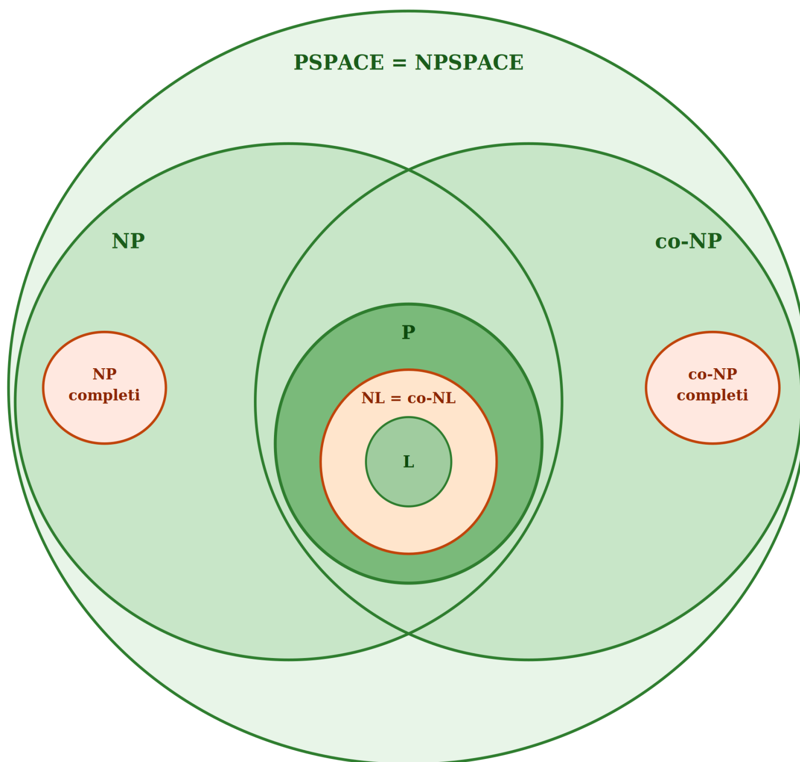
Introduzione alla calcolabilità e alla complessità computazionale

Alberto Bertoni, Massimiliano Goldwurm



Manuali

Alberto Bertoni
Massimiliano Goldwurm

Introduzione alla calcolabilità e alla complessità computazionale

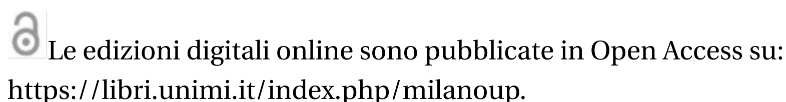
Milano University Press

Introduzione alla calcolabilità e alla complessità computazionale / Alberto Bertoni, Massimiliano Goldwurm. Milano: Milano University Press, 2026.

ISBN 979-12-5510-440-7 (print)
ISBN 979-12-5510-436-0 (PDF)
ISBN 979-12-5510-438-4 (EPUB)
DOI 10.54103/milanoup.284

Questo volume e, in genere, quando non diversamente indicato, le pubblicazioni di Milano University Press sono sottoposti a un processo di revisione esterno sotto la responsabilità del Comitato editoriale e del Comitato Scientifico della casa editrice. Le opere pubblicate vengono valutate e approvate dal Comitato editoriale e devono essere conformi alla politica di revisione tra pari, al codice etico e alle misure antiplagio espressi nelle Linee Guida per pubblicare su MilanoUP.

Le edizioni digitali dell'opera sono rilasciate con licenza Creative Commons Attribution 4.0 - CC-BY-SA, il cui testo integrale è disponibile all'URL: https://creativecommons.org/licenses/by-sa/4.0



Le edizioni digitali online sono pubblicate in Open Access su: https://libri.unimi.it/index.php/milanoup.

© The Author(s), 2026
© Milano University Press per la presente edizione

Pubblicato da:
Milano University Press
Via Festa del Perdono 7 – 20122 Milano
Sito web: https://milanoup.unimi.it
e-mail: redazione.milanoup@unimi.it

L'edizione cartacea del volume può essere ordinata in libreria ed è distribuita da Ledizioni (www.ledizioni.it)

Indice

Premessa **5**

1 Calcolabilità **7**

1.1 Nozioni preliminari e notazione 8
1.2 Linguaggio RAM ridotto 11
1.3 Linguaggio While 14
1.4 Compilatore 17
1.5 Macroistruzioni 22
1.6 Interprete 29
1.6.1 Programma universale 30
1.7 Funzioni ricorsive parziali 33
1.8 Linguaggio While e funzioni ricorsive 38
1.8.1 Tesi di Church 43
1.9 Sistemi di programmazione accettabili 44
1.9.1 Il teorema di ricorsione 47
1.10 Problemi indecidibili 50
1.11 Insiemi ricorsivi 54
1.12 Insiemi ricorsivamente numerabili 55
1.13 Il teorema di Rice 59
1.14 Insiemi completi 63

2 Complessità sequenziale **65**

2.1 Generalità sui linguaggi formali 66
2.2 Grammatiche 67
2.3 Automi a stati finiti 71
2.3.1 Linguaggi regolari e operazioni razionali 76
2.4 Macchine di Turing deterministiche 79
2.4.1 Macchine di Turing e funzioni 83
2.4.2 Macchine a più nastri 84

2.5 Complessità in tempo 85
2.5.1 La classe P e la tesi di Church estesa 90
2.6 Macchine di Turing non deterministiche 93
2.7 Confronto tra le classi DTIME e NTIME 97
2.8 La classe NP 99
2.9 Il problema della soddisfacibilità 103
2.10 Riducibilità polinomiale 104
2.10.1 Esempi notevoli 106
2.11 Problemi NP-completi 108
2.12 Complessità in spazio 112
2.13 Relazioni tra classi di complessità 116
2.14 Complessità in spazio del problema della raggiungibilità . . 119
2.14.1 Il teorema di Savitch 121
2.15 Chiusura rispetto al complemento 123
2.15.1 Il teorema di Immerman-Szelepscényi 125

Bibliografia **131**

Premessa

La calcolabilità delle funzioni e la complessità computazionale dei relativi problemi sono argomenti teorici di base dell'Informatica e in particolare dell'attività programmazione, che hanno stretti legami con diverse discipline matematiche. In questo breve testo si presentano gli aspetti essenziali di queste tematiche con uno spirito principalmente didattico e divulgativo, rivolto sopratutto agli studenti dei corsi di laurea a carattere scientifico, che possono ritrovare questi argomenti in diversi insegnamenti nel loro percorso di studi.

Il lavoro si divide in due capitoli: il primo riguarda la nozione di calcolabilità, mentre il secondo è centrato sulla complessità computazionale dei problemi decidibili. Il primo capitolo è essenzialmente una personale revisione e sistemazione delle dispense per gli studenti, dedicate alla calcolabilità, realizzate da Alberto Bertoni [1] negli anni '80 e '90, utilizzate anche dai suoi allievi in vari insegnamenti nell'ambito dei corsi di laurea di Scienze dell'Informazione e di Informatica [3]. L' obiettivo principale è quello di riproporre e trasmettere un materiale didattico che ritengo prezioso per tutta la comunità universitaria, che contiene molti aspetti originali e significativi e che rischia di andare perduto o di rimanere troppo nascosto senza un'adeguata pubblicazione e possibilità di diffusione in ambito scientifico. Uno dei pregi dell'approccio qui riproposto è la semplicità della presentazione della nozione di calcolabilità, introdotta usando e confrontando tra loro tradizionali costrutti di programmazione (ben noti anche agli studenti dei primi anni), che rendono il testo uno strumento didattico utile, relativamente facile da leggere e assimilare per uno studente universitario. Rispetto a quelle dispense, in questo primo capitolo sono state ridotte od omesse alcune parti iniziali, considerate troppo elementari, e le sezioni dedicate alla semantica a punto fisso, più adatte a un corso sui linguaggi di programmazione, mentre invece è stato ampliato il materiale riguardante i problemi indecidibili, gli insiemi ricorsivamente numerabili, gli insiemi

[1] Vedi il sito https://bertoni.di.unimi.it, liberamente accessibile, che contiene anche una sua biografia e una rassegna della sua attività didattica e di ricerca.

completi e, in generale, risulta più accurato il confronto tra i diversi formalismi usati per introdurre le funzioni calcolabili. Resta inteso che rimango l'unico responsabile delle modifiche apportate: gli eventuali errori presenti nel testo sono dunque riconducibili alla mia attività.

La seconda parte del lavoro (Capitolo 2) è invece dedicata alla complessità computazionale, contiene le nozioni e le proprietà che ritengo fondamentali per affrontare questa tematica, almeno in una fase iniziale. Questa parte è essenzialmente basata su argomenti classici, trattati in varie forme nella letteratura tradizionale, qui ampiamente citata nella bibliografia. I temi principali trattati in questa seconda parte riguardano le classi di complessità in tempo, i problemi NP-completi, il teorema di Cook-Levin e le principali proprietà della complessità in spazio, che comprendono i noti teoremi di Savitch e di Immerman-Szelepscényi.

Massimiliano Goldwurm
Milano, Novembre 2025

Capitolo 1

Calcolabilità

Questo capitolo è dedicato allo studio delle funzioni calcolabili, cioè di quelle funzioni che ammettono un algoritmo, ovvero un procedimento in grado di determinare automaticamente il valore della funzione su ogni possible argomento mediante l'esecuzione di un numero finito di operazioni elementari [1]. Presentata in questo modo, la calcolabilità di un funzione sembra dipendere dal formalismo usato per definire la nozione di algoritmo. È invece noto come tutti i formalismi naturali usati per definire algoritmi (intesi intuitivamente come metodi automatici di calcolo) diano luogo alla stessa famiglia di funzioni calcolabili. La tesi di Church rende precisa questa proprietà, affermando che l'insieme delle funzioni intuitivamente calcolabili coincide con la famiglia delle funzioni ricorsive parziali. Il primo obiettivo di queste note consiste proprio nella presentazione e giustificazione di questa tesi.

L'argomento successivo, altrettanto rilevante, riguarda lo studio delle funzioni non calcolabili. Si tratta di quelle funzioni che non ammettono un algoritmo di calcolo, per le quali quindi non esistono programmi, scritti in un qualunque linguaggio di programmazione, in grado di calcolarne il valore dato un suo argomento ricevuto come input. È evidente che la conoscenza delle caratteristiche di queste funzioni è fondamentale e propedeutica per l'attività di programmazione e per la progettazione di algoritmi.

Il materiale che qui presentiamo è classico. L'impostazione seguita in questo capitolo è ripresa dai testi [3, 15] da cui trae le ispirazioni principali.

[1] In questa sede consideriamo solo le funzioni definite sugli interi non negativi a valori sullo stesso insieme.

Presentazioni simili si trovano anche in altre opere tradizionali [8, 17] dove si possono trovare ulteriori sviluppi e approfondimenti.

1.1 Nozioni preliminari e notazione

In questa sede ci limitiamo a considerare le funzioni $f : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ che associano a ogni elemento $x \in \mathbb{N}$ un valore $f(x) \in \mathbb{N} \cup \{\perp\}$. Qui e nel seguito $\mathbb{N}$ rappresenta l'insieme dei numeri interi non negativi, mentre con $\mathbb{N}_+$ denoteremo la famiglia degli interi positivi. Invece $\perp$ rappresenta il simbolo di indefinito e per questo una funzione $f : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ è anche detta funzione *parziale*; diremo invece che f è *totale* se $f(x) \in \mathbb{N}$ per ogni $x \in \mathbb{N}$.

Ragionando in modo informale possiamo considerare un linguaggio di programmazione come una famiglia di programmi definiti mediante opportune regole (sintassi). È chiaro che ogni programma è specificato da una stringa finita di simboli e quindi codificabile mediante una cifra binaria. Così possiamo rappresentare ogni programma con un numero intero positivo e allineare i programmi in ordine crescente. Di conseguenza, un linguaggio di programmazione $\mathscr{L}$ può essere considerato come una successione di programmi $\{P_i\}_{i \in \mathbb{N}}$, dove P_i è l'i-esimo programma nell'allineamento dato. Inoltre ogni programma P_i riceve in input una stringa di simboli e, se la computazione su tale input si ferma, P_i restituisce un output costituito nuovamente da una stringa di simboli; entrambi input e output sono rappresentabili da cifre binarie e quindi da interi in $\mathbb{N}$. Così possiamo interpretare il comportamento di P_i come una funzione $f : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ tale che, per ogni $x \in \mathbb{N}$, $f(x) = \perp$ se P_i su input x non si ferma, mentre $f(x) = n \in \mathbb{N}$ se P_i su input x si ferma e restituisce n come output. Diremo anche che il programma P_i calcola la funzione f.

Intuitivamente possiamo allora dire che una funzione $f : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ è *calcolabile* se esiste un programma (o un algoritmo), scritto in un qualunque linguaggio di programmazione, che calcola f. Di conseguenza, l'insieme delle funzioni calcolabili in un qualunque linguaggio di programmazione è numerabile. Poiché è noto che l'insieme $\mathbb{N}^{\mathbb{N}}$ di tutte le funzioni (totali) $f : \mathbb{N} \to \mathbb{N}$ non è numerabile, questo prova che esistono funzioni da $\mathbb{N}$ in $\mathbb{N}$ non calcolabili. Più formalmente possiamo dimostrare il seguente risultato, che lascia all'intuizione la nozione di "linguaggio di programmazione".

Proposizione 1.1.1. *Dato un qualunque linguaggio di programmazione $\mathscr{L}$, esistono funzioni $g : \mathbb{N} \to \mathbb{N}$ non calcolabili in $\mathscr{L}$.*

Dimostrazione. Sia $\{P_i\}_{i\in\mathbb{N}}$ la sequenza dei programmi in $\mathscr{L}$ e, per ogni $i \in \mathbb{N}$, sia $f_i : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ la funzione calcolata dal programma P_i. Definiamo allora la funzione $g : \mathbb{N} \to \mathbb{N}$ tale che, per ogni $i \in \mathbb{N}$,

$$g(i) = \begin{cases} 0 & \text{se } f_i(i) = \perp \\ f_i(i) + 1 & \text{altrimenti} \end{cases}$$

Chiaramente g è totale. Supponi per assurdo che sia anche calcolabile. Allora esisterebbe $j \in \mathbb{N}$ tale che $g = f_j$ e quindi $g(j) = f_j(j) \neq \perp$; tuttavia per la stessa definizione di g avremmo $g(j) = f_j(j) + 1$ e quindi $f_j(j) = f_j(j) + 1$, assurdo. □

Abbiamo così mostrato l'esistenza di funzioni non calcolabili, cioè che non ammettono un programma in grado di determinarne il valore. Come vedremo nelle prossime sezioni, questo non dipende dal particolare linguaggio di programmazione considerato. L'interesse per la dimostrazione precedente risiede anche nell'uso della tecnica di diagonalizzazione che verrà spesso usata nel nostro contesto.

Nello studio delle funzioni calcolabili sono fondamentali alcune proprietà dei numeri interi, in particolare quelle che mostrano le corrispondenze tra interi, coppie di interi e più in generale k-ple di interi. Per illustrare questi risultati è particolarmente utile la famosa funzione coppia di Cantor e le sue naturali estensioni che saranno usate ripetutamente nelle sezioni successive.

Ricordiamo innanzitutto che per ogni coppia di insiemi A e B, B^A denota la famiglia delle funzioni definite su A a valori in B, ovvero $B^A = \{f : A \to B\}$. Inoltre, per ogni $n \in \mathbb{N}_+$, A^n rappresenta l'insieme delle n-ple di elementi di A, cioè $A^n = \{(a_1, \ldots, a_n) \mid a_i \in A \; \forall i = 1, \ldots, n\}$.

Gli elementi di $\mathbb{N}^2 = \mathbb{N} \times \mathbb{N}$ possono essere disposti su una matrice con (infinite) righe e colonne numerate dagli interi in $\mathbb{N}$. Così ad ogni coppia $(x, y) \in \mathbb{N}^2$ corrisponde la posizione di riga x e colonna y. Possiamo allora numerare le posizioni della tabella diagonalmente, a partire da 1 in poi, come mostrato nella seguente figura. In questo modo ogni posizione viene associata ad un numero intero positivo e la corrispondenza costruita definisce automaticamente una funzione biunivoca $\langle\cdot,\cdot\rangle : \mathbb{N}^2 \to \mathbb{N}_+$ che chiamiamo *funzione coppia.*

	0	1	2	3	4	...	y
0	1	3	6	10	15	...	...
1	2	5	9	14	...		...
2	4	8	13	...			...
3	7	12	...				...
4	11	...					...
...							...
x	...	...	...	...	...	...	$\langle x, y\rangle$

Per calcolare effettivamente il valore di $\langle x, y\rangle$, per ogni $x, y \in \mathbb{N}$, osserva che gli elementi che si trovano sulla colonna 0 sono dati da

$$\langle x, 0\rangle = 1 + \sum_{1}^{x} i = 1 + \frac{x(x+1)}{2}$$

Inoltre la posizione corrispondente al valore $\langle x, y\rangle$ si trova sulla stessa diagonale di $\langle x+y, 0\rangle$ e la differenza tra il primo e il secondo è proprio $\langle x, y\rangle - \langle x+y, 0\rangle = y$. Quindi, per ogni $x, y \in \mathbb{N}$, abbiamo che

$$\langle x, y\rangle = \langle x+y, 0\rangle + y = \frac{(x+y)(x+y+1)}{2} + y + 1 \qquad (1.1)$$

Analogamente la funzione coppia modificata $[\cdot,\cdot] : \mathbb{N}^2 \to \mathbb{N}$, data da

$$[x, y] = \langle x, y\rangle - 1 \qquad \forall x, y, \mathbb{N}$$

definisce una corrispondenza biunivoca tra $\mathbb{N}^2$ e $\mathbb{N}$.

Altre funzioni sono quelle che definiscono le corrispondenze inverse. Per ogni intero positivo $n \in \mathbb{N}_+$, chiamiamo parte sinistra e parte destra di n rispettivamente i valori $\sin(n) \in \mathbb{N}$ e $\mathrm{des}(n) \in \mathbb{N}$ tali che

$$n = \langle \sin(n), \mathrm{des}(n)\rangle$$

Quindi $\sin(n)$ e $\mathrm{des}(n)$ rappresentano rispettivamente la riga e la colonna corrispondenti alla n-esima posizione della tabella considerata sopra.

La funzione $[\cdot,\cdot]$ può essere estesa nel modo seguente:

- per ogni $x, y, z \in \mathbb{N}$ poniamo $[x, y, z] = [x, [y, z]]$. È chiaro che questa funzione definisce una corrispondenza biunivoca tra $\mathbb{N}^3$ e $\mathbb{N}$;

- per ogni $k \in \mathbb{N}$, $k \geq 2$ e ogni $x_1, x_2, \ldots, x_k \in \mathbb{N}$, definiamo

  $$[x_1, x_2, \ldots, x_k] = [x_1, [x_2, \cdots [x_{k-1}, x_k] \cdots]]$$

  Di nuovo questa funzione stabilisce una corrispondenza biunivoca tra $\mathbb{N}^k$ e $\mathbb{N}$.

Infine si può dimostrare che anche l'insieme $\bigcup_{k=1}^{+\infty} \mathbb{N}^k$ è in corrispondenza biunivoca con $\mathbb{N}_+$. Infatti si può definire la funzione coppia generalizzata

$$\langle\langle \cdots \rangle\rangle : \bigcup_{k=1}^{+\infty} \mathbb{N}^k \to \mathbb{N}_+$$

tale che, per ogni $k \in \mathbb{N}_+$ e ogni $(x_1, x_2, \ldots, x_k) \in \mathbb{N}^k$,

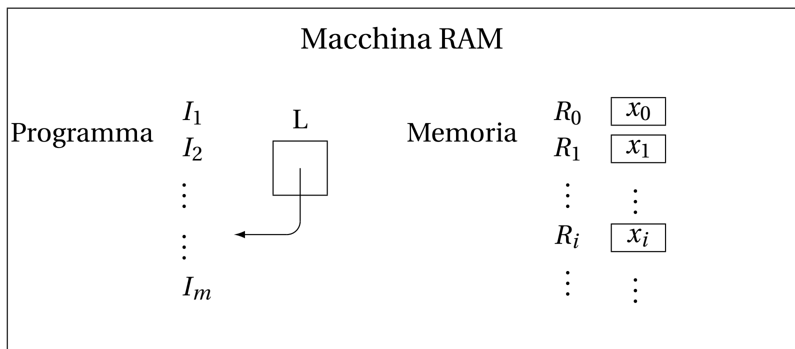
$$\langle\langle x_1, x_2, \ldots, x_k \rangle\rangle = \langle x_1, \langle x_2, \cdots, \langle x_k, 0 \rangle \cdots \rangle \rangle$$

Usando le proprietà di $\langle \cdot, \cdot \rangle$, sin e des, non è difficile provare che anche $\langle\langle \cdots \rangle\rangle$ è suriettiva e iniettiva, e quindi biunivoca.

1.2 Linguaggio RAM ridotto

In questa sezione introduciamo un linguaggio di programmazione molto semplice ottenuto semplificando il tradizionale linguaggio RAM (solitamente considerato nei corsi di Algoritmi [1, 4]).

Questo formalismo si basa sulla nozione di RAM (macchina ad accesso casuale), un modello di calcolo astratto costituito da una memoria, formata da infiniti registri numerati $R_0, R_1, \ldots, R_k, \ldots$, da un programma P, rappresentato da una sequenza finita di istruzioni $I_1, I_2, \ldots, I_m$ e da un contatore L che può essere pensato come un puntatore a una delle istruzioni del programma. Ogni registro contiene un numero intero non negativo (cioè un valore $n \in \mathbb{N}$) di grandezza arbitraria che può variare durante il calcolo. Il programma invece è incorporato nella macchina e non è modificabile; esso è costituito da istruzioni che definiamo nel seguito. Il contatore L indica infine l'istruzione corrente da eseguire.



La sintassi del linguaggio RAM è definita dalle istruzioni e dai programmi. Una istruzione RAM è un'espressione di una delle seguenti forme:

- $R_k \leftarrow R_k + 1$,
- $R_k \leftarrow R_k \dot{-} 1$,
- `if` $R_k = 0$ `then go to` n

dove $k \in \mathbb{N}$ e $n \in \mathbb{N}_+$. Qui $\dot{-}$ rappresenta la sottrazione in $\mathbb{N}$, ovvero per ogni $i, j \in \mathbb{N}$, $i \dot{-} j = 0$ se $i < j$, mentre $i \dot{-} j = i - j$ altrimenti. Denotiamo con $\Im$ l'insieme di tutte le istruzioni RAM.

Un programma RAM è una sequenza finita di istruzioni RAM, quindi una successione $(I_1, I_2, \ldots, I_m)$ dove $I_j \in \Im$ per ogni $j = 1, 2, \ldots, m$. Chiamiamo inoltre *Programmi* l'insieme di tutti i programmi RAM e quindi

$$Programmi = \bigcup_{m \geq 1} \Im^m$$

Nota che l'insieme dei programmi RAM è numerabile.

Definiamo ora la semantica del linguaggio RAM. Vogliamo associare ad ogni $P \in Programmi$ la funzione $\Psi_P : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ calcolata da P. Questa definizione sarà guidata dal modello di macchina RAM descritto sopra. Per questo motivo la semantica che introduciamo è detta semantica operazionale.

Innanzitutto chiamiamo stato di una macchina RAM una qualsiasi funzione

$$S : \{L, R_0, R_1, \ldots, R_k, \ldots\} \to \mathbb{N}$$

Diciamo che una macchina RAM si trova nello stato S se L assume il valore $S(L)$ e ogni registro R_k contiene il valore $S(R_k)$ per tutti i $k \in \mathbb{N}$. Uno stato è quindi una immagine istantanea della macchina durante l'esecuzione di un calcolo, esso definisce il contenuto di tutti i registri della memoria e l'indice dell'istruzione corrente da eseguire. Denotiamo con *Stati* la famiglia di tutti gli stati. Diciamo che uno stato S è finale o di arresto se $S(L) = 0$.

Inoltre, per ogni $x \in \mathbb{N}$, lo stato iniziale su input x è lo stato in_x tale che $\text{in}_x(L) = 1$, $\text{in}_x(R_1) = x$ e $\text{in}_x(R_j) = 0$ per ogni intero non negativo $j \neq 1$. Questo è lo stato nel quale si trova qualsiasi macchina RAM quando inizia la computazione sull'input x.

L'esecuzione di una istruzione può essere descritta formalmente mediante una funzione di transizione tra stati. Questa è definita dalla funzione

$$\delta : Stati \times \text{Programmi} \to Stati \cup \{\perp\}$$

che associa ad ogni stato S e a ogni programma RAM $P = (I_1, I_2, \dots, I_m)$ lo stato T che si ottiene da S eseguendo l'instruzione $I_{S(L)}$ quando questa è ben definita. Formalmente, per ogni $S \in \mathit{Stati}$ e ogni $P \in$ Programmi, poniamo $\delta(S, P) = T$ dove T è definito nel modo seguente:

- se $S(L) = 0$ allora $T = \bot$;
- se $S(L) > m$ allora T è uno stato tale che $T(L) = 0$ e $T(R_k) = S(R_k)$ per ogni $k \in \mathbb{N}$;
- se $1 \leq S(L) \leq m$ allora $T \in \mathit{Stati}$ dipende dall'istruzione $I_{S(L)}$, ovvero
  - $I_{S(L)} \equiv R_k \leftarrow R_k + 1$ implica
  $$\begin{cases} T(L) = S(L) + 1, \\ T(R_k) = S(R_k) + 1, \\ T(R_j) = S(R_j) \quad \text{per ogni } j \neq k. \end{cases}$$
  - $I_{S(L)} \equiv R_k \leftarrow R_k \dot{-} 1$ implica
  $$\begin{cases} T(L) = S(L) + 1, \\ T(R_k) = S(R_k) \dot{-} 1, \\ T(R_j) = S(R_j) \quad \text{per ogni } j \neq k. \end{cases}$$
  - $I_{S(L)} \equiv \texttt{if } R_k = 0 \texttt{ then go to } n$ implica
  $$\begin{cases} T(L) = \begin{cases} n & \text{se } S(R_k) = 0, \\ S(L) + 1 & \text{se } S(R_k) \neq 0, \end{cases} \\ T(R_j) = S(R_j) \quad \text{per ogni } j \in \mathbb{N}. \end{cases}$$

Per ogni programma RAM P e ogni $x \in \mathbb{N}$, diciamo che P si ferma su input x se esiste una sequenza finita di stati $\mathscr{C} = (S_0, S_1, \dots, S_t)$ tale che

1. $S_0 = \mathrm{in}_x$,
2. $S_i = \delta(S_{i-1}, P)$ per ogni $i = 1, 2, \dots, t$,
3. $S_t(L) = 0$.

La sequenza $\mathscr{C}$ è chiamata computazione di P su x e S_t è il suo stato di arresto.

Se invece P non si ferma su input x chiamiamo comunque computazione di P su x la sequenza infinita di stati $\mathscr{C} = \{S_i\}_{i \in \mathbb{N}}$ tale che $S_0 = \mathrm{in}_x$ e $S_i = \delta(S_{i-1}, P)$ per ogni $i \in \mathbb{N}_+$.

Ovviamente esiste sempre una e una sola computazione di un programma RAM P su un dato input x e questa è finita se e solo se P su x si ferma.

Siamo ora in grado di definire la funzione calcolata da un programma P. Questa è data dalla funzione $\Psi_P : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ tale che, per ogni $x \in \mathbb{N}$,

$$\Psi_P(x) = \begin{cases} \perp & \text{se } P \text{ su input } x \text{ non si ferma} \\ S_t(R_0) & \text{se } P \text{ su input } x \text{ si ferma e} \\ & \quad S_t \text{ è lo stato di arresto della computazione di } P \text{ su } x \end{cases}$$

R_0 è quindi il registro che contiene alla fine della computazione il risultato del calcolo, ovvero l'output del programma.

Denotiamo infine con *Funzioni(RAM)* la famiglia delle funzioni calcolabili dai programmi RAM:

$$\mathit{Funzioni(RAM)} = \{f : \mathbb{N} \to \mathbb{N} \cup \{\perp\} \mid \exists P \in \mathit{Programmi} : \Psi_P = f\}$$

1.3 Linguaggio While

Il linguaggio presentato nella sezione precedente è di fatto un formalismo elementare ispirato ai linguaggi macchina. La sue caratteristiche principali sono l'estrema semplicità delle istruzioni e il suo legame con l'architettura di base di un calcolatore. È chiaro però che il linguaggio RAM non è adatto alla definizione e alla progettazione di algoritmi. In particolare l'istruzione di salto condizionato rende i programmi difficili da comprendere e da sviluppare.

Per rappresentare meglio i formalismi utilizzati per questi scopi presentiamo in questa sezione il linguaggio While. Si tratta di un linguaggio ad alto livello, più vicino alla tradizionale attività di programmazione, in grado di descrivere alcuni tipici schemi e costrutti utilizzati spesso nella definizione di algoritmi e procedure di calcolo.

La sintassi del linguaggio While è basata sulla nozione di comando e sulla gestione di un numero finito di variabili $x_0, x_1, \ldots, x_{20}$.

In questo linguaggio un comando è un'espressione di uno dei seguenti tipi:

- Comandi di assegnamento

$$x_k := 0 \quad (1.2)$$

$$x_k := x_j + 1 \quad (1.3)$$

$$x_k := x_j \dot{-} 1 \quad (1.4)$$

dove $k, j \in \{0, 1, \ldots, 20\}$;

- Comando while

$$\texttt{while } x_k \neq 0 \texttt{ do } C$$

dove C è a sua volta un comando e $k = 0, 1, \ldots, 20$;

- Comando composto

$$\texttt{begin } C_1\, C_2 \ldots C_m \texttt{ end}$$

dove C_1, C_2, ..., C_m sono comandi.

Inoltre, un programma While è un comando composto. Nel seguito denoteremo con *Comandi* l'insieme di tutti i comandi appena definiti, e con W-Programmi l'insieme di tutti i programmi While.

Nota l'assenza di una istruzione di salto condizionato o meno (go to). Inoltre, è naturale utilizzare in questo contesto il seguente principio di induzione strutturata: una proprietà R vale per tutti i comandi, e quindi anche per i programmi While, se

- R vale per tutti i comandi di assegnamento;
- supponendo R valida per un comando C si dimostra che R vale per il comando `while` $x_k \neq 0$ `do` C, per ogni $k = 0, \ldots, 20$;
- supponendo R valida per i comandi $C_1, C_2, \ldots, C_m$ si dimostra che R vale anche per il comando `begin` $C_1\, C_2 \ldots C_m$ `end` .

Useremo questo principio sia per provare proprietà dei comandi sia per definire nozioni associate ai programmi While.

Definiamo ora una semantica operazionale dei programmi While. In questo caso associamo ogni comando a una funzione che modifica opportunamente il valore delle variabili $x_0, x_1, \ldots, x_{20}$. Di fatto, nello scenario descritto, il modello di calcolo è semplicemente ridotto alle stesse variabili pensate come registri che contengono un valore.

Innanzitutto chiamiamo w-stato una funzione che associa a ogni variabile un valore intero non negativo

$$y : \{x_0, \ldots, x_{20}\} \to \mathbb{N}$$

Tale funzione può essere rappresentata da un vettore di interi a 21 componenti $(y_0, y_1, \ldots, y_{20})$, dove ogni y_i è $y(x_i)$. Quindi $\mathbb{N}^{21}$ è l'insieme di tutti

i w-stati. Per comodità rappresenteremo sempre le componenti di un w-stato $y \in \mathbb{N}^{21}$ mediante indici da 0 a 20, $y = (y_0, y_1, \ldots, y_{20})$. Per ogni $a \in \mathbb{N}$, il w-stato iniziale su input a è dato da $w\text{-in}(a) = (0, a, 0, \ldots, 0)$.

Per ogni comando C, denotiamo con $[[C]]$ la funzione semantica

$$[[C]] : \mathbb{N}^{21} \to \mathbb{N}^{21} \cup \{\perp\}$$

definita dalle seguenti identità (per qualsiasi $y \in \mathbb{N}^{21}$ e $i \in \{0, \ldots, 20\}$):

1. se $C \equiv x_k := 0$ allora $[[C]](y)_i = \begin{cases} y_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}$

2. se $C \equiv x_k := x_j + 1$ allora $[[C]](y)_i = \begin{cases} y_i & \text{se } i \neq k \\ y_j + 1 & \text{se } i = k \end{cases}$

3. se $C \equiv x_k := x_j \dot{-} 1$ allora $[[C]](y)_i = \begin{cases} y_i & \text{se } i \neq k \\ y_j \dot{-} 1 & \text{se } i = k \end{cases}$

4. se $C = \texttt{begin}\ C_1\ C_2\ \ldots\ C_m\ \texttt{end}$ allora

$$[[C]](y) = [[C_m]](\ldots([[C_1]](y))\ldots)$$

con la convenzione che $[[C]](y) = \perp$ se $[[C_j]](\ldots) = \perp$ per qualche $j \in \{1, 2, \ldots, m\}$.

Infine, per definire $[[C]](y)$ quando C è un comando while, introduciamo le seguenti funzioni che saranno utili anche nel seguito:

- per ogni $y = (y_0, y_1 \ldots, y_{20}) \in \mathbb{N}^{21}$ e ogni $k \in \{0, 1, \ldots, 20\}$,

$$\text{Pro}_k(y) = y_k$$

- per ogni funzione $f : \mathbb{N}^{21} \to \mathbb{N}^{21} \cup \{\perp\}$ e ogni $t \in \mathbb{N}$, definiamo la funzione
$f^t : \mathbb{N}^{21} \to \mathbb{N}^{21} \cup \{\perp\}$ tale che

$$f^t(y) = \begin{cases} y & \text{se } t = 0 \\ f(f^{t-1}(y)) & \text{se } t > 0 \end{cases} \qquad \forall\ y \in \mathbb{N}^{21}$$

con la convezione che $f(\perp) = \perp$;

- per ogni funzione $f : \mathbb{N}^{21} \to \mathbb{N}^{21} \cup \{\perp\}$, ogni $y \in \mathbb{N}^{21}$ e ogni $k \in \{0, 1, \ldots, 20\}$, definiamo

$$\mu(f, k, y) = \begin{cases} r \in \mathbb{N} & \text{se } r = \min\{t \in \mathbb{N} \mid \text{Pro}_k(f^t(y)) = 0\} \text{ esiste} \\ & \quad \text{e inoltre } f^j(y) \neq \perp \text{ per ogni } j = 0, 1, \ldots, r-1\,; \\ \perp & \text{altrimenti} \end{cases}$$

Chiaramente $\mu(f, k, y) = \perp$ se $\text{Pro}_k(f^t(y)) > 0$ per tutti i $t \in \mathbb{N}$.

Allora possiamo porre, per ogni comando $C \equiv \texttt{while}\ x_k \neq 0\ \texttt{do}\ A$ e ogni $y \in \mathbb{N}^{21}$,

5. $[[C]](y) = \begin{cases} \perp & \text{se } \mu([[A]], k, y) = \perp \\ [[A]]^r(y) & \text{altrimenti, dove } r = \mu([[A]], k, y), \end{cases}$

Siamo ora in grado di definire la funzione calcolata da un qualsiasi programma While. Per ogni $P \in W$-Programmi e ogni $x \in \mathbb{N}$

$$\Phi_P(x) = \begin{cases} \perp & \text{se } [[P]](w\text{-in}(x)) = \perp \\ \text{Pro}_0([[P]](w\text{-in}(x))) & \text{altrimenti} \end{cases}$$

Infine, come per il linguaggio RAM, denotiamo con *Funzioni*(While) la famiglia delle funzioni calcolate dai programmi While:

$$\textit{Funzioni}(\text{While}) = \{f : \mathbb{N} \to \mathbb{N} \cup \{\perp\} \mid \exists P \in W\text{-Programmi} : \Phi_P = f\}$$

L'obiettivo delle prossime sezioni è quello di provare che il linguaggio RAM e il linguaggio While hanno la stessa potenza computazionale, ovvero calcolano la stessa classe di funzioni.

Più precisamente vogliamo dimostrare che *Funzioni*(While) = *Funzioni*(RAM).

1.4 Compilatore

Consideriamo due linguaggi di programmazione $\mathscr{L}_1$ e $\mathscr{L}_2$ visti come insiemi di programmi e denotiamo con $\{\phi_P\}_{p \in \mathscr{L}_1}$ e $\{\psi_P\}_{p \in \mathscr{L}_2}$ le rispettive famiglie di funzioni calcolate. Un compilatore (o traduttore) da $\mathscr{L}_1$ a $\mathscr{L}_2$ è una funzione $T : \mathscr{L}_1 \to \mathscr{L}_2$ tale che

1. $\phi_P = \psi_{T(P)}$ per ogni $P \in \mathscr{L}_1$;

2. T è calcolabile in un qualunque linguaggio di programmazione.

In questa sezione intendiamo presentare un compilatore dalla famiglia dei programmi While a quella dei programmi RAM. La sua correttezza implica automaticamente che ogni funzione calcolabile da un programma while è anche calcolabile da un programma RAM. A questo scopo, definiamo una funzione *Comp* che associa a ogni comando while un programma RAM che esegue lo stesso calcolo. L'idea è quella di rappresentare le variabili $x_0, x_1, \ldots, x_{20}$ mediante i registri $R_0, R_1, \ldots, R_{20}$, usando R_{21} e R_{22} come registri ausiliari e simulare il ciclo while mediante il salto condizionato.

Formalmente, definiamo la funzione

$$Comp : \text{Comandi} \to \text{Programmi}$$

per induzione strutturata.

1. Per ogni comando di assegnamento C, il programma $Comp(C)$ è definito nel modo seguente:

   - per ogni $k \in \{0, 1, \ldots, 20\}$ abbiamo

$$Comp(x_k := 0) \equiv \begin{cases} a \;\; \texttt{if } R_k = 0 \texttt{ then go to } e \\ \quad R_k \leftarrow R_k \dot{-} 1 \\ \quad \texttt{if } R_{21} = 0 \texttt{ then go to } a \\ e \;\; R_k \leftarrow R_k \dot{-} 1 \end{cases}$$

     dove $e = a + 3$ e il valore dell'etichetta a sarà poi stabilito da un contatore delle istruzioni del programma da determinare, opportunamente inizializzato e gestito;

   - ovviamente $Comp(x_k := x_k + 1) \equiv R_k \leftarrow R_k + 1$ e

     $Comp(x_k := x_k \dot{-} 1) \equiv R_k \leftarrow R_k \dot{-} 1$, per ogni k;

- per ogni $k, j \in \{0, 1, \ldots, 20\}$ tali che $k \neq j$, abbiamo

$$Comp(x_k := x_j + 1) \equiv \begin{cases} a \;\; \texttt{if } R_k = 0 \texttt{ then go to } e1 \\ \quad R_k \leftarrow R_k \dot{-} 1 \\ \quad \texttt{if } R_{21} = 0 \texttt{ then go to } a \\ e1 \;\; \texttt{if } R_j = 0 \texttt{ then go to } e2 \\ \quad R_j \leftarrow R_j \dot{-} 1 \\ \quad R_{21} \leftarrow R_{21} + 1 \\ \quad \texttt{if } R_{22} = 0 \texttt{ then go to } e1 \\ e2 \;\; \texttt{if } R_{21} = 0 \texttt{ then go to } e3 \\ \quad R_{21} \leftarrow R_{21} \dot{-} 1 \\ \quad R_k \leftarrow R_k + 1 \\ \quad R_j \leftarrow R_j + 1 \\ \quad \texttt{if } R_{22} = 0 \texttt{ then go to } e2 \\ e3 \;\; R_k \leftarrow R_k + 1 \end{cases}$$

  dove di nuovo, $e1 = a + 3$, $e2 = a + 7$, $e3 = a + 12$ e il valore di a è stabilito come prima dal contatore considerato. Inoltre $Comp(x_k := x_j \dot{-} 1)$ è definito in maniera analoga, basta sostituire nel programma precedente l'ultima istruzione con $e3 \; R_k \leftarrow R_k \dot{-} 1$.

2. Per ogni comando composto $C \equiv \texttt{begin } C_1 \, C_2 \ldots C_m \texttt{ end}$ poniamo

$$Comp(C) \equiv \begin{cases} Comp(C_1) \\ Comp(C_2) \\ \quad . \\ \quad . \\ \quad . \\ Comp(C_m) \end{cases}$$

   dove, nella concatenazione dei programmi RAM $Comp(C_i)$, per $i = 1, 2, \ldots, m$, le etichette delle istruzioni sono aggiornate in modo ovvio.

3. Per ogni comando while $C \equiv \texttt{while } x_k \neq 0 \texttt{ do } A$ poniamo

$$Comp(C) \equiv \begin{cases} a \;\; \texttt{if } R_k = 0 \texttt{ then go to } e \\ \quad Comp(A) \\ \quad \texttt{if } R_{21} = 0 \texttt{ then go to } a \\ e \;\; R_{21} \leftarrow R_{21} \dot{-} 1 \end{cases}$$

dove $e = a+2+\ell$, essendo ℓ la lunghezza di $Comp(A)$ (ovvero il numero delle sue istruzioni) e le etichette delle istruzioni presenti in $Comp(A)$ sono opportunamente aggiornate. Anche in questo caso il valore dell'etichetta a può essere determinato dal contatore considerato.

Abbiamo così definito la funzione *Comp*.

È relativamente facile descrivere un algoritmo ad alto livello che, su input $C \in Comandi$, calcola il programma $Comp(C)$. L'algoritmo può essere descritto da un main che inizializza il contatore delle istruzioni RAM (da usare per attribuire i valori alle etichette a delle varie istruzioni via via ottenute) e richiama una procedura ricorsiva la quale a sua volta è definita per induzione strutturata, usando di fatto i punti 1, 2, 3 illustrati sopra.

