Elementi di Calcolabilità e Complessità

Appunti del corso di ECC del Professor Pierpaolo Degano, A.A. 2021/2022

Luca De Paulis

4 dicembre 2021

Introduzione

Questo file contiene i miei appunti (rielaborati) del corso di Elementi di Calcolabilità e Complessità tenuto dal Professor Pierpaolo Degano nell'anno 2021/2022. Dato che questi appunti sono stati scritti per me stesso, potrebbero essere pieni di errori o ragionamenti incomprensibili a tutti tranne che a me: in ogni caso ogni segnalazione è ben accetta!

Luca De Paulis

Indice

1 Introduzione alla Calcolabilità 1
- 1.1 Macchine di Turing . . . 1
- 1.2 Linguaggi FOR e WHILE . . . 4
- 1.3 Calcolabilità di funzioni . . . 8
- 1.4 Funzioni ricorsive . . . 10
  - 1.4.1 Enumerazione di Gödel . . . 13
  - 1.4.2 Funzione di Ackermann . . . 15
  - 1.4.3 Non esiste un formalismo capace di esprimere tutte e sole le funzioni calcolabili totali . . . 15
- 1.5 Funzioni generali ricorsive . . . 16
  - 1.5.1 Tesi di Church-Turing . . . 17
- 1.6 Primi teoremi sulle funzioni calcolabili . . . 17
  - 1.6.1 Enumerazioni effettive . . . 18
  - 1.6.2 Equivalenza tra MdT e funzioni generali ricorsive . . . 19

2 Calcolabilità di problemi 23
- 2.1 Problemi di decisione . . . 23
- 2.2 Separazione di $\mathcal{R}$ e $\mathcal{RE}$ . . . 25
- 2.3 Riduzioni di classi di problemi . . . 27
- 2.4 Studio di $\mathcal{R}$ e $\mathcal{RE}$ tramite riduzioni . . . 29
  - 2.4.1 Altri problemi completi per $\mathcal{RE}$ . . . 31
- 2.5 Il Teorema di Rice . . . 32

3 Teoria della Complessità 36
- 3.1 Complessità deterministica . . . 37
- 3.2 Complessità in spazio . . . 40
- 3.3 Non-Determinismo e Complessità non-deterministica . . . 43
- 3.4 Funzioni di misura appropriata . . . 46

4 Complessità di problemi 49
- 4.1 Classificazione di $\mathcal{P}$ e $\mathcal{NP}$ . . . 49
  - 4.1.1 Logica proposizionale . . . 50
  - 4.1.2 Il problema SAT . . . 52
- 4.2 Circuiti booleani e il Teorema di Cook-Levin . . . 54
  - 4.2.1 CIRCUIT-SAT $\leq_{\mathcal{L}}$ SAT . . . 56
  - 4.2.2 Tabella delle computazioni e $\mathcal{P}$-completezza di CIRCUIT-VALUE . . . 57
  - 4.2.3 Il Teorema di Cook-Levin . . . 60

A Esercizi 62
- A.1 Esercizi di calcolabilità . . . 62

1

Introduzione alla Calcolabilità

Nella prima parte del corso, dedicata alla **Teoria della Calcolabilità**, cercheremo di studiare cosa significhi *calcolare* qualcosa e quali siano i limiti delle *procedure* a disposizione degli esseri umani per calcolare.

Per far ciò bisogna innanzitutto definire il concetto di **algoritmo** oppure procedura: lo faremo definendo dei vincoli che ogni algoritmo deve soddisfare per esser ritenuto tale.

1. Dato che gli uomini possono calcolare solo seguendo procedure finite, un algoritmo deve essere **finito**, ovvero deve essere costituito da un numero finito di istruzioni.
2. Inoltre devono esserci un numero **finito** di istruzioni distinte, e ognuna deve avere un **effetto limitato** su **dati discreti** (nel senso di non continui).
3. Una **computazione** è quindi una sequenza finita di passi discreti con durata finita, né analogici né continui.
4. Ogni passo dipende solo dai **passi precedenti** e viene scelto in modo **deterministico**: se ripetiamo due volte la stessa esatta computazione nelle stesse condizioni dobbiamo ottenere lo stesso risultato e la stessa sequenza di passi.
5. Non imponiamo un limite al numero di passi e alla memoria a disposizione.

Questi vincoli non definiscono precisamente cosa sia un algoritmo, anzi, vedremo che vi sono diversi modelli di computazione che soddisfano questi 5 requisiti. Le domande a cui vogliamo rispondere sono:

- Modelli diversi che rispettano questi vincoli risolvono gli stessi problemi?
- Un tale modello risolve necessariamente tutti i problemi?

1.1 Macchine di Turing

Il primo modello di computazione che vedremo è stato proposto da Alan Turing nel 1936, ed è pertanto chiamato in suo nome.

Definizione 1.1.1 – Macchina di Turing

Una **Macchina di Turing** (MdT per gli amici) è una quadrupla (Q, Σ, δ, q_0) dove

- Q è un insieme finito, detto **insieme degli stati**. In particolare assumiamo che esista uno stato $h \notin Q$, detto stato terminatore o **halting state**.
- Σ è un insieme finito, detto **insieme dei simboli**. In particolare
  - esiste $\# \in \Sigma$ e lo chiameremo **simbolo vuoto**;
  - esiste $\triangleright \in \Sigma$ e lo chiameremo **respingente**.
- δ è una funzione
$$\delta : Q \times \Sigma \to (Q \cup \{h\}) \times \Sigma \times \{L, R, -\}$$
detta **funzione di transizione**. È soggetta al vincolo
$$\forall q \in Q : \exists q' \in Q : \quad \delta(q, \triangleright) = (q', \triangleright, R).$$
- q_0 è un elemento di Q detto **stato iniziale**.

La definizione formale di Macchina di Turing può sembrare complicata, ma l'idea alla base è molto semplice: abbiamo una macchina che opera su un **nastro illimitato** (a destra) su cui sono scritti simboli (ovvero elementi di Σ). In ogni istante di tempo, la *testa* della macchina legge una casella del nastro, contenente il **simbolo corrente**. La macchina mantiene inoltre al suo interno uno stato (ovvero un elemento di Q), inizialmente settato allo stato iniziale q_0.

Un singolo passo di computazione è il seguente:

- la macchina legge il simbolo corrente σ;
- la macchina usa la funzione di transizione δ per effettuare la mossa: in particolare calcola $\delta(q, \sigma)$, dove q è lo stato corrente, e ne ottiene una tripla (q', σ', M);
- la macchina cambia stato da q a q';
- la macchina scrive al posto di σ il simbolo σ';
- la macchina si sposta nella direzione indicata da M: se $M = L$ si sposta di un posto a sinistra, se $M = R$ si sposta di un posto a destra, se $M = -$ rimane ferma.

Per formalizzare questi concetti abbiamo bisogno di altre definizioni.

Definizione 1.1.2 – Monoide libero, o Parole su un Alfabeto

Dato un insieme finito Σ, il **monoide libero** su Σ, anche chiamato **insieme delle parole su** Σ, è l'insieme Σ^* così definito:
$$\Sigma^* := \bigcup_{n \in \mathbb{N}} \Sigma^n$$
dove

- $\Sigma^0 := \{\varepsilon\}$, dove ε è la parola vuota;
- $\Sigma^{n+1} := \{\sigma \cdot w : \sigma \in \Sigma, w \in \Sigma^n\}$ è l'insieme delle parole di lunghezza $n+1$, ottenute preponendo ad una parola di lunghezza n (ovvero $w \in \Sigma^n$) un simbolo $\sigma \in \Sigma$.

Tale insieme ammette un'operazione, ovvero la **concatenazione** di parole, e la parola vuota ε è l'identità destra e sinistra di tale operazione.

Osservazione 1.1.1 (-NoValue-). Un elemento di Σ^* è una stringa di caratteri di Σ di lunghezza arbitraria, ma sempre finita, in quanto ogni elemento di Σ^* deve essere contenuto in un qualche Σ^n.

Il nastro di una MdT può quindi essere formalizzato come un elemento di Σ^*. Questo tuttavia ancora non ci soddisfa per alcuni motivi:

- gli elementi di Σ^* sono illimitati a destra, ma non a sinistra, dunque la MdT potrebbe muoversi a sinistra ripetutamente fino a "cadere fuori dal nastro";
- non stiamo memorizzando da alcuna parte la posizione del cursore della MdT.

Per risolvere il primo problema possiamo assumere che ogni nastro inizi con il simbolo speciale $\triangleright$: per il vincolo sulla funzione di transizione ogni volta che la MdT si troverà nella casella più a sinistra (contenente $\triangleright$) sarà costretta a muoversi verso destra lasciando scritto il respingente.

Per quanto riguarda il secondo invece possiamo dividere il nastro infinito in tre parti:

- la porzione a sinistra del simbolo corrente, che è una stringa di lunghezza arbitraria che inizia per $\triangleright$ e quindi un elemento di $\triangleright\Sigma^*$;
- il simbolo corrente, che è un elemento di Σ;
- la porzione a destra del simbolo corrente, che è una stringa e quindi un elemento di Σ^*.

Quest'ultima porzione è una stringa che potrebbe terminare con un numero infinito di caratteri vuoti (#): dato che non siamo interessati (per il momento) a tenere tutti i simboli vuoti a destra dell'ultimo simbolo non-vuoto del nastro, considereremo la porzione a destra "eliminando" tutti i *blank* superflui.

In particolare indicando sempre con $\varepsilon \in \Sigma^*$ la stringa vuota e convenendo che

- $\#\varepsilon = \varepsilon\# = \varepsilon$ (la concatenazione della stringa vuota con il *blank* dà ancora la stringa vuota);
- $\sigma\varepsilon = \varepsilon\sigma = \sigma$ per ogni $\sigma \neq \#$ (la concatenazione della stringa vuota con un simbolo non-*blank* dà il simbolo)

possiamo considerare l'insieme

$$\Sigma^F := \left(\Sigma^* \cdot (\Sigma \setminus \{\#\})\right) \cup \{\varepsilon\},$$

ovvero l'insieme delle stringhe in Σ che finiscono con un carattere non-*blank*, più la stringa vuota.

Usando queste convenzioni, la stringa che definisce il nastro è finita: siamo pronti a definire la *configurazione* di una MdT in un dato istante.

Definizione 1.1.3 – Configurazione di una MdT

Sia $M = (Q, \Sigma, \delta, q_0)$ una MdT. Una **configurazione** è una quadrupla

$$(q, u, \sigma, v) \in Q \times \triangleright\Sigma^* \times \Sigma \times \Sigma^F.$$

Più nel dettaglio:

- q è lo stato corrente,
- u è la porzione del nastro che precede il simbolo corrente, ed inizia per $\triangleright$,
- σ è il simbolo corrente,
- v è la porzione del nastro che segue il simbolo corrente, ed è vuota oppure termina per un simbolo diverso da #.

Osserviamo che:

- il simbolo corrente può essere #;
- è possibile che il simbolo corrente sia # e $v = \varepsilon$ (cioè vuota),
- è possibile che u sia vuota solo nel caso in cui il simbolo corrente è $\triangleright$, poiché significherebbe trovarsi all'inizio del nastro.

Spesso indicheremo la quadrupla (q, u, σ, v) con $(q, u\underline{\sigma}v)$: la sottolineatura ci indicherà il simbolo corrente. In contesti in cui non sia necessario sapere la posizione del cursore scriveremo semplicemente (q, w) per risparmiare tempo.

Esempio 1.1.4 (-NoValue-). Ad esempio la configurazione

$$(q_0, \triangleright ab\#\#b\#a\underline{\#}b\#a)$$

indica che la MdT è nello stato q_0, sta leggendo il carattere #, a sinistra del simbolo letto ha la stringa $\triangleright ab\#\#b\#a$ e a destra $b\#a$.

1.2 Linguaggi FOR e WHILE

Introduciamo ora un secondo paradigma per il calcolo di algoritmi, ovvero quello dato dai linguaggi `FOR` e `WHILE`. In effetti anche se le MdT rispondono ai nostri requisiti formali per un algoritmo e sono il modello teorico delle **macchine di Von Neumann**, al giorno d'oggi non costruiamo una nuova macchina per ogni algoritmo che dobbiamo risolvere: usiamo dei **linguaggi di programmazione** che verranno interpretati o compilati e restituiranno il risultato del calcolo.

I linguaggi `FOR` e `WHILE` sono quindi la base teorica dei moderni linguaggi imperativi, e anche se sembrano mancare di espressività rispetto ad essi, vedremo che in realtà il linguaggio `WHILE` riesce a risolvere tutti e soli i problemi risolvibili da un linguaggio moderno.

Sintassi astratta

Definizione 1.2.1 – Sintassi astratta di FOR e WHILE

$$\begin{aligned} \texttt{EXPR} &::= n \mid x \mid E_1 + E_2 \mid E_1 \cdot E_2 \mid E_1 - E_2 && \text{Espr. aritmetiche} \\ \texttt{BEXPR} &::= b \mid E_1 < E_2 \mid \neg b \mid B_1 \vee B_2 && \text{Espr. booleane} \\ \texttt{CMD} &::= \texttt{skip} \mid x := E \mid C_1; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 && \text{Comandi} \\ &\quad \mid \texttt{for } x = E_1 \texttt{ to } E_2 \texttt{ do } C \mid \texttt{while } B \texttt{ do } C \end{aligned}$$

dove $n \in \mathbb{N}$, $x \in \texttt{Var}$ (che è un insieme *numerabile* di variabili), $b \in \mathbb{B} := \{\mathcal{T}, \mathcal{F}\}$.

Il linguaggio FOR contiene solo il comando `for`, il linguaggio WHILE contiene solo il comando `while`.

Semantica

Per definire la **semantica** dei linguaggi FOR e WHILE abbiamo bisogno di alcuni costrutti ausiliari. In particolare ogni nostro programma conterrà delle variabili che possono essere valutate oppure aggiornate (tramite il comando di assegnamento): dobbiamo *memorizzare* il loro valore.

Definizione 1.2.2 – Funzione memoria e funzione di aggiornamento

La funzione **memoria** è una funzione

$$\sigma : \texttt{Var} \to \mathbb{N}$$

definita solo per un sottoinsieme finito di Var.

La funzione di **aggiornamento** è una funzione

$$-[-/-] : (\texttt{Var} \times \mathbb{N}) \times \mathbb{N} \times \texttt{Var} \to (\texttt{Var} \times \mathbb{N})$$

definita da

$$\sigma[n/x](y) := \begin{cases} n & \text{se } y = x, \\ \sigma(y) & \text{altrimenti.} \end{cases}$$

Osservazione 1.2.1 (-NoValue-). La funzione di aggiornamento prende una memoria ($\sigma : \texttt{Var} \to \mathbb{N}$), un valore intero ($n \in \mathbb{N}$) e una variabile ($x \in \texttt{Var}$) e produce una nuova memoria $\sigma[n/x] : \texttt{Var} \to \mathbb{N}$ che si comporta come σ su tutte le variabili diverse da x, ma restituisce n quando l'input è x.

Tramite la memoria possiamo definire la funzione di valutazione delle espressioni aritmetiche, ovvero la loro **semantica**.

Definizione 1.2.3 – Funzione di valutazione semantica (aritmetica)

La **funzione di valutazione semantica (aritmetica)** è una funzione

$$\mathcal{E}[\![-]\!]- : \texttt{EXPR} \times (\texttt{Var} \to \mathbb{N}) \to \mathbb{N}$$

definita per induzione strutturale a partire da

$$\begin{aligned}
\mathcal{E}[\![n]\!]\sigma &:= n && \text{(val. dei naturali)}\\
\mathcal{E}[\![x]\!]\sigma &:= \sigma(x) && \text{(val. delle variabili)}\\
\mathcal{E}[\![E_1 + E_2]\!]\sigma &:= \mathcal{E}[\![E_1]\!]\sigma + \mathcal{E}[\![E_2]\!]\sigma && \text{(val. della somma)}\\
\mathcal{E}[\![E_1 \cdot E_2]\!]\sigma &:= \mathcal{E}[\![E_1]\!]\sigma \cdot \mathcal{E}[\![E_2]\!]\sigma && \text{(val. del prodotto)}\\
\mathcal{E}[\![E_1 - E_2]\!]\sigma &:= \mathcal{E}[\![E_1]\!]\sigma - \mathcal{E}[\![E_2]\!]\sigma && \text{(val. della sottrazione)}
\end{aligned}$$

Dato che il nostro linguaggio modella solo numeri naturali (quindi positivi), l'operazione di sottrazione sarà quella data dal **meno limitato**:

$$a - b := \begin{cases} a - b, \text{se } a > b \\ 0, \text{altrimenti.} \end{cases}$$

Analogamente possiamo definire la semantica delle espressioni booleane.

Definizione 1.2.4 – Funzione di valutazione semantica (booleana)

La **funzione di valutazione semantica (booleana)** è una funzione

$$\mathcal{B}[\![-]\!]- : \mathsf{BEXPR} \times (\mathtt{Var} \to \mathbb{N}) \to \mathbb{N}$$

definita per induzione strutturale a partire da

$$\begin{aligned}
\mathcal{B}[\![t]\!]\sigma &:= \mathcal{T} && \text{(val. del } \mathtt{true}\text{)}\\
\mathcal{B}[\![f]\!]\sigma &:= \mathcal{F} && \text{(val. del } \mathtt{false}\text{)}\\
\mathcal{B}[\![E_1 < E_2]\!]\sigma &:= \mathcal{E}[\![E_1]\!]\sigma < \mathcal{E}[\![E_2]\!]\sigma && \text{(val. del minore)}\\
\mathcal{B}[\![\neg B]\!]\sigma &:= \neg\mathcal{B}[\![B]\!]\sigma && \text{(val. del } \mathtt{not}\text{)}\\
\mathcal{B}[\![B_1 \vee B_2]\!]\sigma &:= \mathcal{B}[\![B_1]\!]\sigma \vee \mathcal{B}[\![B_2]\!]\sigma && \text{(val. della sottrazione)}
\end{aligned}$$

Osserviamo che i simboli usati nel linguaggio (come $+, <, \neg$, ed altri) sono solo **simboli formali**: per essere più precisi dovremmo differenziarli dalle funzioni effettive (ovvero quelle che compaiono a destra del $:=$).

Osservazione 1.2.2 (-NoValue-)**.** Le funzioni $\mathcal{E}$ e $\mathcal{B}$ si comportano come un **interprete**: ad esempio $\mathcal{E}$ prende un'espressione aritmetica, una memoria e restituisce la valutazione dell'espressione nella memoria data.

Tuttavia tramite il **currying** possiamo esprimere $\mathcal{E}$ come una funzione

$$\mathcal{E}[\![-]\!]- : \mathsf{EXPR} \to \Big((\mathtt{Var} \to \mathbb{N}) \to \mathbb{N}\Big)$$

ovvero come una funzione che prende un'espressione aritmetica e restituisce una *funzione* che a sua volta prenderà una memoria per restituire finalmente la valutazione dell'espressione nella memoria.

Anche se le due modalità in pratica ci portano allo stesso risultato, la seconda modella più l'azione di un **compilatore**: infatti nella seconda versione $\mathcal{E}$ prende un'espressione,

cioè del codice, e restituisce un *eseguibile* che avrà bisogno dei dati (cioè della memoria) per dare il suo risultato.

Lo stile usato per definire la semantica delle espressioni viene chiamato **semantica denotazionale**: in questo stile cerchiamo di associare ad ogni costrutto del linguaggio una funzione che ne dà la semantica (ad esempio abbiamo associato al + del linguaggio la funzione che somma due naturali).

Per quanto riguarda i comandi adopereremo un altro stile, detto **semantica operazionale**. Come si evince dal nome, cercheremo di definire una *macchina astratta* che modifica il proprio *stato interno* valutando a piccoli passi il comando da eseguire.

Definizione 1.2.5 – Sistema di transizioni

Si dice **sistema di transizioni** una coppia $(\Gamma, \to)$ dove

- Γ è l'insieme delle **configurazioni** oppure stati;
- $\to: \Gamma \to \Gamma$ è una funzione, detta **funzione di transizione**.

Nel caso della nostra macchina astratta, le configurazioni saranno delle coppie

$$\langle c, \sigma \rangle \in \texttt{CMD} \times (\texttt{Var} \to \mathbb{N})$$

ovvero delle coppie "comando da valutare", "memoria".

Per definire la semantica operazionale dei comandi useremo un approccio **small-step**, in cui ogni transizione

$$\langle c, \sigma \rangle \to \langle c', \sigma' \rangle$$

rappresenta un singolo passo dell'esecuzione del programma. Una **computazione** diventa allora una sequenza di passi, ovvero un elemento della chiusura transitiva e riflessiva di $\to$, che indicheremo come al solito come $\to^*$.

Analogamente alle MdT, una computazione **termina con successo** se

$$\langle c, \sigma \rangle \to^* \sigma',$$

ovvero se esauriamo la valutazione del comando c in un numero finito (anche se arbitrario) di passi.

