **chiuso all'ingiù per la riduzione**.
4. $A \leq_F B$, $B \in \mathscr{E} \Rightarrow A \in \mathscr{E}$
   E è **ideale**, o **chiuso all'ingiù per riduzione**.



1.13.3 Problema Arduo

H è $\leq_F$-arduo per $\mathscr{E} \Leftrightarrow \forall A \in \mathscr{E} \quad A \leq_F H$

1.13.4 Problema Completo

C è $\leq_F$-completo per $\mathscr{E} \Leftrightarrow C \leq_F$-arduo e $C \in \mathscr{E}$
Sia $\leq_F$ classifica $\mathscr{D}$ ed $\mathscr{E}$, allora C $\leq_F$-completo per $\mathscr{E}$, C $\subset \mathscr{D} \Leftrightarrow \mathscr{D} = \mathscr{E}$

Per ipotesi, $C \in \mathscr{D}, A \in \mathscr{E}$ allora $A \leq_F C$. Allora, per 3, anche $A \in \mathscr{D}$
Se $A \leq_F$-completo per $\mathscr{E}$, $B \in \mathscr{E}$ e $A \leq_F B \Leftrightarrow B \leq_F$-completo per $\mathscr{E}$. Quindi se un problema completo si riduce ad un altro problema della stessa classe, allora anche quest'altro problema è completo.
Questo perché $\forall D \in \mathscr{E} \Rightarrow D \leq_F A, A \leq_F B$ e per 2 $D \leq_F B$

Se $A \leq_F B$ allora, se la riduzione misura in qualche modo la difficoltà di un problema, A **è al massimo difficile quanto** B e B **è più difficile (o uguale) di** A.
Quindi dovremmo cercare una famiglia di riduzioni che classificano $\mathscr{R}$ (ricorsive) e $\mathscr{RE}$ (ricorsive enumerabili).

1.14 Classificare R ed RE

Definiamo un insieme di funzioni $REC = \{\phi_x \mid dom(\phi_x) = N\}$ come l'**insieme di tutte le funzioni calcolabili**.

$$TOT = \{x \mid \phi_x \in REC\}$$

$$I \leq_{REC} J \Leftrightarrow \exists\, f \in REC \mid x \in I \Leftrightarrow f(x) \in J$$

Per decidere la non appartenenza ad un insieme ricorsivamente enumerabile è necessario un **tempo infinito**.

Th La **relazione di riduzione** $\leq_{REC}$ classifica $\mathscr{R}$ e $\mathscr{RE}$

Dim Sappiamo che $\mathscr{R} \subseteq \mathscr{RE}$. Allora

1. $id \in REC$, dalla definizione di μ-ricorsiva
2. f, $g \in REC \Rightarrow g(f) \in REC$ perché la composizione conserva la totalità
3. $B \in \mathscr{R} \Rightarrow A = \{x \mid f(x) \in B\} \in \mathscr{R}$
   Per vedere se $A \in \mathscr{R}$ devo vedere se la funzione di A è una funzione caratteristica calcolabile totale
   $\chi_A = \chi_B + f$ che è calcolabile totale perché χ_B è calcolabile totale
4. **idem** per $\mathscr{RE}$ con la funzione semi-caratteristica di B, ϕ_i il cui dominino è B

Il fatto che $\leq_{REC}$ classifichi $\mathscr{R}$ ed $\mathscr{RE}$ può essere visto come la capacità che hanno le funzioni calcolabili totali di **separare i problemi ricorsivi da quelli ricorsivamente enumerabili**: ciò avviene **in base al tempo necessario per decidere un problema**. Se il problema è **ricorsivo**, allora avremo la **risposta in tempo finito**, altrimenti il tempo necessario è infinito.
Inoltre, basta trovare un problema che sia $\leq_{REC}$-completo per $\mathscr{R}$ per poter vedere quali problemi sono decidibili e quali no.
Ancora più interessante è trovare un problema $\leq_{REC}$-completo per $\mathscr{RE}$: sapremmo quali sono i problemi *al più* semi-decidibili e quale nemmeno semi-decidibili. Infatti basta ridurre il problema da studiare a quello completo e sapremo che è ricorsivamente enumerabile, oppure ridurre il problema completo a quello da studiare e sapremo che quest'ultimo, alla meglio, è ricorsivamente enumerabile.
Infatti è chiaro anche che $A \leq_{REC} B$, quindi

B è ricorsivamente enumerabile (B $\in \mathscr{RE}$) $\Rightarrow$ A $\in \mathscr{RE}$ (e forse A $\in \mathscr{R}$)

A $\notin \mathscr{RE} \Rightarrow$ B $\notin \mathscr{RE}$ e sicuramente anche B $\notin \mathscr{R}$

Se A $\in \mathscr{R}$, il fatto che A $\leq_{REC}$ B non ci consente di dedurre niente sulla natura di B, che potrebbe essere ricorsivo o ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile.
Analogamente nel caso di A ricorsivamente enumerabile.

$K \leq_{REC} \overline{K}$? No. Perché $K \leq_{REC} \overline{K} \Leftrightarrow \overline{K} \leq_{REC} K$ e sarebbero entrambi $\mathscr{RE}$ ed è assurdo perché sappiamo che K non è ricorsivo.

Oss K è $\leq_{REC}$-completo per $\mathscr{RE}$, cioè K è $\mathscr{RE}$-completo

Dim $K \in \mathscr{RE}$ ok.
Dimostriamo che $\forall A \in \mathscr{RE} \Rightarrow A \leq_{REC} K$
Questo perché $A \to \exists\ \psi \mid A = dom(\phi_i)$ con ψ calcolabile $\Rightarrow \exists\ \psi'$ calcolabile $\mid \psi'(x,y) = \psi(x)$
$A = dom(\psi) = \{x|\psi(x)\downarrow\} = \{x|\psi'(x,y)\downarrow\}$ e siccome è calcolabile allora ha indice: diciamo i
Quindi $A = \{x \mid \phi_i(x,y)\downarrow\} = \{x \mid \phi_{s(i,x)}(y)\downarrow\} = \{x \mid \phi_{s(i,x)}(s(i,x))\downarrow\}$ con $y = \lambda x.s(i,x)$ e perché y non dipende da $\phi_{s(i,x)}$ e per il teorema del parametro.
Ottengo $\{x \mid s(i,x) \in K\}$ quindi $x \in A \Leftrightarrow f(x) \in K$ con $f(x) = \lambda x.s(i,x)$ che è totale, calcolabile e iniettiva perché lo è $s(i,x)$

Esercizio $K \leq_{REC} TOT$?
Definisco $\psi(x,y) = 1$ se x $\in$ K, indef altrimenti
La condizione $x \in K$ deve essere calcoalbile, così l'intera ψ è calcolabile
ψ calcolabile allora ha indice per C-T $\phi_i(x,y)$ ma se ha indice allora ho il teorema del parametro $\phi_{s(i,x)}(y)$ e posso usare la $f(x) = \lambda x.s(i,x)$ quindi

$$\phi_{f(x)}(y)$$

Vediamo i due casi

$x \in K$, $\forall y \psi(x,y) = 1 \Rightarrow \forall y \phi_{f(x)}(y) = 1 \Rightarrow f(x) \in TOT$ perché $\phi_{f(x)}(y)$ è calcolabile totale e $f(x)$ è indice

$x \notin K$, $\forall y \psi(x,y) =$ indef. $\Rightarrow \forall y \phi_{f(x)}(y) =$ indef. $\Rightarrow f(x) \notin TOT$

I, $x \in I$, $\phi_y = \phi_x \Rightarrow y \in I$ insieme degli indici delle funzioni che calcolano la stessa funzione di ϕ_x
I è un **insieme di indici che rappresentano le funzioni**. Caratterizzo le funzioni attraverso gli algoritmi che le calcolano.

Lemma A è un **insieme di indici che rappresentano funzioni** $A \neq \emptyset$ e $A \neq \mathbb{N}$
$\Leftrightarrow \forall\ x, y$ ho $x \in A \wedge \phi_x = \phi_y \Rightarrow y \in A$
$K \leq_{REC} A$ oppure (non esclusivo, possono essere vere entrambe) $K \leq_{REC} \overline{A}$

Dim i_0 sia l'indice della funzione ovunque indefinita $\phi_{i_0} = \lambda x.\text{indefinita}$
$i_0 \in A \vee i_0 \in \overline{A}$, sicuramente in uno dei due perché $A \neq \emptyset$ e $A \neq N$ quindi $\overline{A}$ "*ha spazio*".
Sia $i_0 \in \overline{A}$, dimostro $K \leq_{REC} A$ (viceversa è una dimostrazione analoga) perché sia $i_1 \in A$ allora $\phi_{i_0} \neq \phi_{i_1}$
$\psi(x,y) = \begin{cases} \phi_{i_1}(y) & \text{se } x \in K \\ \phi_{i_0}(y) & \text{altrimenti} \end{cases}$ Calcolabile, per C-T ha indice: quindi, per il ragionamento precedente, $= \phi_{f(x)}(y)$

$x \in K$ allora $\phi_{f(x)}(y) = \phi_{i_1}(y) \Rightarrow$ siccome $i_1 \in A$ insieme di indici che rappresentano funzioni e quindi $f(x) \in A$

$x \notin K$ allora $\phi_{f(x)}(y) = \phi_{i_0}(y) \Rightarrow f(x) \in \overline{A} \Rightarrow f(x) \notin A$

Lemma A è ricorsivo $\Leftrightarrow A = \emptyset \vee A = \mathbb{N}$

1.15 Teorema di Rice

"*Con la sintassi si va poco lontano.*"
Sia $\mathscr{A}$ un insieme di funzioni calcolabili.
L'insieme $A = \{x \mid \phi_x \in \mathscr{A}\}$ è **ricorsivo** $\Leftrightarrow \mathscr{A} = \emptyset$ oppure $\mathscr{A}$ **contiene tutte le funzioni calcolabili**.

Dim Corollario del lemma precedente. Si noti che A è un'insieme di indici, mentre $\mathscr{A}$ è una classe di funzioni.
$A = \emptyset$ allora ok
$A = \mathbb{N}$ allora $\mathscr{A}$ rappresenta tutte le funzioni calcolabili quindi A ricorsivo, altrimenti K si riduce ad A o $\overline{A}$ e si riduce alla dimostrazione di prima.
Si noti che A è un insieme di indici, mentre $\mathscr{A}$ è una **classe di funzioni**: A è **sintassi**, mentre $\mathscr{A}$ è **semantica**.

K non è un insieme di indici che rappresentano funzioni Dimostriamolo.
Non deve valere $\forall\ x \in K \quad \phi_x = \phi_y \Rightarrow y \in K$
Definiamo $\psi(x,y) = \begin{cases} 42 & \text{se } x = y \\ \text{indefinita} & \text{altrimenti} \end{cases}$
Per C-T e per il teorema del parametro ho $\psi(x,y) = \phi_i(x,y) = \phi_{s(i,x)}(y) = \phi_{f(x)}(y)$
$\exists$ un punto fisso per $f(x) \Rightarrow \phi_{f(x)}(y) = \phi_x(y)$
Per $\psi(p,p) = \phi_p(p) = 42 \Rightarrow p \in K$
Per il padding lemma, $\exists$ un indice $z \neq p \mid \phi_z = \phi_p$
Però $\psi(p,q) = \phi_z(t) = \phi_p(z)$ è indefinita. Quindi $q \notin K$ e K non è un i.i.r.f.

1.16 Considerazioni

Questo risultato si ripercuote sulle proprietà che si possono dimostrare sui programmi: ogni metodo di prova si scontra con il problema della fermata. Ci sono però varie tecniche per aggirare il problema, ad esempio l'**analisi statica** del codice, dove si analizza il **testo** del programma, per raccogliere informazioni su come vengono usati durante l'esecuzione i vari oggetti (variabili, chiamate...), per esempio se vengono rispettati i tipi, se si inizializzano...
Si ha successo, con questo tipo di analisi, perché il **programma è approssimato in modo sicuro**: ciò che viene predetto è una sovra-approssimazione di ciò che succederà davvero. Per esempio, può succedere di dire che tra i valori assegnati ad una variabile `int` c'è una `String` senza che ciò accada a runtime, ma non capiterà mai di dire che tutti i valori assegnati sono `int` se a runtime a tale variabile viene assegnata una `String`.
A questa famiglia appartengono vari strumenti, spesso incorporati nei compilatori: type-checker, analizzatori data-flow e control-flow...

Applicazione Un'applicazione del teorema di Rice è che $K_1 = \{x \mid dom(\phi_x) \neq \emptyset\}$, cioè l'insieme degli indici delle funzioni definite in almeno un punto **non è ricorsivo**, sebbene sia ricorsivamente enumerabile.
Inoltre K, K_0 e K_1 si riducono l'un l'altro.

Fine della Calcolabilità

Capitolo 2

Complessità

Fin'ora Abbiamo studiato alcuni aspetti della teoria della calcolabilità: in particolare, abbiamo caratterizzato i **problemi risolubili** (rappresentati da funzioni calcolabili totali o da insiemi ricorsivi), quelli **insolubili** o **semi-decidibili** (rappresentati da funzioni calcolabili parziali o insiemi ricorsivamente enumerabili) e abbiamo discusso **problemi non decidibili** (rappresentati da funzioni non calcolabili o insiemi non ricorsivamente enumerabili.
Ora che abbiamo chiaro *cosa* si può calcolare, ci si può porre il problema del *come* calcolare. Sapendo distinguere il *come* si calcola, riusciamo a distinguere tra loro le varie tecniche di calcolo, quindi a distinguere problemi *facili* da *difficili*. Primi passi verso una **teoria quantitativa** degli algoritmi.
Considereremo solo i problemi decidibili e di decisione ($x \in I$), cioè quei problemi o insiemi per cui esiste un algoritmo che ne calcoli la funzione caratteristica, e parleremo solamente delle risorse di tempo e spazio. Per vederle, modificheremo leggermente le MdT già viste trasformandole in **automi accettori**: cioè macchine che finiranno in uno stato di accettazione se l'ingresso appartiene all'insieme, altrimenti in uno stato di rifiuto.