Dobbiamo ora dimostrare la correttezza della traduzione, ovvero che il comando C e il programma $Comp(C)$ calcolano la stessa funzione per ogni $C \in Comandi$. A questo scopo, definiamo innazintutto una versione della funzione transizione ristretta ai primi 21 registri della macchina RAM.

Per ogni $z \in \mathbb{N}^{21}$, $z = (z_0, z_1, \dots, z_{20})$, denotiamo con S_z lo stato che assegna i valori di z ai primi registri della macchina, azzera gli altri e indica la prima istruzione del programma come istruzione corrente:

$$S_z \text{ è tale che } \begin{cases} S_z(L) = 1 & \\ S_z(R_i) = z_i & \forall\ i = 0, \dots, 20 \\ S_z(R_j) = 0 & \forall\ j \geq 21 \end{cases}$$

Per ogni programma RAM P chiamiamo computazione di P su $z \in \mathbb{N}^{21}$ (se esiste) la sequenza finita di stati $(S_0, S_1, \dots, S_j)$ tale che $S_0 = S_z$, $S_i = \delta(S_{i-1}, P)$ per ogni $i = 1, 2, \dots, j$ e $S_j(L) = 0$. Definiamo allora $\delta_P(z)$ nel modo seguente:

$$\delta_P(z) = \begin{cases} \big(S_j(R_0), S_j(R_1), \dots, S_j(R_{20})\big) & \text{se la computazione di } P \text{ a partire da } S_z \text{ termina} \\ & \text{e } S_j \text{ è il corrispondente stato di arresto} \\ \perp & \text{altrimenti.} \end{cases}$$

Di conseguenza, δ_P è una funzione

$$\delta_P : \mathbb{N}^{21} \to \mathbb{N}^{21} \cup \{\perp\}$$

che intuitivamente rappresenta il calcolo eseguito da P limitatamente ai primi 21 registri.

Proposizione 1.4.1. *Per ogni* $C \in Comandi$, $[[C]] = \delta_{Comp(C)}$.

Dimostrazione. Ragioniamo per induzione strutturata. Se C è un comando di assegnamento la proprietà è ovvia e deriva dalla definizione stessa di $Comp(C)$.

Se $C \equiv \texttt{begin}\ C_1\ C_2\ \dots\ C_m\ \texttt{end}$ e $[[\delta_{Comp(C_i)}]] = [[C_i]]$ per ogni $i = 1, 2, \dots, m$, allora

$$Comp(C) \equiv \begin{cases} Comp(C_1) \\ Comp(C_2) \\ \cdot \\ \cdot \\ \cdot \\ Comp(C_m) \end{cases}$$

e, per ogni $z \in \mathbb{N}^{21}$, abbiamo

$$\begin{aligned} \delta_{Comp(C)}(z) &= \delta_{Comp(C_m)}(\delta_{Comp(C_{m-1})}(\cdots \delta_{Comp(C_1)}(z)\cdots)) \\ &= [[C_m]]([[C_{m-1}]]\cdots[[C_1]](z)\cdots)) = [[C]](z) \end{aligned} \tag{1.5}$$

Infine, sia $C \equiv \texttt{while}\ x_k \neq 0\ \texttt{do}\ A$ e supponi $[[A]] = \delta_{Comp(A)}$. Allora, per ogni $z \in \mathbb{N}^{21}$, sappiamo che

$$[[C]](z) = [[A]]^t(z) \quad \text{dove } t = \min\{n \in \mathbb{N} \mid \text{Pro}_k([[A]]^n(z)) = 0\} = \mu([[A]], k, z)$$

e inoltre

$$\begin{aligned} \delta_{Comp(C)}(z) &= \delta^u_{Comp(A)}(z) \\ &\text{dove} \\ u &= \min\{n \in \mathbb{N} \mid \text{Pro}_k(\delta^n_{Comp(A)}(z)) = 0\} \\ &= \mu(\delta_{Comp(A)}, k, z) \end{aligned}$$

Applicando l'ipotesi di induzione si verifica subito che $t = u$ e quindi per lo stesso motivo $[[C]](z) = \delta_{Comp(C)}(z)$ e questo conclude la prova. □

Corollario 1.4.2. *Per ogni programma While P e ogni $x \in \mathbb{N}$*

$$\Phi_P(x) = \Psi_{Comp(P)}(x)$$

Dimostrazione. Per la proposizione precedente e per definizione di Φ_P abbiamo

$$\Phi_P(x) = \text{Pro}_0([[P]](w\text{-in}(x))) = \text{Pro}_0(\delta_{Comp(C)}(0, x, 0, \dots, 0)) = \Psi_{Comp(P)}(x)$$

□

Abbiamo così ottenuto due risultati significativi.

Teorema 1.4.3. *Valgono le seguenti proprietà:*

1. *La funzione Comp è un traduttore;*
2. *Funzioni*$(While) \subseteq$ *Funzioni*(RAM).

1.5 Macroistruzioni

Per dimostrare l'inclusione nel senso opposto, ovvero per provare che le funzioni RAM sono calcolabili dai programmi While, è opportuno introdurre comandi while più complessi di quelli considerati sinora. Occorre infatti trovare il modo di mantenere con un numero limitato di variabili (21) una memoria RAM formata da un numero arbitrario di registri. A questo scopo, definiamo un nuovo insieme di comandi while che chiameremo "macroistruzioni". Queste possono essere pensate come abbreviazioni di piccoli programmi While, in grado di calcolare varie funzioni, tra cui tutte le operazioni aritmetiche tradizionali, la funzione coppia e le proiezioni corrispondenti, oppure di eseguire altre tradizionali istruzioni di controllo come i comandi "if then else".

Ogni macroistruzione che ora introduciamo è definita da un programma While preciso, soggetto a eventuali vincoli sull'uso delle variabili. In generale, questi nuovi comandi utilizzano le variabili $x_{14}, \ldots, x_{20}$ come registri ausiliari, usati per mantenere risultati parziali e svolgere operazioni collaterali. Nella maggior parte dei casi si tratta di programmi che assegnano alla variabile x_k un valore che dipende dalle variabili x_i e x_j. I valori degli indici i, j e k, a seconda dei casi, potranno coincidere in tutto o in parte, oppure essere tutti diversi tra loro. Essi posssono sempre variare nell'insieme $\{0, 1, \ldots, 13\}$, e in alcuni casi assumere anche i valori successivi, compresi tra 14 e 20, con opportuni vincoli che dipendono dai singoli programmi. In ogni caso, al termine dell'esecuzione di ciascuna macroistruzione, se i e j sono diversi da k i valori di x_i e x_j restano invariati.

Le macroistruzioni più semplici sono le seguenti:

1. $x_k := n$, dove $n \in \mathbb{N}$ e $k \in \{0, 1, \ldots, 20\}$, è definita da

$$\texttt{begin}\ \ x_k := 0\ \ x_k := x_k + 1\ \cdots\ x_k := x_k + 1\ \texttt{end}$$

dove il comando $x_k := x_k + 1$ è ripetuto esattamente n volte;

2. $x_k := x_j$, dove $k, j \in \{0, 1, \dots, 20\}$, è definita da

$$\texttt{begin}\ \ x_k := x_j + 1\ \ x_k := x_k \dot{-} 1\ \texttt{end}$$

3. $x_k := x_i + x_j$ è definita dalla seguente procedura, nella quale gli indici i, j e k sono diversi da 14 (nel seguito scriveremo semplicemente $i, j, k \neq 14$):

$$
\begin{array}{l}
\texttt{begin} \\
\qquad x_{14} := x_j \\
\qquad x_k := x_i \\
\qquad \texttt{while}\ x_{14} \neq 0\ \texttt{do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_k := x_k + 1 \\ \quad x_{14} := x_{14} \dot{-} 1 \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

Osserva che in questa procedura gli indici i, j e k possono coincidere in tutto o in parte (purché siano diversi da 14);

4. $x_k := x_i \dot{-} x_j$ è definita in modo simile, con gli stessi vincoli (determinarla per esercizio);

5. $x_k := x_i \cdot x_j$, dove $i \neq k$ e $i, j, k \neq 14, 15$, equivale a

$$
\begin{array}{l}
\texttt{begin} \\
\qquad x_{15} := x_j \\
\qquad x_k := 0 \\
\qquad \texttt{while}\ x_{15} \neq 0\ \texttt{do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_k := x_k + x_i \\ \quad x_{15} := x_{15} \dot{-} 1 \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

Se $i = k \neq j$ possiamo definire $x_k := x_k \cdot x_j$ come $x_k := x_j \cdot x_k$ mentre, se $i = j = k$ esiste un programma ancora più semplice (che può essere trovato per esercizio);

6. $x_k := x_i : x_j$, intesa come divisione intera, con i vincoli $i, j, k \neq 14, 15$, $j \neq k$ e $x_j \neq 0$, può essere definita in modo simile, sostituendo la somma con una sottrazione all'interno del ciclo while e adattando opportunamente i primi comandi:

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_{15} := x_i + 1 \\
\quad x_{15} := x_{15} \dot{-} x_j \\
\quad x_k := 0 \\
\quad \texttt{while } x_{15} \neq 0 \texttt{ do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_k := x_k + 1 \\ \quad x_{15} := x_{15} \dot{-} x_j \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

7. $x_k := x_i (\text{mod } x_j)$ può essere definita in maniera analoga, con gli stessi vincoli:

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_{15} := x_i + 1 \\
\quad x_{15} := x_{15} \dot{-} x_j \\
\quad x_k := x_i \\
\quad \texttt{while } x_{15} \neq 0 \texttt{ do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_k := x_k \dot{-} x_j \\ \quad x_{15} := x_{15} \dot{-} x_j \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

8. $x_k := \langle x_i, x_j \rangle$, dove $i, j, k \neq 14, 15$, è descritta dal seguente programma che applica la definizione data dalla relazione (1.1):

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_{15} := x_i + x_j \\
\quad x_k := x_j + 1 \\
\quad \texttt{while } x_{15} \neq 0 \texttt{ do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_k := x_k + x_{15} \\ \quad x_{15} := x_{15} \dot{-} 1 \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

9. $x_k := \sin(x_i)$, dove $i \neq k$ e $i, k \neq 14, 15, 16$, può essere definita da

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_{15} := x_i
\end{array}
$$

```
        x_k := 0
        x_16 := 0
                          ⎧ begin
                          ⎪   x_16 := x_16 + 1
        while x_15 ≠ 0 do ⎨   x_15 := x_15 ∸ x_16
                          ⎪   x_k := x_k + x_16
                          ⎩ end
        x_k := x_k ∸ x_i
end
```

10. $x_k := \mathrm{des}(x_i)$, dove $i \neq k$ e $i, k \neq 14, 15, 16$ è invece data da

```
begin
        x_k := sin(x_i)
        x_16 := x_16 ∸ 1
        x_k := x_16 ∸ x_k
end
```

Nelle ultime due macroistruzioni si suppone sempre $x_i > 0$.

Altre macroistruzioni di assegnamento calcolano le seguenti funzioni che di fatto manipolano le parti sinistra e destra di un intero positivo. Innanzitutto definiamo la funzione zero ponendo, per ogni $n \in \mathbb{N}_+$,

- $\mathrm{ze}(n) = \langle\langle \underbrace{0, 0, \ldots, 0}_{n \text{ volte}} \rangle\rangle = \langle \underbrace{0, \langle 0, \cdots, \langle 0}_{n \text{ volte}}, 0\rangle \cdots \rangle \rangle$

Poi, sempre per un valore $n \in \mathbb{N}_+$ qualsiasi, consideriamo gli interi $a_1, \ldots, a_t \in \mathbb{N}$ tali che

$$n = \langle\langle a_1, a_2, \ldots, a_t \rangle\rangle = \langle a_1, \langle a_2, \cdots, \langle a_t, 0\rangle \cdots \rangle \rangle$$

Possiamo allora definire la funzione lunghezza ponendo

- $\mathrm{lunghezza}(n) = t$

Infine, per ogni intero k tale che $1 \leq k \leq t$, definiamo i seguenti valori:

- $\mathrm{Pro}(k, n) = a_k$,
- $\mathrm{Inc}(k, n) = \langle\langle a_1, \ldots, a_{k-1}, a_k + 1, a_{k+1}, \ldots a_t \rangle\rangle$,
- $\mathrm{Dec}(k, n) = \langle\langle a_1, \ldots, a_{k-1}, a_k \dot{-} 1, a_{k+1}, \ldots a_t \rangle\rangle$.

Per ciascuna delle precedenti funzioni è possibile definire una macroistruzione che calcola il relativo valore e lo attribuisce alla variabili x_k, dove $k \in \{0,1,\ldots,13\}$. Vincoli più specifici per le variabili coinvolte (x_i, x_j e x_k) sono descritti nelle singole definizioni che presentiamo nel seguito:

11. $x_k := \mathrm{ze}(x_i)$, per ogni $i, k \neq 14, 15, 16$, è data da

```
begin
    x16 := xi
    xk := 0
    while x16 ≠ 0 do  begin
                        xk := ⟨0, xk⟩
                        x16 := x16 ∸ 1
                      end
end
```

12. $x_k := \mathrm{lunghezza}(x_i)$, con $i, k \neq 14, \ldots, 18$, è data da

```
begin
    x18 := xi
    xk := 0
    while x18 ≠ 0 do  begin
                        xk := xk + 1
                        x17 := des(x18)
                        x18 := x17
                      end
end
```

13. $x_k := \mathrm{Pro}(x_i, x_j)$, con $i \neq j$ e $i, j, k, \neq 14, \ldots, 19$, è definita dal seguente programma nel quale si suppone $x_j > 0$ e $0 < x_i \leq \mathrm{lunghezza}(x_j)$:

```
begin
    x18 := xj
    x17 := xi
    while x17 ≠ 0 do
        begin
            xk := sin(x18)
            x19 := des(x18)
            x18 := x19
            x17 := x17 ∸ 1
```

```
            end
        end
```

14. $x_k := \mathrm{Inc}(x_i, x_j)$ può essere definita dalla seguente procedura nella quale i, j, k variano in $\{0,1,\ldots,13\}$ e possono coincidere in tutto o in parte. Si suppone anche $x_j > 0$ e $0 < x_i \le \text{lunghezza}(x_j)$. La procedura di fatto simula prima la macroistruzione $x_k := \mathrm{Pro}(x_i, x_j)$ conservando però nella variabile x_{20} le componenti di indice minore di x_i; per mantenere questi valori si utilizza la funzione coppia. Una volta incrementata la componente di indice x_i, la procedura inserisce nella soluzione il nuovo valore insieme alle quantità precedentemente memorizzate in x_{20}. La variabile x_{20} funziona qui come una pila tradizionale. Nota che nella procedura si utilizzano (direttamente o nelle varie macroistruzioni usate) tutte le variabili ausiliarie $x_{14}, \ldots, x_{20}$:

```
begin
    x18 := xj
    x19 := xi
    x20 := 0
    while x19 ≠ 0 do
        begin
            xk := sin(x18)
            x20 := ⟨xk, x20⟩
            x17 := des(x18)
            x18 := x17
            x19 := x19 ∸ 1
        end
    xk := xk + 1
    x18 := ⟨xk, x18⟩
    x17 := des(x20)
    x20 := x17
    while x20 ≠ 0 do
        begin
            xk := sin(x20)
            x18 := ⟨xk, x18⟩
            x17 := des(x20)
            x20 := x17
        end
```

```
        x_k := x_18
    end
```

15. La macroistruzione $x_k := \mathrm{Dec}(x_i, x_j)$ è definita nello stesso modo, sostituendo il comando di incremento di x_k, collocato subito dopo il primo ciclo while, con un analogo comando di decremento.

Altre macroistruzioni hanno lo scopo di definire tradizionali istruzioni di controllo che simulano costrutti di uso comune nella programmazione. I più semplici sono i comandi “if then else”. In questa sede illustriamo solo quelli utilizzati nelle prossime sezioni:

16. `if` $x_i = 0$ `then` C,
dove C è un comando e $i \neq 14, 15$, è definita dal programma

```
begin
    x_15 := 1 ∸ x_i
    while x_15 ≠ 0 do begin C  x_15 := 0 end
end
```

Nota che il programma lascia azzerati i valori di x_{14} e x_{15}. Inoltre in maniera simile si definisce la macroistruzione `if` $x_i = n$ `then` C, per qualsiasi $n \in \mathbb{N}$;

17. `if` $x_i = 0$ `then` C_1 `else` C_2,
dove C_1 e C_2 sono comandi e $i \neq 14, 15$, è definita dal programma

```
begin
    x_15 := 1 ∸ x_i
    x_14 := 1
    while x_15 ≠ 0 do begin C_1  x_15 := 0  x_14 := 0 end
    while x_14 ≠ 0 do begin C_2  x_14 := 0 end
end
```

Nota che la procedura è corretta anche se x_{14} e x_{15} sono utilizzate in C_1 e C_2 perché, dopo il loro utilizzo, tali variabili rimangono azzerate;

18. `if` $x_i < x_j$ `then` C_1 `else` C_2,
dove C_1 e C_2 sono comandi e $i, j \neq 14, 15$, è definita dal programma

```
begin
      x15 := xj ∸ xi
      x14 := 1
      while x15 ≠ 0 do begin C1  x15 := 0  x14 := 0 end
      while x14 ≠ 0 do begin C2  x14 := 0 end
end
```

Anche in questo caso, per lo stesso motivo precedente, x_{14} e x_{15} possono essere tranquillamente usate nei comandi C_1 e C_2 senza alterare il risultato.

È abbastanza facile immaginare come simili programmi possano essere definiti per eseguire comandi più generali, della forma `if` D `then` C_1 `else` C_2, dove C_1 e C_2 sono comandi e D è una condizione (vera o falsa) dipendente dal valore delle variabili. Analoghe estensioni possono riguardare la definizione di comandi while generalizzati, del tipo `while` D `do` C, oppure macroistruzioni della forma `repeat` C `until` D, dal significato consueto [15].

Esercizio
Definire in maniera ragionevole le macroistruzioni $x_k := 2^{x_i}$, $x_k := x_i^{x_j}$ e $x_k := \lfloor \log_2 x_i \rfloor$.

1.6 Interprete

In questa sezione vogliamo definire un interprete del linguaggio RAM usando il linguaggio While. Più precisamente intendiamo definire un programma While in grado di eseguire qualunque programma RAM su qualsiasi input. Come conseguenza potremo provare che le funzioni RAM sono calcolabili anche dai programmi While e così, per i risultati della sezione 1.4, avremo dimostrato che i due sistemi calcolano la stessa famiglia di funzioni.

Il primo passo di questa costruzione consiste nella definizione di una codifica dei programmi RAM mediante interi non negativi. Formalmente vogliamo determinare una funzione

$$\mathrm{Cod} : \mathrm{Programmi} \to \mathbb{N}$$

che sia biunivoca e intuitivamente calcolabile (in un qualunque linguaggio di programmazione).

Innanzitutto, per ogni istruzione RAM I possiamo definire il valore $Ar(I) \in \mathbb{N}$ tale che

$$Ar(I) = \begin{cases} 3k & \text{se } I \equiv R_k \leftarrow R_k + 1 \\ 3k+1 & \text{se } I \equiv R_k \leftarrow R_k \dot{-} 1 \\ 3\langle k, n\rangle \dot{-} 1 & \text{se } I \equiv \texttt{if } R_k = 0 \texttt{ then go to } n \end{cases}$$

Allora, per ogni $P \in$ Programmi tale che $P = (I_1, I_2, \ldots, I_m)$, definiamo

$$\mathrm{Cod}(P) = \langle\langle \mathrm{Ar}(I_1), \mathrm{Ar}(I_2), \ldots, \mathrm{Ar}(I_m) \rangle\rangle - 1 \tag{1.6}$$

È facile verificare che la funzione Cod è biunivoca su $\mathbb{N}$ ed inoltre chiaramente calcolabile.

Proposizione 1.6.1. *La funzione Cod*: *Programmi* $\to \mathbb{N}$ *definita dalla equazione (1.6) è biunivoca. Inoltre Cod e la sua inversa* Cod^{-1} *sono calcolabili, ammettono cioè un algoritmo di calcolo in un qualunque linguaggio di programmazione.*

Se $j = \mathrm{Cod}(P)$ si dice che j è il numero di Gödel di P e lo stesso P viene anche chiamato programma di indice j o più semplicemente "programma j".

Nel seguito denotiamo con Ψ_j la funzione calcolata dal programma j, per ogni $j \in \mathbb{N}$. Questo significa che $\{\Psi_0, \Psi_1, \ldots, \Psi_j, \ldots\}$ rappresenta una numerazione delle funzioni calcolate dai programmi RAM, per cui possiamo porre

$$\mathit{Funzioni(RAM)} = \{\Psi_j \mid j \in \mathbb{N}\}$$

1.6.1 Programma universale

Usando la codifica precedente possiamo ora costruire un interprete del linguaggio RAM. Questo è definito da un programma While U che su input $\langle x, j\rangle$ calcola il valore $\Psi_j(x)$, di fatto eseguendo il programma RAM di indice j sull'input x. Nota che U non è un traduttore, non trasforma il programma in input in un altro linguaggio, ma piuttosto esegue la sua computazione sull'ingresso assegnato. Il calcolo viene compiuto usando le variabili del linguaggio while per mantenere lo stato corrente della macchina RAM, e decodificando una dopo l'altra le varie istruzioni RAM da eseguire.

La definizione di U è basata sui seguenti criteri:

- su input $n \in \mathbb{N}_+$ il programma U calcola per prima cosa i valori $x, j \in \mathbb{N}$ tali che $n = \langle x, j\rangle$;

- per ogni $j \in \mathbb{N}$ il programma P di indice j non può usare più di $j+1$ registri. Più precisamente, ogni computazione di P può modificare al più il contenuto dei registri $R_0, R_1, \ldots, R_j$. Di conseguenza gli stati raggiunti possono essere descritti dal contenuto di questi registri, oltre che dal valore del contatore L;

- per eseguire la computazione sull'input $n = \langle x, j \rangle$, il programma U mantiene nelle variabili le seguenti informazioni:

  x_0 assume il valore $\langle\langle S(R_0), S(R_1), \ldots, S(R_j) \rangle\rangle$, dove S è lo stato corrente della macchina RAM. Quindi di fatto x_0 mantiene l'intera memoria usata dalla macchina;

  x_1 assume il valore $S(L)$;

  x_2 conserva l'indice j del programma $P = (I_1, I_2, \ldots I_m)$ da simulare;

  x_3 conserva inizialmente l'input x per il programma P e successivamente la lunghezza m di P;

  x_4 assume il valore $\mathrm{Ar}(I_{S(L)})$, cioè il codice dell'istruzione di P da eseguire (istruzione corrente).

Come sappiamo l'input n di U è inizialmente attribuito alla variabile x_1, mentre l'output del programma sarà il valore di x_0 al termine della computazione. Quindi, tenendo conto della funzione codifica descritta sopra e delle macroistruzioni introdotte nella sezione 1.5, una semplice versione del programma U è definita nella figura 1.1.

Abbiamo quindi provato il seguente risultato.

Proposizione 1.6.2. *Esiste un programma While U tale che, per ogni $x, j \in \mathbb{N}$,*

$$\Phi_U(\langle x, j \rangle) = \Psi_j(x)$$

Inoltre, possiamo considerare il programma RAM $\mathrm{Comp}(U)$, e quindi otteniamo

$$\Psi_{\mathrm{Comp}(U)}(\langle x, j \rangle) = \Psi_j(x) \qquad \forall\ x, j \in \mathbb{N}$$

Di conseguenza anche nel linguaggio RAM esiste una procedura universale in grado di simulare tutti i suoi programmi su un qualsiasi input.

Corollario 1.6.3. *Esiste $u \in \mathbb{N}$ tale che, per ogni $x, j \in \mathbb{N}$,*

$$\Psi_u(\langle x, j \rangle) = \Psi_j(x)$$

```
begin
    x2 := des(x1)
    x3 := sin(x1)
    x0 := ze(x2)
    x0 := ⟨0, ⟨x3, x0⟩⟩
    x1 := 1
    x2 := x2 + 1
    x3 := lunghezza(x2)
    while x1 ≠ 0 do
        if x3 < x1
          then x1 := 0
          else begin
                  x4 := Pro(x1, x2)
                  x5 := x4(mod 3)
                                  { x4 := x4 : 3
                  if x5 = 0 then  { x0 := Inc(x4, x0)
                                  { x1 := x1 + 1
                                  { x4 := x4 : 3
                  if x5 = 1 then  { x0 := Dec(x4, x0)
                                  { x1 := x1 + 1
                                  { x4 := x4 + 1
                                  { x4 := x4 : 3
                                  { x5 := sin(x4)
                  if x5 = 2 then  { x6 := des(x4)
                                  { x7 := Pro(x5, x0)
                                  { if x7 = 0 then x1 := x6
                                  {               else x1 := x1 + 1
               end
    x5 := sin(x0)
    x0 := x5
  end
```

Figura 1.1: *Programma universale*

Altre conseguenze riguardano il confronto tra le funzioni calcolate dai due linguaggi.

Proposizione 1.6.4. *Per ogni $P \in$ Programmi esiste $Q \in W$-Programmi tale che $\Phi_Q = \Psi_P$.*

Dimostrazione. Se P possiede indice j allora il programma Q è definito da

```
begin
    x_0 := j
    x_1 := ⟨x_1, x_0⟩
    x_0 := 0
    U
end
```

dove U è il programma While definito nella figura 1.1. □

Applicando la proposizione precedente e la 2) del teorema 1.4.3, otteniamo il seguente risultato:

Corollario 1.6.5. *Funzioni*$(While)$ = *Funzioni*(RAM)

Abbiamo così provato che i due linguaggi calcolano la stessa famiglia di funzioni.

Osserva ora che il programma U definito nella tabella 1.1 usa in totale 15 variabili, 8 esplicitamente $(x_0, \ldots, x_7)$ e altre 7 variabili $(x_{14}, \ldots, x_{20})$ nelle macroistruzioni. Rimangono libere le 6 variabili $x_8, \ldots, x_{13}$. Per la proposizione precedente questo implica che ogni funzione while è calcolabile da un programma con la medesima proprietà.

Proposizione 1.6.6. *Per ogni programma While P esiste un programma While Q che non utilizza le variabili $x_8, \ldots, x_{13}$ e tale che*

$$\Phi_Q = \Phi_P$$

1.7 Funzioni ricorsive parziali

Introduciamo ora una nuova famiglia di funzioni definita senza fare riferimento diretto a costrutti di programmazione o modelli di calcolo, ma usando piuttosto un insieme di funzioni di base e alcuni operatori funzionali.

Poiché tratteremo funzioni a più argomenti, nel nostro contesto risulta conveniente usare la seguente notazione: per ogni $x \in \mathbb{N}^n$ e ogni $y \in \mathbb{N}^m$ (con $n, m \in \mathbb{N}_+$ qualsiasi), dove $x = (x_1, \dots, x_n)$ e $y = (y_1, \dots, y_m)$, denoteremo con (x, y) il vettore $(x_1, \dots, x_n, y_1, \dots, y_m) \in \mathbb{N}^{n+m}$.

L'insieme delle *funzioni base* è costituita dalla famiglia delle funzioni $\text{suc}, O, \text{Pro}_k^n$, dove $n, k \in \mathbb{N}_+$ e $1 \le k \le n$, tali che

- $\text{suc}(x) = x + 1$, per ogni $x \in \mathbb{N}$,
- $O(x) = 0$, per ogni $x \in \mathbb{N}$,
- $\text{Pro}_k^n(x_1, x_2, \dots, x_n) = x_k$, per ogni $(x_1, x_2, \dots, x_n) \in \mathbb{N}^n$.

Gli operatori che consideriamo sono quelli di *composizione*, *ricorsione primitiva* e *minimalizzazione*, definiti rispettivamente nel modo seguente.

- Date le funzioni $h : \mathbb{N}^m \to \mathbb{N}$ e $g_1, \dots, g_m$, dove $g_i : \mathbb{N}^n \to \mathbb{N}$ per ogni i, denotiamo con $comp(h; g_1, \dots, g_m)$ la funzione $f : \mathbb{N}^n \to \mathbb{N}$ tale che, per ogni $x \in \mathbb{N}^n$,

$$f(x) = h(g_1(x), g_2(x), \dots, g_m(x))$$

- Date le funzioni $g : \mathbb{N}^n \to \mathbb{N}$ e $h : \mathbb{N}^{n+2} \to \mathbb{N}$, denotiamo con $Ricpr(h, g)$ la funzione $f : \mathbb{N}^{n+1} \to \mathbb{N}$ tale che, per ogni $j \in \mathbb{N}$ e ogni $x \in \mathbb{N}^n$,

$$f(j, x) = \begin{cases} g(x) & \text{se } j = 0 \\ h(f(j-1, x), j-1, x) & \text{altrimenti} \end{cases}$$

- Data una funzione $g : \mathbb{N}^{n+1} \to \mathbb{N}$, denotiamo con $\mu y\{g(y, x_1, \dots, x_n) = 0\}$ o più semplicemente con $\mu\{g\}$, la funzione $f : \mathbb{N}^n \to \mathbb{N} \cup \{\perp\}$ tale che, per ogni $x \in \mathbb{N}^n$,

$$f(x) = \begin{cases} y & \text{se } g(y, x) = 0 \text{ e, per ogni } z \in \{0, 1, \dots, y-1\}, \\ & 0 \neq g(z, x) \neq \perp \\ \perp & \text{altrimenti} \end{cases}$$

Definizione 1.7.1. *La famiglia delle funzioni* ricorsive parziali *(o semplicemente* ricorsive*) è il più piccolo insieme contenente le funzioni base e chiuso rispetto alla composizione, alla ricorsione primitiva e alla minimalizzazione.*

Definizione 1.7.2. *La famiglia delle funzioni* ricorsive primitive *è il più piccolo insieme contenente le funzioni base e chiuso rispetto alla composizione e alla ricorsione primitiva.*

Nota che tutte le funzioni ricorsive primitive sono totali mentre la minimalizzazione è l'unico operatore tra i precedenti che definisce funzioni non totali. Chiaramente tutte le funzioni da $\mathbb{N}^n$ a $\mathbb{N}$ a valori costanti sono ricorsive primitive. Per esempio, per ogni intero $n > 1$, la funzione $O^n(x_1,\ldots,x_n) = 0$ soddisfa $O^n = comp(O;\mathrm{Pro}_1^n)$ e quindi è ricorsiva primitiva.

Esempi 1.7.1. È facile verificare che la funzione somma, definita da

$$\mathrm{somma}(x, y) = x + y, \quad \forall x, y, \in \mathbb{N}$$

è ricorsiva primitiva. Infatti abbiamo che

$$\mathrm{somma}(x, y) = \begin{cases} x & \text{se } y = 0 \\ \mathrm{suc}(\mathrm{somma}(x, y-1)) & \text{altrimenti} \end{cases} \qquad \forall x, y \in \mathbb{N}$$

e quindi formalmente somma $= Ricpr(comp(\mathrm{suc};\mathrm{Pro}_1^3),\mathrm{Pro}_1^1)$, e questo prova che la somma è ricorsiva primitiva.
In maniera simile si prova che le funzioni

$\mathrm{prodotto}(x, y) = x\cdot y$, $\mathrm{pred}(x) = x \dot{-} 1$, $\mathrm{diff}(x, y) = x \dot{-} y$, $\mathrm{potenza}(x, y) = x^y$, $\mathrm{modulodiff}(x, y) = |x-y|$, $\mathrm{sgn}(x) = \begin{cases} 0 & \text{se } x = 0 \\ 1 & \text{se } x > 0 \end{cases}$, $\overline{\mathrm{sgn}}(x) = \begin{cases} 1 & \text{se } x = 0 \\ 0 & \text{se } x > 0 \end{cases}$

sono ricorsive primitive.

Esempio 1.7.1. Un'altra famiglia di esempi si ottiene considerando le relazioni (o analogamente i predicati) sui numeri interi. Chiamiamo relazione ricorsiva primitiva un sottoinsieme $R \subseteq \mathbb{N}^n$, per qualche $n \in \mathbb{N}_+$, la cui funzione caratteristica χ_R è ricorsiva primitiva. Valgono le seguenti proprietà :

1. Le relazioni $=, \neq, >, <, \geq, \leq$ definite sulle coppie di interi sono ricorsive primitive. Ad esempio $\chi_=(x, y) = \overline{\mathrm{sgn}}(|x - y|)$ e $\chi_>(x, y) = \mathrm{sgn}(x \dot{-} y)$.

2. Se R e Q sono relazioni ricorsive primitive, definite sullo stesso spazio $\mathbb{N}^n$, allora anche $R \cap Q$, $R \cup Q$ e R^c sono ricorsive primitive.

3. Se $R \subseteq \mathbb{N}^n$ è una relazione ricorsiva primitiva, $r : \mathbb{N}^n \to \mathbb{N}$ e $s : \mathbb{N}^n \to \mathbb{N}$ sono funzioni ricorsive primitive, allora anche la funzione $f : \mathbb{N}^n \to \mathbb{N}$ tale che
$$f(x) = \begin{cases} r(x) & \text{se } x \in R \\ s(x) & \text{altrimenti} \end{cases}, \qquad \forall x \in \mathbb{N}^n$$
è ricorsiva primitiva. Infatti è facile verificare che $f(x) = r(x) \cdot \chi_R(x) + s(x) \cdot (1 - \chi_R(x))$.

Nel seguito useremo funzioni ricorsive primitive definite mediante il seguente operatore di minimalizzazione limitata, ottenuto modificando la definizione dell'operatore μ introdotto sopra.

Definizione 1.7.3. *Data una funzione $g : \mathbb{N}^{n+1} \to \mathbb{N}$ considera la $f : \mathbb{N}^{n+1} \to \mathbb{N}$ definita da*

$$f(y,x) = \begin{cases} \min\{k \in \mathbb{N} \mid k \le y, g(k,x) = 0\} & \textit{se } \{k \in \mathbb{N} \mid k \le y, g(k,x) = 0\} \neq \emptyset \\ 0 & \textit{altrimenti} \end{cases}$$

per ogni $y \in \mathbb{N}$ e $x \in \mathbb{N}^n$. Diciamo allora che f è definita per minimalizzazione limitata *di g e scriviamo $f(y,x) = \mu k \le y\{g(k,x) = 0\}$ o più semplicemente $f = \mu_\le\{g\}$.*

Teorema 1.7.1. *Ogni funzione definita per minimalizzazione limitata di una funzione ricorsiva primitiva è ricorsiva primitiva.*

Dimostrazione. Sia $f = \mu_\le\{g\}$ con g ricorsiva primitiva. Osserva che per ogni $y \in \mathbb{N}_+$ e ogni $x \in \mathbb{N}^n$, le seguenti implicazioni determinano il valore $f(y,x)$ in funzione di $f(y-1,x)$:

$$\begin{aligned} f(y-1,x) \neq 0 &\quad\Rightarrow\quad f(y,x) = f(y-1,x) \\ f(y-1,x) = 0 = g(0,x) &\quad\Rightarrow\quad f(y,x) = f(y-1,x) \\ f(y-1,x) = 0 \wedge g(0,x) \neq 0 \neq g(y,x) &\quad\Rightarrow\quad f(y,x) = f(y-1,x) \\ f(y-1,x) = 0 = g(y,x) \wedge g(0,x) \neq 0 &\quad\Rightarrow\quad f(y,x) = y \end{aligned}$$

Allora, per ogni $y \in \mathbb{N}$ e ogni $x \in \mathbb{N}^n$, si verifica che

$$f(y,x) = \begin{cases} 0 & \text{se } y = 0 \\ h(f(y-1,x), y-1, x) & \text{altrimenti} \end{cases}$$

dove

$$h(f(y-1,x),y-1,x) = \begin{cases} y & \text{se } g(0,x) \neq 0 \text{ e } f(y-1,x) = 0 = g(y,x) \\ f(y-1,x) & \text{altrimenti} \end{cases}$$

Ora osserva che per ogni $z, t \in \mathbb{N}$ e ogni $x \in \mathbb{N}^n$ abbiamo

$$h(z,t,x) = \begin{cases} \mathrm{suc}(t) & \text{se } g(0,x) \neq 0 \text{ e } z = 0 = g(\mathrm{suc}(t),x) \\ z & \text{altrimenti} \end{cases}$$

Quindi, per il punto 3. dell'esempio 1.7.1, h è ricorsiva primitiva ed essendo $f = Ricpr(h, O^n)$ anche f risulta ricorsiva primitiva. □

Esempi 1.7.2. Anche la funzione quoziente, definita da

$$\mathrm{quoziente}(x,y) = \begin{cases} \lfloor x/y \rfloor & \text{se } y \neq 0 \\ 0 & \text{altrimenti} \end{cases}$$

è ricorsiva primitiva. Infatti, si verifica che

$$\mathrm{quoziente}(x,y) = \mu k \leq x\{(x \dot{-} ky) \dot{-} (y \dot{-} 1) = 0\}$$

Così, definendo la funzione ausiliaria

$$\ell(x,y,u) = \mu k \leq x\{(u \dot{-} ky) \dot{-} (y \dot{-} 1) = 0\}$$

per la proposizione precedente abbiamo che $\ell(x,y,u)$ è ricorsiva primitiva. Di conseguenza, anche $\mathrm{quoziente}(x,y) = \ell(x,y,x)$ è ricorsiva primitiva.

Usando la minimalizzazione limitata si dimostra che varie altre funzioni sono ricorsive primitive. Ad esempio

$$\mathrm{resto}(x,y) = \begin{cases} 0 & \text{se } y = 0 \\ x (\mathrm{mod}\ y) & \text{altrimenti} \end{cases}$$

$$\mathrm{primo}(x) = \begin{cases} 1 & \text{se } x \text{ è primo} \\ 0 & \text{altrimenti} \end{cases}$$

Concludiamo questa sezione ricordando due proprietà notevoli delle funzioni ricorsive primitive, la cui dimostrazione può essere trovata in [15].

- Esistono funzioni ricorsive totali che non sono ricorsive primitive. Tra queste citiamo la nota funzione di Ackermann [15].

- La famiglia delle funzioni ricorsive primitive in un argomento coincide con l'insieme delle funzioni calcolate dai programmi *Loop*. Questi ultimi sono definiti come i programmi While, sostituendo però il tradizionale comando while con il comando

  $$\texttt{loop}\ x_k\ \texttt{do}\ C$$

  dove $k \in \{0, 1, \ldots, 20\}$ e C è un comando nel quale non si assegnano valori alla variabile x_k. Tale comando esegue C un numero di volte n fissato, pari al valore iniziale di x_k, ed equivale a

  $$\texttt{while}\ x_k \neq 0\ \texttt{do begin}\ \ C\ \ x_k := x_k \dot{-} 1\ \texttt{end}$$

1.8 Linguaggio While e funzioni ricorsive

In questa sezione vogliamo dimostrare che la famiglia delle funzioni ricorsive parziali coincide con l'insieme delle funzioni calcolate dai programmi While. A questo scopo dobbiamo innanzitutto stabilire cosa intendiamo per funzione a $n > 1$ argomenti calcolata da un programma While.

Cominciamo ricordando la definizione della funzione $[\cdots] : \bigcup_{n\geq 1} \mathbb{N}^n \to \mathbb{N}$, data da

$$\begin{aligned}
[x] &= x, \quad \forall x \in \mathbb{N} \\
[x, y] &= \langle x, y\rangle - 1, \quad \forall x, y \in \mathbb{N} \\
[x, y, z] &= [x, [y, z]], \quad \forall x, y, z \in \mathbb{N} \\
[x_1, x_2, \ldots, x_n] &= [x_1, [x_2, \ldots, [x_{n-1}, x_n] \ldots]], \quad \forall n \in \mathbb{N}_+ \text{ e } \forall x_1, x_2, \ldots, x_n \in \mathbb{N}
\end{aligned}$$

È chiaro che per ogni $n \in \mathbb{N}_+$, $[x_1, x_2, \ldots, x_n]$ stabilisce una corrispondenza biunivoca tra $\mathbb{N}^n$ e $\mathbb{N}$. Possiamo così definire le funzioni inverse $\overline{\text{sin}}$ e $\overline{\text{des}}$, univocamente determinate dalla relazione

$$n = [\overline{\text{sin}}(n), \overline{\text{des}}(n)], \quad \text{per ogni } n \in \mathbb{N}.$$

Inoltre è evidente che queste funzioni sono calcolabili nel linguaggio While. Adattando opportunamente i programmi While 8, 9, 10 della sezione 1.5 è facile definire analoghe macroistruzioni della forma

$$\begin{aligned}
x_k &:= [x_i, x_j], \quad \text{con } i, j, k \neq 14, 15 \\
x_k &:= \overline{\text{sin}}(x_i), \quad \text{con } i \neq k \text{ e } i, k \neq 14, 15, 16 \\
x_k &:= \overline{\text{des}}(x_i), \quad \text{con } i \neq k \text{ e } i, k \neq 14, 15, 16
\end{aligned}$$

Definizione 1.8.1. *Diciamo che una funzione $f : \mathbb{N}^n \to \mathbb{N} \cup \{\bot\}$ è calcolata da un programma While P se*

$$f(x_1, x_2, \dots, x_n) = \Phi_P([x_1, x_2, \dots, x_n]), \qquad \forall x_1, x_2, \dots, x_n \in \mathbb{N}$$

In questo modo un programma While P calcola una funzione per ogni possibile arietà $n \in \mathbb{N}_+$. Nel seguito denoteremo con $\Phi_P^{(n)}$ la funzione $\Phi_P([x_1, x_2, \dots, x_n])$, per ogni $P \in W$-Programmi.

Teorema 1.8.1. *Per ogni funzione ricorsiva parziale $f : \mathbb{N}^n \to \mathbb{N} \cup \{\bot\}$ esiste un programma While P che calcola f.*

Dimostrazione. Si ragiona per induzione strutturata. Chiaramente le funzioni suc e O sono calcolate da comandi di assegnamento. Inoltre, per ogni $k, n \in \mathbb{N}$, $1 \le k \le n$, è facile verificare che anche Pro_k^n è calcolabile da un programma While, basta adattare opportunamente la macroistruzione corrispondente definita nella sezione 1.5.

