INTRODUZIONE ALLA

CALCOLABILITÀ

Dispense del corso di Fondamenti di Informatica II

Alberto Marchetti-Spaccamela

A.A. 2020-2021

Prefazione

L'obiettivo principale di queste dispense è rispondere alla seguente domanda:

COSA PUÒ ESSERE CALCOLATO DA UN CALCOLATORE E QUALI SONO GLI ELEMENTI ESSENZIALI DEL CALCOLO?

Questa domanda è la domanda fondamentale della calcolabilità, che studia aspetti della teoria dell'informatica ma la cui importanza non è limitata all'informatica stessa perché la domanda non è ristretta a cosa possiamo fare con macchine di calcolo ma riguarda anche ciò che è possibile calcolare e quello che, invece, non si può calcolare.

Lo studio di questi aspetti è iniziato ben prima che fosse disponibile un calcolatore reale (almeno come viene inteso ai giorni nostri). Infatti negli anni trenta del secolo scorso diversi ricercatori si chiesero quale fosse il vero significato della parola *"calcolabile"*. Il loro interesse non era limitato ai calcoli matematici ma a quali problemi potessero essere risolti con metodi algoritmici nell'ambito della matematica e della logica.

Le loro ricerche studiavano le propietà e i limiti dei modelli di calcolo astratti. Vedremo nel seguito due di questi modelli di calcolo; per il momento è sufficiente sapere che un modello di calcolo è come una macchina in grado di eseguire programmi composti da istruzioni che codficano un algoritmo di soluzione. La macchina non deve essere fisicamente realizzabile e per questo parliamo di macchine astratte e i programmi non sono da intendersi come i programmi che scriviamo in Python o in un altro linguaggio di programmazione: infatti, le operazioni che questi modelli sono in grado di esguire direttamente sono poche e molto semplici; per questa ragione eseguire operazioni come la moltiplicazione di due numeri ǹon banale. (Ricordiamo infatti che i primi calcolatori programmabili furono realizzati successivamente).

I ricercatori interessati alla calcolabilità avevano due obiettivi principali:

1. stabilire i limiti dei modelli di calcolo e stabilire se esistevano calcoli "impossibili" per i diversi modelli di calcolo;

2. individuare gli elementi essenziali del calcolo, cioè erano interessati a definire modelli di calcolo molto semplici che tuttavia avessero la capacità di risolvere tutti i problemi risolubili con modelli più complessi.

Per quanto riguarda il punto 1 precedente, è stato possibile stabilire che i modelli di calcolo e i linguaggi di programmazione che utilizziamo non sono in grado di risolvere tutti i problemi. Infatti vedremo nel capitolo 2 che

1) ESISTONO PROBLEMI CHE NON POSSONO ESSERE RISOLTI

Nel seguito vedremo usando i risultati proposti da Cantor per lo studio della cardinalit‘egli insieme infiniti che i problemi che non possono essere risolti sono infiniti. Dimostreremo inoltre che il problems di decidere se un programma con un dato input si ferma o cicla non è risolubile; questo risultato vale per i linguaggi di programmazione che si utilizzano (Python, Java, C etc.).

Per quanto riguarda il punto 2 - individuare gli elementi essenziali del calcolo - l'obiettivo era quello di realizzare un modello il più semplice possibile ma che tuttavia avesse le stesse capacità di calcolo dei modelli più potenti. Furono proposti e analizzati diversi modelli di calcolo. Vedremo nel seguito due esempi di questi modelli: la macchina di Turing e le RAM. Il motivo di questa scelta sta nel fatto che questi modelli sono probabilmente quelli più simili agli odierni calcolatori e, quindi, più naturale in un corso di laurea in informatica e per chi conosce l'architettura interna di un calcolatore secondo l'architettura di von Neumann, e che è abituato a considerare un calcolatore come uno strumento di manipolazione di simboli mediante programmi (che a loro volta non sono altro che sequenze di simboli). Questi modelli di calcolo anche se dotati di operazioni elementari sono in grado di calcolare tutto quello che possiamo fare con i moderni linguaggi di programmazione che usiamo.

Nella ricerca di un processo automatico per rispondere a questioni matematiche, Turing prevedeva un dispositivo completamente meccanico, quasi una sorta di macchina da scrivere con memoria.

L'importanza della macchina di Turing deriva dal fatto che essa è sufficientemente complicata da affrontare sofisticate questioni matematiche, ma sufficientemente semplice per essere oggetto di analisi dettagliata. Turing ha usato la sua macchina come un costrutto teorico per dimostrare l'esistenza di problemi non risolubili.

L'interesse iniziale di Turing era la matematica astratta, decidere quali problemi matematici sia possibile risolvere, piuttosto che il calcolo pratico come noi lo concepiamo oggi. Tuttavia, stabilendo l'idea di automatizzare dimostrazioni matematiche astratte piuttosto che automatizzare il calcolo aritmetico, Turing ha stimolato lo sviluppo di elaborazione delle informazioni in modo generale. Questo è avvenuto nei giorni in cui non esistevano computer e per questa sua visione straordinaria del futuro Turing è considerato il padre dell'informatica.

Come abbiamo anticipato, oltre alla macchina di Turing, sono stati proposti altri modelli di calcolo, quali il λ-calcolo (proposto da Church), gli algoritmi di Markov, le macchine di Post e le funzioni ricorsive. Questi modelli sono apparentemente molto diversi fra loro ma, inaspettatamente, è stato mostrato che tutti i modelli proposti equivalenti tra di loro: ci'o che poteva essere calcolato in un dato modello, poteva essere calcolato in un qualunque altro modello. Questi risultati, portarono a formulare quella che oggi è nota come la tesi di Church-Turing che vedremo nel capitolo 3 e che afferma

2) QUELLO CHE POSSIAMO CALCOLARE LO POSSIAMO FARE ANCHE CON MACCHINE E LINGUAGGI MOLTO SEMPLICI: È CALCOLABILE TUTTO CIÒ CHE PUÒ ESSERE CALCOLATO DA UNA MACCHINA DI TURING.

Per quanto detto in precedenza, nella formulazione della tesi di Church-Turing avremmo potuto sostituire una macchina di "Turing" con, ad esempio, "un algoritmo di Post".

La domanda precedente su cosa possiamo calcolare non considera quanto sia costoso (in termini di tempo e di memoria) eseguire determinati calcoli. In altre parole siamo interessati a sapere cosa possiamo effettivamente calcolare in pratica; infatti, non ha molto senso affermare che un problema è risolubile se la sua soluzione richiede di utilizzare tutti i calcolatori del pianeta per mille anni. La teoria della complessità che si è sviluppata a partire dagli anni settanta del secolo scorso ha come obiettivo lo studio e la caratterizzazione dei problemi effettivamente risolubili. La teoria si occupa di definire cosa è calcolabile in modo efficiente e quali sono i possibili approcci nei casi di problemi che sono difficili.

In una parte successiva di questo corso vedremo che

3) NON TUTTO CIÒ CHE POSSIAMO CALCOLARE È CALCOLABILE IN MODO EFFICIENTE: QUELLO CHE POSSIAMO CALCOLARE IN PRATICA SONO QUEI PROBLEMI RISOLUBILI IN TEMPO POLINOMIALE. INOLTRE ESISTONO (TANTI) PROBLEMI CHE NON POSSONO ESSERE RISOLTI IN TEMPO POLINOMIALE E, QUINDI, NON SONO RISOLUBILI IN MODO EFFICIENTE.

L'affermazione precedente quando fu implicitamente formulata quella che è probabilmente la questione aperta piiimportante nel campo dell'informatica teorica, ovvero la congettura $P \neq NP$, la quale afferma quanto segue.

Nonostante centinaia di ricercatori abbiamo tentato di dimostrare tale congettura, quest'ultima è ancora irrisolta: per dare al lettore un'idea della sua significatività, basti pensare che una fondazione americana ha istituito un premio di un milione di dollari a chi la proverà.

La trattazione che faremo nel seguito per rispondere alla domanda iniziale è limitata e non entra nei dettagli di una teoria che non ha solo interesse per l'informatica ma che per sua natura investiga i limiti del ragionamento e della conoscenza umana. Non sorprendentemente esistono numerosi libri di testo dedicati agli argomenti trattati in queste dispense: in particolare, i seguenti due volumi hanno ispirato in parte la stesura delle dispense stesse e sono consigliati come letture aggiuntive:

- P. Crescenzi, *Informatica teorica*, dispense disponibili sul sito dell'autore (U. Firenze).
- D. Harel, Y.Feldman, *Algoritmi, Lo spirito dell'informatica*, Springer 2007.

1 Insiemi finiti e infiniti

Questo capitolo è dedicato a studiare gli insiemi con particolare attenzione agli insiemi infiniti. Prima di procedere ci chiediamo perché studiare insiemi infiniti in un corso sui fondamenti di informatica. Infatti i dati che possono essere memorizzati nella memoria di un computer sono finiti e per quanto grandi siano le dimensioni queste sono sempre comunque finite. Questa considerazione non deve escludere l'infinito dall'attenzione dell'informatica prché è utile progettare programmi che siano in grado di operare con numeri che abbiano un numero infinito di cifre o che possano ricevere sequenze di dati potenzialmente infinite. Analogamente, osserviamo che anche se la maggioranza dei fisici pensa che il nostro universo e l'energia in esso contenuta siano finiti questa non è una ragione sufficiente per non considerare l'idea di infinito in fisica.

Oltre a questa motivazione vedremo che il principale motivo per studiare insiemi infiniti in un corso sui fondamenti dell'informatica è che il loro studio ci permette di studiare in modo formale i limiti dei calcolatori e provare semplicemente l'esistenza di problemi non risolubili con un calcolatore, indipendentemente dalla potenza di calcolo della macchina o dal tempo di calcolo.

1.1 Definizioni

Informalmente un *insieme* è una collezione di oggetti che sono gli *elementi* dell'insieme; gli elementi dell'insieme possono essere qualunque cosa: numeri, persone e anche altri insiemi. Di solito un insieme si rappresenta indicando i suoi elementi fra parentesi graffe, come ad esempio

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$B = \{Antonio, Beatrice, Carlo, Daria\}$$
$$C = \{\{a, b\}, \{c, d\}, \{a, c\}\}$$
$$D = \{2^i, i = 1, 2, \ldots\}$$

Il terzo insieme è un insieme di insiemi, mentre il quarto rappresenta l'insieme infinito delle potenze intere e positive di due.

L'ordine degli elementi non conta e, quindi, $\{a, b\}$ e $\{b, a\}$ sono lo stesso insieme scritto in due modi diversi. Inoltre un elemento fa parte o non fa parte dell'insieme e, quindi, non esiste la possibilità che un elemento compaia due o più volte nell'insieme. Quindi $\{a, a\} = \{a\}$.

Useremo la seguente notazione:

- Denotiamo l'*appartenenza* di un elemento ad un insieme con il simbolo $\in$ e la non appartenenza ad un insieme con $\notin$.
  Con riferimento agli esempi di insiemi precedenti abbiamo quindi che $0 \in A$ denota che 0 è un elemento dell'insieme A, e che $Luca \notin D$ indica la non appartenenza di *Luca* all'insieme D.

- Indichiamo l'insieme vuoto che non ha alcun elemento con il simbolo $\emptyset$.

- Denotiamo con $B \subseteq A$ il fatto che B è un *sottoinsieme* di A.
  Questo equivale ad assumere che, se $B \subseteq A$, allora ogni elemento di B è anche elemento di A.

- Denotiamo con $B \subset A$ il fatto che che B è un *sottoinsieme proprio* di A cioè che B è un sottoinsieme di A ma che i due insiemi non sono uguali.
  Pertanto, se $B \subset A$, esiste almeno un elemento $b \in B$ che non appartiene a A. È facile vedere che se $A \subset B$ allora $A \subseteq B$ e $B \not\subseteq A$.

- Denotiamo con $A = B$ il fatto che i due insiemi A e B sono uguali, hanno cioè stessi elementi. Chiaramente, $A = B$ se e solo se $A \subseteq B$ e $B \subseteq A$.

1.1.1 Operazioni su insiemi

Le principali operazioni definite su insiemi sono:

- L'*unione* di insiemi X e Y si denota con $X \cup Y$ e contiene tutti gli elementi che compaiono in X o Y
- L'*intersezione* di insiemi X e Y si denota con $X \cap Y$ e contiene tutti gli elementi che compaiono sia in X che Y
- La *differenza* di insiemi X e Y si denota con $X-Y$ e contiene tutti gli elementi che compaiono in X ma non in Y

In alcuni casi possiamo considerare un particolare dominio di elementi D. In questo caso dato un sottoinsieme A di D denotiamo con $\bar{A}$ il *complemento* di A, cioè l'insieme di tutti gli elementi di D che non sono in A. Abbiamo quindi che $\bar{A} = D - A$.