Taglia La **taglia** di un problema è il **numero di ingressi** della funzione, denotata con $|x|$

Valutare Avere una f funzione che **valuta asintoticamente l'utilizzo delle risorse**, cioè
$f \mid f(|x|) \geq$ tempo/spazio. Studieremo la **complessità asintotica** al **caso pessimo**, per semplificazione.
Questa $f \geq$ determina una **classe di complessità**.
Le risorse si misurano in vari modi: numero di passi, nastri usati, numero di riduzioni fatte...

Questa teoria è invariante rispetto al cambio di modello di calcolo ed alla rappresentazione dei dati.

Classi di gerarchia Come si relazionano fra loro? Ce ne sono di particolarmente interessanti, caratterizzate da problemi particolarmente interessanti? Sicuramente abbiamo già sentito parlare della **classe di problemi risolvibili in tempo polinomiale deterministico P** e anche quella dei **problemi risolvibili in tempo polinomiale non deterministico NP**, che sono due classi particolarmente interessanti cercando la loro relazione e le loro proprietà.

$$\mathscr{RE} \supset \mathscr{R} \supset NPSPACE = PSPACE \supseteq \mathscr{NP} \supseteq \mathscr{P} \supseteq Logspace$$

Considerando

> *Logspace*: insieme delle funzioni il cui tempo di calcolo è asintoticamente simile al logaritmo della taglia
>
> *NPSPACE* e *PSPACE* uguali, ma diversi da *Logspace*
>
> Non esiste ancora una dimostrazione di $\mathscr{P} \subseteq \mathscr{NP}$
>
> Per decidere la non appartenenza ad $\mathscr{R}$ basta un tempo finito, ma per decidere la non appartenenza a $\mathscr{RE}$ no.

Problema completo Data una famiglia di riduzioni F che classifica due classi $\mathscr{D}$ ed $\mathscr{E}$, un problema I è $\leq_F$-completo per $\mathscr{D}$ (analogo per $\mathscr{E}$) $\Leftrightarrow \forall\, J \in \mathscr{D}\ J \leq_F I$

Tesi di Cook-Karp $\mathscr{P}$ è la **classe dei problemi trattabili** mentre $\mathscr{NP}$ è la **classe dei problemi intrattabili**

Bisogna ora decidere *come* si misuro lo spazio o il tempo necessario per decidere $x \in I$.

2.1 Misure di complessità deterministiche

MdT a k nastri MdT come tutte le altre (Q, Σ, δ, q_0)
La funzione di transizione δ è un po' diversa e h, stato di terminazione, viene diviso in 2: stato **si** (stato di accettazione) e stato **no** (stato di rifiuto), entrambi $\notin$ Q. Operano in **modo sincrono su k nastri**.

Per ciascun simbolo, δ dovrà fornire il nuovo simbolo e spostare sincronamente le testine quindi fornire una direzione: $\delta(q, \sigma_1, \sigma_2, \ldots, \sigma_k) = (q', (\sigma'_1, D_1), (\sigma'_2, D_2), \ldots, (\sigma'_k, D_k))$
La configurazione iniziale sarà leggermente diversa, quindi: $\gamma = (q_0, u_1\sigma_1 v_1, u_2\sigma_2 v_2, \ldots, u_k\sigma_k v_k)$
Una transizione d'esempio: $(q, u_1\sigma_1 v_1, u_2\sigma_2 v_2, \ldots, u_k\sigma_k v_k) \to (q', u'_1\sigma'_1 v'_1, u'_2\sigma'_2 v'_2, \ldots, u'_k\sigma'_k v'_k)$
$\delta : Q \times \Sigma^k \longrightarrow Q \cup \{\text{SI}, \text{NO}\} \times (\Sigma \times \{\text{L}, \text{R}, -\})^k$

Tempo Il tempo di richiesto da M a k nastri per decidere $x \in I$ è $t \Leftrightarrow$ partendo dallo stato iniziale q_0 ho $(q_0, \trianglerighteq x, \trianglerighteq, \ldots, \trianglerighteq) \longrightarrow^t_M (H, w_1, \ldots, w_k)$ con $H \in \{\text{si}, \text{no}\}$
Non si può sempre decidere il numero esatto di passi, ci accontenteremo di una **stima superiore** $f(|x|)$, con $|x| = \text{taglia}(x)$. Non sapremo come calcolare la taglia in generale, ma deve essere relativamente **facile** farlo: lunghezza di un vettore, dimensione in memoria...
M **decide** I in tempo $f \Leftrightarrow \forall\, x$ il tempo t per decidere $x \in I$ è tale che $t \leq f(|x|)$

$$\textbf{TIME}(f) \quad = \{I \mid \exists\, M \text{ che decide } I \text{ in tempo deterministico } f\}$$

$O(f) = \{g \mid \exists\, r \in R$ per cui $g(n) < r \cdot f(n)\}$ salvo un pezzo iniziale. Cioè la f da un certo punto in poi sta sopra la g.
$\Omega(f)$ è quando la f sta sotto.
$\Theta(f)$ è quando le funzioni crescono allo stesso modo.

2.1.1 Teorema di Riduzione del Numero di Nastri

Teorema Teorema che dimostra che usando una macchina parallela si "guadagna" tempo in modo quadratico.
$\forall\, M$ a k nastri che decide I in $O(f)$ deterministica $\Rightarrow \exists\, M'$ a 1 nastro che decide I in $O(f^2)$

Dim : da ${}_k\gamma$ (a k nastri) $\longrightarrow ({}_1q, \triangleright \triangleright' u_1\overline{\sigma_1}v_1 \triangleleft' \triangleright' u_2\overline{\sigma_2}v_2 \triangleleft' \ldots \triangleright' u_k\overline{\sigma_k}v_k\triangleleft')$ usando le "parentesi $\triangleright'$ e $\triangleleft'$ per separare i k nastri sul nastro singolo .
$\sigma_i \longrightarrow \overline{\sigma_i}$

1. Sistemare il nastro iniziale generando la configurazione iniziale di M: $(q_0, \trianglerighteq \triangleright' x \triangleleft' \triangleright' \triangleleft' \ldots \triangleright' \triangleleft')$
   Bastano $2k + \#\Sigma$ nuovi stati: si trova il primo #, si torna indietro cambiando stato e si codifica il simbolo $\neq$ # che c'è, sostituirlo con #, spostarsi a destra scrivendo il simbolo ricordato. Ripetere gli ultimi due passi per $|x| - 1$ volte, mentre andare fino al primo # richiede $|x| + 1$ passi e un nuovo stato. Ora si scrive $\triangleright'$ sulla casella # corrente, si torna sulla prima vuota muovendosi a destra scrivendo $k - 1$ coppie $\triangleright'\triangleleft'$ usando altri $2 \cdot (k - 1)$ stati.

2. Per simulare una mossa di M, M' scorre il nastro da sinistra a destra per scovare i caratteri correnti, quelli sovralineati: avanti e indietro una volta.
   Riscorro il nastro per scrivere i nuovi simboli e fissare i nuovi correnti: avanti e indietro seconda volta (con una cascata di spostamenti a destra se bisogna spostare un $\triangleleft'$ a destra.

Non si può usare più spazio che tempo Non si possono vedere più caselle di quanti passi si fanno.
Inoltre per simulare un passo ci vogliono al massimo 4K passi.
Il nastro contiene al massimo $K = k \cdot (2 + f(|x|)) + 1$ con $|x| = \text{taglia}(x)$
2 per le parentesi di ogni nastro, 1 per il respingente iniziale.

M fa $f(|x|)$ passi, M' fa $f(|x|) \cdot 4K$ quindi $f(|x|) \cdot \not{k} \cdot (\not{2} + f(|x|)) + \not{1})$ con la roba cancellata perché non dipende dall'input ma dalla macchina.
Quindi M fa $O(f)$ mentre M' fa $O(f^2)$

Conseguenze Miglioramenti accettabili "algoritmicamente", come aggiungere nastri o processori che operino in parallelo, non cambiano le funzioni calcolate e non modificano il tempo richiesto se non polinomialmente: le MdT sono stabili anche rispetto alla testi di Cook-Karp.

2.1.2 Teorema di Accelerazione Lineare

Teorema Se $I \in \text{TIME}(f(n)) \Rightarrow \forall\, \epsilon \in R^+$ si ha $I \in \text{TIME}(\epsilon \cdot f(n) + n + 2)$

Dim Intuizione su come determinare ϵ: **condensare un certo numero di caratteri in un carattere unico** $\sigma_1\sigma_2 \dots \sigma_m \longrightarrow [\sigma_1\sigma_2 \dots \sigma_m]$, con $[\sigma_1\sigma_2 \dots \sigma_m]$ **unico carattere**.
Stati codificati in una tripla $[q, \sigma_1\sigma_2 \dots \sigma_m, k]$ con k che indica la posizione del cursore.
Equivale ad avere macchine con parole di dimensioni crescenti, 2^m bit.

M' in 6 passi simula m passi di M, con M' macchina veloce e M macchina lenta, questo perché partizionando l'input in blocchi di m caratteri i cambiamenti possono essere fatti in blocchi contigui. Nei primi 4 passi M' fa a sinistra, poi a destra, poi ancora a destra e poi ritorna sul carattere corrente $s = [\sigma_1\sigma_2 \dots \sigma_m]$ per raccogliere i simboli che M potrebbe visitare e codificarli nep proprio stato. Quando M compie m mosse, M' le simula muovendosi a sinistra, o a destra, del simbolo corrente s, comunque modificando solo due simboli incluso s (le altre 2 mosse).
M' farà $|x| + 2$ passi per condensare l'input $+ 6 \cdot \lceil \frac{f(|x|)}{m} \rceil$ passi, quindi posso prendere come $m = \lceil \frac{6}{\epsilon} \rceil$

Risultati Quindi se ho un algoritmo che decide un problema in tempo deterministico $f(n)$, allora l'algoritmo $\in$ TIME($f(n)$) e posso costruirne un altro che lo calcola in tempo deterministico $\epsilon f(n) + n + 2$, questo per ogni ϵ: posso accelerare l'algoritmo **linearmente** quanto voglio.
Quindi se $f = c \cdot n$ posso eliminare c e mettere $\epsilon = \frac{1}{c}$, se $f = c_1 n^k + \dots + c_{k-1} n + c_k$ posso rendere $c_1 = 1$ e ignorare gli addendi con esponenti minori di k perché vengono dominati da n sufficientemente grande, giustificando l'uso di $O(n^k)$.
Questo teorema **consente di usare gli ordini di grandezza per approssimare** senza *fare niente di male*. Inoltre si possono eliminare le costanti (dipendenti dalle macchine su cui si calcola) perché macchine più potenti tendono a farle rimpicciolire e valori grandi di $|x| = n$ tendono a ridurre l'*importanza* delle costanti.

Grazie a questi risultati definisco $\mathscr{P}$, la classe dei problemi decidibili in tempo polinomiale deterministico, come

$$\mathscr{P} = \bigcup_{k \geq 1} TIME(|n|^k)$$

Ricapitolando

M richiede tempo t per decidere $I \Leftrightarrow M(x) = (q_0, \triangleright x, \triangleright, \dots, \triangleright) \rightarrow^t (h, w_1, \dots, w_k)$ con $h \in \{si, no\}$

M decide I in tempo $f \Leftrightarrow \forall\, x \in I\ M(x) \rightarrow^t (si, w_1, \dots, w_k)$ cioè M richiede tempo t per deciderlo, con $t \leq f(|x|)$

TIME(f) $= \{I \mid \exists\, M$ che decide I in tempo $f\}$
$\mathscr{P} = \bigcup_{k \geq 1} TIME(|n|^k)$

Abbiamo anche scoperto che **non si può usare più spazio che tempo**.

Spazio necessario al calcolo Quantità di celle che il programma utilizza durante l'esecuzione.
Ignoreremo lo spazio utilizzato dal nastro d'ingresso.

2.2 MdT I/O a k nastri

Si tratta di una MdT in cui il **primo nastro contiene l'ingresso** mentre l'**ultimo nastro contiene l'uscita**.

$$\delta(\sigma_1, \dots, \sigma_k) = (q', (\sigma'_1, D_1), \dots, (\sigma'_k, D_k))$$

Nastro d'ingresso Ogni δ come sopra deve essere tale che il primo nastro (σ'_1, D_1) sia di **sola lettura** perché è il nastro dell'input.
La **condizione** quindi è che $\sigma'_1 = \sigma_1$ **sempre**, cioè **il nastro di ingresso è di sola lettura**.

Nastro d'uscita Analogamente ci deve essere una condizione sul nastro d'uscita:

$D_k = R$

oppure $D_k = - \Rightarrow \sigma'_k = \sigma_k$, cioè **nastro d'uscita di sola scrittura**.

Altra condizione Inoltre, bisogna costruire le δ in modo che quando finisco di leggere il nastro d'ingresso, finendo quindi sul carattere bianco #, io debba necessariamente tornare indietro. Altrimenti, anche se non scrivo niente, potrei usare le operazioni di spostamento sui caratteri bianchi per codificare delle operazioni. Quindi

$\sigma_1 = \# \Rightarrow D_1 \in \{L, -\}$

Teorema Collega le MdT a k nastri con le MdT di tipo I/O a $k+2$ nastri
$\forall\, M$ a k nastri che decide I in tempo deterministico $f \Rightarrow \exists\, M'$ a $k+2$ nastri I/O che decide I in tempo $c \cdot f$
La dimostrazione, intuitivamente, consiste per M' nel copiare il nastro d'ingresso sul secondo nastro e, a computazione conclusa ed eseguita come M, copiare il nastro d'uscita sull'ultimo nastro.

2.2.1 Complessità in spazio

M a k nastri di tipo I/O che va $(q_0, \triangleright x, \triangleright, \ldots, \triangleright) \to^t (h, w_1, \ldots, w_k)$ con $h \in \{si, no\}$ richiederà, come spazio, quello dei nastri di lavoro w_i. Hanno spazio costante, quindi so in principio qual'è. Quindi:

M a k nastri I/O richiede spazio $\sum_{i=2}^{k-1} |w_i| \Leftrightarrow (q_0, \triangleright x, \triangleright, \ldots, \triangleright) \to^t (h, w_1, \ldots, w_k)$

Questo perché il numero delle celle usate dai nastri non diminuisce mai: nel primo nastro perché è di sola lettura, nell'ultimo perché è di sola scrittura e nei $k-2$ nastri di lavoro perché i caratteri bianchi a destra non scompaiono mai.