La semantica operazionale dei comandi è dunque data attraverso una serie di assiomi e regole di inferenza, che insieme ci permettono di valutare ogni comando per induzione strutturale.

$$\frac{}{\langle \texttt{skip}, \sigma\rangle \to \sigma} \qquad \text{Assioma dello } \texttt{skip}$$

$$\frac{}{\langle x := E, \sigma\rangle \to \sigma[n/x]} \quad \text{se } \mathcal{E}[\![E]\!]\sigma = n \qquad \text{Assioma dell'assegnamento}$$

$$\frac{\langle C_1, \sigma\rangle \to \langle C_1', \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \to \langle C_1'; C_2, \sigma'\rangle} \qquad \text{Regola della sequenza 1}$$

$$\frac{\langle C_1, \sigma\rangle \to \langle C_2, \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \to \sigma'} \qquad \text{Regola della sequenza 2}$$

$$\frac{}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_1, \sigma\rangle} \quad \text{se } \mathcal{B}[\![B]\!]\sigma = \mathcal{T} \qquad \text{Assioma cond. 1}$$

$$\frac{}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \to \langle C_2, \sigma\rangle} \quad \text{se } \mathcal{B}[\![B]\!]\sigma = \mathcal{F} \qquad \text{Assioma cond. 2}$$

$$\frac{}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \langle i := n; C; \texttt{for } i = n_1 + 1 \texttt{ to } n_2 \texttt{ do } C, \sigma\rangle} \qquad \text{Assioma del } \texttt{for } 1$$
$$\text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = \mathcal{F}, \mathcal{E}[\![E_1]\!]\sigma = n_1, \mathcal{E}[\![E_2]\!]\sigma = n_2$$

$$\frac{}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \to \sigma} \quad \text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = \mathcal{T} \qquad \text{Assioma del } \texttt{for } 2$$

$$\frac{}{\langle \texttt{while } B \texttt{ do } C, \sigma\rangle \to \langle \texttt{if } B \texttt{ then } (C; \texttt{while } B \texttt{ do } C) \texttt{ else skip}, \sigma\rangle} \qquad \text{Assioma del } \texttt{while}$$

Dalla regola di transizione del `for` segue una proprietà fondamentale del linguaggio `FOR`: ogni suo programma termina in tempo finito. Infatti prima della prima iterazione la semantica ci impone di valutare le espressioni E_1 ed E_2, che verranno valutate a dei naturali n_1, n_2. A questo punto il ciclo `for` verrà eseguito esattamente $n_2 - n_1$ volte (dove il $-$ è sempre limitato, per cui se $n_1 > n_2$ non eseguiremo mai il ciclo) in quanto gli estremi di iterazione non possono essere modificati dai comandi del corpo del `for`.

Questo ci dimostra immediatamente che il linguaggio `FOR` **non è equivalente** alle macchine di Turing, ovvero esistono macchine di Turing che codificano algoritmi non risolvibili dal linguaggio `FOR`. (Studieremo in seguito cosa significa *codificare algoritmi*.)

Il linguaggio `WHILE` invece può codificare algoritmi che non terminano. Ad esempio si vede subito che la configurazione

$$\langle \texttt{while } \mathcal{T} \texttt{ do skip}, \sigma\rangle$$

diverge a prescindere da σ.

1.3 Calcolabilità di funzioni

Dopo aver definito le computazioni per le MdT e per i linguaggi `FOR` e `WHILE`, vogliamo spiegare cosa significa che una MdT o un comando *calcola* una funzione.

Funzioni

Prima di tutto, ricordiamo le definizioni di base sulle funzioni.

Definizione 1.3.1 – Funzione

Dati A, B insiemi, una **funzione** f da A in B è un sottoinsieme di $A \times B$ tale che

$$(a, b), (a, b') \in f \implies b = b'.$$

Scriveremo

- $f : A \to B$ per indicare una funzione da A in B
- $b = f(a)$ per dire $(a, b) \in f$.

Notiamo inoltre che non abbiamo fatto assunzioni sulla **totalità** di f: per qualche valore di $a \in A$ potrebbe non esistere un valore $b \in B$ tale che $f(a)$, cioè f potrebbe *non essere definita* in a.

Definizione 1.3.2 – Funzioni totali e parziali

Sia $f : A \to B$.

- f **converge su** $a \in A$ (e lo si indica con $f(a)\downarrow$) se esiste $b \in B$ tale che $f(a) = b$;
- f **diverge su** $a \in A$ (e lo si indica con $f(a)\uparrow$, o con $f(a) = \perp$) se f non converge su a;
- f è **totale** se $f(a)\downarrow$ per ogni $a \in A$;
- f è **parziale** se non è totale.

In generale le nostre funzioni saranno parziali.

Definizione 1.3.3 – Dominio ed immagine

Sia $f : A \to B$. Si dice **dominio** di f l'insieme

$$\operatorname{dom} f := \{\, a \in A \,:\, f(a)\downarrow \,\}.$$

Si dice **immagine** di f l'insieme

$$\operatorname{Im} f := \left\{\, b \in B \,:\, b = f(a) \text{ per qualche } a \in A \,\right\}.$$

Definizione 1.3.4 – Iniettività/surgettività/bigettività

Sia $f : A \to B$ una funzione.

- f è **iniettiva** se per ogni $a, a' \in A$, $a \neq a'$, allora $f(a) \neq f(a')$.
- f è **surgettiva** se $\operatorname{Im} f = B$.
- f è **bigettiva** se è iniettiva e surgettiva.

Calcolare funzioni

Definiamo ora quando una macchina/un comando *implementa* una funzione.

Definizione 1.3.5 – Turing-calcolabilità

Siano $\Sigma, \Sigma_0, \Sigma_1$ alfabeti, $\triangleright, \# \notin \Sigma_0 \cup \Sigma_1 \subsetneq \Sigma$.
Sia inoltre $f : \Sigma_0 \to \Sigma_1$, $M = (Q, \Sigma, \delta, q_0)$ una MdT.
Si dice allora che M **calcola** f (e che f è **Turing-calcolabile**) se per ogni $v \in \Sigma_0$

$$w = f(v) \text{ se e solo se } (q_0, \underline{\triangleright} v) \to^* (h, \triangleright w \underline{\#}).$$

Indicando con $M(v)$ il risultato della computazione della macchina M sulla configurazione iniziale $(q_0, \underline{\triangleright} v)$, questa definizione ci dice che gli output della funzione e della MdT sono esattamente gli stessi. In particolare, dato che le funzioni possono essere *parziali* e le macchine di Turing possono *divergere*, $M(v)\downarrow$ se e solo se f è definita su v, cioè se esiste w tale che $f(v) = w$.

Definizione 1.3.6 – `WHILE`-calcolabilità

Sia C un comando `WHILE`, g : `Var` $\to \mathbb{N}$. Si dice allora che C **calcola** g (e che g è **`WHILE`-calcolabile**) se per ogni σ : `Var` $\to \mathbb{N}$

$$n = g(x) \text{ se e solo se } (C, \sigma) \to^* \sigma^* \text{ e } \sigma^*(x) = n.$$

Analogamente possiamo definire il concetto di funzione `FOR`-calcolabile.
È vero che le funzioni `WHILE`-calcolabili sono tutte e sole le funzioni `FOR`-calcolabili? **No**: infatti dato che non abbiamo fatto assunzioni sulla totalità di g, essa può essere `WHILE`-calcolabile ma non `FOR`-calcolabile.
Un possibile problema nella definizione di calcolabilità data è che abbiamo supposto che le funzioni abbiano una specifica forma: sono funzioni $\Sigma_0^* \to \Sigma_1^*$ nel caso delle MdT, `Var` $\to \mathbb{N}$ nel caso dei comandi. Scegliendo altri insiemi con le stesse caratteristiche (quindi di cardinalità numerabile) cambiano le funzioni calcolabili?
Fortunatamente la risposta è **no**. Consideriamo una funzione $f : A \to B$: se gli insiemi A, B sono numerabili possiamo scegliere delle **codifiche** $A \to \mathbb{N}$, $\mathbb{N} \to B$. A questo punto possiamo

- trasformare l'input $a \in A$ in un naturale tramite la prima codifica;
- fare il calcolo tramite una funzione $\mathbb{N} \to \mathbb{N}$,
- trasformare l'output in un elemento di B tramite la seconda codifica.

In questo modo possiamo limitarci a solo funzioni $\mathbb{N} \to \mathbb{N}$, a patto che la codifica sia **effettiva**, cioè sia calcolabile anch'essa. Vedremo in seguito che esistono codifiche per rappresentare macchine di Turing come numeri.

1.4 Funzioni ricorsive

Introduciamo ora un ultimo formalismo per rappresentare un modello di calcolo, ovvero quello delle funzioni ricorsive.
Per semplificare la notazione useremo la λ-notazione per le funzioni anonime: la funzione $\lambda x.\, f(x)$ è la funzione che prende un unico parametro di ingresso x e restituisce $f(x)$.

Definizione 1.4.1 – Funzioni primitive ricorsive

La classe delle **funzioni primitive ricorsive** $\mathcal{PR}$ è la minima classe di funzioni che contenga gli schemi

I. **Zero:** $\lambda x_1, \dots, x_n.\ 0$ per ogni $n \in \mathbb{N}$

II. **Successore:** $\lambda x.\ x + 1$

III. **Proiezione:** $\lambda x_1, \dots, x_n.\ x_i$ per ogni $n \in \mathbb{N}$, $i = 1, \dots, n$

e che sia chiusa per gli schemi

IV. **Composizione:** se $g_1, \dots, g_k : \mathbb{N}^n \to \mathbb{N}$, $h : \mathbb{N}^k \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

$$\lambda x_1, \dots, x_n.\ h\Big(g_1(x_1, \dots, x_n), \dots, g_k(x_1, \dots, x_n)\Big)$$

appartiene ancora a $\mathcal{PR}$;

V. **Ricorsione Primitiva:** se $h : \mathbb{N}^{n+1} \to \mathbb{N}$, $g : \mathbb{N}^{n-1} \to \mathbb{N}$ appartengono a $\mathcal{PR}$, allora

$$f : \mathbb{N}^n \to \mathbb{N}$$

$$\begin{cases} f(0, x_2, \dots, x_n) = g(x_2, \dots, x_n) \\ f(n+1, x_2, \dots, x_n) = h(n, f(n, x_2, \dots, x_n), x_2, \dots, x_n) \end{cases}$$

appartiene ancora a $\mathcal{PR}$.

Una funzione è quindi primitiva ricorsiva se può essere costruita dagli schemi della definizione, ovvero se esiste una successione di funzioni $f_1, \dots, f_n$ tale che

- $f_n = f$,
- per ogni $i = 1, \dots, n$, f_i è definita come uno dei casi base (ovvero è la funzione zero, successore o proiezione) oppure è ottenuta dagli schemi induttivi (composizione e ricorsione primitiva) e *solamente* dalle funzioni $f_1, \dots, f_{i-1}$.

Osserviamo che il calcolo di ogni funzione ricorsiva primitiva *termina sempre*: infatti i casi base (I, II e III) terminano in un singolo passo e per induzione strutturale si vede che anche i casi IV e V terminano in tempo finito, poiché la ricorsione si fa sempre diminuendo il valore di x_1 nello schema V.

Facciamo degli esempi di funzioni ricorsive primitive.

Esempio 1.4.2 (-NoValue-). Consideriamo la seguente sequenza di funzioni:

$$f_1 = \lambda x.\ x, \qquad f_2 = \lambda x.\ x + 1, \qquad f_3 = \lambda x_1, x_2, x_3.\ x_2,$$

$$f_4 = f_2(f_3(x_1, x_2, x_3)), \qquad f_5 = \begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(n+1, x_2) = f_4(n, f_5(n, x_2), x_2) \end{cases}$$

Ognuna delle funzioni è primitiva ricorsiva: infatti

1. la prima corrisponde alla proiezione per $i = n = 1$;

2. la seconda corrisponde al successore;
3. la terza corrisponde alla proiezione con $n = 3, i = 2$;
4. la quarta è la composizione di f_2 ed f_3;
5. l'ultima è ottenuta per ricorsione primitiva dalle prime quattro funzioni.

Possiamo porci il problema di come si effettui il calcolo di una funzione primitiva ricorsiva quando ci vengono forniti degli argomenti.

Consideriamo due **regole di valutazione**.

▸ Call by value

Nel **call by value** si valutano prima gli operandi di una funzione e poi la funzione esterna: la **redex** (ovvero la prossima espressione da valutare) è la funzione più interna più a sinistra.

$$\begin{aligned}
\underline{f_5(2,3)} &= f_4(1, \underline{f_5(1,3)}, 3) \\
&= f_4(1, f_4(0, \underline{f_5(0,3)}, 3), 3) \\
&= f_4(1, f_4(0, \underline{f_1(3)}, 3), 3) \\
&= f_4(1, \underline{f_4(0,3,3)}, 3) \\
&= f_4(1, f_2(\underline{f_3(0,3,3)}), 3) \\
&= f_4(1, \underline{f_2(3)}, 3) \\
&= \underline{f_4(1,4,3)} \\
&= f_2(\underline{f_3(1,4,3)}) \\
&= \underline{f_2(4)} \\
&= 5.
\end{aligned}$$

▸ Call by need

Nel **call by need** valutiamo un'espressione *solo quando è necessario*: questa regola ci impone di valutare sempre l'espressione più esterna per prima. Per far ciò eviteremo di essere eccessivamente pignoli nella sintassi delle funzioni primitive ricorsive.

$$\begin{aligned}
\underline{f_5(2,3)} &= \underline{f_4(1, f_5(1,3), 3)} \\
&= \underline{f_2(f_3(1, f_5(1,3), 3))} \\
&= \underline{f_3(1, f_5(1,3), 3)} + 1 \\
&= \underline{f_5(1,3)} + 1 \\
&= \underline{f_4(0, f_5(0,3), 3)} + 1 \\
&= \underline{f_2(f_3(0, f_5(0,3), 3))} + 1 \\
&= \underline{f_3(0, f_5(0,3), 3)} + 1 + 1 \\
&= \underline{f_5(0,3)} + 1 + 1 \\
&= \underline{f_1(3)} + 1 + 1 \\
&= 3 + 1 + 1 = 5.
\end{aligned}$$

In entrambi i casi è chiaro che la funzione f_5 così definita è la somma.

1.4.1 Enumerazione di Gödel

Vogliamo ora mostrare che le macchine di Turing sono numerabili e esiste una loro enumerazione fatta da funzioni ricorsive primitive.

Definizione 1.4.3 – Relazione primitiva ricorsiva

Una relazione $P \subseteq \mathbb{N}^k$ è detta **primitiva ricorsiva** se lo è la sua funzione caratteristica $\chi_P : \mathbb{N}^k \to \{0,1\}$, definita da

$$\chi_P(x_1, \ldots, x_k) := \begin{cases} 1 & \text{se } (x_1, \ldots, x_k) \in P \\ 0 & \text{altrimenti.} \end{cases}$$

La funzione caratteristica di un sottoinsieme di $\mathbb{N}^k$ è quindi una funzione a valori booleani (ovvero è un predicato) che restituisce 1 sugli elementi che appartengono al sottoinsieme e 0 altrimenti. In futuro confonderemo senza porci troppi problemi le relazioni e le loro funzioni caratteristiche.

Per costruire l'enumerazione delle MdT ci serve un predicato molto importante, che è il predicato $\mathtt{P} : \mathbb{N} \to \{0,1\}$ definito da

$$\mathtt{P}(p) = 1 \text{ se e solo se } p \text{ è un numero primo.}$$

È possibile dimostrare che tale predicato è primitivo ricorsivo: i numeri primi saranno quindi una parte fondamentale dell'enumerazione.

Gli altri due ingredienti della ricetta sono i seguenti.

- Il **Teorema di Esistenza e Unicità della Fattorizzazione in Primi**, che dice che se $\mathtt{P} = p_0, p_1, \ldots$ è l'insieme dei numeri primi (considerato questa volta come relazione unaria!) allora per ogni $n \in \mathbb{N}$ esiste un numero finito di esponenti $e_i \neq 0$ tali che

$$n = \prod_{i \in \mathbb{N}} p_i^{e_i}.$$

- La funzione che prende un $n = \prod_{i \in \mathbb{N}} p_i^{e_i}$ e restituisce l'esponente e_k del k-esimo fattore della fattorizzazione è primitiva ricorsiva.

In particolare se considerando una sequenza finita di naturali $(n_0, \dots, n_k)$ essa può essere codificata in modo *unico* come il numero

$$N := p_0^{n_0+1} \cdot p_k^{n_k+1}$$

e dalla codifica possiamo ricavare la sequenza originale grazie alle funzioni che danno gli esponenti della fattorizzazione.

Possiamo vedere finalmente una *quasi*-codifica delle macchine di Turing: in particolare delineeremo una funzione iniettiva che mappa le macchine di Turing ai naturali. La codifica vera e propria, chiamata **enumerazione di Gödel**, è in realtà una funzione bigettiva e primitiva ricorsiva.

Data una macchina $M := (Q, \Sigma, \delta, q_0)$, siccome Q e Σ sono finiti possiamo numerare i loro elementi:

$$Q = \{ q_0, \dots, q_n \}, \qquad \Sigma = \{ \sigma_0, \dots, \sigma_m \}.$$

Ogni transizione specificata da δ può essere rappresentata come una quintupla

$$(q_i, \sigma_j, q_k, \sigma_l, D) \in Q \times \Sigma \times Q \times \Sigma \times \{ L, R, - \}$$

e pertanto possiamo codificarla come il naturale

$$p_0^{i+1} \cdot p_1^{j+1} \cdot p_2^{k+1} \cdot p_3^{l+1} \cdot p_4^{m_D}.$$

Dobbiamo risolvere alcune piccole inesattezze:

- q_k potrebbe essere lo stato terminatore h, dunque numereremo h come lo stato q_{k+1};
- non abbiamo definito come numerare i simboli $L, R, -$, ma possiamo semplicemente considerarli come ulteriori simboli rispetto a quelli di Σ, dunque numereremo

$$L \mapsto \sigma_{m+2}, \quad R \mapsto \sigma_{m+3}, \quad - \mapsto \sigma_{m+4}.$$

Abbiamo quindi associato un numero ad ogni quintupla. Ma una macchina di Turing è semplicemente un insieme di quintuple: ordiniamole lessicograficamente (ovvero prima le ordiniamo per i, poi per j, ecc) e quindi abbiamo una sequenza finita di numeri naturali $g_0, \dots, g_r$, ognuno dei quali codifica una quintupla.

Definiremo quindi **numero di Gödel** della macchina M il numero $N_M := p_0^{g_0+1} \cdots p_r^{g_r+1}$. Questa pseudo-codifica è sicuramente iniettiva grazie al Teorema di Fattorizzazione Unica, ma non è detto che sia surgettiva: l'idea della vera enumerazione di Gödel è simile ma la realizzazione è molto più complessa.

Osserviamo che una volta numerate le quintuple possiamo anche enumerare intere computazioni con un singolo numero: infatti una computazione (terminante) è una successione finita di configurazioni $\gamma = (q_i, w_j) \in Q \times \Sigma^*$ e le configurazioni possono certamente essere enumerate.

Infine ogni passaggio fatto finora è primitivo ricorsivo, dunque il procedimento di enumerazione delle MdT e delle computazioni è primitivo ricorsivo.

1.4.2 Funzione di Ackermann

La maggior parte delle funzioni che usiamo comunemente è primitiva ricorsiva: i logici di inizio '900 si chiesero perciò se le funzioni primitive ricorsive potessero modellare *tutti* gli algoritmi che *terminano sempre*, ovvero tutte le funzioni totali.

La risposta purtroppo è **no** e ci è data dalla **funzione di Ackermann**, definita da

$$A(z,x,y) \coloneqq \begin{cases} A(0,0,y) & = y \\ A(0,x+1,y) & = A(0,x,y)+1 \\ A(1,0,y) & = 0 \\ A(z+2,0,y) & = 1 \\ A(z+1,x+1,y) & = A(z,A(z+1,x,y),y). \end{cases}$$

Questa funzione è totale: basta mostrarlo per induzione sui casi ricorsivi, che sono il secondo e il quinto. Il primo termina in tempo finito poiché il secondo parametro decresce ad ogni applicazione; il quinto termina poiché l'applicazione interna avviene col secondo parametro diminuito di uno, mentre nell'applicazione esterna il primo parametro decresce.

Tuttavia l'ultimo caso non può essere espresso in termini di funzioni ricorsive primitive: la doppia ricorsione innestata non rientra nei cinque schemi e non è neanche ricavabile dagli schemi. Inoltre questa funzione è *intuitivamente calcolabile*: ad ogni passo restituiamo un risultato oppure calcoliamo ricorsivamente la funzione diminuendo il valore degli argomenti, quindi è certamente scrivere un programma che la calcola.

Segue che le funzioni primitive ricorsive **non sono tutte le funzioni calcolabili totali**.

1.4.3 Non esiste un formalismo capace di esprimere tutte e sole le funzioni calcolabili totali

La speranza è quindi che esista un formalismo, diverso dalle funzioni primitive ricorsive, capace di esprimere *tutte e sole* le funzioni calcolabili totali.

Teorema 1.4.4 –

Non esiste un formalismo capace di esprimere tutte e soli le funzioni calcolabili totali.

Dimostrazione. Supponiamo per assurdo che esista un formalismo $\mathcal{F}$ capace di esprimere tutte e sole le funzioni calcolabili totali. Sicuramente $|\mathcal{F}| = |\mathbb{N}|$: infatti sono al più numerabili poiché ogni funzione calcolabile è un algoritmo, e quindi è una stringa su un alfabeto finito; d'altro canto sono almeno $|\mathbb{N}|$ poiché le funzioni costanti sono certamente calcolabili totali.

Consideriamo allora una numerazione $\mathbb{N} \to \mathcal{F}$, $n \mapsto f_n$ che sia *bigettiva e calcolabile*.[a]

Costruiamo la funzione $g : \mathbb{N} \to \mathbb{N}$, $g(n) = f_n(n) + 1$. Tale funzione è

- calcolabile, in quanto è la composizione della numerazione di $\mathcal{F}$ che è calcolabile, del calcolo di $f_n(n)$ (che si può fare in quanto f_n è calcolabile) e del successore;
- totale, in quanto composizione di funzioni totali.

Segue che g è una funzione calcolabile totale, e quindi $g \in \mathcal{F}$.

Tuttavia per ogni $n \in \mathbb{N}$ vale che $g(n) = f_n(n) + 1 \neq f_n(n)$, e dunque in particolare $g \neq f_n$ per ogni $n \in \mathbb{N}$. Ma le funzioni di $\mathcal{F}$ sono tutte e sole della forma f_n per qualche $n \in \mathbb{N}$, da cui segue che $g \notin \mathcal{F}$, che è assurdo. □

[a] Dovremmo dimostrare che ne esiste una, ma faremo finta di niente.

La tecnica usata nella dimostrazione di questo teorema viene chiamata **diagonalizzazione** ed è una tecnica usata molto frequentemente nella logica e nella teoria della calcolabilità.

1.5 Funzioni generali ricorsive

Il problema messo in luce dal Teorema 1.4.4 ci impedisce di creare un formalismo che esprima solo funzioni totali: per risolverlo dobbiamo estendere la classe di funzioni di nostro interesse alle *funzioni parziali*. Questo non è assurdo: molte funzioni di nostro interesse (anche aritmetiche) sono non definite su alcuni input, e abbiamo anche visto che esistono MdT e funzioni `WHILE` che non convergono.

