Elementi di Calcolabilità e Complessità

Alessio Delgadillo

A.A. 2021-2022

Indice

1 Calcolabilità **2**

1.1 Idea intuitiva di Algoritmo 3
1.2 Macchine di Turing 4
1.3 Linguaggi *FOR* e *WHILE* 9
1.4 Problemi e Funzioni 13
1.5 Due approcci alla calcolabilità 17
1.6 Funzioni ricorsive primitive 19
1.7 Diagonalizzazione 29
1.8 Funzioni ricorsive generali 31
1.9 Alcuni risultati classici 34
1.10 Problemi Insolubili e Riducibilità 42

2 Complessità **56**

2.1 Una teoria quantitativa degli algoritmi 57
2.2 Misure di complessità deterministiche 62
2.2.1 Macchine di Turing a k-nastri 62
2.2.2 Complessità in tempo deterministico 64
2.2.3 Macchine di Turing I/O 72
2.2.4 Complessità in spazio deterministico 73
2.3 Misure di complessità non deterministiche 77
2.3.1 Macchine di Turing non deterministiche 79
2.3.2 Complessità in tempo e in spazio non deterministici . . 81
2.4 Funzioni di misura, un po' di gerarchia e due assiomi 87
2.5 P e NP 93
2.5.1 Problemi trattabili e intrattabili 94
2.5.2 Alcuni problemi interessanti e riduzioni efficienti tra essi 98
2.5.3 Problemi completi per P e NP 105

Capitolo 1

Calcolabilità

1.1 Idea intuitiva di Algoritmo

Ci sono moltissimi formalismi che sono stati proposti per esprimere algoritmi, tra cui Macchine di Turing, funzioni ricorsive, λ-calcolo, Random Access Machine, algoritmi di Markov, grammatiche generali, sistemi di Post e linguaggi di programmazione. In ciascuno di questi gli algoritmi devono soddisfare i seguenti requisiti:

i) un algoritmo è costituito da un insieme *finito* di istruzioni;

ii) le istruzioni possibili sono in numero *finito* e hanno un effetto *limitato* su dati *discreti*, esprimibili in maniera *finita*;

iii) una computazione è eseguita per *passi discreti* (singoli), senza ricorrere a sistemi analogici o metodi continui, ciascuno dei quali impiega un tempo *finito*;

iv) ogni passo dipende solo dai precedenti e da una porzione finita dei dati, in modo deterministico (cioè senza essere soggetti ad alcuna distribuzione probabilistica non banale);

v) *non c'è limite* al *numero di passi* necessari all'esecuzione di un algoritmo, né alla *memoria* richiesta per contenere i dati (finiti) iniziali, intermedi ed eventualmente finali (si noti che il numero dei passi di calcolo è finito solo quando non vi sia alcuna istruzione dell'algoritmo che si possa eseguire, sia perché abbiamo trovato il risultato e raggiunto uno stato finale, sia perché ci troviamo in uno stato di stallo).

Sotto queste ipotesi, tutte le formulazioni fin ad ora sviluppate sono equivalenti e si *postula* che lo saranno anche tutte quelle future[1].

[1]Un'eccezione è costituita dalle macchine concorrenti/interattive in cui i dati di ingresso variano nel tempo; inoltre vi sono formalismi che tengono conto di quantità continue quali gli algoritmi probabilistici o stocastici, usati per esempio nella descrizione di sistemi biologici o di sistemi ibridi, anche se tali quantità sono poi approssimate a valori discreti, ricadendo così nel caso che consideriamo qui; altre eccezioni sono gli algoritmi nondeterministici, ma per ogni algoritmo di quest'ultimo tipo vi è un algoritmo deterministico del tutto equivalente (vedi teorema 2.3.1).

1.2 Macchine di Turing

Introdurremo di seguito uno dei formalismi più importanti e più diffusi per esprimere algoritmi: le Macchine di Turing, che ricordano con straordinaria verosomiglianza i comuni elaboratori di Von Neumann (o a programma memorizzabile) cui siamo abituati. Ve ne sono moltissime definizioni, che differiscono per varianti spesso irrilevanti; la definizione originale prevede un esecutore umano e fu introdotta da Alan Turing nel 1936, ben prima che ci fossero dei computer funzionanti.

Definizione 1.2.1 (Macchina di Turing)**.** Si dice Macchina di Turing (in breve MdT) una quadrupla

$$M = (Q, \Sigma, \delta, q_0)$$

dove:

- $Q(\ni q_i)$ è l'insieme *finito* degli *stati*, con $h \notin Q$ (con lo stato speciale h indicheremo il caso di arresto "corretto" di un calcolo di M). Attenzione che *computazione terminata* $\neq$ *la macchina si arresta* poiché la macchina si può arrestare senza aver raggiunto la configurazione finale (ad esempio se la macchina entra in uno stato q_k e la funzione δ non è definita per quello stato).
- $\Sigma(\ni \sigma, \sigma', \dots)$ è l'insieme *finito* dei simboli (*alfabeto*) con
  - $\# \in \Sigma$ rappresenta il *carattere bianco*;
  - $\triangleright \in \Sigma$ (*respingente*) rappresenta l'inizio della stringa;
  - $\{L, R, -\} \notin \Sigma$ indicano rispettivamente "scorri a sinistra", "scorri a destra" e "resta fermo" sul nastro.
- $q_0 \in Q$ è lo *stato iniziale*.
- $\delta \subseteq (Q \times \Sigma) \times (Q \cup \{h\}) \times \Sigma \times \{L, R, -\}$ è la *relazione di transizione*, tale che
  $$\text{se } (q, \triangleright, q', \sigma, D) \in \delta \text{ allora } \sigma = \triangleright, D = R$$
  Questa condizione dice che il carattere corrente, ovvero il cursore, non può mai trovarsi a sinistra della *marca di inizio stringa*, $\triangleright$. Più intuitivamente se $\delta(q, \triangleright) = (q', \sigma, D)$ allora $\sigma = \triangleright, D = R$, secondo quanto stipulato sotto.

In questa parte del corso restringeremo la relazione δ in modo che sia una funzione rispetto ai suoi primi due argomenti, imponendo cioè che, per ogni coppia di quintuple:

$$(q, \sigma, q', \sigma', D'), (q, \sigma, q'', \sigma'', D'') \in \delta \Rightarrow q' = q'', \sigma' = \sigma'', D' = D''$$

Grazie a questa condizione possiamo scrivere $\delta(q, \sigma) = (q', \sigma', D')$ al posto della quintupla $(q, \sigma, q', \sigma', D')$. Se volessimo essere più precisi, una relazione è una corrispondenza tra gli elementi di due insiemi; una funzione invece è una particolare relazione che associa ad ogni elemento del dominio uno e un solo elemento del codominio.

È facile verificare che la condizione (i) dell'idea intuitiva di algoritmo elencata dianzi è soddisfatta, così come lo è anche la prima parte della condizione (ii). Esse richiedono che i programmi siano finiti e che le possibili istruzioni, operanti su dati discreti, siano in numero finito. Infatti, ogni macchina ha un numero finito di possibili istruzioni, poiché gli insiemi Q e Σ sono finiti; di conseguenza la sua funzione di transizione δ contiene un numero finito di elementi. Inoltre, i dati su cui opera una MdT sono stringhe w di caratteri appartenenti a Σ, in simboli $w \in \Sigma^*$. Più precisamente Σ^* contiene la stringa vuota ϵ e per tutte le stringhe $w \in \Sigma^*$ e tutti i caratteri $a \in \Sigma^*$ contiene la stringa $a \cdot w$, dove "$\cdot$" è l'operazione associativa (quasi sempre omessa) di giustapposizione tra caratteri esteso alle stringhe; brevemente, Σ^* è il monoide libero generato da Σ avente come unica operazione interna associativa "$\cdot$" ($\forall w, w', w'' \in \Sigma^*$ vale $w \cdot (w' \cdot w'') = (w \cdot w') \cdot w''$) e con identità destra e sinistra ϵ ($\forall w \in \Sigma^*$ vale $\epsilon \cdot w = w = w \cdot \epsilon$).

Esiste una variante delle MdT detta *non-deterministica* in cui non si richiede che δ sia una funzione. Nella seconda parte del corso ne daremo una definizione formale, e ricorderemo che ha la "stessa potenza espressiva" della MdT definita sopra. Più avanti useremo, introducendola in modo intuitivo, anche la variante in cui si hanno più nastri, la cui definizione si trova nella definizione 2.2.1. Il suo potere espressivo non cambia, così come non cambia quello di tutte le altre varianti, per esempio quelle macchine con un nastro infinito anziché semi-infinito, oppure con più di un nastro, oppure ancora quelle che possono o scrivere o spostarsi, o hanno altre diavolerie "ragionevoli". Questa robustezza rafforza la sensazione che il modello sia azzeccato.

Esempio 1.2.1 (Una macchina che qualcosa fa)**.** La tabella riportata sotto rappresenta la funzione di transizione della MdT $M = (Q, \Sigma, \delta, q_0)$, con $Q = \{q_0, q_1\}$ e $\Sigma = \{\#, \triangleright, a\}$.

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

In seguito, nel definire una MdT, ometteremo per brevità la definizione degli insiemi Q e Σ degli stati e dei simboli, i cui elementi possono essere dedotti guardando la tabella che definisce la funzione di transizione δ.

Passiamo ora a descrivere la dinamica delle macchine di Turing, cioè il loro comportamento quando operano su una stringa di caratteri memorizzati sul nastro. In altre parole, vogliamo definire cos'è una computazione di una macchina. Intuitivamente, una computazione di una MdT è una successione di configurazioni, ognuna ottenuta dalla precedente in accordo con la definizione della funzione di transizione δ. Ci manca la definizione di configurazione, che verrà data tra poche righe. Per il momento, e in modo del tutto informale, indichiamo la coppia *Stato/Nastro* una *configurazione istantanea* della macchina; dove *Stato* è lo stato in cui si trova la macchina e *Nastro* è una porzione "abbastanza lunga" e *finita* del nastro che contiene almeno la sua parte non bianca.

La seguente tabella rappresenta informalmente una computazione della MdT introdotta nell'esempio 1.2.1, che si arresta con successo in sette passi. Nelle configurazioni, il carattere sottolineato nella parte *Nastro* sta a indicare la posizione corrente del cursore, o *testina*.