Ad esempio, per $k = 3$ avrò un unico nastro di lavoro, quindi lo spazio occupato dalla macchina sarà quello del nastro di lavoro (come detto prima, ignoriamo l'occupazione in spazio dei nastri input e output).

M decide I in spazio $f \Leftrightarrow \forall\, x$ M richiede spazio $\leq f(|x|)$

SPACE(f) $= \{I \mid \exists\, M$ che decide I in spazio $f\}$

Ignoriamo lo spazio occupato dai nastri di input e output perché altrimenti avremmo complessità in spazio almeno lineari: questo perché il nastro in input misura sempre $|x| + 1$ contenendo $\triangleright x$, mentre il nastro in output contiene un "bit", un segnale per accettare o rifiutare, nel caso dei problemi I di decisione considerati, quindi non rilevante nelle misure.
Esistendo classi interessanti e importanti che sono sub-lineari nello spazio, come visti successivamente, ci interessano misure più fini.

Teorema Posso ridurre linearmente lo spazio.
$I \in \text{SPACE}(f) \Rightarrow \forall\, \epsilon \in R^+$ $I \in \text{SPACE}(2 + \epsilon \cdot f(n))$

$$PSPACE = \bigcup_{k \geq 1} SPACE(n^k)$$

$$LogSPACE = \bigcup_{k \geq 1} SPACE(k \cdot \log(n))$$

Come per $\mathscr{P}$, anche queste classi sono invarianti rispetto al modello di calcolo usato.
Abbiamo inoltre che $LogSPACE \subsetneq PSPACE$

Teorema *LogSPACE* $\subseteq \mathscr{P}$

Dim Sia $I \in$ *LogSPACE* $\Rightarrow \exists\, M$ di tipo I/O che $\forall\, x \in I$ decide x in spazio $O(\log |x|)$
Quante **configurazioni diverse posso avere sul nastro di lavoro**? Dipende da quanti simboli ho sulla macchina. Poniamo x input e la sua lunghezza $|x| = n$, avrò $\#\Sigma^{\log(n)}$ nastri diversi, cioè il numero di simboli (cardinalità di Σ) alla $\log(n)$. Inoltre posso avere il cursore in $\log(n)$ posizioni diverse in ciascun stato (che sono $\#Q$). Inoltre il cursore sta anche nell'input, in $|x|$ posizioni diverse. Quindi le configurazioni diverse del nastro di lavoro sono

$$n \cdot \log(n) \cdot \#Q \cdot \#\Sigma^{\log(n)}$$

Per dimostrare che *LogSPACE* $\subseteq \mathscr{P}$ devo trovare k tale che $n \cdot \log(n) \cdot \#Q \cdot \#\Sigma^{\log(n)} \leq n^k$. Applico il log da entrambi i membri

$$\log(n) + \log(\log(n)) + \log(\#Q) + \log(n)\log(\#\Sigma) \leq k \cdot \log(n)$$

Elimino $\log(\log(n))$ perché è molto piccolo, e $\log(\#Q)$ perché è una costante, poi semplifico per $\log(n)$

$$\cancel{\log(n)} + \cancel{\log(n)}\log(\#\Sigma) \leq k \cdot \cancel{\log(n)}$$

$$1 + \log(\#\Sigma) \leq k$$

Quindi basta usare $k \geq \log(\#\Sigma)$.

Come conseguenza, scopriamo che **lo spazio limita il tempo**, perché se passo troppe volte sopra la solita configurazione vado in ciclo.

Ricapitolando MdT a k nastri, eventualmente di tipo I/O. $\forall x$ macchina $M(x) \to^t M$ in una situazione di alt. Riesce perché: **aggiungere nastri**, **avere I/O**, **aumentare la parola** su cui lavorano le macchine (quindi **compattare** simboli) sono tutte **operazioni algoritmicamente effettive**: si possono fare con algoritmi. Ciò fornisce un bel vantaggio.
Però sorge un **problema**: dobbiamo decidere un insieme (quindi risolvere un problema) che **sappiamo essere decidibile** e calcolare una funzione che **sappiamo essere calcolabile**, ma non riusciamo a individuare una struttura matematica e delle proprietà tramite le quali risolverlo *facilmente*.
Per esempio per costruire una macchina che decideva se una data stringa fosse palindroma, **abbiamo sfruttato una proprietà** delle stringa: che essa sia leggibile da destra a sinistra e da sinistra a destra. In questo caso era banale, ma ci sono problemi per cui non conosciamo la struttura e ciò rende la vita difficile.

2.2.2 Spazio degli stati

Lo spazio degli stati di una MdT può essere

Generato esplicitamente,

Generato implicitamente, guess & try

2.3 MdT non deterministica

N = (Q, Σ, Δ, q_0) **MdT non deterministica**

Relazione di transizione $\Delta \subseteq (Q \times \Sigma) \times (Q \times \{si,\, no\} \times \Sigma \times \{Q, L, -\})$
Possono essere presenti più quintuple associate allo stesso stato ed allo stesso simbolo, quindi molte configurazioni $(q', u'\sigma'v')$ raggiungibili da $(q, u\sigma v)$ in un solo passo

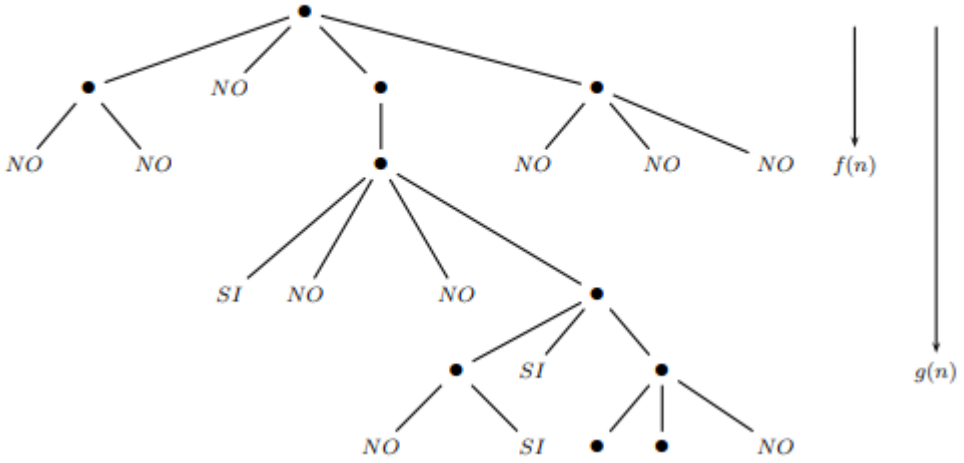
Stato iniziale $\gamma = (q, u\sigma v) \to (q', u'\sigma'v')$

Computazione $M(x) \to \gamma_1 \to \ldots \to \gamma_n$ cioè $M(x) \to^n \gamma_n$, un **cammino** (non un albero!)

La potenza del non determinismo si vede nel modo in cui si accetta: **basta che ci sia un cammino che porti alla soluzione** perché il problema sia risolto.

DNA Computing Calcolo eseguito ricombinando fra loro sequenze biologiche invece che per passi come da tradizione.

Come decide un problema una macchina non deterministica Di seguito la definizione.
N decide I $\Leftrightarrow \forall\, x \in I \Rightarrow \exists$ computazione $M(x) \to^* (si,\, w)$
Se $x \in I$ allora e solamente allora deve esistere *almeno una* computazione, un cammino, che accetta x.
Il contrario è che **tutte rifiutino**, allora $x \notin I$



Numerando gli archi di decisione identifico una computazione tramite la successione di numeri. Data una certa σ ho tante quintuple in Δ, le enumero e le ordino in qualche modo. Prendo la n_1 scelta dallo stato iniziale, la n_2 scelta dallo stato risultante...
Nell'esempio, una scelta Si è identificata dalla computazione $(2, 0, 0)$, un'altra da $(2, 0, 3, 1)$.

2.3.1 Misure di complessità non deterministica

Tempo N MdT non deterministica decide I in **tempo non deterministico** $f(|x|)$ (più semplicemente f) $\Leftrightarrow$

N decide I, e

$\forall\, x \in I \quad \exists\, t \mid N(x) \to^t (si,\, w) \wedge t \leq f(|x|)$

$NTIME(f) = \{I \mid \exists\, N$ che decide I in tempo non deterministico $f\}$
Quindi una MdT non deterministica accetta x in tempo non deterministico f se e solo se c'è **almeno** una computazione che termina in *si* in $t \leq f(|x|)$ passi. Invece, non accetta x se **tutte** le sue computazioni lunghe al massimo $f(|x|)$ terminano in *no*.
C'è nuovamente asimmetria: non basta che per un elemento $x \notin I$ ci sia una computazione che porta in *no* in meno di $f(|x|) + 1$ passi, ma devono farlo **tutte**. L'asimmetria tra il decidere I e $\overline{I}$ è rafforzata dal richiedere che ciò venga svolto in tempo $f(|x|)$.
Questo perché **affinché** N **decida** I **basta che per** ***ogni*** x **ci sia** ***almeno una*** **computazione che lo accetta**.

$$\mathscr{NP} = \bigcup_{k \geq 1} NTIME(n^k)$$

Abbiamo che $\mathscr{P} \subseteq \mathscr{NP}$ dato che ogni MdT deterministica è anche non deterministica.

Teorema Se $I \in NTIME(f) \Rightarrow I \in TIME(c^f)$ con $c \geq 1$ che **dipende soltanto dalla macchina** in $TIME(f)$.
In altre parole: se I è deciso da N a k nastri in tempo non deterministico $f(x)$, allora $\exists\, M$ a $k+1$ nastri che decide I in $O(c^{f(x)})$, $c \geq 1$ che dipende solo da N
In altre parole $NTIME(f) \subseteq TIME(c^f)$

Dim Il grado di diramazione massimo, o il **grado di non determinismo**, è

$d = max\{grado(q,\sigma) \mid q \in Q, \sigma \in \Sigma\}$

dove $grado(q,\sigma) = \#\{(q',\sigma',D) \mid ((q,\sigma),(q',\sigma',D)) \in \Delta\}$

Per ogni stato $q \in Q$ e ogni simbolo $\sigma \in \Sigma$ ordino totalmente, ottenendo così che ogni computazione è una sequenza di scelte lunga t, indicabili come una sequenza di t numeri naturali nell'intervallo $[0 \dots d-1]$. Una macchina M che simula N produce una successione di scelte partendo sempre dal principio. Per esempio riproduce la scelta 20300 e magari arriva in uno stato di accettazione. Se ci arriva, bene **ho finito**.
Se non ci arriva prende il prossimo numero $t+1$ in base d, nell'esempio 20301.
Se supera il numero di archi la stringa aumenta di un carattere e faccio 00, 01, 02... poi 10, 11, 12...
Sarà dell'ordine di $d^{f(n)}$ con $c = d$.
Ho una perdita esponenziale. Ma aggiungere il non determinismo non cambia le classi dei problemi.

Spazio N decide I in **spazio non deterministico** $f(|x|)$ (più semplicemente f) $\Leftrightarrow$

N decide I, e

$$\forall\, x \in I \;\; \exists\, w_1, \dots, w_k \mid (q_0, \trianglerighteq x, \dots, \trianglerighteq) \to_N^* (\mathrm{Si}, w_1, \dots, w_k) \text{ e } \sum_{2 \leq i \leq k-1} |w_i| \leq f(|x|)$$

$NSPACE(f) = \{I \mid \exists\, N$ che decide I in spazio non deterministico $f\}$
$NPSPACE = \bigcup_{k \geq 1} NSPACE(n^k)$

Teorema di Savitch $NPSPACE = PSPACE$

$$LogSpace \subseteq \mathscr{P} \subseteq \mathscr{NP} \subseteq PSPACE + NPSPACE \subset \mathscr{R} \subset \mathscr{RE}$$

$$\mathscr{P} ? \mathscr{NP}$$

Siamo giunti alla conclusione che una MdT non deterministica accetta x in tempo $f \Leftrightarrow \forall\, x \in I \; \exists$ una computazione $N(x) \to^t (\mathrm{Si}, w)$ con $t \leq f(|x|)$
Per confutare invece, bisogna che **tutte** le computazioni rifiutano, cioè per rifiutare x bisogna che $\forall$ computazione $N(x) \to^t (\mathrm{No}, w)$ con $t \leq f(|x|)$

2.3.2 Commesso Viaggiatore

n città connesse da strade, la distanza fra la città i e la città j è $d(i,j)$. Sulle distanze vale la proprietà riflessiva e la diseguaglianza triangolare.
Le distanze sono rappresentate con le coppie (i,j) che sono $O\left(\frac{n \cdot (n-1)}{2}\right)$. Il problema è trovare un cammino che tocchi le città una ed una sola volta, con costo minimo. Il costo è la somma delle distanze percorse.
Cerchiamo quindi una **permutazione** degli indici i, j in modo tale che la somma delle distanze sia minima. Una permutazione è una bigezione $\Pi : [1,n] \to [1,n]$.
Il nostro costo $\sum_{1 \leq i \leq n-1} d(i, i+1) \leq B$ con $\Pi(h) = i$ e $\Pi(k) = i+1$, cioè i e $i+1$ sono permutazioni di, rispettivamente, un certo h ed un certo k.

Forza Bruta Deterministica

1. **Generare l'intero spazio di ricerca**, cioè tutte le possibile permutazioni esistenti Π come stringhe, che sono nell'ordine di $n! = O(n!)$
2. **Certificazione**: prendere la prima permutazione e verificare se è $\leq B$.
   n accessi al nastro d'ingresso, uno per città, e n^2 per vedere la distanza di ciascuna città, quindi $O(n \cdot n^2)$

Esponenziale in tempo ma lineare $O(n)$ in spazio, perché si mantiene una singola permutazione alla volta.
Dire che non esiste soluzione richiede di valutare tutte le permutazioni.

MdT non deterministica

1. Bisogna **scrivere una MdT non deterministica** che generi ad ogni passo un numero tra 1 ed n compresi: abbiamo così una permutazione delle città.
   Con un po' di accorgimenti si evitano stringhe del tipo $111\ldots$, ad esempio scrivendo la MdT in modo che non rigeneri numeri già generati.
2. **Certificazione**: la stringa è una permutazione Π? Si scopre in n passi.
   Dopodiché faccio esattamente la certificazione precedente, per verificare se $\leq B$: tutto ciò si fa in $O(n^3)$ come prima.