Per farlo, introduciamo un ultimo formalismo.

Definizione 1.5.1 – Operatore di minimizzazione

Dato un predicato P, ovvero una funzione $P : \mathbb{N} \to \{0, 1\}$, possiamo costruire l'**operatore di minimizzazione** μ tale che

$$\mu y.\, P(y) := \min\{\, y \;:\; P(y) = 1 \,\}.$$

Osserviamo che $\mu y.\, P(y)$ potrebbe non essere definito: se P è il predicato sempre falso, non esiste un minimo y che lo renda vero.

Definizione 1.5.2 – Funzioni generali ricorsive

L'insieme delle funzioni **generali ricorsive** $\mathcal{GR}$ è il minimo insieme di funzioni contenenti gli schemi I, II, III delle funzioni primitive ricorsive, chiuso per gli schemi IV e V e per lo schema

VI. **Minimizzazione:** se $\varphi : \mathbb{N}^{n+1} \to \mathbb{N} \in \mathcal{GR}$, allora la funzione $\psi : \mathbb{N}^n \to \mathbb{N}$ definita da

$$\psi(x_1, \ldots, x_n) := \mu y.\, \Big(\varphi(\overline{x}, y) = 0 \text{ e } \big(\forall z \leq y.\, \varphi(\overline{x}, z)\downarrow\big)\Big)$$

appartiene ancora a $\mathcal{GR}$.

Una funzione ottenuta per minimizzazione è intuitivamente calcolabile: si calcola $\varphi(\overline{x}, y)$ per $y = 0, 1, 2, \ldots$ e ci si ferma al primo valore che soddisfi entrambe le proprietà. Osserviamo che una ψ ottenuta in tale modo *può essere parziale* in due casi:

- se $\varphi(\overline{x}, y) \neq 0$ per ogni y allora non c'è minimo, e quindi la computazione non termina;
- se $\varphi(\overline{x}, z)\uparrow$ per qualche valore z precedente al primo zero, nel calcolare $\varphi(\overline{x}, z)$ non ci fermeremo mai, e quindi il calcolo di ψ non termina.

Esempio 1.5.3 (-NoValue-). Sia $\varphi := \lambda x, y.\, 3$. φ è primitiva ricorsiva, e quindi è anche totale, ma ψ ottenuta per minimizzazione su φ è **sempre indefinita**.

Infatti $\psi(x)$ è il minimo y per cui $\varphi(x, y) = 0$ (e l'altra condizione) ma questo non accade mai.

1.5.1 Tesi di Church-Turing

Abbiamo quindi visto diversi formalismi capaci di esprimere funzioni intuitivamente calcolabili, ovvero algoritmi. In che relazione sono?

Negli anni '20 e '30 i principali ideatori di questi formalismi (come Church, Turing, Gödel) dimostrarono l'equivalenza di tutti i formalisti che abbiamo visto: in particolare Turing e Church congetturarono che tutti i modelli capaci di esprimere funzioni calcolabili fossero **Turing-equivalenti**.

Teorema 1.5.4 – Tesi di Church-Turing

Le funzioni (intuitivamente) calcolabili sono tutte e sole le funzioni Turing-calcolabili.

Dato che le funzioni Turing-calcolabili sono tutte e sole quelle `WHILE`-calcolabili oppure tutte e sole le funzioni generali ricorsive, da questo momento in poi non specificheremo più il formalismo usato (tanto sono tutti equivalenti!) e parleremo semplicemente di **funzioni calcolabili**.

1.6 Primi teoremi sulle funzioni calcolabili

Avendo stabilito il *framework* in cui lavoreremo, possiamo finalmente iniziare a dimostrare alcune proprietà delle funzioni calcolabili.

Teorema 1.6.1 – Esistenza di funzioni non calcolabili

Le funzioni calcolabili sono in quantità numerabile. In particolare esistono funzioni non calcolabili.

Per dimostrare questo teorema useremo il fatto che le funzioni $\mathbb{N} \to \mathbb{N}$ sono in quantità più che numerabile, quindi dimostriamolo.

Teorema 1.6.2 – Diagonalizzazione di Cantor

Siano A, B insiemi con $|A| \geq |\mathbb{N}|$, $|B| \geq 2$. Allora

$$\left|\{ f : A \to B \}\right| > |\mathbb{N}|.$$

Dimostrazione. È sufficiente dimostrare il Teorema nel caso in cui A sia numerabile e B abbia esattamente due elementi. Allora senza perdita di generalità sia $A = \mathbb{N}$, $B = \{0, 1\}$.

Sia $\mathcal{F} := \left|\{ f : \mathbb{N} \to \{0, 1\} \}\right|$ e supponiamo per assurdo $|\mathcal{F}| = |\mathbb{N}|$. Allora esiste una numerazione bigettiva degli elementi di $\mathcal{F}$, ovvero una funzione $n \mapsto f_n \in \mathcal{F}$.

Consideriamo allora $g : \mathbb{N} \to \{0, 1\}$ (e quindi $g \in \mathcal{F}$) definita da

$$g(n) := \begin{cases} 0 & \text{se } f_n(n) = 1, \\ 1 & \text{altrimenti.} \end{cases}$$

Ma allora $g(n) \neq f_n(n)$ per ogni n, dunque $g \neq f_n$ per ogni n. Ma gli elementi di $\mathcal{F}$ sono tutti e soli della forma f_n al variare di $n \in \mathbb{N}$, dunque $g \notin \mathcal{F}$, che è assurdo. □

Possiamo dimostrare il Teorema 1.6.1.

Dimostrazione del Teorema 1.6.1. Sia $\mathcal{C}$ l'insieme delle funzioni calcolabili. Mostriamo che $|\mathcal{C}| \geq |\mathbb{N}|$ e $|\mathcal{C}| \leq |\mathbb{N}|$.

- $|\mathcal{C}| \geq |\mathbb{N}|$ Le funzioni $\lambda x.\, n$ al variare di $n \in \mathbb{N}$ sono in quantità numerabile e sono tutte calcolabili.
- $|\mathcal{C}| \leq |\mathbb{N}|$ Ogni funzione calcolabile è calcolata da una macchina di Turing, dunque $|\mathcal{C}| \leq |\mathcal{M}|$ dove $\mathcal{M}$ è l'insieme delle MdT; inoltre $|\mathcal{M}| \leq |\mathbb{N}|$ poiché possiamo numerarle tramite l'enumerazione di Gödel, dunque $|\mathcal{C}| \leq |\mathbb{N}|$.

Segue che $|\mathcal{C}| = |\mathbb{N}|$.

Per dimostrare che esistono funzioni non calcolabili osserviamo che $\mathcal{C}$ è un sottoinsieme di tutte le funzioni $\mathbb{N} \to \mathbb{N}$. Tuttavia per il Teorema 1.6.2 le funzioni $\mathbb{N} \to \mathbb{N}$ sono in quantità più che numerabile, dunque $\mathcal{C}$ deve essere un sottoinsieme proprio delle funzioni $\mathbb{N} \to \mathbb{N}$: in particolare devono esistere funzioni che non sono in $\mathcal{C}$. □

1.6.1 Enumerazioni effettive

Dato che abbiamo dimostrato che le funzioni calcolabili sono numerabili possiamo porci il problema di come *enumerarle*. Per far ciò non enumereremo direttamente le funzioni, ma solo le macchine di Turing.

In particolare considereremo sono **enumerazioni effettive**, ovvero enumerazioni che dipendono solo dalla **sintassi** della macchina di Turing, cioè soltanto dai simboli che compaiono al suo interno, e non dal comportamento. [1]

Si può dimostrare che tutti i teoremi che vedremo sono indipendenti dal formalismo e dall'enumerazione scelta, purché quest'ultima sia effettiva: fissiamo quindi una enumerazione effettiva $n \mapsto M_n$. Tale enumerazione induce un'enumerazione sulle funzioni calcolabili: indicheremo con φ_i la funzione calcolata dalla macchina M_i.

Osserviamo però che dati due indici i, j diversi sicuramente vale che $M_i \neq M_j$ ma è possibile che $\varphi_i = \varphi_j$, ovvero è possibile che due macchine diverse calcolino la stessa funzione.

In realtà vale un teorema molto più forte, solitamente conosciuto come **Padding Lemma**.

Teorema 1.6.3 – Padding Lemma

Per ogni indice i esiste un insieme numerabile A_i di indici tale che per ogni $j \in A_i$

$$\varphi_j = \varphi_i,$$

ovvero ogni funzione calcolabile è calcolata da infinite MdT.

Dimostrazione. Consideriamo l'algoritmo che calcola φ_i: aggiungendo uno `skip` alla fine si ottiene un algoritmo diverso (e quindi una macchina di Turing diversa) che calcola la stessa funzione. Aggiungendo altri `skip` otteniamo una quantità numerabile di algoritmi che calcolano φ_i. □

[1] In realtà una enumerazione si dice effettiva se è ottenuta post-componendo l'enumerazione di Gödel con una bigezione $\mathbb{N} \leftrightarrow \mathbb{N}$.

1.6.2 Equivalenza tra MdT e funzioni generali ricorsive

Ora mostriamo che ogni funzione calcolabile può essere scritta in una forma standard, detta **Forma Normale di Kleene**.

Teorema 1.6.4 – Forma Normale di Kleene

Esistono un predicato $T : \mathbb{N}^3 \to \{0,1\}$ (detto **predicato di Kleene**) e una funzione $U : \mathbb{N} \to \mathbb{N}$ entrambi calcolabili totali tali che

$$\forall i, x : \varphi_i(x) = U\big(\mu y.\, T(i,x,y)\big).$$

Inoltre T e U sono primitivi ricorsivi.

Dimostrazione. Definiamo $T(i,x,y) = 1$ se e solo se la *computazione* $M_i(x)$ converge ad una configurazione $(h, \triangleright z)$ e y è la codifica di tale computazione.

T è calcolabile: in effetti basta recuperare la macchina M_i dalla lista (che è un'operazione calcolabile), decodificare y come una computazione $c_1 \dots c_n$ (dove tutte queste sono configurazioni) e verificare che $M_i(x)$ converga effettivamente ad una configurazione $c_n = (h, \triangleright z)$ tramite la computazione codificata da y.[a]

Allora possiamo definire U in modo che $U(y) =$ "la codifica di z". La computazione del membro destro procede quindi in questo modo: si scorrono i valori di y e si trova il primo valore di y che corrisponde alla computazione di $M_i(x)$. Se esiste, ritorniamo la codifica di z, altrimenti il procedimento non termina.

È facile vedere che ciò è uguale a $\varphi_i(x)$ per ogni x: distinguiamo due casi.

- Se $\varphi_i(x) = n$ la computazione di $M_i(x)$ termina e dà come configurazione finale $(h, \triangleright z)$, dove n codifica z. Ma allora esiste un $y \in \mathbb{N}$ che codifica la computazione terminante di $M_i(x)$ (e tale y è anche unico, poiché la computazione è unica) e dunque y è il minimo valore del terzo parametro per cui vale che $T(i,x,y) = 1$. Per definizione di U, $U(y)$ è la codifica di z e dunque è n.
- Se $\varphi_i(x)$ diverge, allora non esiste una codifica della computazione di $M_i(x)$ (poiché la computazione diverge) e pertanto non esiste un y per cui $T(i,x,y) = 1$. In particolare $U(y)$ diverge.

Infine T ed U sono primitivi ricorsivi in quanto lo sono le codifiche e i controlli effettuati, e composizione di funzioni primitive ricorsive è ancora primitiva ricorsiva. □

[a] Osserviamo che questo procedimento termina sempre poiché le computazioni sono di lunghezza finita, dunque possiamo verificare in tempo finito se il calcolo di $M_i(x)$ corrisponde alla computazione cercata oppure no.

Osservazione 1.6.1 (-NoValue-). Il teorema ci dice che ogni funzione calcolabile φ_i è esprimibile come composizione di due funzioni primitive ricorsive (T, U) e una funzione generale ricorsiva (data dalla minimizzazione), o equivalentemente da due comandi `FOR` e un comando `WHILE`.

Il Teorema di Forma Normale è uno strumento molto potente e lo useremo per dimostrare uno dei risultati fondamentali della Teoria della Calcolabilità.

Teorema 1.6.5 – Teorema di Enumerazione

Esiste un indice z tale che per ogni indice i, input x si ha

$$\varphi_z(i, x) = \varphi_i(x).$$

Il Teorema di Enumerazione ci garantisce l'esistenza di una MdT particolare, chiamata **Macchina Universale**, capace di simulare tutte le altre macchine di Turing. Osserviamo che per il Padding Lemma in realtà esistono infinite macchine universali.

Dimostrazione. Siano U, T i predicati dati dal Teorema 1.6.4 e definiamo

$$\varphi_z := \lambda i, x.\ U(\mu y T(i, x, y)).$$

Tale funzione è ben definita e calcolabile (poiché composizione di funzioni calcolabili), inoltre per il Teorema 1.6.4 si ha che $\varphi_i(x) = \varphi_z(i, x)$, come voluto. □

Per quanto il Teorema di Enumerazione possa sembrare eccessivamente potente (abbiamo costruito un algoritmo che può simulare tutti gli algoritmi!) in realtà la macchina Universale svolge esattamente il ruolo di un interprete: prende in input un altro programma e dei dati ed esegue il programma su quei dati.

Osserviamo inoltre che ciò funziona solo perché possiamo trasformare una funzione, cioè un programma, in dati, e ciò è possibile solo perché le funzioni calcolabili sono numerabili.

Abbiamo quindi un metodo per trasformare un indice in un argomento. È possibile fare il contrario?

Teorema 1.6.6 – Teorema del Parametro

Esiste una funzione calcolabile totale ed iniettiva $s : \mathbb{N}^2 \to \mathbb{N}$ tale che per ogni i, x

$$\varphi_{s(i,x)} = \lambda z.\ \varphi_i(x, z).$$

Il senso intuitivo del Teorema del Parametro (anche chiamato Teorema s-1-1 per un motivo che vedremo definendo la sua forma generale) è che se un algoritmo dipende da due input e uno dei due è costante/conosciuto, allora possiamo riscrivere l'algoritmo in modo da *elimiare* il parametro.

Dimostrazione "intuitiva". Dato l'indice i e fissato x, la funzione $\lambda z.\ \varphi_i(x, z)$ è certamente calcolabile (si prende la macchina i-esima e le si danno in input x e z). Per la Tesi di Church-Turing esiste allora un indice $s_{(i,x)}$ tale che

$$\varphi_{s_{(i,x)}} = \lambda z.\ \varphi_i(x, z).$$

Consideriamo allora la funzione $s : \mathbb{N}^2 \to \mathbb{N}$ definita da $(i, x) \mapsto s_{(i,x)}$: tale funzione è calcolabile totale poiché la procedura delineata è un algoritmo per ottenere $s_{(i,x)}$ a partire dagli input e tale algoritmo termina sempre.

Per rendere s iniettiva, scegliamo un ordinamento degli input (ad esempio l'ordinamento lessicografico) e per ogni input (i, x) controlliamo che $s(i, x) > s(i', x')$ per ogni $(i', x') < (i, x)$, ovvero che s sia strettamente crescente almeno fino a (i, x).

Se così non fosse, sostituiamo il valore di $s(i,x)$ con uno degli altri indici che calcola la stessa funzione e che sia maggiore di tutti gli $s(i',x')$: tale indice esiste sempre per il Padding Lemma.[a]

Segue che la s costruita in questo modo è strettamente crescente e dunque iniettiva. □

[a] Infatti esistono solo un numero finito di coppie (i',x') che siano *lessicograficamente minori* di (i,x), mentre esistono un numero infinito di indici che calcolano la stessa macchina dell'indice $s(i,x)$, dunque tra tutti questi infiniti indici ce ne sarà almeno uno più grande di tutti gli $s(i',x')$.

Osservazione 1.6.2. Il Teorema del Parametro ci dice che esiste una funzione calcolabile totale iniettiva a due variabili s tale che $\varphi_{s(i,x)} = \lambda y.\ \varphi_i(x,y)$ per ogni i,x. Allora fissato un indice i possiamo considerare direttamente la funzione ad una variabile

$$f := \lambda x.\ s(i,x),$$

che è ancora calcolabile totale, iniettiva, e tale che

$$\varphi_{f(x)}(y) = \varphi_i(x,y).$$

Il Teorema del Parametro ci permette di dimostrare il seguente teorema.

Teorema 1.6.7 – Teorema di Ricorsione di Kleene

Per ogni f calcolabile totale esiste un indice n tale che

$$\varphi_n = \varphi_{f(n)}.$$

Dimostrazione. Sia f una funzione calcolabile totale e costruiamo n tale che $\varphi_n = \varphi_{f(n)}$.

Consideriamo la funzione $\psi : \mathbb{N}^2 \to \mathbb{N}$ definita da

$$\psi(u,z) := \begin{cases} \varphi_{\varphi_u(u)}(z), & \text{se } \varphi_u(u)\downarrow, \\ \perp, & \text{altrimenti.} \end{cases} \tag{1.1}$$

Tale funzione è intuitivamente calcolabile: si prova a calcolare $\varphi_u(u)$; se non converge allora anche $\psi(u,z)$ non converge, mentre se converge possiamo recuperare la macchina di indice $\varphi_u(u)$ e calcolarla su z.

Segue quindi che esiste un indice i tale che $\varphi_i = \psi$. Per il Teorema del Parametro insieme all'Osservazione 1.6.2 possiamo considerare la funzione $d := \lambda u.\ s(i,u)$ calcolabile totale, iniettiva e tale che

$$\varphi_{d(u)}(z) = \varphi_i(u,z) = \psi(u,z). \tag{1.2}$$

Dato che f e d sono entrambe calcolabili totali, lo sarà anche la loro composizione $f \circ d$. In particolare $f \circ d$ sarà ancora iniettiva, ed esisterà un indice v tale che

$$\varphi_v = f \circ d. \tag{1.3}$$

Per totalità di φ_v, $\varphi_v(v)$ converge: segue che

$$\varphi_{d(v)} \overset{(1.2)}{=} \lambda z.\ \psi(v,z) \overset{(1.1)^2}{=} \lambda z.\ \varphi_{\varphi_v(v)}(z) = \varphi_{\varphi_v(v)}. \tag{1.4}$$

Sia allora $n := d(v)$ e mostriamo che vale la tesi. In effetti

$$\varphi_n = \varphi_{d(v)} \overset{(1.4)}{=} \varphi_{\varphi_v(v)} \overset{(1.3)}{=} \varphi_{f(d(v))} = \varphi_{f(n)},$$

come volevamo. □

Qui usiamo la definizione di ψ insieme al fatto che $\varphi_v(v)$ converge.

2

Calcolabilità di problemi

2.1 Problemi di decisione

Finora abbiamo studiato i vari formalismi per esprimere algoritmi e le loro caratteristiche principali, insieme ai diversi teoremi che ne seguono. Vogliamo ora studiare i *problemi* che possono essere risolti da una determinata classe di funzioni.

I nostri problemi sono **problemi di decisione**: dato un insieme $I \subseteq \mathbb{N}^k$ vogliamo stabilire se un dato elemento $x \in \mathbb{N}^k$ appartenga o no a I. In particolare ogni problema è identificato da un insieme.

Per parlare di *appartenenza* ad un insieme conviene definire due funzioni che saranno di grande rilevanza in seguito.

Definizione 2.1.1 – Funzione caratteristica e semicaratteristica di un insieme

Sia $I \subseteq \mathbb{N}^k$. Si dice **funzione caratteristica** di I la funzione $\chi_I : \mathbb{N}^k \to \{0,1\}$ definita da

$$\chi_I(x) := \begin{cases} 1, & \text{se } x \in I, \\ 0, & \text{se } x \notin I. \end{cases}$$

Si dice inoltre **funzione semicaratteristica** di I la funzione parziale $\tilde{\chi}_I : \mathbb{N}_k \to \{0,1\}$ definita da

$$\tilde{\chi}_I(x) := \begin{cases} 1, & \text{se } x \in I, \\ \bot, & \text{se } x \notin I. \end{cases}$$

Possiamo ora definire le due principali classi di problemi che analizzeremo.

Definizione 2.1.2 – Insiemi ricorsivi e ricorsivamente enumerabili

Sia $I \subseteq \mathbb{N}^k$.

- I si dice **ricorsivo** oppure **decidibile** se χ_I è calcolabile totale.
- I si dice **ricorsivamente enumerabile** (in breve r.e.) oppure **semidecidibile** se esiste un indice i tale che $I = \mathrm{dom}(\varphi_i)$.

Chiameremo $\mathcal{R}$ la classe degli insiemi ricorsivi, $\mathcal{RE}$ la classe degli insiemi ricorsivamente enumerabili.

Intuitivamente I è decidibile se è possibile *decidere* in tempo finito se un elemento appartiene o meno all'insieme. Per quanto riguarda gli insiemi semidecidibili, facciamo un'osservazione iniziale.

Osservazione 2.1.1 (-NoValue-). I è r.e. se e solo se $\tilde{\chi}_I$ è calcolabile (parziale).

Dimostrazione. Se $\tilde{\chi}_I$ è calcolabile, allora esiste un indice i con $\varphi_i = \tilde{\chi}_I$. In particolare $\mathrm{dom}(\varphi_i) = \mathrm{dom}(\tilde{\chi}_I) = I$, dunque I è r.e.

Viceversa, se I è r.e. esiste un indice i tale che $\mathrm{dom}(\varphi_i) = I$. Allora la funzione $\tilde{\chi}_I$ è calcolabile: dato x iniziamo a calcolare $\varphi_i(x)$; se il procedimento termina poniamo $\tilde{\chi}_I(x) = 1$, altrimenti continueremo all'infinto e quindi la computazione di $\tilde{\chi}_I$ non terminerà. □

Quindi un insieme I è semidecidibile se per ogni elemento $x \in I$ possiamo controllare in tempo finito l'appartenenza, mentre per gli elementi $x \notin I$ il procedimento non termina mai.

Il fatto che gli insiemi r.e. si chiamano proprio in questo modo deriva da un'altra particolare caratterizzazione.

Proposizione 2.1.3

I è r.e. se e solo se I è vuoto oppure esiste una funzione f calcolabile totale tale che $I = \mathrm{Im}(f)$.