Consideriamo ora una funzione $f : \mathbb{N}^n \to \mathbb{N} \cup \{\bot\}$, tale che $f = comp(h; g_1, \dots, g_m)$, dove $h, g_1, \dots, g_m$ sono ricorsive parziali. Per ipotesi di induzione, supponiamo che esistano programmi While H, G_1, ..., G_m che calcolano rispettivamente le funzioni $h, g_1, \dots, g_m$. Per la proposizione 1.6.6 possiamo supporre che tali programmi non utilizzino le variabili $x_8, \dots, x_{13}$. Allora, per calcolare la funzione f possiamo definire il seguente programma While P; quest'ultimo usa come macroistruzioni i programmi H, G_1, ..., G_m insieme alla macroistruzione che calcola la funzione $[\cdot, \cdot]$.

```
begin
    x9 := x1
    x0 := Φ_{G_m}(x1)
    x10 := x0
    x1 := x9
    x0 := Φ_{G_{m-1}}(x1)
    x10 := [x0, x10]
    ......
    x1 := x9
    x0 := Φ_{G_1}(x1)
    x10 := [x0, x10]
    x1 := x10
    x0 := Φ_H(x1)
end
```

Sia ora $f = Ricpr(h, g)$, con h e g calcolabili rispettivamente dai programmi While H e G, che supponiamo non utilizzino le variabili "libere" $x_8, \dots, x_{13}$. Allora possiamo definire il seguente programma While che utilizza macroistruzioni per le funzioni h, g, $[\cdot,\cdot]$, $\overline{\text{sin}}$ e $\overline{\text{des}}$ (oltre a un'ovvia estensione del comando while facilmente simulabile mediane l'uso di variabili "libere"):

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_9 := 0 \\
\quad x_{10} := \overline{\text{des}}(x_1) \\
\quad x_{11} := \overline{\text{sin}}(x_1) \\
\quad x_1 := x_{10} \\
\quad x_0 := \Phi_G(x_1) \\
\quad \texttt{while } x_9 \neq x_{11} \texttt{ do} \left\{ \begin{array}{l} \texttt{begin} \\ \quad x_1 := [x_0, x_9, x_{10}] \\ \quad x_0 := \Phi_H(x_1) \\ \quad x_9 := x_9 + 1 \\ \texttt{end} \end{array} \right. \\
\texttt{end}
\end{array}
$$

Chiaramente questo programma calcola la funzione f.

Supponiamo ora $f = \mu\{g\}$, con g calcolabile da un programma While G che non utilizza le variabili $x_8, \dots, x_{13}$. Allora un programma While che calcola f è definito dallo schema seguente.

$$
\begin{array}{l}
\texttt{begin} \\
\quad x_9 := 0 \\
\quad x_{10} := x_1 \\
\quad x_1 := [x_9, x_{10}] \\
\quad x_0 := \Phi_G(x_1) \\
\quad \texttt{while } x_0 \neq 0 \texttt{ do begin } x_9 := x_9 + 1 \;\; x_1 := [x_9, x_{10}] \;\; x_0 := \Phi_G(x_1) \texttt{ end} \\
\quad x_0 := x_9 \\
\quad \texttt{end}
\end{array}
$$

□

Vogliamo ora provare la proprietà inversa, ovvero che ogni funzione calcolabile da un programma While è ricorsiva parziale. A questo scopo dimostriamo alcuni lemmi preparatori.

Lemma 1.8.2. *Per ogni $n \in \mathbb{N}_+$ la funzione $g_n : \mathbb{N}^n \to \mathbb{N}$ tale che*

$$g_n(x_1, \dots, x_n) = [x_1, \dots, x_n], \qquad \forall x_1, \dots, x_n \in \mathbb{N}$$

è ricorsiva primitiva.

Dimostrazione. Ragioniamo per induzione su $n \in \mathbb{N}_+$. Nota che $g_1 = \mathrm{Pro}_1^1$, mentre per ogni $x, y, \in \mathbb{N}$

$$g_2(x,y) = [x,y] = y + \frac{(x+y)(x+y+1)}{2} \tag{1.7}$$

Di conseguenza g_1 e g_2 sono ricorsive primitive. Supponi ora che $n > 2$ e che g_{n-1} sia ricorsiva primitiva. Allora, per ogni $x_1, \ldots, x_n \in \mathbb{N}$, abbiamo

$$g_n(x_1,\ldots,x_n) = [x_1, g_{n-1}(x_2,\ldots,x_n)] = g_2(x_1, g_{n-1}(x_2,\ldots,x_n))$$

e quindi $g_n = comp(g_2; \mathrm{Pro}_1^n, comp(g_{n-1}; \mathrm{Pro}_2^n, \ldots, \mathrm{Pro}_n^n))$ provando che anche g_n è ricorsiva primitiva. □

Lemma 1.8.3. *Le funzioni $\overline{\sin}$ e $\overline{\mathrm{des}}$ sono ricorsive primitive.*

Dimostrazione. Per ogni $x, y \in \mathbb{N}$ sappiamo che $x = \overline{\sin}([x,y])$ e $y = \overline{\mathrm{des}}([x,y])$. Inoltre, posto $n = [x,y]$ e ricordando la (1.7), per calcolare x e y possiamo determinare il minimo $t \in \mathbb{N}$ tale che

$$n < \sum_{i=1}^{t} i$$

(per cui $t = x + y + 1$), computare $\ell = \sum_{i=1}^{t} i$ e ottenere $x = \ell - n - 1$ e $y = t - x - 1$.

Definiamo allora, per ogni $n \in \mathbb{N}$,

$$\bar{t}(n) = \min\left\{k \le n \mid n < \frac{k(k+1)}{2}\right\} = \mu k \le n \left\{n + 1 \dot{-} \frac{k(k+1)}{2} = 0\right\}$$

Nota che $\bar{t}(n)$ è ricorsiva primitiva perché $\bar{t}(n) = f(n,n)$ con

$$f(u,v) = \mu k \le u \left\{v + 1 \dot{-} \frac{k(k+1)}{2} = 0\right\}$$

ottenuta per minimalizzazione limitata (teorema 1.7.1). Di conseguenza, anche

$$\overline{\sin}(n) = \frac{\bar{t}(n)(\bar{t}(n)+1)}{2} - n - 1 \quad \text{e} \quad \overline{\mathrm{des}}(n) = \bar{t}(n) - \overline{\sin}(n) - 1$$

sono ricorsive primitive. □

Lemma 1.8.4. *Per ogni $k, n \in \mathbb{N}_+$ con $1 \leq k \leq n$, la funzione $\overline{Pro}_k^n : \mathbb{N} \to \mathbb{N}$ tale che*

$$\overline{Pro}_k^n([x_1, \ldots, x_n]) = x_k \ , \qquad \forall x_1, \ldots, x_n \in \mathbb{N}$$

è ricorsiva primitiva.

Dimostrazione. La proprietà è chiaramente vera se $n = k = 1$ poiché $\overline{\mathrm{Pro}}_1^1$ è una funzione base. Se $1 \leq k < n$ allora, per ogni $y \in \mathbb{N}$,

$$\overline{\mathrm{Pro}}_k^n(y) = \overline{\sin}\left(\underbrace{\overline{\mathrm{des}}(\overline{\mathrm{des}}(\cdots(\overline{\mathrm{des}}}_{k-1 \text{ volte}}(y))\cdots))\right)$$

e quindi la proprietà segue dal lemma precedente. Lo stesso occorre se $1 < k = n$:

$$\overline{\mathrm{Pro}}_n^n(y) = \underbrace{\overline{\mathrm{des}}(\overline{\mathrm{des}}(\cdots(\overline{\mathrm{des}}}_{n-1 \text{ volte}}(y))\cdots)) \ , \qquad \forall y \in \mathbb{N}$$

□

Definizione 1.8.2. *Per ogni comando while C, denotiamo con f_C la funzione $f_C : \mathbb{N} \to \mathbb{N} \cup \{\bot\}$ tale che, per ogni $x_0, x_1, \ldots, x_{20} \in \mathbb{N}^{21}$,*

$$f_C([x_0, x_1, \ldots, x_{20}]) = \begin{cases} [z_0, z_1, \ldots, z_{20}] & \textit{se } [[C]](x_0, x_1, \ldots, x_{20}) = (z_0, z_1, \ldots, z_{20}) \\ \bot & \textit{se } [[C]](x_0, x_1, \ldots, x_{20}) = \bot \end{cases}$$

Nota che per ogni programma While P, ogni $n \in \mathbb{N}_+$ e ogni $y_1, y_2, \ldots, y_n \in \mathbb{N}^n$,

$$\Phi_P([y_1, y_2, \ldots, y_n]) = \overline{\sin}\left(f_P([\underbrace{0, [y_1, y_2, \ldots, y_n], 0, \ldots, 0}_{21 \text{ componenti}}])\right)$$

Quindi per provare che $\Phi_P^{(n)}$ è ricorsiva parziale basta dimostrare che f_P è ricorsiva parziale.

Proposizione 1.8.5. *Per ogni comando while C la funzione f_C è ricorsiva parziale.*

Dimostrazione. Procediamo per induzione strutturata. Se C è un comando di assegnamento, f_C è composizione delle funzioni O, suc, pred, $\overline{\mathrm{Pro}}_k^n$ e $[\cdot, \cdot]$ che sono tutte ricorsive primitive.

Se $C \equiv \texttt{begin}\, C_1 C_2 \cdots C_m \texttt{end}$ e $f_{C_1}, f_{C_2}, \ldots, f_{C_m}$ sono ricorsive parziali allora

$$f_C([x_0, x_1, \ldots, x_{20}]) = f_{C_m}(f_{C_{m-1}} \cdots (f_{C_1}([x_0, x_1, \ldots, x_{20}])) \cdots)$$

e quindi anche f_C è ricorsiva parziale.
Ora, sia $C \equiv \texttt{while}\ x_k \neq 0\ \texttt{do}\ A$, e assumi f_A ricorsiva parziale. Possiamo considerare le funzioni

$$\begin{aligned} S(e,x) = f_A^e(x) &= \begin{cases} x & \text{se } e = 0 \\ f_A(S(e-1,x)) & \text{altrimenti} \end{cases} \qquad \forall e, x \in \mathbb{N} \\ t(x) &= \mu e \left\{ \overline{\mathrm{Pro}}^{21}_{1+k}(f_A^e(x)) = 0 \right\} \qquad \forall x \in \mathbb{N} \end{aligned} \tag{1.8}$$

Chiaramente $S = Ricpr(f_A, \overline{\mathrm{Pro}}^1_1)$ è ricorsiva parziale e lo stesso dicasi per

$$t = \mu\{comp(\overline{\mathrm{Pro}}^{21}_{1+k}; S)\}$$

Così, poiché $f_C(x) = S(t(x), x)$ per ogni $x \in \mathbb{N}$, anche f_C risulta ricorsiva parziale. □

Teorema 1.8.6. *Per ogni programma While P e ogni $n \in \mathbb{N}_+$ la funzione $\phi_P^{(n)}$ è ricorsiva parziale.*

Dimostrazione. Per ogni $n \in \mathbb{N}_+$ abbiamo definito la funzione $g_n : \mathbb{N}^n \to \mathbb{N}$ tale che

$$g_n(x_1, \ldots, x_n) = [x_1, \ldots, x_n]$$

e sappiamo per il lemma 1.8.2 che tutte le g_n sono ricorsive primitive. Allora, per ogni $x_1, \ldots, x_n \in \mathbb{N}$, abbiamo

$$\phi_P^{(n)}(x_1, \ldots, x_n) = \overline{\sin}(f_P(g_{21}(0, g_n(x_1, \ldots, x_n), 0 \ldots, 0)))$$

Quindi, per i lemmi precedenti e la proposizione 1.8.5, anche $\phi_P^{(n)}$ è ricorsiva parziale. □

1.8.1 Tesi di Church

I teoremi 1.8.1 e 1.8.6 implicano il risultato principale di questa sezione.

Corollario 1.8.7. *La classe delle funzioni ricorsive parziali coincide con la famiglia delle funzioni calcolate dai programmi While.*

Abbiamo così provato che i tre formalismi considerati finora (linguaggio RAM ridotto, linguaggio While e funzioni ricorsive parziali) sono equivalenti da un punto di vista computazionale, nel senso che definiscono la stessa classe di funzioni calcolate.

Lo stesso fenomeno è stato osservato per tutti i formalismi tradizionalmente usati per definire le nozioni di algoritmo e di funzione calcolabile. Questo vale sia per i linguaggi di programmazione generali usati abitualmente nell'attività di programmazione, sia per i modelli di calcolo astratti come le macchine di Turing che introdurremo nelle prossime lezioni e la vasta gamma di modelli di computazione tradizionali noti in letteratura [15].

Possiamo così enunciare il seguente risultato generale che pur non essendo dimostrabile in assoluto è tuttavia comunemente accettata in ambito scientifico.

> **Tesi di Church**
> La classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni ricorsive parziali.

1.9 Sistemi di programmazione accettabili

In questa sezione introduciamo una nozione generale di sistema di programmazione che riassume le proprietà principali dei linguaggi RAM (ridotto) e While presentate finora.

Osserviamo anzitutto che, come nel caso dei programmi While, anche per il linguaggio RAM possiamo estendere la nozione di funzione calcolata da un programma incrementando il numero di argomenti.

Per ogni programma RAM P e ogni $n \in \mathbb{N}_+$, definiamo la funzione $\Psi_P^{(n)} : \mathbb{N}^n \to \mathbb{N} \cup \{\perp\}$ mediante

$$\Psi_P^{(n)}(x_1, \ldots, x_n) = \Psi_P([x_1, \ldots, x_n]) \qquad \forall x_1, \ldots, x_n \in \mathbb{N}$$

Diremo che $\Psi_P^{(n)}$ è la funzione (a n argomenti) calcolata da P. Se $i = \mathrm{Cod}(P)$ denotiamo $\Psi_P^{(n)}$ anche mediante $\Psi_i^{(n)}$. Nel seguito, quando il numero di argomenti è chiaro, ometteremo l'apice (n) in questa notazione; di conseguenza, per ogni $x_1, \ldots, x_n \in \mathbb{N}$ avremo che

$$\Psi_i(x_1, \ldots, x_n) = \Psi_P([x_1, \ldots, x_n])$$

Come sappiamo il linguaggio RAM è dotato di un programma universale, ovvero di un programma in grado di simulare tutti gli altri (Corollario 1.6.3). Questa proprietà può essere riformulata usando formalmente la funzione $[\cdot,\cdot]$ invece di $\langle\cdot,\cdot\rangle$:

> Esiste un indice $u \in \mathbb{N}$ tale che $\Psi_u(x,i) = \Psi_i(x)$, per ogni $i, x \in \mathbb{N}$.

Non è difficile dimostrare che questa proprietà può essere estesa anche ai programmi while, introducendo una opportuna enumerazione di W-Programmi simile a quella definita per il linguaggio RAM (Cod).

Un'altra proprietà rilevante del linguaggio RAM consiste nella possibilità di scambiare automaticamente argomenti e indici dei programmi. Più precisamente, possiamo enunciare il seguente risultato chiamato Teorema S_1^1:

Teorema 1.9.1. *Esiste una funzione ricorsiva totale $S_1^1 : \mathbb{N}^2 \to \mathbb{N}$ tale che, per ogni $x, y, i \in \mathbb{N}$,*

$$\Psi_{S_1^1(i,y)}(x) = \Psi_i(x,y)$$

Dimostrazione. Per definire la funzione S_1^1 consideriamo il programma RAM P di indice i e un qualunque $y \in \mathbb{N}$. Definiamo un nuovo programma RAM $U_{i,y}$ dato da

$$\underbrace{R_0 \leftarrow R_0 + 1; \ldots ; R_0 \leftarrow R_0 + 1}_{y \text{ volte}}; R_1 \leftarrow [R_1, R_0]; R_0 \leftarrow 0; P$$

Qui usiamo una macroistruzione RAM per la funzione $[\cdot,\cdot]$ (che chiaramente esiste per l'equivalenza con i programmi While) e P è aggiunto come spezzone finale del programma. È facile verificare che su input x, $U_{i,y}$ calcola il il valore $\Psi_P([x,y]) = \Psi_i(x,y)$. Definiamo allora $S_1^1(i,y) = \text{Cod}(U_{i,y})$. La funzione S_1^1 è ricorsiva perché si può definire un algoritmo che su input i, y determina l'indice di $U_{i,y}$, usando la funzione Cod introdotta nella sezione 1.6. $\square$

Intuitivamente questo significa non solo che la restrizione di una funzione calcolata da un dato programma è ancora calcolabile da un (altro) programma, ma anche che questo passaggio può essere determinato automaticamente mediante una funzione ricorsiva che trasforma indici e argomenti in nuovi indici.

Come nel caso precedente, anche questa proprietà può essere dimostrata per i programmi While. In realtà, si può dimostrare che questi risultati sono validi per tutti i linguaggi di programmazione usati tradizionalmente. Poiché vogliamo studiare proprietà comuni dei sistemi di programmazione, le proprietà precedenti ci portano a definire una nozione generale di sistema di calcolo.

Sia $\mathscr{R}^1$ la famiglia delle funzioni ricorsive parziali a un argomento:

$$\mathscr{R}^1 = \{f : \mathbb{N} \to \mathbb{N} \cup \{\bot\} \mid f \text{ ricorsiva parziale}\}$$

Definizione 1.9.1. *Un* sistema di programmazione accettabile *(nel seguito s.p.a. per brevità) è una successione di funzioni* $\{\phi_i\}_{i\in\mathbb{N}} = \{\phi_0, \phi_1, \ldots, \phi_i \ldots\}$, *dove* $\phi_i : \mathbb{N} \to \mathbb{N} \cup \{\bot\}$ *per ogni* $i \in \mathbb{N}$, *che soddisfa le seguenti proprietà:*

1. *La famiglia delle funzioni incluse in* $\{\phi_i\}_{i\in\mathbb{N}}$ *coincide con* $\mathscr{R}^1$. *In altre parole,* $\phi_i \in \mathscr{R}^1$ *per ogni* $i \in \mathbb{N}$ *e viceversa, per ogni* $f \in \mathscr{R}^1$, *esiste* $j \in \mathbb{N}$ *tale che* $f = \phi_j$;

2. *Esiste* $u \in \mathbb{N}$ *tale che per ogni* $x, i \in \mathbb{N}$,

$$\phi_u([x, i]) = \phi_i(x)$$

3. *Esiste una funzione ricorsiva totale* $S^1_1 : \mathbb{N}^2 \to \mathbb{N}$ *tale che, per ogni* $x, y, i \in \mathbb{N}$,

$$\phi_{S^1_1(i,y)}(x) = \phi_i([x, y])$$

In maniera più discorsiva possiamo dire che un s.p.a. è un'enumerazione delle funzioni ricorsive parziali in un argomento che le comprende tutte, ammette una funzione universale e soddisfa il teorema S^1_1. È chiaro che, per ogni s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$, possiamo considerare le corrispondenti funzioni con un numero di argomenti $n > 1$ qualsiasi, definendo $\phi^{(n)}_i(x_1, \ldots, x_n) = \phi_i([x_1, \ldots, x_n])$, per ogni $x_1, \ldots, x_n \in \mathbb{N}$. Nel seguito, per semplificare la notazione, quando il numero di argomenti è chiaro ometteremo l'esponente e le parentesi $[,]$ usando semplicemente $\phi_i(x_1, \ldots, x_n)$ per denotare $\phi_i([x_1, \ldots, x_n])$.

La nozione di s.p.a. appena definita è sufficiente per i nostri scopi ed è quella che verrà effettivamente usata nelle sezione successive. Ricordiamo, tuttavia, che in letteratura esistono diverse definizioni che si differenziano per vari aspetti. Quella più nota, considerata per esempio in [15, 17], di fatto estende la nostra e viene qui riportata, anche se non sarà usata nel seguito, perché può essere utilizzata per includere facilmente i tradizionali linguaggi di programmazione.

Definizione 1.9.2. *(versione estesa) Sia $\mathscr{R}^n$ l'insieme delle funzioni ricorsiva parziali da $\mathbb{N}^n$ a $\mathbb{N} \cup \{\bot\}$, per ogni $n \in \mathbb{N}_+$. Allora, un sistema di programmazione accettabile è una famiglia di successioni di funzioni*

$$\{\phi_i^{(1)}\}_{i \in \mathbb{N}}, \{\phi_i^{(2)}\}_{i \in \mathbb{N}}, \ldots, \{\phi_i^{(n)}\}_{i \in \mathbb{N}}, \ldots$$

dove $\phi_i^{(n)} : \mathbb{N}^n \to \mathbb{N} \cup \{\bot\}$ per ogni i e ogni n, che soddisfa le seguenti condizioni:

1. *Per ogni $n \in \mathbb{N}_+$, $\mathscr{R}^n$ coincide con la famiglia delle funzioni incluse in $\{\phi_i^{(n)}\}_{i \in \mathbb{N}}$;*
2. *Per ogni $n \in \mathbb{N}_+$, esiste un indice $u \in \mathbb{N}$ tale che*
$$\phi_i^{(n)}(x_1, \ldots, x_n) = \phi_u^{(n+1)}(x_1, \ldots, x_n, i), \qquad \forall i, x_1, \ldots, x_n \in \mathbb{N}$$
3. *Per ogni $n, m \in \mathbb{N}_+$ esiste una funzione ricorsiva totale $S_n^m : \mathbb{N}^{n+1} \to \mathbb{N}$ tale che*
$$\phi_i^{(n+m)}(x, y) = \phi_{S_n^m(i,y)}^{(m)}(x) \qquad \forall i \in \mathbb{N}, y \in \mathbb{N}^n, x \in \mathbb{N}^m$$

Si tratta di quindi di una famiglia di enumerazioni di funzioni ricorsive parziali, una enumerazione per ciascuna possibile arietà delle funzioni, che contiene una funzione universale per ciascuna arietà e che soddisfa il teorema S_n^m (condizione 3. data sopra).

1.9.1 Il teorema di ricorsione

Sono due le proprietà principali dei sistemi di programmazione accettabili: il teorema di isomorfismo e il teorema di ricorsione. Il primo sostanzialmente afferma che tra due s.p.a. esiste sempre una traduzione biunivoca dal primo al secondo. In questa sede enunciamo semplicemente questo teorema e rimandiamo la dimostrazione a [15].

Teorema 1.9.2. *(di isomorfismo) Dati due sistemi di programmazione accettabili $\{\alpha_i\}_{i \in \mathbb{N}}$, $\{\beta_i\}_{i \in \mathbb{N}}$, esiste una funzione ricorsiva totale biunivoca $T : \mathbb{N} \to \mathbb{N}$ tale che $\alpha_i = \beta_{T(i)}$ per ogni $i \in \mathbb{N}$.*

Osserviamo che l'esistenza di un funzione ricorsiva T tra i due s.p.a. che preserva la semantica dei programmi è una conseguenza abbastanza

semplice della stessa definizione di sistema di programmazione accettabile. Più impegnativo, invece, è costruire una funzione che, oltre a questo, sia anche biunivoca.

L'altro risultato rilevante è il teorema di ricorsione che può essere facilmente dimostrato a partire dalla definizione di s.p.a e ha svariate applicazioni, consentendo di provare una vasta gamma di proprietà degli s.p.a. (molte delle quali, a prima vista, insospettabili).

Teorema 1.9.3. *(di ricorsione) Sia $\{\phi_i\}_{i\in\mathbb{N}}$ un sistema di programmazione accettabile e sia $t : \mathbb{N} \to \mathbb{N}$ una funzione ricorsiva totale. Allora esiste $n \in \mathbb{N}$ tale che $\phi_n = \phi_{t(n)}$.*

Dimostrazione. Considera la funzione $f : \mathbb{N}^2 \to \mathbb{N} \cup \{\perp\}$ tale che

$$f(x,i) = \phi_{\phi_i(i)}(x) \qquad \forall i, x \in \mathbb{N}$$

Si verifica subito che f è ricorsiva parziale poiché $\phi_{\phi_i(i)}(x) = \phi_u(x, \phi_u(i,i))$ e quindi esiste certamente un programma che su input $[x,i]$ calcola $f(x,i)$ applicando due volte il programma universale ϕ_u. Esiste quindi un indice $e \in \mathbb{N}$ tale che $\phi_e(x,i) = f(x,i)$ per ogni $i, x \in \mathbb{N}$.

Applicando allora il teorema S_1^1 abbiamo che

$$\phi_{\phi_i(i)}(x) = \phi_{S_1^1(e,i)}(x) \qquad \forall i, x \in \mathbb{N} \tag{1.9}$$

Inoltre, poiché S_1^1 e t sono funzioni ricorsive totali, esiste $m \in \mathbb{N}$ tale che

$$\phi_m(i) = t(S_1^1(e,i)) \qquad \forall i \in \mathbb{N} \tag{1.10}$$

Ponendo quindi $n = S_1^1(e,m)$ si ottiene la seguente catena di identità

$$\begin{aligned} \phi_n &= \phi_{S_1^1(e,m)} = \phi_{\phi_m(m)} = \\ &= \phi_{t(S_1^1(e,m))} = \phi_{t(n)} \end{aligned}$$

dove la seconda e la terza uguaglianza sono conseguenza rispettivamente di (1.9) e (1.10). □

Il teorema di ricorsione è uno stumento utile per provare varie proprietà generali dei sistemi di programmazione accettabili, tra le quali tipicamente l'esistenza di programmi che calcolano funzioni particolari. Presentiamo qui di seguito alcuni esempi di applicazione.

Esempi 1.9.1.

1) Dato un s.p.a. $\{\phi_i\}$ e una funzione $f : \mathbb{N} \to \mathbb{N}$ ricorsiva, esistono infiniti $n \in \mathbb{N}$ tali che $\phi_n = \phi_{f(n)}$.

Dimostrazione. Dato un qualsiasi $k \in \mathbb{N}_+$, considera le funzioni $\phi_0, \phi_1, \dots, \phi_k$. Chiaramente esiste $\ell \in \mathbb{N}$ tale che $\phi_\ell \neq \phi_j$ per ogni $j = 0, \dots, k$. Definiamo allora la funzione $g_k : \mathbb{N} \to \mathbb{N}$ tale che

$$g_k(x) = \begin{cases} \ell & \text{se } x \leq k \\ f(x) & \text{altrimenti} \end{cases} \qquad \forall x \in \mathbb{N}$$

Poiché k è fissato, g_k è calcolabile e quindi ricorsiva totale. Per il teorema di ricorsione esiste $n \in \mathbb{N}$ tale che $\phi_n = \phi_{g_k(n)}$. Tuttavia per la definizione di g_k, n non può essere minore o uguale a k. Ne segue che $n > k$ e quindi $g_k(n) = f(n)$. Abbiamo così provato che per ogni $k \in \mathbb{N}$ esiste $n > k$ tale che $\phi_n = \phi_{f(n)}$ e questo significa che l'equazione precedente vale per infiniti n.
□

2) Per ogni coppia di s.p.a. $\{\alpha_i\}_{i\in\mathbb{N}}$, $\{\beta_i\}_{i\in\mathbb{N}}$ e ogni funzione ricorsiva totale $g : \mathbb{N} \to \mathbb{N}$ esiste $n \in \mathbb{N}$ tale che $\alpha_n = \beta_{g(n)}$ (intuitivamente: non è possibile costruire un traduttore completamente errato).

Dimostrazione. Per il teorema di isomorfismo esiste una funzione ricorsiva biunivoca $T : \mathbb{N} \to \mathbb{N}$ tale che $\alpha_i = \beta_{T(i)}$ per ogni $i \in \mathbb{N}$. Poiché T è biunivoca la funzione $h(x) = T^{-1}(g(x))$ esiste ed è ricorsiva. Per il teorema di ricorsione questo implica che esiste anche $n \in \mathbb{N}$ tale che $\alpha_n = \alpha_{h(n)} = \alpha_{T^{-1}(g(n))} = \beta_{g(n)}$. □

3) Per ogni s.p.a. $\{\phi_i\}$ esiste un indice $\ell \in \mathbb{N}$ tale che $\phi_\ell(x) = \ell$ per ogni $x \in \mathbb{N}$.

Dimostrazione. La funzione $g : \mathbb{N}^2 \to \mathbb{N}$ tale che $g(x, y) = y$ è certamente calcolabile. Quindi esiste $e \in \mathbb{N}$ tale che $\phi_e(x, y) = g(x, y)$ per ogni $x, y \in \mathbb{N}$. Di conseguenza, per il teorema S_1^1, abbiamo che $\phi_{S_1^1(e,y)}(x) = \phi_e(x, y) = g(x, y)$. Poiché la funzione $S_1^1(e, y)$ è ricorsiva totale, per il teorema di ricorsione esiste $\ell \in \mathbb{N}$ tale che $\phi_\ell = \phi_{S_1^1(e,\ell)}$ e quindi

$$\phi_\ell(x) = \phi_{S_1^1(e,\ell)}(x) = g(x, \ell) = \ell \qquad \forall x \in \mathbb{N}$$

□

4) Per ogni s.p.a. $\{\phi_i\}$ esiste un indice i tale che $\phi_i(x) = \phi_x(i)$ per ogni $x \in \mathbb{N}$.

Dimostrazione. Considera la funzione $g(x, y) = \phi_x(y)$. Poiché $g(x, y) = \phi_u(y, x)$, g è ricorsiva e quindi esiste $e \in \mathbb{N}$ tale che $\phi_e(x, y) = g(x, y)$. Applicando il teorema S_1^1 abbiamo che $\phi_{S_1^1(e,y)}(x) = \phi_e(x, y)$ per ogni $x, y, \in \mathbb{N}$.

Essendo $S^1_1(e,y)$ ricorsiva totale, per il teorema di ricorsione esiste $i \in \mathbb{N}$ tale che

$$\phi_i(x) = \phi_{s^1_1(e,i)}(x) = \phi_e(x,i) = g(x,i) = \phi_x(i) \qquad \forall x \in \mathbb{N}$$

□

1.10 Problemi indecidibili

Una delle prime proprietà presentate in queste note riguarda l'esistenza di funzioni, definite su $\mathbb{N}$ a valori in $\mathbb{N}$, che non sono calcolabili, per le quali cioè non esiste un algoritmo in grado di determinarne il valore per ogni possibile argomento.

Come è noto, infatti, l'insieme delle funzioni $\mathbb{N}^{\mathbb{N}} = \{f : \mathbb{N} \to \mathbb{N}\}$ non è numerabile, mentre in ogni sistema di programmazione accettabile (s.p.a. per brevità) l'insieme dei programmi è sempre numerabile. Ricordiamo, inoltre, che per la tesi di Church, l'insieme delle funzioni intuitivamente calcolabili coincide con la famiglia delle funzioni ricorsive parziali. In questa sezione vogliamo mostrare esempi classici di funzioni non calcolabili (cioè non ricorsive) e a questo scopo introduciamo i problemi indecidibili.

In generale, un problema di decisione è definito come una coppia $\mathscr{P} = (I,p)$, dove I è un insieme numerabile di elementi chiamate istanze, rappresentabile mediante numeri interi, mentre p è un predicato su I, ovvero una funzione $p : I \to \{0,1\}$. Tradizionalmente in letteratura esso viene rappresentato mettendo in evidenza le varie istanze e il corrispondente predicato nel modo seguente:

Problema $\mathscr{P}$
Istanza: $x \in I$.
Domanda: $p(x) = 1$?

Classici problemi di decisione sono quelli di appartenenza. Per esempio, dato un insieme $A \subseteq \mathbb{N}^k$ (con $k > 0$ intero), il problema di appartenenza ad A è definito dalla coppia $(\mathbb{N}^k, \chi_A)$ dove χ_A è la funzione caratteristica di A (ovvero, per ogni $x \in \mathbb{N}^k$, $\chi_A(x) = 1$ se $x \in A$ mentre $\chi_A(x) = 0$ altrimenti).

In questa sezione ci limitiamo a considerare problemi di decisione (I,p) in cui I coincide con l'insieme $\mathbb{N}^k$ delle k-ple di elementi in $\mathbb{N}$, per qualche k intero positivo. La prima proprietà fondamentale riguarda l'esistenza o meno di un algoritmo in grado di risolvere il problema per ogni possibile

istanza. Chiameremo decidibili o risolubili quei problemi che ammettono un tale algoritmo, mentre indecidibili saranno quelli che non dispongono di una tale procedura.

Definizione 1.10.1. *Un problema* decidibile *è un problema di decisione* (N^k, p) *per il quale la funzione p è ricorsiva.*

In altre parole, (N^k, p) è decidibile se esiste un algoritmo che su ogni input $x \in \mathbb{N}^k$ calcola $p(x)$. Per esempio, sono decidibili il problema di stabilire, data una tripla di interi $a, b, c \in \mathbb{N}$, se $c = a + b$ oppure il problema di verificare, sulla stessa istanza, se c è il massimo comun divisore di a e b.

Problema *Somma*
Istanza : $(a, b, c) \in \mathbb{N}^3$.
Domanda : $c = a + b$?

Problema *MCD*
Istanza : $(a, b, c) \in \mathbb{N}^3$.
Domanda : $c = \mathrm{MCD}(a, b)$?

In generale, per ogni funzione ricorsiva totale $f : \mathbb{N}^k \to \mathbb{N}$, il seguente problema *Calcola*(f) è decidibile:

Problema *Calcola*(f)
Istanza : $(x_1, \ldots, x_k, y) \in \mathbb{N}^{k+1}$.
Domanda : $y = f(x_1, \ldots, x_k)$?

Sono invece chiamati "indecidibili" quei problemi di decisione che non ammettono un algoritmo di soluzione.

Definizione 1.10.2. *Un problema* indecidibile *è un problema di decisione* (N^k, p) *nel quale la funzione p non è ricorsiva.*

Nel nostro contesto, quindi, un problema indecidibile è definito da una funzione (totale) non calcolabile a valori in $\{0, 1\}$. Esempi rilevanti di problemi indecidibili sono quelli legati all'arresto dei programmi su uno o più input.

Dato un qualunque sistema di programmazione accettabile $\{\phi_i\}_{i\in\mathbb{N}}$ si può dimostrare che i seguenti problemi sono indecidibili:

Problema *Arresto*$(\{\phi_i\}_{i\in\mathbb{N}})$
Istanza : $i \in I$.
Domanda : $\phi_i(i) \neq \perp$?

Problema *Totalità*$(\{\phi_i\}_{i\in\mathbb{N}})$
Istanza : $i \in I$.
Domanda : $\phi_i(j) \neq \perp$ per ogni $j \in \mathbb{N}$?

Il primo è il classico problema dell'arresto per il sistema $\{\phi_i\}_{i\in\mathbb{N}}$ e consiste nel verificare se il programma di indice i si ferma sull'input i. Il secondo è invece il problema della totalità per $\{\phi_i\}_{i\in\mathbb{N}}$ e richiede di verificare se il programma di indice i si ferma su tutti gli input.

Proposizione 1.10.1. *Per ogni sistema di programmazione accettabile* $\{\phi_i\}_{i\in\mathbb{N}}$ *il problema* Arresto($\{\phi_i\}_{i\in\mathbb{N}}$) *è indecidibile.*

Dimostrazione. Ragioniamo per assurdo e supponiamo che per un dato s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$ il problema dell'arresto sia decidibile. Allora sono ricorsive anche le due funzioni $f:\mathbb{N}\to\mathbb{N}$ e $g:\mathbb{N}\to\mathbb{N}\cup\{\bot\}$ tali che, per ogni $x\in\mathbb{N}$

$$f(x)=\begin{cases}1 & \text{se } \phi_x(x)\neq\bot\\ 0 & \text{altrimenti}\end{cases}\quad,\qquad g(x)=\begin{cases}1 & \text{se } f(x)=0\\ \bot & \text{se } f(x)=1\end{cases}$$

In particolare, esiste un programma ϕ_e che calcola g, e di conseguenza $g(x)=\phi_e(x)$ per ogni $x\in\mathbb{N}$. Valutiamo allora $g(e)$: per la stessa definizione di g otteniamo

$$g(e)=\phi_e(e)=\begin{cases}1 & \text{se } \phi_e(e)=\bot\\ \bot & \text{se } \phi_e(e)\neq\bot\end{cases}$$

Questo significa che

$$g(e)=\bot \text{ se e solo se } g(e)\neq\bot$$

il che è assurdo. □

Proposizione 1.10.2. *Per ogni sistema di programmazione accettabile* $\{\phi_i\}_{i\in\mathbb{N}}$ *il problema* Totalità($\{\phi_i\}_{i\in\mathbb{N}}$) *è indecidibile.*

Dimostrazione. Anche in questo caso ragioniamo per assurdo e applichiamo una tecnica di diagonalizzazione. Supponiamo che per qualche s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$ il problema della totalità sia decidibile. Allora la funzione $f:\mathbb{N}\to\mathbb{N}$ definita da

$$f(x)=\begin{cases}1 & \text{se } \phi_x \text{ è totale}\\ 0 & \text{altrimenti}\end{cases}$$

è ricorsiva. Lo stesso vale per la funzione $g:\mathbb{N}\to\mathbb{N}$ definita da

$$g(x)=\begin{cases}0 & \text{se } f(x)=0\\ 1+\phi_x(x) & \text{altrimenti}\end{cases}$$

Di conseguenza, esiste $e\in\mathbb{N}$ tale che $g=\phi_e$ e poiché g è totale, $g(e)\neq\bot$. Ne segue che, per la stessa definizione di g,

$$\phi_e(e)=g(e)=1+\phi_e(e)$$

e questo è assurdo. □

Un metodo naturale per dimostrare l'indecidibilità dei problemi è basato sulla nozione di riducibilità. Intuitivamente un problema di decisione è riducibile a un altro se il secondo può essere usato per risolvere il primo. Qui usiamo una nozione di riducibilità particolarmente semplice, definita mediante una funzione ricorsiva tra le istanze dei due problemi che conserva le risposte.

Definizione 1.10.3. *Dati due problemi di decisione $P = (\mathbb{N}^k, p)$, $Q = (\mathbb{N}^j, q)$, diciamo che P è* riducibile *a Q (in simboli $P \leq Q$) se esiste una funzione ricorsiva totale $f : \mathbb{N}^k \to \mathbb{N}^j$ (nel senso che tutte le sue componenti sono ricorsive totali) tale che, per ogni $x \in \mathbb{N}^k$, $p(x) = 1$ se e solo se $q(f(x)) = 1$. Diciamo anche che f è una* riduzione *da P a Q.*

È facile verificare che se P è riducibile a Q allora ogni algoritmo per risolvere Q consente di definire un algoritmo per risolvere P. Infatti, se $P = (\mathbb{N}^k, p)$, $Q = (\mathbb{N}^j, q)$, $P \leq Q$ mediante una riduzione f allora $p(x) = q(f(x))$ e quindi se q è ricorsivo anche p è ricorsivo. Di conseguenza, possiamo enunciare la seguente proprietà che permette di provare la dedicibilità o l'indecibilità dei problemi.

Proposizione 1.10.3. *Dati due problemi di decisione P, Q, supponi che $P \leq Q$. Allora se Q è decidibile anche P è decidibile. Di conseguenza, se P è indecidibile anche Q è indecidibile.*

Per fornire un esempio di riducibilità considera il problema di verificare, per un dato s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$, se il programma di indice i calcola la funzione identità. Formalmente il problema è definito dallo schema seguente:

Problema *Identità*$(\{\phi_i\}_{i\in\mathbb{N}})$
Istanza : $i \in I$.
Domanda : $\phi_i(j) = j$ per ogni $j \in \mathbb{N}$?

Proposizione 1.10.4. *Per ogni s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$ il problema* Totalità$(\{\phi_i\}_{i\in\mathbb{N}})$ *è riducibile al problema* Identità$(\{\phi_i\}_{i\in\mathbb{N}})$.

Dimostrazione. Per un qualsiasi s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$, considera la funzione

$$g(x,y) = \begin{cases} x & \text{se } \phi_y(x) \neq \bot \\ \bot & \text{altrimenti} \end{cases}$$

Nota che, per ogni $y \in \mathbb{N}$, ϕ_y è totale se e solo se $g(x,y) = x$ per ogni $x \in \mathbb{N}$. Inoltre, la funzione g è ricorsiva (parziale) poiché possiamo definire un

programma che su input (x, y) attiva il programma universale sullo stesso (x, y) e restituisce x al termine della computazione (se questa non termina il risultato è comunque $\perp$). Quindi esiste $e \in \mathbb{N}$ tale che $\phi_e(x, y) = g(x, y)$ per ogni $x, y \in \mathbb{N}$ e quindi, per il teorema S_1^1,

$$g(x, y) = \phi_e(x, y) = \phi_{S_1^1(e,y)}(x) \qquad (\forall\ x, y \in \mathbb{N})$$

Di conseguenza, per l'osservazione precedente

$$\phi_y \text{ è totale } \iff \phi_{S_1^1(e,y)}(x) = x \text{ per ogni } x \in \mathbb{N}$$

Questo implica la tesi poiché la funzione $S_1^1(e, y)$ è una funzione ricorsiva.
□

Ricordando che il problema della totalità è indecidibile, il risultato precedente prova la seguente proprietà.

Corollario 1.10.5. *Per ogni s.p.a.* $\{\phi_i\}_{i\in\mathbb{N}}$ *il problema* Identità$(\{\phi_i\}_{i\in\mathbb{N}})$ *è indecidibile.*

1.11 Insiemi ricorsivi

I concetti illustrati nella sezione precedente possono essere riformulati come proprietà di insiemi di numeri interi. Questo approccio può essere utile per semplificare la dimostrazione di varie proprietà.

Un insieme $A \subseteq \mathbb{N}^k$ è detto *ricorsivo* se la sua funzione caratteristica χ_A è ricorsiva. In altre parole, $A \subseteq \mathbb{N}^k$ è ricorsivo se esiste un programma (che termina sempre) per verificare se un elemento $x \in \mathbb{N}^k$ appartiene ad A.