Configurazione	Azione effettuata
$q_0/\triangleright\#\#a\#a\underline{a}\# \rightarrow$	"cancella" a cambia stato e vai a sinistra
$q_1/\triangleright\#\#a\#\underline{a}\#\# \rightarrow$	non scrive, non si sposta, cambia stato
$q_0/\triangleright\#\#a\#\underline{a}\#\# \rightarrow$	"cancella" a cambia stato e vai a sinistra
$q_1/\triangleright\#\#a\underline{\#}\#\#\# \rightarrow$	lascia #, cambia stato, va a sinistra
$q_0/\triangleright\#\#\underline{a}\#\#\#\# \rightarrow$	"cancella" a cambia stato e vai a sinistra
$q_1/\triangleright\#\underline{\#}\#\#\#\#\# \rightarrow$	lascia #, cambia stato, va a sinistra
$q_0/\triangleright\underline{\#}\#\#\#\#\#\# \rightarrow$	si arresta
$h/\triangleright\underline{\#}\#\#\#\#\#\# \rightarrow$	

Più precisamente, una *configurazione* C di una MdT è una quadrupla

$$(q, u, \sigma, v) \in (Q \cup \{h\}) \times \Sigma^* \times \Sigma \times \Sigma^F$$

dove q è lo stato corrente, u è la stringa di caratteri a sinistra del simbolo corrente σ e v è quella dei caratteri alla sua destra. Per semplicità scriveremo $(q, u\underline{\sigma}v)$ al posto di (q, u, σ, v), così come abbiamo già fatto sopra. Poiché la stringa $w\sigma v$ deve essere finita (e poiché ora)[2] non ci interessa considerare i simboli $\#$ a destra del simbolo $\sigma_n \neq \#$ più a destra nel nastro (ovvero della porzione v), indichiamo con ϵ la stringa vuota e conveniamo che $\# \cdot \epsilon = \epsilon$ e $\sigma \cdot \epsilon = \sigma$, se $\sigma \neq \#$. Definiamo allora $\Sigma^F = \Sigma^* \cdot (\Sigma \backslash \{\#\}) \cup \{\epsilon\}$. Quindi scriviamo $v = \sigma_0\sigma_1 \ldots \sigma_n$, con $\sigma_n \neq \#$, al posto della stringa infinita $\sigma_0\sigma_1 \ldots \sigma_n \# \# \ldots \# \ldots$. Si noti tuttavia che può benissimo darsi che $\sigma_i = \#$ per qualche $i < n$ e anche che il carattere corrente σ può essere $\#$; inoltre la stringa u può essere vuota solo quando il carattere corrente σ è la marca di inizio stringa $\triangleright$ per la seconda condizione sulla funzione di transizione δ. Con questa convenzione ogni stringa è finita.

Il frammento iniziale della computazione di prima viene rappresentato come:

$(q_0, \triangleright \# \# a \# a, a, \epsilon) \to (q_1, \triangleright \# \# a \#, a, \epsilon) \to$
$(q_0, \triangleright, \# \# a \#, a, \epsilon) \to (q_1, \triangleright \# \# a, \#, \epsilon) \to \ldots$

Un altro esempio di computazione della macchina definita nell'es. 1.2.1 è:

$(q_0, \triangleright \#, a, a) \to (q_1, \triangleright, \#, \#a) \to (q_0, \epsilon, \triangleright, \# \# a) \to$
$(q_0, \triangleright, \#, \#a) \to (h, \triangleright, \#, \#a)$

Per non eccedere nella pignoleria, non sempre utilizzeremo la versione di configurazione nella forma definita sopra, ma ci riterremo liberi di scrivere (q, w) quando non interessi sapere dove si trova il cursore (v. la definizione di computazione più sotto), o di usare altre convenzioni quando il loro significato sia chiaro dal contesto.

Formalmente un *passo di computazione* è definito per casi nel modo seguente, intendendo che le quadruple che vi appaiono siano configurazioni (ricordandosi che $q \neq h$ e usando q' per indicare, oltre a uno stato in Q, anche h; indicando con a, b, c elementi generici di Σ; e ricordando inoltre, soprattutto nel caso (ii), che se $b = \#$ e $v = \epsilon$ allora $bv = \epsilon$ e infine che nel caso (ii) se $a = \triangleright$ allora anche $b = \triangleright$).

[2]Nella seconda parte del corso terremo traccia di tutte le caselle visitate dalla MdT, ottenendo così una stima dello spazio necessario per quella computazione.

i)		$(q, uav) \to (q', ubv)$	se $\delta(q, a) = (q', b, -)$
ii)		$(q, ucav) \to (q', ucbv)$	se $\delta(q, a) = (q', b, L)$
iii)	(a)	$(q, uacv) \to (q', ubcv)$	se $\delta(q, a) = (q', b, R)$
	(b)	$(q, ua) \to (q', ub\#)$	se $\delta(q, a) = (q', b, R)$

Ciascun passo ha un effetto limitato sulle configurazioni, come richiesto dalla seconda parte del punto (ii) nella idea intuitiva di algoritmo. Inoltre, un singolo passo dipende *unicamente da un solo simbolo*, quello corrente,[3] e da *un solo stato*, quello corrente (cf. i punti (iii) e (iv) richiesti dall'intuizione). Si noti che sia per determinare la regola da applicare che per applicarla diamo per primitiva la capacità dell'esecutore di ricorrere al "pattern matching".

Una *computazione* è una successione finita di passi

$$(q_0, \triangleright w) \to^* (q', w')$$

dove $\to^*$ è la chiusura riflessiva e transitiva di $\to$. Come d'abitudine, se vi sono n passi, la computazione è lunga n e la scriviamo così

$$(q_0, \triangleright w) \to^n (q', w')$$

Diremo invece che la computazione $(q_0, w) \to^* (q', w')$ *termina* (converge, $\downarrow$) su w sse $q' = h$, e che *non termina* (diverge, $\uparrow$) sse per ogni q', w' tali che $(q_0, w) \to^* (q', w')$ esistono q'', w'' tali che $(q', w') \to (q'', w'')$. Se fosse necessario precisare che la computazione in esame è una di quelle di una particolare macchina M, scriveremo $\to^*_M$; con $M(w)$ esprimeremo che la macchina M inizia la sua computazione dalla configurazione $(q_0, \trianglerighteq w)$, cioè avendo come dato iniziale la stringa w, ovvero che *applichiamo* M a w.

C'è un limite ai passi di computazione o allo spazio necessario a contenerne i dati (punto (v) della definizione intuitiva di algoritmo)? **NO**, come si vede dal seguente esempio.

q	σ	δ
q_0	$\triangleright$	$q_0, \triangleright, R$
q_0	a	q_0, a, R
q_0	$\#$	$q_0, \#, R$

Esempio 1.2.2 (Una macchina che non converge per nessun ingresso)**.** Un esempio di computazione non terminante della macchina di sotto è

$$(q_0, \trianglerighteq a\#a\#) \to (q_0, \triangleright a\#\underline{a}\#) \to^* (q_0, \triangleright a\#a\# \ldots \underline{\#}\#) \to \ldots$$

[3] Si noti che l'unica maniera per ispezionare una parte non finita del nastro sarebbe quella di accedere a tutto il nastro a destra del cursore.

1.3 Linguaggi *FOR* e *WHILE*

Introduciamo adesso un altro formalismo per esprimere funzioni, certamente più familiare: un linguaggio di programmazione, o meglio il suo nucleo, in forma essenziale. Si tratta di un linguaggio imperativo semplicissimo, che opera solo sui numeri naturali e sui valori di verità, con strutture di controllo che sono presenti in ogni linguaggio di programmazione.

Sintassi (astratta)

$$
\begin{array}{lll}
E \to & n \mid x \mid E_1 + E_2 \mid E_1 \times E_2 \mid E_1 - E_2 & \text{Espr. Aritmetiche} \\
B \to & b \mid E_1 < E_2 \mid \neg B \mid B_1 \vee B_2 & \text{Espr. Booleane} \\
C \to & \texttt{skip} \mid x := E \mid C_1; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid & \text{Comandi} \\
 & \texttt{for } x = E_1 \texttt{ to } E_2 \texttt{ do } C \mid \texttt{while } B \texttt{ do } C &
\end{array}
$$

dove $n \in \mathbb{N}$ (i numeri naturali), $x \in \mathit{Var}$ (insieme numerabile di variabili) e $b \in \mathit{Bool} = \{tt, \mathit{ff}\}$[4] (valore di verità).

Chiameremo *WHILE* il linguaggio definito dalla grammatica BNF definita sopra[5]; chiameremo invece *FOR* il linguaggio risultante dall'omissione del comando `while` B `do` C nella definizione della sintassi.

Semantica (ovvero calcolare è dimostrare)

Prima di definire la dinamica del nostro linguaggio abbiamo bisogno di un paio di nozioni ausiliarie. Rappresentiamo la memoria tramite una funzione (da sottoinsiemi *finiti* di *Var* nei numeri naturali, tanto in un programma ci possono essere solo un numero finito di variabili)

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

e definiamo l'operazione di *aggiornamento* tramite la funzione, o meglio il funzionale a tre argomenti

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

[4]Poiché non vi sono comandi di lettura né di scrittura su periferiche, assumeremo nel seguito che i programmi abbiano già la memoria inizializzata con i dati di ingresso (cioè che abbiamo una σ iniziale, vedi sotto) e che i risultati finali siano leggibili in memoria.

[5]Di solito si indica con *WHILE* il linguaggio definito dalla grammatica di cui sopra, privata della produzione relativa al comando `for` $x = E_1$ `to` E_2 `do` C. Avere o non avere comandi di questo tipo non crea alcun problema, in quanto essi sono "esprimibili" mediante un oppotuno comando di tipo `while` B `do` C.

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

Il significato, o *semantica* di un'espressione aritmetica è data dalla seguente funzione, definita ovunque. Seguendo la tradizione scriveremo il suo argomento principale, l'espressione aritmetica, tra le parentesi $[\![$ e $]\!]$, cui viene giustapposto il secondo argomento, cioè la memoria in cui l'espressione va valutata.

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

$$\begin{array}{lcl} \mathcal{E}[\![n]\!]\sigma & = & n \\ \mathcal{E}[\![x]\!]\sigma & = & \sigma(x) \\ \mathcal{E}[\![E_1 + E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \ \mathit{pi\acute{u}} \ \mathcal{E}[\![E_2]\!]\sigma \\ \mathcal{E}[\![E_1 \times E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \ \mathit{per} \ \mathcal{E}[\![E_2]\!]\sigma \\ \mathcal{E}[\![E_1 - E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \ \mathit{meno} \ \mathcal{E}[\![E_2]\!]\sigma \end{array}$$

dove *più, meno, per* sono "vere" funzioni da coppie di naturali a numeri naturali, a differenza di $+, -, \times$ che sono solo simboli (*tokens*) del nostro linguaggio di programmazione. Poiché i nostri dati sono solo numeri naturali, la funzione *meno* che impieghiamo è la funzione, di solito chiamata *meno limitato*, che si comporta come aspettato quando il minuendo è maggiore o uguale al sottraendo e restituisce 0 altrimenti.