Dimostrazione. □

Quali sono le relazioni tra insiemi ricorsivi e insiemi r.e.? Vediamone alcune che seguono immediatamente dalle definizioni.

Proposizione 2.1.4 – $\mathcal{R} \subseteq \mathcal{RE}$

Se I è ricorsivo, allora I è ricorsivamente enumerabile.

Dimostrazione. Infatti se I è ricorsivo la funzione $\tilde{\chi}_I$ è calcolabile: in effetti dato x, se $\chi_I(x) = 1$ allora $\tilde{\chi}_I(x) = 1$, altrimenti $\tilde{\chi}_I(x)$ è indefinito. Segue che I è r.e. poiché $I = \mathrm{dom}(\tilde{\chi}_I)$. □

Per la prossima proposizione abbiamo bisogno di definire il **complementare** di un problema.

Definizione 2.1.5 – Complementare di un problema

Dato un insieme I, il suo complementare $\overline{I}$ è definito da

$$\overline{I} := \{ x : x \notin I \}.$$

Proposizione 2.1.6 –

Se I e $\bar{I}$ sono entrambi r.e., allora sono entrambi ricorsivi.

Dimostrazione. Osserviamo che basta mostrare che I sia ricorsivo: a questo punto replicando il ragionamento su $\bar{I}$ e $\bar{\bar{I}} = I$ si ottiene che anche $\bar{I}$ è ricorsivo.
Per definizione di insieme r.e., esistono due indici i, j tali che

$$I = \mathrm{dom}(\varphi_i), \qquad \bar{I} = \mathrm{dom}(\varphi_j).$$

Per calcolare χ_I, dato x eseguiamo questa seguenza di passi:

- eseguiamo un passo di computazione di $\varphi_i(x)$: se converge (cioè $x \in \mathrm{dom}(\varphi_i) = I$) allora $\chi_I(x) = 1$, altrimenti continuiamo;
- eseguiamo un passo di computazione di $\varphi_j(x)$: se converge (cioè $x \in \mathrm{dom}(\varphi_j) = \bar{I}$ allora $\chi_I(x) = 0$, altrimenti continuiamo;
- eseguiamo due passi di computazione di $\varphi_i(x)$...

e così via. Ma x deve appartenere ad uno tra I e $\bar{I}$, dunque questo procedimento ad un certo punto termina. Segue che χ_I è calcolabile. □

2.2 Separazione di $\mathcal{R}$ e $\mathcal{RE}$

Dai risultati ottenuti nella sezione precedente potremmo essere indotti a sperare che tutti i problemi semidecidibili siano anche decidibili. Purtoppo ciò non è vero, e lo dimostriamo attraverso un particolare insieme, chiamato tradizionalmente K:

$$K := \left\{ n \,:\, \varphi_n(n)\downarrow \right\} \tag{2.1}$$

Teorema 2.2.1

K è r.e. ma non è ricorsivo.

Per chiarezza dividiamo in due parti la dimostrazione.

Dimostrazione che K è r.e. Per quanto detto precedentemente, per mostrare che K è r.e. basta far vedere che la sua funzione semicaratteristica

$$\tilde{\chi}_K := n \mapsto \begin{cases} 1, & \text{se } \varphi_n(n)\downarrow \\ \perp, & \text{altrimenti} \end{cases}$$

è calcolabile. Ma questa funzione è intuitivamente calcolabile:

- si esegue un passo del calolo di $\varphi_0(0)$: se converge allora $\tilde{\chi}_K(0)$ vale 1, altrimenti si continua;

- si esegue un passo del calcolo di $\varphi_1(1)$: se converge allora $\tilde{\chi}_K(0)$ vale 1, altrimenti si continua;
- si eseguono due passi del calcolo di $\varphi_0(0)$...

e così via. Questo procedimento ad un certo punto termina per tutti i valori di n che sono in K, e per gli altri invece non termina mai. □

Dimostrazione che K non è ricorsivo. Supponiamo per assurdo che K sia ricorsivo: per definizione allora χ_K è calcolabile totale. Definiamo allora $f : \mathbb{N} \to \mathbb{N}$ data da

$$f(n) := \begin{cases} \varphi_n(n) + 1, & \text{se } \chi_K(n) = 1 \\ 0, & \text{altrimenti.} \end{cases}$$

Dato che χ_K è calcolabile totale possiamo calcolare $\chi_K(n)$; inoltre se $\chi_K(n) = 1$ (ovvero $n \in K$) per definizione di K si ha che $\varphi_n(n)$ converge.

Per la Tesi di Church-Turing allora esiste un indice i tale che $\varphi_i = f$,e quindi in particolare $\varphi_i(i) = f(i)$. Ma ciò è assurdo:

- se $\varphi_i(i)$ converge allora $f(i) = \varphi_i(i) + 1 \neq \varphi_i(i)$;
- se $\varphi_i(i)$ diverge allora $f(i)$ converge (a 0).

Segue in particolare che K non può essere ricorsivo. □

▸ Piccola parentesi sul *bootstrapping*

L'insieme K può risultare abbastanza contorto e quindi è facile farsi venire l'idea che la dimostrazione funzioni solo perché abbiamo scelto un insieme "innaturale". In realtà l'auto-applicazione non è strana, e compare anche nel mondo reale, ad esempio quando si parla di compilatori.

Infatti dato un compilatore $C_L^{L\to A}$ scritto nel linguaggio L e che compila codice L (alto livello) in codice scritto nel linguaggio A (più a basso livello), potrebbe essere necessario dover usare il compilatore in una macchina che nativamente non sa far girare il codice L, ma sa far girare solo codice nel linguaggio A. Vogliamo perciò un modo efficiente per ottenere un compilatore $L \to A$ scritto in A.

Il metodo più semplice è detto **bootstrapping**: in pratica si dà in pasto al compilatore $C_L^{L\to A}$ il suo stesso codice, ovvero si dà il comando $C_L^{L\to A}\left(C_L^{L\to A}\right)$. Il risultato è un codice scritto in A che mantiene la semantica originale, ovvero è un compilatore $C_A^{L\to A}$.

Tuttavia il bootstrapping non è l'unico motivo per cui il problema K è importante anche da un punto di vista applicativo: K è strettamente legato al **problema della fermata**, che ora definiremo formalmente.

Definizione 2.2.2 – Problema della fermata

Dato un indice i e un input x, dire se $\varphi_i(x)$ converge.

Come ogni problema decisionale, il problema della fermata può essere rappresentato tramite un insieme. In questo caso l'insieme è indicato con

$$K_0 := \left\{ (i, x) \ : \ \varphi_i(x)\downarrow \right\}.$$

Questo problema è intuitivamente molto importante: se fosse decidibile, potremmo decidere in tempo finito se una funzione φ_i si arresta sull'input x oppure continua in eterno.

Teorema 2.2.3 – K_0 non è ricorsivo

L'insieme K_0 non è ricorsivo.

Dimostrazione. Osserviamo che $(x, x) \in K_0$ se e solo se $x \in K$: se K_0 fosse decidibile lo sarebbe anche K, ma ciò è assurdo. □

2.3 Riduzioni di classi di problemi

Nella dimostrazione del Teorema 2.2.3 abbiamo sfruttato una tecnica comune in matematica: abbiamo *ridotto* il problema K_0 al problema K e in questo modo abbiamo dimostrato che K_0 non può essere decidibile.

Vogliamo ora generalizzare questo concetto.

Definizione 2.3.1 – Riduzione secondo una funzione

Dati due problemi A, B, si dice che A **si riduce secondo** f a B (e si scrive $A \leq_f B$) se

$$x \in A \iff f(x) \in B.$$

Osservazione 2.3.1 (-NoValue-). $K \leq_f K_0$ secondo la funzione $f : \mathbb{N} \to \mathbb{N}^2$ definita da $f(x) := (x, x)$.

Osservazione 2.3.2 (-NoValue-). $A \leq_f B$ se e solo se $\overline{A} \leq_f \overline{B}$.

Spesso non ci interessa quale sia la funzione che permette la riduzione di A a B, ma solo a quale classe di funzioni appartiene.

Definizione 2.3.2 – Riduzione secondo una classe di funzioni

Siano A, B problemi, $\mathcal{F}$ insieme di funzioni. Allora si dice che A **si riduce secondo** $\mathcal{F}$ a B (e si scrive $A \leq_{\mathcal{F}} B$, o anche $A \leq B$ se l'insieme $\mathcal{F}$ è deducibile dal contesto) se esiste una $f \in \mathcal{F}$ tale che $A \leq_f B$.

Come scegliamo l'insieme $\mathcal{F}$? In generale dipende dalle classi di problemi che vogliamo confrontare: infatti per poter studiare queste classi è importante che la riduzione rispetti alcune proprietà.

Definizione 2.3.3 – Riduzione che classifica due classi

Date $\mathcal{D}, \mathcal{E}$ classi di problemi con $\mathcal{D} \subseteq \mathcal{E}$, $\mathcal{F}$ insieme di funzioni, si dice che la relazione $\leq_{\mathcal{F}}$ **classifica** le classi $\mathcal{D}, \mathcal{E}$ se

1. $A \leq_{\mathcal{F}} A$,
2. se $A \leq_{\mathcal{F}} B$ e $B \leq_{\mathcal{F}} C$, allora $A \leq_{\mathcal{F}} C$,
3. se $A \leq_{\mathcal{F}} B$ e $B \in \mathcal{D}$, allora $A \in \mathcal{D}$,
4. se $A \leq_{\mathcal{F}} B$ e $B \in \mathcal{E}$, allora $A \in \mathcal{E}$.

Osservazione 2.3.3 (-NoValue-). Sfruttando la definizione di riduzione, possiamo riscrivere le proprietà in forma *algebrica*:

1. $\mathrm{id}_A \in \mathcal{F}$,
2. se f e g appartengono a $\mathcal{F}$, allora anche la composizione gf appartiene ad $\mathcal{F}$,
3. se $f \in \mathcal{F}$ e $B \in \mathcal{D}$, allora $f^{-1}[B] \in \mathcal{D}$,
4. se $f \in \mathcal{F}$ e $B \in \mathcal{E}$, allora $f^{-1}[B] \in \mathcal{E}$.

Questo in particolare mi dice che se $\leq_{\mathcal{F}}$ classifica $\mathcal{D}, \mathcal{E}$, allora $\leq_{\mathcal{F}}$ è un **preordine** su $\mathcal{D}$ e su $\mathcal{E}$.

▸ Intuizione Qual è il significato intuitivo di una relazione che classifica due classi? Possiamo leggere $\leq_{\mathcal{F}}$ come "è al più difficile quanto":

- A è al più difficile quanto se stesso,
- se A è al più difficile quanto B e B è al più difficile quanto C, allora A è al più difficile quanto C,
- i problemi difficili al più quanto B si trovano tutti nel più piccolo insieme contenente B: se A è al più difficile quanto B e B appartiene a $\mathcal{D}$ (risp. $\mathcal{E}$), allora anche A appartiene a $\mathcal{D}$ (risp. $\mathcal{E}$), cioè A **non sta fuori** $\mathcal{D}$ **(risp.** $\mathcal{E}$**)**.

Fissiamo ora due classi $\mathcal{D}, \mathcal{E}$ con $\mathcal{D} \subseteq \mathcal{E}$ e una classe di funzioni $\mathcal{F}$.

Definizione 2.3.4 – Grado di un problema

Definiamo il **grado** di un problema A è

$$\deg A := \{ B : A \leq_{\mathcal{F}} B \text{ e } B \leq_{\mathcal{F}} A \},$$

ovvero è l'insieme di tutti i problemi con la stessa difficoltà di A.

Definizione 2.3.5 – Problemi ardui e completi

Sia H un problema.

- H si dice **$\leq_{\mathcal{F}}$-arduo per $\mathcal{E}$** se per ogni $A \in \mathcal{E}$ si ha $A \leq_{\mathcal{F}} H$, ovvero se H è almeno difficile quanto tutti i problemi di $\mathcal{E}$.
- H si dice **$\leq_{\mathcal{F}}$-completo per $\mathcal{E}$** se è $\leq_{\mathcal{F}}$-arduo per $\mathcal{E}$ e $H \in \mathcal{E}$.

In particolare possiamo studiare la relazioni tra $\mathcal{D}$ e $\mathcal{E}$ tramite i problemi $\leq_{\mathcal{F}}$-ardui/completi per $\mathcal{E}$.

Proposizione 2.3.6

Se C è $\leq_{\mathcal{F}}$-completo per $\mathcal{E}$ e $C \in \mathcal{D}$, allora $\mathcal{E} = \mathcal{D}$.

Dimostrazione. Dato che $\mathcal{D} \subseteq \mathcal{E}$, basta mostrare che $\mathcal{E} \subseteq \mathcal{D}$. Sia allora $B \in \mathcal{E}$: dato che C è $\leq_{\mathcal{F}}$-completo per $\mathcal{E}$ segue che $B \leq_{\mathcal{F}} C$. Ma $C \in \mathcal{D}$, dunque anche $B \in \mathcal{D}$. □

Proposizione 2.3.7

Se A è $\leq_{\mathcal{F}}$-arduo per $\mathcal{E}$ e $A \leq_{\mathcal{F}} B$, allora B è $\leq_{\mathcal{F}}$-arduo per $\mathcal{E}$. In particolare se $B \in \mathbb{E}$ allora A, B sono $\leq_{\mathcal{F}}$-completi per $\mathcal{E}$.

Dimostrazione. Se $H \in \mathcal{E}$. Dato che A è $\leq_{\mathcal{F}}$-arduo per $\mathcal{E}$ si ha $H \leq_{\mathcal{F}} A$; inoltre $A \leq_{\mathcal{F}} B$ dunque per transitività $H \leq_{\mathcal{F}} B$, ovvero B è $\leq_{\mathcal{F}}$-arduo.

Inoltre se $B \in \mathcal{E}$ segue anche che A lo è, poiché $A \leq_{\mathcal{F}} B$. Allora per definizione A, B sono $\leq_{\mathcal{F}}$-completi per $\mathcal{E}$. □

2.4 Studio di $\mathcal{R}$ e $\mathcal{RE}$ tramite riduzioni

Vogliamo studiare ora le classi di problemi $\mathcal{R}$ e $\mathcal{RE}$ tramite riduzioni: come prima cosa dobbiamo identificare una classe di funzioni.

Definizione 2.4.1 – Classe `rec`

Indicheremo con

$$\texttt{rec} := \{ \varphi_i \ : \ \mathrm{dom}(\varphi_i) = \mathbb{N} \}$$

la classe di tutte le funzioni calcolabili totali, che quindi chiameremo (per motivi storici) **ricorsive**.

Proposizione 2.4.2

$\leq_{\texttt{rec}}$ classifica $\mathcal{R} \subseteq \mathcal{RE}$.

Dimostrazione. Basta mostrare le quattro condizioni date dalla definizione. In particolare lo faremo attraverso le condizioni algebriche equivalenti.

1. L'identità è ricorsiva.

2. Se f, g sono ricorsive, allora anche la loro composizione gf lo è.

3. Supponiamo $A \leq_{rec} B$ con $B \in \mathcal{R}$ (cioè χ_B è ricorsiva). Vogliamo dimostrare che $A \in \mathcal{R}$, cioè che χ_A è ricorsiva.

   Per definizione di $\leq_{rec}$ esiste una funzione $f \in \texttt{rec}$ con $A \leq_f B$, cioè $x \in A$ se e solo se $f(x) \in B$, ovvero $\chi_A(x) = 1$ se e solo se $\chi_B(f(x)) = \chi_B \circ f(x) = 1$. Ma allora $\chi_A = \chi_B f$ e dunque χ_A è ricorsiva poiché composizione di funzioni ricorsive.

4. Supponiamo $A \leq_{rec} B$ con $B \in \mathcal{RE}$, ovvero con $\tilde{\chi}_B$ calcolabile. Vogliamo dimostrare che A è r.e., cioè che $\tilde{\chi}_A$ è calcolabile.

   Per definizione di $\leq_{rec}$ esiste una funzione $f \in \texttt{rec}$ con $A \leq_f B$, ovvero $x \in A$ se e solo se $f(x) \in B$. Mostriamo che $\tilde{\chi}_A = \tilde{\chi}_B f$:

   - se $x \in A$ (e quindi $\tilde{\chi}_A(x) = 1$) allora $f(x) \in B$ e quindi $\tilde{\chi}_B(f(x)) = \tilde{\chi}_B f(x) = 1$;
   - se $x \notin A$ (cioè $\tilde{\chi}_A(x)$ diverge) allora $f(x) \notin B$, e quindi $\tilde{\chi}_B(f(x)) = \tilde{\chi}_B \circ f(x)$ diverge.

   Segue che $\tilde{\chi}_A = \tilde{\chi}_B f$. In particolare $\tilde{\chi}_A$ è calcolabile, poiché composizione di funzioni calcolabili. □

Dunque in questa sezione diremo che un problema è arduo/completo per $\mathcal{RE}$ sottointendendo la relazione $\leq_{rec}$.

Osservazione 2.4.1 (-NoValue-). Osserviamo che $\overline{K} \not\leq_{rec} K$ e $K \not\leq_{rec} \overline{K}$:

- se per assurdo $\overline{K} \leq_{rec} K$, allora $\overline{K}$ sarebbe r.e. (perché K lo è), ma questo implicherebbe (per la Proposizione 2.1.6) che sia K che $\overline{K}$ sono ricorisvi, il che è assurdo (poiché K non è ricorsivo);
- come osservato in precedenza, $A \leq_{\mathcal{F}} B$ se e solo se $\overline{A} \leq_{\mathcal{F}} \overline{B}$, dunque $K \leq_{rec} \overline{K}$ se e solo se $\overline{K} \leq_{rec} K$, che abbiamo dimostrato essere falso.

▸ Problemi co-$\mathcal{RE}$ I problemi tali che i loro complementari sono in $\mathcal{RE}$ formano una classe chiamata co-$\mathcal{RE}$. Dato che ogni problema ricorsivo ha anche un complementare ricorsivo, $\mathcal{R} \subseteq$ co-$\mathcal{RE}$; tuttavia esistono anche problemi al di fuori di co-$\mathcal{RE}$.

Per studiare le relazioni tra gli insiemi $\mathcal{R}$ e $\mathcal{RE}$ vorremo trovare un problema completo per $\mathcal{RE}$.

Teorema 2.4.3

K è completo per $\mathcal{RE}$.

Dimostrazione. Dato che K è r.e., basta dimostrare che K è arduo per $\mathcal{RE}$. Sia allora $A \in \mathcal{RE}$, ovvero tale che esista un indice i tale che $\mathrm{dom}(\varphi_i) = A$, ovvero $A = \{x : \varphi_i(x)\downarrow\}$.

Definiamo allora la funzione $\psi : \mathbb{N}^2 \to \mathbb{N}$ data da $\psi(x, y) := \varphi_i(x)$. (Il parametro y non fa niente.) Dato che ψ è calcolabile dovrà esistere un indice j con $\psi = \varphi_j$. In particolare $A = \{x : \varphi_j(x, y)\downarrow\}$.

Per il Teorema del Parametro e l'Osservazione 1.6.2, possiamo considerare la funzione $f := \lambda x.\, s(j, x)$: il Teorema ci garantisce che $\varphi_j(x, y) = \varphi_{f(x)}(y)$ per ogni x, y. Osserviamo in particolare che f è calcolabile totale.

Ma y non ha un effetto sulla computazione ($\varphi_j(x, y) = \varphi_j(x, y')$ per ogni y, y'), ovvero $\varphi_{f(x)}$ è **costante**. Segue che

$$A = \{ x : \varphi_{f(x)}(y)\downarrow \} = \{ x : \varphi_{f(x)}(f(x))\downarrow \} = \{ x : f(x) \in K \},$$

ovvero $A \leq_f K$. Dato che $f \in \texttt{rec}$, segue la tesi. □

2.4.1 Altri problemi completi per $\mathcal{RE}$

Il fatto che $K \leq_{\texttt{rec}} K_0$ mostra (per il Proposizione 2.3.7) che anche K_0 è completo per $\mathcal{RE}$. Cerchiamo altri problemi completi per $\mathcal{RE}$.

▸ TOT è completo per $\mathcal{RE}$

Consideriamo il problema

$$\texttt{TOT} := \{ i : \mathrm{dom}(\varphi_i) = \mathbb{N} \} = \{ i : \varphi_i \in \texttt{rec} \}.$$

Proposizione 2.4.4 –

Vale la riduzione

$$K \leq_{\texttt{rec}} \texttt{TOT}.$$

In particolare TOT è completo per $\mathcal{RE}$.

Dimostrazione. Consideriamo la funzione $\psi : \mathbb{N}^2 \to \mathbb{N}$ definita da

$$\psi(x, y) := \begin{cases} 1, & \text{se } x \in K \\ \bot, & \text{se } x \notin K. \end{cases}$$

Dato che K è semidecidibile ψ è calcolabile: basta calcolare $\tilde{\chi}_K(x)$ (che è calcolabile); se la computazione termina (resitutendo 1) poniamo $\psi(x, y) = 1$, altrimenti la computazione di $\tilde{\chi}_K(x)$ non termina e quindi anche $\psi(x, y)$ diverge.

Siccome ψ è calcolabile esisterà un indice i tale che $\varphi_i = \psi$. Per il Teorema del Parametro (insieme all'Osservazione 1.6.2) esisterà f calcolabile totale, $f := \lambda x.\, s(i, x)$ tale che

$$\varphi_i(x, y) = \varphi_{f(x)}(y).$$

Ora abbiamo due casi:

- se $x \in K$ allora per ogni y si ha

  $$\varphi_{f(x)}(y) = \psi(x, y) = 1,$$

  dunque $f(x)$ è indice di una funzione calcolabile totale, ovvero $f(x) \in \texttt{TOT}$;

- se $x \notin K$ allora per ogni y si ha

$$\varphi_{f(x)}(y) = \psi(x, y) = \perp,$$

dunque $f(x)$ è non è indice di una funzione calcolabile totale (è sempre indefinita!) e quindi $f(x) \notin \texttt{TOT}$.