2.4 Funzioni di valutazione

Una **funzione di valutazione** potrebbe essere una qualunque funzione calcolabile che dalla dimensione dell'input porti in un numero naturale che indica la valutazione: numero di passi, di moltiplicazioni...
Sappiamo che le classi di complessità formano una gerarchia: ad esempio la classe $TIME(O(n^3))$ cioè i problemi risolvibili in tempo cubico, include propriamente quelli risolvibili in tempo quadratico. Questo lo sappiamo dall'esperienza.
Perché queste funzioni funzionino dobbiamo porre loro dei vincoli: chiameremo queste funzioni **appropriate**/oneste/costruibili.
Vincoli:

Monotone crescenti
Significa che se devo risolvere un problema la cui dimensione è maggiore di un altro problema ci devo mettere più tempo che spazio.

$\exists M | \forall x \in \Sigma^*$ si arresta (dando come risultato $\diamond^{f(|x|)}$, $\diamond \notin \Sigma$ carattere speciale, lungo tanti caratteri quanto $f(|x|)$) in tempo $O(f(|x|) + |x|)$ con l'addendo $|x|$ per rendere lineare quando $f(|x|)$ è sub-lineare e spazio $O(f(|x|))$

Esempi: $n^k, \lfloor \log n \rfloor, n!, k^n, \ldots$
Se f, g sono appropriate, anche $f(g)$, f^g, $f \cdot g$, $f + g \ldots$: la classe è chiusa per una serie di operazioni.

2.4.1 Teorema di gerarchia

Data f appropriata, allora (ricordando che $TIME(f(n)) = \{I \mid \exists\, M$ che decide I in tempo $f(n)\}$)

$TIME(f(n)) \subsetneq TIME(f(2n+1)^3)$

$SPACE(f(n)) \subsetneq SPACE(f(n) \cdot \log(f(n)))$

La dimostrazione è omessa, si noti che quella del primo punto si basa sulla dimostrazione che il problema $\{x \mid \phi_x(x)$ termina in $f(|x|)$ passi$\}$, che è un pezzetto di K, appartiene a $TIME(f(2n+1)^3)$ ma non a $TIME(f(n))$
Analogamente per le MdT non deterministiche.

Possiamo ora dimostrare la nostra gerarchia, in particolare che $\mathscr{P} \subsetneq EXP = \bigcup_{k \geq 1} TIME(2^{n^k})$

Dim Sappiamo che $\mathscr{P} \subseteq EXP$, cioè $\mathscr{P} \subseteq TIME(2^n)$ perché al crescere di n per qualunque k, $n^k \leq 2^n$
Abbiamo quindi $\mathscr{P} \subseteq TIME(2^n) \subsetneq TIME(f(2^{(2n+1)^3})) \subseteq \bigcup_{k \geq 1} TIME(2^{n^k})$

2.4.2 Qualche assioma

- $SPACE(f(n)) \subseteq NSPACE(f(n))$
- $TIME(f(n)) \subseteq NTIME(f(n))$
- $NSPACE(f(n)) \subseteq TIME(k^{\log n + f(n)})$

Avevamo anche dimostrato che $NTIME(f(n)) \subseteq TIME(c^{f(n)})$ da cui deriva $\mathscr{NP} \subseteq EXP$
Si ricorda quindi anche che

$$LogSpace \subseteq \mathscr{P} \subseteq \mathscr{NP} \subseteq EXP$$

Abbiamo dei motivi per cui non possiamo fare a meno delle funzioni di misura appropriate.

2.4.3 Teorema

Troveremo sempre problemi via via più difficili, cioè la gerarchia non è superiormente limitata.
$\forall\, g$ calcolabile totale $\exists\, I$ problema $\mid I \in TIME(f(n)) \wedge I \notin TIME(g(n))$, con $f > g$ quasi ovunque
Quindi la gerarchia tratta dal teorema di gerarchia continua a crescere senza fermarsi mai.
Con *quasi ovunque* si intende **dappertutto eccetto che in un numero finito di punti**.

2.4.4 Teorema di Accelerazione (Blum)

Il teorema di accelerazione *lineare* diceva che possiamo accelerare *linearmente* la soluzione del problema.
Blum ci dice che **non sempre esiste un algoritmo ottimo** per un problema.

$\forall\, h$ calcolabile totale
$\exists\, I \mid \forall\, M$ che decide I in tempo f $\exists\, M'$ che decide I in tempo $f' \mid f(n) > h(f'(n))$ quasi ovunque

h indica quanto è più veloce M' rispetto a M
Intuitivamente: supponendo di avere una macchina universale lenta, e di ottenere una macchina universale nuova e velocissima, allora avremmo dei programmi che rimangono più veloci sulla macchina vecchia che non sulla nuova.
In altre parole, si può dimostrare l'esistenza di una successione di algoritmi via via più efficienti per risolvere il problema costruito in funzione di h. Si sa solo che esiste, ma non come costruirla.

2.4.5 Teorema della Lacuna (Borodin)

$\exists\, f$ calcolabile totale $\mid TIME(f) = TIME(2^f)$
Intuitivamente: c'è un insieme di programmi che sono altrettanto efficienti/veloci su una macchina lenta che su una macchina nuova.
Questo cozza con un risultato del teorema di gerarchia, che arrivava a dire (a grandi linee)
$TIME(f(n)) \subsetneq TIME(f(2n+1)^3)$ perché avrei $TIME(f^2) = TIME(f) \subsetneq TIME(f^3)$, ma quindi avrei una situazione assurda del tipo $A = B < C$, come può essere $A = C$? Ma il teorema di gerarchia vale. Ciò che cambia è che nel teorema della lacuna f non è richiesta appropriata. Questo ci dice che **non possiamo fare a meno delle funzioni di misura appropriate**, o di classi di funzioni con vincoli a come misuriamo il consumo delle risorse in modo che il teorema di gerarchia valga.

2.4.6 Teoria della Complessità Astratta

Si svincola dal tipo di risorsa misurata e dal modello di calcolo.
Una funzione ϕ è una misura di complessità se restituisce un numero naturale avendo come ingressi una funzione ψ e x parametro di ψ. Inoltre soddisfa entrambi i seguenti assioni:

> $\phi(\psi, x)$ è definita (converge) $\Leftrightarrow$ $\psi(x)$ è definita (converge)
> Questo assioma ci dice che ϕ misura la complessità del calcolo di $\psi(x)$.

> $\forall\, \psi, x, k$ è decidibile se $\phi(\psi, x) = k$
> Questo assioma assicura che si può davvero misurare la complessità del calcolo di $\psi(x)$.

Quindi se ϕ misurasse il numero di passi, riotterremmo la definizione di complessità in tempo, mentre se misurasse il numero di celle riotterremmo quella in spazio.

2.4.7 Tesi di Cook-Karp

- $\mathscr{P}$ sono i problemi **trattabili**
- $\mathscr{NP}$ sono i problemi **intrattabili**

Nel senso che avendo un algoritmo che funziona in tempo polinomiale ($\mathscr{P}$), allora il problema può essere risolto "efficientemente", altrimenti se il problema sta in $\mathscr{NP}$ allora è troppo difficile per essere gestito da un programma.

Sono **robuste**, cioè resistono al cambio di modello. Ponendo M, M' **modelli di calcolo** (quindi MdT, `while`-calcolabili...), allora posso passare da $M \longrightarrow M'$ mediante un polinomio.

Inoltre $\mathscr{P}$ e $\mathscr{NP}$ sono chiusi rispetto alle classiche operazioni ma anche alle riduzioni in sottoclassi dei polinomi. Significa che se riduco il problema $I \leq_F J$, con $F \subseteq$ polinomi e $I \in \mathscr{C} \Rightarrow f(I) = J \in \mathscr{C}$

Ci sono algoritmi esponenziali al caso pessimo, ma efficientissimi nei casi più comuni: ad esempio l'algoritmo del simplesso, la paginazione, l'inferenza dei tipi in ML...
Ma non sempre posso sapere qual è il caso medio di un problema.

2.4.8 Riduzione efficiente

I si riduce efficientemente a $J \Leftrightarrow (x \in I \Leftrightarrow f(x) \in J$ con $f \in$ *logspace*) e si può scrivere $I \leq J$
Otteniamo che $\leq$ classificano *logspace* e una qualunque classe $\mathscr{E} \in \{\mathscr{P}, \mathscr{NP}, EXP, PSPACE, NPSPACE\}$
Se prendo una $\mathscr{D} \subseteq \mathscr{E}$, allora $\leq$ classificano $\mathscr{D}$ ed $\mathscr{E}$ (quindi ad esempio $\mathscr{P}$ ed $\mathscr{NP}$)
Dato che *logspace* $\leq \mathscr{P} \Rightarrow \leq_{\mathscr{P}}$ classificano $\mathscr{D}$ ed $\mathscr{E}$. Lo si dimostra così, tramite le proprietà della classificazione di una famiglia di riduzioni:

1. $id \in$ *logspace*, banalmente

2. $f, g \in$ *logspace* $\Rightarrow f(g) \in$ *logspace*
   $f \in$ *logspace* $\Rightarrow \exists\, M$ che opera in *logspace*, analogamente per la g avrò M'. La nuova macchina può considerare l'ultimo nastro di M come nastro di lavoro di M' e calcolare così la composizione delle due. Questa nuova macchina sta in *logspace*? Se prendo ad esempio $f = id$ ho input e output lungo n, e la somma dei nastri di lavoro è n, quindi lineare e non logaritmica. Allora si fa partire M' in modo da richiedere a M i dati di lavoro e usarli all'occorrenza sostituendoli sempre su una sola casella. Otteniamo così il *logspace*

3. $I \leq J$, $J \in \mathscr{D} \Rightarrow I \in \mathscr{D}$
   Se I si riduce per un logaritmo a J, e J lo risolvo in tempo logaritmico allora risolvo in tempo logaritmico anche I

4. $I \leq J$, $J \in \mathscr{E} \Rightarrow I \in \mathscr{E}$
   Analogo al punto precedente.

Quindi $\leq_{logspace}$ **sono riduzioni efficienti** e **classificano** $\mathscr{P}$ ed $\mathscr{NP}$
Una delle conseguenze è che se trovo un problema completo dentro $\mathscr{P}$ (cioè che tutti i problemi di $\mathscr{P}$ si riducono a lui e lui $\in \mathscr{P}$) allora in qualche modo ho rappresentato la difficoltà massima e $\mathscr{P}$ non ha problemi più difficili perché tutti i problemi si riducono a lui.
Poiché riesco a trovare soluzioni tra problemi in $\mathscr{P}$, questo vuol dire che le soluzioni di tutti i problemi all'interno di una classe hanno una struttura matematicamente analoga tra loro.

2.5 Richiami di logica

Ponendo $x_1, \ldots, x_n \in X$ **variabili** e la **grammatica**

$$B \longrightarrow tt \mid ff \mid x \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2$$

con tt, ff, x e $\neg x$ chiamati **letterali** $l_1, \ldots$.
Diciamo che $V : X' \longrightarrow \{T, F\}$ con $X' \subset X$ è un **assegnamento booleano**.
B è **chiusa** se **non ha variabili**. Un assegnamento booleano è **buono** se **lega tutte le variabili di B**.

Soddisfacibilità Cioè quando un assegnamento booleano rende vera un'espressione booleana.

$V \vDash tt$ banalmente

$V \vDash x$ quando $V(x) = tt$

$V \vDash \neg B$ quando $V \nvDash B$

$V \vDash B_1 \vee B_2$ quando $V \vDash B_1$ oppure $V \vDash B_2$

$V \vDash B_1 \wedge B_2$ quando $V \vDash B_1$ e $V \vDash B_2$

Forma normale B è in **forma normale congiuntiva** $\Leftrightarrow B = \bigwedge_{i=1}^{n} C_i$ e $C_i = \bigvee_{j=1}^{m} l_j$ con l_j letterali

B è in **forma normale disgiuntiva** $\Leftrightarrow B = \bigvee_{i=1}^{n} C_i$ e $C_i = \bigwedge_{j=1}^{m} l_j$ con l_j letterali

Inoltre $\forall\, B\ \exists\, B'$ in forma normale (una delle due) $\mid (V \vDash B \Leftrightarrow V \vDash B')$.
Questo si ottiene avendo, da B con $O(n)$ simboli, un B' con $O(2^n)$ simboli.

Considereremo espressioni solo in forma normale congiuntiva.

2.6 Alcuni problemi

2.6.1 Problema SAT

Anche chiamato **problema della soddisfacibilità**. La formulazione è semplice:

$$\text{Dato } B, \text{ allora } \exists\, V \mid V \vDash B \ ?$$

SAT $\in \mathscr{NP}$. Come faccio a dirlo? Dovrei costruire una macchina non deterministica che verifica questa cosa, cioè che dice se l'assegnamento esiste o meno. Come si fa?
Cominciando dallo stato iniziale. Supponendo n variabili, allora ho n scelte all'inizio, per ognuna delle quale ho due scelte: assegnare tt o assegnare ff. Lo faccio per la prima, due scelte, poi per la seconda... facendo così tutti i possibili assegnamenti. **Un cammino è un assegnamento**.
Vedremo invece che la verifica se un assegnamento soddisfa possiamo farla dinamicamente, quindi è **polinomiale** perché è nel numero delle variabili.

2.6.2 Problema HAM

Problema del cammino hamiltoniano. Dato un grafo orientato $G = (N, A)$, $\exists$? cammino che tocca tutti i nodi una ed una sola volta?
Vediamo che HAM $\leq$ SAT. Devo definire $f \in$ *logspace* $\mid$ (G ha un cammino hamiltoniano $\Leftrightarrow f(G) = B$ è tale che $\exists\, V \vDash B$)
Vedremo una riduzione, ma non è la migliore.