Invece, nella prima equazione, non abbiamo fatto distinzione tra i *numerali* e i *numeri naturali* e li abbiamo rappresentati con gli stessi simboli; pur non essendo del tutto corretto, ciò viene comunemente fatto e non dovrebbe creare problemi perché di solito è chiaro quando n rappresenta un numero e quando un simbolo del linguaggio.

La semantica di un'espressione booleana è data dalla seguente funzione ovunque definita, che usa la $\mathcal{E}$ appena introdotta.

$$\mathcal{B}[\![-]\!]- : B \times (\mathit{Var} \to \mathbb{N}) \to \mathit{Bool}$$

$$\begin{array}{lcl} \mathcal{B}[\![t]\!]\sigma & = & \mathit{tt} \\ \mathcal{B}[\![f]\!]\sigma & = & \mathit{ff} \\ \mathcal{B}[\![E_1 < E_2]\!]\sigma & = & \mathcal{E}[\![E_1]\!]\sigma \ \mathit{minore} \ \mathcal{E}[\![E_2]\!]\sigma \\ \mathcal{B}[\![\neg B]\!]\sigma & = & \mathit{not} \ \mathcal{B}[\![B]\!]\sigma \\ \mathcal{B}[\![B_1 \vee B_2]\!]\sigma & = & \mathcal{B}[\![B_1]\!]\sigma \ \mathit{or} \ \mathcal{B}[\![B_2]\!]\sigma \end{array}$$

dove *minore*, *not*, *or* sono "vere" funzioni.

Lo stile di definizione seguito per dar semantica alle espressioni va sotto il nome di stile *denotazionale* e si propone di associare una funzione a ciascun operatore, o più in generale, a ciascun costrutto del linguaggio. La definizione di un costrutto composto viene data in termini dei suoi componenti, e per questa proprietà a volte questo stile viene anche detto *composizionale*. Adesso daremo la semantica dei comandi usando un approccio *operazionale*, guidato dalla sintassi. In effetti, andremo a definire una sorta di macchina astratta, che procede per passi discreti, apportando modifiche discrete sulle sue configurazioni, o stati, che possiamo immaginare come coppie $\langle comando, \sigma \rangle$. Questa macchina astratta, che definisce in modo intensionale la semantica di un linguaggio di programmazione o di un sistema, va sotto il nome di *sistema di transizioni* e formalmente è una coppia $(\Gamma, \rightarrow)$ dove

- Γ è la classe delle *configurazioni* (nel nostro caso la classe delle coppie $\langle c \in C, \sigma \rangle$, dove σ è definita per ogni variabile che appaia in *comando*; la coppia con il comando c vuoto ($\neq$ `skip`) è abbreviata da σ).
- $\rightarrow \subseteq \Gamma \times \Gamma$ è una funzione, detta di *transizione*.

Noi seguiremo un approccio detto SOS (Structural Operational Semantics) (o *small-step*), in cui ciascuna transizione

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

rappresenta un singolo passo di computazione, ovvero il fatto che la macchina ha attraversato lo stato $\langle c, \sigma \rangle$ e transisce nello stato $\langle c', \sigma' \rangle$ a causa di una sua certa attività. [6]

In maniera del tutto simile a quanto fatto con le macchine di Turing, si può allora definire una *computazione* come la chiusura riflessiva e transitiva della funzione di transizione, formalmente, una *computazione* è

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

Una computazione termina con successo, o converge, se $\langle c, \sigma \rangle \rightarrow^* \sigma'$.

[6] Un approccio alternativo consiste nel definire una semantica *naturale* o *big-step*, in cui le transizioni hanno tutte la forma

$$\langle c \in C, \sigma \rangle \Rightarrow \sigma'$$

cioè σ' memorizza le modifiche prodotte dall'intera esecuzione del comando c nella memoria iniziale σ. Un'intera computazione viene allora vista come la deduzione della transizione in questione. Potremmo però trovarci nell'impossibilità di dimostare l'esistenza di una transizione per uno stato: in questo caso avremmo o una situazione di stallo, o una situazione di non-terminazione della computazione, quando la deduzione non sia finita.

Si noti che la semantica operazionale di un programma è allora data dal grafo $(\Gamma, \rightarrow)$, mentre le computazioni danno una descrizione dei passi necessari al calcolo con granularità fine quanto si voglia.

In questo approccio, l'attività svolta dalla macchina quando transisce da una configurazione nella successiva viene a volte resa esplicita, etichettando la funzione di transizione e ottenendo così i *sistemi di transizioni etichettati*. [7] Un'ulteriore estensione piuttosto comune riguarda l'arricchimento della coppia $(\Gamma, \rightarrow)$ con un insieme di configurazioni terminali, ovvero di stati finali (a volte si aggiunge alla coppia anche una configurazione iniziale, da cui si assume partano tutte le computazioni). Nel nostro caso, il sistema di transizioni che useremo avrà come configurazioni finali le coppie formate da un comando vuoto e una memoria σ, che abbiamo già abbreviato con σ.

È evidente come gli stili di definizione denotazionale e operazionale siano differenti, in quanto non c'è modo immediato di comporre il "risultato" di un'esecuzione con quello di un'altra esecuzione. Tuttavia, anche l'approccio operazionale è in larga misura composizionale, per il modo che seguiamo nel definire la classe delle transizioni. Infatti, di seguito introduciamo un insieme di assiomi e regole di inferenza, che inducendo sulla sintassi dei comandi, ci permettono di dedurre tutte e sole le transizioni (ovvero prendiamo la minima classe che contiene tutte le instanze degli assiomi ed è chiusa rispetto alle regole di inferenza).

$$\frac{-}{\langle \texttt{skip}, \sigma\rangle \rightarrow \sigma} \qquad \frac{-}{\langle x := E, \sigma\rangle \rightarrow \sigma[n/x]},\ \text{se } \mathcal{E}[\![E]\!]\sigma = n$$

$$\frac{\langle C_1, \sigma\rangle \rightarrow \langle C_1', \sigma'\rangle}{\langle C_1; C_2, \sigma\rangle \rightarrow \langle C_1'; C_2, \sigma'\rangle} \qquad \frac{\langle C_1, \sigma\rangle \rightarrow \sigma'}{\langle C_1; C_2, \sigma\rangle \rightarrow \langle C_2, \sigma'\rangle}$$

$$\frac{-}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \rightarrow \langle C_1, \sigma\rangle},\ \text{se } \mathcal{B}[\![B]\!]\sigma = tt$$

$$\frac{-}{\langle \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \sigma\rangle \rightarrow \langle C_2, \sigma\rangle},\ \text{se } \mathcal{B}[\![B]\!]\sigma = ff$$

$$\frac{-}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \rightarrow \langle i := n_1;\ C; \texttt{for } i = n_1 + 1 \texttt{ to } n_2 \texttt{ do } C, \sigma\rangle}$$
$$\text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = ff \wedge \mathcal{E}[\![E_1]\!]\sigma = n_1 \wedge \mathcal{E}[\![E_2]\!]\sigma = n_2$$

$$\frac{-}{\langle \texttt{for } i = E_1 \texttt{ to } E_2 \texttt{ do } C, \sigma\rangle \rightarrow \sigma},\ \text{se } \mathcal{B}[\![E_2 < E_1]\!]\sigma = tt$$

$$\frac{-}{\langle \texttt{while } B \texttt{ do } C, \sigma\rangle \rightarrow \langle \texttt{if } B \texttt{ then } C;\ \texttt{while } B \texttt{ do } C \texttt{ else skip}, \sigma\rangle}$$

[7] Ovviamente questo può venir fatto anche nell'approccio naturale, componendo le etichette e rappresentando così le computazioni attraverso una sequenza di etichette.

1.4 Problemi e Funzioni

Ma cos'è un problema? Qualcosa del tipo: una domanda, con alcuni parametri da assegnare, ovvero una classe (non limitata) di domande, ciascuna delle quali vorremmo che avesse una risposta esatta e finita.

Nel nostro caso l'essenza di un problema è formalizzata come:

- calcolare una funzione, oppure
- decidere l'appartenenza a un dato insieme.

Definizione 1.4.1. Siano $\Sigma, \Sigma_0, \Sigma_1$ alfabeti, con $\#, \triangleright \notin \Sigma_0 \cup \Sigma_1, \Sigma_0 \cup \Sigma_1 \subset \Sigma$, e $f : \Sigma_0^* \to \Sigma_1^*$ una funzione. Allora si dice che una MdT $M = (Q, \Sigma, \delta, q_0)$ *calcola* f, oppure che f è *Turing-calcolabile* o semplicemente *T-calcolabile*, se e solamente se

$$\forall w \in \Sigma_0^*, z \in \Sigma_1^* : f(w) = z \text{ se e solamente se } (q_0, \trianglerighteq w) \to_M^* (h, \triangleright z\underline{\#})$$

In maniera del tutto analoga si definisce la nozione di *WHILE*-calcolabilità.

Definizione 1.4.2. Un comando C *calcola* $f : Var \to \mathbb{N}$, oppure f è *WHILE-calcolabile*, se e solamente se

$$\forall \sigma \in Var \to \mathbb{N} : f(x) = n \text{ se e solamente se } \langle C, \sigma \rangle \to^* \sigma' \text{ e } \sigma'(x) = n$$

(Si noti che la variabile x viene usata per memorizzare sia il valore di ingresso che quello di uscita di C.)

Domanda 1.4.1. Ma se la f fosse una funzione che opera su dati che non sono stringhe o memorie o numeri naturali?

Si supera la difficoltà per mezzo di opportune *codifiche* dei dati, ovvero funzioni biunivoche che siano *effettive* e *facili*[8], nel modo seguente.

1. Dato x in formato A codificalo come y in formato B.
2. Applica la MdT a y e se e quando ottieni z in formato B,
3. traduci z dal formato B al formato A.

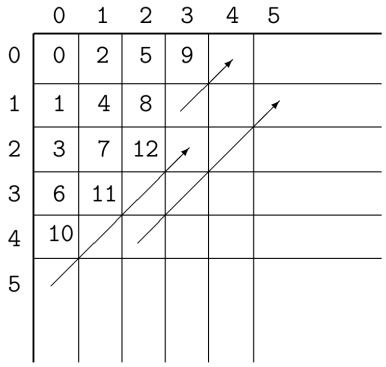
Quindi con il rischio, ma non il timore, di banalizzare considereremo da qui in avanti solo i *numeri naturali* come nostri dati; del resto lo abbiamo (quasi) già fatto nei linguaggi *WHILE* e *FOR*.