Segue che $x \in K$ se e solo se $f(x) \in \texttt{TOT}$, ovvero (dato che f è calcolabile totale) $K \leq_{rec} \texttt{TOT}$. □

2.5 Il Teorema di Rice

Alla fine dell'ultima sezione abbiamo mostrato come TOT sia un altro esempio di insieme non ricorsivo. Per costruire TOT abbiamo sfruttato un'idea particolarmente semplice: abbiamo preso tutti e soli gli *indici* di tutte le funzioni calcolabili totali. Questo approccio ci consente di definire diversi altri insiemi interessanti:

$$\begin{aligned}
&\texttt{FIN} := \{\, i : \mathrm{dom}(\varphi_i) \text{ è finito } \} \\
&\texttt{INF} := \{\, i : \mathrm{dom}(\varphi_i) \text{ è infinito } \} = \overline{\texttt{FIN}} \\
&\texttt{REC} := \{\, i : \mathrm{dom}(\varphi_i) \text{ è ricorsivo } \} \\
&\texttt{CONST} := \{\, i : \varphi_i \text{ è costante e totale } \} \\
&\texttt{EXT} := \{\, i : \varphi_i \text{ è estendibile ad una funzione calc. tot. } \}
\end{aligned} \tag{2.2}$$

Possiamo studiare più velocemente tali insiemi?

Definizione 2.5.1 – Insiemi di indici che rappresentano funzioni

Sia $I \subseteq \mathbb{N}$ un insieme di indici. I si dice **insieme di indici che rappresentano funzioni** (**iirf** per gli amici) se per ogni x, y vale che

$$x \in I \text{ e } \varphi_y = \varphi_x \implies y \in I.$$

In pratica un insieme di indici è un iirf se quando contiene un indice i allora contiene tutti gli indici che rappresentano φ_i (che sappiamo essere infiniti per il Padding Lemma).

L'importanza di questo tipo di insiemi di indici è che ci consentono di collegare sintassi e semantica: se riusciamo a dimostrare che un iirf gode di una determinata proprietà, allora le *funzioni rappresentate dall'insieme* godono di tale proprietà.

Proposizione 2.5.2 –

Se I è un iirf, allora $\overline{I}$ è un iirf.

Dimostrazione. Supponiamo per assurdo che $\overline{I}$ non sia un iirf: dovranno esistere x, y tali che $x \in \overline{I}$, $\varphi_x = \varphi_y$ ma $y \notin \overline{I}$, ovvero $y \in I$. Ma I è un iirf, dunque anche x deve appartenere ad I: questo è assurdo poiché abbiamo supposto appartenesse al suo complementare. Segue la tesi. □

Teorema 2.5.3 – Index Set Theorem

Sia I un iirf, $I \neq \emptyset$, $I \neq \mathbb{N}$. Allora $K \leq_{rec} I$ oppure $K \leq_{rec} \overline{I}$.

Dimostrazione. Sia i_0 tale che φ_{i_0} sia la funzione sempre indefinita e supponiamo senza perdita di generalità $i_0 \in \overline{I}$.[a] Siccome I è non vuoto dovrà esistere un $i_1 \in I$; inoltre dato che I è un iirf certamente $\varphi_{i_1} \neq \varphi_{i_0}$ (altrimenti $i_0 \in I$), dunque φ_{i_1} non è ovunque indefinita.

Definiamo allora la seguente funzione:

$$\psi(x, y) := \begin{cases} \varphi_{i_1}(y), & \text{se } x \in K \\ \perp, & \text{altrimenti.} \end{cases}$$

Questa funzione è intuitivamente calcolabile: se $x \in K$ (possiamo verificarlo in tempo finito siccome K è semidecidibile) allora basta prendere la funzione i_1-esima della numerazione e calcolarla su y; altrimenti non riusciremo mai a verificare che $x \notin K$ e dunque ψ sarà indefinita.

Per la Tesi di Church-Turing, insieme al Teorema del Parametro e l'Osservazione 1.6.2 possiamo allora considerare la funzione f calcolabile totale tale che

$$\varphi_{f(x)}(y) = \psi(x, y).$$

Mostriamo che $K \leq_f I$:

- se $x \in K$ allora $\varphi_{f(x)} = \varphi_{i_1}$, ma dato che $i_1 \in I$ e quest'ultimo è un iirf segue che $f(x) \in I$;
- se $x \notin K$ allora $\varphi_{f(x)}$ è sempre indefinita, ovvero $\varphi_{f(x)} = \varphi_{i_0}$. Ma dato che per la Proposizione 2.5.2 anche $\overline{I}$ è un iirf segue che $f(x) \in \overline{I}$, ovvero $f(x)$ non appartiene ad I.

Segue la tesi. □

[a] Se i_0 appartenesse a I potremmo scambiare I e $\overline{I}$ e alla fine dimostreremmo che $K \leq_{rec} \overline{I}$.

L'Index Set Theorem ci permette di dimostrare uno dei risultati più importanti della Teoria della Calcolabilità, che è il **Teorema di Rice**.

Teorema 2.5.4 – Teorema di Rice

Sia $\mathcal{F}$ un insieme di funzioni calcolabili, $I_{\mathcal{F}} := \{ i : \varphi_i \in \mathcal{F} \}$ l'insieme degli indici delle funzioni di $\mathcal{F}$.

Allora $I_{\mathcal{F}}$ è ricorsivo se e solo se $\mathcal{F}$ è vuoto oppure contiene tutte le funzioni calcolabili.

Dimostrazione. Mostriamo entrambi i versi dell'implicazione.

$\Longleftarrow$ Ovvio: se $\mathcal{F}$ è vuoto allora $I_{\mathcal{F}}$ è vuoto, se contiene tutte le funzioni calcolabili allora $I_{\mathcal{F}} = \mathbb{N}$ e entrambi questi insiemi sono ricorsivi.[a]

$\Longrightarrow$ Se $\mathcal{F}$ è non vuoto e non contiene tutte le funzioni calcolabili, allora $I_{\mathcal{F}}$ è un insiemi

di indici strettamente compreso tra $\varnothing$ e $\mathbb{N}$. Per l'Index Set Theorem segue che $K \leq_{rec} I_{\mathcal{F}}$ o al suo complemento, e dunque $I_{\mathcal{F}}$ non è ricorsivo.[b] □

[a] Le loro funzioni caratteristiche sono rispettivamente la funzione costante 0 e la funzione costante 1, che sono ovviamente calcolabili.

[b] Se $K \leq_{rec} \overline{I_{\mathcal{F}}}$ allora $\overline{I_{\mathcal{F}}}$ è non ricorsivo, ma un insieme è ricorsivo se e solo se lo è il suo complemento, dunque neanche $I_{\mathcal{F}}$ lo è.

Il Teorema di Rice ci dice in breve che, data una **qualsiasi proprietà semantica** non banale di un programma, il problema di decidere se una funzione soddisfi o no quella proprietà **non è decidibile**. In particolare tutti gli insiemi definiti in (2.2) non sono ricorsivi.

In realtà **nessuno** di quegli insiemi è ricorsivamente enumerabile: possiamo mostrare che $\overline{K}$ si riduce ad ognuno di essi, e sappiamo già che $\overline{K}$ non è r.e. Vediamolo per esercizio su `FIN` e `INF`.

▸ `FIN` non è ricorsivamente enumerabile

Mostriamo che $\overline{K} \leq_{rec}$ `FIN`.

Dimostrazione. Consideriamo la funzione

$$\psi(x, y) := \begin{cases} 1, & \text{se } x \in K \\ \perp, & \text{altrimenti.} \end{cases}$$

Questa funzione è intuitivamente calcolabile: siccome K è semidecidibile possiamo decidere in tempo finito che x appartiene a K, e in tal caso poniamo $\psi(x, y) = 1$; altrimenti non sapremo mai se x appartiene a K e la funzione ψ divergerà.

Per la Tesi di Church-Turing, il Teorema del Parametro e l'Osservazione 1.6.2 esiste una funzione f calcolabile totale tale che

$$\varphi_{f(x)}(y) = \psi(x, y).$$

Mostriamo che $\overline{K} \leq_f$ `FIN`:

- se $x \in \overline{K}$, ovvero x non appartiene a K, allora $\varphi_{f(x)}$ è sempre indefinita. In particolare il suo dominio è vuoto e quindi è finito, dunque $f(x) \in$ `FIN`;
- se $x \notin \overline{K}$ si ha che $x \in K$, dunque $\varphi_{f(x)}$ è costantemente 1, dunque il suo dominio è tutto $\mathbb{N}$ e quindi $f(x)$ non appartiene a `FIN`.

Segue la tesi. □

▸ `INF` non è ricorsivamente enumerabile

Mostriamo che $\overline{K} \leq_{rec}$ `INF`.

Dimostrazione. Consideriamo la funzione

$$\psi(x, y) := \begin{cases} 1, & \text{se } \varphi_x(x) \text{ non converge in meno di } y \text{ passi} \\ \perp, & \text{altrimenti.} \end{cases}$$

Questa funzione è intuitivamente calcolabile: proviamo a calcolare $\varphi_x(x)$; se in y passi ancora non ha terminato la sua computazione poniamo $\psi(x, y) = 1$, altrimenti la poniamo indefinita.

Per la Tesi di Church-Turing, il Teorema del Parametro e l'Osservazione 1.6.2 esiste una funzione f calcolabile totale tale che

$$\varphi_{f(x)}(y) = \psi(x, y).$$

Mostriamo che $\overline{K} \leq_f$ `INF`:

- se $x \in \overline{K}$, ovvero x non appartiene a K, allora $\varphi_x(x)$ non convergerà mai e dunque $\psi(x, y)$ sarà 1 per ogni y. In particolare $\varphi_{f(x)}$ è costantemente 1, dunque il suo dominio è tutto $\mathbb{N}$, dunque $f(x) \in$ `INF`;
- se $x \notin \overline{K}$ si ha che $x \in K$, dunque dovrà esistere un $\bar{y} \in \mathbb{N}$ tale che $\varphi_x(x)$ converge in $\bar{y}$ passi. Ma allora

  $$\psi(x, y) = \varphi_{f(x)}(y) \text{ diverge per ogni } y \geq \bar{y}$$

  dunque il dominio di $\psi(x, y)$ è al più l'insieme $\{0, \ldots, \bar{y}\}$, dunque è finito. Segue che $f(x)$ non appartiene a `INF`.

Segue la tesi. □

Osservazione 2.5.1 (-NoValue-). Siccome `FIN` e `INF` sono l'uno il complementare dell'altro e sono entrambi non r.e., sono anche **non co-**$\mathcal{RE}$.

3

Teoria della Complessità

Nella prima parte del corso abbiamo studiato la risolubilità di problemi: in particolare abbiamo studiato come, sotto determinate ipotesi, esistono problemi decidibili, problemi semidecidibili e problemi ancora più difficili. Abbiamo visto che esiste una gerarchia rispetto alla *difficoltà* e abbiamo studiato una parte di questa gerarchia, ovvero le classi $\mathcal{R}$ e $\mathcal{RE}$.

Vogliamo ora chiederci *come* si calcola e *quante risorse* siano necessarie: nella realtà non abbiamo a disposizione memoria infinita o tempo illimitato, e perciò vogliamo sapere quali problemi possono essere risolti data la quantità di risorse che abbiamo a disposizione.

Iniziamo a introdurre dei vincoli sulla nostra teoria. Innanzitutto considereremo come risorse solamente il *tempo* e lo *spazio*, anche se queste non sono le uniche risorse esistenti. Potremmo infatti studiare anche il numero di messaggi scambiati, la banda di rete utilizzata, l'energia consumata, eccetera.

Nel far ciò ci scontriamo immediatamente con una difficoltà: per i problemi non decidibili è possibile che la valutazione di un'istanza del problema (ovvero, decidere se un elemento appartiene o no all'insieme che codifica il problema) diverga, e quindi nessuna quantità finita di risorse è sufficiente. Per questo motivo ci limiteremo da ora in poi a studiare *problemi decidibili*.

A questo punto siamo interessati a calcolare quante risorse sono necessarie in termini della *grandezza dell'input*. Per far ciò insieme ad ogni problema P e al suo insieme di input $I \supseteq P$[1]consideriamo una funzione

$$|\cdot| : I \to \mathbb{N}$$

detta **taglia**, che associa ad ogni $x \in I$ la sua "grandezza", indicata con $|x|$.

Tipici esempi di taglie sono il numero di bit nei problemi che hanno come input numeri, oppure il numero di elementi del vettore in input se il problema ha come input vettori, o il numero di nodi se l'input è un grafo, eccetera. Non ci occuperemo di studiare le taglie associate a problemi classici, anche perché spesso sono molto intuitive.

Dato un problema P vogliamo quindi misurare la complessità di ogni singola *istanza* del problema, ed in particolare vogliamo misurare la complessità del **caso pessimo**. Per far ciò possiamo cercare una funzione

$$f : \mathbb{N} \to \mathbb{N}$$

che, data la taglia di un input $x \in I$, maggiora il numero di risorse necessarie alla macchina scelta per risolvere l'istanza. Siccome siamo interessati principalmente a stime asintotiche e ai casi

[1]Un problema P può essere considerato come un insieme, e la domanda è stabilire se un dato elemento di input $x \in I$ appartenga o no a P: in particolare ogni elemento di P può essere scelto come input, e quindi $P \subseteq I$.

pessimi, vogliamo inoltre che se $|x| < |y|$ allora $f(|x|) \leq f(|y|)$. In casi particolarmente fortunati riusciremo a trovare la *minima* funzione che maggiora la quantità di risorse necessarie.

Osserviamo ora che se un problema richiede al più k risorse, allora qualsiasi sistema con $k' \geq k$ risorse può risolvere il problema dato: possiamo quindi creare una gerarchia di classi di problemi, e per studiare queste classi useremo gli stessi strumenti visti nella prima parte, ovvero le riduzioni e i problemi completi.

Tuttavia la gerarchia tra classi dipende a priori dal contesto, ovvero da come misuriamo il consumo di risorse e anche dal modello di computazione scelto. Per questo richiediamo infine che la nostra teoria sia *invariante* rispetto sia ai modelli di calcolo, sia rispetto alla rappresentazione dei dati.

Sotto tutte queste ipotesi in questa parte del corso riusciremo a descrivere un piccolo frammento della gerarchia delle classi di complessità:

$$\texttt{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}\texttt{SPACE} = \mathcal{NP}\texttt{SPACE} \subseteq \mathcal{R} \subseteq \mathcal{RE}.$$

L'interesse moderno per questa gerarchia è che abbiamo dimostrato che $\texttt{LOGSPACE} \subsetneq \mathcal{P}\texttt{SPACE}$, ovvero abbiamo trovato un problema in $\mathcal{P}\texttt{SPACE}$ che non è in $\texttt{LOGSPACE}$. Questo significa che almeno una delle inclusioni

$$\texttt{LOGSPACE} \subseteq \mathcal{P}, \qquad \mathcal{P} \subseteq \mathcal{NP}, \qquad \mathcal{NP} \subseteq \mathcal{P}\texttt{SPACE}$$

deve essere stretta, tuttavia ancora non sappiamo quale.

3.1 Complessità deterministica

Vogliamo studiare la complessità in tempo e spazio usando come modello base di *calcolatore* le Macchine di Turing. Come prima cosa, facciamo una piccola estensione delle MdT normali.

Definizione 3.1.1 – Macchina di Turing a k nastri

Una **Macchina di Turing a k nastri** è una quintupla $M = (Q, \Sigma, \delta, q_0)$ tale che

(1) $\#, \triangleright \in \Sigma$, mentre $\texttt{L}, \texttt{R}, - \notin \Sigma$,

(2) $\texttt{SÌ}, \texttt{NO}$ sono due stati speciali che non appartengono a Q,

(3) la funzione di transizione ha la forma

$$\delta : Q \times \Sigma^k \to (Q \cup \{\texttt{SÌ}, \texttt{NO}\}) \times (\Sigma \times \{\texttt{L}, \texttt{R}, -\})^k$$

e inoltre valgono le altre condizioni delle MdT, opportunamente riscritte.

Una MdT a k nastri quindi è una normale MdT che può leggere contemporaneamente k simboli diversi e la cui funzione di transizione lavora considerando lo stato corrente e tutti i k simboli letti.

In particolare una configurazione ha la forma

$$(q, u_1 \underline{\sigma_1} v_1, u_2 \underline{\sigma_2} v_2, \ldots, u_k \underline{\sigma_k} v_k).$$

Una macchina a k nastri può essere quindi pensata come una macchina a k *processori*: ogni nastro contiene dell'informazione, e i vari nastri possono eseguire calcoli aiutandosi l'un l'altro oppure ognuno per i fatti suoi.

Perché non abbiamo diviso lo stato in k-uple? Perché in generale non cambia niente: possiamo immaginare l'insieme Q composto da k-uple e ogni coordinata della k-upla rappresenta lo stato del singolo nastro.

Introduciamo le macchine a k nastri formalmente solo ora perché da un punto di vista della calcolabilità non apportano nessun miglioramento: un problema è decidibile/semidecidibile da una macchina a k nastri se e solo se è decidibile/semidecidibile da una macchina standard ad 1 nastro.

Definizione 3.1.2 – Tempo deterministico richiesto

Sia M una MdT a k nastri che risolve il problema I. Diremo che $t \in \mathbb{N}$ è **tempo deterministico richiesto** per decidere l'istanza $x \in I$ se

$$(q_0, \trianglerighteq x, \trianglerighteq, \dots, \trianglerighteq) \to^t (h, u_1, u_2, \dots, u_t)$$

con $h \in \{\texttt{SÌ}, \texttt{NO}\}$.

Il *deterministico* nella definizione precedente evidenzia il fatto che la MdT considerata è deterministica, ovvero che la sua δ è una funzione di transizione e non semplicemente una relazione. Questa differenza, finora immateriale, sarà fondamentale nello studio della Teoria della Complessità.

Questo ci dà un modo per misurare il tempo richiesto per la decisione di ogni istanza, ma noi vorremmo una funzione che varia in base alla taglia $|x|$ del caso $x \in I$.

Definizione 3.1.3 – Tempo per la decisione

Sia M una MdT a k nastri che risolve il problema I. Sia inoltre $f : \mathbb{N} \to \mathbb{N}$ una funzione crescente.

Diremo che M **decide I in tempo deterministico f** se per ogni $x \in I$ il tempo deterministico t_x richiesto da M per decidere l'istanza $x \in I$ è tale che

$$t \leq f(|x|).$$

Definizione 3.1.4 – Classe delle funzioni decidibili in tempo det. f

Sia $f : \mathbb{N} \to \mathbb{N}$ monotona crescente. La **classe di complessità di tempo deterministico f** è la classe

$$\texttt{TIME}(f) := \{ I : \exists M \text{ che decide I in tempo det. } f \}.$$

Spesso ci converrà ragionare in termini di ordini di grandezza: ricordiamo che date due funzioni $f, g : A \to \mathcal{R}$ diremo che f è **$\mathcal{O}$-grande** di g (e lo indichiamo impropriamente con $f = \mathcal{O}(g)$) se esiste $c \in [0, +\infty]$ tale che

$$f(x) \leq c \cdot g(x) \qquad \text{definitivamente},$$

o equivalentemente se

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} \leq c.$$

Scriveremo quindi spesso $\texttt{TIME}(f)$ per indicare in realtà $\texttt{TIME}(\mathcal{O}(f))$, ovvero la classe di tutti gli algoritmi decidibili in tempo deterministico *ordine di* f.

Vogliamo ora dimostrare che l'aggiunta di nastri è un procedimento *effettivo*, ovvero che non altera in modo non-algoritmico la capacità di computazione delle macchine di Turing.

Teorema 3.1.5 – Riduzione del numero di nastri

Sia M a k nastri che decide I in tempo deterministico f. Allora esiste M' ad un nastro che decide I in tempo deterministico $\mathcal{O}(f^2)$.

Dimostrazione. Consideriamo una configurazione della macchina a k nastri

$$(q, \underline{\triangleright} w_1, \dots, \underline{\text{resp}} w_k)_M.$$

Per farla diventare ad un nastro la rappresentiamo come

$$(q', \triangleright\triangleright' w_1 \triangleleft' \triangleright' w_2 \triangleleft' \dots \triangleright' w_k \triangleleft')_{M'}.$$

Per memorizzare il simbolo corrente, aggiungiamo a Σ un simbolo $\overline{\sigma}$ per ogni $\sigma \in \Sigma$.

Il primo passo da fare è prendere l'input della macchina M e trasformarlo nel relativo input della macchina M'. L'input di M sarà della forma

$$(q, \triangleright x, \triangleright, \dots, \triangleright);$$

per scriverlo in M' eseguiamo i seguenti passi:

$$\begin{aligned}(q, \triangleright x) &\to^{|x|+1} (q, \triangleright\triangleright' x) \\ &\to^{\mathcal{O}(|x|)} (q, \underline{\triangleright}\triangleright' x \triangleleft' \triangleright' \triangleleft' \dots \triangleright' \triangleleft').\end{aligned}$$

Dunque per scrivere l'input su M' abbiamo bisogno (al caso pessimo) di $\mathcal{O}(|x|)$ passi.

Vogliamo studiare ora il tempo (massimo) necessario per l'esecuzione di un passo di computazione: data una configurazione di M'

$$(q, \underline{\triangleright}\triangleright' u_1 \overline{\sigma_1} v_1 \triangleleft' \cdots \triangleright' u_k \overline{\sigma_k} v_k \triangleleft')$$

con il cursore nel respingente "vero", per eseguire tutti i passi ci basta

(1) scorrere il nastro verso destra per leggere tutti i caratteri correnti,

(2) tornare al respingente a sinistra,

(3) scorrere il nastro ed eseguire le computazioni,

(4) tornare al respingente.

Dunque sono sufficienti 4 letture complete del nastro di M'.

Osserviamo che ogni *sottonastro* di M', ovvero ogni stringa $u_i \overline{\sigma_i} v_i$, è lunga al più $f(|x|)$: in effetti la macchina M esegue al più $f(|x|)$ passi, e in un passo può scrivere al più una casella del nastro.

Segue che il nastro di M' in totale è lungo al più

$$\ell := k\big(f(|x|) + 2\big) + 1.^{2}$$

Segue che per ogni passo di computazione eseguiamo $\mathcal{O}(\ell) = \mathcal{O}(f)$ operazioni; dato che eseguiamo al più $f(|x|)$ passi di computazione segue che M' decide I in al più

$$f \cdot \mathcal{O}(f) = \mathcal{O}(f^2)$$

passi. □

Il +2 è dovuto ai due respingenti finti $\triangleright'$ e $\triangleleft'$, mentre il +1 è dovuto al respingente vero $\triangleright$ di M'.