Supponiamo che $G = (N, A)$ ha n nodi (cioè l'insieme $\{1, 2, \ldots, n\}$), allora B ha n^2 variabili $x_{i,j}$: ciascuna rappresenta che nell'i-esimo posto di un cammino appare il nodo j. Posso rappresentarle anche come (i, j) con $i, j \in [1, n]$.
Definiamo una permutazione $\Pi : [1, n] \to [1, n] \mid (\Pi(i), \Pi(i+1)) \in A$, dai nodi ai nodi. In realtà nella nostra permutazione, i primi sono i nodi mentre i secondi sono le posizioni che occupano nel cammino. con $\Pi(i)$ rappresento il nodo in posizione i.
Π deve essere davvero una funzione:

1. Un nodo non può essere mappato due volte. Quindi: $\neg(x_{ij} \wedge x_{kj})$ con $i \neq k$. Questo però non è un congiunto, quindi applico DeMorgan
   $\Longrightarrow \neg x_{ij} \vee \neg x_{kj}$ con $i \neq k$

2. La funzione deve essere definita ovunque (totale), quindi ogni nodo deve apparire nel cammino.
   $\Longrightarrow x_{1i} \vee x_{2i} \vee \ldots \vee x_{ni}$

   Ho definito che la funzione è totale, quindi ora devo definire la sua surgettività.

3. Ogni posizione deve ricevere un nodo, sennò rimane un pezzo "vuoto".
   $\Longrightarrow x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}$

4. Non posso avere due nodi nella solita posizione. Quindi: $\neg(x_{ij} \wedge x_{ik})$ con $j \neq k$. Applico DeMorgan.
   $\Longrightarrow \neg x_{ij} \vee \neg x_{ik}$ con $j \neq k$

   Ho ben definito la permutazione Π, quindi ora se riesco a prendere tutti i disgiunti, metterli in and e trovo un assegnamento booleano che li rende veri ho trovato un cammino hamiltoniano. Manca da definire $(\Pi(i), \Pi(i+1)) \in A$

5. Se $(i, j) \notin A \Longrightarrow \neg x_{k,i} \vee \neg x_{k+1,j}$

Adesso devo dimostrare che se c'è un cammino hamiltoniano allora esiste un assegnamento che soddisfa. Dimostro i due versi dell'implicazione separatamente:

$\Leftarrow$ $\forall\, j\ \exists!\, i \mid V(x_{ij}) = tt$, così definiamo la valutazione. Cioè per ogni posizione del cammino i c'è un solo nodo j, quindi solo un x_{ij} è vera per ogni i.
Nello stesso modo $\forall\, i\ \exists!\, j \mid V(x_{ij}) = tt$, cioè ogni nodo j è in una sola posizione del cammino, quindi una sola i per cui x_{ij} è vera.
Grazie anche alla 5, che garantisce che ci sia il cammino, ho definito la V

$\Rightarrow$ Ho un cammino, composto da $(\Pi(1), \Pi(2), \ldots, \Pi(n))$, quindi definisco la $V(x_{ij}) = \begin{cases} tt & \text{se } \Pi(j) = i \\ ff & \text{se } \Pi(j) \neq i \end{cases}$

Finito. Ho definito una permutazione che rispetta il fatto che gli archi siano ben rappresentati.
Manca di dimostrare che $f \in$ *logspace*.

Dobbiamo costruire una macchina di tipo I/O, così sappiamo misurare lo spazio necessario (somma dello spazio usato dai nastri di lavoro). Naturalmente l'alfabeto della nostra macchina conterrà: simboli di verità, connettivi logici, parentesi, e due caratteri 0 e 1. Perché gli indici delle variabili sono rappresentati in binario.
L'input è la successione degli archi. La macchina scrive in binario il numero di nodi n sul nastro di lavoro. Poi avrò tre ulteriori nastri di lavoro, poiché nelle formule 1—5 compaiono 3 indici: un nastro per i, uno per j e uno per k.
Per scorrere le clausole farà scorrere da 1 a n gli indici sui nastri di lavoro. Quando ha scritto le quattro formule 1—4, comincia a scrivere sull'output l'ultima formula, la 5. Scorrerà il nastro input anche questa volta per verificare la clausola, finendo ad un certo punto.
Perché è in *logspace*? $\sum_{i=2}^{5} |w_i|$, ma i arriva fino ad n che rappresentato in binario richiede $\log n$. Quindi abbiamo 5 nastri di $\log n$, quindi $\sum_{i=2}^{5} |w_i| = 5 \log n$ cioè $f \in SPACE(5 \cdot \log n)$

Da un **problema nel mondo dei grafi** ho ottenuto **la sua soluzione mediante la soluzione di un problema di deduzione logica**. Abbiamo **ricondotto un problema sui grafi ad un problema di logica**.
Diventa esplicita la **struttura comune** fra i due problemi.

2.6.3 Problema CRICCA

Un problema risolto positivamente se in un certo grafo non orientato esiste un sottoinsieme dei nodi detto **cricca** tale per cui, per ogni coppia di nodi all'interno della cricca c'è un arco che li congiunge.
Formalmente: dato $G = (N, A)$ $\exists?\, C \subseteq N \mid \forall\, i, j \in C\ (i, j) \in A$.
C, cioè la cricca, è di ordine k se contiene k nodi.
Vediamo come SAT $\leq$ CRICCA, cioè **riconduciamo un problema di logica ad uno sui grafi**.

$\exists\, V \vDash B = \bigwedge_{i=1}^{n} C_i \Leftrightarrow f(B) = (N, A)$ ha n-cricca.
Otteniamo che

N è l'insieme delle occorrenze dei letterali l in B

$A = \{(i, j) \mid i \in C_k \Rightarrow (j \notin C_k \wedge i \neq \neg j)\}$ cioè metto un arco tra due nodi solo se non sono all'interno dello stesso letterale né se sono uno il negato dell'altro.

Da questo ottengo che se c'è una cricca c'è un assegnamento booleano, viceversa se c'è un assegnamento booleano c'è una cricca, quindi è una riduzione.
Appartiene a *logspace* perché posso usare lo stesso meccanismo della rappresentazione binaria e mantenendo gli indici sui nastri di lavoro.

Funzioni Booleane

$$V \vDash B \Leftrightarrow f(V(x_1), \ldots, V(x_n)) = tt \quad f : X \to \{0,1\}$$

f si **costruisce mediante un circuito booleano**, cioè un grafo $G = (N, A)$ **aciclico**, con

$i \in N$ nodi
Sorta $s(i) = \{tt, ff, \neg, \wedge, \vee\} \bigcup X$ (X variabili)
Ingressi le porte con $s(i) \subset \{tt, ff\} \bigcup X$
Uscita i la porta più in alto (il massimo del circuito quando viene ordinato con ordinamento parziale)
Se $s(i) = \neg \Rightarrow i$ ha 1 ingresso ed 1 uscita
Se $s(i) \subset \{\wedge, \vee\} \Rightarrow i$ ha 2 ingressi e 1 uscita
Per calcolare l'uscita ho bisogno di una funzione di valutazione che assegni i valori di verità agli ingressi, definiamone la semantica

$[i]_V = tt$ se $s(i) = tt$
$[i]_V = f$ se $s(i) = ff$
$[i]_V = V(x)$ se $s(i) = x$
$[i]_V = not([j]_V)$ se $(j,i) \in A$ e $s(i) = \neg$
$[i]_V = [j]_V$ *and* $[h]_B$ se $(j,i),(h,i) \in A$ e $s(i) = \wedge$
$[i]_V = [j]_V$ *or* $[h]_B$ se $(j,i),(h,i) \in A$ e $s(i) = \vee$

$(i,j) \in A$

Esempio $(x \vee (x \wedge y)) \vee ((x \wedge y) \wedge \neg(y \vee z))$

2.6.4 Circuit SAT

$$\exists\, V \mid [C]_V = tt?$$

con $[C]_V = [n]_V$ con n porta di uscita. Questo problema $\in \mathscr{NP}$
Circuit Value: se C non ha variabili, $[C]_\emptyset = tt$? Questo è il problema della soddisfacibilità, per verificare se dato un assegnamento il circuito è soddisfatto o meno.
Circuit Value $\in \mathscr{P}$, perché tengo sul nastro input la rappresentazione del grafo, cioè l'insieme delle coppie che formano gli archi e la sorta di ogni nodo (porta). Sui nastri di lavoro tengo i valori dei vari livelli di porte, ma non ho ripetizioni quindi è polinomiale: non bisogna farsi confondere dal fatto che il numero totale delle porte può essere esponenziale rispetto alla profondità del circuito (cioè al livello della porta d'uscita), perché la taglia del circuito è data dal numero n di porte.

Notiamo che **Circuit Value** $\leq$ **Circuit SAT**, perché SAT è la versione più generale di Value.
Value è la versione di SAT in cui non ho bisogno della funzione di assegnamento.

Più interessante è scoprire che Circuit SAT $\leq_{logspace}$ SAT. Intuitivamente è vero, costruiamo la riduzione.
Significa che $\forall$ circuito con variabili $\in X$, dobbiamo trovare una $f \in logspace \mid [C]_V = tt \Leftrightarrow \exists\, V' \supseteq V \quad V' \vDash f(C)$
$V(x) = tt \Rightarrow V'(x) = tt$ e $f(C) = \bigwedge B_k$
Le variabili di $f(C)$ includono tutte le variabili di C, cioè X, unito ad una variabile per ogni porta,
cioè $X \cup \{x_i = i \mid i \in N\}$

Se g è la porta di uscita $\Rightarrow$ genera un congiunto g variabile

Se $s(i) = tt$ o $ff \Rightarrow$ genera un congiunto i nel primo caso, $\neg i$ nel secondo caso

Se $s(i) = x \Rightarrow (i \Leftrightarrow x)$ cioè i è vero se e soltanto se x è vero, cioè $(i \Rightarrow x) \wedge (x \Rightarrow i)$ cioè $(\neg i \vee x) \wedge (\neg x \vee i)$

Se $s(i) = \wedge \Rightarrow (i$ è vero$\Leftrightarrow h \wedge k)$ con $(h,i),(k,i) \in A$, espandendola diventa $\neg(i \vee h) \wedge (\neg i \vee k) \wedge (\neg h \vee \neg k \vee i)$

Se $s(i) = \vee \Rightarrow (i$ è vero$\Leftrightarrow h \vee k)$ con $(h,i),(k,i) \in A$, espandendola diventa $(\neg i \vee h \vee k) \wedge (\neg h \vee i) \wedge (\neg k \vee i)$

Usando $x \wedge ff$ come esempio, mettendo come nodi $h = x$, $g = \wedge$ e $k = ff$, ottengo la formula

$$g \wedge \neg k \wedge (\neg h \vee x) \wedge (h \wedge \neg x) \wedge (\neg g \vee h) \wedge (\neg g \vee k) \wedge (\neg h \vee \neg k \vee g)$$

Vediamo che $[g] = ff \ \forall V$

$g\wedge \longrightarrow V(g) = tt$

$\neg k\wedge \longrightarrow V(k) = ff$

$(\neg h \vee x)\wedge$

$(h \wedge \neg x)\wedge$

$(\neg g \vee h)\wedge$

$(\neg g \vee k)\wedge \longrightarrow V(k) = tt$ ottenendo un assurdo

$(\neg h \vee \neg k \vee g)$

2.7 Tabella di computazione

Prendiamo un problema $I \in \mathscr{P} \Leftrightarrow \exists\, M \mid \forall\, x \in IM(x) \longrightarrow_t$ (si/no, w) con $t \leq |x|^k$
Ho la configurazione iniziale $\trianglerighteq a_1 \dots a_n$, copo un certo passo ho $\triangleright \underline{a_1} \dots a_n$ e così via.

Posso immaginare di mettere il tutto in una matrice:

$\trianglerighteq$	a_1	$\dots$	a_n
$\triangleright$	$\underline{a_1}$	$\dots$	a_n
$\triangleright$			
$\triangleright$			

Ad ogni riga i ho il passo di computazione e la configurazione al passo i: ho rappresentato l'intera computazione in una matrice. Possiamo dare a questa **tabella di computazione** un **formato standard**.

Tabella di computazione Una matrice quadrata $T[i, j]$ con $1 \leq i, j \leq |x|^k$ se $M(x)$ termina in $|x|^k - 2$ passi.
Condizioni:

1. La macchina termina in meno di $|x|^k - 2$ passi, come detto
2. Presa la riga i, essa comincia con il respingente e termina con tutti i caratteri bianchi.
   Tutte le caselle non significative di una riga, quindi, sono riempite con #
   Siccoma la lunghezza della riga è $|x|^k$, ma la macchina termina in $|x|^k - 2$ passi, non avrò mai il cursore in ultima posizione perché il tempo limita lo spazio.
3. Supponiamo il cursore in una posizione, posso codificare lo stato nell'alfabeto
   L'alfabeto contiene $\sigma_q \in (\Sigma \times Q \times \{h\})$ che registra che nella configurazione i-esima il cursore si trova nella posizione j, si legge σ e lo stato è q. Basta prendere $\Sigma \times Q$ nuovi simboli.
4. All'inizio il cursore è sul primo carattere subito dopo il respingente. In più i passaggi sul respingente "indietro-avanti", cioè i due passi obbligati successivi di andare sul respingente e tornare a destra, vengono condensati in un singolo passo. Così facendo, il cursore non si troverà mai sul respingente. C'è un'eccezione, nel caso seguente
5. Quando $T[i, j] =$ si/no, allora sposta il cursore fino alla seconda colonna (con massimo $O(|x|^k)$ passi), introducendo uno stato ausiliario di finto arresto. Lo stato di accettazione, se raggiunto, è quindi sempre in $T(l, 2)$ per qualche $l \leq |x|^k$.
   Si ammette che il cursore passi sopra il simbolo $\triangleright$ quando lo stato sia q_{SI} (abbreviazione di $\sigma_{q_{SI}}$), con il vincolo che non debba mai toccare il $\triangleright$ più a sinistra (che rappresenta l'inizio del nastro).
6. Se σ_{SI} oppure σ_{NO} appaiono sulla riga $p < |x|^k$ e nella seconda colonna, allora tutte le righe di indice q con $p \leq q \leq |x|^k$ sono uguali alla p-esima

Abbiamo l'ovvia condizione di terminazione con successo: M accetta $x \Leftrightarrow \exists\, i \mid T(i, 2) = \sigma_{SI} (= T(|x|^k, 2))$
Su una tabella di computazione T di una macchina di Turing M che decide I in $|x|^k$, con x lungo n caratteri, ho una serie di fatti:

$T(1, 2)$ contiene lo stato iniziale e il primo carattere di x x_q^1
Inoltre $\forall\, j, 2 \leq j \leq |x| + 1$, la casella $T(1, j)$ contiene il $j - 1$-esimo simbolo di x
Infine, $\forall\, j, |x| + 2 \leq j \leq |x|^k$, la casella $T(1, j)$ contiene #

$\forall\, i, T(i,1) = \triangleright$

$\forall\, i, T(i,|x|^k) = \#$ (si impiega meno spazio che tempo!)