[8]Preciseremo in seguito cosa intendiamo per *effettivo* e per *facile*. Lo faremo considerando solo i numeri naturali, loro coppie ecc., quindi qui imbrogliamo un bel po', dando per buoni gli studi sull'argomento che si trovano in letteratura, e nella speranza di essere perdonati.

Esempio 1.4.1 (Coda di colomba)**.** La seguente funzione codifica coppie di naturali come un singolo naturale

$$(x, y) \mapsto \frac{1}{2}\left(x^2 + 2xy + y^2 + 3x + y\right)$$

ed è graficamente rappresentata come segue:



Risparmiamo al lettore sia la dimostrazione che questa funzione è biunivoca, sia lo sforzo per immaginare la funzione inversa, cioè la funzione di decodifica, che risulta essere definita così (si noti che nella colonna 0, l'elemento n-esimo è la somma dei primi n naturali a partire da 1):

$$n \mapsto \left(n - \frac{1}{2}k \times (k+1)\ ,\ k - \left(n - \frac{1}{2}k \times (k+1)\right)\right)$$

dove

$$k = \left\lfloor \frac{1}{2}\left(\sqrt{1 + 8 \times n} - 1\right) \right\rfloor$$

Definizione 1.4.3 (Funzione totale)**.** $f : A \to B$, sottoinsieme di $A \times B$ è una *funzione totale* se e solamente se

- $\forall a \in A,\ \exists b \in B : (a, b) \in f$ (*la funzione è definita ovunque*)
- $(a, b), (a, c) \in f \Rightarrow b = c$ (*unicità*)

Vediamo adesso un esempio di funzione un po' strana, nella cui definizione si menziona la congettura di Goldbach, definita più sotto, che l'autore sottopose nel 1742 in una lettera all'attenzione di Eulero, il quale non rispose mai; tuttora non si sa se la congettura sia vera o meno e per il momento essa è stata verificata "solo" sui numeri fino a 400 miliardi. Ritorneremo su questa funzione per esaminare le relazioni che intercorrono tra funzioni e algoritmi.

Esempio 1.4.2. $gb : \mathbb{N} \to \mathbb{N}$

$$gb(n) = \begin{cases} 1 & \text{se la congettura di Goldbach è vera} \\ 0 & \text{altrimenti} \end{cases}$$

Definizione 1.4.4 (Congettura di Goldbach)**.**

$$\forall m > 1 \text{ si ha } 2m = p_1 + p_2 \text{ con } p_1, p_2 \text{ primi.}$$

Definizione 1.4.5 (Funzione parziale)**.** $f : A \to B$ è una *funzione parziale* se è un sottoinsieme di $A \times B$ tale che

- $(a, b), (a, c) \in f \Rightarrow b = c$ (*unicità*), ovvero esiste al più un $b \in B$ tale che $f(a) = b$

e quindi non si richiede che f sia ovunque definita.

Notazione

Data una funzione $f : A \to B$:

- diremo che f è *definita* o *converge* su a (in simboli $f(a) \downarrow$) se $\exists b$ tale che $(a, b) \in f$ (cioè $f(a) = b$).
- altrimenti se $\nexists b$ tale che $(a, b) \in f$, diremo che f non è definita o *diverge* su a ($f(a) \uparrow$).

Chiamiamo anche

- *dominio* di f l'insieme $\{a \in A \mid f(a) \downarrow\}$ (coincidente con A solo se f è totale);
- *codominio* di f l'insieme B;
- *immagine* (o *range*) di f l'insieme $\{b \in B \mid \exists a \in A : f(a) = b\}$.

Infine ricordiamo che

- f è *iniettiva* se e solamente se $\forall a, b \in A.\ a \neq b$ implica $f(a) \neq f(b)$ (a volte nella terminologia nord-americana si dice che f è uno-a-uno).
- f è *surgettiva* se e solamente se $\forall b \in B.\ \exists a \in A$ tale che $f(a) = b$ (ovvero se l'immagine e il codominio di f coincidono).
- f è *biunivoca* se e solamente se è iniettiva e surgettiva.

(Si noti che la funzione iniettiva è invertibile sull'immagine.)

Abbiamo gli algoritmi, sia pure sotto forma di macchine di Turing o di programmi *WHILE* e abbiamo le funzioni (non necessariamente T- o *WHILE*-caclolabili). Allora siamo arrivati a porci una domanda cruciale.

Domanda 1.4.2. Qual è il rapporto tra funzioni e algoritmi?

Una funzione f è un insieme di coppie, cioè f associa all'argomento il risultato senza dire *come fare a calcolarlo*. Di conseguenza, *non ci sono due funzioni diverse che per ogni argomento restituiscono lo stesso risultato*, il che riscritto in termini insiemistici si legge: *non esistono due insiemi* diversi *che hanno gli stessi elementi*!

Un algoritmo (quando c'è!) invece specifica *come si calcola il risultato* a partire dall'argomento. In altre parole un algoritmo *calcola* o *rappresenta*, in modo *finito*, una funzione. È facile vedere che ci possono essere più algoritmi che calcolano la stessa funzione — saremo più precisi in seguito: qui basta considerare un programma *WHILE* e aggiungergli un po' di `skip`.

Infine, riprendiamo la funzione $gb(n)$ dell'esempio 1.4.2. Non sappiamo a tutt'oggi se la congettura di Goldbach vale. Eppure un algoritmo per calcolarla esiste! ma non sappiamo quale esso sia, o meglio, quali essi siano: quelli che danno come risultato 1 per tutti gli ingressi, o quelli che danno sempre come risultato 1? [9]

Finalmente siamo arrivati al nocciolo:

- Quali sono le funzioni calcolabili? Per ora sappiamo quali sono le T-calcolabili e le *WHILE*-calcolabili.
- Di quali proprietà godono?
- Esistono funzioni (totali/parziali), ovvero problemi, che non sono calcolabili? ovvero per cui si *dimostra* che non esiste algoritmo che le calcoli? Tra questi problemi non calcolabili, ce ne sono di interessanti?

In questa prima parte del corso studieremo algoritmi e funzioni, con maggior attenzione alle seconde. In termini classici, l'enfasi sarà sugli aspetti *estensionali*, perché ci occuperemo di ciò che è rappresentato (la semantica, il significato, la funzione) piuttosto di ciò che rappresenta (l'algoritmo, il programma): *cosa si calcola*, piuttosto che come si calcola.

La seconda parte del corso terrà invece in maggior considerazione gli algoritmi: *come si calcola*.

[9] Gli intuizionisti rifiutano questo discorso: una entità matematica esiste se e solamente se la puoi costruire, e qui non esisterebbe (ancora)! Ciò che viene escluso è il *tertium non datur* aristotelico.

1.5 Due approcci alla calcolabilità

Abbiamo visto due modelli, i quali sottolineano due punti di vista leggermente diversi e complementari per descrivere algoritmi. In entrambi gli approcci, una configurazione è la coppia

(istruzione corrente, stato della memoria)

mentre cambia il modo con cui si interpretano le istruzioni.

Nel primo punto di vista, che per semplicità chiameremo approccio *hardware*, un algoritmo è una macchina in cui *l'insieme delle istruzioni rappresenta l'architettura*. Il fatto che il programma stesso sia considerato come una macchina può sorprendere e richiede quindi una spiegazione ulteriore, prendendo in esame le macchine di Turing. In effetti, la funzione di transizione rappresenta un programma e l'hardware pare essere composto solo dal nastro e dalla testina, dotata di un meccanismo di spostamento. Tuttavia, in una macchina di Turing si cabla l'intero programma: ogni macchina ha un numero fissato di stati e di simboli ed è completamente definita dalla sua funzione di transizione. Infatti non appena cambiamo quest'ultima, definiamo una nuova macchina. In altre parole, una macchina di Turing è la realizzazione di un *unico* algoritmo: l'unica cosa che varia è il dato di ingresso.

Analogamente, ogni altro modello nell'approccio hardware ha una memoria che è infinita tout court o è indefinitamente espandibile, e ogni suo elemento realizza un singolo algoritmo. Più in generale, può addirittura accadere che si usino macchine via via più grandi al crescere delle dimensioni del problema, che però sono tra loro simili o uniformi (si veda la cosiddetta "circuit complexity"); vedremo un esempio di quanto detto subito prima della dimostrazione del teorema di Cook, in particolare quando useremo quella che viene detta tabella di computazione di una macchina di Turing (vedi pagina 107).

L'approccio software vede invece un algoritmo come un programma "interpretato" da un agente di calcolo, esso stesso una macchina (vedi il teorema di enumerazione 1.9.5), che può essere realizzata in:

- hardware (o firmware): la macchina fisica su cui "gira";
- software, cioè un programma a più basso livello: la macchina astratta che lo "esegue".

Anche in questo caso, la configurazione della macchina è, come prima, formata dalla coppia costituita da un puntatore all'istruzione corrente e dallo stato della memoria. A differenza dell'approccio hardware, qui i programmi *non*

sono parte integrante della macchina, ma sono contenuti nella memoria (cf. ancora il teorema 1.9.5). Di conseguenza, programmi più lunghi *non* richiedono agenti di calcolo di dimensioni maggiori: l'agente di calcolo o l'interprete è fisso e non cambiano solo i dati di ingresso, ma anche i programmi.

Tra i formalismi visti, i programmi *WHILE* hanno proprio quest'ultima caratteristica: l'interprete è (una qualunque realizzazione del)la semantica del linguaggio, la memoria è infinita, modellata attraverso la funzione σ, il contatore istruzioni è rappresentato dal programma che dobbiamo ancora eseguire (quest'ultimo punto è particolarmente evidente nell'approccio SOS alla semantica operazionale).

Introdurremo tra poco un terzo formalismo, da ascrivere tra quelli software: quello delle funzioni ricorsive, il quale è molto rilevante sia in informatica che nella teoria della calcolabilità classica, tanto che a volte questa è chiamata teoria delle *funzioni ricorsive*. Uso e ragione per l'introduzione dei tre principali formalismi che vedremo sono elencati qui sotto.

- Funzioni Ricorsive *CHIAREZZA*

  Sono la base per: Programmazione Funzionale; Semantica Denotazionale.

- Programmi *WHILE/FOR* *FAMILIARITÀ*

  Sono la base per: Programmazione Imperativa; Semantica Operazionale; Complessità (Loop Programs).

- Macchine di Turing *SEMPLICITÀ*

  Sono la base per: Descrizione Macchine; Algoritmi; Complessità.