Infine, dato un insieme A denotiamo con $\mathcal{P}(\mathcal{A})$ l'insieme di tutti i sottoinsiemi di A che è detto *insieme potenza*. L'insieme $\mathcal{P}(\mathcal{A})$ include sia l'insieme vuoto che A fra i suoi elementi. Ad esempio se $A = \{Testa, Croce\}$ abbiamo che

$$\mathcal{P}(A) = \{\emptyset, \{Testa\}, \{Croce\}, \{Testa, Croce\}\}$$

Teorema 1. *Le operazioni di* $\cup$ *and* $\cap$ *verificano la proprietà distributiva:*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

1.2 Cardinalità di un insieme

La cardinalità di un insieme rappresenta la dimensione dell'insieme che denotiamo con due barre verticali ai lati dell'insieme; quindi $|A|$ rappresenta la cardinalità dell'insieme A.

Definizione 1. La cardinalità di un insieme A è denotata con $|A|$ e rappresenta il numero di elementi in A.

Nel caso di insiemi finiti la cardinalità è un concetto semplice: la cardinalità di un insieme finito è data dal numero di elementi dell'insieme ed è, quindi, un numero intero non negativo (possibilmente zero se l'insieme è vuoto).

Nel caso di insiemi infiniti il concetto di cardinalità è complesso e non facilmente intuibile. Una difficoltà è data dal fatto che è difficile immaginare insiemi infiniti. Per questa ragione lo studio di insiemi infiniti si basa essenzialmente sul confronto reciproco di insiemi. In particolare, dati due insiemi A e B, li confronteremo usando le proprietà di funzioni f, del tipo $f : A \to B$.

Definizione 2. Dati due insiemi (finiti o infiniti) A e B e una funzione $f : A \to B$ diciamo che

- f è *suriettiva* se per ogni elemento $b \in B$ esiste almeno un elemento di $a \in A$ tale che $f(a) = b$;
- f è *iniettiva* dati a, a', $a \in A, a' \in A, a \neq a'$ abbiamo $f(a) \neq f(a')$
- f è una *corrispondenza biunivoca* se f è suriettive e iniettiva.

In altre parole una funzione suriettiva "copre" tutti gli elementi di B e non ne lascia nessuno scoperto. In questo caso possiamo affermare che $|A| \geq |B|$. Abbiamo inoltre che $f : A \to B$ è suriettiva se e solo se la funzione inversa f^{-1}, $f^{-1} : B \to A$ è iniettiva. Quindi possiamo affermare che una funzione $f : A \to B$ è una corrispondenza biunivoca se sia f che la funzione inversa f^{-1} sono suriettive.

La definizione precedente ci permette di confrontare la cardinalità di insiemi. Infatti, se la cardinalità di A è strettamente maggiore della cardinalità di B allora possiamo trovare una funzione suriettiva da A a B.

Ad esempio supponiamo che A e B siano insiemi finiti; in particolare sia A l'insieme dei dei colori fondamentali $A = \{giallo, rosso, blu\}$ e B l'insieme delle due possibili facce di una moneta $B = \{testa, croce\}$. In questo caso possiamo definire un funzione suriettiva f, $f : A \to B$, nel seguente modo:

$$f(giallo) = testa,\ f(rosso) = croce,\ f(blu) = croce$$

Consideriamo una corrispondenza biunivoca $f : A \to B$. Poiché f è suriettiva, allora $|A| \geq |B|$. Dato che f è una corrsipondenza biunivoca allora la funzione inversa f^{-1}, $f^{-1} : B \to A$ è anch'essa una funzione suriettiva e, quindi, possiamo affermare anche che $|B| \geq |A|$ e, quindi, possiamo concludere $|A| = |B|$.

Il seguente teorema riassume le considerazioni fatte.

Teorema 2. *Dati due insiemi (non necessariamente finiti) A e B, abbiamo che*
i) $|A| \geq |B|$ se esiste una funzione suriettiva $f : A \to B$
ii) $|A| > |B|$ se esiste una funzione suriettiva $f : A \to B$ ma non esiste una funzione suriettiva $g : B \to A$
iii) $|A| = |B|$ se esiste una corrispondenza biunivoca fra A e B.

Teorema 3. *Se un insieme finito A contiene n elementi allora $\mathcal{P}(\mathcal{A})$ contiene 2^n elementi.*

Il teorema precedente motiva il fatto che spesso per denotare l'insieme potenza di A usiamo anche la notazione 2^A.

1.3 Insiemi infiniti: insiemi numerabili e non numerabili

Un insieme infinito ben conosciuto è l'insieme dei numeri naturali che denotiamo con $\mathbb{N}$ e che contiene 0 e tutti gli interi positivi: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$

Il fatto che $\mathbb{N}$ sia un insieme infinito è intuitivo: per ogni dato intero i esiste un intero $i + 1$ che è più grande di i. In altre parole non esiste un numero che è più grande di ogni altro numero e cercare di scrivere i numeri in $\mathbb{N}$ è un compito infinito perché qualunque sia il numero di interi che scriviamo ne mancano sempre tanti. Possiamo quindi affermare che $\mathbb{N}$ è un insieme infinito.

Nel seguito vedremo che non esiste un solo tipo di infinito ma è possibile definire infiniti gradi di infinito. Immaginare infiniti gradi di infinito non è affatto semplice per la mente umana e questo concetto non può essere spiegato a parole.

Il problema principale per comprendere il concetto di infinito è capire che non siamo capaci di immaginare un qualunque insieme infinito: non possiamo vederli o scriverli e, quindi, non possiamo parlare di dimensione dell'insieme infinito. Quello che faremo è confrontare insiemi utilizzando il Teorema 3 e un metodo di prova introdotto da Cantor, nell'ottocento.

1.3.1 Insiemi numerabili

Definizione 3. Un insieme A è *numerabile* se i suoi elementi possono essere ordinati

$$a_1, a_2, a_3, a_4 \ldots, a_n, \ldots$$

.

La definizione precedente implica che se un insieme è finito allora è anche numerabile e che $\mathbb{N}$ è un insieme numerabile (infatti $0, 1, 2, 3, \ldots$ è un possibile ordinamento di $\mathbb{N}$). Inoltre, dato un insieme infinito A numerabile, allora possiamo ordinare i suoi elementi e, quindi, definire una corrispondenza biunivoca fra $\mathbb{N}$ e A. Riassumiamo le considerazioni fatte.

- $\mathbb{N}$ è un insieme *infinito numerabile.*
- Un insieme A è *infinito numerabile* se e solo esiste una corrispondenza biunivoca fra $\mathbb{N}$ e A.
- Un insieme è *numerabile* se e solo se è finito o infinito numerabile.

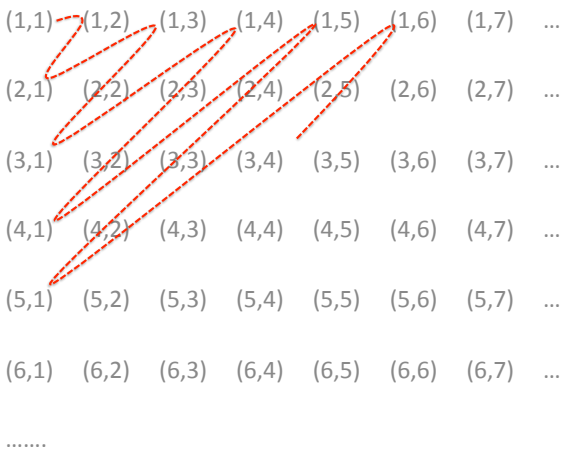

Figure 1: La linea tratteggiata mostra l'ordinamento degli elementi dell'insieme $(\mathbb{N} \times \mathbb{N})$.

Le considerazioni precedenti ci permettono di mostrare che altri insiemi sono numerabili. In particolare l'insieme $\mathbb{Z}$ di tutti gli interi (positivi, negativi e con lo 0) è numerabile perché possiamo enumerare gli interi positivi e negativi nel seguente modo

$$0, 1, -1, 2, -2, 3, -3 \ldots$$

Formalmente l'ordinamento precedente utilizza la corrispondenza biunivoca f così definita

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ è pari} \\ -(n+1)/2 & \text{se } n \text{ è dispari} \end{cases}$$

Possiamo anche mostrare che l'insieme $(\mathbb{N} \times \mathbb{N})$ formato dalle coppie (i, j) di interi non negativi è numerabile. Per fare questo utilizziamo la Figura 1 che rappresenta gli elementi dell'insieme $(\mathbb{N} \times \mathbb{N})$ in forma tabellare. In particolare, la prima riga rappresenta tutti gli elementi in cui il primo elemento della coppia è 1; la seconda riga gli elementi in cui il primo elemento della coppia (i, j) è 2 e così via. Si noti che la colonna j rappresenta tutte le coppie del tipo (i, j). Non diamo la definizione formale della corrispondenza biunivoca in questo caso e ci limitiamo ad osservare che la linea tratteggiata rappresenta un possibile oridnamento degli elementi dell'insieme $(\mathbb{N} \times \mathbb{N})$; petanto in base alla definizione l'insieme è numerabile.

Dopo aver provato che le coppie di interi sono numerabili proviamo che l'insieme dei numeri razionali positivi $\mathbb{Q}^+$ è anch'esso numerabile. Questo a prima vista potrebbe sorprendere perché i numeri razionali riempiono lo spazio fra gli interi. In particolare ricordiamo la definizione di numero razionale

Definizione 4. q è un numero razionale positivo se esistono due interi positivi n e m tale che

$$q = \frac{n}{m}$$

Innanzitutto osserviamo che $|\mathbb{Q}^+| \geq |\mathbb{N}|$ mostrando una funzione suriettiva f,, $f : \mathbb{Q}^+ \to \mathbb{N}$ (che lasciamo come facile esercizio).

Rimane quindi da dimostrare che $|\mathbb{N}| \geq |\mathbb{Q}^+|$. Dato che $\mathbb{N}$ e $(\mathbb{N} \times \mathbb{N})$ sono entrambi numerabili è sufficiente mostrare una funzione suriettiva f, $f(\mathbb{N} \times \mathbb{N}) \to \mathbb{Q}^+$. Analogamente al caso dell'insieme

Figure 2: La linea tratteggiata mostra un possibile ordinamento dei numeri razionali.

$\mathbb{N} \times \mathbb{N}$ la figura 2 mostra una tale funzione. Notiamo che per ogni razionale positivo q esistono molte coppie (n, m) tali che $q = \frac{n}{m}$. Ad esempio,per $q = 1,5$ abbiamo

$$q = \frac{3}{2}, q = \frac{6}{4}, q = \frac{9}{6}, \ldots$$

In questo caso possiamo concludere che la riga tratteggiata nella figura mostra una funzione suriettiva $f : (\mathbb{N} \times \mathbb{N}) \to \mathbb{Q}$ e che quindi mostra che $|\mathbb{Q}| \leq |(\mathbb{N} \times \mathbb{N})$.

Dopo aver mostrato che i razionali positivi sono numerabili possiamo anche provare che i numeri razionali (positivi e negativi) sono numerabili nello stesso modo con cui abbiamo dimostrato che $\mathbb{Z}$ è numerabile.

Concludendo abbiamo il seguente teorema.

Teorema 4. *I seguenti insiemi sono infiniti numerabili:* $\mathbb{Z}, \mathbb{N} \times \mathbb{N}, \mathbb{Q}^+, \mathbb{Q}$ *e quindi*

$$|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{N} \times \mathbb{N}| = |\mathbb{Q}^+| = |\mathbb{Q}|$$

I risultati precedenti evidenziano una significativa differenza fra insiemi finiti e insiemi infiniti. Nel caso di insiemi finiti aggiungere nuovi elementi all'insieme rende l'insieme più grande. In particolare, dato un insieme finito A ed un elemento $b \notin A$ abbiamo $|A \cup \{b\}| = |A| + 1$, cioè aggiungendo b aumentiamo la cardinalità di A. Questo implica anche che se A è un insieme finito che è sottoinsieme proprio di B, cioè $A \subset B$, allora $|A| < |B|$.

Queste proprietà non valgono per insiemi infiniti. Infatti osservaimo che $\mathbb{N} \subset \mathbb{Q}$: infatti un numero naturale è anche un razionale ed esistono infiniti numeri razionali che non sono interi. Tuttavia abbiamo anche mostrato che $\mathbb{N}$ e $\mathbb{Q}$ sono ambedue numerabili.

1.3.2 Non uno ma tanti infiniti

In questa sezione vediamo il risultato fondamentale di Cantor che ha mostrato l'esistenza di insiemi di insiemi infiniti non numerabili e quindi di cardinalità superiore ai naturali. In particolare Cantor ha mostrato che l'insieme dei reali non sono numerabili.

Teorema 5. *L'insieme dei numeri reali* $\mathbb{R}$ *non è numerabile:* $|\mathbb{R}| > |\mathbb{N}|$.