$T(i,j)$ = come determinarlo? (Osservazione tratta dall'esempio successivo)
L'elemento $T(i,j)$ dipende dall'elemento $T(i-1, j \pm 1)$ oppure da $T(i-1,j)$. Nell'esempio successivo, $T(4,4)$ dipende da $T(3,4)$, $T(8,3)$ da $T(7,4)$ e $T(11,5)$ da $T(10,4)$
Ciascuna cella, quindi, **dipende solo e solamente dalle tre precedenti**. Questo perché ad ogni passo la macchina si sposta di un posto solo.
Quindi $T(i,j)$ dipende solo da $T(i-1,j-1)$, $T(i-1,j)$, $T(i-1,j+1)$ e dalla δ.

Una tabella di computazione, quadrata, ha $|x|^k$ righe/colonne. Siccome si ferma in $|x|^k - 2$ passi, abbiamo la sicurezza che la prima colonna è composta da tutti $\triangleright$ e l'ultima è composta da tutti # perché non si può usare più spazio che tempo.
La prima riga, oltre il respingente, avrà i caratteri $x_1 x_2 \dots x^n$ e dopo tutti caratteri bianchi #. Le righe rappresentano i passi di computazione, fino a che ad un certo punto (in una certa riga = un certo passo di computazione) in un certo carattere si troverà uno stato di accettazione σ_h. Questo stato di accettazione viene portato, nei passi successivi, in seconda posizione (subito dopo il respingente) e da quella riga in poi sono tutte uguali, per cui l'ultima riga sarà $\triangleright\, \sigma'_h \dots \#$

Esempio MdT che verifica se una stringa è palindroma. Esempio: la stringa `abba`, 16 passi per verificare, quindi 18 righe e colonne

	1	2	3	4	5	6	7	…	17	18
1	$\triangleright$	a_{q_0}	b	b	a	#	#	…	#	#
2	$\triangleright$	$\triangleright$	b_{q_A}	b	a	#	#	…	#	#
3	$\triangleright$	$\triangleright$	b	b_{q_A}	a	#	#	…	#	#
4	$\triangleright$	$\triangleright$	b	b	a_{q_A}	#	#	…	#	#
5	$\triangleright$	$\triangleright$	b	b	a	$\#_{q_A}$	#	…	#	#
6	$\triangleright$	$\triangleright$	b	b	$a_{q'_A}$	#	#	…	#	#
7	$\triangleright$	$\triangleright$	b	b_{q_1}	#	#	#	…	#	#
8	$\triangleright$	$\triangleright$	b_{q_1}	b	#	#	#	…	#	#
9	$\triangleright$	$\triangleright$	b_{q_0}	b	#	#	#	…	#	#
10	$\triangleright$	$\triangleright$	$\triangleright$	b_{q_B}	#	#	#	…	#	#
11	$\triangleright$	$\triangleright$	$\triangleright$	b	$\#_{q_B}$	#	#	…	#	#
12	$\triangleright$	$\triangleright$	$\triangleright$	$b_{q'_B}$	#	#	#	…	#	#
13	$\triangleright$	$\triangleright$	$\triangleright$	$\#_{q_0}$	#	#	#	…	#	#
14	$\triangleright$	$\triangleright$	$\triangleright_{SI}$	#	#	#	#	…	#	#
15	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	#	#	#	#	…	#	#
16	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	#	#	#	#	…	#	#
17	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	#	#	#	#	…	#	#
18	$\triangleright$	$\triangleright_{SI}$	$\triangleright$	#	#	#	#	…	#	#

2.7.1 Circuit Value è $\mathscr{P}$-completo

$\forall\, I \in \mathscr{P}$, $x \in I \Leftrightarrow [f(x)]_\emptyset = tt$ con $f \in$ *logspace*
$I \in \mathscr{P} \Rightarrow \exists\, M$ che decide x in $|x|^k = n^k \Rightarrow M(x) \to^t$ (SI, w) con $t \le n^k - 2$
Allora **costruiamo la tabella di computazione**, aggiungendo ad M $\Sigma \times Q \cup \{h\}$ nuovi simboli. Chiamo Σ' il nuovo alfabeto.
Ciascun simbolo $\rho \in \Sigma'$ viene mappato in una stringa di bit, successioni di 0/1 $(s_1, \dots, s_m) \in \{tt, ff\}^m$, con $m = \lceil \log_2(\#\Sigma') \rceil$ cioè numero di bit necessari per rappresentare i $\#\Sigma'$ simboli contenuti in Σ'.
Posso codificare il respingente e il carattere bianco come preferisco, ad esempio $\triangleright = tt \dots tt$ e $\# = ff \dots ff$
Ogni elemento della tabella quindi diventa una sequenza di m bit: $T(i,j) = s_{i,j,1}, s_{i,j,2}, \dots, s_{i,j,m}$, cioè m simboli ciascuno che codifica anche la riga i e la colonna j.

$$T(i,j) = f\begin{pmatrix} s_{i-1,j-1,2}, & s_{i-1,j-1,2}, & \dots, & s_{i-1,j-1,m}, \\ s_{i-1,j,2}, & s_{i-1,j,2}, & \dots, & s_{i-1,j,m}, \\ s_{i-1,j+1,2}, & s_{i-1,j+1,2}, & \dots, & s_{i-1,j+1,m} \end{pmatrix}$$ perché dipende solamente da $\begin{matrix} T(i-1,j-1) \\ T(i-1,j) \\ T(i-1,j+1) \end{matrix}$

Abbiamo la tabella di computazione che decide x, accettandolo o rifiutandolo. Prendo questa tabella e la codifico in binario, ottenendo un'altra tabella in cui invece dei simboli ho 0/1. **Ottengo un circuito**.
La f sopra indicata è una funzione booleana, quindi $\exists$ un circuito $\overline{C}$ che la realizza. Questo circuito avrà m uscite e $3 \cdot m$ ingressi. $\overline{C}$ dipende dall'input x? No, perché dipende soltanto dalla δ e dai $3m$ valori in ingresso. Quindi il suo

costo è **costante**, indipendente da x.
Ogni gruppo di m input di $\overline{C}$ proviene da un circuito $\overline{C}$ a sua volta, perché è il gruppo di bit di una cella, e così via...
Riempio la tabella binaria, con $\overline{C}_\triangleright$ e $\overline{C}_\#$ circuiti costanti che fanno le veci del respingente e del carattere bianco.
Al centro avrò le varie porte del circuito, che si alimentano l'un l'altra incastrando le une con le entrate delle altre.
Ogni cella riceve input dalle tre celle sottostanti (il segnale fluisce verso l'altro nel grafo dei circuiti). La prima riga, invece, riceve input da x.

$$\begin{array}{c|ccc|c} \overline{C}_\triangleright & & \dots & & \overline{C}_\# \\ \overline{C}_\triangleright & & \overline{C} & & \overline{C}_\# \\ \overline{C}_\triangleright & \overline{C} & \overline{C} & \overline{C} & \overline{C}_\# \\ \overline{C}_\triangleright & & \dots & & \overline{C}_\# \\ \overline{C}_\triangleright & & & & \overline{C}_\# \\ \overline{C}_\triangleright & \overline{C}_{x_1} & \dots & \overline{C}_{x_n} & \overline{C}_\# \end{array}$$

Verifichiamo che f stia in *logspace*. Per verificare questo è necessario considerare l'indice di riga, colonna e la dimensione della matrice. Gli indici li rappresento in binario, con $\lceil \log_2(|x|) \rceil$ bit.
Con gli indici scorro la matrice avanti e indietro (due for annidati), 3 nastri di lavoro.
Quindi sto in *logspace*.
Abbiamo visto come realizzare un'intuizione avuta all'inizio: la MdT ha un approccio hardware alla computabilità (e di conseguenza alla complessità), ma solo ora abbiamo visto un processo diretto di trasformazione di una computazione in un circuito.

2.7.2 Monotone Circuit Value

Circuito Monotono Circuito dove non compare il NOT ($\neg$). Mentre tutti gli altri operatori logici sono monotoni, nel senso che mantengono l'informazione, il $\neg$ la butta via. Ma d'altra parte, senza il $\neg$ non si possono esprimere tutte le formule.
Il problema di calcolare il valore in un circuito monotono è un problema $in\mathscr{P}$, perché è un caso particolare del Circuit Value. C'è però una riduzione Circuit Valute $\leq$ Monotone Circuit Value.

Nel grafo del circuito, riscritto dall'alto verso il basso e da sinistra verso destra, lo riscrivo mantenendo però un "cambio di stato" quando incontro un $\neg$. I nodi successivi saranno quindi invertiti secondo le leggi di DeMorgan, ad esempio $\vee$ diventa $\wedge$.

Grammatica $G = (N, \Sigma, P, S)$

N **alfabeto**, insieme di simboli non terminali

Σ insieme di **simboli terminali**

P insieme di **produzioni**, $P = \{A \to \alpha \mid A \in N,\ \alpha \in (N \cup \Sigma)^+\}$, cioè α è una stringa non vuota.

$S \in N$ **simbolo distinto**

Una certa stringa si trasforma secondo la G, cioè $\gamma A \beta \Rightarrow_G \gamma\alpha\beta$ se $A \to \alpha \in P$ senza nessun vincolo (**grammatica libera da contesto**)
Il **linguaggio generato dalla grammatica** è l'insieme $L(G) = \{w \in \Sigma^+ \mid S \Rightarrow^* w\}$ cioè tutte le stringhe w generate da S tramite la grammatica G in un numero qualsiasi di passi ($\Rightarrow^*$)

Esempio $S \to ()|(S)|SS$ genera il linguaggio delle parentesi bilanciate, tipo $S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()())$

Verificare Data una grammatica libera G, $L(G) = \emptyset$? Questo problema viene spesso considerato il capostipite dei problemi $\mathscr{P}$-completi.
Si costruisce una tabella simile alla tabella di computazione, in ogni riga ci metto cioè che la grammatica produce a partire dalla precedente fino alla fine. Passare da una riga alla successiva è determinato solamente dalla G.

2.8 SAT è $\mathscr{NP}$-completo

Passiamo attraverso Circuit SAT, dimostrando che Circuit SAT è $\mathscr{NP}$-completo. Devo prendere un problema $I \in \mathscr{NP}$ e costruire $f \in logspace \mid x \in I \Leftrightarrow f(x) = C$ ed $\exists\, V \mid [C]_V = tt$

$x \in I$ vuol dire che $\exists$ una macchina non deterministica N ed $\exists$ una computazione $N(x) \longrightarrow^t$ (si, w) con $t \leq |x|^k$
Da N posso ottenere una N' *equivalente* (cioè se N termina con si anche N' termina con si, analogo per il no) che calcola esattamente la stessa cosa, il cui grado di diramazione (grado di non determinismo) = 2
Come farlo? Se la macchina in uno stato q che va in q' posso sdoppiare lo stato di arrivo q' in q'_1, q'_2. Se ho $n > 2$ scelte, prendo la prima scelta e considero tutte le altre come se fossero un singolo nuovo stato. In questo nuovo stato, prendo il primo degli $n-1$ stati rimanenti e considero i successivi come un singolo nuovo stato e così via.
Posso farlo in tempo polinomiale (basta ricopiare gli stati) e il rallentamento della nuova macchina è dovuto al numero di nuovi stati ma sempre polinomiale.

Se la prima scelta la codifico con ff, la seconda scelta la codifico con tt, allora la computazione $N(x) \longrightarrow^t$ (si, w) è rappresentata da una successione di bit $b_0 b_1 \dots b_{|x|^n-1}$. T_{ij} non dipenderà solamente dalle tre celle superiori della tabella di computazione, ma anche dal bit b_i
Costruisco il circuito $\overline{C}$ con $3m$ input e m output, ma stavolta con un bit ulteriore in input. Ho ottenuto $\overline{C_n}$ che ci serve per riempire la tabella, a costo costante che dipende solo dalla Δ' di N'.
Riempiamo il circuito con $|x|^k$ circuiti per $\triangleright$, $|x|$ circuiti per $\#$, e $|x|^{k-2} \times |x|^k$ copie di $\overline{C_n}$ opportunamente connesse.
Il **teorema di Cook è dimostrato**.

Altri problemi $\mathscr{NP}$-completi HAM, CRICCA, commesso viaggiatore, programmazione lineare, e altri e altri ancora...

Capitolo 3

Esercizi

3.1 Calcolabilità

Esercizio 1 Data un'enumerazione effettiva, esiste una funzione calcolabile totale $t(i,n)$ che maggiora il tempo di calcolo di $M_i(n)$?

Svolgimento Tale funzione non esiste.
Si prenda $t(i,n) = \begin{cases} k & M_i(n)\downarrow \textit{ in meno di } k \textit{ passi} \\ 0 & \textit{altrimenti} \end{cases}$ e si supponga per assurdo $t(i,n)$ calcolabile totale.
Poniamo T_i la misura *esatta* del tempo di calcolo di M_i. $\forall n$, la funzione test $T_i(n) \leq^? t(i,n)$ è calcolabile totale, basta far girare $M_i(n)$ al più $t(i,n)$ passi: se l'algoritmo si è arrestato allora la risposta è positiva, altrimenti negativa.
Si definisca $\psi(x) = \begin{cases} \phi_x(x)+1 & T_x(x) \leq t(x,x) \\ 0 & \textit{altrimenti} \end{cases}$ e, per il solito ragionamento, si osservi che $\psi(x)$ è calcolabile totale.
Quindi, per C-T, $\exists\, i \mid \psi = \phi_i$, ma ottengo $\phi_i(i) = \psi(i) = \begin{cases} \phi_i(i)+1 & T_i(i) \leq t(i,i) \\ 0 & \textit{altrimenti} \end{cases}$ e poiché $\phi_i(i) \neq \phi_i(i)+1$ segue che $\phi_i(i) = 0$ e dunque $T_i(i) > t(i,i)$
Ciò contraddice l'ipotesi $T_i(i) \leq t(i,i)$, quindi $t(i,n)$ non è calcolabile totale.