Esempi di insiemi ricorsivi sono $\{n \in \mathbb{N} : n \text{ primo}\}$, $\{(n, n^2) : n \in \mathbb{N}\}$, $\{(a, b, c) \in \mathbb{N}^3 : c = a+b\}$ e, più in generale, $\{(x_1, \ldots, x_k, y) \in \mathbb{N}^{k+1} : y = f(x_1, \ldots, x_k)\}$ dove $f : \mathbb{N}^k \to \mathbb{N}$ è una funzione ricorsiva (totale).

Elenchiamo alcune proprietà elementari degli insiemi ricorsivi che si deducono facilmente dalla definizione:

1. gli insiemi $\emptyset$, $\mathbb{N}^k$ e tutti i sottoinsiemi finiti di $\mathbb{N}^k$ sono ricorsivi;

2. per ogni $A \subseteq \mathbb{N}^k$, A è ricorsivo se e solo se $A^c = \mathbb{N}^k \setminus A$ è ricorsivo;

3. se $A, B \subseteq \mathbb{N}^k$ sono ricorsivi allora anche $A \cup B$ e $A \cap B$ sono ricorsivi.

È evidente che un insieme $A \subseteq \mathbb{N}^k$ è ricorsivo se e solo se il problema di appartenenza ad A è decidibile. Per questo motivo le proprietà degli insiemi ricorsivi sono riconducibili alle proprietà dei problemi decidibili e molte nozioni introdotte per questi ultimi possono essere riformulate per i precedenti. Un esempio rilevante è quello della riducibilità tra problemi che può essere definita anche sugli insiemi.

Definizione 1.11.1. *Dati due insiemi $A \subseteq \mathbb{N}^k$ e $B \subseteq N^j$, diciamo che A è* riducibile *a B se esiste una funzione ricorsiva totale $f : \mathbb{N}^k \to \mathbb{N}^j$ tale che $f(A) \subseteq B$ e $f(A^c) \subseteq B^c$ (in altre parole, per ogni $x \in \mathbb{N}^k$, $x \in A$ se e solo se $f(x) \in B$).*

Così, dai risultati presentati nella sezione 1.10 possiamo dedurre che, per un qualunque sistema di programmazione accettabile $\{\phi_i\}$, i seguenti insiemi non sono ricorsivi:

$$K = \{i \in \mathbb{N} : \phi_i(i) \neq \bot\}, \quad K^c = \{i \in \mathbb{N} : \phi_i(i) = \bot\}, \quad T = \{i \in \mathbb{N} : \phi_i \text{ è totale}\}$$

Osserviamo, tuttavia, che i due insiemi K e T hanno una natura diversa, pur essendo entrambi non ricorsivi. Per K possiamo definire almeno una procedura parziale che tenta di risolvere il problema di appartenenza: per verificare se $i \in K$ possiamo attivare il programma universale ϕ_u su input (i, i) e, se la computazione si ferma, restituire una risposta positiva. La stessa cosa non è così evidente per l'insieme T: per verificare se $i \in T$ dovremmo attivare ϕ_u su tutti gli input (x, i) al variare di x in $\mathbb{N}$. Non disponiamo quindi nemmeno di una evidente procedura parziale, che risolve il problema quando la risposta è positiva. In qualche modo T sembra meno ricorsivo di K. Per cogliere formalmente questa differenza introduciamo una nuova famiglia di insiemi.

1.12 Insiemi ricorsivamente numerabili

Un insieme $A \subseteq \mathbb{N}^k$ è detto *ricorsivamente numerabile* (r.n. per brevità) se $A = \emptyset$ oppure esiste una funzione $f : \mathbb{N} \to \mathbb{N}^k$ ricorsiva totale (nel senso che tutte le sue componenti sono ricorsive totali) tale che $A = f(\mathbb{N})$, ovvero

$$A = \{f(0), f(1), \ldots, f(n), \ldots\}$$

Quindi un insieme non vuoto in $\mathbb{N}^k$ è ricorsivamente numerabile se esiste una procedura per elencare i suoi elementi.

Valgono le seguenti proprietà:

1. Ogni insieme ricorsivo è ricorsivamente numerabile.

   Infatti, se un insieme ricorsivo $A \subseteq \mathbb{N}^k$ non è vuoto allora esiste $a \in A$ e possiamo definire la funzione $f : \mathbb{N} \to \mathbb{N}^k$ tale che

   $$f(n) = \begin{cases} (x_1, x_2, \ldots, x_k) & \text{se } n = [x_1, x_2, \ldots, x_k] \text{ e } (x_1, x_2, \ldots, x_k) \in A \\ a & \text{altrimenti} \end{cases}$$

   Chiaramente f è ricorsiva e $A = f(\mathbb{N})$.

2. Esistono insiemi ricorsivamente numerabili che non sono ricorsivi.

   Per esempio considera l'insieme $H = \{i \in \mathbb{N} : \psi_i(i) \neq \perp\}$ dove $\{\psi_i\}_i$ è il sistema di programmazione RAM (ridotto). Sappiamo già che H non è ricorsivo. Inoltre, possiamo considerare un indice $e \in H$ e definire la funzione $f : \mathbb{N} \to \mathbb{N}$ tale che, per ogni $x \in \mathbb{N}$,

   $$f(x) = \begin{cases} i & \text{se } x = [i, n] \text{ e } \psi_i(i) \text{ termina eseguendo al più } n \text{ istruzioni RAM,} \\ e & \text{altrimenti.} \end{cases}$$

   Chiaramente $H = f(\mathbb{N})$ ed è facile verificare che f è ricorsiva.

3. Esistono insiemi che non sono ricorsivamente numerabili.

   Considera ad esempio l'insieme $T = \{i \in \mathbb{N} : \phi_i(x) \neq \perp\ \forall x \in \mathbb{N}\}$ dove $\{\phi_i\}_{i \in \mathbb{N}}$ è un qualunque s.p.a.. Proviamo per diagonalizzazione che T non è ricorsivamente numerabile. Infatti, supponi per assurdo che esista una $f : \mathbb{N} \to \mathbb{N}$ ricorsiva totale tale che $T = f(\mathbb{N})$. Allora possiamo considerare la funzione $g : \mathbb{N} \to \mathbb{N}$ tale che

   $$g(x) = \phi_{f(x)}(x) + 1 \qquad (\forall\ x \in \mathbb{N})$$

   Poiché $g(x) = 1 + \phi_u(x, f(x))$ anche g è ricorsiva totale e quindi $g = \phi_{f(e)}$ per qualche $e \in \mathbb{N}$. Di conseguenza otteniamo $\phi_{f(e)}(e) = g(e) = \phi_{f(e)}(e) + 1$ e questo è assurdo.

4. Se $A, B \subseteq \mathbb{N}^k$ sono ricorsivamente numerabili allora anche $A \cup B$ e $A \cap B$ sono ricorsivamente numerabili.

   Infatti, date le funzioni ricorsive che elencano gli elementi di A e B si possono facilmente costruire le funzioni (ricorsive) per elencare $A \cup B$ e $A \cap B$ (esercizio).

5. Per ogni $A \subseteq \mathbb{N}^k$, A è ricorsivo se e solo se sia A che il suo complementare $A^c = \mathbb{N}^k \setminus A$ sono ricorsivamente numerabili.

In un senso l'implicazione è ovvia. Nel senso opposto, supponi che $A = f(\mathbb{N})$ e $A^c = g(\mathbb{N})$ per due funzioni ricorsive totali f, g. Allora è possibile definire un programma per calcolare χ_A in un qualunque s.p.a.. Su input $x \in \mathbb{N}^k$ tale programma calcola iterativamente i valori $f(i)$ e $g(i)$ per $i = 0, 1, 2, \ldots$, e si ferma al primo i per il quale $f(i) = x$ oppure $g(i) = x$: nel primo caso restituisce 1, nel secondo 0. L'algoritmo termina sempre perché ogni x appartiene a uno (e uno solo) dei due insiemi.

Come gli insiemi ricorsivi, anche gli insiemi ricorsivamente numerabili sono chiusi rispetto alla relazione di riducibilità. Vale cioè la seguente proprietà.

Proposizione 1.12.1. *Supponi che $A \subseteq \mathbb{N}^k$, $B \subseteq \mathbb{N}^j$ e che $A \leq B$. Allora valgono le seguenti proprietà:*
- se B è r.n. anche A è r.n.;
- se A non è r.n. allora neppure B è r.n. .

Dimostrazione. Dimostriamo solo la prima proprietà poiché la seconda è una sua conseguenza immediata. Sia $f : \mathbb{N}^k \to \mathbb{N}^j$ una riduzione da A a B, ovvero una funzione ricorsiva (totale) tale che per ogni $x \in \mathbb{N}^k$, $x \in A$ se e solo se $f(x) \in B$. Supponi che $A \neq \emptyset$ e che $B = g(\mathbb{N})$ per una funzione ricorsiva $g : \mathbb{N} \to \mathbb{N}^j$. Allora possiamo considerare un elemento $a \in A$ e definire la funzione $h : \mathbb{N} \to \mathbb{N}^k$ tale che, per ogni $n \in \mathbb{N}$,

$$h(n) = \begin{cases} (x_1, x_2, \ldots, x_k) & \text{se } n = [x_1, \ldots, x_k, m] \text{ e} \\ & f(x_1, \ldots, x_k) \text{ appartiene a } \{g(0), \ldots, g(m)\} \\ a & \text{altrimenti} \end{cases}$$

Poiché f e g sono funzioni ricorsive, si deduce facilmente che anche h è ricorsiva. Inoltre è facile verificare che $h(\mathbb{N}) = A$. □

Come applicazione di questa proprietà si può dimostrare che per ogni s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$ l'insieme $ID = \{i \in \mathbb{N} : \phi_i(x) = x \ \forall x \in \mathbb{N}\}$ non è ricorsivamente numerabile. Infatti, usando la dimostrazione della Proposizione 1.10.4 è facile verificare che $T \leq ID$. Inoltre, per la precedente proprietà 3 sappiamo che T non è r.n. e quindi anche l'insieme ID non è ricorsivamente numerabile.

Infine, ricordiamo alcune proprietà caratterizzanti degli insiemi ricorsivamente numerabili. Esse possono essere utilizzate come definizione alternativa di questa nozione.

Proposizione 1.12.2. *Per ogni insieme $A \subseteq \mathbb{N}^k$ i seguenti enunciati sono equivalenti:*

a) *A è ricorsivamente numerabile;*

b) *esiste un insieme $B \subseteq \mathbb{N}^{k+1}$ ricorsivo tale che A è la proiezione di B, ovvero*

$$A = \{(x_1, \dots, x_k) \in \mathbb{N}^k : \exists y \in \mathbb{N} \text{ tale che } (x_1, \dots, x_k, y) \in B\}$$

c) *esiste una funzione $g : \mathbb{N}^k \to \mathbb{N} \cup \{\bot\}$ ricorsiva parziale tale che $A = dom(g)$, ovvero*

$$A = \{x \in \mathbb{N}^k : g(x) \neq \bot\}$$

d) *esiste una funzione $h : \mathbb{N} \to \mathbb{N}^k \cup \{\bot\}$ ricorsiva parziale (nel senso che ogni sua componente è ricorsiva parziale) tale che $A = cod(\mathbb{N})$, ovvero*

$$A = \{x \in \mathbb{N}^k : \exists y \in \mathbb{N} \text{ tale che } h(y) = x\}$$

Dimostrazione.

(***a*** $\Rightarrow$ ***b***) Se $A = \emptyset$ basta scegliere $B = \emptyset$. Altrimenti, supponendo $A = f(\mathbb{N})$ con f ricorsiva totale, considero

$$B = \{(x_1, \dots, x_k, y) \in \mathbb{N}^{k+1} : f(y) = (x_1, \dots, x_k)\}$$

È chiaro che A è proiezione di B. Inoltre B è ricorsivo perché χ_B può essere calcolata da un programma che su input $(x_1, \dots, x_k, y) \in \mathbb{N}^{k+1}$ determina $f(y) = (z_1, \dots, z_k)$ e poi verifica se $z_j = x_j$ per ogni $j = 1, \dots, k$. In caso affermativo il programma restituisce 1, altrimenti 0.

(***b*** $\Rightarrow$ ***c***) Data l'ipotesi ***b***, possiamo definire la funzione $g : \mathbb{N}^k \to \mathbb{N} \cup \{\bot\}$ tale che

$$g(x_1, \dots, x_k) = \begin{cases} 1 & \text{se } \exists y \in \mathbb{N} : (x_1, \dots, x_k, y) \in B \\ \bot & \text{altrimenti} \end{cases}$$

È chiaro che A è il dominio di g. Inoltre g è ricorsiva (parziale) perché possiamo descrivere facilmente un algoritmo che la calcola: su input $(x_1, \dots, x_k)$ tale algoritmo calcola $\chi_B(x_1, \dots, x_k, y)$ per $y = 0, 1, 2, \dots$, uno dopo l'altro, fermandosi al primo y tale che $\chi_B(x_1, \dots, x_k, y) = 1$ e restituendo 1 come output; chiaramente se $\chi_B(x_1, \dots, x_k, y) = 0$ per ogni $y \in \mathbb{N}$ la procedura non si ferma, ma in questo caso $g(x_1, \dots, x_k) = \bot$ e quindi l'algoritmo calcola proprio la funzione g.

($\boldsymbol{c} \Rightarrow \boldsymbol{d}$) Supponiamo che $A = \mathrm{dom}(g)$ con g ricorsiva parziale. Allora possiamo definire la funzione $f : \mathbb{N} \to \mathbb{N}^k \cup \{\perp\}$ tale che, per ogni $n \in \mathbb{N}$,

$$h(n) = \begin{cases} (x_1, \ldots, x_k) & \text{se } n = [x_1, \ldots, x_k] \text{ e } g(x_1, \ldots, x_k) \neq \perp \\ \perp & \text{altrimenti} \end{cases}$$

La funzione h è ricorsiva parziale perché esiste un programma P che su ogni input $n \in \mathbb{N}$ calcola $h(n)$. Tale programma determina i valori $x_1, \ldots, x_k \in \mathbb{N}$ tali che $n = [x_1, \ldots, x_k]$, e poi calcola il valore $g(x_1, \ldots, x_k)$. Se quest'ultimo calcolo termina allora $g(x_1, \ldots, x_k) \neq \perp$ e il programma P restituisce proprio la stessa k-pla $(x_1, \ldots, x_k)$; se invece non termina anche P non termina e infatti il valore di $h(n)$ è proprio $\perp$. Inoltre è facile verificare che $A = h(\mathbb{N})$.

($\boldsymbol{d} \Rightarrow \boldsymbol{a}$) Assumendo la proprietà $\boldsymbol{d}$, se $h(n) = \perp$ per ogni n allora $A = \emptyset$ e la $\boldsymbol{a}$ è soddisfatta. In caso contrario esiste una k-pla $\underline{a} \in A$ e inoltre possiamo considerare un programma RAM P che, su ogni input $i \in \mathbb{N}$, calcola $h(i)$. Definiamo allora la funzione $f : \mathbb{N} \to \mathbb{N}^k$ tale che, per ogni $n \in \mathbb{N}$,

$$f(n) = \begin{cases} (x_1, \ldots, x_k) & \text{se } n = [i, m] \text{ il programma } P \text{ su input } i \text{ termina entro } m \text{ passi,} \\ & \text{e restituisce } (x_1, \ldots, x_k) \\ \underline{a} & \text{altrimenti.} \end{cases}$$

Chiaramente f è ricorsiva e $f(\mathbb{N}) = A$. □

Esercizi

1. Assumendo che $\{\phi_i\}_{i \in \mathbb{N}}$ sia un qualunque s.p.a., verificare se i seguenti insiemi sono ricorsivamente numerabili:

$$A = \{i \in \mathbb{N} : \phi_i(2) = 3\}, \quad B = \{i \in \mathbb{N} : \phi_i(2) \neq \perp\},$$
$$C = \{i \in \mathbb{N} : \phi_i(2) = \perp\}, \quad D = \{i \in \mathbb{N} : \phi_2(i) \neq \perp\}$$

Cosa possiamo dire dei loro complementari?

2. Sia $f : \mathbb{N} \to \mathbb{N}$ una funzione ricorsiva totale e, per un qualunque s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$, considera l'insieme $C(f) = \{i \in \mathbb{N} : \phi_i = f\}$. Dimostrare che $T \leq C(f)$. L'insieme $C(f)$ è ricorsivamente numerabile? È ricorsivo?

3. Continuando l'esercizio precedente, sia $g : \mathbb{N} \to \mathbb{N} \cup \{\perp\}$ una funzione ricorsiva non totale. È ancora vero che $T \leq C(g)$?

1.13 Il teorema di Rice

Introduciamo ora uno strumento classico per provare che un insieme non è ricorsivo. Si tratta del noto teorema di Rice che intuitivamente afferma

che le proprietà semantiche dei programmi (cioè le proprietà delle funzioni calcolate) non possono essere verificate automaticamente. Di questo teorema esiste una versione anche per gli insiemi ricorsivamente numerabili che fornisce in maniera analoga uno strumento generale per dimostrare che un insieme non è r.n..

Ricordiamo innanzitutto che un insieme $A \subseteq \mathbb{N}$ *rispetta le funzioni* per un dato s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$ se, per ogni $i \in A$ e ogni $j \in \mathbb{N}$, $\phi_i = \phi_j$ implica $j \in A$.

Per esempio, i seguenti insiemi rispettano le funzioni:

$$\{i \in \mathbb{N} : \phi_i(x) \neq \bot \ \forall x \in \mathbb{N}\}, \quad \{i \in \mathbb{N} : \exists x \text{ tale che } \phi_i(x) \neq \bot\}, \tag{1.11}$$

$$\{i \in \mathbb{N} : \phi_i(1) = 2\}, \quad \{i \in \mathbb{N} : \phi_i \text{ è iniettiva}\} \tag{1.12}$$

È facile immaginare che non tutti gli insiemi rispettano le funzioni. Un esempio notevole è dato dalla seguente proposizione.

Proposizione 1.13.1. *Per ogni s.p.a.* $\{\phi_i\}_{i\in\mathbb{N}}$ *l'insieme* $K = \{i \in \mathbb{N} : \phi_i(i) \neq \bot\}$ *non rispetta le funzioni.*

Dimostrazione. Mostriamo innanzitutto che esiste un indice $t \in \mathbb{N}$ tale che

$$\phi_t(x) = \begin{cases} 1 & \text{se } t = x \\ \bot & \text{altrimenti} \end{cases} \tag{1.13}$$

Infatti, possiamo considerare la funzione $f : \mathbb{N}^2 \to \mathbb{N} \cup \{\bot\}$ tale che

$$f(x,i) = \begin{cases} 1 & \text{se } x = i \\ \bot & \text{altrimenti} \end{cases}$$

Chiaramente f è ricorsiva (parziale) e quindi esiste $e \in \mathbb{N}$ tale che $f = \phi_e$ e per il teorema S^1_1, per ogni $x, i \in \mathbb{N}$, abbiamo

$$\phi_e(x,i) = \phi_{S^1_1(e,i)}(x)$$

Poiché la funzione $S^1_1(e,x)$ nella variabile x è ricorsiva totale, applicando il teorema di ricorsione sappiamo che esiste un indice $t \in \mathbb{N}$ tale che

$$\phi_t(x) = \phi_{S^1_1(e,t)}(x) \qquad (\forall x \in \mathbb{N})$$

Allora, per ogni $x \in \mathbb{N}$ possiamo scrivere

$$\phi_t(x) = \phi_{S^1_1(e,t)}(x) = \phi_e(t,x) = f(x,t)$$

e questo prova la relazione (1.13).

Di conseguenza $t \in K$, ma per ogni $i \neq t$ tale che $\phi_i = \phi_t$ abbiamo $\phi_i(i) = \bot$ e quindi $i \notin K$. Questo prova che K non rispetta le funzioni. □

Si può dimostrare che gli unici insiemi ricorsivi che rispettano le funzioni sono $\emptyset$ e $\mathbb{N}$.

Teorema 1.13.2. *(di Rice) Sia $A \subseteq \mathbb{N}$ un insieme che rispetta le funzioni per una qualunque s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$. Allora A è ricorsivo se e solo se $A = \emptyset$ oppure $A = \mathbb{N}$.*

Dimostrazione. In un senso l'implicazione è ovvia perché $\emptyset$ e $\mathbb{N}$ sono ricorsivi. Dimostriamo viceversa che ogni insieme ricorsivo $A \subseteq \mathbb{N}$ che rispetta le funzioni per un qualunque s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$, coincide con $\emptyset$ oppure con $\mathbb{N}$. Per assurdo supponi infatti che ci sia un elemento $i \in A$ e un elemento $j \notin A$. Allora possiamo definire la funzione $f : \mathbb{N} \to \mathbb{N}$ tale che, per ogni $x \in \mathbb{N}$,

$$f(x) = \begin{cases} j & \text{se } x \in A \\ i & \text{altrimenti} \end{cases}$$

Nota che f è ricorsiva e che, per ogni $x \in \mathbb{N}$,

$$x \in A \iff f(x) \notin A \tag{1.14}$$

Ora, per il teorema di ricorsione sappiamo che esiste $t \in \mathbb{N}$ tale che $\phi_t = \phi_{f(t)}$ e, poiché A rispetta le funzioni, abbiamo che $t \in A$ se e solo se $f(t) \in A$, ma questo contraddice la relazione (1.14). □

Come conseguenza del teorema precedente è immediato verificare che gli insiemi definiti in (1.11) e (1.12) non sono ricorsivi.

Un analogo del teorema di Rice può essere formulato anche per gli insiemi multidimensionali.

Diciamo che un insieme $A \subseteq \mathbb{N}^k$ rispetta le funzioni per un dato s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$ se, per ogni $(i_1, \ldots, i_k) \in A$ e ogni $(j_1, \ldots, j_k) \in \mathbb{N}^k$ tale che $\phi_{i_t} = \phi_{j_t}$ per ogni $t = 1, \ldots, k$, anche $(j_1, \ldots, j_k)$ appartiene ad A.

Teorema 1.13.3. *Sia $A \subseteq \mathbb{N}^k$ un insieme che rispetta le funzioni per una qualunque s.p.a. $\{\phi_i\}_{i \in \mathbb{N}}$. Allora A è ricorsivo se e solo se $A = \emptyset$ oppure $A = \mathbb{N}^k$.*

Per la dimostrazione si può vedere [15].

Come conseguenza del teorema precedente è facile dedurre che i seguenti insiemi non sono ricorsivi:

$$\{(i, j) \in \mathbb{N}^2 : \phi_i = \phi_j\}, \quad \{(i, j, k) \in \mathbb{N}^3 : \phi_i(x) + \phi_j(x) = \phi_k(x)\ \forall x \in \mathbb{N}\}$$

Intuitivamente possiamo quindi affermare che quasi tutti gli insiemi che rispettano le funzioni non sono ricorsivi. È naturale ora chiedersi se la stessa proprietà vale per gli insiemi ricorsivamente numerabili. In generale, la risposta a questa domanda è negativa: per esempio, gli insiemi $\{i \in \mathbb{N} : \phi_i(0) = 1\}$ e $\{i \in \mathbb{N} : \phi_i(x) \neq \perp \ \forall x \in \mathbb{N}\}$ rispettano le funzioni, ma il primo è r.n., mentre il secondo no.

Il seguente teorema fornisce alcune condizioni affinché un insieme che rispetta le funzioni non sia ricorsivamente numerabile.

Teorema 1.13.4. *Sia $A \subseteq \mathbb{N}$ un insieme che rispetta le funzioni per una qualunque s.p.a. $\{\phi_i\}_{i\in\mathbb{N}}$; siano inoltre ξ e η due funzioni da $\mathbb{N}$ in $\mathbb{N}\cup\{\perp\}$ ricorsive (parziali) tali che:*

1) $\{i \in \mathbb{N} : \phi_i = \xi\} \subseteq A$,

2) $\{i \in \mathbb{N} : \phi_i = \eta\} \subseteq A^c$,

3) η è un'estensione di ξ, ovvero $\xi(x) = \eta(x)$ per ogni $x \in dom(\xi)$.

Allora A non è ricorsivamente numerabile.

Dimostrazione. Per dimostrare il teorema costruiamo una riduzione da $K^c = \{i \in \mathbb{N} : \phi_i(i) = \perp\}$ ad A. Poiché K^c non è r.n. questo implica che anche A non è ricorsivamente numerabile.

Considera la funzione $g : \mathbb{N}^2 \to \mathbb{N} \cup \{\perp\}$ definita da

$$g(i,j) = \begin{cases} \xi(j) & \text{se } \phi_i(i) = \perp \\ \eta(j) & \text{altrimenti} \end{cases} \qquad (\forall (i,j) \in \mathbb{N}^2)$$

Si dimostra che g è ricorsiva parziale. Infatti, consideriamo i tre programmi P, P_ξ e P_η che calcolano rispettivamente le funzioni $\phi_u(i,i)$, $\xi(j)$ e $\eta(j)$. Definiamo un nuovo programma P_g che su input (i,j) attiva in parallelo le procedure P e P_ξ rispettivamente su input i e j. Se P_ξ termina la computazione prima di P allora P_g restituisce il suo risultato e si ferma. Se invece termina prima P allora P_g arresta la computazione di P_ξ e attiva il programma P_η su input j e restituisce il suo (eventuale) risultato. Nota che se entrambe le computazioni di P e P_ξ non terminano anche P_g non termina e il valore $\perp$ è proprio quello di $g(i,j)$ in questo caso.

Quindi la funzione g è ricorsiva parziale, per cui esiste un indice $e \in \mathbb{N}$ tale che $\phi_e(j,i) = g(i,j)$. Per il teorema S_1^1 sappiamo che, per ogni $i, j \in \mathbb{N}$,

$$\phi_e(j,i) = \phi_{S_1^1(e,i)}(j) \qquad (= g(i,j))$$

Questo significa che per ogni $i \in \mathbb{N}$, $\phi_{S_1^1(e,i)} = \xi$ se $\phi_i(i) = \perp$, mentre $\phi_{S_1^1(e,i)} = \eta$ se $\phi_i(i) \neq \perp$. Per le proprietà 1) e 2) questo implica che

$$S_1^1(e,i) \in A \iff \phi_i(i) = \perp \iff i \in K^c$$

Poiché $S_1^1(e,i)$ è una funzione ricorsiva totale nella variabile i questo prova che $K^c \leq A$. □

Applicando il teorema precedente è facile verificare che i seguenti insiemi non sono ricorsivamente numerabili:

$$\{i \in \mathbb{N} : \phi_i(x) = \bot \ \forall x \in \mathbb{N}\}, \quad \{i \in \mathbb{N} : \text{ dom}(\phi_i) \text{ è finito}\}, \quad \{i \in \mathbb{N} : 2 \notin \phi_i(\mathbb{N})\}$$

Esercizio
Assumendo che $\{\phi_i\}_{i \in \mathbb{N}}$ sia un qualunque s.p.a., verificare se i seguenti insiemi sono ricorsivi o ricorsivamente numerabili:

$$A = \{i \in \mathbb{N} : a \in \phi_i(\mathbb{N})\}, \text{ con } a \in \mathbb{N} \text{ fissato}$$
$$B = \{i \in \mathbb{N} : a \notin \phi_i(\mathbb{N})\}, \text{ con } a \in \mathbb{N} \text{ fissato}$$
$$C = \{i \in \mathbb{N} : \exists a \in \mathbb{N}\ \phi_i(a) = \bot\},$$
$$D = \{i \in \mathbb{N} : \phi_i(x) = \bot \ \forall x \in \mathbb{N}\},$$
$$E = \{i \in \mathbb{N} : \exists a \in \mathbb{N}\ \phi_i(a) \neq \bot\},$$
$$F = \{i \in \mathbb{N} : \phi_i(x) = f(x) \ \forall x \in \mathbb{N}\}, \text{ dove } f \text{ è una funzione ricorsiva parziale non totale},$$
$$G = \{i \in \mathbb{N} : \phi_i \neq f\}, \text{ dove } f \text{ è una funzione ricorsiva parziale non identicamente } \bot,$$
$$H = \{i \in \mathbb{N} : \phi_i(x) = f(x) \ \forall x \in \mathbb{N}\}, \text{ dove } f \text{ è una funzione ricorsiva totale.}$$

1.14 Insiemi completi

Intuitivamente gli insiemi completi rappresentano la famiglia di insiemi più difficili da riconoscere tra tutti gli insiemi r.n.. Formalmente un insieme $B \subseteq \mathbb{N}$ si dice *completo* se B è r.n. e per ogni insieme $A \subseteq \mathbb{N}$ ricorsivamente numerabile vale $A \leq B$.

Proposizione 1.14.1. *Per ogni s.p.a.* $\{\phi_i\}_{i \in \mathbb{N}}$ *l'insieme* $K = \{i \in \mathbb{N} : \phi_i(i) \neq \bot\}$ *è completo.*

Dimostrazione. Poiché $K = \text{dom}(\phi_u(i,i))$ è chiaro che K è r.n.. Costruiamo allora una riduzione da un qualunque insieme ricorsivamente numerabile $A \subseteq \mathbb{N}^j$ all'insieme K. Sia $g : \mathbb{N} \to \mathbb{N}^j$ ricorsiva (totale) tale che $A = g(\mathbb{N})$. Definiamo la funzione $f : \mathbb{N}^{j+1} \to \mathbb{N} \cup \{\bot\}$ tale che, per ogni $x \in \mathbb{N}$ e ogni $y \in \mathbb{N}^j$,

$$f(x,y) = \begin{cases} 1 & \text{se } y \in A \\ \bot & \text{altrimenti} \end{cases}$$

Chiaramente $f(x,y) = 1$ se e solo se esiste $z \in \mathbb{N}$ tale che $g(z) = y$. Questo implica che f è ricorsiva (parziale) perché possiamo definire un programma che su input x, y calcola $g(z)$ per $z = 0,1,2\dots$ e si ferma al primo z tale

che $g(z) = y$, restituendo il valore 1. Allora esiste $e \in \mathbb{N}$ tale che $\phi_e = f$ e, per il teorema S^1_j possiamo scrivere

$$\phi_{S^1_j(e,y)}(x) = \phi_e(x, y) = f(x, y) \qquad (\forall x \in \mathbb{N}, y \in \mathbb{N}^j)$$

Ne segue che $\phi_{S^1_j(e,y)}$ è uguale alla funzione identicamente 1 se e solo se $y \in A$ e questo implica

$$S^1_j(e, y) \in K \iff \phi_{S^1_j(e,y)}\big(S^1_j(e, y)\big) \neq \perp \iff \phi_{S^1_j(e,y)}\big(S^1_j(e, y)\big) = 1 \iff y \in A \quad (\forall y \in \mathbb{N}^j)$$

cioè $A \le K$. □

Esercizio
Assumendo che $\{\phi_i\}_{i\in\mathbb{N}}$ sia un qualunque s.p.a., definiamo $A = \{i \in \mathbb{N} : \phi_i(2) = 3\}$. Verificare se l'insieme A è completo.

Capitolo 2

Complessità sequenziale

Questo capitolo è dedicato allo studio della complessità dei problemi decidibili, e più precisamente all'analisi e alla classificazione di questi problemi sulla base della quantità di risorse utilizzate dagli algoritmi che li risolvono. In questa sede assumiamo modelli di calcolo sequenziali, dotati cioè di una sola unità di computazione in grado di eseguire una operazione alla volta, per i quali le tradizionali risorse sono quelle del tempo di calcolo e dello spazio di memoria. Segnaliamo che è possibile svolgere un'analisi simile basata invece su modelli di calcolo parallelo, dotati cioè di unità in grado di svolgere più operazioni contemporaneamente. Un esempio tipico è quello delle famiglie di circuiti Booleani [19, 5, 2].

Il modello di calcolo principale che utilizziamo in questo capitolo è quello della macchina di Turing, che permette una facile e naturale definizione del tempo di calcolo e dello spazio di memoria. Anticipiamo fin da subito che le macchine di Turing formano effettivamente un Sistema di Programmazione Accettabile, secondo la definizione data nel capitolo precedente, e possono quindi rappresentare pienamente una formalizzazione della nozione di algoritmo. Per introdurre questo modello è naturale ricordare preliminarmente le proprietà generali di linguaggi formali, la nozione di grammatica e di automa a stati finiti che consentono di collocare e confrontare fra di loro le nozioni successive.

Tra gli obiettivi principali di questo capitolo citiamo i problemi NP-completi e il teorema di Cook per quanto riguarda la complessità in tempo, il teorema di Savitch e quello di Immerman-Szelepscényi per quella in spazio. Anche questo materiale è ormai classico e consolidato, ulteriori sviluppi e approfondimenti si trovano per esempio in [14, 16, 13, 2].

2.1 Generalità sui linguaggi formali

Un *alfabeto* è un insieme finito non vuoto di simboli che chiameremo anche lettere. Una *parola* (o stringa) su un alfabeto Σ è una concatenazione finita (o un allineamento finito) di elementi di Σ eventualmente ripetuti. Tra tutte le parole comprendiamo anche la parola vuota ε che non contiene simboli. Si denota con Σ^* l'insieme di tutte le parole su Σ, mentre con Σ^+ si rappresenta la famiglia di tutte le stringhe esclusa la parola vuota: $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Tra parole si definisce la tradizionale operazione di *concatenazione*: per ogni $x, y \in \Sigma^*$ la stringa $x \cdot y$ è quella che si ottiene concatenando x e y e che denoteremo più semplicemente mediante xy. In particolare assumiamo che per ogni $x \in \Sigma^*$, $x \cdot \varepsilon = \varepsilon \cdot x = x$. Quindi $\cdot$ è un'operazione associativa (in generale non commutativa) e ε è la sua unità. Di conseguenza l'insieme Σ^* dotato dell'operazione $\cdot$ forma un monoide, chiamato *monoide libero* su Σ.

Per ogni parola $x \in \Sigma^*$ la *lunghezza* di x, denotata $|x|$, è il numero di occorrenze di simboli di Σ in x mentre, per ogni lettera $a \in \Sigma$, $|x|_a$ rappresenta il numero di occorrenze di a in x. Chiaramente $|\varepsilon| = 0$. Chiamiamo inoltre *fattore* di x una stringa $z \in \Sigma^*$ tale che $x = yzw$ per qualche $y, w \in \Sigma^*$. Se invece $x = yw$ per qualche $y, w \in \Sigma^*$ diremo allora che y e w sono rispettivamente *prefisso* e *suffisso* di x; in particolare, ε e x sono sempre prefissi e suffissi dello stesso x. Osserva che il numero di prefissi (e di suffissi) di x è sempre $1 + |x|$.

Un *linguaggio* è un insieme di parole definite su un dato alfabeto. Fissato quindi un alfabeto Σ un linguaggio è un insieme $L \subseteq \Sigma^*$. L'insieme di tutti i linguaggi su Σ sarà denotato da $P(\Sigma^*)$. Si possono inoltre considerare le tradizionali operazioni insiemistiche sui linguaggi: per ogni $A, B \in P(\Sigma^*)$, denotiamo con $A \cup B$, $A \cap B$ l'unione e l'intersezione di A e B mentre $A^c = \{x \in \Sigma^* \mid x \notin A\}$ è il complemento di A. Inoltre, per ogni insieme finito $A \subseteq \Sigma^*$, $\sharp A$ denota la cardinalità di A cioè il numero dei suoi elementi.

Altre operazioni classiche sui linguaggi sono date dal prodotto e dalla chiusura di Kleene. Per ogni $A, B \in P(\Sigma^*)$, poniamo

$$A \cdot B = \{x \in \Sigma^* \mid x = yw, y \in A, w \in B\}$$

$$A^0 = \{\varepsilon\}, \quad A^n = A \cdot A^{n-1} \quad \text{per ogni } n \in \mathbb{N}$$

$$A^* = \bigcup_{n=0}^{+\infty} A^n, \quad A^+ = \bigcup_{n=1}^{+\infty} A^n$$

Qui l'operazione $\cdot$ denota il *prodotto* tra linguaggi mentre l'operazione * è chiamata *chiusura di Kleene*. Nota che in generale il prodotto di linguaggi non è commutativo. Osserviamo infine che $P(\Sigma^*)$ dotato delle operazioni $\cup$ e $\cdot$, insieme alle unità $\emptyset$ e $\{\varepsilon\}$, forma un semianello.

Le operazioni $\cup$, $\cdot$, * sono chiamate operazioni *razionali*.

2.2 Grammatiche

Le grammatiche sono sistemi formali che permettono la definizione di linguaggi e descrivono un metodo preciso per generare i loro elementi. Esse permettono di rappresentare un linguaggio, che in generale può possedere infiniti elementi, mediante una quantità finita di informazione.

Definizione 2.2.1. *Una* grammatica *è una quartupla* $G = (V, \Sigma, S, P)$ *tale che:*

- V *e* Σ *sono alfabeti disgiunti i cui elementi sono chiamati rispettivamente* variabili *e* simboli terminali*;*

- $S \in V$ *è chiamato* simbolo iniziale*;*

- P *è un insieme finito di stringhe dette* produzioni *della forma* $\alpha \to \beta$*, dove* $\alpha \in (V \cup \Sigma)^+$ *e* $\beta \in (V \cup \Sigma)^*$*.*

Data una grammatica $G = (V, \Sigma, S, P)$ possiamo definire la relazione $\Rightarrow_G$ di derivazione in un passo (denotata anche da $\Rightarrow$ se G è sottintesa): per ogni $x, y \in (V \cup \Sigma)^*$

$$x \Rightarrow_G y \text{ se esistono } \alpha, \beta, \gamma, \delta \in (V \cup \Sigma)^* \text{ tali che}$$
$$x = \gamma\alpha\delta,\; y = \gamma\beta\delta \text{ e } (\alpha \to \beta) \in P$$

Analogamente possiamo definire la relazione $\Rightarrow_G^*$ di derivazione in più passi (denotata anche con $\Rightarrow^*$) come la chiusura riflessiva e transitiva di $\Rightarrow_G$: per ogni $x, y \in (V \cup \Sigma)^*$, $x \Rightarrow_G^* y$ se $x = y$ oppure esistono $x_0, x_1, \ldots, x_n \in (V \cup \Sigma)^*$ tali che

$$x = x_0 \Rightarrow_G x_1 \Rightarrow_G \cdots x_{n-1} \Rightarrow_G x_n = y$$

Il linguaggio *generato* dalla grammatica $G = (V, \Sigma, S, P)$ è l'insieme $L(G) \subseteq \Sigma^*$ delle stringhe di simboli terminali derivabili dal simbolo iniziale S, ovvero

$$L(G) = \{x \in \Sigma^* \mid S \Rightarrow_G^* x\}$$

Nel seguito rappresenteremo solitamente i simboli terminali di una grammatica mediante le lettere minuscole all'inizio dell'alfabeto $(a, b, c, \ldots)$, mentre invece useremo le lettere maiuscole iniziali $(A, B, C, \ldots)$ per denotare le variabili. Useremo le lettere minuscole finali $(x, y, z, \ldots)$ per le parole di simboli terminali e le lettere dell'alfabeto greco $(\alpha, \beta, \gamma, \ldots)$ per le parole composte da simboli sia da variabili che da simboli terminali (appartenenti cioè a $(V \cup \Sigma)^*$).

Esempi 2.2.1.

1. *Il linguaggio* $L = \{a^n b^n \mid n \in \mathbb{N}\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$ *è generato dalla grammatica* $G = (\{S\}, S, \{a, b\}, \{S \to aSb, S \to \varepsilon\})$.
2. *Il linguaggio* $L = \{ab\}^* = \{\varepsilon, ab, abab, ababab, \ldots\}$ *è generato dalla grammatica di produzioni* $A \to abA$, $A \to \varepsilon$.
3. *Il linguaggio* $L = \{x \in \{a, b\}^* \mid x \text{ non contiene due } a \text{ consecutive}\}$ *è generato dalla grammatica di produzioni* $S \to aB$, $S \to bS$, $S \to \varepsilon$, $B \to bS$, $B \to \varepsilon$.
4. *Il linguaggio* $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ *è generato dalla grammatica di produzioni*

$$S \to aSBC, S \to \varepsilon, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$$

5. *Il linguaggio* $L \subseteq \{(,)\}^* = \{\varepsilon, (), ()(), (()), \ldots\}$ *delle parentesi* $(,)$ *correttamente innestate è generato dalla grammatica di produzioni* $S \to (S)S$, $S \to \varepsilon$.

Le grammatiche possono essere raggruppate in classi a seconda della forma delle loro produzioni. Ciascuna classe di grammatiche definisce automaticamente la corrispondente famiglia dei linguaggi generati. Di conseguenza anche i linguaggi possono essere suddivisi in base al tipo di produzione che consente di generarne gli elementi. La classificazione che qui ricordiamo è quella classica, chiamata *gerarchia di Chomsky*. Essa è formata in prima istanza da quattro famiglie di grammatiche e quindi di linguaggi corrispondenti: le grammatiche di tipo 0, le grammatiche dipendenti da contesto, quelle libere da contesto e le grammatiche regolari.

Una grammatica di *tipo* 0 è una grammatica generica, priva di vincoli particolari; la sua definizione coincide quindi con quella data all'inizio di questa sezione. Un linguaggio di tipo 0 è quindi un linguaggio generato da una grammatica qualsiasi. È chiaro che ogni grammatica G defisce automaticamente una procedura per elencare gli elementi di $L(G)$. Per questo

motivo i linguaggi di tipo 0 sono anche detti *linguaggi ricorsivamente numerabili*. Le proprietà di questi linguaggi sono analoghe a quelle dei sottoinsiemi di $\mathbb{N}$ ricorsivamente numerabili, presentati nella sezione 1.12. Si può anzi dimostrare, mediante una opportuna corrispondenza biunivoca (calcolabile), che ogni linguaggio ricorsivamente numerabile definito su un dato alfabeto può essere trasformato in un sottoinsieme di $\mathbb{N}$ ricorsivamente numerabile, e viceversa. Anche in questo caso quindi, come per gli insiemi di numeri naturali, si può dimostrare che esistono linguaggi che non sono ricorsivamente numerabili. Inoltre, intuitivamente, è chiaro che un linguaggio è ricorsivamente numerabile se e solo se esiste una procedura, definita in un qualunque linguaggio di programmazione, che stampa tutti i suoi elementi.