La prova di Cantor mostra che non esiste una funzione suriettiva fra $\mathbb{N}$ e $\mathbb{R}$; per il Teorema 2 questo implica che $|\mathbb{R}| > |\mathbb{N}|$. Prima di vedere la prova geniale di Cantor sulla non esistena di

una funzione suriettiva fra $\mathbb{N}$ e $\mathbb{R}$ osserviamo che in matematica e, in genere nelle scienze esatte, dimostrare risultati di non esistenza o l'impossibilità di un evento sono in genere i risultati più difficili da provare perché bisogna mostrare che ogni possibile approccio non funziona. Ad esempio, nel nostro caso dobbiamo mostrare che tutte le possibili funzioni $f : \mathbb{N} \to \mathbb{R}$ nessuna è suriettiva: la prova di questo non è banale perché le possibili funzioni $f : \mathbb{N} \to \mathbb{R}$ sono infinite.

Non potendo analizzare una ad una tutte le funzioni fra $\mathbb{N}$ e $\mathbb{R}$ la prova del teorema è per contraddizione. Assumiamo che esista una funzione suriettiva f, fra i naturali e i reali. Ovviamente se i numeri reali sono numerabili lo sono anche i numeri reali in $[0, 1)$ e, quindi, possiamo assumere che esiste una tabella T come la seguente che contiene in qualche ordine tutti i numeri reali fra 0 e 1.

Table 1: Ipotetico ordinamento dei numeri reali

y_1	0,	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	...
y_2	0,	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	...
y_3	0,	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	...
y_4	0,	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	...
y_5	0,	.	.	.	.	.	...
.	0,	.	.	.	.	.	...
.	0,	.	.	.	.	.	...

La prima riga mostra il primo numero reale nell'ordinamento una cifra per colonna; il primo numero della lista è, quindi,

$$y_1 = 0, a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, \ldots$$

dove i simboli $a_{1,1}, a_{1,2}, a_{1,3}, a_{1,4}, \ldots$ sono le cifre del primo numero. In generale avremo che l'i-esimo numero reale della nostra lista è

$$y_i = 0, a_{i,1}, a_{i,2}, a_{i,3}, a_{i,4}, \ldots$$

Ora mostriamo che esiste un numero reale che non fa parte della lista T, contraddicendo quindi l'ipotesi che T contenga tutti i numeri reali.

Mostriamo ora che esiste un numero reale in $[0, 1]$ che non appartiene alla lista. Assumiamo che $x = 0, b_1 b_2 b_3 b_4 \ldots$; il valore di della i-esima cifra decimale b_i di x è determinato sulla base di $a_{i,i}$, la i-esima cifra dell'i-esimo numero dell'ordinamento dei numeri reali. In particolare se $a_{i,i} \neq 0$ allora $b_i = a_{i,i} - 1$ se $a_{i,i} = 0$ allora $b_i = 1$.

Notiamo che x è diverso da y_1 nella prima cifra decimale, da y_2 nella seconda cifra decimale e così via. Abbiamo quindi mostrato che x è diverso da ogni numero y_i almeno nella posizione i. Quindi abbiamo mostrato che y è diverso da ogni numero nella lista T. Quuindi T non è una lista di tutti i numeri reali in $[0, 1]$. Poiché il ragionamento precedente può essere applicato ad un qualunque ordinamento dei numeri possiamo concludere che l'insieme dei numeri reali ha cardinalità superiore a $|\mathbb{N}|$.

Esempio.
Per esemplificare l'idea di Cantor assumiamo uno specifico ordinamento come ad esempio quello dato nella seguente tabella.

Dato che le cifre sulla diagonale sono $3, 0, 1, 8 \ldots$ e, quindi, le prime cifre del numero x che definiamo sono $x = 0, 2907 \ldots$ □

Il metodo di prova proposto da Cantor è noto come *diagonalizzazione*: l'idea infatti utilizza la diagonale della tabella che rappresenta l'ipotetica lista dei numeri per definire un numero diverso da ogni numero nella lista. L'idea può essere applicata per mostrare altri risultati. Vediamo ora come dimostrare che l'insieme delle parti di un insieme infinito A ha cardinalità superiore a A.

Table 2: Ordinamento dei numeri reali

y_1	0,	3	1	7	5	2	...
y_2	0,	2	1	2	8	1	...
y_3	0,	4	4	0	9	7	...
y_4	0,	1	5	6	8	2	...
y_5	0,	.	.	.	.	.	...
.	0,	.	.	.	.	.	...
.	0,	.	.	.	.	.	...

Teorema 6. *Dato un insieme infinito A, $|\mathcal{P}(A)| > |A|$.*

É facile mostrare che $|\mathcal{P}(A)| \geq |A|$. A tale scopo è sufficiente mostrare che esiste una funzione suriettiva fra $\mathcal{P}(A)$ e A. Lasciamo questa dimostrazione per esercizio. Per completare il teorema in base al Teorema 2 dobbiamo mostrare che non esiste una funzione suriettiva f, $f : A \to \mathcal{P}(A)$.

Come nel teorema precedente la prova della non esistenza di tale funzione è per contraddizione. Assumiamo che il teorema non sia vero e che quindi esista una funzione suriettiva f che associa ad ogni elemento $a \in A$ un insieme $S \in \mathcal{P}(A)$. Vedremo come sia possibile costruire un insieme T, $T \in \mathcal{P}(A)$ tale che non esiste a, $a \in A$ per cui $f(a) = T$.

Consideriamo per ogni a, $a \in A$ l'insieme $f(a)$ e definiamo un insieme T nel seguente modo; a appartiene a T se e solo se $a \notin f(a)$. In questo modo T è diverso da ogni insieme $f(a)$ per ogni $a \in A$. In questo modo abbiamo provato che f non è una funzione suriettiva e il teorema è provato.

Il teorema precedente permette di ottenere un risultato inatteso: esistono infiniti gradi di infinito. Infatti applicando il teorema precedente all'insieme $\mathbb{N}$ ne segue che $N' = \mathcal{P}(\mathbb{N})$ ha cardinalità superiore a $\mathbb{N}$. Ora se applichiamo il teorema a $\mathcal{P}(\mathbb{N})$ otteniamo che l'insieme $\mathcal{P}(N')$ ha cardinalità superiore a N'. Procedendo in questo modo otteniamo un numero infinito di insiemi di cardinalità ogni volta maggiore.

1.4 Paradossi

I risultati precedenti mostrano che strutture matematiche apparentemente semplici come gli insiemi possano essere complesse da trattare. Infatti, i primi tentativi di formalizzare il concetto di insieme hanno portato a conclusioni paradossali. In particolare la possibilità di definire insiemi in modo autoreferenziale permette di ottenere conclusioni paradossali. Ricordiamo che un paradosso è un'affermazione che non possiamo mostrare nè vera nè falsa. I seguenti paradossi sono stati proposti nel passato.

Nel seguito vediamo alcuni esempi che sono stati proposti nell'antichità. Ecco alcuni esempi:

- Epimenide di Creta (VI sec. a. C.):
  Tutti i cretesi sono bugiardi.

  Questa affermazione non può essere vera ma può essere falsa; quindi non è un vero paradosso.

- Eubulide di Mileto (IV sec. a. C.):
  Questa frase è falsa oppure io sto mentendo.

  La frase precedente non può essere nè vera nè falsa. Infatti se sto mentendo la frase vèra e questo contraddice il fatto che sto mentendo; se sto dicendo la verità allora la frase deve essere falsa e questo contraddice il fatto che sto dicendo la verità.

- Buridano (XIV sec.):
  Socrate dice: "Platone dice il falso"
  Platone dice: "Socrate dice il vero"

  Le due affermazioni congiunte sono un paradosso.

Agli inizi del secolo scorso uno dei temi di maggiore rilevanza per i matematici era poter dimostrare che la matematica potesse essere fondata a partire da un insieme di assiomi e da un insieme di regole la cui ripetuta assunzione per il matematico G. Frege aveva proposto una teoria assiomatica degli insiemi. La teoria di Frege fu distrutta dal giovane matematico B. Russell[1] che propose un paradosso che affondava la teoria di Frege.

- Bertrand Russell
  Sia S sia una variabile i cui valori sono possibili insiemi.

  L'insieme di tutti gli insiemi che non appartengono a se stessi, appartiene o no a se stesso?

  Per verificare come l'affermazione precedente sia effettivamente un paradosso definiamo T come l'insieme degli insiemi che non appartengono a se stessi; formalmente

  $$T = \{S | S \notin S\}$$

  Per definizione di T abbiamo che per ogni S, $S \in T$ se e solo se $S \notin S$ Osserviamo però se poniamo $S = T$ possiamo ottenere una contraddizione. Infatti abbiamo

  $$T \in T \text{ se e solo se } T \notin T$$

Vediamo infine altri paradossi che esprimono in modo non strettamente matematico il paradosso precedente.

- *In un villaggio il barbiere rade tutti e soli coloro che non si radono da soli. Chi rade il barbiere?* (B.Russell)

  Se il barbiere non si rade da solo allora si contraddice la prima affermazione (il barbiere rade chi non si rade da solo). Se il barbiere si rade da solo allora in questo modo si contraddice l'affermazione che il barbiere non rade chi si rade da solo.

- *In una biblioteca può esistere un catalogo di tutti i cataloghi bibliografici che non contengono se stessi?* (Gonseth)

  Per mostrare che questa affermazione è un paradosso possiamo ragionare in modo analogo al caso precedente.

Osserviamo che tutti e tre i paradossi precedenti descrivono insiemi e situazioni in modo autoreferenziale. Vedremo nel prossimo capitolo come questo meccanismo autoreferenziale sia utile per provare un importante risultato fondamentale dell'informatica.

Non è questa la sede per discutere come ottenere una teoria degli insiemi che eviti il paradosso di Russell. Non entriamo nei dettagli e ci limitiamo ad osservare che un possibile modo per eliminare il paradosso è assumere che T *non possa essere un insieme.* In questo modo nella prova del paradosso non possiamo pi ù assumere che T sia un possibile valore per la variabile S. In questo modo eliminiamo alla radice il problema.

[1]Bertrand Russell è diventato famoso poco più che trentenne per il paradosso che porta il suo nome e per i suoi studi sui fondamenti della logica e della matematica. Successivamente ritenne di essere troppo vecchio per continuare a fare una buona ricerca in matematica e iniziò a studiare e a scrivere di filosofia. Successivamente, quando ritenne di essere troppo vecchio per fare filosofia iniziò a scrivere di politica. Fu un pacifista convinto e uno dei pionieri dell'obiezione di coscienza; per le sue convinzioni fu carcerato per aver obiettato alla leva durante la prima guerra mondiale e per le sue manifestazioni di dissenso. Nel 1950 ricevette il Premio Nobel per la letteratura per la sua opera di divulgazione filosofica.

2 Problemi indecidibili: il problema della fermata

Nella sezione precedente abbiamo definito un insieme infinito numerabile se i suoi elementi possono essere numerati definendo il primo elemento, il secondo, il terzo e via via fino a considerare tutti gli elementi dell'insieme. In questa In questa sezione studieremo i limiti della calcolabilità mostrando che esistono funzioni e problemi che **non possono essere risolti da un computer**. L'affermazione è valida in generale; infatti l'impossibilità non è dovuta a capacità di calcolo limitata dei calcolatori oggi disponibili, o al linguaggio di programmazione che utilizziamo o al fatto che nuovi algoritmi di soluzione devono essere scoperti. Il limite che vedremo è intrinsico alla capacità logica umana e per questa ragione invalicabile; per questa ragione i problemi che non possono essere risolti sono detti **indecidibili**.

Vedremo prima che i problemi che non sappiamo risolvere sono infiniti ma senza fornire un esempio specifico di problema irrisolubile. Successivamente daremo esempi concreti di problemi che non si possono risolvere. A questo scopo utilizzeremo le idee e il metodo introdotto da Cantor nella prova che i numeri reali non sono numerabili. In particolare vedremo nel seguito come il metodo della diagonalizzazione possa essere adattato per provare che un particolare problema di notevole importanza per la programmazione sia indecidibile.

2.1 Ci sono più problemi che programmi per risolverli

In questa sezione confronteremo il numero di programmi con il numero dei possibili problemi e vedremo che l'insieme dei problemi diversi fra loro ha cardinalità maggiore del numero di programmi. Per questa ragione devono esistere necessariamente infiniti problemi per cui non abbiamo un programma di soluzione.

2.1.1 Il numero dei programmi è numerabile

Nel seguito facciamo riferimento ad uno specifico linguaggio di programmazione, ad esempio Python. Ovviamente il numero dei programmi Python che possiamo scrivere è infinito: dato un programma Python possiamo scrivere un programma con una istruzione in più. Mostriamo nel seguito il seguente teorema che stabilisce che il numero dei possibili programmi è numerabile.

Teorema 7. *Il numero dei programmi ha cardinalità pari a* $|\mathbb{N}|$.

Per dimostrare il teorema analiizziamo quanti testi possiamo scrivere usando un computer o una macchina da scrivere. Ogni testo è una sequenza di simboli della tastiera; oltre ai caratteri alfabetici (maiuscoli e minuscoli) dobbiamo includere nell'insieme di simboli ance le cifre $(0, 1, 2 \ldots, 9)$, i simboli di punteggiature (";", ":", "!", "." ecc.) e un insieme di simboli come ad esempio: -,+,/,$. Inoltre dobbiamo includere altri simboli speciali come, ad esempio, lo spazio che ci permette di separare le parole e che è rappresentato spesso con il simbolo $\lrcorner$ e il simbolo con cui segnaliamo la richiesta di andare a capo.