Come già detto, i primi due paradigmi si possono classificare nell'approccio software, mentre il terzo è un paradigma più vicino all'hardware. Questa distinzione è largamente arbitraria e formalmente vaga, ma può spiegare perché vi siano differenti presentazioni della materia; inoltre può apparire sfuggente a questo punto del corso, ma verrà auspicabilmente chiarita nella sua seconda parte, ad esempio discutendo il già citato teorema di Cook.

Ovviamente tutta la trattazione che segue potrebbe essere basata su uno qualunque di questi formalismi, e su altri ancora che non abbiamo menzionato, tra cui ad esempio le grammatiche e sistemi di Post, gli algoritmi di Markov, il λ-calcolo, le macchine a registri (che vedremo di sfuggita), e molti altri ancora. Ciascuno di essi chiarisce in maggiore o minore misura i vari aspetti che andremo a considerare nel seguito senza per altro definire classi diverse di funzioni calcolabili o classi di complessità, come abbiamo già più volte ripetuto.

1.6 Funzioni ricorsive primitive

Cominciamo a vedere un paio di esempi di funzioni definite per ricorrenza che sono molto popolari. In seguito introdurremo la classe delle funzioni *ricorsive primitive*, della cui debolezza e del modo di sopperirvi si discuterà nel prossimo capitolo.

La prima funzione che consideriamo è il fattoriale, definita come spesso si fa, da una coppia di equazioni, una nel caso (base) in cui l'argomento sia 0, l'altra nel caso (ricorsivo) in cui l'argomento non sia nullo.

$$\mathit{Fattoriale} = \begin{cases} 0! & = & 1 \\ (x+1)! & = & (x+1) \times x! \end{cases}$$

Eccone una versione in *WHILE*, che "lascia" il valore di $x!$ nella variabile `fatt`.

```
fatt := 1;
while 0 < x do
    fatt := fatt * x;
    x := x - 1;
```

La seconda funzione, ben nota agli informatici (e ai matematici, botanici, fisici, architetti, ecc.) è la funzione di Fibonacci, la cui definizione data qui sotto ha, per ovvie ragioni, due casi base

$$\mathit{Fibonacci} = \begin{cases} \mathit{fib}(0) & = & 0 \\ \mathit{fib}(1) & = & 1 \\ \mathit{fib}(x+2) & = & \mathit{fib}(x+1) + \mathit{fib}(x) \end{cases}$$

La funzione di Fibonacci si può anche scrivere in forma chiusa, non ricorsiva:

$$\mathit{fib}(x) = \frac{\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^x - \frac{\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^x$$

Adesso proviamo a formalizzare i modi diversi che abbiamo usato per definire le funzioni *Fattoriale* e *Fibonacci*. Nel far ciò, usiamo quella che si chiama λ-notazione per individuare all'interno di un'espressione (scritta seguendo un'opportuna sintassi) che descrive una funzione quali sono i suoi argomenti. Scriveremo quindi

$$\lambda x, y, z.\ \mathit{espressione}$$

quando gli argomenti della *espressione* sono proprio x, y e z; si dice anche che x, y e z appaiono *legate* da λ in *espressione*. Invece, un qualsiasi altro

simbolo (di variabile) w che appaia in *espressione*, *non* sarà da considerarsi un argomento della funzione rappresentata da *espressione*; a volte un tale w viene chiamato *libero* in *espressione*.[10] Ad esempio,

$$\lambda x, y.\ x + y \qquad \text{(o più precisamente } \lambda x.\lambda y.\ x + y)$$

è ovviamente la funzione che somma x ad y, una volta che si sia interpretato il simbolo + come si suole. In questo caso potremmo anche scrivere

$$somma(x, y) = x + y$$

dando così il nome *somma* alla funzione, magari per usi futuri (si confrontino questi due modi di definire funzioni con nome o senza nome con i modi di definire e costruire funzioni, in voga nei linguaggi funzionali, tipo ML).

Introduciamo adesso la prima classe di funzioni che ci interessano. Per seguire l'uso corrente e per non appesantire eccessivamente la notazione, di seguito mescoleremo la sintassi e la semantica, dicendo che definiamo una funzione f dai naturali ai naturali, mentre in realtà stiamo definendo un *algoritmo* che *rappresenta* la "vera" funzione f.

Definizione 1.6.1. La classe delle funzioni *ricorsive primitive* è la minima classe di funzioni $\mathcal{C}$ da $\mathbb{N}^n$, $n > 0$, in $\mathbb{N}$ cui appartengono:

I *Zero* $\lambda x_1, x_2, ..., x_k.\ 0$ con $k \geq 0$

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

III *Identità* (o *proiezione*) $\lambda x_1, \ldots, x_k.\ x_i$ con $1 \leq i \leq k$

dette anche *schemi primitivi di base*, e che è chiusa per:

IV *Composizione*

Se $g_1, \ldots, g_k \in \mathcal{C}$ sono funzioni in m variabili, ed $h \in \mathcal{C}$ una funzione in k variabili, appartiene a $\mathcal{C}$ anche la loro composizione

$$\lambda x_1, \ldots, x_m.\ h(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

V *Ricorsione Primitiva*

Se $h \in \mathcal{C}$ è una funzione in $k + 1$ variabili, $g \in \mathcal{C}$ è una funzione in $k - 1$ variabili, allora appartiene a $\mathcal{C}$ anche la funzione f in k variabili definita da:

$$\begin{cases} f(0, x_2, \ldots, x_k) & = & g(x_2, \ldots, x_k) \\ f(x_1 + 1, x_2, \ldots, x_k) & = & h(x_1, f(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k) \end{cases}$$

[10] Più precisamente si dovrebbe parlare di apparizioni, o con un calco dall'inglese, *occorrenze* legate o libere di quelle o questa variabile. Nell'ultimo caso, la funzione descritta da *espressione* è da interpretarsi come una funzione *parametrica* in w, nel senso dato alla parola nei corsi di Analisi.

Si noti che, se $k = 1$, allora g ha 0 argomenti ed è ricorsiva primitiva in quanto è una costante. Così come avviene per le MdT, anche la definizione della classe delle funzioni ricorsive primitive ha moltissime varianti, tutte equivalenti tra loro, nel senso che le classi che esse definiscono coincidono tutte con la classe $\mathcal{C}$. Per esempio, invece di avere lo schema I si può avere lo schema che introduce le costanti:

$$\text{I}' \ \textit{Costanti} \ \lambda x_1, \ldots, x_k.\ m \text{ con } k, m \geq 0$$

Chiaramente, componendo un numero opportuno di volte la funzione successore a partire dalla funzione costante 0 si ottengono tutte le costanti m introdotte in un sol colpo dallo schema appena definito. Altre varianti piuttosto comuni riguardano la posizione in cui appare la chiamata ricorsiva alla funzione f nella parte destra della seconda equazione nello schema V.

Poiché $\mathcal{C}$ è la minima classe che soddisfa le condizioni espresse sopra, affinché f sia ricorsiva primitiva, occorre e basta che ci sia una successione finita, o *derivazione*, della seguente forma

$$f_1, f_2, \ldots, f_n$$

tale che $f = f_n$ e $\forall i$ tale che $1 \leq i \leq n$ vale uno dei seguenti casi:

- $f_i \in \mathcal{C}$ per I-III (ovvero f_i è definita secondo i casi base);
- f_i è ottenibile mediante applicazione delle regole IV, V da $f_j,\ j < i$ (ovvero f_i è definita da funzioni già definite precedentemente).

Esempio 1.6.1. Vogliamo definire $f_5 = f = \lambda x, y.\ x + y$ di cui una possibile derivazione è la seguente, in cui abbiamo indicato accanto ad ogni f_i la regola applicata per ottenerla.

$$\begin{array}{ll}
f_1 = \lambda x.\ x & \textit{III} \\
f_2 = \lambda x.\ x + 1 & \textit{II} \\
f_3 = \lambda x_1, x_2, x_3.\ x_2 & \textit{III} \\
f_4 = f_2(f_3(x_1, x_2, x_3)) & \textit{IV} \\
\begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) \end{cases} & \textit{V}
\end{array}$$

Vediamo adesso come si calcola $f_5(2, 3)$, cioè la somma di $2 + 3$. Nel passare da un rigo al successivo, espandiamo la "chiamata di funzione" sottolineata (detta *redex*). Si noti che usiamo la regola di valutazione *interna sinistra*: il redex prescelto è quello più interno all'espressione da valutare più a sinistra.

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

Notare che invece di valutare prima le funzioni interne, avremmo potuto valutare prima le funzioni esterne tuttavia per far questo bisogna poter legare a una variabile x non solo un numero naturale, ma anche un'intera espressione. Questa regola di valutazione viene chiamata *esterna*; usiamola nell'esempio di sopra.

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

In realtà si può dimostrare che la scelta del redex non è determinante, ovvero che le due regole di valutazione sono equivalenti.[11] In questo caso, le riduzioni

[11] Questo non è più vero in questi termini per la classe di funzioni che introdurremo tra due capitoli, in quanto la regola di valutazione esterna potrebbe portare a un risultato anche in presenza di un calcolo non terminante, eseguito usando la regola interna sinistra. In effetti, c'è un teorema che assicura che tutte le volte che, applicando la regola interna, si arriva a un risultato, si ottiene il medesimo risultato con la regola esterna; mentre il viceversa può non essere vero. Più in generale, si può dire che tale risultato è raggiungibile anche *senza scegliere a priori alcuna regola* per selezionare il redex.

necessarie, sia con la regola interna sinistra che con quella destra sono 9, se non contiamo le applicazioni della funzione primitiva successore. In generale questo può non essere vero e per rendersene conto si consideri la seguente espressione

$$f_3(f_5(2,3),0,f_5(2,3))$$

la cui valutazione con la regola esterna richiede una sola riduzione e 19 con la regola interna. (È sempre vero che in questo formalismo la regola esterna è più efficiente di quella interna, ovvero richiede meno o al più lo stesso numero di passi di calcolo?)

Naturalmente per noi informatici è di grande importanza usare una regola di valutazione che minimizzi il numero di passi necessari al calcolo e anche lo spazio necessario a conservare i risultati intermedi – si pensi ad esempio alla realizzazione di interpreti o compilatori efficienti per linguaggi funzionali (e le regole di valutazione *by value*, *by need*, *lazy* ecc.).