Teorema 3.1.6 – Teorema di Accelerazione Lineare

Se $I \in \texttt{TIME}(f)$, allora per ogni $\varepsilon > 0$ si ha che

$$I \in \texttt{TIME}(\varepsilon \cdot f(n) + n + 2).$$

Possiamo finalmente definire una delle più importanti classi di complessità studiate.

Definizione 3.1.7 – Classe $\mathcal{P}$

La **classe di complessità** $\mathcal{P}$ è definita come

$$\mathcal{P} := \bigcup_{k \in \mathbb{N}} \texttt{TIME}(n^k),$$

considerando gli ordini di grandezza.

3.2 Complessità in spazio

Vogliamo ora definire le misure di complessità in spazio per i problemi decidibili. Iniziamo discutendo una nuova estensione delle Macchine di Turing.

Definizione 3.2.1 – Macchina di Turing I/O

Una **Macchina di Turing I/O** è una macchina di Turing a k nastri ($k \geq 3$) $M = (Q, \Sigma, \delta, q_0)$ con i seguenti vincoli aggiuntivi:

(1) Se
$$\delta(q, \sigma_1, \ldots, \sigma_k) = \left(q', (\sigma'_1, D_1), \ldots, (\sigma'_k, D_k)\right)$$
allora $\sigma'_1 = \sigma_1$ e $D_k \in \{\texttt{R}, -\}$. Inoltre se $D_k = -$ allora $\sigma'_k = \sigma_k$.

(2) Se $\sigma_1 = \#$ allora $D_1 \in \{-, \texttt{L}\}$.

L'idea delle macchine I/O è che il primo e l'ultimo nastro hanno i ruoli privilegiati di *lettore* e *scrittore*. Infatti il primo nastro non può essere modificato grazie alla prima condizione, e inoltre arrivati alla fine (cioè alla porzione vuota) non possiamo proseguire verso destra. Analogamente l'ultimo nastro può essere solo scritto: infatti non si può riscrivere un carattere già scritto e ci si può solo muovere verso destra.

Anche questa estensione delle MdT è effettiva, come mostrato dal seguente Teorema.

Teorema 3.2.2

Per ogni MdT a k nastri che decide I in tempo deterministico f, esiste una MdT I/O a $(k+2)$-nastri che decide I in tempo deterministico $c \cdot f$ per qualche $c \in [0, +\infty]$.

Dimostrazione. Basta copiare il contenuto del nastro di input nei nastri $2, \dots, k+1$, eseguire gli stessi passi della macchina di partenza sui nastri *di lavoro* e infine copiare l'output sul nastro $k+2$. □

Il nostro scopo è misurare lo spazio necessario ad una macchina I/O per risolvere un'istanza di un problema. Per farlo potremmo eseguire la macchina e infine contare il numero di caselle non bianche, ma questo non funzionerebbe poiché la macchina può cancellare pezzi scritti durante la computazione.

Introduciamo quindi un'altra banale estensione, ovvero il simbolo di fine stringa $\triangleleft$: ogni volta che abbiamo necessità di più spazio ci basterà spostare tale simbolo a destra, ma avremo cura di non spostarlo mai verso sinistra o cancellarlo in modo da memorizzare la quantità *massima* di memoria usata.

Definizione 3.2.3 – Spazio richiesto per la decisione e classe SPACE

Sia M una MdT a k nastri I/O che decide il problema I.

Data un'istanza $x \in I$ e la computazione di decisione

$$(q_0, \trianglerighteq x \triangleleft, \trianglerighteq \triangleleft, \dots, \trianglerighteq \triangleleft) \to^* (\texttt{SÌ}, w_1, \dots, w_k)$$

diremo che lo **spazio richiesto** per la decisione dell'istanza x è

$$\sum_{i=2}^{k-1} |w_i|.^a$$

Diremo inoltre che M **decide I in spazio deterministico f** se per ogni $x \in I$ lo spazio richiesto per la decisione di x è minore o uguale a $f(|x|)$.

Infine la **classe SPACE(f)** è la classe

$$\texttt{SPACE}(f) := \left\{ I \ : \ \exists M \text{ a } k \text{ nastri I/O che decide } I \text{ in spazio det. } f \right\}.$$

[a] Qui $|w_i|$ indica la lunghezza della stringa w_i.

Osservazione 3.2.1 (Alcune osservazioni sulle definizioni date).

- Lo "spazio richiesto" è dato dalla somma delle lunghezze di tutti i nastri di lavoro, ma non del nastro di lettura né di quello di scrittura.

  In effetti il nastro di scrittura non ha molto peso, in quanto contiene solamente la risposta SÌ oppure NO. Invece se includessimo il nastro di lettura otterremo che lo spazio è sempre almeno lineare nella dimensione dell'input, e questo *appiattirebbe* la gerarchia che cercheremo di delineare.

- Talvolta lo spazio richiesto viene definito come

$$\max\{\,|w_i| \;:\; i = 2, \dots, k-1\,\}.$$

Siccome

$$\sum_{i=2}^{k-1} |w_i| \leq (k-2) \cdot \max_{i=2,\dots,k-2} |w_i|$$

e noi siamo interessati solamente all'ordine di grandezza, queste definizioni sono equivalenti.

Enunciamo un analogo del Teorema 3.1.6.

Teorema 3.2.4 – Compressione lineare dello spazio

Sia $I \in \texttt{SPACE}(f)$: allora per ogni $\varepsilon \in [0, +\infty]$ si ha che

$$I \in \texttt{SPACE}(2 + \varepsilon \cdot f).$$

Definiamo ora le classi di complessità in spazio che studieremo in questo corso.

Definizione 3.2.5

Le classi $\mathcal{P}$**SPACE** e **LOGSPACE** sono definite come segue:

$$\mathcal{P}\texttt{SPACE} := \bigcup_{k \in \mathbb{N}} \texttt{SPACE}(n^k) \qquad \texttt{LOGSPACE} := \bigcup_{k \in \mathbb{N}} \texttt{SPACE}\big(k \log(n)\big).$$

Uno dei risultati fondamentali della Teoria della Complessità (che noi non dimostreremo) è il seguente:

Teorema 3.2.6

$$\texttt{LOGSPACE} \subsetneq \mathcal{P}\texttt{SPACE}$$

ovvero esiste un problema risolvibile in spazio lineare che non può essere risolto in spazio logaritmico.

Per costruire la gerarchia vogliamo scoprire in che relazione è $\mathcal{P}$ con le classi LOGSPACE e $\mathcal{P}$SPACE.

Teorema 3.2.7

$$\texttt{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{P}\texttt{SPACE}.$$

Dimostrazione. Il fatto che $\mathcal{P}$ sia un sottoinsieme di $\mathcal{P}$SPACE è una conseguenza del principio generale che in t unità di tempo una MdT può scrivere al più t caselle del suo nastro, dunque in tempo polinomiale una MdT può scrivere al più un numero polinomiale di caselle.

Dimostriamo che $\texttt{LOGSPACE} \subseteq \mathcal{P}$: sia allora M una MdT a 3 nastri [a] che risolve un

problema I in spazio logaritmico, ovvero la lunghezza massima del singolo nastro di lavoro è al più $k \log n$, dove n è la taglia dell'input.

Segue che il numero massimo di configurazioni attraversabili da M è al più

$$N := |\Sigma|^{k \log n} \cdot k \log n \cdot |Q| \cdot n;$$

infatti $\Sigma^{k \log n}$ è il numero massimo di possibili stringhe scritte sul nastro, $k \log n$ è il numero massimo di posizioni del cursore, $|Q|$ è il numero massimo di stati e n è il numero massimo di posizioni del cursore nel nastro di input.

Vogliamo dimostrare che esiste t tale che $N \leq n^t$, ovvero tale che $\log N \leq t \log n$. Per definizione di N però

$$\begin{aligned} \log N &= \log\left(|\Sigma|^{k \log n} \cdot k \log n \cdot |Q| \cdot n\right) \\ &= k \log n \log|\Sigma| + \log(k \log n) + \log|Q| + \log n, \end{aligned}$$

dunque dividendo per $\log n$ otteniamo un possibile valore di t. □

[a] Se fossero più nastri basterebbe dividere ad un certo punto per un'opportuna costante.

3.3 Non-Determinismo e Complessità non-deterministica

Studiamo infine un'ultima estensione delle macchine di Turing.

Definizione 3.3.1 – Macchine di Turing non deterministiche

Una **macchina di Turing non deterministica** è una quadrupla $N = (Q, \Sigma, \Delta, q_0)$ definita allo stesso modo di una macchina di Turing solita tranne per il fatto che

$$\Delta \subseteq (Q \times \Sigma) \times ((Q \cup \{h\}) \times \Sigma \times \{L, R, -\})$$

è una relazione, detta **relazione di transizione**.

La differenza è quindi che a partire da una configurazione vi sono più configurazioni raggiungibili in output, e la macchina ogni volta ne sceglie una in modo arbitrario e quindi *non deterministico*.

Questa definizione può sembrare fuori dal mondo, ma è collegata strettamente a due tipi di strategie di risoluzione dei problemi molto usate.

- Forza bruta Dato un problema, un possibile modo di risolverlo è sempre quello di generare tutte le possibili soluzioni e controllarle una ad una: tale metodo viene detto *bruteforce* o metodo del *forza bruta*, in quanto brutalmente tentiamo ogni possibilità. Generando lo spazio delle soluzioni esplicitamente, ogni volta che ne tentiamo una stiamo in realtà percorrendo un cammino su tale albero. Una macchina non deterministica dà un risultato positivo se e solo se esiste un cammino sull'albero delle computazioni che termini nello stato di accettazione.

- Guess and Try Un altro modo di risolvere problemi è *tirare ad indovinare*: si prova una possibilità e si controlla se tale possibilità effettivamente risolve il problema o meno. Per quanto tale strategia

sembri diversa dal bruteforce, in realtà stiamo implicitamente generando tutte le soluzioni.[3] Potremmo allora considerare il non determinismo come un metodo per generare automaticamente una soluzione del problema (assumendo che esista) e poi verificarla.

Definizione 3.3.2 – Decisione non deterministica e tempo richiesto

Una macchina di Turing non deterministica N **decide** un problema I se per ogni x vale che $x \in I$ se e solo se esiste una computazione terminante

$$N(x) \to_N^* (\texttt{SÌ}, w).$$

N **decide** I **in tempo non deterministico** f se decide I e per ogni $x \in I$ esiste una computazione terminante

$$N(x) \to_N^t (\texttt{SÌ}, w)$$

con $t \leq f(|x|)$. Infine denotiamo con $\texttt{NTIME}(f)$ la classe di problemi

$$\texttt{NTIME}(f) := \{ I : \exists N \text{ non det. che decide I in tempo non det. } f \}.$$

Avendo definito NTIME, vorremmo sapere in che relazione è con la classe TIME definita in precedenza.

Teorema 3.3.3 – Relazione tra NTIME e TIME

$\texttt{NTIME}(f) \subseteq \texttt{TIME}(c^f)$ per qualche costante c.

Dimostrazione. Sia I un problema in $\texttt{NTIME}(f)$ e sia M una macchina che lo risolve in tempo non det. f: vogliamo trovare una macchina M' che lo risolve in tempo c^f per qualche costante c.

Data la relazione di transizione Δ, consideriamo una coppia $(q, \sigma) \in Q \times \Sigma$ e definiamo

$$\deg(q, \sigma) := |\{ (q', \sigma', D') \}(q, \sigma, q', \sigma', D') \in \Delta|$$
$$\deg \Delta := \max\{ \deg(q, \sigma) : (q, \sigma) \in Q \times \Sigma \}.$$

Intuitivamente, $\deg(q, \sigma)$ è il numero di possibili scelte che la macchina non deterministica può fare quando legge σ nello stato q, cioè è il fattore di ramificazione dell'albero delle computazioni quando ci troviamo in (q, σ), mentre $\deg \Delta$ è il massimo tra tutti i fattori di ramificazione.

Per semplicità, chiamiamo $d := \deg \Delta$. Ordiniamo lessicograficamente le quintuple di Δ: a questo punto ogni computazione consiste in una sequenza di scelte $(c_1, \ldots, c_t)$ dove ognuna delle scelte è necessariamente compresa tra 0 e $d - 1$.

Per simulare la computazione di M attraverso una macchina deterministica, adottiamo una strategia di visita dell'albero delle computazioni a *profondità limitata*: fissiamo inizialmente $t = 1$, $c_1 = 0$ e controlliamo se la computazione codificata da (c_1) termina. Se sì bene, altrimenti incrementiamo c_1 di 1 e eseguiamo il controllo. Arrivati al punto in cui $c_1 = d - 1$ (e quindi non possiamo incrementarlo ulteriormente) aumentiamo t e controlliamo tutte le computazioni lunghe 2.

Dato che M decide I in tempo non deterministico f, per ogni input x avremo che il

[3] specialmente se siamo sfortunati

tempo t richiesto per decidere il caso $x \in I$ è al più $f(|x|)$. In particolare ogni sequenza che rappresenta una computazione è al più lunga $f(|x|)$. Segue che la macchina M' deve fare al più $d^{f(|x|)}$ scelte, da cui $I \in \texttt{TIME}(d^f)$. □

Introduciamo finalmente la classe dei problemi $\mathcal{NP}$ e la sua analoga versione per lo spazio, $\mathcal{NP}\texttt{SPACE}$.

Definizione 3.3.4 – Classe $\mathcal{NP}$

Definiamo la classe $\mathcal{NP}$ come

$$\mathcal{NP} := \bigcup_{k \in \mathbb{N}} \texttt{NTIME}(n^k).$$

Osserviamo che ovviamente

$$\mathcal{P} \subseteq \mathcal{NP}$$

in quanto ogni problema risolvibile in tempo polinomiale deterministico può anche essere risolto non deterministicamente. La domanda da *un miliardo di dollari*[4] è se il contenimento è stretto, ovvero esiste un problema in $\mathcal{NP}$ che non appartiene a P, o se le due classi sono in realtà le stesse.

Definizione 3.3.5 – Classe $\mathcal{NP}\texttt{SPACE}$

Un problema I si dice **decidibile in spazio non deterministico f** se esiste una macchina di Turing I/O non deterministica N tale che per ogni x *esista* una computazione

$$N(x, \triangleright, \ldots, \triangleright) \longrightarrow_N^* (\texttt{SÌ}, w_1, \ldots, w_k)$$

tale che

$$\sum_{i=2}^{n-1} |w_i| \leq f(|x|).$$

Chiamiamo $\texttt{NSPACE}(f)$ la classe

$$\texttt{NSPACE}(f) := \{ I : \exists N \text{ non det. che decide I in spazio non det. } f \}.$$

Infine, definiamo la classe $\mathcal{NP}\texttt{SPACE}$ come

$$\mathcal{NP}\texttt{SPACE} := \bigcup_{k \in \mathbb{N}} \texttt{NSPACE}(n^k).$$

Analogamente al caso temporale, $\mathcal{P}\texttt{SPACE} \subseteq \mathcal{NP}\texttt{SPACE}$: potremmo dunque porci lo stesso problema riguardo al contenimento dei due insiemi. Tuttavia, al contrario del problema $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, il problema $\mathcal{P}\texttt{SPACE} \stackrel{?}{=} \mathcal{NP}\texttt{SPACE}$ è stato risolto.

Teorema 3.3.6 – Teorema di Savitch

$$\mathcal{P}\texttt{SPACE} = \mathcal{NP}\texttt{SPACE}.$$

[4] Letteralmente.

Osserviamo infine che la classe $\mathcal{NP}$ può essere descritta in un modo più naturale pensandola come un *Guess and Try con i superpoteri*. Infatti, in ottica Guess and Try, un problema è in $\mathcal{NP}$ se e solo se, avendo indovinato una soluzione del problema, possiamo verificarla in tempo polinomiale. Questo ci porta alla definizione dei **certificati**: un problema è in $\mathcal{NP}$ se ammette un certificato polinomiale, ovvero se data una soluzione del problema sappiamo verificare che è tale in tempo polinomiale deterministico.

Tale approccio è, per quanto detto precedentemente, equivalente a chiedere che una macchina non deterministica decida il problema in tempo polinomiale, e dunque, dato che è più semplice immaginare i certificati piuttosto che il non determinismo, spesso dimostreremo l'appartenenza ad $\mathcal{NP}$ facendo vedere che i certificati possono essere verificati in tempo polinomiale.

3.4 Funzioni di misura appropriata

Finora abbiamo studiato le classi `TIME` e `SPACE` (e le corrispettive non deterministiche `NTIME`, `NSPACE`) con funzioni di misura f qualsiasi. In realtà per ottenere risultati significativi abbiamo bisogno di alcuni vincoli in più sulla funzione di misura.

Definizione 3.4.1 – Funzione di misura appropriata

Una funzione $f : \mathbb{N} \to \mathbb{N}$ si dice **di misura appropriata** se

(1) è (debolmente) monotona crescente,

(2) è calcolabile ed esiste una macchina M che calcola f sull'input x in

- tempo $\mathcal{O}\big(f(|x|) + |x|\big)$,
- spazio $\mathcal{O}\big(f(|x|)\big)$.

Fortunatamente i polinomi e le funzioni esponenziali/logaritmiche rispettano queste condizioni; inoltre si può dimostrare che somma/prodotto/composizione di funzioni appropriate è ancora appropriata, dunque la teoria delle funzioni di misura appropriate include i casi a cui siamo davvero interessati.

Teorema 3.4.2 – Teorema di Gerarchia

Se f è appropriata allora

(1) $\texttt{TIME}(f(n)) \subsetneq \texttt{TIME}\big(f(2n+1)^3\big)$,[5]

(2) $\texttt{SPACE}(f(n)) \subsetneq \texttt{SPACE}(f(x) \cdot \log(f(x)))$.

Come conseguenza otteniamo i seguenti due teoremi.

Teorema 3.4.3

$$\mathcal{P} \subsetneq \texttt{EXP} := \bigcup_{k \in \mathbb{N}} \texttt{TIME}(2^{n^k}).$$

[5] Sorprendentemente, questo fatto si dimostra facendo vedere che il sottoinsieme $\{ x : \varphi_x(x)\downarrow \text{ in al più } f(|x|) \text{ passi} \}$ di K appartiene a $\texttt{TIME}\big(f(2n+3)^3\big)$ ma non a $\texttt{TIME}(f(n))$.

Dimostrazione. Innanzitutto $\mathcal{P} \subseteq \texttt{TIME}(2^n)$ poiché i polinomi sono sempre limitati superiormente dalle funzioni esponenziali (definitivamente). Per il Teorema di Gerarchia segue che $\texttt{TIME}(2^n) \subsetneq \texttt{TIME}\left(2^{(2n+1)^3}\right)$, che è a sua volta un sottoinsieme di EXP. Segue dunque la tesi. □

Teorema 3.4.4

$\mathcal{P} \subseteq \mathcal{NP} \subseteq \texttt{EXP}$.

Dimostrazione. Il fatto che $\mathcal{P} \subseteq \mathcal{NP}$ è già stato dimostrato; l'inclusione di $\mathcal{NP}$ in EXP deriva dal Teorema 3.3.3. □

Le funzioni di misura appropriate ci consentono finalmente di descrivere la gerarchia di cui parliamo dall'inizio del capitolo.

Teorema 3.4.5

Sia f appropriata, k costante. Allora

(1) $\texttt{TIME}(f) \subseteq \texttt{NTIME}(f)$,

(2) $\texttt{SPACE}(f) \subseteq \texttt{NSPACE}(f)$,

(3) $\texttt{NTIME}(f) \subseteq \texttt{TIME}\left(k^{\log n + f(n)}\right)$,

(4) vale la gerarchia

$$\texttt{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}\texttt{SPACE} = \mathcal{NP}\texttt{SPACE}.$$

Inoltre $\mathcal{NP} \subseteq \texttt{EXP}$.

La gerarchia definita sopra tuttavia non è superiormente limitata, ovvero non esiste una classe che include tutti i problemi. In effetti una tale classe non può esistere, come garantito dal seguente teorema.

Teorema 3.4.6 – Illimitatezza della gerarchia di complessità

Per ogni funzione $g : \mathbb{N} \to \mathbb{N}$ calcolabile totale[a] esiste una funzione $f : \mathbb{N} \to \mathbb{N}$ calc. tot. tale che

(1) esiste un problema I tale che $I \in \texttt{TIME}(f)$ ma $I \notin \texttt{TIME}(g)$,

(2) $f \geq g$ definitivamente, ovvero $f(n) \geq g(n)$ per tutti gli $n \in \mathbb{N}$ eccetto un numero finito.

[a] non necessariamente appropriata

Concludiamo la discussione delle funzioni di misura appropriate mostrando quale è il prezzo da pagare se volessimo considerare tutte le funzioni calcolabili totali, e non solo quelle appropriate. Lo facciamo enunciando due teoremi, ma senza dimostrarli.

Teorema 3.4.7 – Teorema di Accelerazione, Blum

Per ogni funzione $h : \mathbb{N} \to \mathbb{N}$ calcolabile totale, ma non appropriata, esiste un problema I tale che, per ogni macchina M che decide I in tempo f, esiste una macchina M' che decide I in tempo f' con

$$f \geq h \circ f'$$

quasi ovunque.

Il Teorema di Accelerazione di Blum ci dice che data una funzione non appropriata h possiamo costruire un problema I che *non ammette algoritmo ottimo*: il suo tempo di esecuzione può essere ridotto all'infinito seguendo la successione di macchine $M, M', M'', \ldots$ descritto nell'enunciato.

Teorema 3.4.8 – Teorema della Lacuna, Borodin

Esiste una funzione calcolabile totale f (non appropriata) tale che

$$\texttt{TIME}(f) = \texttt{TIME}(2^f).$$

Se il Teorema della Lacuna fosse ammissibile la gerarchia delineata verrebbe distrutta: le classi $\mathcal{P}$ ed $\texttt{EXP}$ collasserebbero e studiare la complessità sarebbe inutile.