Una grammatica di tipo 1 o *dipendente dal contesto* è una grammatica $G = (V, \Sigma, S, P)$ nella quale ogni produzione è della forma $\alpha \to \beta$ dove $|\alpha| \le |\beta|$. Un linguaggio è detto dipendente da contesto se è generato da una grammatica di tipo 1. Modificando opportunamente le grammatiche presentate nell'Esempio 2.2.1 è facile verificare che i seguenti linguaggi sono dipendenti da contesto: $\{a^n b^n \mid n \in \mathbb{N}, n > 0\}$, $\{ab\}^+$, $\{x \in \{a,b\}^+ \mid x \text{ non contiene due } a \text{ consecutive}\}$, $\{a^n b^n c^n \mid n \in \mathbb{N}, n > 0\}$.

Una grammatica di tipo 2 o *libera da contesto* (context-free) è una grammatica $G = (V, \Sigma, S, P)$ nella quale ogni produzione in P è della forma $A \to \beta$ dove $A \in V$ e $\beta \neq \varepsilon$. Un linguaggio si dice libero da contesto (o di tipo 2) se è generato da una grammatica libera da contesto. Per esempio, i linguaggi $\{ab\}^+$, $\{x \in \{a,b\}^+ \mid x \text{ non contiene due } a \text{ consecutive}\}$ e $\{a^n b^n \mid n \in \mathbb{N}, n > 0\}$ sono liberi da contesto.

Come ulteriore esempio considera la grammatica $(\{A, B, S\}, \{a, b\}, S, P)$ nella quale

$$P = \{S \to aB, S \to bA, A \to a, A \to aS, A \to bAA, B \to b, B \to bS, B \to aBB\}$$

È facile verificare che tale grammatica genera il linguaggio $\{x \in \{a,b\}^+ \mid |x|_a = |x|_b\}$ che risulta quindi libero da contesto.

Infine, una grammatica di tipo 3 o *regolare* è una grammatica $G = (V, \Sigma, S, P)$ nella quale ogni produzione in P è della forma $A \to a$ oppure $A \to aB$, dove $A, B \in V$ e $a \in \Sigma$. Un linguaggio si dice regolare se è generato da una grammatica regolare. Come esempio considera la grammatica $(\{A, B, S\}, \{a, b\}, S, P)$ nella quale $P = \{S \to aA, A \to bC, C \to a, C \to b, C \to aC, C \to bC\}$, che genera il linguaggio $ab\{a,b\}^+ = \{abx \mid x \in \{a,b\}^+\}$. Inoltre, è facile verificare che anche i linguaggi $\{ab\}^+$ e $\{x \in \{a,b\}^+ \mid x \text{ non contiene due } a \text{ consecutive}\}$ considerati sopra sono regolari.

Dalle definizioni precedenti è chiaro che ogni grammatica regolare è anche libera da contesto e ogni grammatica libera da contesto è a sua volta dipendente da contesto. Così la famiglia dei linguaggi regolari è inclusa in quella dei linguaggi liberi da contesto e quest'ultima è contenuta nell'insieme dei linguaggi dipendenti da contesto. Si può dimostrare che tutte queste inclusioni sono strette:

$$\text{Linguaggi regolari } \subsetneq \text{ Linguaggi liberi da contesto}$$
$$\subsetneq \text{ Linguaggi dipendenti da contesto } (\subsetneq \text{ Linguaggi di tipo 0 })$$

Osserviamo che per le definizioni precedenti un linguaggio contenente la parola vuota ε non può essere generato da una grammatica dipendente da contesto e quindi non può essere di nessuno dei tre tipi. Poiché vogliamo dare nozioni che siano indipendenti dalla presenza o meno di una singola parola nel linguaggio, estendiamo le definizioni precedenti (di grammatica dipendente da contesto, libera da contesto e regolare) permettendo la presenza della produzione $S \to \varepsilon$, dove S è simbolo iniziale della grammatica, ammesso però che S non appaia nella parte destra di alcuna produzione. In questo caso quindi S può essere usata solo all'inizio di ogni derivazione. Osserviamo, per inciso, che ogni grammatica di tipo 1 (ripettivamente di tipo 2 o 3) può essere trasformata in una grammatica equivalente (cioè che genera lo stesso linguaggio) dello stesso tipo, nella quale il simbolo iniziale non compare nella parte destra di alcuna produzione.

Così diciamo che una grammatica $G = (V, \Sigma, S, P)$ è dipendente da contesto se ogni produzione è della forma $\alpha \to \beta$, dove $|\alpha| \leq |\beta|$, oppure è la produzione $S \to \varepsilon$ e in questo caso S non appare nella parte destra di alcuna produzione. Le definizioni di grammatica libera da contesto e di grammatica regolare possono essere estese nello stesso modo. Le predendenti proprietà di inclusione tra famiglie di linguaggi (e di grammatiche) restano valide anche con la nuova definizione. Inoltre si può dimostrare che se un linguaggio L è dipendente da contesto, libero da contesto o regolare, allora anche $L \cup \{\varepsilon\}$ e $L - \{\varepsilon\}$ sono rispettivamente dipendenti da contesto, liberi da contesto o regolari.

Di conseguenza, i linguaggi $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, $\{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$ e $\{ab\}^*$ sono ora, rispettivamente, dipendente da contesto, libero da contesto e regolare.

2.3 Automi a stati finiti

Nella sezione precedente abbiamo visto come un linguaggio contenente potenzialmente un numero infinito di elementi, possa essere rappresentato da una grammatica, ovvero da un sistema formale, descritto mediante una quantità finita di informazione, in grado di generare tutte e sole le parole del linguaggio. Un altro modo per rappresentare un linguaggio L su un dato alfabeto Σ è quello di definire una macchina formale in grado di leggere una qualunque parola in Σ^* e di verificare, in un numero finito di passi se tale parola appartiene a L. Un automa a stati finiti è un modello di questo tipo particolarmente semplice. Esso è dotato di uno stato interno che può assumere un numero finito di valori, tra i quali vi sono lo stato iniziale e un sottoinsieme di stati finali, e da un nastro di ingresso contenente una stringa di input, formata da simboli di Σ. La macchina legge i simboli della stringa di input uno dopo l'altro mediante una testina di lettura che scandisce una lettera alla volta e si muove sempre verso destra. Inizialmente l'automa si trova nello stato iniziale e legge il primo simbolo della stringa di input. Ad ogni passo, a seconda del simbolo letto e dello stato corrente la macchina cambia stato e muove la testina di una posizione verso destra leggendo il simbolo successivo. Se lo stato raggiunto dopo aver letto l'ultimo simbolo di ingresso è uno stato finale diciamo che l'automa accetta la stringa di input, altrimenti si dice che l'automa rifiuta la stringa. Il linguaggio riconosciuto dall'automa è l'insieme delle stringhe accettate.

Formalmente, fissato un alfabeto Σ, un automa a stati finiti (per brevità, automa s.f.) è una quartupla $\mathscr{A} = (Q, q_0, \delta, F)$ nella quale Q è un insieme finito di stati, $q_0 \in Q$ è lo stato iniziale, $\delta : Q \times \Sigma \to Q$ è un funzione detta funzione di transizione e $F \subseteq Q$ è la famiglia degli stati finali. Tale automa sarà anche chiamato deterministico per distinguerlo dalla versione non deterministica che definiremo in seguito.

La funzione transizione può essere estesa all'insieme $Q \times \Sigma^*$ ponendo, per ogni $q \in Q$, ogni $a \in \Sigma$ e ogni $x \in \Sigma^*$,

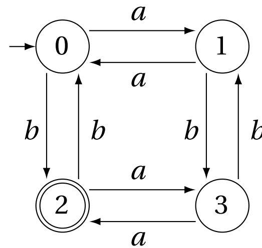
$$\delta(q, \varepsilon) = q, \qquad \delta(q, ax) = \delta(\delta(q, a), x)$$

In questo modo, per ogni $q \in Q$ e ogni $x \in \Sigma^*$, $\delta(q, x)$ appartiene a Q e rappresenta lo stato raggiunto dall'automa leggendo l'input x a partire dallo stato q. Diciamo che una parola $x \in \Sigma^*$ è *accettata* da $\mathscr{A}$ se $\delta(q_0, x) \in F$. Il linguaggio *riconosciuto* dall'automa, tradizionalmente denotato $L(\mathscr{A})$, è

definito come l'insieme delle parole accettate da $\mathscr{A}$, ovvero

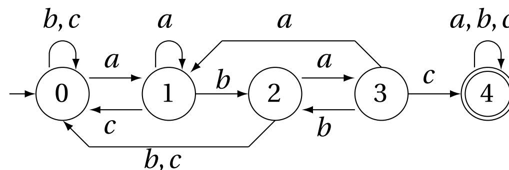
$$L(\mathscr{A}) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$$

Un automa a stati finiti $\mathscr{A} = (Q, q_0, \delta, F)$ sull'alfabeto Σ può essere descritto mediante un grafo orientato con lati etichettati da lettere di Σ, nel quale Q è l'insieme dei vertici e per ogni $q, p \in Q$ esiste un arco da q a p etichettato da una lettera a se e solo se $\delta(q, a) = p$. Inoltre lo stato iniziale q_0 è marcato con una freccia entrante mentre ogni stato finale è denotato da un doppio cerchio. Per esempio, il seguente automa sull'alfabeto $\{a, b\}$ riconosce il linguaggio $L = \{x \in \Sigma^* \mid |x|_a$ è pari , $|x|_b$ è dispari$\}$:



Un altro esempio è dato dal seguente automa a stati finiti che riconosce il linguaggio di tutte le parole dell'alfabeto $\{a, b, c\}$ contenenti il fattore $abac$, cioè

$$L = \{a, b, c\}^* abac\{a, b, c\}^* = \{x \in \{a, b, c\}^* \mid \exists\ u, v \in \{a, b, c, \}^* : x = uabacv\}$$



Un altro modello classico di riconoscitore di linguaggi è dato dagli automi a stati finiti non deterministici. Questi forniscono un esempio elementare di macchina non deterministica, nella quale ad ogni passo non vi è un'unica possibile mossa, ma la macchina può scegliere la nuova configurazione in un dato insieme finito di possibili configurazioni. La definizione che introduciamo è simile alla precedente, la differenza principale è data dalla funzione transizione che in questo caso associa allo stato corrente e al simbolo letto un insieme di stati.

Un automa a stati finiti non deterministico su un alfabeto Σ è una quartupla $\mathscr{A} = (Q, q_0, \delta, F)$ nella quale Q, q_0 e F sono definiti come nel caso deterministico, mentre δ è una funzione $\delta : Q \times \Sigma \to 2^Q$ (dove 2^Q è l'insieme delle parti di Q). Intuitivamente, per ogni $q \in Q$ e $a \in \Sigma$, se $\delta(q,a) = \emptyset$ allora l'automa leggendo il simbolo a nello stato q non compie alcuna mossa e si ferma (l'input in questo caso viene comunque rifiutato), se $\delta(q,a)$ contiene un solo stato p la macchina entra nello stato p come nel caso deterministico, se invece $\delta(q,a)$ contiene più di un elemento allora la macchina sceglie tra questi lo stato in cui entrare; in questo modo, per ogni stringa di input abbiamo una famiglia di computazioni, una per ogni possibile sequenza di scelte compiute dalla macchina sull'input dato. La stringa viene accettata se tra tutte queste computazioni ne abbiamo almeno una che termina in uno stato finale.

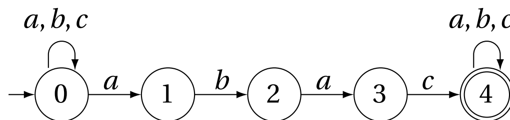
Formalmente possiamo estendere la funzione δ all'insieme $Q \times \Sigma^*$, ponendo per ogni $q \in Q$, $a \in \Sigma$, $x \in \Sigma^*$

$$\delta(q,\varepsilon) = \{q\}, \qquad \delta(q,ax) = \bigcup_{p \in \delta(q,a)} \delta(p,x)$$

Così, $\delta(q,x)$ costituisce l'insieme degli stati raggiungibili dall'automa a partire dallo stato q leggendo la stringa x. Il linguaggio riconosciuto dall'automa $\mathscr{A}$ è quindi definito da

$$L(\mathscr{A}) = \{x \in \Sigma^* \mid \delta(q_0,x) \cap F \neq \emptyset\}$$

Per esempio, nel seguente automa abbiamo $\delta(0, abab) = \{0,2\}$ e il linguaggio riconosciuto è nuovamente dato dalle parole sull'alfabeto $\{a,b,c\}$ che contengono il fattore $abac$.



Si può dimostrare facilmente che i due modelli di automa sopra definiti riconoscono la stessa famiglia di linguaggi. Inoltre, quest'ultima coincide proprio con l'insieme dei linguaggi regolari introdotti nella sezione precedente.

Proposizione 2.3.1. *Un linguaggio $L \subseteq \Sigma^*$ è riconosciuto da un automa a stati finiti deterministico se e solo se L è riconosciuto da un automa a stati finiti non deterministico.*

Dimostrazione. In un senso l'implicazione è ovvia perché ogni automa s.f. deterministico è di fatto un automa s.f. non deterministico.

Supponiamo ora che un linguaggio $L \subseteq \Sigma^*$ sia riconosciuto da un automa s.f. non deterministico $\mathscr{A} = (Q, q_0, \delta, F)$. Definiamo allora l'automa s.f. determistico $\mathscr{B} = (2^Q, \{q_0\}, \delta', F')$, dove $F' = \{X \subseteq Q \mid X \cap F \neq \emptyset\}$ e $\delta'(X, a) = \bigcup_{p \in X} \delta(p, a)$ per ogni $X \in 2^Q$ e ogni $a \in \Sigma$. È facile verificare che una qualunque stringa $x \in \Sigma^*$ appartiene a $L(\mathscr{A})$ se e solo se $\delta'(\{q_0\}, x) \in F'$ e quindi $L(\mathscr{A}) = L(\mathscr{B})$.

Nota che in generale il numero degli stati dell'automa deterministico $\mathscr{B}$ è esponenziale rispetto a quello di $\mathscr{A}$. Quindi, un automa non deterministico può essere simulato da uno deterministico ma al prezzo di aumentare considerevolmente il numero degli stati. □

Proposizione 2.3.2. *Un linguaggio L è regolare se e solo se L è riconosciuto da un automa a stati finiti.*

Dimostrazione. Sia $L \subseteq \Sigma^*$ un linguaggio regolare e costruiamo un automa s.f. non deterministico $\mathscr{A}$ che riconosce L. Considera una grammatica regolare $G = (V, \Sigma, S, P)$ che genera L. P contiene solo produzioni del tipo $A \to a$ e $A \to aB$, tranne al più la produzione $S \to \varepsilon$ (nel qual caso S non appare nella parte destra di alcuna produzione in P). Definiamo allora l'automa non deterministico $\mathscr{A}$ su Σ, $\mathscr{A} = (Q, q_0, \delta, F)$, tale che $Q = V \cup \{q_f\}$ con q_f nuovo simbolo non incluso in V, $q_0 = S$, e $F = \{S, q_f\}$ se $\varepsilon \in L$ mentre $F = \{q_f\}$ se $\varepsilon \notin L$; inoltre, per ogni $a \in \Sigma$ e ogni $q \in V$,

$$\delta(q, a) = \begin{cases} \{p \in V \mid (q \to ap) \in P\} & \text{se } (q \to a) \notin P \\ \{p \in V \mid (q \to ap) \in P\} \cup \{q_f\} & \text{altrimenti} \end{cases}$$

mentre $\delta(q_f, a) = \emptyset$.

Per dimostrare che L coincide con il linguaggio riconosciuto da $\mathscr{A}$, considera una parola $x \in \Sigma^+$ generabile in G, cioè $S \Rightarrow_G^* x$. La successione delle produzioni usate nella derivazione corrisponde ai passi di computazione dell'automa e siccome l'ultima produzione applicata è della forma $(q \to a)$ abbiamo che $q_f \in \delta(q_0, x)$, per cui x è riconosciuta dall'automa $\mathscr{A}$. Viceversa, se $q_f \in \delta(q_0, x)$ allora la stessa sequenza di stati definita dalla computazione di $\mathscr{A}$ su x determina univocamente una derivazione di x nella grammatica, ovvero $S \Rightarrow_G^* x$ e di conseguenza $x \in L$. La stessa proprietà vale per la parola vuota ε, cioè $(S \to \varepsilon) \in P$ se e solo se $q_0 \in F$, per cui $\varepsilon \in L$ se e solo se ε è accettato da $\mathscr{A}$. Di conseguenza L coincide proprio con il linguaggio riconosciuto dall'automa.

Per dimostrare l'implicazione nel senso opposto, considera un linguaggio $L \subseteq \Sigma^*$ riconosciuto da un automa s.f. $\mathscr{A} = (Q, q_0, \delta, F)$. Per la proposizione precedente possiamo supporre che $\mathscr{A}$ sia deterministico. Vogliamo costruire una grammatica regolare che genera L. A questo scopo, supponiamo che $\varepsilon \notin L$ e definiamo la grammatica $G = (Q, \Sigma, q_0, P)$ tale che

$$P = \{q \to ap \mid \delta(q, a) = p\} \cup \{q \to a \mid \exists p \in F \, : \, \delta(q, a) = p\}$$

È chiaro che la grammatica G è regolare e si dimostra facilmente che il linguaggio generato da G coincide con L. Se invece $\varepsilon \in L$ basta modificare la grammatica G aggiungendo alle variabili un nuovo simbolo iniziale S (non presente in Q), dotato delle stesse produzioni uscenti da q_0 e in più della produzione $S \to \varepsilon$. Anche in questo caso si prova che L coincide con il linguaggio generato da G. □

Esercizio
Dimostrare che l'unione, l'intersezione e il complemento di linguaggi regolari sono ancora regolari. In questo modo la famiglia dei linguaggi regolari su un dato alfabeto forma un'algebra di Boole.

Illustriamo ora un'altra proprietà dei linguaggi regolari chiamata Lemma di iterazione che fornisce sostanzialmente una condizione necessaria affinché un linguaggio sia regolare. Questa condizione può essere utilizzata per provare che un linguaggio non è regolare, ovvero che non esiste un automa a stati finiti in grado di riconoscerlo, né una grammatica regolare in grado di generarlo.

Proposizione 2.3.3. *Se $L \subseteq \Sigma^*$ è un linguaggio regolare allora esiste un intero $n > 0$ tale che, per ogni $x \in L$ di lunghezza maggiore di n esistono $u, v, w \in \Sigma^*$ che soddisfano le seguenti condizioni: $x = uvw$, $|uv| \leq n$, $|v| \geq 1$, $uv^k w \in L$ per ogni $k \in \mathbb{N}$.*

Dimostrazione. Poiché L è regolare consideriamo un automa s.f. deterministico $\mathscr{A} = (Q, q_0, \delta, F)$ che riconosce L e sia n la cardinalità di Q. Ogni parola $x \in L$ di lunghezza maggiore di n può essere rappresentata nella forma $x = a_1 a_2 \cdots a_m$, con $m > n$ e $a_i \in \Sigma$ per tutti gli i. Possiamo inoltre considerare la computazione di $\mathscr{A}$ su input x e denotare con $q_0, q_1, \ldots, q_m$ la sequenza degli stati nei quali l'automa entra leggendo una dopo l'altra le lettere di x, compreso lo stato di partenza q_0. Nota che q_0 è lo stato iniziale e $q_m \in F$. Poiché $m > n$ esistono almeno due stati uguali che si ripetono nella sequenza; siano allora i, j, $0 \leq i < j \leq m$, i primi due indici tali che

$q_i = q_j$. La stringa x può allora essere decomposta nella forma uvw, con $u = a_1 \cdots a_i$, $v = a_{i+1} \cdots a_j$, $w = a_{j+1} \cdots a_m$ (con la convenzione che $u = \varepsilon$ se $i = 0$, mentre $w = \varepsilon$ se $j = m$). È chiaro che $|uv| \leq n$ perché gli stati $q_0, \ldots, q_{j-1}$ sono tutti distinti tra loro, $|v| \geq 1$ perché v non può essere vuota e inoltre, per ogni $k \in \mathbb{N}$, $uv^k w$ appartiene a L perché su input $uv^k w$ l'automa $\mathscr{A}$, partendo da q_0, arriva comunque allo stato finale q_m (ripetendo k volte il ciclo $q_i \cdots q_j$). Le proprietà di x sono quindi tutte vere e questo conclude la prova. □

Usando la proposizione precedente è facile provare che il linguaggio $\{a^n b^n \mid n \in \mathbb{N}\}$ non è regolare.

Esercizio
Dimostrare che il linguaggio $\{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$ non è regolare.

2.3.1 Linguaggi regolari e operazioni razionali

Concludiamo questa sezione ricordando una caratterizzazione rilevante dei linguaggi regolari basata sulle operazioni razionali, nota come Teorema di Kleene. Oltre a questo risultato, la teoria degli automi finiti contiene molte altre proprietà computazionali e algoritmiche di notevole interesse che in questa sede non possono essere presentate. Rimandiamo la loro discussione alle opere più specifiche [9, 14, 13].

Denotiamo anzitutto con *Rat* la più piccola famiglia di linguaggi contenente gli insiemi finiti di stringhe e chiusa rispetto alle operazioni di unione, prodotto e chiusura di Kleene.

Teorema 2.3.4. *La classe* Rat *coincide con la famiglia dei linguaggi regolari.*

Dimostrazione. Proviamo per prima cosa che ogni L appartenente a *Rat* è regolare. Questo significa dimostrare che la famiglia dei linguaggi regolari contiene gli insiemi finiti ed è chiusa rispetto alle operazioni razionali. Proviamo anzitutto la chiusura rispetto all'unione.

Siano $\mathscr{A}_1 = (Q_1, q_1, \delta_1, F_1)$ e $\mathscr{A}_2 = (Q_2, q_2, \delta_2, F_2)$ due automi a stati finiti deterministici che riconoscono, rispettivamente, i linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$. Costruiamo l'automa s.f. $\mathscr{A}$ che riconosce $L_1 \cup L_2$. Osserva anzitutto che, se gli alfabeti Σ_1 e Σ_2 sono diversi, possiamo estendere gli automi $\mathscr{A}_1$ e $\mathscr{A}_2$ a tutte i simboli dell'alfabeto $\Sigma = \Sigma_1 \cup \Sigma_2$, introducendo un nuovo stato $t \notin Q_1 \cup Q_2$ non finale, detto stato "trappola", dal quale non si potrà uscire. Formalmente, definiamo $\delta_1(q, a) = t$ per ogni

$q \in Q_1$ e ogni $a \in \Sigma_2 - \Sigma_1$, insieme a $\delta_1(t,b) = t$ per ogni $b \in \Sigma$. Così l'automa $(Q_1 \cup \{t\}, q_1, \delta_1, F_1)$ è definito su tutto Σ e riconosce ancora L_1. Analogamente possiamo operare su $\mathscr{A}_2$, aggiungendo t all'insieme di stati Q_2 ed estendendo in maniera simmetrica la funzione δ_2. Definiamo allora $\mathscr{A} = ((Q_1 \cup \{t\}) \times (Q_2 \cup \{t\}), (q_1, q_2), \delta, F)$, dove $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ e $\delta((q,p),a) = (\delta_1(q,a), \delta_2(p,a))$ per ogni $a \in \Sigma$, ogni $q \in Q_1 \cup \{t\}$ e ogni $p \in Q_2 \cup \{t\}$. In pratica il nuovo automa, su ogni input $x \in \Sigma^*$, esegue in parallelo le computazioni di $\mathscr{A}_1$ e $\mathscr{A}_2$ sullo stesso input, accettando solo quando uno dei due automi accetta. Di conseguenza $\mathscr{A}$ riconosce proprio il linguaggio $L_1 \cup L_2$.

È facile inoltre costruire un automa s.f. deterministico che riconosce una sola stringa $w = w_1 w_2 \cdots w_k$: basta fissare $k+1$ stati distinti $q_0, q_1, \ldots, q_k$ e uno stato trappola t, ponendo q_0 stato iniziale, q_k unico stato finale, con le transizioni $\delta(q_{i-1}, w_i) = q_i$ per ogni $i = 1, 2 \ldots, k$ e mandando tutte le altre transizioni in t (dal quale non si può uscire). L'automa così definito riconosce chiaramente la sola stringa w. Applicando ora la proprietà di chiusura rispetto all'unione (appena dimostrata), è immediato dedurre che ogni linguaggio finito è regolare.

Dimostriamo ora la chiusura rispetto al prodotto. Siano di nuovo $\mathscr{A}_1 = (Q_1, q_1, \delta_1, F_1)$ e $\mathscr{A}_2 = (Q_2, q_2, \delta_2, F_2)$ due automi s.f. deterministici che riconoscono, rispettivamente, i linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$. Senza perdita di generalità possiamo supporre che Q_1 e Q_2 siano disgiunti e, ragionando come nel caso precedente, entrambi gli automi siano definiti sul medesimo alfabeto $\Sigma = \Sigma_1 \cup \Sigma_2$, per cui $L_1 \subseteq \Sigma^*$ e $L_2 \subseteq \Sigma^*$. Costruiamo allora un nuovo automa s.f. non deterministico $\mathscr{A} = (Q_1 \cup Q_2, q_1, \delta, F)$, dove la funzione transizione δ è definita nel modo seguente per ogni $a \in \Sigma$:

$$\delta(q,a) = \begin{cases} \{\delta_1(q,a)\} & \text{se } q \in Q_1 - F_1 \\ \{\delta_1(q,a), \delta_2(q_2,a)\} & \text{se } q \in F_1 \\ \{\delta_2(q,a)\} & \text{se } q \in Q_2 \end{cases}$$

In questo modo l'automa $\mathscr{A}$ inizia ogni computazione simulando l'automa $\mathscr{A}_1$ e, trovandosi in uno stato finale $q \in F_1$, $\mathscr{A}$ sceglie in modo non deterministico se proseguire la simulazione di $\mathscr{A}_1$ oppure iniziare a comportarsi come $\mathscr{A}_2$, eseguendo la transizione definita dallo stato q_2. L'insieme F degli stati finali di $\mathscr{A}$ sarà inoltre dato da

$$F = \begin{cases} F_2 & \text{se } \varepsilon \notin L_2 \text{ (cioè } q_2 \notin F_2) \\ F_1 \cup F_2 & \text{altrimenti} \end{cases}$$

L'automa $\mathscr{A}$ riconosce chiaramente il prodotto $L_1 \cdot L_2$, cioè l'insieme della parole $w \in \Sigma^*$ tali che $w = xy$ per qualche $x \in L_1$ e qualche $y \in L_2$.

Proviamo ora che per ogni $L \subseteq \Sigma^*$ regolare anche L^* è regolare. Sia $\mathscr{A} = (Q, q_0, \delta, F)$ un automa s.f. deterministico che riconosce L. L'idea è quella di costruire un nuovo automa s.f. non deterministico che simula $\mathscr{A}$ ma che, prima di entrare in uno stato finale q, sceglie in modo non deterministico se eseguire effettivamente la transizione in q oppure entrare nello stato iniziale q_0. Inoltre, per riconoscere la parola vuota, il nuovo automa avrà un nuovo stato iniziale p_0, non incluso in Q, che sarà anche finale e si comporterà come q_0. Formalmente, definiamo l'automa $\mathscr{B} = (Q \cup \{p_0\}, p_0, \tau, F \cup \{p_0\})$, nel quale la funzione transizione τ è data dalle seguenti relazioni, per ogni $a \in \Sigma$:

$$\text{per ogni } q \in Q \text{ , } \tau(q,a) = \begin{cases} \{\delta(q,a), q_0\} & \text{se } \delta(q,a) \in F \\ \{\delta(q,a)\} & \text{altrimenti} \end{cases} ,$$

$$\tau(p_0,a) = \begin{cases} \{\delta(q_0,a), q_0\} & \text{se } \delta(q_0,a) \in F \\ \{\delta(q_0,a)\} & \text{altrimenti} \end{cases}$$

Si può dimostrare direttamente che per ogni parola $w \in \Sigma^*$, w appartiene a L^* se e solo se $\mathscr{B}$ riconosce w. Ne segue che anche L^* è regolare.

Proviamo ora che ogni linguaggio regolare appartiene a *Rat*, ovvero che può essere ricavato dagli insiemi finiti di stringhe usando le operazioni razionali ($\cup$, $\cdot$, $*$). Sia $L \subseteq \Sigma^*$ un linguaggio regolare e sia $\mathscr{A} = (Q, q_1, \delta, F)$ un automa s.f. deterministico che riconosce L. Supponiamo che l'insieme degli stati dell'automa sia $Q = \{q_1, q_2, \dots, q_n\}$. Allora, per ogni tripla di indici $i, j, k \in \{1, 2, \dots, n\}$, possiamo definire l'insieme R^k_{ij} come la famiglia di tutte le parole in Σ^* che conducono l'automa dallo stato q_i allo stato q_j senza passare per gli stati q_ℓ di indice $\ell > k$. Formalmente, ogni R^k_{ij} è definito nel modo seguente:

$$R^k_{ij} = \{\, w \in \Sigma^* : \quad \delta(q_i, w) = q_j \ \text{ e, per ogni prefisso } y \text{ di } w \text{ diverso da } \varepsilon \text{ e da } w, \ \text{vale } \ \delta(q_i, y) = q_\ell \ \Rightarrow \ l \le k \,\}$$

Nota che i e j possono essere maggiori k e nello stesso tempo R^k_{ij} non essere vuoto. Inoltre è chiaro che $L = \bigcup_{q_j \in F} R^n_{1j}$. Estendiamo quindi la definizione dei linguaggi R^k_{ij} al caso $k = 0$ ponendo

$$R^0_{ij} = \begin{cases} \{a \in \Sigma : \delta(q_i, a) = q_j\} & \text{se } i \ne j \\ \{a \in \Sigma : \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & \text{se } i = j \end{cases}$$

Allora, dalla stessa definizione appena data, si ottiene il seguente sistema di uguaglianze, per tutti gli indici $i, j, k \in \{1, 2, \ldots, n\}$:

$$R_{ij}^{k} = R_{ij}^{k-1} \cup R_{ik}^{k-1} \cdot (R_{kk}^{k-1})^{*} \cdot R_{kj}^{k-1}$$

Infatti, ogni parola w che appartiene a R_{ij}^{k} conduce l'automa da q_i a q_j senza passare per lo stato q_k (e allora appartiene a R_{ij}^{k-1}), oppure passando effettivamente per lo stato q_k, e allora w è necessariamente della forma $w = xyz$ con $x \in R_{ik}^{k-1}$, $y \in (R_{kk}^{k-1})^{*}$ e $z \in R_{kj}^{k-1}$.

Questo significa che la famiglia di linguaggi $\{R_{ij}^{k} : i, j \in \{1, 2, \ldots, n\}\}$ può essere ottenuta dalla famiglia $\{R_{ij}^{k-1} : i, j \in \{1, 2, \ldots, n\}\}$ usando le operazioni $\cup$, $\cdot$, $*$. Poiché R_{ij}^{0} è finito per ogni $i, j \in \{1, 2, \ldots, n\}$, possiamo concludere che tutti i linguaggi R_{ij}^{k} appartengono a *Rat*. Quindi, essendo $L = \bigcup_{q_j \in F} R_{1j}^{n}$, anche L appartiene a *Rat*. □

2.4 Macchine di Turing deterministiche

Una macchina di Turing deterministica è un modello di calcolo generale introdotto per formalizzare la nozione di algoritmo, in grado sia di riconoscere linguaggi sia di calcolare funzioni (parziali) da un monoide libero a un altro.

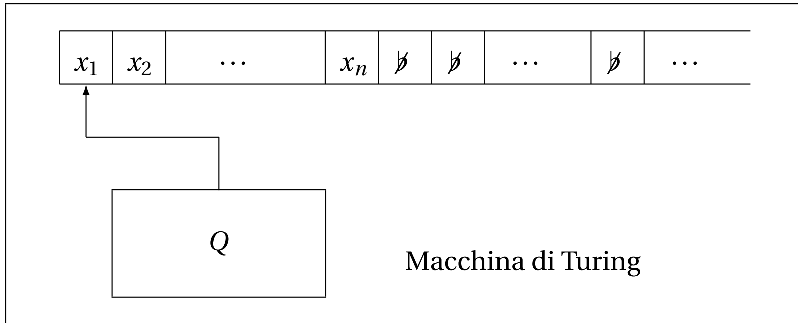
Diamo innazitutto una descrizione discorsiva e informale di questo modello. Esso è costituito essenzialmente da tre dispositivi:

- un nastro suddiviso in infinite celle, numerate e allineate verso destra, ciascuna delle quali contiene un simbolo estratto da un alfabeto di lavoro;

- una testina di lettura e scrittura che scandisce una cella del nastro alla volta e può spostarsi ad ogni mossa della macchina in una delle due celle adiacenti, oppure rimanere ferma;

- un insieme finito di stati che rappresentano una condizione interna della macchina. Come negli automi a stati finiti, in ogni momento del calcolo la macchina può trovarsi in un solo stato, detto anche stato corrente.

All'inizio di una computazione il modello si trova nella seguente configurazione: (i) la testina è collocata sulla prima cella del nastro; (ii) la macchina si trova in uno stato particolare, detto stato iniziale; (iii) una stringa di input $x = x_1 x_2 \cdots x_n$ è inserita nelle prime n celle del nastro, dove ogni simbolo x_i appartiene a un alfabeto di input, mentre tutte le altre celle contengono un simbolo speciale chiamato blank, che denotiamo con $\not b$.



A partire da questa configurazione la macchina può eseguire una sequenza (eventualmente infinita) di mosse. Ogni mossa è univocamente determinata dallo stato nel quale la macchina si trova e dal simbolo letto dalla testina. A ogni mossa la macchina entra in un nuovo stato, stampa un simbolo nella cella scandita dalla testina, cancellando il simbolo contenuto in precedenza, e la testina si muove di una posizione verso destra o verso sinistra, oppure rimane sulla stessa cella. Vi sono due stati particolari, q_s e q_n, nei quali la macchina si ferma e non esegue alcuna mossa. La stringa di input è accettata se la macchina raggiunge lo stato q_s; se invece questa raggiunge lo stato q_n la stringa di ingresso è rifiutata. Il linguaggio accettato dalla macchina è costituito dall'insieme delle stringhe accettate. Nota che una stringa x non appartiene al linguaggio accettato se la macchina, su input x, entra nello stato q_n oppure non si ferma, eseguendo così infinite mosse.

Formalmente una macchina di Turing deterministica (MdT nel seguito) è una sestupla

$$M = (Q, q_0, \Gamma, \Sigma, \delta, \{q_s, q_n\})$$

nella quale:

1. Q è un insieme finito di stati;

2. $q_0 \in Q$ è lo stato iniziale;

3. Γ è l'alfabeto di lavoro, che contiene tra i suoi elementi il simbolo $\not{b}$, detto blank;

4. Σ è l'alfabeto di input (o di ingresso), $\Sigma \subset \Gamma$ ma $\not{b} \not\in \Sigma$ (e quindi $\not{b} \in \Gamma - \Sigma$);

5. δ è la funzione transizione tale che

$$\delta \ : \ (Q - \{q_s, q_n\}) \times \Gamma \ \longrightarrow \ Q \times (\Gamma - \{\not{b}\}) \times \{-1, 0, 1\}$$

6. $\{q_s, q_n\} \subset Q$ è l'insieme degli stati di arresto.

Per ogni $q \in Q - \{q_s, q_n\}$ e ogni $a \in \Gamma$, il valore $\delta(q, a)$ definisce la mossa di M quando la macchina si trova nello stato q e la testina legge il simbolo a: se $\delta(q, a) = (p, b, \ell)$ allora M entra nello stato p, stampa il simbolo b al posto di a nella cella scandita dalla testina (nota che b è sempre diverso da $\not{b}$) e sposta la testina di una posizione verso sinistra se $\ell = -1$, verso destra se $\ell = 1$, mentre la lascia sulla stessa cella se $\ell = 0$. Ribadiamo il fatto che la macchina non può mai stampare il blank, il quale a sua volta non fa parte dei simboli di input. Si suppone inoltre che leggendo la prima cella del nastro la macchina non muova mai la testina verso sinistra. Questo si può realizzare definendo opportunamente la funzione di transizione, in modo tale che alla prima mossa la macchina stampi nella prima cella solo simboli particolari leggendo i quali lo spostamento a sinistra della testina viene impedito.

Una *configurazione* di M è una descrizione istantanea della macchina tra una mossa e l'altra. Essa è definita da una stringa uqv dove $q \in Q$ denota lo stato corrente, u, v appartengono a $(\Gamma - \not{b})^*$ e uv è la porzione non blank del nastro, cioè la parola contenuta nelle celle iniziali fino al primo blank escluso. In particolare u rappresenta la stringa collocata a sinistra della cella scandita dalla testina, v è la parte rimanente e il primo simbolo di v è quello letto dalla macchina (con la convenzione che tale simbolo sia proprio $\not{b}$ se $v = \varepsilon$). La configurazione iniziale di M su input $x \in \Sigma^*$ è $q_0 x$. Invece una configurazione uqv è detta *di arresto* se $q \in \{q_s, q_n\}$; in particolare uqv è *accettante* se $q = q_s$, mentre è detta *di rifiuto* se $q = q_n$.

Possiamo quindi denotare con $\mathscr{C}_M$ l'insieme di tutte le configurazioni di M e rappresentare con $\vdash_M$ la relazione di transizione in un passo: per ogni $\alpha, \beta \in \mathscr{C}_M$, vale $\alpha \vdash_M \beta$ se $\alpha = uqv$, $q \not\in \{q_s, q_n\}$ e β è la configurazione ottenuta da α eseguendo la mossa $\delta(q, a)$, con a il primo simbolo di v (assumendo $a = \not{b}$ se $v = \varepsilon$). È chiaro che per ogni $\alpha \in \mathscr{C}_M$ esiste al più una

sola configurazione β tale che $\alpha \vdash_M \beta$; per questo la macchina è detta *deterministica*. Ovviamente, se $\alpha \in \mathscr{C}_M$ è di arresto, non esiste alcun β tale che $\alpha \vdash_M \beta$.

Denotiamo inoltre con $\vdash_M^*$ la chiusura riflessiva e transitiva di $\vdash_M$. Diciamo che una stringa $x \in \Sigma^*$ è *accettata* (o *riconosciuta*) da M se esiste una configurazione accettante $A \in \mathscr{C}_M$ tale che $q_0 x \vdash_M^* A$. Il linguaggio *accettato* (o *riconosciuto*) dalla macchina M è l'insieme $L(M) \subseteq \Sigma^*$ delle stringhe accettate da M:

$$L(M) = \{x \in \Sigma^* \mid \exists\, u, v \in \Gamma^* \text{ tali che } q_0 x \vdash_M^* u q_s v\}$$

Infine, diciamo che la macchina M si ferma (o si arresta) su un dato input $x \in \Sigma^*$ se esiste una configurazione di arresto $\alpha \in \mathscr{C}_M$ tale che $q_0 x \vdash_M^* \alpha$. In questo caso, la computazione di M su input x è la sequenza di configurazioni

$$\{C_0, C_1, \ldots, C_m\}$$

tali che $C_0 = q_0 x$, $C_m = \alpha$, e $C_{i-1} \vdash_M C_i$ per ogni $i = 1, \ldots, m$. Se invece non esiste alcuna configurazione di arresto α tale che $q_0 x \vdash_M^* \alpha$ allora diciamo che M non si ferma sull'input x; in questo caso la computazione è una sequenza infinita di configurazioni $\{C_i\}_{i \in \mathbb{N}}$ per le quali $C_0 = q_0 x$ e $C_{i-1} \vdash_M C_i$ per ogni $i \in \mathbb{N}_+$.

Nota che su ogni input $x \in \Sigma^*$ che non appartiene a $L(M)$ la macchina M si ferma in una configurazione di rifiuto oppure non si ferma.

È possibile dimostrare il seguente risultato (per la prova si può consultare [14] o [16]):

Proposizione 2.4.1. *Per ogni linguaggio $L \subseteq \Sigma^*$ le seguenti proprietà sono equivalenti:*

1. *esiste una MdT M che accetta L, ovvero tale che $L = L(M)$;*
2. *$L = \emptyset$ oppure esiste una MdT che su input ε stampa sul nastro tutti gli elementi di L;*
3. *esiste una grammatica G di tipo 0 che genera L, ovvero tale che $L = L(G)$.*

Ricordando le definizioni date nella sezione 2.2, questo significa che la famiglia dei linguaggi accettati dalle macchine di Turing coincide con l'insieme dei linguaggi *ricorsivamente numerabili*.

Invece, un linguaggio $L \subseteq \Sigma^*$ è detto *ricorsivo* se esiste una MdT M che si ferma su tutti gli input e che accetta L. Le proprietà dei linguaggi ricorsivi e ricorsivamente numerabili sono analoghe a quelle dei sottoinsiemi di $\mathbb{N}$ ricorsivi e ricorsivamente numerabili presentate nelle sezioni precedenti. In particolare, è chiaro che ogni linguaggio ricorsivo è anche ricorsivamente numerabile, mentre esistono linguaggi ricorsivamente numerabili che non sono ricorsivi. Inoltre, come accennato nella prossima sezione, le macchine di Turing possono essere considerate come sistemi di programmazione accettabile. Quindi i linguaggi ricorsivi sono intuitivamente quei linguaggi per i quali esiste un algoritmo (che termina sempre) in grado di verificare se la stringa data in input appartiene al linguaggio o meno.

2.4.1 Macchine di Turing e funzioni

Oltre che accettori, le macchine di Turing possono rappresentare modelli di calcolo per funzioni tra monoidi liberi (e quindi anche tra interi). Dato un alfabeto Σ e una funzione (parziale) $f : \Sigma^* \longrightarrow \Sigma^* \cup \{\perp\}$ diciamo che f è calcolata da una MdT M se Σ è l'alfabeto di ingresso di M e inoltre valgono le seguenti condizioni:

- per ogni $x \in \Sigma^*$ tale che $f(x) \neq \perp$ la macchina M su input x si ferma in una configurazione di arresto nella quale la stringa $f(x)$ è la porzione non blank del nastro;
- per ogni $x \in \Sigma^*$ tale che $f(x) = \perp$ la macchina M su input x non si ferma.