Se includiamo tutti i possibili simboli otteniamo *un insieme finito di simboli*. La cardinalità dell'insieme potrà variare a seconda del linguaggio che utilizziamo (ad esempio è noto che l'alfabeto inglese è composto da ventisei lettere mentre il nostro da sole ventuno) ma in ogni lingua e in ogni linguaggio di programmazione la cardinalità dell'insieme dei simboli utilizzati è finita.

A conferma di questo ricordiamo che per rappresentare le informazioni testuali utilizziamo codici. Un codice molto comune è il codice ASCII che utilizza otto bit per rappresentare un carattere e può quindi rappresentare 256 diversi caratteri. Il codice ASCII non è sufficientemente ricco per rappresentare i caratteri di tutte le lingue del pianeta ma lo è per scrivere i linguaggi di programmazione noti.

Osserviamo ora che *un programa scritto in Python (o in un qualunque linguaggio di programmazione) è un testo*. Non tutti i testi sono programmi ma se riusciamo a provare che il numero di testi che possiamo scrivere ha cardinalità $\mathbb{N}$ abbiamo dimostrato anche che il numero di programmi ha cardnalità $\mathbb{N}$ (ricordando che se $A \subseteq B$ allora $|A| \leq |B|$).

Vediamo ora come ordinare i diversi testi completando la prova che il numero di testi che possiamo scrivere ha la cardinalità di $\mathbb{N}$. L'idea è quella di ordinare i testi *ponendo i testi più brevi prima di quelli più lunghi.*

Questo significa che all'inizio abbiamo i testi formati da un solo carattere, poi quelli di due e così via. Come dobbiamo ordinare testi della medesima lunghezza? Per fare questo possiamo utilizzare la stessa modalità con cui le parole sono ordinate sul dizionario. Nel dizionario stabiliamo un ordine dei caratteri alfabetici e poi ordiniamo le parole in ordine lessicografico. Ad esempio ipotizziamo che i caratteri che usiamo siano così ordinati:

a b c d ... z A B C...Z 0 1 ...9 ... + / $

in questo caso avremo per testi di lunghezza 4 il seguente ordinamento

aaaa
aaab
aaac
...
aaaz
aaba
aabb
aabc
...
zzzz
...
$$$$

È chiaro che in questo modo possiamo ordinare tutti i testi che possiamo scrivere: avremo prima i testi (ordinati lessicograficamente) di lunghezza uno, poi quelli di lunghezza due e così via. Ovviamente non tutti i testi che elenchiamo in questo modo sono programmi Python ma osserviamo che in questo modo otteniamo comunque un ordinamento di tutti i programmi. Abbiamo quindi completato la prova del teorema 7. □

Osserviamo che la prova data non fa riferimento ad uno specifico linguaggio di programmazione ma può essere applicata ad un qualunque linguaggio di programmazione.

2.1.2 Problemi decidibili e indecidibili

Definizione 5. Un *problema di decisione* è una arbitraria domanda del tipo sì-o-no o vero-o-falso relativo ad un insieme infinito di possibili ingressi. Per un dato problema l'insieme dei possibili input viene partizionato di un due insiemi, quello che soddisfa la domanda la proprietà richeista e gli altri.

Equivalentemente possiamo definire un problema decisionale come l'insieme di ingressi per cui il problema restituisce sì (o `true`). I possibili ingressi possono essere numeri naturali, ma anche altri valori di altro tipo, come ad esempio stringhe di caratteri. I programmi che risolvono problemi decisionali prendono in input stringhe di caratteri e riconoscono se l'input verifica o meno una data proprietá: il programma fornisce in uscita il valore sí o `true` se la stringa verifica la proprietá e no o `false` altrimenti.

Esempi.
Un primo esempio di programma che riceve in input una stringa e riconosce se la stringa verifca una data proprietà è quello in cui vogliamo riconoscere le stringhe x, $x \in ASCII^*$ che hanno un numero pari del carattere "a" dalle altre. Un programma per questo problema riceve in ingresso una stringa $ASCII^*$ x e, analizzando x carattere per carattere, distingue le stringhe che verificano la condizione richiesta dalle altre. In questo caso il problema è risolubile: non è difficile scrivere un programma Python che riconosca queste stringhe.

Un secondo esempio è quello in cui il programma riceve in ingresso una coppia di stringhe (x, y) e si vuole riconoscere quando le due stringhe sono uguali (cioé quando $x = y$). Anche questo programma non è difficile da scrivere.

Il terzo esempio, più complesso, è quello di riconoscere le stringhe x, $x \in$ASCII* che corrispondono a programmi Python sintatticamente corretti (cioè vogliamo la risposta `true` se il programma verifica le regole del linguaggio anche se è errato dal punto di vista semantico; se il programma non rispetta le regole della sintassi di Python la risposta deve essere `false`). Questo esempio, anche se più complesso è analogo ai precedenti. Infatti se ricordiamo che un programma Python non è altro che una stringa di caratteri ASCII il problema chiede di stabilire se esiste un programma che prende in ingresso una stringa di caratteri e decide se questa rappresenta un programma Python sintatticamente corretto. □

Definizione 6. Un problema di decisione T

- è *decidibile* se esiste un programma che termina sempre e riconosce esattamente le stringhe ASCII che verificano T
- è *semidecidibile* se esiste un programma che termina sempre quando la risposta è sì (o `true`).

Un problema di decisione non decidibile è detto *indecidibile*. Osserviamo inoltre che, nel caso di un problema semidecidibile, non si richiede che il programma termini quando l'input non verifca la proprietà richiesta.

Non è difficile convincerci che le proprietà richieste nei tre esempi precedenti sono decidibili; nei primi due casi è facile scrivere programmi che verificano quanto richiesto; il terzo esempio è meno semplice ma sappiamo che è risolto dai compilatori Python che usiamo.

Mostriamo ora che il numero possibile di problemi di decisione ha cardinalità superiore a $|\mathbb{N}|$. Per fare questo limitiamo la nostra attenzione ad una classe particolare di problemi: i problemi che chiedono di riconoscere insiemi dei numeri naturali. I programmi di questo tipo hanno come input un numero naturale y e il programma deve determinare se y verifica una data proprietà.

La definizione seguente estende al caso di insiemi di numeri naturali i concetti di decidibile e indecidibile espressi dalla definizione precedente.

Definizione 7. Sia S un sottoinsieme dei numeri naturali. Diciamo che

- S è decidibile se *esiste un programma $P(S)$ che riconosce S* se $P(S)$ riceve in ingresso un intero y termina sempre e stampa `true` se $y \in S$ `false` se $y \notin S$
- S *indecidibile* se un tale programma non esiste.

Un esempio di problema di riconoscimento di insiemi è quello che chiede di riconoscere l'insieme S_1 dei numeri pari; in questo caso il nostro programma riceve in input un numero intero y e stampa `true` se il numero è pari e `false` altrimenti. È facile mostrare che l'insieme S_1 è decidibile. Un altro esempio che possiamo facilmente mostrare decidibile è quello di riconoscere l'insieme S_2 dei numeri che sono potenze di due.

2.1.3 I problemi di decisione sono un insieme non numerabile

Mostriamo ora che l'insieme dei problemi che chiedono di riconoscere insiemi dei numeri ha cardinalità superiore a $\mathbb{N}$ e che, quindi, esistono insiemi non decidibili.

Teorema 8. *Il numero dei problemi diversi fra loro ha cardinalità strettamente superiore a $|\mathbb{N}|$.*

Per dimostrare il teorema usiamo il metodo della diagonalizzazione. Assumiamo per contraddizione che per ogni insieme S dei numeri naturali esista un programma che decide S. Dato che sia i programmi che i possibili input sono numerabili possiamo costruire una matrice M con infinite righe e infinite colonne che rappresenta tutti i possibili risultati di tutti i programmi che riconoscono insiemi dei numeri naturali.

In particolare nella riga i della matrice poniamo la sequenza di risultati del programma i-esimo nella numerazione die programmi con input 0 e poi con input 1, 2, 3 e così via. Ad esempio, se il programma i-esimo è il programma che riconosce i numeri pari i valori della riga i della matrice M sono `true`, `false`, `true`, `false`, `true`, ...

Pertanto la riga i-esima della matrice M rappresenta i possibili risultati del programma i-esimo della nostra enumerazione dei programmi al variare dell'input e quindi definisce un insieme decidibile; denotiamo con S_i l'insieme ottenuto in corrispondenza alla riga i. Analogamente la colonna j-esima rappresenta la sequenza dei risultati calcolati dai diversi programmi con input j.

A questo punto è facile vedere che esiste un insieme T che non è decidibile. T è definito per diagonalizzazione dalla tabella M. In particolare se $M(i,i) = \texttt{true}$ allora $i \notin T$; se $M(i,i) = \texttt{false}$ allora $i \in T$. Chiaramente T è diverso da ciascun insieme S_i di M per l'intero i e, quindi, non appartiene alla sequenza degli insiemi decidibili.

Poiché il numero dei programmi ha cardinalità inferiore al numero dei problemi abbiamo così mostrato che esistono infiniti problemi indecidibili per cui non esiste un programma di soluzione.

2.2 Il problema della fermata

La sezione precedente ha mostrato l'esistenza di (infiniti) problemi indecidibili ma non ci ha dato un esempio concreto. Scopo principale di questa sezione è dimostrare l'indecidibilità di un particolare problema, il problema della fermata, che chiede di scrivere un programma Q per risolvere il seguente problema:

Problema della fermata: *Esiste un programma Q che riceve in ingresso la specifica di un programma P e un possibile input y per P e decide se P si ferma con input y.*

Per una migliore comprensione della difficoltá del problema della fermata, osserviamo che è facile sapere se un programma P con input y si ferma: basta eseguire il programma e aspettare il tempo sufficiente alla sua esecuzione. La dfficoltá del problema si verifica quando il programma non si ferma; in altre parole come possiamo decidere se un programma non si ferma? Infatti, se il nostro programma non termina in un minuto non possiamo affermare con sicurezza che il programma non termina: potrebbe terminare in dieci minuti. Ovviamente lo stesso ragionamento si applica se il programma non termina in dieci minuti o in un'ora o in un giorno. La difficoltá del problema é, quindi, quella di decidere quando il programma non termina con un dato input.

Prima di definire in modo formale il problema osserviamo che la formulazione del problema fa riferimento ad altri programmi (cioè a programmi il cui input è un altro programma); programmi di questo tipo sono fondamentali in informatica per la loro rilevanza pratica. Ad esempio, compilatori e interpreti sono programmi che permettono di tradurre programmi scritti in linguaggio ad alto livello in programmi scritti in linguaggio macchina e, quindi, direttamente eseguibili da un computer. Vedrete nel corso degli studi che esistono altri casi in cui programmi di questo tipo sono utili nella pratica. L'esempio precedente motiva l'importanza non solo teorica di sapere quali programmi che operano su altri programmi possano essere realizzati e quali no. In particolare, notiamo che se fosse possibile avere un programma per risolvere il problema della fermata il lavoro del programmatore sarebbe molto semplificato.

Siamo pronti per una descrizione formale del problema: consideriamo un qualunque linguaggio di programmazione, ad esempio Python (o C, C++, Java, Fortran, Java ecc.) e restringiamo la nostra attenzione a *problemi di decisione.*

Mostreremo nel seguito che il problema della fermata non è decidibile dimostrando il seguente teorema.

Teorema 9. *Sia L un linguaggio di programmazione. Non esiste un programma Q scritto in L che, con input una coppia (x, y), dove x è la stringa che descrive un programma P_x scritto in L e y è una stringa di simboli di input per P_x, termina sempre in tempo finito e decide se P_x termina o no con input y.*

Il problema è illustrato in figura 3: x è la stringa di caratteri che definisce un programma P_x che termina quando y è l'input del programma. La procedura Q deve ritornare il valore `false` se la stringa x non corrisponde ad un programma corretto o se il programma P_x non termina (cicla per sempre) con input y.

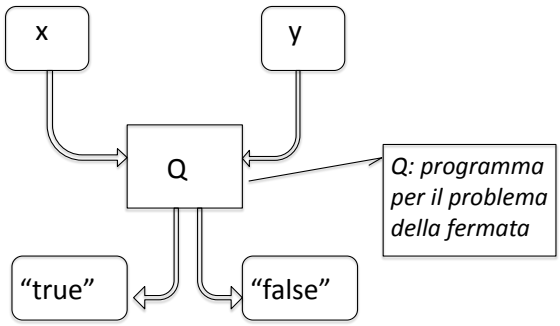

Figure 3: il problema della fermata.