Esercizio 2 Esiste una funzione calcolabile totale che determina se un dato programma M_i calcolato su uno specifico dato x all'interno di un calcolatore C con memoria finita è in ciclo?

Svolgimento Definiamo $h(i,x,C) = \begin{cases} 1 & M_i(x)\uparrow \textit{ sul calcolatore } C \\ 0 & \textit{altrimenti} \end{cases}$
h è totale, ma è calcolabile? Aver limitata la memoria di C ci permette di **stimare il numero massimo di configurazioni** di C, dunque di avere una procedura che in un numero finito di passi stabilisce se $M_i(x)$ è in ciclo o meno. Infatti, se si rappresenta C come una MdT possiamo indicare con $n = \#\Sigma_C$ il **numero di simboli** di C, $m-1 = \#Q_c$ il **numero di stati** di C e k il **numero di celle del nastro** di C. Questo implica che si possono avere al più n^k **stringhe diverse** sul nastro, su cui la testina può trovarsi in al più k **posizioni diverse** in al più m **stati diversi**. Quindi ho un **numero massimo di configurazioni** differenti che C può assumere pari a $l = k \cdot n^k \cdot m$.

Sia H la MdT a 4 nastri che all'inizio contengono:

sul primo nastro $\langle M_i \rangle$, la codifica della MdT M_i

sul secondo nastro $\langle x \rangle$, la codifica del dato x

sul terzo nastro la codifica dello stato iniziale di M_i

sul quarto stato la codifica in unario di l

H simula il comportamento di $M_i(x)$ lavorando come la MdTU, e in più decrementa di una tacca il quarto nastro ogni volta che simula la computazione di $M_i(x)$. Se sul terzo nastro compare lo stato di arresto di M_i e sul quarto nastro vi sono ancora tacche, allora H si ferma e restituisce $h(i,x,C) = 0$, altrimenti se sul quarto nastro non vi sono più tacche ed M_i non è nello stato di arresto allora H si ferma e restituisce $h(i,x,C) = 1$ poiché M_i è **destinata a tornare in una delle configurazioni già incontrate**.

Bisogna notare come aver definito i nastri delle MdT infiniti a destra renda possibile la definizione di H. Se imponessimo un limite di spazio, in particolare al quarto nastro di H, sarebbe possibile trovare un calcolatore C con l maggiore del limite, e dunque $h(i, x, C)$ diverrebbe non calcolabile.
In altre parole, se non avessimo il vincolo della memoria finita di C, la funzione $h(i, x)$ (a due posti invece di tre) non sarebbe calcolabile, infatti risulterebbe $h(i, i) = 1 - \chi_K(i)$ che sappiamo non essere calcolabile.

Esercizio 3 Si dimostri che un insieme finito A è ricorsivo se e solo se può essere generato in modo strettamente crescente, cioè esiste una funzione strettamente crescente g tale che $A = Imm(g)$

Svolgimento Se A è ricorsivo, allora la sua funzione caratteristica è ricorsiva, dunque possiamo definire g come

$$\begin{cases} g(0) & = \mu y[\chi_A(y) = 1] \\ g(n-1) & = \mu y[\chi_A(y) = 1 \wedge g(n) < y] \end{cases}$$

La funzione che verifica $<$ è primitiva ricorsiva, e la funzione g è strettamente crescente. Per dimostrare che $A = Imm(g)$, basta ordinare in modo crescente gli elementi di A, cioè riordinare $A = a_{ii}$ in modo che $a_{i_{n+1}}$ sia l'elemento di A immediatamente successivo a a_{i_n}, e osservare che $\forall\, n$ vale $a_{i_n} = g(n)$.
Cioè $A = \{a_{i_0}, a_{i_1}, \ldots, a_{i_n}, \ldots\} = \{g(0), g(1), \ldots, g(n), \ldots\}$

Se $A = Imm(g)$ con g calcolabile totale strettamente crescente, per dimostrare che A è ricorsivo bisogna fare vedere che χ_A è calcolabile totale. Si definisce la seguente funzione che è banalmente calcolabile totale (B è un insieme finito)

$$\phi_A(n) = \begin{cases} 1 & n \in B = \{g(0), \ldots, g(n)\} \subset A \\ 0 & \textit{altrimenti} \end{cases}$$

Verifico che $\phi_A = \chi_A$, cioè che la funzione appena definita è davvero la funzione caratteristica di A:

Se $n \in B$ allora $n \in A$ per definizione

Se $n \notin B$ allora $n < g(n) < g(n+1) < \ldots$ e quindi $n \notin A$

Esercizio 4 Si dimostri che ogni insieme ricorsivamente enumerabile non ricorsivo (quindi non vuoto) ha almeno un sottoinsieme ricorsivo infinito.

Svolgimento Sia f la funzione calcolabile totale che enumera l'insieme ricorsivamente enumerabile dato. A partire da f, si costruisca $\begin{cases} g(0) & = f(0) \\ g(n+1) & = f(\mu k[f(k) > g(n)]) \end{cases}$
L'immagine di g è un insieme finito e g è strettamente crescente. Per l'esercizio 3, quindi, la sua immagine è un insieme ricorsivo.

Esercizio 5 Si definisca $W_i = \{n \mid \phi_i(n) \downarrow\} = dom(\phi_i)$ e si considerino due funzioni calcolabili $\phi_i, \phi_j \mid \emptyset \neq W_i \subsetneq W_j$
Si discuta dell'esistenza di una funzione calcolabile totale $f \mid \forall\, x\ W_{f(x)} = \begin{cases} W_i & \textit{se } \phi_x(x) \uparrow, \textit{ cioè se } x \notin K \\ W_j & \textit{se } \phi_x(x) \downarrow, \textit{ cioè se } x \in K \end{cases}$

Svolgimento Al contrario di quanto sembra, una tale f esiste. Sappiamo che W_i è ricorsivamente enumerabile, dunque esiste una funzione calcolabile totale h_i tale che $W_i = Imm(h_i)$, e similmente esiste h_j tale che $W_j = Imm(h_j)$. Con la seguente procedura si generano gli infiniti elementi di $W_{f(x)}$ e, poiché è definita per ogni x, la f è totale:

```
n := 0;
W[f(x)] := empty;
while (phi[x](x) non converge in n passi) do
        w[f(x)] := W[f(x)] U h[i](n);
        n := n + 1;
n := 0;
while (true) do
        w[f(x)] := W[f(x)] U h[j](n);
        n := n + 1;
```

A parole: essendo $W_i \subsetneq W_j$, posso iniziare a costruire $W_{f(x)}$ generando alcuni elementi di W_i con la funzione h_i e inserendoli in $W_{f(x)}$. Non appena mi accorgo che la $\phi_x(x)$ termina, inizio ad emettere gli elementi di W_j (alcuni magari già calcolati, per l'inclusione) in $W_{f(x)}$ con la funzione h_j
Si noti che se W_i e W_j fossero disgiunti, f sarebbe la caratteristica di K, quindi non sarebbe calcolabile.

Esercizio 8 Si dimostri che CONST $= \{x \mid \phi_x$ totale e costante$\}$ non è ricorsivo

Svolgimento Basta ridurre K a CONST, dato che K non è ricorsivo.
Definisco $\psi(x,y) = \begin{cases} 1 & \text{se } \exists\, z > y \mid \phi_x(x)\downarrow \text{ in meno di } z \text{ passi} \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$ che è calcolabile parziale, essendo la funzione semi-caratteristica di K.
Allora, per il teorema s-m-n, esiste f calcolabile totale iniettiva tale che $\phi_{f(x)}(y) = \psi(x,y)$ (con il solito trucco di scegliere uno degli indici che calcolano la ψ per poi ingorarlo). Adesso:

$x \in K \Rightarrow \phi_{f(x)} = \psi(x,y) = \lambda y.1 \Rightarrow f(x) \in$ CONST

$x \notin K \Rightarrow \phi_{f(x)} = \lambda y.\textit{indefinito} \Rightarrow f(x) \in$ CONST

Quindi CONST non è ricorsivo, e nemmeno ricorsivamente numerabile.
In alternativa, si può verificare che CONST è un insieme di indici rappresentanti funzioni (i.i.r.f.) e che non è vuoto, di conseguenza non è ricorsivo.

Esercizio 9 Si dimostri che INF $= \{x \mid dom(\phi_x)$ è infinito$\} \leq_{rec}$ CONST $= \{x \mid \phi_x$ totale e costante$\}$

Svolgimento Definiamo $\psi(x,y) = \begin{cases} 1 & \text{se } \exists\, z > y \mid \phi_x(z)\downarrow \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$ che è calcolabile parziale. Per il teorema s-m-n, esiste f calcolabile totale iniettiva tale che $\phi_{f(x)}(y) = \psi(x,y)$. Adesso:

$x \in$ INF $\Rightarrow \phi_{f(x)} = \psi(x,y) = \lambda y.1 \Rightarrow f(x) \in$ CONST
Questo perché se $x \in$ INF, significa che qualsiasi $z > y$ io dia come parametro a $\phi_x(z)$ essa convergerà, perché $dom(\phi_x)$ è infinito. Poiché il risultato è sempre 1, allora $f(x) \in$ CONST

$x \notin$ INF $\Rightarrow \exists\, y.\phi_{f(x)}(y)$ è indefinito $\Rightarrow f(x) \notin$ CONST

Esercizio 10 Si dimostri che TOT $= \{x \mid dom(\phi_x) = \mathbb{N}\} \leq_{rec}$ INF $= \{x \mid dom(\phi_x)$ è infinito$\}$

Svolgimento Definiamo $\phi_{f(x)}(y) = \psi(x,y) = \begin{cases} 1 & \text{se } \forall z < y\ \phi_x(z)\downarrow \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$
Questa funzione, a cui è stato applicato s-m-n, è calcolabile parziale. Adesso

$x \in$ TOT $\Rightarrow \phi_{f(x)} = \psi(x,y) = \lambda y.1 \Rightarrow x \in$ INF
Questo perché se $x \in$ TOT allora ϕ_x ha dominio uguale a $\mathbb{N}$, quindi presa una y, $\phi_x(z)$ con $z < y$ convergerà sempre e questo vale $\forall\, y \in \mathbb{N}$, che è infinito, rendendo il dominio di $\psi(x,y) = \phi_{f(x)}(y)$ infinito.

$x \notin$ TOT $\Rightarrow \exists\, \overline{y}\, \forall\, y > \overline{y}$ ho $\phi_{f(x)}(y)$ indefinita $\Rightarrow f(x) \notin$ INF
Perché se ϕ_x non ha dominio pari a $\mathbb{N}$, allora posso trovare una $z < y$ per cui $\phi_x(z)\uparrow$. Avendo una situazione di indefinito, il dominio di $\phi_{f(x)}$ non può essere infinito.

Esercizio 11 Si dimostri che FIN, l'insieme degli indici delle funzioni calcolabili con dominio finito, non è ricorsivamente enumerabile.

Svolgimento Dimostriamo che $\overline{K} \leq_{rec}$ FIN, da cui la tesi.
Definiamo $\psi(x,y) = \begin{cases} 1 & x \in K \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$ che è calcolabile parziale, quindi possiamo applicare s-m-n e ottenere $\phi_{f(x)} = \psi(x,y)$. Si ha:

$x \in \overline{K} \Leftrightarrow x \notin K \Rightarrow \phi_{f(x)} = \lambda y.\textit{indefinita} \Rightarrow dom(\phi_{f(x)}) = \emptyset \Rightarrow f(x) \in$ FIN

$x \notin \overline{K} \Leftrightarrow x \in K \Rightarrow \phi_{f(x)} = \lambda y.1 \Rightarrow dom(\phi_{f(x)}) = \mathbb{N} \Rightarrow f(x) \notin$ FIN

Nota: è immediato vedere che $\emptyset \neq$ FIN $\neq \mathbb{N}$, da cui, per il lemma usato per dimostrare il teorema di Rice, segue che non è ricorsivo.

Esercizio 12 L'insieme $I = \{i \mid dom(\phi_i) = \{3\}\}$ è ricorsivo? Ricorsivamente enumerabile? E se $dom(\phi_i)$ fosse l'insieme dei numeri pari?

Svolgimento Banalmente $\emptyset \neq I \neq \mathbb{N}$ e inoltre I è un insieme di indici che rappresentano funzioni, pertanto I non è ricorsivo.
Inoltre $K \leq_{rec} I$, infatti si consideri $\phi_{f(x)}(y) = \psi(x,y) = \begin{cases} 1 & x \in K \wedge y = 3 \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$ che è calcolabile parziale.

$$x \in K \Rightarrow \phi_{f(x)}(y) = \begin{cases} 1 & y = 3 \\ \textit{indefinita} & \textit{altrimenti} \end{cases} \Rightarrow dom(\phi_{f(x)}) = \{3\} \Rightarrow f(x) \in I$$

$$x \notin K \Rightarrow \phi_{f(x)} = \lambda y.\textit{indefinito} \Rightarrow f(x) \notin I$$

Infine possiamo dimostrare che I non è nemmeno ricorsivamente enumerabile, perché $\overline{K} \leq_{rec} I$. Infatti $K \leq_{rec} \overline{I}$ attraverso la seguente funzione di riduzione che modifica leggermente quella sopra
$\phi_{g(x)}(y) = \psi(x,y) = \begin{cases} 1 & x \in K \vee y = 3 \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$

$$x \in K \Rightarrow \phi_{g(x)}(y) = 1 \Rightarrow dom(\phi_{g(x)}) = \mathbb{N} \Rightarrow g(x) \notin I$$

$$x \notin K \Rightarrow \phi_{g(x)}(y) = \begin{cases} 1 & y = 3 \\ \textit{indefinita} & \textit{altrimenti} \end{cases} \Rightarrow dom(\phi_{g(x)}) = \{3\} \Rightarrow g(x) \in I$$

Esercizio 13 Esistono due insiemi A e B non ricorsivi la cui intersezione è un insieme ricorsivo infinito?

Svolgimento Certo: si consideri un insieme ricorsivo infinito P, ad esempio $P = \{2n\}$
Si pongano $A = K \cup P$ e $B = \overline{K} \cup P \Rightarrow A \cap B = P$ è ricorsivo infinito.

Esercizio 14 La classe degli insiemi di indici che rappresentano funzioni forma un'algebra booleana?