Concludiamo l'esempio aggiungendo alla derivazione data sopra per la funzione di somma la seguente clausola, che ci permette di ottenere una definizione per la funzione che raddoppia il suo argomento $f_6 = \lambda x.\ 2 \times x$

$$f_6 = f_5(f_1(x), f_1(x)) \quad IV$$

Bene, abbiamo un bel formalismo, semplice, elegante, anche se piuttosto verboso. È facile vedere che le usuali funzioni ($x \times y$, x^y, $x - y$, ecc.) sono tutte ricorsive primitive. Adesso rendiamoci la vita più comoda usando identificatori $x, y, z, \ldots$ al posto di $x_1, x_2, \ldots$ (quanti simboli servono per rappresentare le x_i?) e permettendo la ricorsione in posizioni diverse dalla seconda. Definiamo prima le funzioni proiezione (a due posti) che restituiscono il primo e il secondo argomento e quella che restituisce il predecessore del suo argomento, se maggiore di 0, altrimenti restituisce 0 (e chiamiamola *pred*, dandogli un nome appena più evocativo di quelli usati finora e semplifichiamoci la vita stipulando che 0 è una funzione a nessun argomento):

$$f_7(x,y) = y \qquad f_8(x,y) = x \qquad \begin{cases} pred(0) & = & 0 \\ pred(x+1) & = & f_8(x, pred(x)) \end{cases}$$

Poi introduciamo la funzione ausiliaria che sottrae (limitatamente) 1 al secondo dei suoi tre argomenti per definire l'ultima funzione ausiliaria che sottrae (limitatamente) il primo argomento al secondo:

$$f_9(x,y,z) = pred(f_3(x,y,z)) \qquad \begin{cases} f_{10}(0,y) & = & f_1(y) \\ f_{10}(x+1,y) & = & f_9(x, f_{10}(x,y), y) \end{cases}$$

Finalmente ci siamo, il meno limitato è:

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

Il lettore avrà certamente notato un'ulteriore semplificazione: nell'ultima definizione abbiamo usato la notazione infissa che è assai più comune. Semplifichiamoci ancora la vita: diamo per intese le necessarie composizioni e proiezioni e guardiamo alle seguenti definizioni, in cui riappare la definizione della somma, indicata da + (da non confondersi con il successore +1), già vista in tutti i dettagli nell'esempio 1.6.1. Le altre due funzioni sono il prodotto e l'esponente, in cui stipuliamo che $0^0 = 1$. [12] Una rapida ispezione, e forse un paio di esempi, dovrebbero convincerci che il "+" generalizza il successore +1, nel senso che questo viene applicato x volte a y, e che il $\times$ generalizza il + e infine che l'esponente generalizza il $\times$ (ci sarà una funzione che generalizza l'esponente? si veda la nota a pagina 27).

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

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

Per scopi futuri, definiamo anche le relazioni ricorsive primitive.

Definizione 1.6.2. Diciamo che la relazione $P(x_1, ..., x_k) \subseteq \mathbb{N}^k$ è *ricorsiva primitiva* se lo è la sua funzione caratteristica χ_P definita come:

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

Come esempio di relazione ricorsiva primitiva vediamo l'eguaglianza, definendo la sua funzione caratteristica $\chi_=$ (ricorrendo su due argomenti contemporaneamente):

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

Inoltre ci servirà sapere che

- $R = \{x \in \mathbb{N} \mid x \text{ è un numero primo}\}$ è ricorsiva primitiva
- (unica fattorizzazione) Se $p_0 < \cdots < p_k \cdots \in R$ (sono numeri primi), allora $\forall x \in \mathbb{N}$ esiste un numero *finito* di esponenti $x_i \neq 0$ tali che
$$x = p_0^{x_0} p_1^{x_1} \dots p_k^{x_k} \dots$$
- la funzione $(x)_i$ che restituisce l'esponente dell'i-esimo fattore p_i della fattorizzazione di x è ricorsiva primitiva.

[12] In alternativa a tale ipotesi sventurata si potrebbe porre come caso base $(x+1)^0 = 1$, ma allora la funzione sarebbe indefinita sullo 0 (forse però i nostri professori di matematica sarebbero meno scandalizzati).

Una conseguenza assai interessante è che ogni sequenza di numeri naturali $n_0 n_1 \dots n_k$ può essere codificata *univocamente* come $n = p_0^{n_0+1} p_1^{n_1+1} \dots p_k^{n_k+1}$ (con p_i primi), ovvero come il prodotto di un numero *finito* di fattori, e viceversa. In altre parole, data la sequenza $n_0 n_1 \dots n_k$ esiste un unico n che verifica l'uguaglianza $n = p_0^{n_0+1} p_1^{n_1+1} \dots p_k^{n_k+1}$ e viceversa. Questa è la base per dimostrare che le *funzioni di codifica sono ricorsive primitive*.

Vediamo ora, intuitivamente e non in dettaglio, come si può sfruttare questo teorema per costruire una codifica delle macchine di Turing e delle loro computazioni. A essere precisi *non* faremo vedere una codifica, ma delineeremo a spanne una funzione che è iniettiva, ma *non* surgettiva. Ritorneremo più avanti su questo fatto, legato a pigrizia e desiderio di tenere le cose il più semplici possibile. Sia

$$M = (Q, \Sigma, \delta, q_0)$$

una MdT con

$$Q = \{q_0, \dots, q_n\} \text{ e } \Sigma = \{\sigma_0, \dots, \sigma_m\}$$

Ogni quintupla $(q_i, \sigma_j, q_k, \sigma_l, D) \in \delta$ è codificata come

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

dove $p_0 < \dots < p_4$ sono numeri primi. Inoltre, consideriamo i simboli di direzione D come simboli veri e propri: $L = \sigma_{m+2}$, $R = \sigma_{m+3}$, $- = \sigma_{m+4}$ e poniamo allora $m_D \in \{m_L = m+2, m_R = m+3, m_- = m+4\}$. Infine, facciamo in modo che lo stato "speciale" h sia identificato come lo stato q_{n+1}.

Grazie al teorema di unica fattorizzazione, a ogni quintupla è associato un *solo* intero e viceversa; questa osservazione è la base per dimostrare che la funzione che stiamo proponendo è iniettiva.

Ora ordiniamo l'insieme di quintuple che caratterizza M, ponendo ad esempio $(q_i, \sigma_j, q_k, \sigma_l, D)$ minore di $(q_{i'}, \sigma_{j'}, q_{k'}, \sigma_{l'}, D')$ se $i < i'$ o se $i = i'$ e $j < j'$, e così via. A questo punto, abbiamo una successione di quintuple e ciascuna di esse viene codificata come un numero, ottenendo una successione $a_0 a_1 \dots a_n$ di numeri naturali, tali che $a_i \neq a_j$ se $i \neq j$. Adesso anche questa successione viene codificata, e si ottiene finalmente il numero i, detto anche *indice*, che vogliamo associare alla macchina M.

Come già accennato sopra, è immediato rendersi conto che la funzione proposta non è surgettiva: non sempre da un numero i posso recuperare una macchina M; ma il problema si aggira facilmente. Infatti, uno decodifica l'indice i; se il risultato ottenuto è una MdT, bene; altrimenti i non è nell'immagine (ovvero non è un vero e proprio indice) e butto via tutto, oppure restituisco un valore speciale. In altre parole, ci si riduce a considerare la coppia codifica/decodifica come operanti rispettivamente dalle MdT a $\mathbb{N}$ e

dalla *immagine* della codifica alle MdT (oppure da $\mathbb{N}$ alle MdT più il valore speciale di cui sopra). Si può certamente fare di meglio e avere una vera funzione di codifica che sia quindi biunivoca, al prezzo di una definizione molto astuta (e complicata): l'ha già fatto per noi Kurt Gödel, seppure non per le MdT, tanto che il procedimento sopra delineato viene chiamato *gödelizzazione* e *numero di Gödel* di M il numero così ottenuto dalla macchina M data.

Basti qui notare che abbiamo un modo per *enumerare* le macchine di Turing, ovvero di metterle in lista basandoci *unicamente* sui simboli usati per definir(e le loro quintup)le; inoltre il modo è "facile", perché le funzioni ricorsive primitive lo sono, in un senso che sarà più chiaro nel seguito.

Naturalmente il procedimento accennato sopra può essere applicato alle configurazioni e poi anche alle computazioni, che non sono altro che successioni di configurazioni. Un numero naturale allora può essere interpretato come la *codifica di un'intera computazione*! Del resto, un programma caricato in memoria può esser visto come una successione di bit, la quale rappresenta un numero naturale e così la sequenza di configurazioni attraversate.

Ovviamente il medesimo procedimento di enumerazione può essere applicato anche ai programmi *FOR* e *WHILE*, alle funzioni ricorsive primitive e a quelle μ-ricorsive che introdurremo nel prossimo capitolo, nonché a ogni formalismo per rappresentare funzioni che rispetti i vincoli posti nel capitolo 1.1. Quindi la codifica che ci ha regalato Gödel è davvero un'arma potentissima, che permette di enumerare, o se volete di digitalizzare le rappresentazioni delle funzioni e il loro calcolo (e anche di altre realtà, per esempio immagini o musica!).

Ritorniamo ora alle nostre funzioni definite per ricorsione primitiva. È facile verificare con un ragionamento induttivo che tutte le funzioni definibili mediante gli schemi di ricorsione primitiva sono *totali*.

Dal punto di vista informatico è interessante notare che le funzioni ricorsive primitive e il linguaggio *FOR* sono equivalenti, nel senso che data una funzione ricorsiva primitiva si può scrivere un programma *FOR* che, con gli stessi argomenti, produce lo stesso risultato e viceversa.

Domanda 1.6.1. Abbiamo trovato il formalismo giusto, che esprime tutte le funzioni calcolabili? e per di più solo quelle totali? Purtroppo **NO**.

Infatti esiste la funzione di Ackermann, che *non è definibile* mediante gli schemi di ricorsione primitiva I-V, che è totale e ha una definizione che intuitivamente è accettabilissima. Vediamola, osservando che una sola delle regole seguenti è applicabile una volta fissati gli argomenti x, y, z.[13]

[13]C'è una variante più semplice di questa funzione in cui si può eliminare la variabile y,

$$
\begin{array}{lcl}
A(0,0,y) & = & y \\
A(0,x+1,y) & = & A(0,x,y)+1 \\
A(1,0,y) & = & 0 \\
A(z+2,0,y) & = & 1 \\
A(z+1,x+1,y) & = & A(z,A(z+1,x,y),y)
\end{array}
$$

Nota: C'è una "doppia" ricorsione, ma tutti i valori su cui si ricorre decrescono, e quindi i valori di $A(z,x,y)$ sono definiti in termini di un numero *finito* di valori della funzione A applicata ad argomenti z' e x' tali che $z' \leq z$ e $x' < x$. Quindi intuitivamente A è calcolabile.