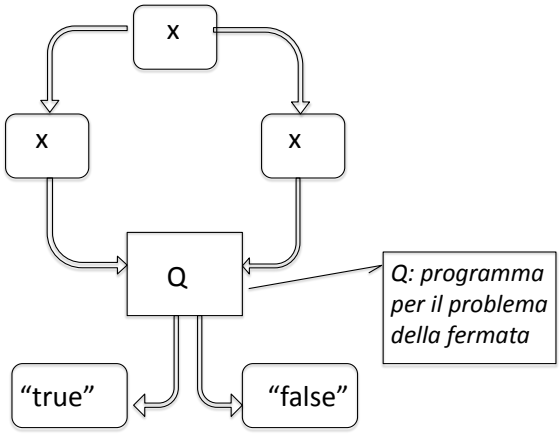

Figure 4: Caso particolare del problema della fermata.

Si noti che nella formulazione del teorema lo stesso linguaggio L è utilizzato sia per scrivere il programma per il problema della fermata sia per scrivere il programma P_x. Questo aspetto ha un ruolo fondamentale nella prova. Osserviamo però che è possibile estendere il teorema anche al caso in cui abbiamo due linguaggi diversi per scrivere Q e per scrivere i programmi analizzati da Q.

Da un punto di vista matematico il problema della fermata chiede di scrivere un programma per calcolare la funzione $H(x, y)$, così definita

$$H(x,y) = \begin{cases} \texttt{true} & \text{se il programma } P_x \text{ termina con input } y \\ \texttt{false} & \text{altrimenti} \end{cases}$$

Dimostrazione che il problema della fermata è indecidibile
Per dimostrare che il problema non è decidibile procediamo per contraddizione e assumiamo che Q sia una procedura che risolve il problema e che termina per ogni input (x, y). Analizziamo un caso

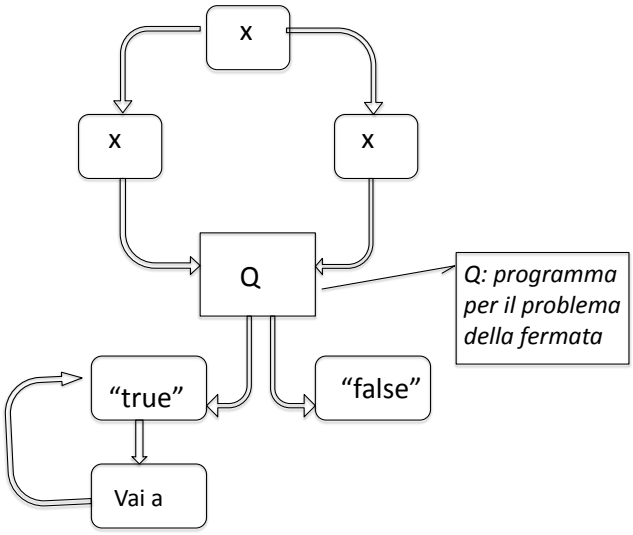

Figure 5: Programma R che utilizza Q come procedura.

particolare del problema in cui le due stringhe x e y sono uguali e ci chiediamo se il programma P_x descritto dalla stringa x termina con input la stringa x.

Si noti che nella definizione precedente la stringa x assume due ruoli diversi: viene usata per specificare il programma P_x e i dati in ingresso a P_x (vedi figura 4).

Se Q esiste allora esiste anche il programma il programma R la cui struttura è mostrata in figura 5. La modifica è evidenziata in basso a sinistra nella figura in cui si mostra che, nel caso che Q dia in output `true` il programma cicla. Questa modifica è semplice da fare: basta inserire dopo la chiamata della procedura Q un opportuno test che fa ciclare il programma.

Sia quindi r la stringa di caratteri che descrive il programma R e analizziamo ora il comportamento di Q con input r. Per definizione Q con input (r, r) risolve il problema della terminazione di R con input r. Mostriamo ora che nessuna delle due risposte `true`/`false` è possibile perché ciascuna porta ad una contraddizione.

Supponiamo che Q con input (r, r) ritorni `true`, cioè che Q decide che R termina con input r e osserviamo la figura 5; la figura mostra che se Q con input r ritorna `true` allora il programma R abbiamo un ciclo infinito e R non termina! Questo contraddice l'ipotesi che Q decida che R termina con input r.

Assumiamo invece che Q con input (r, r) ritorni il valore `false` e, quindi, decida che R non termina. In questo caso, esaminando la figura 5, vediamo che se Q ritorna il valore `false` allora R termina. Questo contraddice l'ipotesi che Q decida che R non termini.

A questo punto la prova è completa: abbiamo assunto per contraddizione che Q risolve il problema della fermata e abbiamo mostrato che esiste un programma R per cui Q non fornisce una risposta corretta. Possiamo quindi concludere che cade l'assunzione iniziale che esista un programma Q che sia in grado di risolvere il problema della fermata. Pertanto il problema della fermata è indecidibile.

La prove precedente è breve ma non è semplice da comprendere. La difficoltà principale nella sua comprensione è dovuta al fatto che la prova è autoreferenziale: il programma R definito nella prova utilizza Q in modo tale da creare una contraddizione sulla risposta di Q quando riceve in ingresso una versione modificata di se stessa.

La natura autoreferenziale della prova utilizza l'idea di Cantor della diagonalizzazione. Vediamo ora una seconda prova del teorema 9 che usa direttamente la diagonalizzazione.

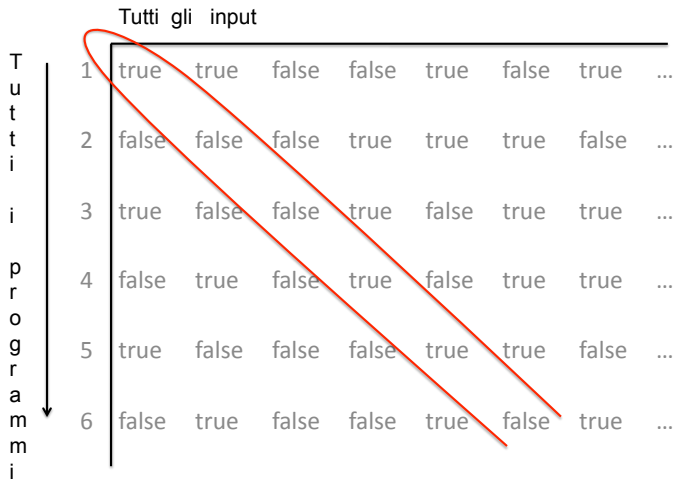

Figure 6: Tabella dei risultati del programma Q che risolve il problema della fermata.

Una seconda dimostrazione del teorema della fermata
Sia P_x il programma codificato dalla stringa x e assumiamo per contraddizione che esista un programma Q che decide per ogni coppia di input (x, y) se il programma P_x termina con input la stringa y.

Sappiamo che le stringhe di caratteri sono numerabili e consideriamo quindi un loro ordinamento. Costruiamo ora una tabella in cui poniamo tutti i possibili risultati di Q (vedi figura 6). La riga i della tabella codifica le possibili risposte di Q relativamente al programma codificato dalla i-esima stringa nell'ordinamento delle stringhe; in particolare il valore di riga i e colonna j riporta `true` o `false` a seconda che il programma i-esimo termini o non termini con input la j-esima stringa. L'ipotesi fatta che esista Q implica che una tale tabella può essere costruita usando Q.

Abbiamo pertanto una tabella come quella di figura 6. Consideriamo ora gli elementi della diagonale della figura, da sinistra a destra

$$\texttt{true}, \texttt{false}, \texttt{false}, \texttt{true}, \texttt{true}, \texttt{false} \ldots$$

e definiamo un programma S che sia diverso da ogni programma della nostra numerazione di tutti i programmi per almeno un input. In particolare S quando prende in input la j-esima stringa dell'ordinamento dei possibili input si comporta diversamente dal j-esimo programma con input la j-esima stringa. In altre parole S cicla se il j-esimo programma termina con input la j-esima stringa e termina se il j-esimo programma cicla. Con riferimento alla figura avremmo la seguente sequenza che descrive se S termini o meno con i vari input della lista

$$\texttt{false}, \texttt{true}, \texttt{true}, \texttt{false}, \texttt{false}, \texttt{true}, \ldots$$

L'applicazione del metodo della diagonalizzazione implica che S sia un programma diverso da tutti i possibili programmi e, quindi, che S non esiste. D'altro canto l'ipotesi fatta che Q sia un programma che risolve il problema della fermata ci permette di scrivere S utilizzando Q (il valore calcolato da S è la negazione del valore calcolato da Q). Abbiamo quindi ottenuto una contraddizione e, quindi, l'ipotesi iniziale che Q esista è falsa. Questo conclude la prova del teorema.

Notiamo che le due prove del teorema della fermata utilizzano in modi diversi la medesima idea della diagonalizzazione. Infatti osserviamo che i due programmi R e S usati nelle due prove sono in effetti lo stesso programma.

Si noti che **il teorema 9 vale se Q e i programmi analizzati da Q sono scritti in un qualunque linguaggio di programmazione.**
Osserviamo inoltre che nelle dimostrazioni date dell'indecidibilità del problema della fermata si usa in modo sostanziale il fatto che il programma Q e i programmi che sono analizzati da Q sono scritti nello stesso linguaggio di programmazione; questo permette di utilizzare un meccanismo autoreferenziale. Ci possiamo pertanto chiedere se possa esistere un programma Q scritto in un linguaggio di programmazione (ad esempio C) che sia in grado di stabilire la terminazione di programmi scritti in un altro linguaggio (ad esempio Python). Anche in questo caso il problema è indecidibile. Per dimostrare questo è sufficiente osservare che di norma dati due linguaggi di programmazione L e L' possiamo scrivere un programma traduttore che traduca programmi scritti in L in programmi scritti in L' (o viceversa). Diciamo in questo caso che il potere espressivo di L e L' è lo stesso[2]. Pertanto l'esistenza di un programma C che risolva il problema della fermata di programmi Python implicherebbe immediatamente l'esistenza di un programma Python che risolve il problema. Come abbiamo visto questo programma non può esistere.

Esistono casi risolubili del problema della fermata?
Il primo istinto di molte persone quando vedono la dimostrazione del Teorema della fermata è non crederci fino in fondo. Cioè molte persone credono allaffermazione matematica, ma intuitivamente non pensano che il problema della fermata sia davvero cos difficile.

In realtà i programmatori sembrano risolvere il problema della fermata informalmente (o formalmente) ogni qualvolta sostengono (o dimostrano) che i loro programmi si fermano. vero che i loro programmi sono scritti in C o Python, al contrario delle macchine di Turing, ma quello non fa differenza: possiamo facilmente tradurre avanti e indietro tra questo modello e qualsiasi altro linguaggio di programmazione.

È chiaro che affermare che il teorema precedente non esclude che il problema sia risolubile in quasi tutti i casi di programmi sviluppati. In altre parole il fatto che analizzare proprietà di programmi arbitrari sia indecidibile in generale non implica che sia impossibile scrivere programmi che analizzano i programmi e diano risposte corrette in casi pratici. Ad esempio, è possibile scrivere un analizzatore che prende in ingresso un programma $\mathcal{P}$ e un possibile input y e decide se $\mathcal{P}$ termina dopo un prefissato tempo di esecuzione.

Quello che risulta impossibile è trovare un metodo che permette di analizzare (o verificare o ottimizzare) un arbitrario programma scritto in un linguaggio di programmazione come ad esempio Python, C, C++, Java. Ovviamente avere a disposizione un programma che risolve il problema in modo automatico una volta per tutte è anche una cosa molto comoda.

Ci chiediamo ora se esiste un modo per risolvere il problema. Alcune persone sostengono che possono personalmente, se pensano abbastanza, determinare se qualsiasi programma concreto che viene dato si fermer o meno. Altri hanno persino sostenuto che gli umani in generale ne abbiano la capacitá, e quindi ritengono che gli umani abbiano un'intelligenza intrinsecamente superiore computer o qualsiasi altra cosa modellata da macchine di Turing.

La migliore risposta che abbiamo finora che non ci sia modo di risolvere il problema della fermata, sia che si tratti di Mac, PC, computer quantistici, umani o qualsiasi altra combinazione di dispositivi elettronici, meccanici e biologici che siano programmabili con un linguaggio con la stessa potenza espressiva di Python o C. In effetti questa affermazione il contenuto della tesi di Church-Turing. Questo naturalmente non significa che per ogni programma P difficile decidere se P entra in un ciclo infinito o no. Alcuni programmi non hanno anche dei loop (e quindi banalmente non ciclano), e ce ne sono molti altri esempi meno banali di programmi che possiamo certificare di non fare mai inserisci un ciclo infinito (o programmi che sappiamo per certo che lo faranno inserire un tale ciclo).

Tuttavia, non esiste una procedura generale che lo farebbe determinare per un programma arbitrario P se si ferma o no. Inoltre, ci sono alcuni programmi molto semplici per i quali nessuno sa se si fermano o no.

[2]Osserviamo che ogni linguaggio ha un insieme di caratteristiche che lo rendono adatto a risolvere particolari classi di problemi. Stabilire che il potere espressivo di due linguaggi è lo stesso implica che risolvono gli stessi problemi anche se può essere opportuno scegliere un linguaggio per alcuni problemi e l'altro per altri problemi.