In maniera simile, per un qualsiasi intero $k > 0$, è possibile definire le funzioni $g : \mathbb{N}^k \longrightarrow \mathbb{N} \cup \{\perp\}$ calcolate dalle macchine di Turing. Si può poi dimostrare che la famiglia delle MdT che calcolano tali funzioni forma un sistema di programmazione accettabile, nozione che abbiamo trattato nella sezione 1.9. In altre parole, tali macchine si possono numerare in modo ricorsivo, calcolano tutte e sole le funzioni ricorsive parziali, ammettono una MdT universale in grado di simulare tutte le altre e soddisfano i teoremi $S_{m,n}$ di scambio automatico di indici e argomenti. Per questo motivo possiamo considerare le macchine di Turing come un modello per la definizione di algoritmi e per la classificazione dei problemi in base alle risorse di calcolo richieste per la loro soluzione. In questa sede non ci occuperemo quindi dei problemi di calcolabilità, già considerati nel capitolo 1, ma useremo le MdT come modelli per valutare la complessità di un algoritmo, ovvero il tempo e lo spazio richiesti per eseguire una computazione.

2.4.2 Macchine a più nastri

Finora abbiamo considerato un modello di macchina di Turing dotato di un solo nastro. Si può generalizzare facilmente tale nozione definendo una macchina che dispone di più nastri. In questo caso abbiamo una testina di lettura per ogni nastro e ogni mossa dipende da tutti i simboli letti dalle testine oltre che dallo stato corrente. Ad ogni mossa la macchina entra in un nuovo stato, stampa un nuovo simbolo in ogni cella letta da una testina e può muovere ognuna di queste di una posizione verso sinistra o verso destra (oppure ancora la lascia ferma). Inizialmente il primo nastro contiene la stringa di input con le solite convenzioni, mentre gli altri contengono solo il blank. La definizione di questo modello e le nozioni associate sono del tutto analoghe a quelle della macchina a un solo nastro e qui le ricordiamo solo per completezza.

Formalmente, una macchina di Turing a $k > 1$ nastri, è ancora una sestupla

$$M = (Q, q_0, \Gamma, \Sigma, \delta, \{q_s, q_n\})$$

dove Q, q_0, Γ, Σ e $\{q_s, q_n\}$ sono definiti come per le MdT tradizionali, mentre δ è una funzione transizione tale che

$$\delta : (Q - \{q_s, q_n\}) \times \Gamma^k \longrightarrow Q \times ((\Gamma - \{\not b\}) \times \{-1, 0, 1\})^k$$

Per ogni $q \in Q$ e ogni $\underline{a} = (a_1, a_2, \ldots, a_k) \in \Gamma^k$, il valore $\delta(q, \underline{a})$ definisce la mossa di M quando la macchina si trova nello stato q e legge il simbolo a_i sul nastro i-esimo, per ogni $i = 1, 2, \ldots, k$, con gli stessi criteri adottati nel modello a un solo nastro. In particolare, se $\delta(q, \underline{a}) = (p, (b_1, \ell_1), (b_2, \ell_2), \ldots, (b_k, \ell_k))$, allora $p \in Q$ rappresenta il nuovo stato della macchina mentre, per ogni $i = 1, 2, \ldots, k$, $b_i \in \Gamma$ è il simbolo stampato sul nastro i-esimo e $\ell_i \in \{-1, 0, 1\}$ rappresenta lo spostamento della relativa testina. Nota che ancora $\delta(q, \underline{a})$ non è definita se $q \in \{q_s, q_n\}$, mentre i valori b_i sono sempre diversi dal blank $\not b$, per ogni $i = 1, 2, \ldots, k$. Inoltre, anche in questo caso supponiamo che fin dall'inizio la macchina stampi sulla prima cella di ogni nastro solo simboli che impediscano di spostare la testina verso sinistra.

Come per le MdT a un nastro si può definire una nozione di configurazione della macchina M in un qualunque momento del calcolo; essa è rappresentata dallo stato corrente della macchina, dal contenuto della porzione non blank di ciascun nastro e dalla posizione di tutte le testine di lettura. Una configurazione è quindi formalmente definita come una k-pla di stringhe

$$C = (u_1 q v_1, u_2 \$ v_2, \ldots, u_k \$ v_k)$$

dove $q \in Q$ è lo stato corrente e \$ è un simbolo speciale di separazione (non incluso in Γ); inoltre, per ogni $i = 1, 2 \ldots, k$, $u_i v_i$ rappresenta la porzione non blank del nastro i-esimo, u_i è la stringa che si trova a sinistra della testina, il primo simbolo di v_i è quello letto dalla stessa testina (con la solita convenzione che $v_i = \varepsilon$ se questa legge il blank).

Denotiamo con $C_0(x)$ la configurazione iniziale di M su input $x \in \Sigma^*$, qui definita formalmente da $C_0(x) = (q_0 x, \$, \cdots, \$)$. Una configurazione è di arresto se lo stato corrispondente appartiene a $\{q_s, q_n\}$, è accettante se tale stato coincide con q_s mentre è di rifiuto se questo coincide con q_n. Le relazioni $\vdash_M$ e $\vdash_M^*$ di transizione in uno o più passi sono definite come nel caso a un solo nastro. Una stringa $x \in \Sigma^*$ è accettata da M se esiste una configurazione accettante A tale che $C_0(x) \vdash_M^* A$ e denotiamo con $L(M)$ il linguaggio di tutte le parole accettate:

$$L(M) = \{x \in \Sigma^* \mid C_0(x) \vdash_M^* A \text{ per qualche configurazione accettante } A\}$$

Diciamo che M si ferma su input $x \in \Sigma^*$ se esiste una configurazione di arresto α tale che $C_0(x) \vdash_M^* \alpha$; in questo caso la computazione di M su input x è la sequenza finita di configurazioni $\{C_0, C_1, \ldots, C_m\}$ tale che $C_0 = C_0(x)$, $C_m = \alpha$ e $C_{i-1} \vdash_M C_i$ per ogni $i = 1, \ldots, m$ (nota che in questa computazione M esegue esattamente m mosse). Se invece M non si ferma su input $x \in \Sigma^*$ allora la computazione della macchina è una sequenza infinita di configurazioni $\{C_i\}_{i \in \mathbb{N}}$, nella quale $C_0 = C_0(x)$ e $C_{n-1} \vdash_M C_n$ per ogni $n \geq 1$.

Anche in questo caso è bene sottolineare che una parola $w \in \Sigma^*$ non appartiene a $L(M)$ se e solo se, su input w, la macchina M si ferma in una configurazione di rifiuto oppure non si ferma.

Come vedremo meglio nella prossima sezione, ogni MdT a $k > 1$ nastri è simulabile da una macchina a un nastro solo (a patto si incrementare opportunamente il numero di mosse). Per questo motivo l'enunciato della Proposizione 2.4.1 rimane valido anche per le MdT a più nastri, e possiamo affermare che un linguaggio è ricorsivo se e solo se esso è accettato da una MdT dotata di un numero qualsiasi di nastri che si ferma su tutti gli input.

2.5 Complessità in tempo

Definiamo ora il tempo di calcolo di una macchina di Turing. Data una MdT a $k \geq 1$ nastri $M = (Q, q_0, \Gamma, \Sigma, \delta, \{q_s, q_n\})$, per ogni $x \in \Sigma^*$ denotiamo con $T_M(x)$ il numero di mosse compiute da M su input x; se M non si ferma

su tale input poniamo $T_M(x) = +\infty$. Inoltre, per ogni $n \in \mathbb{N}$, denotiamo con $T_M(n)$ il massimo numero di mosse compiute da M su un input di lunghezza n, ovvero:

$$T_M(n) = \max\{T_M(x) \mid x \in \Sigma^*, |x| = n\}$$

Diremo che la funzione $T_M : \mathbb{N} \longrightarrow \mathbb{N} \cup \{+\infty\}$ rappresenta la complessità in tempo di M.

Per esempio, il linguaggio $L \subsetneq \{0,1\}^*$ delle stringhe binarie che rappresentano numeri pari può essere accettato da una MdT M a un nastro tale che $T_M(n) = n+1$ per tutti gli $n \in \mathbb{N}$. Infatti, su input $x \in \{0,1\}^+$, M rifiuta subito se x inizia con 0 e $|x| > 1$; altrimenti legge tutta la stringa in ingresso e accetta se l'ultimo simbolo è 0 (in caso contrario rifiuta). È evidente che il numero di passi compiuto su un input di lunghezza n (nel caso peggiore) è proprio $n+1$.

In generale è facile verificare che ogni linguaggio regolare può essere accettato da una MdT M a un nastro tale che $T_M(n) = n+1$ per ogni $n \in \mathbb{N}$, basta infatti simulare un automa (deterministico) che riconosce il linguaggio. Osserva anche se $T_M(n) \in \mathbb{N}$ per tutti gli $n \in \mathbb{N}$ allora $L(M)$ è per forza ricorsivo. Inoltre, per svincolare la nozione di tempo di calcolo dai valori interi, introduciamo la seguente definizione.

Definizione 2.5.1. *Data una funzione a valori reali positivi $f : \mathbb{N} \longrightarrow \mathbb{R}_+$, diciamo che una MdT a $k \geq 1$ nastri M* funziona *(o* lavora*) in tempo $f(n)$ se $T_M(n) \leq f(n)$ per ogni $n \in \mathbb{N}$. Inoltre diciamo che un linguaggio $L \subseteq \Sigma^*$ è* riconoscibile in tempo $f(n)$ *se esiste una MdT a $k \geq 1$ nastri che riconosce L e funziona in tempo $f(n)$.*

Una prima proprietà riguarda il numero di nastri usati nella computazione. Infatti il tempo necessario per riconoscere un dato linguaggio può dipendere dal numero di nastri di cui la macchina dispone.

Esempio 2.5.1. Consideriamo il linguaggio delle parole palindrome $L = \{x \in \{a,b\}^* \mid x = x^R\}$, dove x^R è l'inversa di x. Si può facilmente descrivere una MdT a un nastro che riconosce L in tempo $O(n^2)$. Tale macchina, su un input di lunghezza n, confronta i simboli di posizione i e $n-i+1$, per ogni $i = 1,2,\ldots,n$, e accetta se questi sono uguali fra loro per tutti i valori di i. Questa operazione può essere eseguita scorrendo l'input avanti e indietro n volte e marcando in modo opportuno i vari simboli. Per questo motivo la macchina funziona in tempo $O(n^2)$. Invece, usando una MdT a 2 nastri, è possibile riconoscere L in un numero di passi $O(n)$. In questo caso

la macchina può copiare l'input sul secondo nastro, riposizionare la testina del nastro di ingresso sulla prima cella, confrontare uno dopo l'altro i simboli letti dalle due testine spostando sempre la prima verso destra e la seconda verso sinistra. È evidente che in questo modo la MdT riconosce L in $3n + c$ passi, per una costante c opportuna.

L'esempio precedente mostra come una MdT a 2 nastri possa essere simulata da una macchina a un solo nastro a patto di rendere quadratico il numero di mosse compiute. La seguente proposizione prova che questa proprietà è del tutto generale.

Proposizione 2.5.1. *Se un linguaggio L è riconoscibile in tempo $f(n)$ da una MdT a $k > 1$ nastri, con $f(n) \geq n+1$ per ogni $n \in \mathbb{N}$, allora L è riconoscibile da una MdT a un nastro in un tempo $O(f(n)^2)$.*

Dimostrazione. Sia M una MdT a k nastri che riconosce L in tempo $f(n)$ e sia Γ il suo alfabeto di lavoro. Possiamo definire un nuovo alfabeto di lavoro $\Gamma' = (\Gamma \times \{0,1\})^k$. Così, in maniera del tutto ovvia, una parola $w \in \Gamma'^*$ può descrivere il contenuto delle porzioni non blank dei k nastri di M e la posizione delle corrispondenti testine di lettura. Infatti se $w_j = ((a_1, \ell_1), \ldots, (a_k, \ell_k))$ è la j-esima lettera di w, allora ogni a_i è il contenuto della j-esima cella nel nastro i-esimo, mentre $\ell_i = 1$ se e solo se su tale cella è posizionata la relativa testina.

Definiamo ora una nuova MdT M' a un solo nastro, avente $\Gamma \cup \Gamma'$ come alfabeto di lavoro, che simula M. La computazione di M' su un generico input è suddivisa in fasi, una per ogni mossa della macchina M. In ogni fase M' scorre la porzione non blank del suo nastro dalla prima all'ultima posizione e riporta quindi la testina sulla prima cella. Durante questa doppia passata la macchina aggiorna opportunamente il contenuto del nastro in modo da simulare un passo di M: nella prima scansione si aggiornano le componenti corrispondenti agli spostamenti delle testine di M verso destra, mentre nella seconda si aggiornano quelli relativi agli spostamenti verso sinistra. Al termine di ogni fase, M' avrà memorizzato nello stato il simbolo letto da ciascuna testina di M in modo da determinare la mossa da simulare nella fase successiva (cioè i nuovi simboli da stampare e gli spostamenti delle testine). Inoltre, nella prima fase M' trasforma l'input (appartenente a Γ^*) nella corrispondente stringa in Γ'^*.

Poiché M in $f(n)$ passi può visitare al più $f(n) + 1$ celle su ogni nastro, ogni fase di M' su un input di dimensione n può essere eseguita in $2f(n)+1$ passi. Ne segue che il numero totale di mosse compiute da M' è al più uguale a $2f(n)^2 + f(n) = O(f(n)^2)$. □

La proposizione precedente mostra come il funzionamento di MdT a più nastri possa essere simulato da una macchina a un solo nastro, senza un incremento eccessivo dei tempi di calcolo. In particolare, se la prima macchina funziona in tempo polinomiale anche quella a un solo nastro richiede al più un tempo limitato da un polinomio.

Una seconda proprietà della complessità in tempo delle MdT riguarda la costante principale dei tempi di calcolo che, nei casi significativi, può essere ridotta a piacere a patto di ampliare opportunamente l'alfabeto di lavoro ed eventualmente il numero degli stati. Questa proprietà di "accelerazione" (o "speed up") è tipica di diversi tipi di automi e modelli di calcolo.

Proposizione 2.5.2. *Data una funzione* $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ *tale che* $\lim_{n\to+\infty} \frac{f(n)}{n} = +\infty$*, sia* L *un linguaggio riconoscibile in tempo* $f(n)$*. Allora, per ogni* $c > 0$*,* L *può essere riconosciuto in tempo* $g(n) = \max\{n+1, cf(n)\}$*.*

Dimostrazione. Chiaramente l'enunciato è significativo per valori positivi c minori di 1. Consideriamo una macchina di Turing M a $k > 1$ nastri che riconosce L, sia Γ il suo alfabeto di lavoro e sia m un intero positivo qualsiasi. Consideriamo un nuovo alfabeto Γ' che codifica le stringhe di lunghezza m in Γ^*. In questo modo ogni simbolo di Γ' rappresenta una m-pla di simboli di Γ e viceversa.

Definiamo ora una nuova MdT a k nastri M' che simula M e utilizza $\Gamma \cup \Gamma'$ come alfabeto di lavoro. Innanzitutto M' legge l'input trascrivendo sul secondo nastro la codifica della stringa di input nel nuovo alfabeto Γ'. Quindi la macchina considera il secondo nastro come nastro di ingresso e usa il primo come nastro di lavoro. M' riporta la testina del secondo nastro nella posizione iniziale e comincia la simulazione di M senza più riutilizzare le celle del primo nastro occupate dall'input originale. Ogni ciclo di m mosse consecutive di M può essere simulato da M', mediante al più 6 passi di calcolo, nel modo seguente. A ogni ciclo la macchina M può visitare al più m celle consecutive su ogni nastro. Quindi, per simulare il suo comportamento, M' legge su ogni nastro la cella che precede e quella che segue la posizione scandita dalla testina (è evidente che 4 passi sono sufficienti per eseguire questa operazione). A questo punto, conoscendo la funzione transizione di M, M' può raggiungere direttamente la configurazione desiderata mediante al più altri due passi, completando così la simulazione del ciclo. Durante il calcolo, per ogni simbolo $a \in \Gamma'$ letto, M' può memorizzare nello stato la posizione della testina di M all'interno del-

la m-pla corrispondente ad a. In questo modo, simulando M, la macchina M' riconosce esattamente il linguaggio L.

Il numero totale di passi eseguiti da M' su un input di lunghezza n è

$$T_{M'}(n) \leq n+1+\lceil n/m \rceil + 6\lceil f(n)/m \rceil \leq n + \frac{n}{m} + 8 + \frac{6}{m} f(n)$$

Poiché per n abbastanza grande $n + \frac{n}{m} + 8 \leq \frac{f(n)}{m}$, esiste un $t \in \mathbb{N}$ tale che per ogni $n \geq t$,

$$T_{M'}(n) < \frac{7}{m} f(n)$$

Ora, per ogni $c > 0$ possiamo scegliere $m = \lceil 7/c \rceil$, ottenendo così $T_{M'}(n) < cf(n)$ per ogni $n \geq t$. Inoltre, si può modificare M' permettendo alla macchina di riconoscere le parole in L di lunghezza minore di t durante la prima fase di lettura e trascrizione dell'input, usando solo le transizioni tra stati. In questo modo, risulta $T_{M'}(n) = n+1$ per ogni $n < t$ e quindi otteniamo l'asserto. □

Usando un ragionamento simile al precedente si può estendere la proprietà appena considerata ai linguaggi riconosciuti in tempo lineare. A tal proposito enunciamo qui la seguente proposizione rimandando a [14] la relativa dimostrazione.

Proposizione 2.5.3. *Per ogni costante $a > 1$, se L è un linguaggio riconoscibile in tempo an allora, per ogni $\epsilon > 0$, L può essere riconosciuto in tempo $(1+\epsilon)n$.*

Le nozioni e le proprietà definite finora ci permettono di suddividere e quindi classificare i linguaggi ricorsivi in base al tempo necessario per riconoscerli mediante una MdT (deterministica). Per ogni funzione $f : \mathbb{N} \longrightarrow \mathbb{R}_+$, possiamo definire DTIME($f(n)$) come la classe dei linguaggi riconoscibili in un tempo definitivamente minore o uguale a $f(n)$:

$$\text{DTIME}(f(n)) = \{L : \text{esistono una MdT } M \text{ a } k \geq 1 \text{ nastri e un intero } n_0 \in \mathbb{N} \text{ tali che } L = L(M) \text{ e } T_M(n) \leq f(n) \text{ per ogni } n \geq n_0\}$$

Applicando le proposizioni 2.5.2 e 2.5.3 è ora facile provare che, per tutte le funzioni f naturali, la classe di complessità DTIME($f(n)$) dipende solo dall'ordine di grandezza di $f(n)$ e non dalla sua costante principale. Più precisamente valgono le seguenti proprietà.

Corollario 2.5.4. *Per ogni $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ tale che $\lim_{n\to+\infty} \frac{f(n)}{n} = +\infty$, vale l'inclusione*

$$DTIME(f(n)) \subseteq DTIME(cf(n))$$

per ogni $c > 0$.

Inoltre, per ogni costante $a > 1$*, vale l'inclusione*

$$DTIME(an) \subseteq DTIME((1+\epsilon)n)$$

per ogni $\epsilon > 0$.

Queste proprietà giustificano anche la tradizionale analisi degli algoritmi, nella quale si cerca in generale di determinare l'ordine di grandezza del tempo di calcolo richiesto da una procedura su input di lunghezza n (nel caso peggiore o nel caso medio) e non la sua espressione asintotica.

2.5.1 La classe P e la tesi di Church estesa

Una classe di particolare interesse è la classe **P** dei linguaggi riconoscibili in tempo polinomiale. Tale classe è quindi definita nel modo seguente:

$$\begin{aligned}\mathbf{P} \quad = \quad & \{L : \text{ esiste un polinomio } p(x) \text{ e una MdT } M \text{ a } k \geq 1 \text{ nastri tali che} \\ & \quad L = L(M) \text{ e } T_M(n) \leq p(n) \text{ per ogni } n \in \mathbb{N}\}\end{aligned}$$

Tenendo conto delle nozioni illustrate nella sezione precedente, è facile verificare che

$$\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k)$$

Notiamo anche che per la proposizione 2.5.1 la classe appena definita non dipende dal numero di nastri delle MdT considerate. Potremmo cioè limitarci a considerare le MdT a un solo nastro; in altre parole **P** può essere definita come la classe dei linguaggi riconoscibili da una MdT a un nastro in un tempo $O(n^k)$ per un qualche $k \in \mathbb{N}$. Come vedremo meglio in seguito, tale classe rimane invariata anche usando altri modelli di calcolo. Intuitivamente possiamo quindi affermare che l'appartenenza o meno di un linguaggio L alla classe **P** dipende dalla complessità intrinseca di L e non dal modello di calcolo usato per riconoscerlo.

La classe **P** può essere considerata più realisticamente come una famiglia di problemi di decisione piuttosto che un insieme di linguaggi. In questo modo essa viene a rappresentare una classe di problemi significativi anche da un punto di vista pratico.

Estendendo la nozione introdotta nella sezione 1.10, possiamo definire un problema di decisione come una coppia $\pi = (I, p)$, dove I è un insieme numerabile di elementi che chiamiamo *istanze* e che possiamo

codificare in maniera univoca e semplice con parole su un dato alfabeto Σ, mentre p è una funzione $p : I \longrightarrow \{0,1\}$, che chiameremo *predicato*, e che stabilisce intuitivamente se ciascuna istanza gode di una certa proprietà o meno. Si conviene che un'istanza $x \in I$ ammetta risposta positiva se $p(x) = 1$, negativa se $p(x) = 0$. Come al solito si usa definire un problema di decisione mettendo in rilievo le singole istanze ed esprimendo il predicato corrispondente mediante una domanda esplicita. Come esempio possiamo considerare i seguenti problemi di decisione che studieremo meglio nelle sezioni successive. Qui e nel seguito, per ogni insieme finito A, denoteremo con $\sharp A$ il numero dei suoi elementi.

Problema *Raggiungibilità*

Istanza: un grafo orientato $G = (V, E)$ e due nodi distinti $u, v \in V$.

Domanda: esiste in G un cammino da u a v?

Problema *Soddisfacibilità*

Istanza: una formula booleana ϕ su un insieme X di variabili.

Domanda: esiste una assegnamento di valori 0 e 1 alle variabili in X che rende vera ϕ?

Problema *Clique*

Istanza: un grafo non orientato $G = (V, E)$ e un intero $k \leq \sharp V$.

Domanda: esiste una clique di dimensione k in G (ovvero un insieme $C \subseteq V$ di cardinalità k tale che $\{v, w\} \in E$, per ogni coppia di nodi distinti $v, w \in C$) ?

Problema *Circuito hamiltoniano*

Istanza: un grafo non orientato $G = (V, E)$.

Domanda: esiste un circuito hamiltoniano in G (ovvero una permutazione $(v_1, v_2, \ldots, v_n)$ degli elementi di V tale che $\{v_n, v_1\} \in E$ e $\{v_i, v_{i+1}\} \in E$ per ogni $i = 1, \ldots, n-1$)?

Problema *Commesso viaggiatore*

Istanza: un grafo completo $G = (V, E)$, una distanza $d(v, w) \in \mathbb{N}$ per ogni coppia di nodi distinti $v, w \in V$ e un intero $k > 0$.

Domanda: esiste un ciclo che congiunge tutti i nodi del grafo di lunghezza minore o uguale a k (ovvero una permutazione $(v_1, v_2, \ldots, v_n)$ degli elementi di V tale che $\sum_{i=1}^{n-1} d(v_i, v_{i+1}) + d(v_n, v_1) \leq k$)?

Una MdT che riconosce un dato linguaggio può essere interpretata facilmente come procedura o algoritmo per risolvere un problema di decisione. A tale scopo, d'ora in poi, identifichiamo per semplicità le istanze di ogni problema di decisione con le parole dell'alfabeto che le codificano. Formalmente quindi, dato un problema di decisione $\pi = (I, p)$, con $I \subseteq \Sigma^*$ per un opportuno alfabeto Σ, diciamo che una MdT M *risolve* π se M possiede Σ come alfabeto d'ingresso, $L(M) = \{x \in I : p(x) = 1\}$ e M si ferma su tutti gli input. Quindi risolvere un problema di decisione equivale a riconoscere l'insieme delle istanze che ammettono risposta positiva, fermandosi sempre su tutti gli input. Viceversa, riconoscere un linguaggio $L \subseteq \Sigma^*$ fermandosi su tutti gli input, equivale a risolvere il problema di appartenenza a L, ovvero il problema di decisione (Σ^*, χ_L), dove $\chi_L : \Sigma^* \longrightarrow \{0,1\}$ è la funzione caratteristica di L. Le due formulazioni sono dunque equivalenti. Tuttavia d'ora in poi preferiamo in linea di massima riferire il comportamento di una MdT a un problema di decisione perché spesso quest'ultimo può rappresentare una tematica interessante di per sè e la sua complessità computazionale può essere più naturalmente confrontata con quella di problemi con caratteristiche simili. Così le varie classi di linguaggi DTIME$(f(n))$ possono essere considerate più propriamente come famiglie di problemi di decisione. Di conseguenza, anche la classe **P** appena introdotta può essere definita come la classe dei problemi di decisione risolubili in tempo polinomiale da una macchina di Turing deterministica. Per esempio è noto che il problema *Raggiungibilità* sopra definito appartiene a **P**. Infatti, i tradizionali metodi di esplorazione di un grafo sono in grado di verificare, in tempo polinomiale, se in un dato grafo orientato esiste un cammino da un nodo a un altro [1, 6].

Ricordiamo infine la proprietà fondamentale della classe **P**. Essa è infatti considerata come la famiglia dei problemi di decisione risolubili in modo efficiente. Questo fatto è dovuto principalmente a due motivi fondamentali. Il primo è che tutti i modelli di calcolo comunemente usati per definire la nozione di algoritmo (per esempio le macchine di Turing deterministiche a uno o più nastri, le RAM con criterio di costo logaritmico [1], le famiglie di circuiti booleani con un numero di porte polinomiale [19]) sono tra di loro polinomialmente equivalenti per quanto riguarda il tempo

di calcolo. Più precisamente si può dimostrare che la famiglia dei problemi risolubili in tempo polinomiale rimane la stessa anche cambiando il modello di calcolo [1, 16]. Si dice anche che la classe **P** è *robusta* rispetto al modello di calcolo utilizzato per definirla. Inoltre, l'esperienza ha dimostrato che intuitivamente gli algoritmi che funzionano in tempo polinomiale, se il grado del polinomio non è troppo elevato, possono funzionare in tempi accettabili per lo meno su input di dimensioni ragionevoli.

Viceversa, ed è questa la seconda ragione cui accennavamo prima, gli algoritmi che funzionano in tempo esponenziale sono di fatto inutilizzabili anche per dimensioni di input di poche decine di unità, poiché i tempi di calcolo risultano solitamente troppo elevati. Questo è un fatto principalmente sperimentale, ormai acquisito dall'esperienza, che però ha anche semplici spiegazioni formali basate sull'analisi del tempo richiesto da computazioni eseguite su computer ideali, estremamente veloci, che per ipotesi possono compiere un gran numero di operazioni nella singola unità di tempo (vedi ad esempio [4], [1] oppure [11, Capitolo 2]).

Queste considerazioni portano ad enunciare la seguente legge generale che, come la tesi di Church illustrata nella sezione 1.8.1, viene comunemente accettata anche se si tratta di una proprietà intuitiva che rimane indimostrabile in generale.

Tesi di Church estesa

La famiglia dei problemi di decisione che ammettono algoritmi di soluzione efficienti coincide con la classe **P** dei problemi di decisione risolubili in tempo polinomiale da una macchina di Turing deterministica.

Questa proprietà viene solitamente estesa alle funzioni in generale. Dati due alfabeti Σ e Γ, una funzione $f : \Sigma^* \longrightarrow \Gamma^*$ è generalmente considerata *calcolabile in modo efficiente* se esiste una MdT deterministica che la calcola funzionando in tempo polinomiale.

2.6 Macchine di Turing non deterministiche

Come nel caso degli automi a stati finiti, anche per le macchine di Turing è possibile definire una versione non deterministica, nella quale ad ogni passo la macchina sceglie la mossa da compiere in un insieme finito di possibiltà. Si tratta di un modello di calcolo usato principalmente per riconoscere linguaggi e classificare problemi di decisione. In particolare permet-

te di definire la classe **NP** e i problemi NP-completi, argomenti principali studiati nelle sezioni successive.

Una Macchina di Turing non deterministica possiede gli stessi dispositivi di una MdT deterministica tradizionale a un nastro, nella quale le varie nozioni sono definite nello stesso modo, con l'unica eccezione della funzione transizione che, nel caso non deterministico, associa ad ogni stato e a ogni simbolo letto un insieme finito di possibili mosse tra le quali la macchina sceglie il passo effettivo da eseguire. Così, a ogni configurazione corrisponde un insieme finito di configurazioni successive e, a ogni input, corrisponde in generale un insieme di computazioni, una per ogni possibile successione di scelte compiute dalla macchina a partire dalla configurazione iniziale. La stringa di input è accettata se tra tutte queste possibili computazioni ne esiste una che raggiunge lo stato accettante; viene invece rifiutata se tutte le computazioni si fermano in uno stato non accettante oppure non terminano. Intuitivamente quindi la stringa è accettata se la macchina può "indovinare" una sequenza di mosse che la portano nello stato accettante.

Formalmente una macchina di Turing non deterministica (a un nastro) è un sestupla $M = (Q, q_0, \Gamma, \Sigma, \delta, q_s)$, dove Q, q_0, Γ, Σ sono definite come nel caso deterministico (sezione 2.4), $q_s \in Q$ è lo stato di accettazione e δ è una funzione

$$\delta : Q \times \Gamma \longrightarrow 2^{Q \times (\Gamma - \not b) \times \{-1,0,+1\}}$$

tale che $\delta(q_s, a) = \emptyset$ per ogni $a \in \Gamma$. In maniera analoga a prima, per ogni $q \in Q$ e ogni $a \in \Gamma$, ciascuna tripla $(p, b, \ell) \in \delta(q, a)$ definisce una possibile mossa di M quando la macchina si trova nello stato q e legge il simbolo a.

Di nuovo, una configurazione di M si definisce come nel caso deterministico. Anche qui $q_0 x$ è la configurazione iniziale su input x; denotiamo ancora con $\mathscr{C}_M$, $\vdash_M$ e $\vdash_M^*$, rispettivamente, la famiglia di tutte le configurazioni di M, la relazione di transizione in un passo e la sua chiusura riflessiva e transitiva. Una configurazione $A = uqv \in \mathscr{C}_M$ è *accettante* se $q = q_s$. Invece, la configurazione $A = uqv$ è detta *di arresto* se $\delta(q, a) = \emptyset$, dove a è il primo simbolo di v (oppure coincide con $\not b$ se $v = \varepsilon$). Nota che ogni configurazione accettante è anche di arresto. Viceversa, una configurazione di arresto uqv è detta *di rifiuto* se $q \neq q_s$. Inoltre, per ogni configurazione $A = uqv \in \mathscr{C}_M$ nella quale M legge il simbolo a sul nastro, l'insieme delle configurazioni successive $\{B \in \mathscr{C}_M : A \vdash_M B\}$ possiede la stessa cardinalità di $\delta(q, a)$.

Una parola $x \in \Sigma^*$ è accettata da M se esiste una configurazione accettante $A \in \mathscr{C}_M$ tale che $q_0 x \vdash_M^* A$. Il linguaggio *accettato* (o *riconosciuto*) da M è l'insieme $L(M)$ delle parole accettate dalla macchina:

$$L(M) = \{x \in \Sigma^* \mid \text{ esistono } u, v \in \Gamma^* \text{ tali che } q_0 x \vdash_M^* u q_s v\}$$

Una notevole differenza rispetto alla versione deterministica è che ora, per ogni input, vi sono tante (eventualmente infinite) computazioni di M su x, ciascuna delle quali potrebbe essere composta da infinite configurazioni.

Un modo conveniente per rappresentare tutte le computazioni di una macchina non deterministica su un dato input, è quello di considerare un albero, detto *albero di computazione*, nel quale ogni cammino che parte dalla radice e termina in una foglia (o non termina) rappresenta una possibile computazione della macchina sull'input considerato. Formalmente, per ogni input $x \in \Sigma^*$, l'albero di computazione di M su x è un albero con radice [1], dotato eventualmente di un numero infinito di nodi, che gode delle seguenti proprietà:

(i) ogni nodo è etichettato da una configurazione di M;

(ii) la radice è etichettata dalla configurazione iniziale $q_0 x$;

(iii) se un nodo v è etichettato da una configurazione C e $\{C_1, C_2, \ldots, C_m\} = \{\beta \in \mathscr{C}_M \mid C \vdash_M \beta\}$ è l'insieme delle configurazioni successive, allora v possiede m figli $v_1, v_2, \ldots, v_m$ tali che v_i è etichettato da C_i per ogni $i = 1, 2, \ldots, m$;

(iv) un nodo v è una foglia (cioè un nodo senza figli) se e solo se v è etichettato da una configurazione di arresto.

Dalla definizione precedente è evidente che una computazione di M su un dato input è determinata dalla sequenza di etichette dei nodi che formano un cammino che parte dalla radice e scende lungo l'albero passando di padre in figlio fino ad arrivare eventualmente a una foglia (o percorrere un ramo infinito dell'albero). Chiaramente una parola $x \in \Sigma^*$ è accettata dalla macchina se e solo se esiste, nel corrispondente albero di computazione, una foglia etichettata con una configurazione accettante.

Nota che in ogni albero di computazione di M il numero dei figli di ciascun nodo interno è al più dato dal valore $d_M = \max\{\sharp\delta(q, a) : q \in Q, a \in \Gamma\}$, che dipende dalla macchina (ma non dall'input) ed è chiamato *grado di ambiguità* di M. In generale d_M rappresenta il massimo numero di scelte che la macchina M può compiere ad ogni passo. Chiaramente se $d_M = 1$ la macchina M è di fatto deterministica e ogni albero di computazione si riduce a un unico ramo che parte dalla radice, nel quale tutti i nodi interni

[1] Per una definizione formale degli alberi con radice si può consultare [4].

hanno un solo figlio. In questo caso, chiaramente, la macchina si ferma sull'input se e solo se il ramo è finito (e la configurazione di arresto è quella della foglia).

In modo del tutto analogo si possono definire le macchine di Turing non deterministiche a più nastri. L'estensione è del tutto simile a quella illustrata nella sezione 2.4.2 e viene qui omessa.

Definiamo ora il tempo di calcolo di una macchina di Turing non deterministica. Data una MdT non deterministica $M = (Q, q_0, \Gamma, \Sigma, \delta, F)$ e una stringa $w \in \Sigma^*$, denotiamo con $T_M(w)$ il massimo numero di mosse che la macchina può compiere in una computazione su input w. In altre parole $T_M(w)$ rappresenta l'altezza dell'albero di computazione di M su input w (ovvero la massima distanza della radice da una foglia). Chiaramente, $T_M(w) = +\infty$ se (e solo se) esiste una computazione di M su input w che non si arresta.

Inoltre, per ogni $n \in \mathbb{N}$, denotiamo con $T_M(n)$ il massimo valore di $T_M(w)$ al variare delle parole $w \in \Sigma^*$ di lunghezza n:

$$T_M(n) = \max\{T_M(w) : w \in \Sigma^*, |w| = n\}$$

Di nuovo, data una funzione $f : \mathbb{N} \longrightarrow \mathbb{R}_+$, diremo che una MdT non deterministica M *funziona* (o *lavora*) in tempo $f(n)$ se $T_M(n) \leq f(n)$ per ogni $n \in \mathbb{N}$. Possiamo quindi definire la classe NTIME$(f(n))$ come la classe dei linguaggi riconoscibili da una MdT non deterministica che lavora in un tempo definitivamente minore o uguale a $f(n)$:

$$\text{NTIME}(f(n)) = \{L : \begin{array}{l}\text{esistono una MdT } M \text{ non deterministica e un intero } n_0 > 0 \\ \text{tali che} \\ \quad L = L(M) \text{ e } T_M(n) \leq f(n) \text{ per ogni } n \geq n_0\}\end{array}$$

Anche le MdT non deterministiche soddisfano le proposizioni 2.5.1 e 2.5.2 dimostrate per il modello deterministico. Quindi, anche nel caso non deterministico, possiamo sostanzialmente ricondurci ad un solo nastro e le classi di complessità in tempo dipendono solo dall'ordine di grandezza delle funzioni coinvolte e non dalle costanti principali.

Corollario 2.6.1. *Per ogni $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ tale che $\lim_{n\to+\infty} \frac{f(n)}{n} = +\infty$, vale l'inclusione*

$$NTIME(f(n)) \subseteq NTIME(cf(n))$$

per ogni $c > 0$.

Inoltre, per ogni costante $a > 1$*, vale l'inclusione*

$$NTIME(an) \subseteq NTIME((1+\epsilon)n)$$

per ogni $\epsilon > 0$.

2.7 Confronto tra le classi DTIME e NTIME

Vogliamo ora confrontare le classi di complessità DTIME($f(n)$) e NTIME($f(n)$) per ogni funzione f. È evidente che la prima classe è contenuta nella seconda perché ogni macchina deterministica è una particolare macchina non deterministica. Di conseguenza, per ogni $f : \mathbb{N} \longrightarrow \mathbb{R}_+$, abbiamo

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$$

Viceversa si può simulare una MdT non deterministica mediante una deterministica, a patto di incrementare opportunamente i tempi di calcolo.

Proposizione 2.7.1. *Per ogni funzione* $f : \mathbb{N} \longrightarrow \mathbb{R}_+$ *tale che* $f(n) \geq n+1$ *per ciascun* $n \in \mathbb{N}$ *e ogni linguaggio* $L \in NTIME(f(n))$*, esiste una costante* $c > 1$ *tale che* $L \in DTIME(c^{f(n)})$.

Dimostrazione. Sia N una MdT non deterministica che riconosce L in tempo $f(n)$ con insieme degli stati Q, alfabeto di ingresso Σ e di lavoro Γ. L'idea è quella di simulare N mediante una MdT deterministica D che, su input x, esplora l'albero di computazione di N su x per verificare se tra le etichette delle foglie si trova una configurazione accettante. La visita potrebbe essere eseguita in profondità o in ampiezza [1, 4, 6], tuttavia si preferisce qui adottare un metodo particolare (una sorta di miscela dei due classici tipi di visita) che consente di limitare lo spazio richiesto per il calcolo.

Osserva che l'altezza dell'albero di computazione di N su x è minore o uguale a $f(|x|)$. Per semplicità denotiamo ora con d il valore d_N, cioè il grado di ambiguità di N. Esso rappresenta il massimo numero di scelte che N può compiere ad ogni passo. Questo significa che ogni nodo interno dell'albero possiede al più d figli. Così una computazione di N su x (ovvero un cammino dalla radice a una foglia) può essere rappresentata da una sequenza di interi $(c_1, c_2, \ldots, c_m)$ dove $m \leq f(|x|)$ è la lunghezza del cammino e ogni $c_i \in \{1, 2, \ldots, d\}$ denota la mossa scelta da N al passo i-esimo. È chiaro che i primi t passi di questa computazione, per ogni $t = 1, \ldots, m$, sono rappresentati da $(c_1, c_2, \ldots, c_t)$. Allora la macchina D, senza conoscere il

valore di $f(|x|)$, può generare sistematicamente le sequenze $(c_1, c_2, \ldots, c_t)$, con ciascun $c_i \in \{1, 2, \ldots, d\}$, al crescere di t da 1 in poi. Per ciascuna di queste, D esegue la corrispondente computazione parziale di N su x (di lunghezza t) scegliendo ad ogni passo la mossa c_i-esima. In questo processo D si ferma quando incontra una configurazione accettante di N (e allora anche D accetta) oppure quando trova un t per il quale tutte le computazioni eseguite di lunghezza minore o uguale a t sono terminate in una configurazione di arresto non accettante (e allora D rifiuta). L'algoritmo eseguito da D su input x può essere descritto dalla seguente procedura nella quale l'intero d è noto alla macchina e non dipende da x.

```
begin
  Sia C_0(x) la configurazione iniziale di N su input x
  esci = 0
  t = 1
  A = 0
  while esci = 0 do
        begin
              continua = 0
              for (c_1, c_2, ..., c_t) ∈ {1, 2, ..., d}^t do
                  begin
                        entra nella configurazione C_0(x)
                        simula N per t passi scegliendo al passo i-esimo
                          la mossa c_i-esima, per ciascun i = 1, ..., t
                          e sia α ∈ 𝒞_N la configurazione raggiunta
                        if α accettante then { A = 1
                                             { esci = 1
                        if α non è di arresto then continua = 1
                  end
              if continua = 1 then t = t + 1
                                 else esci = 1
        end
  if A = 1 then accetta
           else rifiuta
end
```

La macchina D può eseguire la procedura precedente usando lo stesso numero di nastri di N per eseguire la simulazione, più altri due nastri per mantenere rispettivamente la sequenza $(c_1, c_2, \ldots, c_t)$ corrente e l'input x.

Posto $n = |x|$ il numero totale delle computazioni parziali eseguite è al più

$$\sum_{t=1}^{f(n)} d^t = O(d^{f(n)+1})$$

Ogni computazione parziale richiede al più $O(f(n))$ passi per essere eseguita, per aggiornare la sequenza $(c_1, c_2, \ldots, c_t)$ corrente e per ricopiare l'input sul primo nastro quando necessario (sfruttiamo qui l'ipotesi $f(n) \geq n+1$ per ogni $n \in \mathbb{N}$). In totale il tempo richiesto è $O(f(n)d^{f(n)}) \leq c^{f(n)}$ per un opportuno $c > 1$. Nota che usando la Proposizione 2.5.1 possiamo trasfomare D in una MdT deterministica a un nastro che riconosce sempre lo stesso linguaggio L in un tempo $O(c^{2f(n)})$. □

Osserviamo che, prescindendo dal tempo di calcolo, il ragionamento precedente consente di provare che i due modelli di macchina *accettano* la stessa famiglia di linguaggi (cioè gli insiemi ricorsivamente numerabili). Inoltre, usando la stessa simulazione, è facile provare che un linguaggio L è ricorsivo se e solo se è accettato da una MdT non deterministica che ammette solo computazioni finite su tutti gli input.