Un ultimo tentativo è quello di condensare le definizioni di complessità in spazio e tempo usando il cosiddetto **approccio assiomatico alla complessità**, dovuto ancora una volta a Blum.

Definizione 3.4.9 – Funzioni che misurano la complessità

Una funzione φ **misura la complessità** se è della forma

$$\varphi : \big((\mathbb{N} \to \mathbb{N}) \times \mathbb{N}\big) \to \mathbb{N}^a$$

e soddisfa le seguenti due condizioni:

(1) per ogni $\psi : \mathbb{N} \to \mathbb{N}$, $x \in \mathbb{N}$ si ha che $\varphi(\psi, x)$ converge se e solo se $\psi(x)$ converge,

(2) per ogni $\psi : \mathbb{N} \to \mathbb{N}$, $x, k \in \mathbb{N}$ è possibile decidere se $\varphi(\psi, x) = k$.

[a] Ovvero prende in input una funzione $\mathbb{N} \to \mathbb{N}$ e un numero naturale e restituisce in output un altro numero naturale.

Il primo assioma ci dice che φ misura la complessità del calcolo di ψ, il secondo ci assicura che è possibile calcolare la complessità usando φ. Se φ contasse il numero di passi per calcolare $\psi(x)$ otterremmo la misura di complessità in tempo; se contasse lo spazio otterremmo la complessità in spazio.

Le funzioni di misura appropriata soddisfano gli assiomi, tuttavia esistono anche funzioni non appropriate che misurano la complessità e ancora una volta ci portano a situazioni controintuitive (come il Teorema 3.4.8).

4

Complessità di problemi

4.1 Classificazione di $\mathcal{P}$ e $\mathcal{NP}$

Vogliamo ora studiare più nel dettaglio le classi di complessità $\mathcal{P}$ e $\mathcal{NP}$: esattamente come abbiamo fatto con $\mathcal{R}$ e $\mathcal{RE}$ vogliamo dunque trovare una riduzione che *classifichi* le classi $\mathcal{P} \subseteq \mathcal{NP}$ e dei problemi completi per queste due classi.

Per capire perché ciò è interessante è sufficiente enunciare la cosiddetta **Tesi di Cook-Karp**, che rappresenta un analogo nel mondo della complessità della Tesi di Church-Turing.

Teorema 4.1.1 – Tesi di Cook-Karp

I problemi in $\mathcal{P}$ sono *trattabili*, i problemi di $\mathcal{NP}$ sono i problemi *intrattabili*.

Possiamo spiegarci la Tesi di Cook-Karp nel seguente modo.

- I problemi in $\mathcal{P}$ sono chiusi per *cambiamento di modello*, ovvero se un problema $\mathsf{P} \in \mathcal{P}$ può essere rappresentato in un'altra forma $\mathsf{P}' \in \mathcal{P}$ allora possiamo sempre trovare un algoritmo di conversione che abbia complessità polinomiale.

  In altre parole, la classe $\mathcal{P}$ è chiusa per *composizione polinomiale sinistra*.

- La classe $\mathcal{P}$ è chiusa rispetto a somma/prodotto e alle riduzioni $\leq_{\mathcal{L}}$ dove $\mathcal{L}$ è una sottoclasse di problemi. Mostreremo questo fatto usando in particolare la sottoclasse $\mathcal{L} = \texttt{LOGSPACE}$.

  Riformulando, $\mathcal{P}$ è anche chiusa per *composizione polinomiale destra*.

- Inoltre gli algoritmi polinomiali, specialmente se con costanti piccole e esponenti bassi, sono davvero (nella pratica) più *trattabili* di quelli esponenziali.

Tuttavia non sempre gli algoritmi di $\mathcal{P}$ sono migliori di quelli esponenziali, soprattutto se le costanti sono molto grandi e i corrispettivi algoritmi esponenziali sono in realtà polinomiali nel caso medio, che spesso è quello più interessante.

Per semplicità noi ci limiteremo comunque al caso pessimo e ignoreremo tranquillamente i problemi pratici legati alle costanti grandi o agli esponenti alti.

Come accennato in precedenza, invece di studiare riduzioni polinomiali ci limiteremo a studiare un gruppo di riduzioni più piccolo, ovvero quelle *logaritmiche*.

Definizione 4.1.2 – Riduzione efficiente

Un problema I si **riduce efficientemente** ad un problema J se

$$I \leq_{\texttt{LOGSPACE}} J,$$

ovvero se esiste una funzione $f \in \texttt{LOGSPACE}$ tale che

$$x \in I \qquad \text{se e solo se} \qquad f(x) \in J.$$

Per semplicità, pigrizia e motivi estetici[1] in seguito scriveremo $\leq_{\mathcal{L}}$ invece di $\leq_{\texttt{LOGSPACE}}$.

Teorema 4.1.3

Siano $\mathcal{D}, \mathcal{E}$ due classi tra $\{\texttt{LOGSPACE}, \mathcal{P}, \mathcal{NP}, \texttt{EXP}, \mathcal{P}\texttt{SPACE}\}$ tali che $\mathcal{D} \subseteq \mathcal{E}$.
Allora $\leq_{\mathcal{L}}$ (e quindi a maggior ragione $\leq_{\mathcal{P}}$) classifica $\mathcal{D}$ ed $\mathcal{E}$.

Dimostrazione. Per mostrare che una riduzione classifica due classi di problemi dobbiamo mostrare gli assiomi dati in Definizioni 2.3.3.

(1) Certamente $A \leq_{\mathcal{L}} A$ in quanto l'identità (che copia l'input nell'output) è logaritmica nello spazio di lavoro.[a]

(2) Vorremmo dimostrare che la composizione di algoritmi logaritmici in spazio è ancora logaritmico in spazio, ma non possiamo semplicemente incollare le due MdT tra di loro, perché a questo punto l'output della prima diventa un nastro di lavoro, e quindi può rendere l'algoritmo polinomiale.

Possiamo tuttavia operare con uno stile *master-slave*: la seconda macchina (che fa la parte del *master*) legge un carattere alla volta dalla prima (che fa la parte dello *slave*) e quando ha letto tutto l'input esegue i suoi calcoli. In questo modo magari sprechiamo molto tempo, ma il risultato è logaritmico in spazio.

(3) Sia $f \in \texttt{LOGSPACE}$ la funzione che riduce A a B, che appartiene a $\mathcal{D}$: per mostrare che A appartiene a $\mathcal{D}$ basta trasformare A in $\mathbb{B}$ tramite f e poi risolvere B. Dato che $\texttt{LOGSPACE} \subseteq \mathcal{D}$ segue che questa composizione appartiene ancora a $\mathcal{D}$, come volevamo.

(4) Analogo al punto precedente, ma usando l'ipotesi $B \in \mathcal{E}$.

□

[a] Non lo usa!

4.1.1 Logica proposizionale

Prima di introdurre i principali problemi che studieremo nell'ambito della complessità abbiamo bisogno di introdurre alcune semplici nozioni di logica proposizionale.

[1] Soprattutto gli ultimi due.

Definizione 4.1.4 – Espressione booleana

Un'**espressione booleana** nell'insieme di variabili X è una espressione della forma

$$B ::= \mathcal{T} \mid \mathcal{F} \mid x \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2$$

dove $x \in X$. Le espressioni $\mathcal{T}, \mathcal{F}, x, \neg x$ sono dette **letterali**.

Definizione 4.1.5 – Interpretazione e soddisfacibilità

Dato un insieme di variabili X, un'**interpretazione** o **valutazione** è una funzione

$$\mathcal{V} : X \to \{\mathcal{T}, \mathcal{F}\}.$$

Se B è un'espressione booleana su X, diremo che l'interpretazione $\mathcal{V}$ **soddisfa** B se e solo se vale il predicato $\mathcal{V} \models B$ definito per induzione strutturale:

$\mathcal{V} \models \mathcal{T}$	
$\mathcal{V} \models x$	se $\mathcal{V}(x) = \mathcal{T}$
$\mathcal{V} \models \neg B$	se non vale $\mathcal{V} \models B$
$\mathcal{V} \models B_1 \vee B_2$	se $\mathcal{V} \models B_1$ oppure $\mathcal{V} \models B_2$
$\mathcal{V} \models B_1 \wedge B_2$	se $\mathcal{V} \models B_1$ e $\mathcal{V} \models B_2$.

Infine una formula B si dice **soddisfacibile** se esiste un'interpretazione $\mathcal{V}$ che soddisfa B.

Definizione 4.1.6 – Forma normale congiuntiva

Una formula booleana si dice **in forma normale congiuntiva** se

$$B = \bigwedge_i C_i$$

dove i termini C_i si dicono **clausole** o **vincoli** e sono della forma

$$C_i = \bigvee_j c_{ij}$$

e ogni c_{ij} è un letterale.

Più semplicemente, una formula è in forma normale congiuntiva se è scritta come congiunzione (cioè *and*) di formule disgiuntive (cioè contententi solamente *or*):

$$(c_{11} \vee \cdots \vee c_{1,n_1}) \wedge \cdots \wedge (c_{t1} \vee \cdots \vee c_{t,n_t}).$$

L'importanza delle formule in FNC viene dal seguente teorema.

Teorema 4.1.7

Ogni formula può essere espressa equivalentemente in FNC, ovvero data B formula booleana sull'insieme di variabili X esiste B′ su X tale che

1. B′ è in FNC

2. per ogni interpretazione $\mathcal{V}$ si ha che $\mathcal{V} \models B$ se e solo se $\mathcal{V} \models B'$.

4.1.2 Il problema SAT

Definiamo ora il più importante problema di $\mathcal{NP}$: il resto del corso sarà dedicato a dimostrare che esso è in realtà $\mathcal{NP}$-completo.

Definizione 4.1.8 – Il problema SAT

$$\texttt{SAT} := \{\, B \text{ formula booleana} \;:\; \exists \mathcal{V} \text{ valutazione tale che } \mathcal{V} \models B \,\}.$$

SAT appartiene ad $\mathcal{NP}$: infatti data una valutazione $\mathcal{V}$ sono necessari un numero polinomiale di passi per certificare che $\mathcal{V}$ soddisfi B.

Confrontiamo SAT con un altro classico problema di $\mathcal{NP}$, che questa volta viene dalla teoria dei grafi.

Definizione 4.1.9 – Problema del Ciclo Hamiltoniano

Il problema HAM è l'insieme di tutti i grafi $G := (V, E)$ che ammettono un **ciclo hamiltoniano**, ovvero un ciclo che tocca tutti e soli i nodi del grafo una ed una sola volta.

Teorema 4.1.10

$$\texttt{HAM} \leq_{\mathcal{L}} \texttt{SAT}.$$

Dimostrazione. Dimostrare che HAM si riduce a SAT in spazio logaritmico significa mostrare che possiamo codificare tutti e soli i grafi che ammettono un ciclo hamiltoniano in una formula soddisfacibile, e tale codifica deve essere logaritmica in spazio.

Sia allora $G = (V, E)$ un grafo, $n := |V|$: costruiamo una formula booleana B sull'insieme di variabili

$$X\{\, x_{ij} \;:\; i, j = 1, \dots, n \,\}.$$

Osserviamo che un ciclo hamiltoniano è identificato univocamente da una sequenza di vertici $(v_1, \dots, v_n)$, che indica l'ordine di attraversamento dei nodi nel grafo: in altre parole, ogni ciclo hamiltoniano è in realtà una funzione bigettiva

$$\sigma : V \xrightarrow{\sim} \{1, \dots, n\}.$$

La variabile x_{ij} sarà $\mathcal{T}$ se e solo se il nodo j occupa l'i-esimo posto nella permutazione $(v_1, \dots, v_n)$, ovvero se $\sigma(j) = i$.

Vogliamo costruire una formula che sia soddisfacibile se e solo se esiste una permutazione dei nodi di V per cui ogni coppia di nodi consecutivi sia connessa da un arco: lo facciamo imponendo 5 tipi di vincoli, che verranno portati in forma normale congiuntiva.

(1) Per ogni $i \neq k$ aggiungiamo i vincoli

$$\neg(x_{ij} \wedge x_{kj}) \;\equiv\; \neg x_{ij} \vee \neg x_{kj}$$

al variare di $j = 1, \dots, n$. Questi vincoli ci dicono che il nodo j non può occupare contemporaneamente le posizioni i e k della permutazione.

(2) Per ogni j aggiungiamo il vincolo

$$x_{1j} \vee x_{2j} \vee \dots x_{nj}.$$

Ciò impone che il nodo j debba essere presente nella permutazione in almeno una posizione.

(3) Per ogni i aggiungiamo il vincolo

$$x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}.$$

In questo modo imponiamo la condizione che in posizione i ci sia almeno un nodo.

(4) Per ogni $j \neq k$ aggiungiamo i vincoli

$$\neg(x_{ij} \wedge x_{ik}) \equiv \neg x_{ij} \vee \neg x_{ik}$$

al variare di $i = 1, \dots, n$. Questi vincoli ci dicono che nella posizione i non possono esserci contemporaneamente i nodi j e k.

(5) Per ogni coppia j, k tali che $(j, k) \notin E$ aggiungiamo i vincoli

$$\neg(x_{ij} \wedge x_{i+1,k}) \equiv \neg x_{ij} \vee \neg x_{i+1,k},$$

per ogni i,[a] ovvero i nodi j, k non possono comparire in posizioni consecutive della permutazione.

Osserviamo che le prime 4 condizioni in ordine impongono che σ sia univalente, totale, surgettiva e iniettiva, ovvero che σ sia una bigezione. Inoltre se σ esiste (cioè il grafo ammette un ciclo hamiltoniano) allora per costruzione la valutazione

$$\mathcal{V}(x_{ij}) := \begin{cases} \mathcal{T}, & \text{se } \sigma(j) = i \\ \mathcal{F}, & \text{altrimenti} \end{cases}$$

soddisfa la formula costruita sopra. Invece se σ non esistesse certamente la formula risulterebbe insoddisfacibile.

Rimane solo da mostrare che la funzione di riduzione f sia in `LOGSPACE`: tale f prende sul nastro di lavoro il grafo, rappresentato come insieme dei nodi e insieme degli archi, e restituisce in output la formula booleana che corrisponde ai vincoli necessari per costruire la permutazione. Sui nastri di lavoro dobbiamo solo scorrere le variabili i, j, k che servono alla costruzione dei vincoli e ricordare il valore di n.

Ma allora rappresentando n, i, j, k in binario ognuno di essi occupa al più $\log n$ caselle, dunque per i nastri di lavoro abbiamo bisogno di al più $(4 \log n)$ caselle, che è logaritmico nella taglia dell'input, come volevamo. □

[a] Se $i = n$, allora interpretiamo $i+1$ *in modulo*, e quindi $i+1 = 1$: questo è necessario perché siamo interessati ai *cicli* hamiltoniani.

4.2 Circuiti booleani e il Teorema di Cook-Levin

Per dimostrare che SAT è $\mathcal{NP}$-completo definiremo e dimostreremo un problema simile, che è il problema CIRCUIT-SAT della *soddisfacibilità di un circuito booleano*.

Definizione 4.2.1 – Funzioni e circuiti booleani

Una **funzione booleana** è una funzione $f : \{0,1\}^n \to \{0,1\}$.

Un **circuito booleano** su un insieme di variabili X è un grafo diretto aciclico $G = (V, E)$ insieme ad una funzione

$$s : V \to \{0, 1, \neg, \vee, \wedge\} \cup X$$

dove

- i nodi si chiamano **porte**,
- X è un insieme di variabili,
- esiste un nodo speciale $u \in V$, detto **uscita** del circuito,
- la funzione s assegna ad ogni nodo la sua **sorta** e soddisfa la seguente condizione: per ogni $n \in V$
  (1) se $s(n) \in \{0,1\} \cup X$ allora n si dice **porta costante** ed ha zero ingressi e un'uscita;
  (2) se $s(n) = \neg$ allora n ha un ingresso e un'uscita;
  (3) se $s(n) \in \{\wedge, \vee\}$ allora n ha due ingressi e un'uscita.

  Tuttavia se $n = u$ è l'uscita del circuito, il nodo n ha 0 uscite (anche se la sorta indicherebbe altrimenti).

Analogamente alle formule booleane, diremo che G è **chiuso** se non contiene variabili, cioè se $X = \varnothing$.

Osserviamo che, seppure le condizioni sul numero di ingressi siano importanti, un nodo può avere più[2] uscite di quelle permesse dalla sua sorta: possiamo immaginare di duplicare il filo, o di duplicare tutto il sottocircuito relativo al nodo da cui vogliamo più uscite.

Definizione 4.2.2 – Soddisfacibilità di un circuito

Sia $G = (V, E)$ un circuito booleano su X con uscita $u \in V$; sia inoltre $\mathcal{V} : X \to \{0,1\}$ un'interpretazione. Possiamo estendere $\mathcal{V}$ ad una funzione

$$\mathcal{V}_G : V \to \{0,1\}$$

dove

- $\mathcal{V}_G(0) = 0$ e $\mathcal{V}_G(1) = 1$,
- $\mathcal{V}_G(x) = \mathcal{V}(x)$ per ogni $x \in X$,
- se $n = \neg$ e la sua unica entrata è $(n', n) \in E$, vale che $\mathcal{V}_G(n) = 1$ se e solo se $\mathcal{V}_G(n') = 0$,

[2] Ma non meno.

- se $n = \wedge$ e le sue entrate sono $(n_1, n), (n_2, n) \in E$, vale che $\mathcal{V}_G(n) = 1$ se e solo se $\mathcal{V}_G(n_1) = 1$ e $\mathcal{V}_G(n_2) = 1$,
- se $n = \vee$ e le sue entrate sono $(n_1, n), (n_2, n) \in E$, vale che $\mathcal{V}_G(n) = 1$ se e solo se $\mathcal{V}_G(n_1) = 1$ oppure $\mathcal{V}_G(n_2) = 1$.

La valutazione del circuito G sarà quindi $\mathcal{V}_G(u)$. Diremo infine che $\mathcal{V}$ **soddisfa** G ($\mathcal{V} \models G$) se $\mathcal{V}_G(u) = 1$.

Come nel caso delle formule booleane, sostituendo le variabili di un circuito con le loro interpretazioni si ottiene un circuito chiuso: in particolare possiamo parlare della *valutazione di un circuito chiuso* senza dover pensare ad una particolare interpretazione. In questi casi scriveremo perciò $\varnothing \models G$ per dire che il circuito chiuso è soddisfatto.

Questo motiva la prossima definizione.

Definizione 4.2.3 – `CIRCUIT-VALUE` e `CIRCUIT-SAT`

Definiamo i problemi `CIRCUIT-VALUE` e `CIRCUIT-SAT` come

$$\texttt{CIRCUIT-VALUE} = \{\, C \text{ circuito booleano chiuso} \,:\, \varnothing \models C \,\}$$
$$\texttt{CIRCUIT-SAT} = \{\, C \text{ circuito booleano} \,:\, \exists \mathcal{V} \text{ tale che } \mathcal{V} \models C \,\}.$$

Il problema `CIRCUIT-VALUE` è ovviamente in $\mathcal{P}$: la taglia del problema è il numero di porte, e per calcolare il valore della porta di uscita ci basta eseguire un'operazione per porta. Analogamente è facile vedere che `CIRCUIT-SAT` è in $\mathcal{NP}$: dato un certificato, ovvero una valuazione $\mathcal{V}$, verificare che $\mathcal{V}$ soddisfi C è come verificare che il circuito chiuso $\bar{C}$ ottenuto per sostituzione è soddisfatto, ovvero è come verificare che $\bar{C}$ appartenga a `CIRCUIT-VALUE`, che si può fare in tempo polinomiale perché `CIRCUIT-VALUE` appartiene a $\mathcal{P}$.

Inoltre `CIRCUIT-VALUE` è banalmente un caso particolare di `CIRCUIT-SAT`, dunque vale la seguente riduzione.

Proposizione 4.2.4

`CIRCUIT-VALUE` $\leq_{\mathcal{L}}$ `CIRCUIT-SAT` e la funzione di riduzione è id $\in$ `LOGSPACE`.

Delineamo ora il piano per dimostrare l'$\mathcal{NP}$-completezza di `SAT`.

(1) Innanzitutto dimostreremo che `CIRCUIT-SAT` $\leq_{\mathcal{L}}$ `SAT`: in questo modo ci basterà dimostrare che `CIRCUIT-SAT` è $\mathcal{NP}$-completo per ottenere l'analogo risultato per `SAT` (grazie alla Proposizione 2.3.7).

(2) Mostreremo poi che `CIRCUIT-VALUE` è $\mathcal{P}$-completo: nel far ciò costruiremo la **tabella di computazione** di un problema, che ci sarà utile per l'ultimo passo.

(3) Infine mostreremo che `CIRCUIT-SAT` è $\mathcal{NP}$-completo, per cui dal passo (1) seguirà l'$\mathcal{NP}$-completezza di `SAT`.

4.2.1 CIRCUIT-SAT $\leq_{\mathcal{L}}$ SAT

Teorema 4.2.5

CIRCUIT-SAT $\leq_{\mathcal{L}}$ SAT.

Dimostrazione. Vogliamo una funzione $f \in$ LOGSPACE che trasformi circuiti in formule booleane e tale che per ogni circuito C

$$\exists \mathcal{V} \text{ tale che } \mathcal{V} \models C \qquad \text{se e solo se} \qquad \exists \mathcal{V}' \text{ tale che } \mathcal{V}' \models f(C).$$

Dato un circuito C con variabili X, costruiamo una formula booleana con variabili

$$X' := X \cup \{ x_g \ : \ g \text{ è una porta di } C \}.$$

Per costruire la formula $f(C)$, costruiamo delle clausole che rappresentano i vincoli del circuito booleano C.

In particolare aggiungiamo un vincolo per ogni porta, più uno per l'uscita. Sia allora $g \in V$ un nodo.