Ma cosa calcola A? Una sorta di esponenziale generalizzato (si tenga conto che il $+$ è la funzione *successore* generalizzata, il $\times$ è il $+$ generalizzato, la funzione esponente è il $\times$ generalizzato).

$$
\begin{array}{lcl}
A(0,x,y) & = & y+x \\
A(1,x,y) & = & y \times x \\
A(2,x,y) & = & y^x \\
A(3,x,y) & = & \underbrace{y^{\cdot^{\cdot^{y}}}}_{x \text{ volte}}
\end{array}
$$

Sfortunatamente, la funzione di Ackermann cresce più velocemente di ogni funzione ricorsiva primitiva. Per cui vale il seguente teorema, che non dimostriamo; intuitivamente, la dimostrazione si basa sul fatto che questa funzione mette in piedi un numero di chiamate a sé stessa maggiori di quante ne possa fare qualsiasi funzione ricorsiva primitiva.

Teorema 1.6.1. *La funzione di Ackermann non è ricorsiva primitiva.*

Evidentemente tra le funzioni che si riescono a definire secondo gli schemi di ricorsione primitiva I–V (e che quindi sono calcolabili e totali) ne manca almeno una. Di più e peggio: ne mancano moltissime, ad esempio tutte quelle che si ottengono dalla funzione di Ackermann sommandogli una costante. Ma non si potrebbe aggiungere a forza la funzione di Ackermann tra le ricorsive primitive, o meglio estendere i suoi schemi con quella doppia ricorsione e ottenere un formalismo che esprime *solo* le funzioni totali e le esprime *tutte*? La questione diventa allora la seguente.

---

dopo averla considerata identicamente uguale a 1:

$$
\begin{array}{lcl}
A'(0,x) & = & x+1 \\
A'(z+1,0) & = & A'(z,1) \\
A'(z+1,x+1) & = & A'(z,A'(z+1,x))
\end{array}
$$

Domanda 1.6.2. Esiste un formalismo capace di esprimere *tutte e sole* le funzioni totali? Sfortunatamente, o meglio fortunatissimamente **NO**.

Per vederlo, definiremo nel capitolo seguente una tecnica di dimostrazione, detta *diagonalizzazione* e la useremo nel caso delle funzioni ricorsive primitive. Deve essere ben chiaro però che questo metodo si applica a *qualsiasi* formalismo che esprime *solo* funzioni totali.

1.7 Diagonalizzazione

Vediamo adesso una tecnica basilare della teoria della calcolabilità, che va sotto il nome di *diagonalizzazione*. Essa è strettamente legata alla dimostrazione che Cantor diede della non numerabilità dell'insieme dei sottoinsiemi dei numeri naturali. Il modo con cui applichiamo la diagonalizzazione alle funzioni ricorsive primitive è in realtà indipendente da questo formalismo, essendo invece del tutto generale: si applica infatti a *tutti* i formalismi con cui si possano definire *solo* funzioni totali. Di seguito usiamo questa tecnica per dimostrare che le funzioni ricorsive primitive non sono tutte le funzioni calcolabili totali in quanto ne manca almeno una che intuitivamente lo è. Inoltre, la stessa dimostrazione fa vedere che comunque si estenda la classe delle funzioni ricorsive primitive a contenere *solo* funzioni totali, si ricasca nello stesso problema: si può costruire una funzione intuitivamente calcolabile non esprimibile con quella estensione. Di conseguenza, la classe delle funzioni ricorsive primitive, o qualunque classe che la estenda con solo funzioni totali calcolabili, non conterrà *tutte* le funzioni intuitivamente calcolabili. Quella che segue non è una vera dimostrazione, ma solo una traccia di come si dovrebbe procedere.

i) Ogni derivazione di una funzione ricorsiva primitiva è una stringa *finita* di simboli presi da un alfabeto *finito*. Quindi tali rappresentazioni si possono enumerare, per esempio con la funzione di Gödel, e indichiamo con f_n la funzione definita dalla n-esima derivazione.

ii) Definisci $g(x) = f_x(x) + 1$. Questa è effettivamente calcolabile: prendi la x-esima definizione, applicala avendo come argomento il suo stesso indice x, trova il risultato e sommagli 1. (Si noti che la scelta dell'algoritmo dipende dall'argomento.) Inoltre, è facile vedere che la funzione g è totale.

iii) La g non si trova nella lista delle funzioni ricorsive primitive, perché $\forall n.\ g(n) \neq f_n(n)$, e quindi $\forall n.\ g \neq f_n$, o meglio la funzione calcolata dalla g su n restituisce un valore diverso dalla funzione calcolata dalla f_n su n.

Come già anticipato, l'argomento usato sopra per le funzioni ricorsive primitive si applica ad *ogni* formalismo che definisca *solo* funzioni totali: basta costruire l'elenco delle funzioni definite in quel formalismo come fatto dal passo (i) di sopra, enumerandole come fatto nel capitolo precedente per le MdT, e poi diagonalizzare come nel passo (ii): la funzione così definita non appare nell'elenco costruito.

Quindi siamo obbligati a considerare anche *funzioni parziali*, che indicheremo spesso mediante lettere dell'alfabeto greco, quali φ e ψ. Per fortuna, la diagonalizzazione non si applica alle funzioni parziali. Infatti sia ψ_n la funzione con n-esima definizione, cioè che appare in posizione n-esima nella lista (per esempio calcolata dall'n-esima MdT), e proviamo a diagonalizzare. Definiamo quindi

$$\varphi(x) = \psi_x(x) + 1$$

Supponiamo adesso che φ sia rappresentata dall'n-esimo algoritmo: non posso tuttavia concludere $\varphi \neq \psi_n$ perché $\psi_n(n)$ può non essere definita!

Ma se prendessi proprio degli algoritmi che definiscono funzioni totali per applicare la diagonalizzazione? In questo modo otterrei nuovamente una funzione che non trovo nella lista. Vedremo alla fine di questa parte del corso che questa cosa non si può fare effettivamente, cioè che *non esiste* un algoritmo che permetta di scegliere nella lista proprio le definizioni di funzioni totali.

Inoltre, fortunatamente le funzioni parziali hanno senso. Per esempio, si consideri la funzione seguente, che è definita solo se $y \neq 0$

$$div(x, y) = \lfloor x \div y \rfloor$$

C'è ancora un sospetto da fugare. Si potrebbero estendere tutte le funzioni parziali a funzioni totali, come facciamo nell'esempio seguente rendendo totale la funzione *div* alla funzione *div** come suggerito più avanti? La risposta è **NO**, perché, come vedremo alla fine della prima parte del corso, *non sempre* c'è un algoritmo che calcola la funzione estesa (nel nostro caso *div**).

Esempio 1.7.1. Il primo modo è quello di definire accuratamente il dominio della funzione, per esempio ponendo

$$div : \mathbb{N} \times \mathbb{N}\backslash\{0\} \to \mathbb{N}$$

Però la funzione risultante non copre tutte le coppie di naturali. Possiamo far meglio: posto $* \notin \mathbb{N}$, abbiamo ancora una funzione su tutte le coppie di naturali [14]

$$div : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{*\}$$

$$div^*(x, y) = \begin{cases} div(x, y) & \text{se } y \neq 0 \\ * & \text{se } y = 0 \end{cases}$$

Poiché non possiamo e nemmeno vogliamo liberarci della parzialità, bisogna trovare il modo per poter definire anche funzioni parziali.

[14] Il suo codominio non sono però i naturali; se volessi essere pignolo allora potrei definire $div^*(x, 0) = 0$, facendo inorridire i nostri professori di matematica delle superiori.

1.8 Funzioni ricorsive generali

Arricchiamo di seguito gli schemi per la definizione delle funzioni ricorsive primitive con un nuovo schema, attraverso il quale è possibile esprimere anche funzioni parziali. In esso, si fa uso dell'operatore μ, detto di *minimizzazione*, il quale applicato a un insieme di numeri naturali ne restituisce il minimo (se c'è! ovvero se l'insieme in questione non è vuoto).

Definizione 1.8.1 (Funzioni μ-ricorsive)**.** La classe delle *funzioni μ-ricorsive* (o *ricorsive generali*) è la minima classe $\mathcal{R}$ tale che soddisfa le condizioni

I-V per le ricorsive primitive e

VI (*Minimizzazione*). Se $\varphi(x_1, ..., x_n, y) \in \mathcal{R}$ in $n+1$ variabili, allora la funzione ψ in n variabili è in $\mathcal{R}$ se è definita da

$$\psi(x_1, .., x_n) = \mu y[\varphi(x_1, \ldots, x_n, y) = 0 \text{ e } \forall z \leq y.\ \varphi(x_1, \ldots, x_n, z) \downarrow] \qquad (*)$$

A volte le funzioni μ-ricorsive *totali* le chiameremo semplicemente *ricorsive*, soprattutto per ragioni storiche. Nota bene: *non* sono *solo* le funzioni ricorsive primitive! (in cui, per esempio manca la funzione di Ackermann che è ricorsiva ma non ricorsiva primitiva.)

Una funzione μ-ricorsiva è intuitivamente calcolabile? **Sì**: l'algoritmo "intuitivo" che la calcola è composto da un ciclo in cui si incrementa la variabile y (inizialmente posta a 0), si calcola la φ e si ripetono questi passi finché il risultato non è 0. I primi passi dell'esecuzione di questo algoritmo potrebbero essere dunque:

1. calcola $\phi(x_1, \ldots, x_n, 0)$; se il risultato è 0 allora $\psi(x_1, \ldots, x_n) = 0$;
2. altrimenti calcola $\varphi(x_1, \ldots, x_n, 1)$; se il risultato è 0 allora $\psi(x_1, \ldots, x_n) = 1$;
3. ...
   ⋮

Intuitivamente, potrei non finire mai o perché per ogni valore di y esiste un m_y tale che $\varphi(x_1, \ldots, x_n, y) = m_y \neq 0$, o perché per i primi k numeri naturali $\varphi(x_1, \ldots, x_n, z) = n_z \neq 0$ e $\varphi(x_1, \ldots, x_n, k) \uparrow$. Infatti nel primo caso continuiamo a calcolare la $\varphi(x_1, \ldots, x_n, y)$ per valori crescenti di y senza terminare mai, e nel secondo caso non ci arrestiamo mai nel calcolo di

$\varphi(x_1, \ldots, x_n, k)$: da qui la parzialità di ψ. Se dovessi scrivere un programma, userei un comando di tipo `while`.

Vediamo adesso un semplicissima definizione μ-ricorsiva, tramite la quale rappresentiamo l'archetipo delle funzioni parziali: la funzione ovunque indefinita, che è calcolabilissima!