2.8 La classe NP

Possiamo ora definire la classe **NP** dei linguaggi riconoscibili in tempo polinomiale da MdT non deterministiche :

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

Come per le MdT deterministiche, possiamo interpretare questo insieme come una classe di problemi di decisione. Infatti, anche per le MdT non deterministiche possiamo definire il comportamento di una macchina mediante problemi di decisione. Diciamo che un problema di decisione $\pi = (I, q)$ con $I \subseteq \Sigma^*$ è risolto da una MdT non deterministica M se $L(M) = \{x \in I \mid q(x) = 1\}$ e, per ogni input $x \in \Sigma^*$, l'altezza dell'albero di computazione di M su x è finita.

Quindi un problema di decisione $\pi = (I, q)$ appartiene alla classe **NP** se e solo se esistono un polinomio $p(y)$ e una MdT non deterministica M che risolve π tale che $T_M(n) \leq p(n)$ per ogni $n \in \mathbb{N}$.

La classe **NP** contiene classici problemi di decisione su grafi. Tra questi ricordiamo il problema della clique e quello del circuito hamiltoniano che abbiamo già incontrati nella sezione 2.5.1. Dato un grafo non orientato

$G = \langle V, E \rangle$ dotato di n nodi, una clique di G è un insieme $C \subseteq V$ tale che, per ogni $a, b \in C$, se $a \neq b$ allora $\{a, b\} \in E$; la dimensione di C è semplicemente il numero dei suoi elementi.

Il problema della clique è definito quindi nel modo seguente.

> Problema *Clique*
> Istanza: un grafo non orientato $G = (V, E)$ e un intero $k \leq \sharp V$.
> Domanda: esiste una clique di dimensione k in G?

Il problema può essere risolto da una MdT che genera in modo non deterministico un insieme $A \subseteq V$ e poi verifica (in maniera deterministica) se A possiede cardinalità k e forma una clique: in caso affermativo accetta, altrimenti rifiuta. Usando un'istruzione Choice$(0,1)$ di scelta (non deterministica) tra i valori 0 e 1, il funzionamento della macchina può essere descritto dalla seguente procedura:

```
begin
    A := ∅
    for v ∈ V do { i := Choice(0,1)
                 { if i = 1 then A := A ∪ {v}
    if ♯A = k ∧ A forma una clique { then accetta
                                   { else rifiuta
end
```

Nota che se $n = \sharp V$ allora vi sono proprio 2^n computazioni sull'input (G, k) e il numero di computazioni accettanti è il numero di clique di dimensione k in G. È facile verificare che il tempo richiesto dalla macchina è al più polinomiale nelle dimensioni dell'input.

In maniera analoga si può provare che anche il seguente problema appartiene a **NP**.

> Problema *Vertex cover*
> Istanza: un grafo non orientato $G = (V, E)$ e un intero $k \leq \sharp V$.
> Domanda: esiste un insieme $B \subseteq V$ di dimensione $\sharp B = k$ tale che, per ogni $\{u, v\} \in E$, B contiene almeno uno dei due nodi u e v?

La costruzione di una procedura non deterministica che risolve questo problema in tempo polinomiale è molto simile alla precedente e viene lasciata per esercizio.

Un altro problema in **NP** è definito dai circuiti hamiltoniani. Ricordiamo che in un grafo non orientato $G = (V, E)$ un circuito hamiltoniano è una permutazione $(v_1, v_2, \ldots, v_n)$ dell'insieme V tale che $\{v_i, v_i + 1\} \in E$ per ogni $i = 1, \ldots, n-1$, e inoltre $\{v_n, v_1\} \in E$. Il problema corrispondente è definito come segue.

> Problema *Circuito hamiltoniano*
> Istanza: un grafo non orientato $G = (V, E)$.
> Domanda: esiste un circuito hamiltoniano in G?

Anche qui possiamo definire una MdT che genera in modo non deterministico una sequenza S di $n = \sharp V$ nodi e poi controlla (in modo deterministico) se S forma una permutazione ed è un ciclo del grafo G.

```
begin
    siano v_1, v_2, ..., v_n i vertici di G
    S := Λ (lista vuota)
    for j = 1,2,...,n do
        begin
            k := 1
            for i = 1,2,...,n-1 do { ℓ := Choice(0,1)
                                   { k := k + ℓ
            S := INSERISCI_IN_TESTA(S, v_k)
        end
    if esistono due nodi uguali in S then rifiuta
        else if S forma un ciclo in G { then accetta
                                      { else rifiuta
end
```

Anche in questo caso è facile verificare che la macchina lavora in tempo polinomiale nelle dimensioni dell'input. Abbiamo così provato che i problemi *Clique*, *Vertex cover* e *Circuito hamiltoniano* appartengono a **NP**.

Una proprietà che caratterizza la classe **NP** è legata alla nozione di relazione polinomiale, che ricorda un'analoga caratterizzazione degli insiemi ricorsivamente numerabili (vedi il punto **b** della proposizione 1.12.2). Dati

due alfabeti Σ e Δ, una relazione $R \subseteq (\Sigma^* \times \Delta^*)$ si dice *polinomiale* se esiste una MdT deterministica che riconosce R in tempo polinomiale e inoltre, per un opportuno polinomio p, se $(x, y) \in R$ allora $|y| \leq p(|x|)$.

Proposizione 2.8.1. *Un linguaggio $L \subseteq \Sigma^*$ appartiene a* **NP** *se e solo se esiste una relazione polinomiale $R \subseteq (\Sigma^* \times \Delta^*)$ tale che*

$$L = \{x \in \Sigma^* : \exists\, y \in \Delta^* \ \ \textit{tale che} \ \ (x, y) \in R\} \tag{2.1}$$

Dimostrazione. Sia N una MdT non deterministica che riconosce L in un tempo $p(n)$, per un opportuno polinomio p. Come nella dimostrazione della Proposizione 2.7.1, denotiamo con d il grado di ambiguità N e codifichiamo ogni computazione di M su un dato input x mediante una stringa $y = c_1\$c_2\$\cdots\$c_t$, dove t è la lunghezza della computazione e ogni $c_i \in \{1, 2, \ldots, d\}$ rappresenta la mossa compiuta dalla macchina al passo i-esimo nella computazione data. Consideriamo allora la relazione R formata da tutte le coppie (x, y) dove y è la codifica di una computazione accettante di M su input x. È chiaro che R è una relazione polinomiale e che il linguaggio L soddisfa la (2.1).

Viceversa, supponiamo che L soddisfi l'uguaglianza (2.1) per una opportuna relazione polinomiale R. Allora è facile definire una MdT non deterministica M che riconosce L in tempo polinomiale. Su un input $x \in \Sigma^*$, M sceglie in modo non deterministico una stringa $y \in \Delta^*$ tale che $|y| \leq p(|x|)$; quindi M verifica se (x, y) appartiene a R simulando in modo deterministico la macchina che riconosce la relazione. In caso affermativo M accetta, altrimenti rifiuta. Poiché la lunghezza di y è al più $p(|x|)$ e R è riconoscibile in tempo polinomiale (da una MdT deterministica), anche M funziona in tempo polinomiale. □

Definiamo ora la classe **EXPTIME** come la famiglia dei linguaggi riconoscibili da una MdT deterministica in un tempo $O(2^{p(n)})$ per qualche polinomio p. Applicando allora la Proposizione 2.7.1 e usando la stessa definizione di **NP**, otteniamo il seguente risultato.

Proposizione 2.8.2. $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXPTIME}$

Esercizio

Mostrare che il problema del Commesso viaggiatore appartiene alla classe **NP**.

2.9 Il problema della soddisfacibilità

In questa sezione illustriamo il problema della soddisfacibilità per formule booleane in forma normale congiunta.

Innanzitutto ricordiamo che, per definizione, una formula booleana può essere un letterale, cioè una variabile x o una variabile negata $\overline{x}$, oppure una delle seguenti espressioni:

- $(\neg\phi)$,
- $\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_k$,
- $(\phi_1 \vee \phi_2 \vee \cdots \vee \phi_k)$,

dove $k \geq 2$, mentre $\phi, \phi_1, \phi_2, \ldots, \phi_k$ sono formule booleane. Nel seguito rappresenteremo con i simboli di somma e prodotto le tradizionali operazioni $\vee$ e $\wedge$. Ogni formula booleana nella quale compaiono k variabili distinte definisce in modo ovvio una funzione $f : \{0,1\}^k \longrightarrow \{0,1\}$.

Diciamo che una formula booleana ϕ è in forma normale congiunta (FNC per brevità) se ϕ è un prodotto di clausole di letterali, ovvero

$$\phi = E_1 \cdot E_2 \cdot \ldots \cdot E_k \tag{2.2}$$

dove $E_i = (\ell_{i1} + \ell_{i2} + \cdots + \ell_{it_i})$ per ogni $i = 1, 2, \ldots, k$, e ciascun ℓ_{ij} è un letterale.

Un esempio di formula booleana in FNC è dato dall'espressione

$$U(y_1, y_2, \ldots, y_k) = (\sum_{i=1}^{k} y_i) \cdot \prod_{i \neq j} (\overline{y_i} + \overline{y_j}).$$

È evidente che $U(y_1, y_2, \ldots, y_k)$ vale 1 se e solo se esattamente una delle sue variabili y_i assume valore 1.

Si può dimostrare che ogni funzione booleana può essere rappresentata da una formula in forma normale congiunta. Per verificare questa semplice proprietà, introduciamo la seguente definizione: per ogni variabile x e ogni $c \in \{0,1\}$, poniamo

$$\overline{x}^c = \begin{cases} \overline{x} & \text{se } c = 1 \\ x & \text{se } c = 0 \end{cases}$$

Nota che $\overline{x}^c$ assume il valore 1 se e solo se i valori di x e c sono diversi.

Lemma 2.9.1. *Sia* $f : \{0,1\}^n \longrightarrow \{0,1\}$ *una funzione booleana e sia* $S_0 = \{\underline{c} \in \{0,1\}^n : f(\underline{c}) = 0\} \neq \emptyset$. *Allora per ogni* $(x_1, x_2, \ldots x_n) \in \{0,1\}^n$, *denotando con* $\underline{c}$ *il vettore* $(c_1, c_2, \ldots c_n)$, *abbiamo*

$$f(x_1, x_2, \ldots x_n) = \prod_{\underline{c} \in S_0} (\overline{x}_1^{c_1} + \overline{x}_2^{c_2} + \cdots + \overline{x}_n^{c_n})$$

Dimostrazione. Osserviamo che, per la definizione precedente, la clausola $(\overline{x}_1^{c_1} + \overline{x}_2^{c_2} + \cdots + \overline{x}_n^{c_n})$ ha valore 1 se e solo se, per qualche i, x_i assume un valore diverso da c_i. Ne segue che la produttoria precedente è uguale a 1 se e solo se la n-pla $(x_1, x_2, \dots x_n)$ assume un valore in $\{0,1\}^n$ che non appartiene a S_0. □

Il problema della soddisfacibilità per formule booleane in FNC è definito nel modo seguente:

> Problema *SODD-FNC*
> Istanza: una formula booleana ϕ in forma normale congiunta.
> Domanda: esiste una assegnamento di valori 0 e 1 alle variabili che rende vera ϕ?

È facile provare che questo problema appartiene a **NP**. Infatti, su input ϕ, una MdT può scegliere in modo non deterministico un assegnamento A di valori alle variabili di ϕ. Quindi la macchina verifica in modo deterministico se l'assegnamento A rende vera ϕ. In caso affermativo la macchina accetta, altrimenti rifiuta. È evidente che ogni computazione della macchina termina dopo un numero polinomiale di passi.

Proposizione 2.9.2. SODD-FNC *appartiene alla classe* **NP**.

Esercizi

1. Dare la definizione di formula booleana in forma normale *disgiunta* e dimostrare che ogni funzione booleana (a un solo valore) può essere rappresentata da una formula di questo tipo.
2. Il problema della soddisfacibilità per formule booleane in forma normale *disgiunta* appartiene a **P**?

2.10 Riducibilità polinomiale

Dati due alfabeti Σ, Δ e due linguaggi $A \subseteq \Sigma^*$ e $B \subseteq \Delta^*$, diciamo che A è *polinomialmente riducibile* a B ($A \leq_p B$) se esiste una funzione $f : \Sigma^* \longrightarrow \Delta^*$, calcolabile da una MdT deterministica in tempo polinomiale, tale che per ogni $x \in \Sigma^*$, x appartiene a A se e solo se $f(x)$ appartiene a B. Diremo anche che f definisce una riduzione polinomiale da A a B o anche che A è riducibile a B mendiante la funzione f.

La definizione può essere chiaramente riformulata per i problemi di decisione. Consideriamo due problemi $\pi_1 = (I_1, q_1)$ e $\pi_2 = (I_2, q_2)$, dove I_1 e I_2 sono gli insiemi delle istanze, mentre q_1 e q_2 i rispettivi predicati. Diciamo che π_1 è polinomialmente riducibile a π_2 se esiste una funzione $f : I_1 \longrightarrow I_2$, calcolabile in tempo polinomiale da una MdT deterministica, tale che per ogni $x \in I_1$, $q_1(x) = 1$ se e solo se $q_2(f(x)) = 1$.

Esempio 2.10.1. Un semplice esempio di riducibilità polinomiale si ottiene considerando i problemi *Clique* e *Vertex cover* definiti nella sezione 2.8. Per ogni grafo non orientato $G = (V, E)$ denotiamo con $\bar{G}$ il grafo complementare $(V, \bar{E})$, dove

$$\bar{E} = \{\{u, v\} : u, v \in V, u \neq v, \{u, v\} \not\in E\}$$

Chiamiamo inoltre *copertura* di G un insieme di vertici $B \subseteq V$ tali che, per ogni $\{u, v\} \in E$, B contiene almeno uno dei due vertici u, v. Dato ora un insieme $A \subseteq V$ e un intero positivo $k \leq n$, dove $n = \sharp V$, si verifica facilmente che A è una clique di dimensione k del grafo G se e solo se $A^c = V - A$ è una copertura del grafo $\bar{G}$ formata da $n - k$ nodi. In altre parole, l'istanza (G, k) per il problema *Clique* ammette risposta positiva se e solo se l'istanza $(\bar{G}, n - k)$ per il problema *Vertex cover* ammette risposta positiva. Inoltre è chiaro che $(\bar{G}, n-k)$ può essere calcolato in tempo polinomiale da (G, k). Di conseguenza possiamo affermare che *Clique* è polinomialmente riducibile a *Vertex cover*.

Le classi **P** e **NP** definite nelle sezioni precedenti sono chiaramente chiuse rispetto alla riduzione polinomiale.

Proposizione 2.10.1. *Dati due linguaggi A e B, supponiamo che A sia polinomialmente riducibile a B. Allora se B appartiene a* **P** *anche A appartiene a* **P**. *Analogamente, se B appartiene a* **NP**, *anche A appartiene a* **NP**.

Dimostrazione. Sia f una funzione che definisce una riduzione polinomiale da A a B e sia M una MdT che calcola f in tempo $p(n)$, per un opportuno polinomio p. Se $B \in$ **P** allora esiste una MdT deterministica M' che riconosce B in tempo $q(n)$, dove q è un altro polinomio opportuno. Quindi possiamo definire una MdT deterministica M'' che, su input $x \in \Sigma^*$, calcola inizialmente la stringa $y = f(x)$ simulando la macchina M e, successivamente, simula la macchina M' su input $f(x)$ mantenendo la risposta di quest'ultima. Poiché la lunghezza di $f(x)$ è al più data da $p(|x|)$, la complessità in tempo di M'' è maggiorata da un polinomio:

$$T_{M''}(|x|) \leq p(|x|) + q(p(|x|))$$

In modo analogo si prova che, se B appartiene a **NP**, anche A appartiene a **NP**. □

Usando un ragionamento simile, è facile verificare che la riduzione polinomiale gode della proprietà transitiva. Infatti, dati tre linguaggi $A \subseteq \Sigma^*, B \subseteq \Delta^*$ e $C \subseteq \Gamma^*$, supponi che A sia riducibile a B mediante una funzione f calcolabile in un tempo polinomiale $p(n)$, e B sia riducibile a C mediante una funzione g calcolabile in un tempo polinomiale $q(n)$. Allora la funzione composta $h(x) = g(f(x))$ definisce una riduzione da A a C: per ogni $x \in \Sigma^*$, $x \in A$ se e solo se $h(x) \in C$. La funzione h è calcolabile da una MdT che, su input $x \in \Sigma^*$, simula prima la macchina che calcola f, ottenendo $f(x)$, e quindi la macchina che calcola g, ricavando $g(f(x))$. Il tempo complessivo richiesto è minore o uguale a $p(|x|) + q(p(|x|))$ che è di nuovo polinomiale poiché lo sono p e q.

2.10.1 Esempi notevoli

In questa sezione presentiamo due esempi significativi di riduzione polinomiale tra problemi di decisione.

Proposizione 2.10.2. SODD-FNC *è polinomialmente riducibile a* Clique.

Dimostrazione. Descriviamo una funzione f tra le istanze dei due problemi che definisce la riduzione. Sia ϕ una formula in FNC definita come nella sezione 2.9, ovvero $\phi = E_1 \cdot E_2 \cdot \ldots \cdot E_k$ dove $E_i = (\ell_{i1} + \ell_{i2} + \cdots + \ell_{it_i})$ per ogni $i = 1, 2, \ldots, k$, e ciascun ℓ_{ij} è un letterale. Allora $f(\phi)$ è l'istanza del problema CLIQUE data dall'intero k, pari al numero di clausole di ϕ, e dal grafo $G = (V, E)$ definito nel modo seguente:

- $V = \{[i, j] : i \in \{1, 2, \ldots, k\}, j \in \{1, 2, \ldots, t_i\}\}$;
- $E = \{\{[i, j], [u, v]\} : i \neq u, \ell_{ij} \neq \overline{\ell_{uv}}\}$.

Quindi i nodi di G sono esattamente le occorrenze di letterali in ϕ, mentre i suoi lati sono le coppie di letterali che compaiono in clausole distinte e che non sono l'uno il negato dell'altro.

È facile verificare che la funzione f è calcolabile in tempo polinomiale.

Proviamo ora che ϕ ammette un assegnamento che la rende vera se e solo se G ha una clique di dimensione k. Supponiamo che esista un tale assegnamento A. Chiaramente A assegna valore 1 ad almeno un letterale ℓ_{ij_i} per ogni clausola E_i di ϕ. Sia $\{j_1, j_2, \ldots, j_k\}$ l'insieme degli indici di questi letterali. Allora l'insieme $C = \{[i, j_i] : i = 1, 2, \ldots, k\}$ è una clique del grafo G perché, se per qualche $i, u \in \{1, 2, \ldots, k\}$, $\ell_{ij_i} = \overline{\ell_{uj_u}}$, l'assegnamento A non potrebbe rendere veri entrambi i letterali.

Viceversa, sia $C \subseteq V$ una clique di G di dimensione k. Allora, per ogni coppia $[i, j]$, $[u, v]$ in C, abbiamo $i \neq u$ e $\ell_{ij} \neq \overline{\ell_{uv}}$. Sia ora S_1 l'insieme delle variabili x tali che $x = \ell_{ij}$ per qualche $[i, j] \in C$. Analogamente, denotiamo con S_0 l'insieme delle variabili y tali che $\overline{y} = \ell_{ij}$ per qualche $[i, j] \in C$. Definiamo ora l'assegnamento A che attribuisce valore 1 alle variabili in S_1 e valore 0 alle variabili in S_0. A è ben definito perché, essendo C una clique, l'intersezione $S_1 \cap S_0$ è vuota. Inoltre A rende vera la formula ϕ, perché per ogni clausola E_i vi è un letterale ℓ_{ij_i} che assume valore 1. □

Un altro esempio di riduzione polinomiale riguarda il problema 3-*SODD-FNC*. Quest'ultimo è una variante di *SODD-FNC* che si ottiene restringendo le istanze alle formule booleane in FNC che possiedono solo clausole con 3 letterali.

> Problema 3-*SODD-FNC*
> Istanza: un formula booleana $\psi = F_1 \cdot F_2 \cdot \ldots \cdot F_k$, dove $F_i = (\ell_{i1} + \ell_{i2} + \ell_{i3})$ per ogni $i = 1, 2, \ldots, k$ e ogni ℓ_{ij} è una variabile o una variabile negata.
> Domanda: esiste una assegnamento di valori 0 e 1 alle variabili che rende vera ψ?

Proposizione 2.10.3. SODD-FNC *è polinomialmente riducibile a* 3-SODD-FNC.

Dimostrazione. Sia ϕ una formula booleana in FNC e sia $E = (\ell_1 + \ell_2 + \cdots + \ell_t)$ una clausola di ϕ dotata di $t \geq 4$ letterali. Sostituiamo E in ϕ con un prodotto di clausole $f(E)$ definito da

$$f(E) = (\ell_1 + \ell_2 + y_1) \cdot (\ell_3 + \overline{y_1} + y_2) \cdot (\ell_4 + \overline{y_2} + y_3) \cdot \ldots \cdot (\ell_{t-2} + \overline{y_{t-4}} + y_{t-3}) \cdot (\ell_{t-1} + \ell_t + \overline{y_{t-3}})$$

dove $y_1, y_2, \ldots, y_{t-3}$ sono nuove variabili.

Proviamo ora che esiste un assegnamento che rende vera la clausola E se e solo se ne esiste uno che rende vera $f(E)$. Infatti, se per qualche i $\ell_i = 1$, basta porre $y_j = 1$ per ogni $j < i - 1$ e $y_j = 0$ per ogni $j \geq i - 1$; in questo modo otteniamo un assegnamento che rende vera $f(E)$. Viceversa, se esiste un assegnamento che rende vera $f(E)$ allora deve esistere qualche letterale ℓ_i che assume valore 1. Altrimenti è facile verificare che il valore di $f(E)$ sarebbe 0. Ne segue che lo stesso assegnamento rende vera la E.

Operando questa sostituzione per tutte le clausole di ϕ dotate di più di 3 addendi, otteniamo una formula 3-FNC che soddisfa le condizioni richieste. È inoltre evidente che il tempo di calcolo necessario per realizzare la sostituzione è polinomiale. □

2.11 Problemi NP-completi

Diciamo che un problema di decisione π è *NP-completo* se π appartiene a **NP** e inoltre ogni altro problema in **NP** è polinomialmente riducibile a π.

Intuitivamente i problemi NP-completi sono i problemi più difficili nella classe **NP**. L'esistenza di un algoritmo che risolve in tempo polinomiale un problema NP-completo implicherebbe l'equivalenza delle classi **P** e **NP**. Anche se non è stato ancora provato che le due classi sono diverse, tuttavia l'ipotesi **P**=**NP** è considerata molto improbabile. Quindi dimostrare che un problema π è NP-completo significa sostanzialmente provare che il problema è intrattabile ovvero che, quasi certamente, π non ammette algoritmi di soluzione che lavorano in tempo polinomiale.

Notiamo che la definizione di problema NP-completo è simile a quella di insieme completo nella famiglia degli insiemi ricorsivamente numerabili (r.n.) presentata nella sezione 1.14. Le due problematiche e gli argomenti relativi appaiono però molto diversi. Infatti, come sappiamo, si dimostra facilmente che un insieme completo non può essere ricorsivo mentre, al contrario, non è stato ancora provato che i problemi NP-completi non appartengono alla classe **P**.

La teoria della NP-completezza si è sviluppata storicamente a partire dal seguente risultato che ha dimostrato per la prima volta l'esistenza di problemi NP-completi. In [12] si può trovare una prima raccolta di problemi di questo tipo e una trattazione delle principali tematiche associate. Questa classe di problemi è presentata e studiata nel dettaglio in gran parte dei testi di complessità computazionale e analisi di algoritmi [1, 2, 6, 14, 16, 13, 18].

Teorema 2.11.1. (Cook-Levin, 1971) *Il problema* SODD-FNC *è NP-completo.*

Dimostrazione. Nella sezione (2.9) abbiamo già dimostrato che il problema appartiene a **NP**. Dobbiamo quindi provare che ogni problema $\pi \in$ **NP** è polinomialmente riducibile a *SODD-FNC.* In altre parole, vogliamo dimostrare che per ogni MdT non deterministica M che lavora in tempo polinomiale, esiste una funzione f calcolabile in tempo polinomiale che associa a ciascuna stringa di input w di M una formula booleana ϕ, in forma normale congiunta, tale che w è accettata dalla macchina se e solo se esiste un assegnamento di valori alle variabili che rende vera ϕ.

Senza perdita di generalità, possiamo supporre che M abbia un solo nastro e che, per un opportuno polinomio p e per ogni input w di M, tutte le computazione della macchina su w abbiano la stessa lunghezza

$p(|w|)$. Infatti, se M non soddisfa quest'ultima condizione, possiamo sempre costruire una nuova macchina che, su input w, prima calcola $p(|w|)$, poi simula M su w tenendo un contatore del numero di mosse eseguite e prolungando ogni computazione fino a $p(|w|)$ passi.

Supponiamo inoltre che la macchina M sia definita da $M = (Q, q_1, \Gamma, \Sigma, \delta, q_s)$, dove $Q = \{q_1, q_2, \ldots, q_z\}$, $\Gamma = \{a_1, a_2, \ldots, a_r\}$ e $s \in \{2, 3, \ldots, z\}$, per opportuni $z, r \in \mathbb{N}$. Data ora una stringa $w \in \Sigma^*$ di lunghezza n, ogni computazione di M su w è una sequenza di $p(n)+1$ configurazioni, ciascuna delle quali è rappresentabile come sappiamo da una stringa uqv tale che $q \in Q$, $u, v \in \Gamma^*$ e $|uv| \leq p(n)+1$. La corrispondente formula $\phi = f(w)$ sarà definita su tre tipi di variabili Booleane: $S(u,t), C(i,j,t)$ e $L(i,t)$, dove gli indici u, t, i, j variano in modo opportuno. Il significato intuitivo di queste variabili è il seguente:

- la variabile $S(u,t)$ assumerà il valore 1 se al tempo t la macchina si trova nello stato q_u;

- la variabile $C(i,j,t)$ assumerà valore 1 se al tempo t nella cella i-esima si trova il simbolo a_j;

- la variabile $L(i,t)$ assumerà il valore 1 se al tempo t la testina di lettura è posizionata sulla cella i-esima.

Negli altri casi le tre variabili assumeranno valore 0. È chiaro che u dovrà rappresentare uno stato e quindi $u \in \{1, 2, \ldots, z\}$; t invece rappresenta il tempo e di conseguenza $t \in \{0, 1, \ldots, p(n)\}$, mentre i e j denotano rispettivamente una cella del nastro e un simbolo dell'alfabeto di lavoro, per cui $i \in \{1, 2, \ldots, p(n)+1\}$ e $j \in \{1, 2, \ldots, r\}$.

Un assegnamento di valori 0 e 1 a queste variabili rappresenta una computazione accettante di M su input w se le seguenti condizioni sono soddisfatte:

1. per ogni t esiste un solo u tale che $S(u,t) = 1$ ovvero, in ogni istante la macchina si può trovare in un solo stato;

2. per ogni t esiste un solo i tale che $L(i,t) = 1$ ovvero, in ogni istante la macchina legge esattamente una cella;

3. per ogni t e ogni i esiste un solo j tale che $C(i,j,t) = 1$ ovvero, in ogni istante ciascuna cella contiene esattamente un simbolo;

4. i valori delle variabili che hanno indice $t = 0$ rappresentano la configurazione iniziale su input w;
5. la variabile $S(s, p(n))$ assume valore 1, ovvero M raggiunge lo stato accettante al termine della computazione;
6. per ogni t e ogni i, se $L(i, t) = 0$, allora le variabili $C(i, j, t)$ e $C(i, j, t+1)$ assumono lo stesso valore. In altre parole il contenuto delle celle che non sono lette dalla macchina in un dato istante, resta invariato all'istante successivo;
7. se invece $L(i, t) = 1$, allora il valore delle variabili $C(i, j, t+1), S(u, t+1)$ e $L(i, t+1)$ rispettano le mosse della macchina all'istante t.

Associamo ora a ciascuna condizione una formula booleana, definita sulle variabili date, in modo tale che ogni assegnamento soddisfi la condizione se e solo se rende vera la formula associata. A tale scopo utilizziamo l'espressione $U(y_1, y_2, \dots, y_k)$ definita nella sezione 2.9 che assume valore 1 se e solo se una sola delle sue variabili $y_1, y_2, \dots, y_k$ possiede valore 1. La sua lunghezza è limitata da un polinomio nel numero delle variabili (sicuramente possiamo affermare che, per k crescente, $|U(y_1, y_2, \dots, y_k)| = O(k^3)$).

1. La prima condizione richiede che per ogni t vi sia un solo u tale che $S(u, t) = 1$. Possiamo allora scrivere la seguente formula

   $$A = \prod_{t=0}^{p(n)} U(S(1, t), S(2, t), \dots, S(z, t))$$

   la cui lunghezza è chiaramente $O(np(n))$, perché z è una costante che dipende solo da M (non dalla dimensione dell'input) e inoltre ogni t è rappresentabile da $1 + \lceil \log_2 p(n) \rceil = O(n)$ bit. È chiaro che un assegnamento rende vera la formula A se e solo se soddisfa la condizione 1.

   Le altre formule si ottengono in modo simile e hanno tutte lunghezza polinomiale in n.

2. $B = \prod_{t=0}^{p(n)} U(L(1, t), L(2, t), \dots, L(p(n)+1, t))$
3. $C = \prod_{t,i} U(C(i, 1, t), C(i, 2, t), \dots, C(i, r, t))$

4. Supponendo che $w = x_1 x_2 \cdots x_n$ e rappresentando impropriamente gli indici di $\not b$ e di ogni x_i, definiamo

$$D = S(1,0) \cdot L(1,0) \cdot \prod_{i=1}^{n} C(i, x_i, 0) \cdot \prod_{i=n+1}^{p(n)+1} C(i, \not b, 0)$$

5. $E = S(s, p(n))$

6. Per rappresentare la sesta condizione, per ogni coppia di variabili booleane x e y, denotiamo con $x \equiv y$ l'espressione $(x + \overline{y}) \cdot (\overline{x} + y)$. Tale formula vale 1 se e solo se le variabili x e y assumono lo stesso valore. Allora, per ogni i e ogni t ($t \neq p(n)$), definiamo

$$F_{it} = L(i,t) + \prod_{j=1}^{r} (C(i,j,t) \equiv C(i,j,t+1))$$

Tale formula non è in forma normale congiunta. Tuttavia, per il Lemma 2.9.1, esiste una formula $\tilde{F}_{it}$ in forma normale congiunta equivalente a F_{it}, per ogni i e ogni t. Nota anche che ogni F_{it} possiede un numero costante di letterali, un numero cioè che dipende solo dalla macchina M e non da n; la sua lunghezza quindi è dell'ordine $O(\log n)$ dovendo mantenere un numero costante di indici i, j, t. Lo stesso discorso vale per $\tilde{F}_{it}$ e di conseguenza anche la sua lunghezza è dell'ordine $O(\log n)$. Possiamo quindi definire la formula

$$F = \prod_{t=0}^{p(n)-1} \prod_{i=1}^{p(n)+1} \tilde{F}_{it}$$

che risulta in forma normale congiunta e possiede una lunghezza $O(p(n)^2 \log n)$.

7. Per ogni u, t, i, j definiamo

$$\begin{aligned} G_{utij} = {} & \overline{S(u,t)} + \overline{L(i,t)} + \overline{C(i,j,t)} + \\ & + \sum_{(q_{u'}, a_{j'}, \ell) \in \delta(q_u, a_j)} (S(u', t+1) \cdot C(i, j', t+1) \cdot L(i+\ell, t+1)) \end{aligned}$$

Osserviamo che anche questa formula non è in forma normale congiunta. Tuttavia, per quanto dimostrato nella sezione 2.9, sappiamo che esiste una formula equivalente in FNC che denoteremo con $\tilde{G}_{utij}$. Con un ragionamento analogo al precedente si può verificare

che anche la lunghezza di $\tilde{G}_{utij}$ è $O(\log n)$. Ne segue che la formula associata alla settima condizione è

$$G = \prod_{u,t,i,j} \tilde{G}_{utij}$$

e pure la sua lunghezza risulta polinomiale in n.

Poiché l'espressione U è in forma normale congiunta, tutte le formule precedenti sono in FNC. Di conseguenza, anche la formula ϕ, ottenuta dal prodotto booleano delle formule precedenti, è in FNC:

$$\phi = A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G$$

Tale formula seleziona esattamente gli assegnamenti che rappresentano computazioni accettanti di M su input w. Possiamo allora affermare che esiste un assegnamento che rende vera la formula ϕ se e solo la macchina M accetta l'input w.

È inoltre facile verificare che, per una qualsiasi fissata MdT non deterministica M, la formula ϕ può essere costruita in tempo polinomiale a partire dall'input w. □

Poiché la riducibilità polinomiale gode della proprietà transitiva, se *SODD-FNC* è polinomialmente riducibile a un problema $\pi \in$ **NP**, anche quest'ultimo risulta NP-completo. Applicando quindi i risultati presentati nella sezione precedente, possiamo enunciare la seguente proposizione.

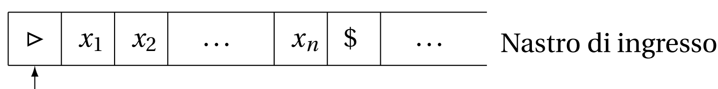
Corollario 2.11.2. *I problemi* Clique*,* Vertex cover *e 3-*SODD-FNC *sono NP-completi.*

2.12 Complessità in spazio

In questa sezione studiamo la complessità in spazio di una macchina di Turing intesa come quantità di memoria necessaria per svolgere una computazione. Innazitutto, definiamo un modello di macchina che metta in evidenza tale grandezza, distinguendola dallo spazio necessario per mantenere l'input ed eventualmente l'output della computazione. Infatti, vogliamo valutare solo lo spazio strettamente necessario per eseguire la computazione.

Una *macchina di Turing off-line* è una MdT deterministica dotata di un nastro di ingresso, un nastro di lavoro ed eventualmente un nastro di output con le seguenti proprietà:

1. il nastro di ingresso è a sola lettura e contiene l'input $x = x_1 x_2 \cdots x_n$ delimitato da due simboli particolari, $\rhd$ e \$, che non appartengono all'alfabeto d'ingresso e segnalano le celle tra le quali la testina si può muovere. Quest'ultima quindi può solo leggere il contenuto di una cella ma non lo può modificare, si può spostare di una posizione in entrambe le direzioni tranne quando scandisce le celle che delimitano l'input; su queste celle, leggendo $\rhd$ o \$, la testina si sposta sempre rispettivamente a destra e a sinistra.



2. Il nastro di lavoro è a lettura e scrittura. Inizialmente tutte le sue celle contengono il blank e la testina è posizionata sulla prima cella. Come al solito supponiamo che leggendo la prima cella la macchina muova sempre la testina verso destra (questo vincolo può essere garantito stampando sempre un simbolo speciale su tale cella).

3. Il nastro di output è usato solo nelle macchine che calcolano una funzione (nel caso di semplici accettori non è effettivamente necessario). Inizialmente contiene solo blank, la sua testina può unicamente scrivere sulla cella scandita e, dopo aver stampato un simbolo, si muove sempre di una posizione verso destra.

Come sempre inizialmente tutte le testine sono collocate sulla prima cella del nastro e la macchina si trova nello stato iniziale. Il suo funzionamento è determinato dalla funzione transizione, definita come per le MdT a più nastri con i vincoli descritti sopra. Anche il linguaggio accettato dalla macchina (o la funzione calcolata) sono definite come nel caso tradizionale.

Per ogni MdT off-line M e ogni suo input x denotiamo con $S_M(x)$ il massimo numero di celle usate sul nastro di lavoro durante la computazione di M su x. Formalmente, se $\{C_0, C_1, \ldots, C_m, \ldots\}$ è la computazione di M su x (finita o infinita). Possiamo denotare con s_i la lunghezza della porzione non-blank del nastro di lavoro in ciascuna C_i. Allora definiamo

$$S_M(x) = \max\{s_i \mid i = 0, 1, \ldots, m, \ldots\}$$

con la convenzione che $S_M(x) = +\infty$ se la macchina M non termina su input x e la successione $\{s_i\}_{i\in\mathbb{N}}$ non ammette massimo. Inoltre, per ogni $n \in \mathbb{N}$ definiamo

$$S_M(n) = \max\{S_M(x) \mid |x| = n\}$$

D'ora in poi, quando parleremo dello spazio richiesto da una MdT per riconoscere un certo linguaggio o calcolare una data funzione, faremo sempre riferimento al modello off-line.

Esempio 2.12.1. Considera il linguaggio $L = \{x \in \{a,b\}^* \mid x = x^R\}$ dell'esempio 2.5.1. Tale linguaggio può essere riconosciuto da una MdT off-line che copia l'input sul nastro di lavoro, riposiziona la testina di quest'ultimo sulla cella iniziale e quindi legge contemporaneamente il nastro di ingresso e quello di lavoro rispettivamente da destra verso sinistra e da sinistra verso destra, controllando che i simboli scanditi siano uguali; se ne trova due diversi rifiuta altrimenti accetta. È chiaro che questa MdT funziona in spazio $\Theta(n)$ e tempo $\Theta(n)$. Lo stesso linguaggio può essere riconosciuto in spazio $\Theta(\log n)$ e tempo $O(n^2 \log n)$ da una MdT off-line che prima calcola la lunghezza n dell'input in notazione binaria sul nastro di lavoro e poi, per ogni $i = 1, 2, \ldots, \lfloor n/2 \rfloor$, controlla che i simboli sul nastro di ingresso in posizione $i+1$ e $n-i+2$ siano uguali. Nota che il calcolo di questi interi e delle corrispondenti posizioni sul nastro di ingresso può essere sempre svolto in spazio $O(\log n)$.

Oltre al modello deterministico, si può definire la MdT off-line non deterministica. L'unica differenza rispetto al modello deterministico è che in questo caso la funzione transizione definisce un insieme di possibili mosse che la macchina può compiere ad ogni passo. Chiaramente tale insieme è finito e dipende solo dai simboli letti sul nastro di ingresso e su quello di lavoro e dallo stato corrente della macchina. In questo modo ci possono essere più computazioni su un dato input, una per ogni sequenza di possibili scelte compiute dalla macchina a ciascun passo. Se M è una MdT off-line non deterministica, per ogni input x il valore $S_M(x)$ sarà definito in maniera corrispondente: esso rappresenta il massimo numero di celle del nastro di lavoro utilizzate in una qualunque computazione di M su x. Analogamente $S_M(n)$ è il massimo di questi valori al variare di x tra tutti gli input di lunghezza n, per ogni $n \in \mathbb{N}$.

Data una funzione $f : \mathbb{N}_+ \to \mathbb{R}_+$ tale che $f(n) \geq 1$ per ogni n abbastanza grande, diremo che una MdT off-line M funziona (o lavora) in spazio $f(n)$ se $S_M(n) \leq \max\{1, \lceil f(n) \rceil\}$. Così le classi DSPACE($f(n)$) e NSPACE($f(n)$) sono definite come la famiglia dei linguaggi riconosciuti rispettivamente da MdT deterministiche e non deterministiche che funzionano in spazio $f(n)$.

$$\text{DSPACE}(f(n)) = \left\{L \mid \exists M \text{ det.} : L = L(M), S_M(n) \leq \max\{1, \lceil f(n) \rceil\}\right\} \quad (2.3)$$

$$\text{NSPACE}(f(n)) = \{L \mid \exists M \text{ non det. } : L = L(M), S_M(n) \leq \max\{1, \lceil f(n) \rceil\}\} \quad (2.4)$$

Per la complessità in spazio valgono le seguenti proprietà, analoghe a quelle relative al tempo di calcolo presentate nelle sezioni precedenti (anche le prove sono simili):

1. Una MdT off-line con più nastri di lavoro è definita in maniera ovvia: si tratta di una MdT off-line dotata però di $k > 1$ nastri di lavoro (invece di uno solo) che funzionano come nel caso tradizionale. Tuttavia si può dimostrare che tale modello non incrementa la famiglia di linguaggi riconosciuti in un dato spazio. Più precisamente, possiamo dire che un linguaggio L è riconosciuto da una MdT off-line con $k > 1$ nastri di lavoro in spazio $S(n)$ se ogni computazione su input di lunghezza n richiede al più $S(n)$ celle su ciascun nastro (esclusi quelli di input e output). Si dimostra che un tale linguaggio è sempre riconoscibile da una MdT off-line con un solo nastro di lavoro che funziona in spazio $S(n)$. Come nei casi precedenti l'idea della prova consiste nell'espandere opportunamento l'alfabeto di lavoro della macchina.

2. Si può dimostrare che se un linguaggio L è riconosciuto da una MdT che funziona in spazio $S(n)$ allora, per ogni $c > 0$, L è riconosciuto in spazio $cS(n)$. Possiamo quindi sempre ridurre di un fattore costante lo spazio richiesto per riconoscere un dato linguaggio. La dimostrazione di questo risultato è simile a quella della Proposizione 2.5.2 e si ottiene ancora espandendo in modo opportuno l'alfabeto di lavoro della macchina. La stessa proprietà vale anche nel caso non deterministico. Di conseguenza, $\text{DSPACE}(S(n)) = \text{DSPACE}(cS(n))$ e $\text{NSPACE}(S(n)) = \text{NSPACE}(cS(n))$ per ogni $c > 0$. Così , per ogni $f : \mathbb{N} \to \mathbb{R}_+$, le classi $\text{DSPACE}(f(n))$ e $\text{NSPACE}(f(n))$ dipendono solo dall'ordine di grandezza di $f(n)$.