Ad esempio, la congettura di Goldbach uno dei pi vecchi problemi irrisolti nella teoria dei numeri. Essa afferma che ogni numero pari maggiore di 2 puó essere scritto come somma di due numeri primi (che possono essere anche uguali).

Il seguente programma Python si fermer se e solo se la congettura di Goldbach falsa.

```
def isprime(p):
return all(p % i for i in range(2,p-1))
def Goldbach(n):
       return any( (isprime(p) and isprime(n-p))
                for p in range(2,n-1))

 n = 4
 while True:
        if not Goldbach(n): break
                 n+= 2
```

Ricordiamo che la congettura di Goldbach è un problema matematico aperto dal 1742. Non è chiaro se gli essere umani avranno labiltià di decidere se questo programma si arresta o meno in poco tempo.

Altri problemi indecidibili. La tecnica di prova precedente può essere estesa per mostrare che altre proprietà dei programmi sono indecidibili. In particolare, non è possibile scrivere programmi che verificano altre proprietà a tempo di esecuzione di un programma arbitrario (come, ad esempio, chiedersi se una variabile è aggiornata o se se si verifica un accesso in memoria con un errore di tipo) o di analizzare o di ottimizzare un programma arbitrario o, infine, verificarne la correttezza.

3 Le macchine di Turing

Lo scopo di questo capitolo è investigare dispositivi di calcolo molto semplici e valutare le loro capacità di calcolo. Vedremo che le macchine di calcolo astratte e i modelli di calcolo proposti negli anni trenta del secolo scorso, cioè prima dell'introduzione dei computer elettronici hanno la stessa capacità computazionali dei calcolatori moderni. Infatti sono in grado di risolvere esattamente gli stessi problemi.

La ricerca di modelli di calcolo astratti e molto semplici ma equivalenti ai computer in uso può apparire un esercizio intellettuale di scarsa (o nessuna) rilevanza pratica. Osserviamo però che alla base di ogni scienza è studiare e determinare i concetti e le leggi fondamentali da cui conseguono le altre leggi della scienza. Per questa ragione, riuscire a individuare modelli molto semplici ma con la stessa capacità di calcolo fa parte della scienza dell'informazione perchè ci permette di capire i meccanismi fondamentali del calcolo. Un altro motivo è che lo studio di modelli semplici facilita la formulazione e la risoluzione di problemi che valgono anche se utilizziamo un modello di calcolo reale.

Nel 1936 Alan Turing ha descritto un modello teorico di una macchina di calcolo. Sebbene molto semplice, la Macchina di Turing possiede notevoli proprietà. In particolare, una macchina di Turing non è meno potente di qualsiasi altro dispositivo di elaborazione oggi disponibile ed è in grado di risolvere un qualunque problema risolubile con uno dei calcolatori che abbiamo a disposizione. La semplicità della operazioni eseguibili con una macchina di Turing influenza il tempo di calcolo che risulta significativamente maggiore.

I dati della macchina di Turing sono organizzati in maniera elementare, non solo con riferimento alle sofisticate tecniche utilizzate oggi dai moderni computer, ma anche con riferimento al modello della memoria descritto nella macchina di von Neumann. Infatti, la memoria in una macchina di Turing è organizzata linearmente come le cellule di un nastro infinito; ogni cella del nastro può contenere esattamente un simbolo di un alfabeto con un numero finito di simboli. La macchina è dotata di una testina di lettura e scrittura del nastro che, ad ogni istante, è posizionata su una cella del nastro. Ad ogni passo di calcolo la macchina può leggere e scrivere solo la cella dove è

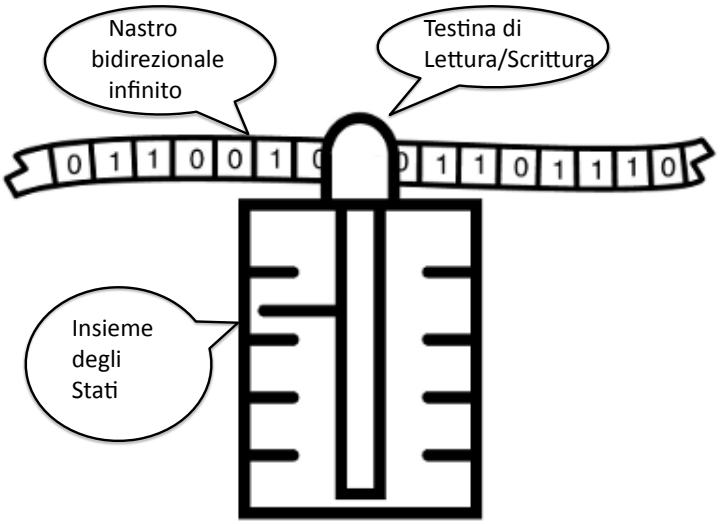

Figure 7: La figura illustra una versione schematica di una Macchina di Turing con un alfabeto binario e dieci possibili stati (schematizzati dalle dieci possibili posizioni del cambio).

posizionata la testina e può spostarsi di una posizione a destra o a sinistra. All'inizio, i dati di ingresso sono posti sul nastro che per la parte rimanente è vuoto e la testina di lettura/scrittura è posizionata sul primo carattere dell'input.

La parte di controllo è anch'essa molto semplice. Intuitivamente possiamo immaginarla come una scatola del cambio che assume un numero prefissato e finito di posizioni. Ogni possibile posizione è un possibile *stato* della macchina. Il numero di stati varia da macchina a macchina. La figura 7 esemplifica una macchina di Turing con un alfabeto binario e con dieci possibili posizioni del "cambio" (stati).

Il calcolo di una macchina di Turing inizia con la macchina in uno stato iniziale e la testina posizionata sul primo carattere dell'input. Ad ogni passo di calcolo la parte controllo della macchina legge un carattere sul nastro e, a seconda dello stato in cui si trova, scrive un nuovo carattere sul nastro, passa in un nuovo stato e muove la testina di una posizione a destra o a sinistra. Il calcolo prosegue fino a quando la macchina entra in uno di di due stati speciali ARRESTA o RIFIUTA; in questi casi la macchina si arresta accettando o rifiutando.

In particolare, la parte di controllo è così specificata. Ad ogni istante la macchina di Turing si trova in uno *stato* e la macchina legge la cella del nastro su cui è posizionata la testina. Poiché sia l'alfabeto sul nastro che il numero di stati della macchina sono finiti abbiamo un numero finito di possibili coppie <*stato, carattere letto*>; il controllo, sulla base della coppia <stato, carattere letto> che rappresenta lo stato corrente e il simbolo letto dalla testina, esegue i seguenti passi:

1. stampa un nuovo simbolo sul nastro (o lascia sul nastro lo stesso simbolo letto)
2. sposta la testina di lettura/scrittura di una posizione a destra o a sinistra sul nastro
3. passa allo stato successivo (che può anche essere lo stesso stato).

I tre punti precedenti definiscono una *mossa* in cui si utilizza la *funzione di transizione* che è definita se la macchina si trova in uno stato diverso da uno dei due designati stati speciali, ACCETTA e RIFIUTA. Sia Q l'insieme finito degli stati e Σ l'insieme finito dei caratteri dell'alfabeto. La mossa della macchina è determinata da una funzione di transizione che associa alla coppia data dallo stato attuale e dal simbolo corrente sottoposto a scansione una tripla formata da

(stato successivo, simbolo da scrivere, spostamento della testina)

Dato che non vi è prossima mossa nel caso la macchina sia nello stato ACCETTA e RIFIUTA la funzione di transizione è definita solo se lo stato è diverso da ACCETTA o RIFIUTA. Inoltre, se il problema richiede di calcolare un risultato allora, quando la macchina accetta, il risultato del calcolo è scritto sul nastro.

Formalmente, abbiamo la seguente definizione di una macchina di Turing.

Definizione 8. Una macchina di Turing è definita da

1. un alfabeto finito di caratteri sul nastro Σ
2. un insieme finito S di stati
3. uno stato iniziale e due stati speciali: ACCETTA, RIFIUTA entrambi in S
4. una funzione di transizione

$$\delta : (Q - \{\text{ ACCETTA, RIFIUTA}\}, \Sigma) \to (Q, \Sigma, \{D, S\})$$

Confrontiamo brevemente le definizioni di Automa a stati finiti e di Macchina di Turing. La principale differenza consiste nel fatto che in un automa a stati finiti il nastro è di sola lettura e viene letto un carattere alla volta (ad ogni transizione lo spostamento della testina di ettrua è sempre D. Invece la Macchina di Turing è dotata di un nastro su cui può anche scrivere sul nastro; inoltre nella transizione di stato può muoversi liberamente a destra o a sinistra. Possiamo dire che nel caso delle Macchine di Turing il nastro funziona come una memoria di lavoro.

Per indicare che il nastro è vuoto usiamo il simbolo "␣"; assumiamo quindi che $␣ \in \Sigma$. All'inizio del calcolo la prima cella con il simbolo ␣ indica la fine della parola di ingresso.

Esempio 1. Un contatore di parità.
Data è una stringa binaria $w \in \{0,1\}^*$, mostriamo la macchina di Turing M che riconosce le stringhe con un numero pari di 1, In particolare, il nastro contiene una stringa di 0 e 1 e assumiamo che in ogni altra cella del nastro contenga il carattere "␣"; la testina del nastro è posta all'inizio sul primo carattere di input e la macchina esamina il contenuto del nastro; quando la stringa è stata eseminata (e quindi la macchina ha letto il primo carattere ␣) la macchina si deve fermare in stato ACCETTA se il numero di 1 sul nastro è zero o pari e fermarsi in stato RIFIUTA se il numero di 1 è dispari.

Abbiamo il seguente insieme di stati: $S = \{q_0, q_1, ACCETTA, RIFIUTA\}$.

Intuitivamente q_0 segnala che sono stati esaminati un numero pari di 0 mentre q_1 indica che sono stati esaminati un numero dispari di 1. q_0 è lo stato iniziale.

La funzione di transizione δ è così definita:

$$\delta(q_0, 0) = (q_0, 0, d), \quad \delta(q_0, 1) = (q_1, 1, d), \quad \delta(q_0, ␣) = (ACCETTA, ␣, STOP)$$
$$\delta(q_1, 0) = (q_1, 0, d), \quad \delta(q_1, 1) = (q_0, 1, d), \quad \delta(q_1, ␣) = (RIFIUTA, ␣, STOP)$$

in partciolare $\delta(q_0, 0) = (q_0, 0, d)$ specifica che quando la macchina si trova nello stato q_0 e la testina legge il carattere 0 allora la macchina rimane nello stato q_0 riscrive 0 sul nastro e quindi la testina si sposta di un carattere a destra. Analogamente $\delta(q_0, 1) = (q_1, 1, d)$ specifica che nello stato q_0 leggendo sul nastro 1 la macchina passa allo stato q_1.

Si noti che la macchina precedente riscrive sul nastro il carattere letto e si arresta quando incontra il primo carattere ␣. □

Osserviamo che la semplicitádell'esempio precedente non evidenzia le carattersitiche della Macchina di Turing: abbiamo solo due stati, il nastro non viene riscritto e la testina si muove sempre in una sola direzione.

La definizione data e l'esempio precedente evidenziano come le macchine di Turing non sono programmabili come lo sono i nostri elaboratori. In particolare una macchina di Turing è in

grado di eseguire un solo programma, specificato dalla funzione di transizione. Intuitivamente l'hardware della macchina di Turing è costituito dal nastro, dalla testina di lettura e scrittura e dal meccanismo che si sposta fra i diversi stati e che rappresenta l'analogo dell'unità di controllo nell'architettura di von Neumann. Questa ultima parte è la stessa per tutte le macchine di Turing e, per questa ragione spesso si fa riferimento alla programmazione di una macchina di Turing. Pertanto, possiamo affermare che la parte software della Macchina di Turing è rappresentata dalla funzione di transizione.

Una macchina di Turing continua l'esecuzione fino al raggiungimento di uno dei due stati ACCETTA/RIFIUTA; se non si raggiunge mai uno di questi stati, il calcolo continua per sempre. Data una macchina di Turing possiamo quindi dividere i possibili input in tre categorie:

1. l'insieme degli input per cui la macchina termina nello stato ACCETTA
2. l'insieme degli input per cui la macchina termina nello stato RIGETTA
3. l'insieme degli input per cui la macchina non termina

3.0.1 Macchine di Turing e riconoscimento di linguaggi

Le macchine di Turing possono essere usate per riconoscere proprietà o per calcolare funzioni. Nel primo caso la macchina distingue gli input che verificano una data proprietà dagli altri.

Definizione 9. Dato un alfabeto Σ un *linguaggio* L è un sottoinsieme delle stringhe in Σ^*.

Definizione 10. Data una una macchina di Turing M, il linguaggio *accettato* da M è

$$L(M) = \{w \in \Sigma^* | M \text{ con input } w \text{ termina nello stato ACCETTA}\}$$

Definizione 11. Un linguaggio L è

- *Turing-accettabile* se esiste una macchina di Turing che lo accetta.
- *Turing-decidibile* se L è accettato da una Macchina di Turing che si ferma per ogni input.