Svolgimento Basta mostrare che:

$\emptyset$ è un i.i.r.f. (nessuna funzione)

I è i.i.r.f. $\Rightarrow \overline{I}$ è i.i.r.f.

I, J sono i.i.r.f. $\Rightarrow I \cap J$ è i.i.r.f.

Esercizio 15 Si dica se esiste una funzione calcolabile totale f tale che $\forall\, x.dom(\phi_{f(x)}) = \mathbb{N}$

Svolgimento Sia i uno degli indici della funzione $\lambda\, x.0$ e si ponga $f(x) = \lambda\, x.i$
Chiaramente $dom(\phi_{f(x)}) = dom(\phi_i) = \mathbb{N}$

Esercizio 16 Si dimostri che per ogni enumerazione effettiva esiste $i \in K$ tale che $\phi_i(x) = \begin{cases} 0 & x = i \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$

Svolgimento Si definisca la seguente funzione $\psi(x,y)$ che è intuitivamente calcolabile, quindi possiamo applicare s-m-n al suo indice e ad x e ottenere la funzione calcolabile totale f col solito metodo:

$$\phi_{f(x)}(y) = \psi(x,y) = \begin{cases} 0 & x = y \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$$

Per il teorema di ricorsione, $\exists\, n \mid \phi_n = \phi_{f(n)}$ e quindi $\forall\, y$ si ha

$$\phi_n(y) = \phi_{f(n)}(y) = \psi(n,y) = \begin{cases} 0 & y = n \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$$

Quindi esiste una funzione calcolabile che converge solamente sul suo indice, indipendentemente dall'enumerazione scelta.

Esercizio 17 Sia A_i l'insieme di indici costruito nel padding lemma e si dimostri che $A_i \leq_{rec} K$

Svolgimento A_i è generato da una funzione ricorsiva primitiva, quindi è ricorsivamente enumerabile e la tesi segue dalla completezza di K

Esercizio 18 Dato i, si dica se l'insieme $K_i = \{j \mid j \in K \wedge j \leq i\}$ è ricorsivo, ricorsivamente enumerabile o non ricorsivamente enumerabile.

Svolgimento Dato che K_i è finito, è ricorsivo.

Esercizio 19 Si dimostri che $A = \{x \mid \forall\, y.\phi_x(y) \uparrow\}$ non è ricorsivamente enumerabile.

Svolgimento Per farlo, dimostriamo che $\overline{K} \leq A$, cioè $K \leq \overline{A} = K_1$
Si definisca la seguente $\psi(x, y)$ che è intuitivamente calcolabile, e quindi possiamo applicare s-m-n al suo x e ottenere come al solito la funzione calcolabile totale f

$$\phi_{f(x)}(y) = \psi(x, y) = \begin{cases} 1 & x \in K \wedge \exists\, z.\phi_x(z) \downarrow \\ \textit{indefinita} & \textit{altrimenti} \end{cases}$$

$x \in K \Rightarrow \phi_{f(x)} = \lambda y.1 \Rightarrow f(x) \in \overline{A}$ (basta prendere $z = x$)

$x \notin K \Rightarrow \phi_{f(x)} = \lambda y.\textit{indefinita} \Rightarrow f(x) \in A$

Esercizio 20 Un insieme di indici che rappresenta una singola funzione può essere sempre costruito usando il padding lemma?

Svolgimento No, perché ogni insieme di indici A_x costruito come nel padding lemma a partire dall'indice x è ricorsivamente enumerabile, in quanto immagine di una funzione primitiva ricorsiva (e quindi calcolabile totale) mentre ci sono insiemi di indici che rappresentano funzioni, per esempio quella che rappresenta la funzione ovunque indefinita, che non sono ricorsivamente enumerabili (vedi esercizio 19)

Esercizio $\Leftarrow A \subseteq \mathbb{N}$
g è adesso in nostro possesso, nota per ipotesi, totale e strettamente crescente. Voglio far vedere che la sua funzione caratteristica è calcolabile.
$x \in A$? Voglio rispondere in modo costruttivo: valuto $g(0), g(1), g(2) \ldots$ e mi fermo quando $g(n) > x$
Mi fermo sempre? Si, perché g strettamente crescente, quindi $g(0) < g(1) < g(2) \ldots$ e sono certo di trovare una $g(n) > x$ per il **buon ordinamento di** $\mathbb{N}$ ($\forall\, z \in \mathbb{N}$ ho che l'insieme $[0, z] \cap \mathbb{N}$ è finito), quindi il processo di enumerazione si ferma.
Se $g(n-1) = x$ allora $\chi_A(x) := 1$, altrimenti $\chi_A(x) := 0$
χ_A è totale? Si, un certo numero può essere $m = x$ o $m \neq x$, non ho altri casi. Quindi prendo $g(n-1) = m \ldots$ Calcolabile? Si $\Rightarrow$ A è ricorsivo

3.2 Complessità

SAT $F = \bigwedge_i C_i$ formula in CNF (Conjuctive Normal Form): un certo numero di clausole di OR unite tra loro con AND. Quindi $C_i = \bigvee_j l_{ij}$
Con SAT ci chiediamo se esiste un assegnamento che rende vera la formula. Vediamo come SAT$\leq_f$ 3-SAT, dove imponiamo che le clausole abbiano esattamente 3 letterali. Partiamo da una formula con un numero arbitrario di letterali per finire in una formula con esattamente 3 letterali per clausola. Dividiamo per casi:

C_i ha un letterale , $C_i = l$. Posso aggiungere z_1, z_2 nuove variabili
$\Rightarrow C_i' = (l \vee z_1 \vee z_2) \wedge (l \vee \overline{z_1} \vee z_2) \wedge (l \vee z_1 \vee \overline{z_2}) \wedge (l \vee \overline{z_1} \vee \overline{z_2})$

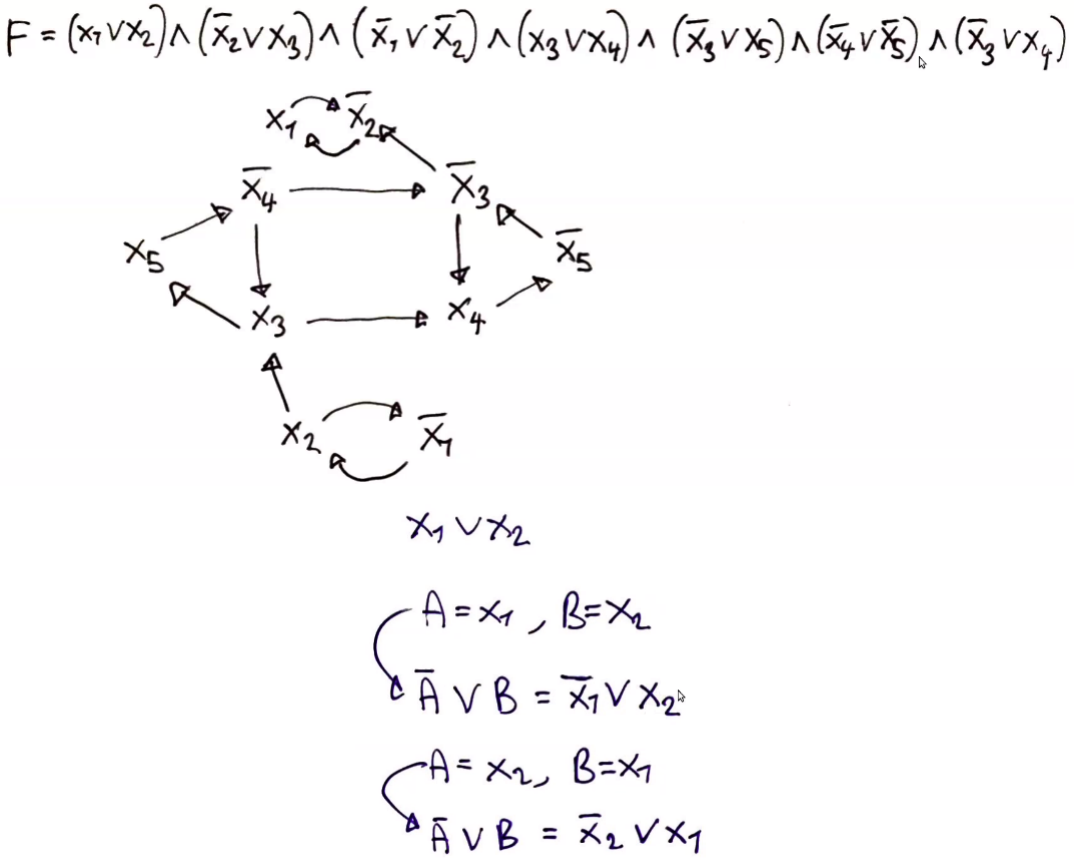
C_i ha 2 letterali , $C_i = l_1 \vee l_2$ possiamo aggiungere una nuova variabile e fare in modo che indipendentemente dal valore di verità della nuova variabile l'espressione di C_i non cambi
$\Rightarrow C_i' = (l_1 \vee l_2 \vee z) \wedge (l_1 \vee l_2 \vee \overline{z})$

C_i ha 3 letterali $\Rightarrow C_i' = C_i$

C_i ha m letterali. Aggiungo $m-3$ letterali $z_i, \dots, z_{m-3}$
$\Rightarrow C'_i = (l_{i1} \vee l_{i2} \vee z_i) \wedge (l_{i3} \vee \overline{z_1} \vee z_3) \wedge \dots \wedge (l_{i4} \vee \overline{z_2} \vee z_3) \wedge \dots \wedge (l_{im-2} \vee \overline{z_{m-4}} \vee z_{m-3}) \wedge (l_{im-1} \vee l_{im-2} \vee \overline{z_{m-3}})$
avendo cura di mettere un letterale e successivamente il suo negato, così che la formula mantenga l'espressione.

Quanti and e variabili ho aggiunto? Sono in numero ragionevole, sta in logspace? Si. L'algoritmo che fa questa riduzione, che implementa F, è adatto ai nostri scopi di riduzione? Si, è una riduzione corretta da SAT a 3-SAT.
SAT è NP-Completo, quindi anche 3-SAT lo è

2-SAT Sta in $\mathscr{P}$
$A \Rightarrow B \equiv \overline{A} \vee B$ perché l'unico caso in cui l'implicazione è falsa è quando il vero implica il falso.
Quindi la formula in 2-SAT è composta da congiunti di clausole con 2 letterali $F = \bigwedge_{i=1}^{m} (l_{i1} \vee l_{i2})$ su n variabili
Partendo da una formula in CNF mi definisco un grafo $G = (V, E)$ con $V = \{\text{tutte variabili e loro negazioni}\} = \{x_1 \dots x_n\} \cup \{\overline{x_1} \dots \overline{x_n}\}$
$(\overline{l_{i1}}, l_{i2}) \in E \; \forall \, i$ che fa le veci di $\overline{A} \vee B$, quindi $l_{i1} \Rightarrow l_{i2}$
$(\overline{l_{i2}}, l_{i1}) \in E \; \forall \, i$, quindi $l_{i2} \Rightarrow l_{i1}$



$x_i \Rightarrow x_2$
$x_2 \Rightarrow x_1$

Abbiamo definito un grafo orientato a partire dall'espressione logica. 2SAT sta in P, ma per passarci da SAT servono 2^n nuove clausole, rendendo la riduzione esponenziale.
Se esiste un arco $l \to \bar{l}$ e non esiste $\bar{l} \to l$, allora $(l = tt \Rightarrow \bar{l} = tt$ assurdo$) \Rightarrow l = ff$
Se esiste $\bar{l} \to l$ e non esiste $l \to \bar{l}$ allora $(\bar{l} = tt \Rightarrow l = tt$ assurdo$) \Rightarrow l = tt$
Nel caso ci siano entrambi gli archi, sono incompatibili quindi non esiste nessun assegnamento a l, quindi la formula originale è insoddisfacibile.
Da questo si deduce l'algoritmo
Se nella stessa componente totalmente connessa di G trovo sia l che $\bar{l}$, allora F non è soddisfacibile.
Qual è la complessità della ricerca di un componenti totalmente connesse di un grafo diretto? Kosaraju in $O(|E|+|V|)$

Maxcut $\leq$ Qubo

Maxcut: ho $G(V, E)$ indiretto voglio trovare taglio $A \cup B = V$ che massimizza il numero di archi da A a B.

Quadratic Unconstraint Binary Optimization: assegnamento a variabili binarie $x_1, \ldots, x_n$ che massimizzi una formula del tipo $max \sum_{i,j}(b_i x_i + a_{ij} x_i x_j)$

Massimizzare distanza oggetti connessi da archi: se x oggetto in A e x connesso a y allora voglio evitare di prendere x, così da evitare di prendere anche y. Ricorda lo xor. Algebra di Bool x_i xor $x_j = x_i + x_j - 2x_i x_j$
$max \sum_{(i,j)\in E}(x_i + x_j - 2x_i x_j)$

$x_1 = 1$ allora $i \in A$

$x_i = 0$ allora $i \in B$

CRICCA $\leq$ NodeCover

CRICCA: dato un $G = (V, E)$ e k, ci chiediamo se esiste CRICCA con almeno k vertici in G (chiamando la CRICCA K)

NodeCover (Cover): sottoinsieme vertici $C \subseteq V$ tali che per ogni arco $(u, v) \in E$ di G o $u \in C$ o $v \in C$

Si definisce un altro grafo a partire da G che è il grafo complementare: stesso insieme di vertici ma insieme degli archi è il complementare dell'insieme dato (cioè, ho un arco $(u, v) \in \overline{E} \Leftrightarrow (u, v) \notin E$)
Esiste una CRICCA con k nodi in $G \Leftrightarrow$ esiste un NodeCover con al più $n - k$ nodi in G' con $n = |V|$
Si osserva che se un certo arco sta negli archi che coinvolgono gli elementi della cricca, che è sottoinsieme di E, significa che quell'arco non può stare in $\overline{E}$ cioè $\underline{(u, v) \in E_K \subseteq E_G} \Rightarrow (u, v) \notin \overline{E}$
Quindi $(u, v) \notin E \Rightarrow (u, v) \in \overline{E}$ allora $\underline{(u \in V_K \wedge v \in V_K)}$ quindi o $u \in V - V_K$ o $vinV - V_K$ con $|V - V_K| \leq n - k$
Devo trovare che in G' i nodi rimasti sono collegati tra loro (quindi CRICCA).