3. Ogni MdT che funziona in spazio $f(n)$, per una qualunque funzione $f : \mathbb{N} \to \mathbb{R}_+$, può essere simulata da una MdT che si ferma su tutti gli input e richiede lo stesso spazio di memoria. Più precisamente, per ogni MdT M che lavora in spazio $f(n)$ esiste una MdT M' che

funziona sempre in spazio $f(n)$, si ferma su tutti gli input e riconosce esattamente il linguaggio delle stringhe di input accettate da M (ovvero M' si ferma e rifiuta anche su quelle stringhe di ingresso sulle quali M non si ferma). Nella prossima sezione vedremo come nei casi significativi il tempo di calcolo di M' sia limitato da una funzione esponenziale in f.

4. Le seguenti inclusioni si dimostrano facilmente a partire dalle definizioni precedenti:

$$\mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$$
$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$$
$$\mathrm{NTIME}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$$

Sono di particolare importanza le seguenti classi di complessità:

$$\mathbf{L} = \mathrm{DSPACE}(\log n)\ , \qquad \mathbf{NL} = \mathrm{NSPACE}(\log n)\ ,$$
$$\mathbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}(n^k)\ , \qquad \mathbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{NSPACE}(n^k)$$

Un'altra classe significativa è la famiglia NSPACE(n) dei linguaggi riconoscibili in spazio lineare da una MdT non deterministica. Infatti non è difficile dimostrare che tale classe coincide con la famiglia dei linguaggi dipendenti da contesto definita nella sezione 2.2 (vedi per esempio [14, sez. 9.3]).

2.13 Relazioni tra classi di complessità

In questa sezione confrontiamo le varie classi di complessità in tempo e spazio introdotte nelle sezioni precedenti. Un primo risultato riguarda la simulazione delle MdT non deterministiche che lavorano in un dato tempo, mediante macchine di Turing deterministiche. Sappiamo già che questa simulazione è possibile a patto di rendere esponenziale il tempo di calcolo (Proposizione 2.7.1). Valutiamo ora lo spazio di memoria richiesto da questa computazione.

Proposizione 2.13.1. *Per ogni funzione $f : \mathbb{N} \to \mathbb{R}_+$ vale l'inclusione*

$$\mathit{NTIME}(f(n)) \subseteq \mathit{DSPACE}(f(n))$$

Dimostrazione. Sia N una MdT non deterministica che funziona in tempo $f(n)$. Possiamo definire una MdT off-line deterministica M che su input x esegue la procedura definita nella dimostrazione della Proposizione 2.7.1. La macchina M esplora l'albero di computazione di N su x mantenendo e aggiornando nel nastro di lavoro la sequenza di scelte compiute da N su tale input; queste sono rappresentate da successioni finite di interi $(c_1, c_2, \ldots, c_i)$, dove $1 \leq c_j \leq d$ per ogni j, essendo $d \in \mathbb{N}$ il grado di ambiguità (o di nondeterminismo) di N, e $i \leq f(|x|)$. Durante la simulazione M mantiene sul nastro di lavoro anche la configurazione corrente di N, che ha al più lunghezza $f(|x|)$. Inoltre M può mantenere nel suo controllo finito le informazioni relative alla funzione transizione e agli stati di N. Così l'intera computazione può essere eseguita in spazio $O(f(|x|))$. $\Box$

Applicando la proposizione precedente alle funzioni polinomiali otteniamo immediatamente il seguente risultato.

Corollario 2.13.2.

$$\mathbf{NP} \subseteq \mathbf{PSPACE}$$

Valutiamo ora il tempo richiesto da una macchina deterministica per simulare una non deterministica che lavora in un dato spazio. In questo caso possiamo illustrare il funzionamento della procedura mediante il grafo delle configurazioni di una macchina di Turing. Tale grafo è definito nel modo seguente.

Considera una MdT off-line non deterministica N (a un solo nastro di lavoro) che funziona in spazio $f(n) \geq \log_2 n$ e sia x un input di N. Ogni configurazione raggiungibile da N in una qualsiasi computazione sull'input x è rappresentabile da una coppia (i, uqv) dove $i \in \mathbb{N}$ soddisfa la condizione $1 \leq i \leq |x| + 2$ e denota la posizione della testina di lettura sul nastro di input, q è lo stato corrente e uv è la porzione non blank del nastro di lavoro. Se $n = |x|$ allora l'intero i è rappresentabile da una stringa binaria di lunghezza $O(\log n)$. Quindi, poiché N funziona in spazio $f(n)$ e $f(n) \geq \log_2 n$, ogni configurazione è rappresentabile da una stringa di lunghezza $O(f(n))$. Inoltre, il numero totale di queste configurazioni è $O(nd^{f(n)})$ per qualche $d > 1$ dipendente solo da N. Definiamo allora il grafo $G(N, x)$ come un grafo orientato nel quale l'insieme dei nodi V è la famiglia delle configurazioni (i, uqv) sopra definite, mentre l'insieme dei lati E è dato da tutte le coppie $(\alpha, \beta) \in V \times V$ tali che $\alpha \vdash_{N,x} \beta$ (cioè quando N nella configurazione α, con x sul nastro di ingresso, può raggiungere in un passo la configurazione β).

Il grafo $G(N, x)$ è appunto chiamato *grafo delle configurazioni* di N su x. Nota che tra i nodi di $G(N, x)$ si trova certamente la coppia $C_0 = (1, q_0)$

che rappresenta la configurazione iniziale di N. Inoltre x è accettata da N se e solo se esiste in $G(N,x)$ un cammino da C_0 ad una configurazione accettate, ovvero un nodo $(i, uq_s v)$, dove q_s è lo stato di accettazione di N.

Proposizione 2.13.3. *Sia $f : \mathbb{N} \to \mathbb{R}_+$ una funzione tale che $f(n) \geq \log_2 n$, per ogni $n \in \mathbb{N}$. Allora, per ogni $L \in$ NSPACE$(f(n))$ esiste $c > 1$ tale che $L \in$ DTIME$(c^{f(n)})$.*

Dimostrazione. Sia L un linguaggio in NSPACE$(f(n))$ e sia N una MdT off-line non deterministica che riconosce L in spazio $f(n)$. Per un qualsiasi input x di N possiamo considerare il grafo delle configurazioni $G(N,x)$. Per la discussione precedente sappiamo che x è accettato da N se e solo se esiste in $G(N,x)$ un cammino dalla configurazione iniziale C_0 di N ad una configurazione accettate. Abbiamo quindi ricondotto il problema di riconoscere il linguaggio L a un problema di raggiungibilità su grafi che sappiamo essere risolubile da una MdT deterministica in tempo polinomiale (nel numero di nodi del grafo) [1, 6, 4]. Nel nostro caso specifico, non occorre generare l'intero grafo $G(N,x)$. È sufficiente, su input x, esplorare $G(N,x)$ in ampiezza o in profondità a partire dal nodo C_0, incorporando negli stati e nella funzione transizione della macchina deterministica le informazioni che permettono di generare, a partire da un nodo α di $G(N,x)$, tutti i nodi β tali che $\alpha \vdash_{N,x} \beta$. Volendo applicare una visita in ampiezza, una macchina deterministica M (a più nastri) può mantenere l'input x sul nastro di ingresso rendendolo a sola lettura, conservare nel secondo nastro il nodo corrente di $G(N,x)$ e mantenere nel terzo la lista dei nodi visitati in ordine di distanza dalla sorgente C_0; in questo modo gli ultimi elementi della lista, opportunamente marcati, formano la coda che guida la visita. Usando questi tre nastri la macchina deterministica può eseguire la visita in ampiezza in modo tradizionale: preleva il primo nodo della coda dal terzo nastro, lo marca come esterno alla coda senza però cancellarlo dalla lista, lo ricopia sul secondo nastro e ne genera i nodi adiacenti; se uno di questi è accettante la macchina M accetta e si ferma, altrimenti per ciascuno di questi M controlla se non sia già stato raggiunto durante la visita (scorrendo il terzo nastro) e in questo caso lo aggiunge in fondo alla coda (altrimenti lo scarta). Se la coda si svuota senza che la macchina abbia accettato, M rifiuta e si ferma. Valutiamo ora il tempo di calcolo richiesto da M. Se $n = |x|$ il numero di nodi di $G(N,x)$ è $O(nd^{f(n)})$ per qualche $d > 1$ costante e ciascuno di questi richiede al più $O((\log n + f(n))nd^{f(n)})$ passi per essere processato. Quindi, per le ipotesi su f, il tempo complessivo richiesto da M risulta $O(c^{f(n)})$ per qualche $c > 1$. □

Applicando la proposizione appena dimostrata si provano immediatamente i due seguenti risultati:

- per ogni funzione $f : \mathbb{N} \to \mathbb{R}_+$ tale che $f(n) \geq \log n$ ($\forall\ n \in \mathbb{N}$),

$$\text{NSPACE}(f(n)) \subseteq \bigcup_{k>1} \text{DTIME}(k^{f(n)})$$

- $\mathbf{NL} \subseteq \mathbf{P}$

Combinando questi risultati con le proprietà ottenute in precedenza si ricava la seguente catena di inclusioni.

Corollario 2.13.4. $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$

2.14 Complessità in spazio del problema della raggiungibilità

Dalla dimostrazione della proposizione 2.13.3 è chiaro che il problema della raggiungibilità su grafi gioca un ruolo importante nell'analisi delle classi di complessità in spazio. Abbiamo già incontrato questo problema nella sezione 2.5.1. Come sappiamo esso consiste nel verificare, dato un grafo orientato $G = (V, E)$ e due nodi $u, v \in V$, se esiste un cammino da u a v in G. La dimensione dell'istanza è qui data dal numero n di nodi di G. È facile verificare che il problema della raggiungibilità è risolubile in spazio $O(\log n)$ da una macchina non deterministica.

Proposizione 2.14.1. *Il problema della raggiungibilità appartiene alla classe* **NL**.

Dimostrazione. Possiamo considerare una MdT off-line che su input (G, u, v), con G di n nodi, genera in modo non deterministico un cammino di lunghezza al più $n-1$ partendo dal nodo u. Dato un nodo x del cammino si può scegliere quello successivo in modo non deterministico tra i vertici adiacenti a x. Se durante questo processo si incontra il nodo v la macchina accetta, in caso contrario rifiuta. Inoltre, per eseguire questo calcolo, la macchina può mantenere sul nastro di lavoro l'ultimo nodo corrente raggiunto (cancellando il precedente) e la lunghezza del cammino parziale generato, in modo da verificare che sia minore di n. Tutte le altre informazioni, necessarie per generare uno dopo l'altro tutti i nodi del cammino, sono ricavabili dalla lettura dell'input. Quindi questa computazione richiede spazio $O(\log n)$. □

Usando invece una MdT deterministica possiamo risolvere il problema della raggiungibilità in tempo polinomiale, come implicitamente dimostrato nella proposizione 2.13.3. Tuttavia la visita in ampiezza usata in quel caso (così come una eventuale visita in profondità) richiedono nel caso peggiore uno spazio $\Theta(n)$ per mantenere la coda (rispettivamente, la pila) che esegue il calcolo. È invece possibile descrivere un algoritmo che risolve il problema della raggiungibilità in spazio $O((\log n)^2)$, richiedendo però un tempo di calcolo che non è più polinomiale nelle dimensioni dell'input (osserviamo anche che non è ancora noto se la classe DSPACE($\log^2 n$) sia inclusa in **P**).

Teorema 2.14.2. *Il problema della raggiungibilità appartiene a DSPACE* ($\log^2 n$).

Dimostrazione. Consideriamo un grafo orientato $G = (V, E)$ di n nodi. Per ogni coppia di nodi $x, y \in V$ e ogni $i \in \mathbb{N}$ possiamo definire il predicato $C(x, y, i)$ che vale 1 se in G esiste un cammino da x a y di lunghezza minore o uguale a 2^i, e 0 altrimenti. È evidente che esiste un cammino da u a v in G se e solo se $C(u, v, \lceil\log_2 n\rceil) = 1$. Il problema della raggiungibilità può essere quindi risolto da un algoritmo che calcola $C(x, y, i)$ per una qualunque coppia di nodi $x, y \in V$ e un qualunque intero i tale che $0 \le i \le \lceil\log_2 n\rceil$. Un algoritmo di questo genere è descritto dalla seguente procedura ricorsiva nella quale, per semplicità, rappresentiamo ogni nodo $z \in V$ come un intero compreso tra 1 e n.

```
Procedura Raggiunge(x, y, i)
if i = 0 then if x = y ∨ (x, y) ∈ E { then return 1
                                     { else return 0
        else
             begin
               c = 0
               z = 1
               while c = 0 ∧ z ≤ n do
                    { a = Raggiunge(x, z, i − 1)
                    { b = Raggiunge(z, y, i − 1)
                    { if a = 1 ∧ b = 1 then c = 1
                    {                       else z = z + 1
               return c
             end
```

Questa procedura calcola correttamente il predicato $C(x, y, i)$, infatti per ogni $i > 0$ esiste un cammino da x a y di lunghezza al più 2^i se e solo se, per qualche $z \in V$ ne esiste uno da x a z e uno da z a y, entrambi lunghezza minore a uguale a 2^{i-1}. Il nostro algoritmo si riduce alla semplice chiamata della procedura *Raggiunge* sull'input $(u, v, \lceil \log_2 n \rceil)$. La naturale versione iterativa di questo algoritmo può essere eseguito da una MdT off-line deterministica M che mantiene sul nastro di ingresso il grafo G e l'input $(u, v, \lceil \log_2 n \rceil)$, e utilizza due nastri di lavoro: nel primo M mantiene la pila che esegue la ricorsione, formata da una sequenza di record di attivazione, uno per ogni chiamata aperta e non ancora conclusa, con il record in cima alla pila collocato più a destra nel nastro; sul secondo nastro di lavoro M esegue gli altri calcoli necessari, tra cui i confronti tra i nodi e il calcolo degli interi via via considerati. Qui, ogni record di attivazione è essenzialmente ridotto alla tripla (x, y, i) su cui è eseguita la chiamata, insieme all'indirizzo di ritorno e agli altri parametri necessari per eseguire il procedimento. Avendo (x, y, i) in cima alla pila, se $i = 0$, M calcola il valore richiesto semplicemente confrontando x e y o leggendo l'input sul nastro di ingresso. Se invece $i > 0$, M genera le triple $(x, z, i-1)$ per tutti gli $z \in V$ e per ciascuno di questi calcola il valore cercato inserendo (il record di attivazione di) $(x, z, i-1)$ in cima alla pila e riapplicando il procedimento; se il valore ottenuto è 1 la macchina cancella $(x, z, i-1)$ dalla cima della pila e inserisce la tripla $(z, y, i-1)$, ripetendo di nuovo il calcolo per quest'ultima. Se invece il valore è 0 la macchina passa a considerare un valore di z successivo, fino eventualmente a riportare il valore 0 anche alla tripla precedente (x, y, i). M può quindi gestire correttamente la ricorsione leggendo il nastro di input e confrontando ripetutamente le due triple che si trovano in cima alla pila.

È chiaro che durante la computazione la pila contiene al più $\lceil \log_2 n \rceil$ triple $(x, z, i-1)$ ciascuna delle quali richiede uno spazio $O(\log n)$ per essere conservata. Ne segue che lo spazio richiesto dal primo nastro di lavoro è $O((\log n)^2)$ mentre lo spazio utilizzato sul secondo rimane dell'ordine $O(\log n)$. □

2.14.1 Il teorema di Savitch

Presentiamo ora un risultato particolarmente significativo riguardante il confronto tra classi di complessità in spazio deterministiche e non deterministiche. Vogliamo mostrare che ogni MdT non deterministica funzionante in spazio $f(n)$ può essere simulata da una deterministica che lavora

in spazio $f(n)^2$. Per dimostrare questa proprietà occorre però assumere una condizione di regolarità della funzione f che eviti esplicitamente le funzioni più strane o troppo particolari che comunque non rappresenterebbero classi di complessità significative.

Definizione 2.14.1. *Diciamo che una funzione* $f : \mathbb{N} \to \mathbb{R}_+$ *è* spazio costruibile *se esiste una MdT off-line deterministica che su ogni input di dimensione n occupa sul nastro di lavoro esattamente spazio* $\lceil f(n) \rceil$.

Si può dimostrare che la famiglia delle funzioni spazio costruibili è molto ampia [2]. Essa include tutte le funzioni rilevanti che possono rappresentare il tempo o lo spazio richiesto da computazioni significative. Risultano spazio costruibili le funzioni che associano ad ogni $n \in \mathbb{N}$ i valori $\log_2 n$, n, n^k e $(\log_2 n)^k$ per ogni $k \in \mathbb{N}$, 2^n, $n!$. In particolare, si può provare che se due funzioni f, g sono spazio costruibili, anche $f + g$, $f \cdot g$ e 2^f sono spazio costruibili.

Teorema 2.14.3. (Savitch, 1970) *Se* $f : \mathbb{N} \to \mathbb{R}_+$ *è spazio costruibile e* $f(n) \geq \log_2 n$ *per ogni* $n \in \mathbb{N}$, *allora vale l'inclusione*

$$NSPACE(f(n)) \subseteq DSPACE(f(n)^2)$$

Dimostrazione. Sia L un linguaggio riconosciuto da una MdT non deterministica N che lavora in spazio $f(n)$. Senza perdità di generalità possiamo supporre che su ogni input x la macchina N possa raggiungere al più un'unica configurazione accettante. Infatti, sappiamo che N possiede un unico stato accettante p e possiamo supporre che, appena prima di entrare in p, N stampi un simbolo convenzionale sulle prime $f(|x|)$ celle del nastro di lavoro (lo può fare perché f è spazio costruibile) e quindi sposti entrambe le testine all'inizio del nastro. Così, per ogni input x, possiamo denotare con A_x la configurazione accettante di N. Possiamo allora definire una MdT off-line deterministica M che su ogni input x di N verifica se nel grafo delle configurazioni $G(N, x)$ la configurazione accettante A_x è raggiungibile dalla configurazione iniziale C_0. A tale scopo M può eseguire la procedura *Raggiunge* descritta nella dimostrazione del teorema 2.14.2. In questo modo la macchina risolve di fatto il problema di raggiungibilità sull'istanza formata dal grafo $G(N, x)$ e dai nodi C_0 e A_x. Poiché f è spazio costruibile M può generare i nodi richiesti dalla procedura senza bisogno di costruire effettivamente l'intero grafo $G(N, x)$. Usando gli stati e la funzione transizione di N, insieme all'input x, la macchina M può

[2] In letteratura queste funzioni sono spesso chiamate *fully space constructible* [14].

sempre verificare direttamente se $\alpha \vdash_{N,x} \beta$ per qualsiasi coppia di nodi α, β di $G(N,x)$. Poiché $f(n) \geq \log_2 n$ ogni configurazione è rappresentabile in spazio $O(f(n))$ e il numero di nodi di $G(N,x)$ è al più $O(nd^{f(n)})$ per qualche $d > 1$. Di conseguenza, lo stesso teorema 2.14.2 garantisce che lo spazio richiesto da M è $O((\log nd^{f(n)})^2) = O(f(n)^2)$. □

Una immediata conseguenza della proposizione precedente riguarda le classi di complessità deterministiche e non deterministiche in spazio polinomiale. Contrariamente a quanto si congettura sia valido per il tempo di calcolo, nel caso dello spazio le due classi coincidono.

Corollario 2.14.4.

$$\mathbf{PSPACE} = \mathbf{NPSPACE}$$

Così, oltre all'inclusione $\mathbf{NL} \subseteq \mathrm{DSPACE}((\log n)^2)$, un'altra rilevante implicazione del teorema di Savitch è data da $\mathrm{NSPACE}(n) \subseteq \mathrm{DSPACE}(n^2)$. Di conseguenza, tutti i linguaggi dipendenti da contesto sono riconoscibili in spazio $O(n^2)$ da una MdT deterministica.

2.15 Chiusura rispetto al complemento

In questa sezione consideriamo il problema di verificare se una data classe di complessità $\mathscr{C}$ è chiusa rispetto al complemento, ovvero se per ogni linguaggio $L \subseteq \Sigma^*$ contenuto in $\mathscr{C}$ anche il suo complementare $L^c = \{x \in \Sigma^* : x \notin L\}$ appartiene a $\mathscr{C}$. Sappiamo che nel caso delle classi deterministiche (definite rispetto al tempo oppure allo spazio) tale proprietà è generalmente vera. Infatti se una MdT deterministica riconosce un dato linguaggio L, terminando su tutti gli input, basta scambiare lo stato accettante con quello di rifiuto per ottenere una MdT che riconosce il linguaggio complementare L^c. In questo modo ogni configurazione accettante della macchina diventa una configurazione di arresto non accettante e viceversa. Lo stesso scambio però non è più sufficiente per le macchine non deterministiche: in questo caso, infatti, per verificare se $x \in L^c$ occorrerebbe verificare che *tutte* le computazioni della macchina su input x terminano in una configurazione di arresto non accettante.

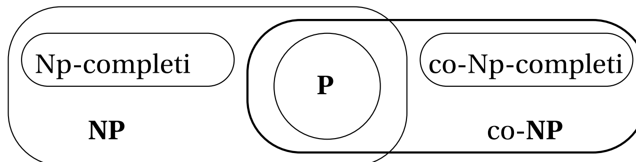
In particolare, per quanto riguarda il tempo di calcolo, si congettura che le classi NTIME$(f(n))$ in generale non siano chiuse rispetto al complemento. Così è naturale definire la classe co-**NP** dei linguaggi complementari di linguaggi in **NP**:

$$\text{co-}\mathbf{NP} = \{L \subseteq \Sigma^* : L^c \in \mathbf{NP}\}$$

La classe co-**NP** è quindi la famiglia dei linguaggi $L \subseteq \Sigma^*$ per i quali esiste una MdT non deterministica N, funzionante in tempo polinomiale, tale che ogni $x \in \Sigma^*$ appartiene a L se e solo se *tutte* le computazioni di N su x terminano in una configurazione accettante.

Per esempio, un classico problema che appartiene a co-**NP** è quello della primalità: stabilire se una stringa binaria $x \in \{0,1\}^+$ rappresenta un numero primo. Infatti si può definire una MdT non deterministica, funzionante in tempo polinomiale, che su input x genera in modo non deterministico una stringa y di lunghezza minore o uguale a $|x|$ e quindi verifica se l'intero rappresentato da x è divisibile per quello rappresentato da y. In caso affermativo rifiuta, altrimenti accetta [3].

Per le definizioni date è evidente che $\mathbf{P} \subseteq \mathbf{NP} \cap$ co-**NP**. Inoltre, si possono definire i problemi co-NP-completi come i problemi di decisione che sono complementari di problemi NP-completi. Queste due famiglie hanno proprietà simili; chiaramente l'esistenza di un algoritmo polinomiale in tempo per un problema co-NP-completo implicherebbe il collasso delle tre classi, ovvero **P** = **NP**= co-**NP**. Per questi motivi, anche se nessuno lo ha finora dimostrato, si congettura che le tre classi siano diverse e in particolare che **NP** sia diverso da co-**NP**. In queste ipotesi le proprietà di inclusione tra questi insiemi sono rappresentate dalla seguente figura.



Proprietà analoghe invece non sono valide per le classi di complessità in spazio. È stato infatti provato che, per ogni funzione f spazio costruibile, la classe NSPACE($f(n)$) è chiusa rispetto al complemento. Così per esempio le classi **NL** e NSPACE(n) risultano chiuse rispetto al complemento (e quindi il complementare di ogni linguaggio dipendente da contesto è ancora dipendente da contesto). Questo risultato di chiusura è noto come Teorema di Immerman-Szelepscényi. La sua dimostrazione è interessante perché utilizza una proprietà significativa delle macchine non deterministiche, basata sul conteggio del numero di computazioni accettanti su un

[3] Il problema della primalità non è tuttavia rappresentativo della classe co-**NP** poiché nel 2002 è stato collocato in **P** (vedi [10] per una rassegna dei risultati relativi alla complessità di questo problema).

input fissato. Intuitivamente, calcolando tale quantità, anche una macchina non determistica riesce a verificare che tutte le computazioni su un dato input siano di rifiuto, usando la stessa quantità di spazio.

2.15.1 Il teorema di Immerman-Szelepscényi

Il primo passo necessario per illustrare questo risultato consiste nella definizione delle funzioni calcolate da una MdT non deterministica. Finora, infatti, abbiamo considerato tali modelli semplicemente come riconoscitori, in grado cioè di accettare o rifiutare la parola data in input. Introduciamo ora una nuova nozione, definendo come una MdT non deterministica possa calcolare una funzione; l'idea è quella di imporre che tutte le computazioni accettanti su un dato input calcolino lo stesso valore.

Definizione 2.15.1. *Dati due alfabeti Σ e Γ, e una funzione $F : \Sigma^* \longrightarrow \Gamma^*$, diciamo che una MdT non deterministica N calcola F se N possiede un nastro di output e, per ogni $x \in \Sigma^*$, valgono le seguenti proprietà:*

1. *esiste sempre almeno una computazione accettante di N su input x;*
2. tutte *le computazioni accettanti di N su x terminano dopo aver stampato la stringa $F(x)$ sul nastro di output.*

Nota che in questo modo N accetta tutti gli input e quindi riconosce proprio il linguaggio Σ^*. Qui il risultato di una computazione non è più dato solo dallo stato raggiunto dalla macchina, ma anche dalla stringa che risulta stampata sul nastro di uscita alla fine del processo. Mentre tutte le computazioni accettanti su un dato $x \in \Sigma^*$ sono tenute a determinare lo stesso valore $F(x)$, nel caso delle computazioni non accettanti il valore stampato sul nastro di uscita è irrilevante.

Per calcolare una funzione F, una MdT non deterministica su input x può quindi indovinare in modo non deterministico il valore associato a x e poi stabilire se tale valore è giusto (cioé uguale a $F(x)$) o sbagliato (diverso da $F(x)$): nel primo caso accetta, nel secondo rifiuta.

Osserviamo anche che, secondo la presente definizione, per una MdT non deterministica riconoscere un linguaggio $L \subseteq \Sigma^*$ non significa calcolare la funzione caratteristica χ_L, data come al solito da

$$\chi_L(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{se } x \notin L \end{cases} \qquad \forall\, x \in \Sigma^*$$

In particolare rifiutare un input $x \not\in L$ non significa calcolare il valore $\chi_L(x) = 0$. Infatti, nel primo caso tutte le computazioni su input $x \not\in L$ terminano in una configurazione di rifiuto (o non terminano). Nel secondo, la macchina accetta anche tale input, tuttavia tutte le computazioni accettanti devono terminare stampando il valore 0 sul nastro di uscita. È inoltre evidente che una volta nota una MdT non deterministica N per calcolare χ_L si può immediatamente determinare una nuova MdT per calcolare χ_{L^c} quasi identica a N: basta infatti scambiare i valori 1 e 0 da stampare sul nastro di uscita alla fine di ogni computazione.

Una seconda proprietà che introduciamo per i nostri scopi riguarda la complessità del seguente problema di conteggio. Si tratta in parole povere di determinare in un grafo orientato il numero di vertici raggiungibili da un nodo dato.

> Problema *Cardinalità nodi raggiungibili*
> Istanza: un grafo orientato $G = (V, E)$ e un nodo $u \in V$.
> Soluzione: $\sharp\{v \in V :$ esiste in G un cammino da u a $v\}$.

Rappresentando gli interi in notazione binaria, il problema è di fatto ricondotto al calcolo di una funzione $F : \Sigma^* \longrightarrow \{0,1\}^*$, dove Σ è un alfabeto opportuno che codifica le istanze. Come al solito, la dimensione naturale di una istanza è data dal numero di nodi del grafo G, ovvero dal valore $n = \sharp V$. Così ogni soluzione è una stringa binaria di lunghezza al più $1 + \lfloor \log_2 n \rfloor = O(\log n)$. Vogliamo provare che questa funzione può essere calcolata da una MdT non deterministica in spazio $O(\log n)$.

A tale scopo definiamo l'insieme $S(k)$ per un qualunque $k \in \{0, 1, \ldots, n\}$:

$$S(k) = \{v \in V : \text{ esiste in } G \text{ un cammino da } u \text{ a } v \text{ di lunghezza minore o uguale a } k\}$$

Il nostro problema è quindi ricondotto al calcolo di $\sharp S(n-1)$. Per determinare tale quantità possiamo definire un algoritmo che, per ogni $k \in \{1, 2, \ldots, n-1\}$, calcoli il valore $\sharp S(k)$ usando $\sharp S(k-1)$. Chiaramente il valore iniziale è $\sharp S(0) = 1$ poiché $S(0) = \{u\}$.

```
begin
  ♯S(0) = 1
  for k = 1,2,...,n−1 do
      calcola ♯S(k) a partire da ♯S(k−1)
end
```

Si vuole determinare $\sharp S(k)$ conoscendo semplicemente il valore $\sharp S(k-1)$, senza bisogno di mantenere i risultati delle iterazioni precedenti, ovvero gli interi $\sharp S(1), \sharp S(2), \ldots, \sharp S(k-2)$. L'algoritmo complessivo è definito da un programma principale, che chiamiamo *Main*, e dalle procedure *Calcola*, *Appartiene* e *Guess*, che si richiamano a vicenda, l'ultima delle quali contiene esplicitamente i passi non deterministici del procedimento. L'intero algoritmo riceve in ingresso l'insieme dei nodi V, l'insieme dei lati E e il nodo di partenza $u \in V$. I termini V, E e u sono qui considerati come variabili globali e non vengono modificati durante la computazione.

Main
```
begin
  x = 1
  for k = 1,2,...,n-1 do
      { y := Calcola(x,k)
      { x := y
end
```

Quindi la chiamata *Calcola*(x,k) restituisce il valore $\sharp S(k)$ ricevendo in input, oltre allo stesso k, anche l'intero $x = \sharp S(k-1)$.

Procedura *Calcola*(x,k)
```
begin
  ℓ = 0
  for v ∈ V do
      { c := Appartiene(v,x,k)
      { ℓ := ℓ + c
  return ℓ
end
```

Invece, la procedura non deterministica *Appartiene* calcola la funzione $\chi_{S(k)}$ nel senso della definizione 2.15.1. Più precisamente, la chiamata *Appartiene*(v,x,k) restituisce il valore $\chi_{S(k)}(v)$ usando per il calcolo l'intero $x = \sharp S(k-1)$ ricevuto in ingresso. Essa si basa sulla procedura non deterministica *Guess* che restituisce sempre un valore booleano ed è descritta dal seguente schema:

Procedura *Guess*$(u,w,k-1)$
```
begin
  z := u
  i := 1
```

```
  while i ≤ k−1 ∧ z ≠ w do
          { scegli y ∈ V tale che (z, y) ∈ E in modo non deterministico
          { z := y
          { i := i + 1
  if  z = w then return  vero
             else return  falso
end
```

Su input $(u, w, k-1)$, la procedura *Guess* genera in modo non deterministico un cammino di lunghezza al più $k-1$ partendo da u, verificando ad ogni passo se il nodo corrente incontrato (denotato da z nello schema) coincide con w o meno; in caso affermativo *Guess* restituisce *vero*, altrimenti, se w non viene mai incontrato, restituisce *falso*.

Nota che la procedura *Guess* non rifiuta mai e si ferma sempre; essa non “riconosce un linguaggio” in senso tradizionale e neppure “calcola una funzione” nel senso della definizione 2.15.1. Invece, essa svolge semplicemente una ricerca non deterministica nel grafo, seguendo un cammino di lunghezza al più $k-1$ che parte da u, verificando se il nodo raggiunto ad ogni passo coincide con w.

Così la procedura *Appartiene* può essere descritta dal seguente schema, dal quale è evidente come i suoi passi non deterministici siano di fatto eseguiti dalle chiamate alla procedura *Guess*.

```
Procedura Appartiene(v, x, k)
begin
  m := 0
  out := 0
  for w ∈ V do
      if  Guess(u, w, k−1) then { m := m + 1
                                { if  w = v ∨ (w, v) ∈ E then  out := 1
  if m < x then RIFIUTA e ARRESTA
           else return out
end
```

Nota che ogni computazione di $Appartiene(v, x, k)$ calcola nella variabile m il numero di nodi individuati dalle n chiamate di $Guess(u, w, k-1)$. Se tale numero è diverso da $x = \sharp S(k-1)$ vuol dire che la computazione non ha effettivamente determinato tutti i possibili nodi raggiungibili da u in $k-1$ passi, e quindi l’algoritmo si arresta senza accettare. Viceversa, se

$m = x$ allora tutti quei nodi sono stati scoperti dalla computazione e l'algoritmo determina effettivamente se v è raggiungibile in k passi da u, mantenendo il risultato nella variabile di ritorno out; in questo caso la computazione non si ferma e continua con le iterazioni successive. In conclusione, le uniche computazioni di *Appartiene*(v,x,k) che non rifiutano fermandosi sono quelle che restituiscono effettivamente il valore $out = \chi_{S(k)}(v)$.

Proposizione 2.15.1. *Il problema* Cardinalità nodi raggiungibili *è risolubile da una MdT non deterministica in spazio* $O(\log n)$.

Dimostrazione. L'algoritmo descritto sopra può essere chiaramente eseguito da una MdT off-line non deterministica che mantiene sul nastro di input le quantità ricevute in ingresso, ovvero V, E, u, calcolando invece nel nastro di lavoro, aggiornandoli di volta in volta, i valori delle variabili necessarie alla computazione. Tali variabili sono x, y, k, ℓ, m, e i, che assumono valori in $\mathbb{N}$, e v, w, z, che variano in V. Queste quantità possono essere rappresentate da interi non negativi minori o uguali a n e quindi ogni computazione richiede spazio $O(\log n)$.

Dimostriamo ora che l'algoritmo è corretto, ovvero che per ogni $k \in \{1,2,\ldots,n-1\}$ la chiamata *Calcola*(x,k) restituisce proprio $\sharp S(k)$. Ragionando per induzione si verifica direttamente che la proprietà è vera per $k=1$. Assumiamo $k>1$ e supponiamo la proprietà vera per $k-1$. Questo significa che durante l'esecuzione di *Calcola*(x,k), e in tutte le chiamate *Appartiene*(v,x,k) (per ogni $v \in V$), il valore di x è proprio $\sharp S(k-1)$. Quindi, per il ragionamento svolto appena sopra, tutte le computazioni valide di ciascuna chiamata *Appartiene*(v,x,k) restituiscono il valore $out = \chi_{S(k)}(v)$, e di conseguenza *Calcola*(x,k) restituisce il valore corretto $\ell = \sharp S(k)$. □

Teorema 2.15.2. *Sia* $f : \mathbb{N} \longrightarrow \mathbb{N}$ *una funzione spazio costruibile tale che* $f(n) \geq \log_2 n$ *per ogni* $n \in \mathbb{N}$. *Allora NSPACE*$(f(n))$ *è chiusa rispetto al complemento.*

Dimostrazione. Consideriamo un linguaggio $L \subseteq \Sigma^*$ incluso in NSPACE$(f(n))$ e sia M una MdT off-line non deterministica che riconosce L in spazio $O(f(n))$. Per un qualsiasi $x \in \Sigma^*$ di lunghezza n, sia $G_{M,x} = (V,E)$ il grafo delle configurazioni di M su input x. Nota che $\sharp V = O(c^{f(n)})$ per qualche $c > 1$ e, per ogni $v \in V$, abbiamo $|v| = O(f(n))$. Inoltre possiamo denotare con $C_0 \in V$ la configurazione iniziale di M su x. Applicando la proposizione 2.15.1 sappiamo che il numero di vertici $v \in V$ raggiungibili da C_0 in $G_{M,x}$, ovvero $\sharp\{v \in V : C_0 \vdash^*_{M,x} v\}$, è calcolabile in spazio $O(\log(\sharp V)) = O(f(n))$ da

una MdT non deterministica. Possiamo allora definire una nuova MdT non deterministica N che riconosce L^c. Tale macchina, su input $x \in \Sigma^*$, dapprima calcola $t = \sharp\{v \in V : C_0 \vdash^*_{M,x} v\}$, poi genera in modo non deterministico i nodi in V raggiungibili da C_0, controllando nello stesso tempo se tali nodi sono configurazioni accettanti di M e determinando il loro numero. Quindi, se tale numero è diverso da t rifiuta e si ferma, altrimenti, se ne ha trovato uno accettante per M rifiuta di nuovo, in caso contrario (quando i nodi raggiunti sono tutti configurazioni di rifiuto per M) accetta. Il procedimento non deterministico è descritto formalmente dalla seguente procedura che utilizza la funzione *Guess* definita sopra e applica di fatto la stessa strategia utilizzata nella proposizione 2.15.1 per calcolare il numero di nodi raggiungibili da un vertice dato.

```
begin
  t := ♯{v ∈ V : C_0 ⊢*_{M,x} v}
  out := 0
  m := 0
  for v ∈ V do
      if  Guess(C_0, v, ♯V − 1) then { m := m + 1
                                      { if  v accettante per M then  out := 1
  if  m < t then RIFIUTA
          else if  out = 0 then ACCETTA
                            else  RIFIUTA
end
```

Valutiamo ora lo spazio richiesto dall'esecuzione della procedura appena descritta. Nota che anche in questo caso la macchina N non ha bisogno di generare effettivamente tutto il grafo $G_{M,x} = (V, E)$. Essa può incorporare la funzione transizione di M e verificare direttamente, per ogni $v, w \in V$, se $(v, w) \in E$. Inoltre, conoscendo la macchina M, la stessa N può generare tutti i nodi di V, uno alla volta, in un ordine qualsiasi, riutilizzando lo stesso spazio di memoria e mantendo solo quelli necessari per proseguire il calcolo. Sappiamo già dalla proposizione precedente che il calcolo di $\sharp\{v \in V : C_0 \vdash^*_{M,x} v\}$ e la chiamata *Guess*$(C_0, v, \sharp V - 1)$ richiedono al più spazio $O(f(n))$. Le altre quantità usate nella procedura (m, v e out) possiedono anche loro la stessa dimensione e quindi l'intero calcolo può essere eseguito in spazio $O(f(n))$. □

Bibliografia

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[2] S. Arora, B. Barak, *Computational complexity: a modern approach*, Cambridge University Press, 2009.

[3] A. Bertoni, *Introduzione alla calcolabilità*, Dispense del corso di Informatica Teorica, Università degli Studi di Milano, 1990, reperibile al sito https://bertoni.di.unimi.it/Calcolabilita.pdf .

[4] A. Bertoni, M. Goldwurm, *Progetto e analisi di algoritmi*, Dispense del corso di Algoritmi e Strutture Dati, Rapporto interno n.230-98, Dip. Scienze dell'Informazione - Università degli Studi di Milano, Settembre 2004, reperibile al sito https://bertoni.di.unimi.it/Algoritmi_e_Strutture_Dati.pdf.

[5] A. Bertoni, C. Mereghetti, *Dispense del corso di Informatica Teorica (A.A. 2000/2001)*, Dip. Scienze dell'Informazione, Università degli Studi di Milano, reperibile al sito https://bertoni.di.unimi.it/Circuiti_e_complessita.pdf .

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati* (quarta edizione), McGraw-Hill, 2023.

[7] S. Crespi Reghizzi, *Linguaggi formali e compilazione*, Pitagora Editrice Bologna, 2006.

[8] M.D. Davis, R. Sigal, E.J. Weyuker, *Computability, complexity and languages*, Academic Press, 1994.

[9] A. de Luca, F. D'Alessandro, *Teoria degli automi finiti*, Springer-Verlag, 2013.

[10] M. Dietzfelbinger, *Primality testing in polynomial time, from randomized algorithms to "PRIMES is in P"*, Lecture Notes in Computer Science, vol. 3000, Springer–Verlag, 2004.

[11] C. Froidevaux, M.C. Gaudel, M. Soria, *Types de données et algorithmes*, Ediscience International, 1993.

[12] M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W.H. Freeman and Company, 1979.

[13] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Automi, linguaggi e calcolabilità,* Pearson, Addison-Wesley, 2009.

[14] J.E. Hopcroft, J.D. Ullman, *Introduction to automata theory, languages and computation,* Addison-Wesley, 1979.

[15] A.J. Kfoury, R.N. Moll, M.A. Arbib, *A programming approach to computability*, Springer, 1982 (versione italiana: *Programmazione e computabilità,* ETAS Libri, 1986).

[16] C. Papadimitriou, *Computational complexity,* Addison-Wesley, 1994.

[17] H. Rogers, *Theory of recursive functions and effective computability*, MIT Press, 1967.

[18] S. Skiena, *The algorithm desing manual*, Springer, 2020.

[19] I. Wegener, *The complexity of Boolean functions*, John Wiley & Sons Inc, 1987.

Introduzione alla calcolabilità e alla complessità computazionale

Alberto Bertoni, Massimiliano Goldwurm

La calcolabilità delle funzioni e la complessità computazionale dei problemi sono due argomenti classici di Informatica Teorica. In questo testo si presentano gli aspetti principali di queste tematiche con uno scopo didattico e divulgativo. La nozione di calcolabilità è legata all'esistenza di algoritmi in grado di calcolare una funzione o di risolvere un problema. In questo contesto sono di interesse anche le funzioni e i problemi che non ammettono algoritmi. La complessità computazionale invece riguarda l'analisi delle risorse (tipicamente tempo e spazio) richieste da un algoritmo per risolvere un dato problema o calcolare una certa funzione. Entrambe queste tematiche sono considerate di base per una laurea in Informatica o in Matematica e il presente testo si rivolge in particolare agli studenti dei corsi di laurea a carattere scientifico che possono ritrovare questi argomenti in diversi insegnamenti nel loro percorso di studi.

ISBN 979-12-5510-440-7 (print)
ISBN 979-12-5510-436-0 (PDF)
ISBN 979-12-5510-438-4 (EPUB)