Di solito scriveremo "accettabile" invece di "Turing-accettabile", e "decidibile" invece di "Turing-decidibile". Si noti la distinzione tra queste due definizioni.

In particolare, se M decide L, allora per tutti gli input w, se $w \in L$ allora M entra nello stato ACCETTA, se $w \notin L$ allora M si ferma nello stato RIGETTA. Invece, se M accetta L, per tutti gli input w, se $w \in L$ allora M entra nello stato ACCETTA, ma se $w \notin L$ allora M non deve necessariamente fermarsi (o si ferma nello stato RIGETTA o cicla per sempre).

In altre parole, ogni macchina di Turing M accetta un linguaggio, ma una macchina di Turing potrebbe non essere un decisore per qualsiasi linguaggio semplicemente perché non si ferma su tutti gli input.

Osserviamo inoltre che dato un alfabeto Σ e una proprietà P l'insieme delle stringhe in Σ^* che verificano P definisce un linguaggio $L(P)$. Quindi possiamo affermare che il problema di decidere una proprietà P o riconoscere il linguaggio $L(P)$ sono equivalenti.

Le macchina di Turing come le abbiamo formulate permettono lo studio del riconoscimento di linguaggi. Tuttavia, è importante rendersi conto che le macchine di Turing possono anche calcolare funzioni fornendo in uscita i valori calcolati. A tale scopo è sufficiente richiedere che quando la macchina raggiunge lo stato ACCETTA il nastro di input contiene il risultato.

Definizione 12. Una macchina di Turing M calcola la funzione parziale f, $f(x_1, x_2, \ldots, x_n)$ quando il nastro iniziale contiene i valori $a_1, a_2, \ldots a_n$ allora se se $f(a_1, a_2, \ldots, a_n)$ è definito allora M raggiunge lo stato ACCETTA con il nastro che contiene i valori $f(a_1, a_2, \ldots, a_n)$.

Si noti che se la funzione non è definita allora la macchina potrebbe non fermarsi.

3.1 Varianti delle macchine di Turing

Discuteremo nel prossimo capitolo come le macchine di Turing abbiano la stessa capacità computazionale di ogni altro dispositivo di elaborazione. Cioè le macchine di Turing accettano e decidono gli stessi linguaggi e calcolano esattamente le stesse funzioni di qualsiasi altro dispositivo di elaborazione.

Ma quali sono le differenze tra una macchina di Turing e i computer che usiamo ogni giorno? Una è che i nostri computer operano su parole a 32 o 64 bit, mentre la macchina di Turing è in grado di elaborare solo un carattere per ogni passo di calcolo. Questa differenza influenza la velocit à dei calcoli ma non è essenziale nel determinare la capacità computazionale; non entriamo nei dettagli e ci limitiamo ad osservare che una macchina di Turing è in grado di eseguire una sequenza di passi ciascuno dei quali modifica un bit e che, complessivamente, modificano i 32 (o 64) bit di una parola.

Un'altra differenza significativa è che i computer che usiamo sono in grado di eseguire un "accesso casuale" alla memoria, specificando l'indirizzo della locazione di memoria su cui si vuole operare. Ad esempio in un'istruzione si può leggere i contenuti della posizione di memoria 100 e poi, all'istruzione successiva, leggere la locazione di memoria 1000. La macchina di Turing può facilmente simulare questo, ma deve fare 900 spostamenti per muovere la testina dalla cella 100 alla cella 1000. Pertanto, come nel caso precedente, la differenza è nella efficienza del calcolo piuttosto che nella capacità di eseguire calcoli che non potenza di calcolo essenziale.

Il resto del capitolo è dedicato ad espandere questa breve discussione, e a dare evidenza che la macchina di Turing è potente come qualsiasi altro computer. In questa sezione introdurremo una importante variazione del modello di Turing, la macchina di Turing a k nastri, che è la naturale estensione. e dimostreremo che essa sia equivalente alle macchine di Turing. Vedremo in particolare come esiste una macchina di Turing universale. La sezione successiva illustra la tesi di Church pi ù in dettaglio, e poi esamineremo le macchine ad accesso casuale più tecnicamente rispetto al precedente paragrafo.

Definizione 13. Una *macchina di Turing multinastro* ha k nastri, ciascuno con la propria testa lettura/scrittura. In particolare, quando M è nello stato q legge i k simboli delle celle dove sono posizionati i diversi nastri e in funzione dello stato e dei k simboli letti, scrive su ciascun nastro, si sposta a sinistra o a destra su ogni nastro, ed entra in un nuovo stato.

Formalmente, quando la macchina si trova nello stato q e simboli acquisiti $a_1, ..., a_k$ sui nastri $1, 2, ...k$, la funzione di transizione calcola

$$\delta(q, a_1, a_2, \ldots, a_k) = (p, b_1, \ldots, b_k, M_1, \ldots, M_k)$$

dove $b_1, b_2, \ldots, b_k)$ sono i nuovi simboli da scrivere sui nastri, $M_1, \ldots, M_k$ sono le direzioni (destra o sinistra), in cui le testine si muovono, e p è lo stato successivo.

Osserviamo che nella definizione precedente gli spostamenti delle testine sui diversi nastri possono essere qualunque (ad esempio, la testina dei nastri 1,2, 4 possono spostarsi a destra mentre quelle degli altri nastri vanno a sinistra).

Il teorema seguente afferma che ogni macchina multinastro Turing è equivalente a una macchina di Turing con un solo nastro. Il nostro interesse per questo risultato è duplice. In primo luogo, mostra che aggiungendo nastri di lettura e scrittura non aumenta la potenza di calcolo e aiuta a convincerci come la macchina di Turing sia un modello molto potente di calcolo. Secondo, la macchina di Turing con più nastri è un modello utile per mostrare il principale risultato relativo all'esisternza di una macchina universale. Osserviamo infine che la macchina multitape Turing è più efficiente e più facile da programmare di macchine di Turing a nastro singolo.

Non diamo la dimostrazione del seguente teorema che stabilisce che tutto quello che possiamo calcolare con una macchina di Turing con più nastri lo possiamo callcolare con una macchina di Turing con un solo nastro.

Teorema 10. *Per ogni macchina di Turing con più nastri esiste una macchina di Turing con un solo nastro equivalente.*

3.2 La macchina di Turing universale

Abbiamo visto che ogni macchina di Turing calcola una certa funzione e, in questo senso, si comporta come un prefissato programma. In questa sezione vediamo come costruire una macchina che sia in grado di realizzare i calcoli realizzati da una qualunque altra macchina. La macchina di Turing universale (U), proposta da Turing nel 1936, è una macchina di Turing in grado di simulare una qualunque altra macchina di Turing arbitraria su input arbitrari. La macchina universale prende in input sia la descrizione della macchina M da simulare e un input I e calcola il risultato di M con input I. [3]

Per valutare la profondità della visione di Turing citiamo alcune parole del suo articolo e ricordiamo che queste parole sono state scritte nel 1936, ben prima che i calcolatori cos ì come li conosciamo fossero ideati.

...È possibile inventare una sola macchina che può essere utilizzata per calcolare qualsiasi sequenza computabile. Se questa macchina U è dotata di un nastro su cui all'inizio è fornita la "descrizione standard" della funzione di transizione di una macchina M, allora U calcolerà la stessa sequenza di M.

Vedremo fra poco cosa sia "la descrizione standard" cui fa cenno Turing; per il momento è sufficiente assumere che essa sia la descrizione di una macchina di Turing utilizzando una stringa di caratteri e posta su uno dei nastri della macchina universale.

Per esemplificare come è realizzata U supponiamo che sia data una macchina di Turing M con un solo nastro e input I e vogliamo vedere come la macchina universale U realizzi i passi eseguiti da M con input I. Assumiamo che l'alfabeto Σ di M sia l'insieme binario $\{0, 1\}$. La macchina universale U ha tre nastri:

1. il primo nastro è di sola lettura e contiene la descrizione di M secondo una rappresentazione che vedremo fra breve

2. il secondo nastro è usato da U per la simulazione della macchina M (in altre parole rappresenta la memoria di lavoro) e all'inizio contiene un solo carattere 0

3. il terzo nastro rappresenta il nastro usato dalla macchina M e all'inizio contiene l'input I.

Dato che assumiamo che l'alfabeto di M sia l'alfabeto binario per descrivere M è sufficiente specificare il numero degli stati e la funzione di transizione. Nel seguito assumiamo che U utilizzi come alfabeto l'insieme $\Sigma' = \{0, 1, \& D, S\}$; il simbolo "& viene usato come separatore[4].

Assumiamo una codifica binaria degli stati di M, in particolare lo stato iniziale $q_0$di M, è codificato con 0 e gli stati ACCETTA e RIFIUTA sono i due ultimi stati e sono codificati anch'esso in binario.

Esempio 2. Per esemplificare la codifica della macchina M, consideriamo la macchina di Turing che abbiamo dato nell'esempio 1, Con riferimento all'esempio abbiamo che M ha quattro stati $(0, 1, 10, 11)$ e, quindi, codifichiamo lo stato iniziale con 0 e gli stati ACCETTA e RIGETTA con 10 e 11.

In questo caso sul primo nastro di U scriveremo:

$$100\&\&0\&0\&00D\&\&0\&1\&11D\&\& \ldots$$

[3]Alcuni considerano la macchina di Turing universale come l'origine del programma memorizzato su computer in grado di essere eseguiti da una semplice macchina. Questa affermazione non trova tutti concordi. Infatti, alcuni ritengono che, anche se la teoria delle funzioni computabili di Turing sia anteriore allo sviluppo dei moderni calcolatori, la teoria di Turing non ha molto influenzato il progetto e la costruzione di computer digitali. Infatti gli aspetti della teoria e della realizzazione pratica sono stati sviluppati quasi completamente indipendente l'uno dall'altro. Il motivo principale di questa separazione è dovuto al fatto che i logici sono interessati a questioni radicalmente diverse da quelle di interesse per i progettisti e gli utenti dei calcolatori. Tuttavia è sorprendente come gli stessi concetti siano stati espressi in modo diverso nei due sviluppi.

[4]L'introduzione di $\&, D, S$ è stata fatta per maggiore chiarezza; non è indispensabile e potremmo codificare la macchina M sul primo nastro utilizzando solo un alfabeto binario.

nella stringa data abbiamo prima 100 che codifica il numero degli stati di M (quattro) e quindi la funzione di transizione di M.

Ciascuna posizione della funzione di transizione è codificata fra due coppie di simboli &&. Ad esempio, all'inizio, dopo la stringa che specifica il numero degli stati abbiamo &&0&0&0, $0D$&&; questa parte della sequenza codifica il valore della funzione per $(q_0, 0)$, cioè $\delta(q_0, 0) = (0, 0, D)$; successivamente codifichiamo il valore di $\delta(q_0, 1)$ e così via.

□

Descriviamo ora la funzione di transizione di U. Non descriveremo U in dettaglio ma ci limiteremo a descrivere i passi essenziali con cui U esegue un passo di calcolo di M. Il secondo nastro è un nastro di lavoro che contiene la codifica binaria dello stato in cui si trova ad ogni istante la macchina M. Il terzo nastro è il nastro utilizzato da M e, quindi, all'inizio contiene l'input I.

La simulazione di un passo di calcolo di M richiede l'esecuzione di molti passi da parte di U che raggruppiamo nel seguente modo:

1. all'inizio U si posiziona sul primo carattere non vuoto del secondo nastro e legge su questo nastro la stringa di caratteri relativa allo stato q in cui si trova M e sul terzo nastro il carattere b letto dalla testina di M;

2. con queste due informazioni U ricerca nel primo nastro l'informazione relativa alla transizione dalla coppia (q, b) nella tabella di transizione di M; supponiamo che $\delta(q, b) = (q', b', M)$;

3. U completa l'esecuzione di un passo di calcolo di M scrivendo q' sul secondo nastro, scrivendo b' sul terzo nastro e spostando la testina del terzo nastro a destra o a sinistra a seconda del valore indicato da M.

4. infone U verifica se la simulazione di M è terminata; per fare questo verifica se lo stato q' attualemnte memorizzato sul secondo nastro è uno stato finale (ACCETTA o RIGETTA); per fare questo è sufficiente verificare q' con il numero totale degli stati. Se q' non è uno stato finale allora U ritorna ad eseguire il passo 1 precedetne altrimenti termina.

La descrizione dei tre passi precedenti mostra come la simulazione di un passo di M richieda molti passi di calcolo della macchina U.