- Se g ha sorta 1 (risp. 0) aggiungiamo la clausola (x_g) (risp. $(\neg x_g)$).
- Se g ha sorta $x \in X$ (ovvero g è una variabile), x_g deve essere vero se e solo se lo è x. Portando il "se e solo se" in forma a clausole otteniamo due clausole:

$$(\neg x_g \vee x) \wedge (x_g \vee \neg x).$$

- Se g ha sorta $\neg$ esiste un unico arco $(h, g) \in E$: vogliamo che x_g sia vero se e solo se x_h è falso. In forma a clausole:

$$(x_g \vee x_h) \wedge (\neg x_g \vee \neg x_h).$$

- Se g ha sorta $\vee$ esistono due archi distinti $(h, g), (k, g) \in E$: vogliamo che x_g sia vero se e solo se lo è almeno uno tra x_k e x_h. Portiamolo in forma a clausole:

$$\begin{aligned}(x_g \iff x_h \vee x_k) &\equiv (x_g \implies x_h \vee x_k) \wedge (x_h \vee x_k \implies x_g) \\ &\equiv (\neg x_g \vee x_h \vee x_k) \wedge (\neg(x_h \vee x_k) \vee x_g) \\ &\equiv (\neg x_g \vee x_h \vee x_k) \wedge ((\neg x_h \wedge \neg x_k) \vee x_g) \\ &\equiv (\neg x_g \vee x_h \vee x_k) \wedge (\neg x_h) \wedge (\neg x_k \vee x_g).\end{aligned}$$

- Se g ha sorta $\wedge$ esistono due archi distinti $(h, g), (k, g) \in E$: vogliamo che x_g sia vero se e solo se lo sono x_k e x_h. Portiamolo in forma a clausole:

$$\begin{aligned}(x_g \iff x_h \wedge x_k) &\equiv (x_g \implies x_h \wedge x_k) \wedge (x_h \wedge x_k \implies x_g) \\ &\equiv (\neg x_g \vee (x_h \wedge x_k)) \wedge (\neg(x_h \wedge x_k) \vee x_g) \\ &\equiv (\neg x_g \vee x_h) \wedge (\neg x_g \vee x_k) \wedge (\neg x_h \vee \neg x_k \vee x_g).\end{aligned}$$

Infine se g è anche l'uscita del circuito aggiungiamo la clausola (x_g), in quanto siamo interessati solo alle interpretazioni che rendano vere il circuito.

Tale costruzione è sicuramente ben definita ed è evidente che il circuito C è soddisfacibile se e solo se la formula $f(C)$ è soddisfacibile, in quanto $f(C)$ è la *riscrittura* di C come formula booleana.

Infine, la costruzione può essere compiuta in spazio logaritmico poiché abbiamo bisogno di memorizzare solamente i valori di g, h, k, n (dove $n := |V|$) che sono tutti logaritmici in n. □

4.2.2 Tabella delle computazioni e $\mathcal{P}$-completezza di CIRCUIT-VALUE

Il secondo passo nel nostro "programma" è dimostrare la $\mathcal{P}$-completezza di CIRCUIT-VALUE: anche se questo non ha direttamente peso nella dimostrazione della $\mathcal{NP}$-completezza di CIRCUIT-SAT, per farlo useremo uno strumento che ci tornerà molto utile.

Data una macchina di Turing M e un input x, vogliamo costruire una matrice quadrata T, detta **tabella di computazione**, che riassuma i passi fatti nel calcolo di $M(x)$: in particolare definiamo T in modo che la sua i-esima riga contenga il nastro di M all'i-esimo passo di computazione, e quindi la posizione $T(i, j)$ contenga la j-esima casella del nastro all'i-esimo passo.

Una tabella così definita non racchiude in sé tutta l'informazione necessaria per capire come si è mossa la macchina: mancano almeno delle informazioni sul cursore e sullo stato. Iniziamo quindi ad aggiungere qualche vincolo che può tornarci utile.

1. Innanzitutto, siccome lavoreremo con problemi che sono in $\mathcal{P}$, sicuramente il numero di passi necessari alla macchina per decidere il problema è minore di $|x|^k$ per qualche $k \in \mathbb{N}$. Allora scegliamo k sufficientemente grande, in modo che M si arresti in meno di $|x|^k - 2$ passi, e imponiamo che T sia di taglia $|x|^k \times |x|^k$.

2. Osserviamo che T ha abbastanza colonne per contenere i nastri di M: infatti in tempo t si possono scrivere al più t caselle, dunque sicuramente non avremo problemi di spazio. In particolare riempiremo tutte le caselle a destra dell'ultima casella scritta con caratteri vuoti (#): siccome la macchina termina la sua computazione in meno di $|x|^k$ passi, questo ci assicura subito che l'ultima colonna sarà sempre formata da tutti caratteri vuoti e non verrà mai raggiunta.

3. Per memorizzare lo stato e la posizione del cursore cambiamo i simboli dell'alfabeto Σ in

$$\Sigma' := \Sigma \times (Q \cup \{*\})$$

dove $*$ è un simbolo a caso che non rappresenti uno stato. Imponiamo allora che tutte le caselle di una riga di T tranne esattamente una contengano un simbolo della forma $(\sigma, *)$, dove $\sigma \in \Sigma$. L'interpretazione che diamo a questa nuova scrittura è la seguente:

   - se in una casella c'è un simbolo della forma $(\sigma, *)$, allora il cursore non si trova in questa posizione e la casella del nastro di M contiene il simbolo σ;
   - l'unica casella della forma (σ, q) con $q \neq *$ è la casella del nastro contenente il cursore; inoltre tale casella ci indica che lo stato della macchina in tale passo è $q \in Q$.

4. Facciamo in modo che il cursore della tabella di computazione non sia mai sul respingente, ma sia al massimo sulla casella appena a destra. Per far ciò potremmo introdurre un secondo respingente, oppure far partire la computazione dalla seconda casella e, ogni qualvolta il cursore di M vada sul respingente, condensare due passi in uno e rispostarlo sulla casella appena dopo.

5. Quando la macchina arriva nello stato di arresto (`SÌ` oppure `NO`) aggiungiamo dei passi che la riportano nella seconda casella del nastro. Ciò comporta:
   - dover potenzialmente aumentare k, poiché abbiamo bisogno di più passi;
   - passare sopra il respingente in caso ve ne siano alcuni sul nastro.

   In ogni caso non passeremo mai sul respingente in colonna 1: questo implica che la prima colonna sia fatta soltanto di respingenti.
6. Infine, una volta arrivati allo stato di arresto e aver portato il cursore in seconda colonna, riempiamo tutte le righe successive con il contenuto di quest'ultima riga (per ottenere una matrice quadrata).

Osserviamo ora che per come abbiamo definito la tabella

- la prima riga è completamente determinata dall'input: contiene infatti il respingente in posizione $T(1,1)$, l'input dalla posizione $T(1,2)$ e infine lo spazio rimanente è occupato da caratteri vuoti;
- la prima colonna contiene solo respingenti;
- l'ultima colonna contiene solo caratteri vuoti.

Come determiniamo le righe successive alla prima? L'osservazione cruciale è che per $1 < i, j, < |x|^k$ la casella $T(i,j)$ dipende solo dalle tre caselle

$$T(i-1,j-1), \quad T(i-1,j) \quad T(i-1,j+1)$$

e dalla funzione di transizione δ. Infatti

- il simbolo σ scritto nella posizione j può cambiare solo se al passo precedente il cursore era già in posizione j;
- il cursore può arrivare in posizione j solo se al passo precedente si trovava in posizione $j-1, j$ oppure $j+1$ (quindi in una posizione adiacente);
- se il cursore al passo $i-1$ si trova in una posizione diversa dalle tre appena nominate, sicuramente $T(i,j) = T(i-1,j)$.

Infine, se il cursore effettivamente è in una delle posizioni $T(i-1,j-1)$, $T(i-1,j)$ o $T(i-1,j+1)$, automaticamente conosciamo lo stato q della macchina M al passo $i-1$ e quindi sappiamo quale mossa deve compiere.

Teorema 4.2.6 – $\mathcal{P}$-completezza di `CIRCUIT-VALUE`

Il problema `CIRCUIT-VALUE` è $\leq_{\mathcal{L}}$-completo per $\mathcal{P}$.

Dimostrazione. Sia I un problema in $\mathcal{P}$, M la macchina che lo risolve in tempo polinomiale e x un dato di ingresso: vogliamo una funzione f che trasformi x in un circuito chiuso $f(x)$ che sia soddisfatto (dall'assegnamento vuoto) se e solo se $x \in I$. Per semplicità chiamiamo anche $n := |x|$.

Costruzione della funzione f

Consideriamo la tabella di computazione di $M(x)$: essa avrà taglia $n^k \times n^k$ e avrà come simboli elementi di $\Sigma' := \Sigma \times Q \cup \{*\}$ come descritto precedentemente. Come primo passo codifichiamo ogni possibile simbolo come una stringa di bit

$$(s_1, \ldots, s_m) \in \{0,1\}^m,$$

dove $m := \lceil \log|\Sigma'| \rceil$. In questo modo ogni riga della tabella di computazione diventa una stringa di bit lunga $m \cdot n^k$. Denotiamo dunque con $s_{i,j,k}$ il k-esimo bit del blocco alla riga i e colonna j. Indicheremo inoltre

$$S_{i,j} := (s_{i,j,1}, \ldots, s_{i,j,m})$$

ovvero tutti i bit del blocco $T(i,j)$.

Grazie ai vincoli imposti sulla tabella di computazione, i bit di $S_{i,1}$ rappresentano il respingente, mentre i bit di S_{i,n^k} rappresentano il carattere vuoto. Infine la prima riga di bit, ovvero $S_{1,j}$ al variare di j, è determinata dall'input.

Per i ragionamenti fatti in precedenza sulla forma della tabella di computazione, quando i,j rappresenta una posizione centrale della tabella (cioè escludendo la prima riga e la prima e l'ultima colonna) si ha che $S_{i,j}$ è univocamente determinato da $S_{i-1,j-1}$, $S_{i-1,j}$, $S_{i-1,j+1}$, ovvero esiste una funzione F che prende $3m$ bit in ingresso e ne restituisce m che calcola le varie posizioni della tabella a partire da quelle immediatamente precedenti.

Tale funzione è booleana[a] e dunque può essere rappresentata come un circuito, che chiameremo $\bar{C}$. Osserviamo che il circuito $\bar{C}$ costruito dipende solamente dalla funzione di transizione δ e *non* dall'input x, pertanto ogni copia di $\bar{C}$ ha una taglia fissata, *costante* rispetto alla taglia dell'istanza!

Inoltre combinando dei circuiti (rispettando il numero di ingressi ed uscite) si ottiene un nuovo circuito: possiamo allora collegare le uscite di tre copie di $\bar{C}$ agli ingressi di un'ulteriore copia di $\bar{C}$ e ottenere un circuito composto.

Costruiamo allora $f(x)$ componendo circuiti in questo modo:

- la prima riga è formata da circuiti costanti che rappresentano la prima riga di T (ovvero l'input seguito da #);
- la prima colonna è formata da circuiti costanti che rappresentano il respingente;
- l'ultima colonna è formata da circuiti costanti che rappresentano il carattere vuoto #;
- le altre posizioni contengono copie del circuito $\bar{C}$: in particolare la copia di $\bar{C}$ che si trova in posizione (i,j) prende in input il risultato dei circuiti che si trovano in posizione $(i-1, j-1)$, $(i-1, j)$ e $(i-1, j+1)$. In totale avremo quindi bisogno di $(n^k - 1) \times (n^k - 2)$ copie di $\bar{C}$.

Inoltre per semplicità chiameremo $C_{i,j}$ la copia di $\bar{C}$ che si trova in posizione i,j.

La funzione f è una riduzione

Dobbiamo ora convincerci che $x \in I$ se e solo se $f(x) \in$ `CIRCUIT-VALUE`, ovvero se e solo se il circuito $C_{n^k,2}$[b] dà come risultato la codifica di `SÌ`: dimostriamo allora che $C_{i,j}$ ha come uscita la codifica di $T(i,j)$ per induzione.

Caso base Per $i = 1$ è banale, in quanto per definizione abbiamo scelto come $C_{1,j}$ il circuito che rappresenta l'input, ovvero il simbolo $T(1,j)$.

Passo induttivo Vogliamo dimostrare che l'uscita del circuito $C_{i+1,j}$ è una sequenza di bit $r_1, \ldots, r_m$ che codifica esattamente il simbolo $T_{i+1,j}$, sapendo che $C_{i,j-1}$, $C_{i,j}$ e $C_{i,j+1}$ codificano i simboli $T_{i,j-1}$, $T_{i,j}$ e $T_{i,j+1}$ rispettivamente.

Ma le uscite $r_1, \ldots, r_m$ di $C_{i+1,j}$ sono calcolate a partire dalle uscite di $C_{i,j-1}$, $C_{i,j}$ e $C_{i,j+1}$ tramite la funzione booleana F, che codifica il comportamento della δ della macchina M. Per ipotesi induttiva i tre circuiti al livello i danno codifiche corrette dei simboli della tabella di computazione, dunque per correttezza di F anche $C_{i+1,j}$ deve essere corretto.

Segue in particolare che per $i = n^k$ il risultato di $C_{n^k,2}$ è la codifica di $T_{n^k,2}$, che è `SÌ` se e solo se $x \in I$. Dunque f è una riduzione da I a `CIRCUIT-VALUE`.

La funzione f è logaritmica in spazio

Infine vogliamo mostrare che f è logaritmica in spazio. Sui nastri di lavoro dobbiamo scrivere

- le porte di ingresso, ovvero l'input: ci basta contare in binario fino ad n^k (che si può fare in spazio logaritmico) e scrivere x seguito da caratteri vuoti;
- gli elementi della prima e ultima colonna, che sono $2n^k$ e quindi ancora esprimibili in spazio logaritmico;
- le copie del circuito $\bar{C}$: ognuna di esse occupa spazio costante e quindi trascurabile, dunque il costo viene solamente da associare ad ognuna i propri indici i, j, che in binario possono essere espressi in spazio logaritmico in n.

Segue che f è calcolabile in spazio logaritmico. □

[a] Abbiamo definito le funzioni booleane solo quando l'output ha una sola uscita, ma combinando più funzioni booleane insieme otteniamo una versione analoga ma a più uscite.

[b] La tabella di computazione dà il suo output in posizione $(n^k, 2)$.

4.2.3 Il Teorema di Cook-Levin

Usando la tabella di computazione definita precedentemente possiamo finalmente dimostrare che `CIRCUIT-SAT` è $\mathcal{NP}$-completo.

Teorema 4.2.7 – $\mathcal{NP}$-completezza di `CIRCUIT-SAT`

`CIRCUIT-SAT` è $\leq_{\mathcal{L}}$-completo per $\mathcal{NP}$.

Dimostrazione. Sia I un problema in $\mathcal{NP}$, ovvero tale che esista una macchina non deterministica N che decide I in tempo polinomiale (non deterministico), ovvero tale che per ogni x esista una sequenza di scelte $(s_1, \ldots, s_t)$ che porta $N(x)$ in `SÌ` se $x \in I$, in `NO` altrimenti. Inoltre tale t è al più n^k, dove $n = |x|$.

Supponiamo per semplicità che il **grado di non determinismo** di I sia al più 2, ovvero che ad ogni passo si possa scegliere al più tra 2 quintuple: allora ogni scelta s_i può essere rappresentata con un singolo bit $s_i \in \{0, 1\}$.

Nella realizzazione della tabella di computazione come circuito [a] aggiungiamo ad ogni

circuito $C_{i,j}$ un ulteriore ingresso, ovvero il bit s_{i-1} che rappresenta la scelta fatta al passo precedente. Otterremo quindi un circuito $\mathcal{C}$ sulle variabili $s_1, \ldots, s_{n^k}$ che rappresenta il problema I. Analogamente a quanto fatto per `CIRCUIT-VALUE`, tale trasformazione è una funzione logaritmica in spazio; inoltre è ovvio che il circuito $f(x)$ risultante da tale funzione sia soddisfacibile se e solo se esiste una sequenza di scelte che mostra che $x \in I$, dunque f riduce I a `CIRCUIT-SAT`.

Infine se il grado di non determinismo di un nodo è $d > 2$ possiamo ordinare le possibilità da 1 a d e indicare la scelta fatta tramite una sequenza di 0 terminata da un 1: il numero 0 indicherà il numero di possibilità scartate, l'1 indicherà finalmente la scelta compiuta. [b] □

[a] vista per bene nella dimostrazione del Teorema 4.2.6

[b] Questo ragionamento è equivalente a dire *non questa, non questa, non questa, . . . , sì questa*.

Da tutto il lavoro fatto finora segue, come preannunciato, il Teorema di Cook-Levin.

Teorema 4.2.8 – Teorema di Cook-Levin

`SAT` è $\leq_{\mathcal{L}}$-completo per $\mathcal{NP}$.

A

Esercizi

A.1 Esercizi di calcolabilità

In questa sezione raccogliamo alcuni esercizi sulla parte di Calcolabilità.

Nessun limite al tempo

Esiste una t calcolabile totale tale che $t(i, n)$ maggiora il tempo di calcolo di $\varphi_i(n)$?

Soluzione. Supponiamo per assurdo esista una tale t calcolabile totale. Essa dovrà essere necessariamente della forma

$$t(i, n) := \begin{cases} k_{i,n}, & \text{se } \varphi_i(n) \text{ converge} \\ 0, & \text{se } \varphi_i(n) \text{ diverge.} \end{cases}$$

dove $k_{i,n}$ è maggiore o uguale del numero di passi di computazione effettuati nel calcolo di $\varphi_i(n)$. Indicheremo il numero di passi esatti con $T_i(n)$.[a]

Dato che t è calcolabile possiamo definire

$$f(x) := \begin{cases} \varphi_x(x) + 1, & \text{se } T_x(x) \leq t(x, x) \\ 0, & \text{altrimenti.} \end{cases}$$

Osserviamo che questa definizione ci dice che $f(x)$ è $\varphi_x(x) + 1$ se $\varphi_x(x)$ converge, 0 altrimenti.

Per la Tesi di Church-Turing esiste i tale che $\varphi_i = f$. Mostriamo che ciò è assurdo: in effetti $\varphi_i(i) \neq f(i)$ in quanto

- se $\varphi_i(i)$ converge, allora $f(i) = \varphi_i(i) + 1 \neq \varphi_i(i)$;
- se $\varphi_i(i)$ diverge, allora $f(i) = 0$.

Segue che t non può essere calcolabile. □

[a] Questa funzione non è necessariamente calcolabile!

Osserviamo che la seconda parte della dimostrazione è identica alla dimostrazione della non-ricorsività di K: in alternativa possiamo quindi mostrare che se t fosse calcolabile allora K sarebbe ricorsivo.

Soluzione alternativa. Supponiamo come sopra che esista una t calcolabile e della forma vista. Allora definiamo

$$f(x) := \begin{cases} 1, & \text{se } t(x,x) > 0 \\ 0, & \text{se } t(x,x) = 0. \end{cases}$$

Tale f è certamente calcolabile, in quanto t lo è.

Ma f è esattamente la funzione caratteristica di K: se $t(i,i)$ è diverso da 0 allora la macchina φ_i calcolata su i termina la sua computazione, e quindi $i \in K$; viceversa se $t(i,i)$ è 0 allora $\varphi_i(i)$ diverge, e quindi i non appartiene a K.

Tuttavia K non è ricorsivo, dunque segue l'assurdo. □

Calcolatori con memoria finita in ciclo

Esiste una funzione calcolabile totale che determina se $\varphi_i(x)$ se uno specifico calcolatore C con memoria finita è in ciclo?

Soluzione. La risposta è sì: siano $Q_C, \Sigma_C, \delta_C, q_0$ le caratteristiche della MdT che modella C.

Siano inoltre $m - 1 := |Q_C|$, $n := |\Sigma_C|$, k la lunghezza del nastro di C. Allora il numero totale di configurazioni è

$$l := k^n \cdot m \cdot k.^a$$

Costruiamo allora una macchina universale U che calcola la macchina M_i su x a partire dallo stato q_0 e con un campo aggiuntivo inizialmente inizializzato a l: ad ogni passo di M_i diminuiamo di 1 il valore di l.

Alla fine del calcolo possiamo controllare il valore finale di l: se è maggiore di 0 allora ci sono ancora degli stati non visitati, dunque M_i non è in ciclo; altrimenti lo è. □

[a] k^n è il numero di possibili nastri, m è il numero di possibiltà per lo stato (incluso il terminatore h), k è il numero di possibilità per la testina.

Esercizio su insiemi non ricorsivi

L'insieme $I := \{ i : \mathrm{dom}(\varphi_i) = \{3\} \}$ è ricorsivo?

Soluzione 1. Dimostriamo che I non è ricorsivo applicando l'Index Set Theorem: abbiamo bisogno di mostrare che $\varnothing \neq I \neq \mathbb{N}$ e che I è un iirf.

(1) I non è vuoto perché la funzione

$$\lambda x. \begin{cases} 1, & \text{se } x = 3 \\ \bot, & \text{alrimenti} \end{cases}$$

è calcolabile (ad esempio è facile scrivere un programma `WHILE` che la calcola) ed ha come dominio esattamente $\{3\}$.

(2) I non è tutto $\mathbb{N}$ in quanto ad esempio la funzione costantemente uguale a 1 è calcolabile ma non ha dominio $\{3\}$.

(3) I è un iirf: supponiamo che $x \in I$ e y sia un altro indice tale che $\varphi_x = \varphi_y$. Ma allora

$$\mathrm{dom}(\varphi_y) = \mathrm{dom}(\varphi_x) = \{3\}$$

e dunque $y \in I$.

Per l'Index Set Theorem segue la tesi. □

Soluzione 2. Mostriamo che $K \leq_{\mathrm{rec}} I$: definiamo

$$\psi(x,y) := \begin{cases} 1, & \text{se } x \in K \text{ e } y = 3 \\ \perp, & \text{altrimenti.} \end{cases}$$

Per la Tesi di Church-Turing insieme al Teorema del Parametro e all'Osservazione 1.6.2 esiste una funzione f calcolabile totale tale che $\varphi_{f(x)}(y) = \psi(x,y)$.

Allora $K \leq_f I$: in effetti

- se $x \in K$ allora $\varphi_{f(x)}$ vale 1 quando l'input è 3 ed è indefinita altrimenti, dunque $\mathrm{dom}(\varphi_{f(x)}) = \{3\}$, ovvero $f(x)$ appartiene ad I;
- se $x \notin K$ allora $\varphi_{f(x)}$ è sempre indefinita e dunque ha dominio vuoto, ovvero $f(x)$ non appartiene a I.

Segue quindi che I non è ricorsivo. □