4 La tesi di Church-Turing

Il padre della scienza informatica è senza dubbio Alan Turing (1912-1954), e il suo profeta è Charles Babbage (1791-1871). Babbage concepì la maggior parte degli elementi essenziali di un computer moderno e li realizzo nella sua *macchina analitica*, anche se ai suoi tempi non era disponibile la tecnologia per realizzare compiutamente le sue idee. Un secolo dopo prima macchina analitica di Babbage è stato migliorato quando Turing descrisse la macchina di Turing universale nel 1936. Il genio di Turing è stato quello di chiarire esattamente che cosa un elaboratore possa essere in grado di fare, e per sottolineare il ruolo della programmazione, vale a dire il software, più ancora di quanto Babbage aveva ipotizzato.

4.1 Il programma di Hilbert

Turing fu stimolato nelle sue ricerche dalle idee di David Hilbert e Kurt Gödel. Hilbert aveva sottolineato alla fine del XIX secolo l'importanza di porre domande sui fondamenti della matematica. Per spiegare il programma di Hilbert la matematica fino ad allora era interessata a rispondere sulla verità o falsità di un teorema chiedendosi "è questa proposizione matematica vera?"

Hilbert non era interessato alla verità di una singola affermazione e si chiedeva se "*è possibile che ogni proposizione matematica possa essere provata vera o confutata?*". Hilbert, e la maggior

parte dei matematici, ritenevano che si potesse dimostrare la verità o la falsità di una qualunque affermazione matematica. In altre parole Hilbert riteneva che la matematica fosse completa, in modo che tutte le affermazioni potessero essere provate o confutate in qualche modo, anche se per fare questo ci fosse bisogno di nuove idee.

La seconda domanda che si poneva Hilbert era se fosse possibile identificare un metodo automatico per provare la verità o la falsità di una proposizione matematica. I progressi in matematica sono basati sull'uso della immaginazione creativa, ma successivamente le dimostrazioni matematiche sembravano poter essere automatizzate, in modo che ogni passo seguisse in modo meccanico dal precedente. Hilbert si chiese se il processo di prova matematica potesse essere catturato da un processo "automatico". In altre parole, Hilbert si chiedeva se ci fosse un metodo matematico universale, per stabilire la verità o la falsità di ogni asserzione matematica?

Gödel ha distrutto la prima speranza di Hilbert mostrando l'esistenza di proposizioni matematiche che erano indecidibili, il che significa che potrebbero essere né dimostrato né confutate.

Chiaramente il risultato di Gödel ha reso necessario adattare e riformulare anche il secondo obiettivo che è stato riformulato in quello di stabilire la decidibilità, piuttosto che la verità, di una proposizione. Osserviamo che questo è l'obiettivo principale che Turing ha cercato di affrontare ipotizzando una macchina di calcolo elementare che potesse realizzare tutti calcoli e le dimostrazioni concepibili.

4.2 Tanti modelli di calcolo equivalenti

Turing non era il solo ricercatore a cercare un modello di calcolo in grado di eseguire in modo automatico tutto quanto fosse decidibile. Infatti, negli stessi anni in cui fu proposta la macchina di Turing, Emil Post ideò la macchina di Post, Alonzo Church ha sviluppato il λ-calcolo anche loro con l'obiettivo di realizzare una "macchina per un algoritmo universale". I modelli proposti da Turing, Post e Church sono di natura molto diversa fra loro ma hanno la stessa capacità di calcolo, cioè sono in grado di calcolare le stesse funzioni.

Quando Church propose il suo modello congetturò che ogni funzione che potesse essere calcolata da un algoritmo può essere definita usando il λ-calcolo. La congettura di Church, ha identificato computabilità effettiva, un concetto fino ad allora un concetto impreciso, con una specifica formulazione matematica. Poi, in modo indipendente, Turing ha formulato la tesi che ogni algoritmo può essere programmato su una macchina di Turing.

Poichè i modelli di calcolo proposti da Church e Turing sono equivalenti, allora le due tesi esprimono la stessa affermazione che oggi è nota come tesi di Church-Turing. Essa asserisce che

Tesi di Church-Turing. *Tutto quello che si può calcolare lo si può calcolare con le macchine di Turing.*

La tesi di Church-Turing non può essere "provata", perché non possiamo escludere che nel futuro sia possibile trovare modelli di calcolo più potenti delle macchine di Turing, cioè un modello di calcolo in grado di risolvere problemi indecidibili con le macchine di Turing. In realtà l'evidenza della tesi sono molte. L'evidenza principale è che tutti i modelli di calcolo proposti si sono dimostrati tutti equivalenti. Fra essi ricordiamo oltre alle macchine di Post, gli algoritmi di Markov, le RAM. Tutti questi modelli di calcolo utilizzano formalismi e passi di calcolo sostanzialmente diversi fra loro. Ciononostante in modo sorprendente tutti i modelli si sono rivelati equivalenti fra loro.

Una conseguenza della tesi di Church è risolubile solo se è risolubile con una macchina di Turing. In particolare un problema di decisione è decidibile solo se esiste una macchina di Turing che riconosce il linguaggio associato e una funzione è calcolabile solo se esiste una macchina di Turing che la calcola.

4.3 Le macchine RAM

Una macchina ad accesso casuale (Random Access Machine o semplicemente RAM) è un modello concettuale di un computer digitale. Come un elaboratore moderno la RAM permette di scrivere

programmi che fanno riferimento a registri di memoria su cui si realizzano operazioni.

Le differenze principali fra la RAM e il linguaggio macchina di un computer sono tre. La prima è che poiché siamo interessati ad un modello semplice per studiare cosa sia calcolabile e cosa ci focalizziamo sul calcolo aritmetico. Per questa ragione i registri di una RAM contengono solo numeri naturali $(0, 1, 2, 3 \ldots)$ e le operazioni sono definite solo su numeri naturali.

Le altre due differenze sono relative alla potenza di calcolo delle istruzioni. La seconda è che la RAM è dotata di un numero limitato di istruzioni rispetto al linguaggio macchina di una CPU odierna. Questa differenza non è essenziale dal punto di vista di cosa sia possibile calcolare perché le istruzioni mancanti possono essere simulate con un frammento di programma contenente sole istruzioni della RAM.

La terza differenza è che il modello delle RAM non ha vincoli sulla dimensione di una parola di memoria: una cella di memoria di una macchina RAM non ha vincoli e può contenere un numero arbitrario. In questo senso la RAM è apparentemente più potente di un computer moderno. In realtà questa maggiore capacità è solo apparente perché possiamo memorizzare dati su più celle di memoria. Non elaboriamo ulteriormente sulla equivalenza fra RAM e i moderni elaboratori e concludiamo che la RAM sono in grado di calcolare tutte le funzioni aritmetiche che possiamo calcolare con un computer.

Nel seguito presenteremo il modello delle RAM e poi discuteremo come esso sia equivalente alle macchine di Turing.

Una RAM ha un numero potenzialmente infinito di registri e un contatore di programma: come nel modello di von Neumann ogni registro è individuato da un indirizzo che assumiamo intero positivo e può contenere un numero naturale arbitrariamente grande. Un programma scritto per una RAM se termina può utilizzare un numero arbitrariamente grande di registri ma comunque finito.

Nel seguito, indichiamo con n un numero naturale, con (n) il contenuto del registro il cui indirizzo è n e con $[n]$ il contenuto del registro il cui indirizzo è contenuto nel registro il cui indirizzo è n (indirizzamento indiretto).

Un programma RAM è una sequenza ordinata di istruzioni (eventualmente dotate di un'etichetta) di uno dei seguenti tipi:

- $(n) := \alpha$ <operatore-aritmetico> β o [n] := α <operatore-aritmetico> β
  dove α e β sono per qualche intero positivo m o il valore m o il contenuto di (m) il contenuto di $[m]$ e <operatore-aritmetico> indica una delle quattro operazioni aritmetiche elementari: $+, -, *, /$. Questa istruzione assegna a (n) o a $[n]$ un nuovo valore dato dal risultato dell'operazione eseguita su α e β

- `goto etichetta` oppure `if` α <operatore-confronto> β `goto etichetta`
  dove <operatore-confronto> indica una delle quattro operazioni relazionali $=, \neq, \leq, <$ e etichetta indica un'istruzione del programma.
  Per comprendere la semantica di queste istruzioni ricordiamo che le istruzioni possono avere un'etichetta. L'etichetta di una istruzione è unica (cioè non esistono due istruzioni con la stessa etichetta). Nel caso di `goto etichetta` l'istruzione permette di modificare l'esecuzione del programma specificando l'etichetta della prossima istruzione da eseguire; nel secondo caso il salto è condizionato alla verifica del confronto fra α e β.

- `end`
  che segnala la fine del programma.

All'inizio dell'esecuzione di un programma RAM, i primi n registri contengono i numeri naturali che costituiscono l'input mentre tutti gli altri registri contengono 0 e il contatore di programma è uguale a 0.

Esempio Esempio calcolo somma n numeri

Come nel modello di von Neumann, ad ogni passo la macchina esegue l'istruzione specificata dal contatore di programma. Se tale istruzione non è un istruzione `if` oppure è un'istruzione `if` la cui

condizione non è verificata, il contatore di programma aumenta di 1 e, quindi, si esegue l'istruzione successiva; in caso contrario abbiamo un'istruzione di salto in cui il contatore di programma viene posto uguale all'indice dell'istruzione corrispondente all'etichetta dell'istruzione `if`.

Il programma termina quando si esegue l'istruzione end; in tal caso, l'output prodotto è uguale al contenuto del registro con indice 0.

Definizione 14. Una funzione $f : \mathbb{N}^n \to \mathbb{N}$ è RAM-calcolabile se esiste un programma RAM che, dato in input una sequenza $x_1, ..., x_n$ di n numeri naturali, termina producendo in output il valore $f(x_1, ..., x_n)$.

Anche se il repertorio delle istruzioni di una macchina RAM è molto ridotto rispetto all'insieme delle istruzioni di una moderna CPU osserviamo che le istruzioni del linguaggio macchina non presenti possono essere simulate facilmente con un frammento di programma per una RAM.

Nel seguito di come le macchine RAM non siano più potenti di quelle di Turing e che, quindi, esistano funzioni che non sono RAM-calcolabili.

Teorema 11. *Per ogni programma RAM esiste una macchina di Turing equivalente.*

Non diamo una prova formale e completa e ci limitiamo a discutere i passi essenziali di come possiamo simulare con una macchina di Turing un qualunque programma RAM.

Supponiamo quindi che sia dato un programma P scritto per una RAM e mostriamo come esista una macchina di Turing M equivalente. M ha diversi nastri:

1. il primo nastro contiene l'input di P e contiene, quindi, la sequenza di n numeri naturali $x_1, ..., x_n$ separati dal simbolo speciale &;

2. il secondo nastro funge da memoria e contiene inizialmente il solo simbolo speciale @; ad ogni passo nel corso della simulazione il secondo nastro contiene i valori memorizzati nella memoria della RAM e contiene una stringa in cui sono memorizzati tutti i valori dei registri utilizzati dalla RAM. La stringa è del tipo

   $$r1 : n_1 \& r2 : n_2 \ldots$$

   In essa, $r1$ indica il registro 1 e n_1 indica il valore intero del registro, e analogamente per gli altri registri. All'avvio di T, il nastro di memoria viene dunque inizializzato con i valori tutti posti a zero.

3. il terzo nastro contiene l'indicazione della prossima istruzione del programma P da eseguire. Quindi all'inzio il valore memorizzato è 1 (specificando la prima istruzione del programma).

4. i rimanenti nastri sono nastri di lavoro, inizialmente vuoti e il cui utilizzo sarà chiarito nel corso della definizione di T.

La macchina di Turing che simula il programma può essere suddivisa in tante macchine una per ciascuna istruzione del programma RAM. Se l'istruzione i-esima è un'istruzione di assegnazione la sottomacchina T_i è una macchina di Turing che inizialmente esamina la memoria alla ricerca dei dati specificati dall'istruzione ed esegue l'operazione specificata. Quindi T_i scrive il risultato sul secondo nastro e termina l'esecuzione passando il controllo alla sottomacchina T_{i+1}.

Se l'istruzione i-esima è un'istruzione `if` la sottomacchina T_i all'inizio legge gli operandi α e β in memoria e verifica la condizione. Se la condizione è verificata allora passa il controllo alla sottomacchina specificata nell'etichetta altrimenti alla sottomacchina T_{i+1}.

Se l'istruzione i-esima è una istruzione `end` la sottomacchina T_i ripulisce il nastro di memoria e termina.

5 Domande ed Esercizi

Esercizio 1. Definire in termini di insiemi i concetti di relazione e funzione.

Esercizio 2. Dire cos'è una funzione suriettiva; dare un esempio di funzione che lo è e una che non lo e'.

Esercizio 3. Dire la differenza in termini formali fra una relazione e una funzione; dare un esempio di ognuna.

Esercizio 4. Dimostrare che l'insieme $\{2^i \mid i >= 0\}$ e l'insieme dei numeri dispari e l'insieme dei numeri primi sono numerabili.

Esercizio 5. Dimostrare la non contabilita' dell'insieme delle funzioni da interi a interi (o delle relazioni su coppie di interi).

Esercizio 6. Dire almeno un problema non decidibile (dando di ognuno una definizione completa).

Esercizio 7. Esporre la tesi di Church-Turing.