Esempio 1.8.1. La seguente è una delle possibili derivazioni per la funzione ovunque indefinita $\psi_\uparrow(x)$:

$$\varphi = \lambda x, y.\ 3$$

$$\psi_\uparrow = \lambda x.\ (\mu y.\ \varphi(x, y) = 0)$$

Per verificare che quanto scritto sopra è davvero una derivazione basta controllare che la funzione (ricorsiva primitiva) φ è definita per tutti i suoi argomenti, ovvero che il calcolo di $\varphi(x, y)$ termina per ogni x; il che è banale. In questo caso è altrettanto facile vedere che per nessun x esiste un y per cui $\varphi(x, y) = 0$ e che quindi la funzione $\psi(x)$ è indefinita per ogni valore di x; attenzione però: nella maggior parte dei casi questo controllo è molto molto difficile, in un senso che sarà precisato formalmente tra poco.

Ricapitolando: si comincia con le ricorsive primitive, si applica l'operatore di minimizzazione μ e si ottiene una funzione μ-ricorsiva, che può essere ora usata nella prossima definizione. Tutto ciò a patto che la condizione (*) valga, altrimenti si può uscire dalla classe, cioè, se φ non termina per qualche valore z minore del minimo y su cui φ vale 0, allora la funzione ψ potrebbe non essere μ-ricorsiva: terminazione e non terminazione sono *importantissime*!

Si noti anche che

$$f(x) = \begin{cases} \mu y[y < g(x), h(x, y) = 0] & \text{se esiste tale } y \\ 0 & \text{altrimenti} \end{cases}$$

è ricorsiva primitiva se g e h lo sono. La ragione è che g impone un limite ai tentativi di ricercare il minimo y, e quindi o lo troviamo in meno di $g(x)$ applicazioni di h o diamo come risultato 0 dopo al più $g(x)$ applicazioni di h, che è un numero determinabile in tempo *finito* perché sia g che h sono totali in quanto ricorsive primitive. In altre parole anche la f sarebbe totale, e quindi avremmo definito un formalismo che definisce solo funzioni totali e quindi inadatto a rappresentare *tutte* le funzioni calcolabili.

Diamo ora alcune definizioni ausiliarie che ci saranno utili in seguito.

Definizione 1.8.2. Una relazione $I \subseteq \mathbb{N}^n$, $n \geq 1$ è *ricorsiva* (come sinonimo di totale) (o rispettivamente è *ricorsiva primitiva*, ha la proprietà P) se la sua funzione caratteristica χ_I è ricorsiva totale (è ricorsiva primitiva, ha la proprietà P). Un caso particolare e interessante si ha con gli *insiemi ricorsivi* $I \subseteq \mathbb{N}$, cioè quando $n = 1$.

In analogia a quanto fatto con le funzioni T-calcolabili, diciamo che una funzione è μ-*calcolabile* se la sua definizione è μ-ricorsiva.

Adesso abbiamo le funzioni T-calcolabili, quelle *WHILE*-calcolabili e quelle μ-calcolabili e il bello è che formano *esattamente la stessa classe di funzioni calcolabili*, ciò che è stato accuratamente dimostrato. Abbiamo già annunciato che molti altri formalismi sono stati proposti e che tutti questi (quando siano sufficientemente potenti in un senso che renderemo preciso tra poco) definiscono la *stessa classe di funzioni*; in altre parole sono *Turing equivalenti*. Pertanto possiamo, o meglio vogliamo stipulare come vera la

Tesi di Church-Turing: Le funzioni (*intuitivamente*) calcolabili sono tutte e sole le funzioni (parziali) T-calcolabili.

In realtà questa è un'ipotesi, ma è talmente forte che la prendiamo come tesi. In termini informatici, questo significa che non importa quale linguaggio di programmazione usiamo, né su quale macchina facciamo girare i nostri programmi, purché si abbia a disposizione memoria e tempo illimitati: ciò che possiamo calcolare *non* cambia — può forse cambiare *come* lo si calcola.

Chiaramente è dimostrabile solo l'equivalenza tra i formalismi esistenti, ed è certamente molto difficile immaginare una dimostrazione di equivalenza tra tutti i *possibili* formalismi, inclusi quelli ancora da inventare.

La tesi di Church-Turing postula che la nozione di calcolabilità "intuitiva" è *robusta*. Inoltre, la tesi cade se si rilascia anche una sola delle ipotesi fatte sulla natura degli algoritmi.

Bene, di qui in avanti parleremo solo di *funzioni calcolabili*, senza qualificare ulteriormente il formalismo usato per definirle. Quante sono? E ce ne sono di non calcolabili? Se sì, ne vedremo una interessante?

1.9 Alcuni risultati classici

Introdurremo brevemente alcuni risultati basilari della teoria della calcolabilità che ne illustrano l'essenza, caratterizzando la classe delle funzioni, ovvero dei problemi calcolabili, mediante alcuni teoremi di "chiusura". Privilegeremo una presentazione orientata ai fondamenti dell'informatica, a volte purtroppo senza la profondità e l'accuratezza che sarebbero necessari nel presentare una teoria in cui precisione e attenzione ai dettagli giocano un ruolo essenziale. Prima di enunciare questi risultati, insistiamo a ricordare che, grazie alla tesi di Church, possiamo chiamare *calcolabili* indifferentemente le funzioni esprimibili nel formalismo delle macchine di Turing o le funzioni μ-ricorsive o i programmi *WHILE* o ciò che volete voi, purché la loro definizione rispetti le cinque condizioni intuitive poste agli algoritmi che sono state espresse nel primo capitolo: le nostre ipotesi di lavoro.

Cominciamo con un semplice risultato sulla cardinalità[15] dell'insieme delle funzioni calcolabili, da cui segue che vi sono funzioni *non calcolabili*.

Teorema 1.9.1.

i) Le funzioni calcolabili sono $\#(\mathbb{N})$; inoltre anche le funzioni calcolabili totali sono $\#(\mathbb{N})$.

ii) Esistono funzioni non calcolabili.

Dimostrazione.

i) Costruisci $\#(\mathbb{N})$ MdT M_i che svuotano il nastro dall'input, ci scrivono la stringa $|^i$ e si arrestano (sono tutte le funzioni costanti). Che non siano più di $\#(\mathbb{N})$ segue dal fatto che le MdT si possono enumerare, come fatto intuire a pagina 26.

ii) Con una costruzione analoga a quella di Cantor (la classe dei sottoinsiemi di $\mathbb{N}$ non è numerabile) si vede che $\{f : \mathbb{N} \to \mathbb{N}\}$ ha cardinalità $2^{\#(\mathbb{N})}$.

□

Abbiamo già visto nel capitolo 1.6 che possiamo associare un indice alle macchine di Turing codificandole (veramente avevamo una funzione solamente iniettiva, ma sappiamo che Gödel ne ha definito una anche surgettiva); analogamente nel capitolo successivo abbiamo accennato a come enumerare le

[15] Dato un insieme A, indicheremo con $\#(A)$ la sua cardinalità, ovvero il numero dei suoi elementi.

funzioni ricorsive primitive e non è difficile immaginare una sua estensione alle funzioni μ-ricorsive. Oltre alla superficialità con cui sono stati presentati, i due modi hanno in comune il fatto che si basano *solamente* sui simboli usati nel definire gli algoritmi, il che è bene.

Infatti, sotto ipotesi molto ragionevoli, per i nostri scopi non c'è sostanziale differenza tra una enumerazione e un'altra, purché sia *effettiva*. Quindi siamo liberi di scegliere quella che più ci aggrada. Basti qui dire che una buona enumerazione deve essere una funzione biunivoca che dipende *solo* dalla sintassi con cui scriviamo gli algoritmi e *non* dal significato che attribuiamo loro. Di nuovo, la già ricordata enumerazione del capitolo 1.6 andrebbe bene se fosse surgettiva; invece una enumerazione che pretendesse, ad esempio, di elencare prima tutte le funzioni costanti e poi le altre non sarebbe effettiva. L'osservazione appena fatta ci consente anche di fissare una volta per tutte un elenco di MdT (programmi *WHILE*, funzioni μ-ricorsive, ...) e di indicare con M_i la MdT (il programma, la funzione μ-ricorsiva, ...) che vi appare in posizione i-ma, o meglio l'algoritmo i-mo. Ancor meglio, si può usare la seguente notazione, che finalmente evidenzia la differenza tra *funzione* e *algoritmo* che la calcola.

NOTAZIONE Data un'enumerazione effettiva, indicheremo con φ la funzione (parziale) che la macchina, o meglio l'algoritmo, M_i calcola e chiameremo i *indice* (non della funzione, bensì della macchina! quindi può darsi benissimo che per $i \neq j$ sia $\varphi_i = \varphi_j$, mentre sicuramente $M_i \neq M_j$).

Entriamo adesso nel vivo della presentazione dei teoremi più importanti di questa prima parte del corso, ribadendo ancora una volta il seguente fatto: i risultati che riportiamo nel seguito sono tutti *invarianti* rispetto all'enumerazione scelta.

Il primo teorema, che spesso è chiamato *padding lemma*, ci dice che ci sono infinite (numerabili) MdT, ovvero infiniti numerabili algoritmi che calcolano la stessa funzione, e che *alcuni* di essi si possono costruire "facilmente" da un algoritmo dato (ossia esprimibile con una funzione primitiva ricorsiva o un programma *FOR*).

Teorema 1.9.2 (Padding Lemma)**.** *Ogni funzione calcolabile φ_x ha $\#(\mathbb{N})$ indici. Inoltre $\forall x$ si può costruire, mediante una funzione ricorsiva primitiva, un insieme infinito A_x di indici tale che*

$$\forall y \in A_x.\ \varphi_y = \varphi_x$$

cioè $\varphi_y(n) = m$ sse $\varphi_x(n) = m$ (e ovviamente $\varphi_y(n) \uparrow$ sse $\varphi_x(n) \uparrow$).

Dimostrazione. Per ogni macchina M_x, se $Q = \{q_0, \ldots, q_k\}$, ottieni la prima macchina M_{x_1} con $x_1 \in A_x$ aggiungendo lo stato $q_{k+1} \notin Q$ e la quintupla $(q_{k+1}, \#, q_{k+1}, \#, -)$; ottieni la seconda M_{x_2} aggiungendo lo stato q_{k+2} e la quintupla $(q_{k+2}, \#, q_{k+2}, \#, -)$, ... □

Il prossimo teorema dice che tra tutti gli algoritmi che calcolano una data funzione ce n'è uno privilegiato, nel senso che ha una forma speciale. Di conseguenza, ogni funzione ha una rappresentazione privilegiata.

Teorema 1.9.3 (Forma normale)**.** *Esistono un predicato $T(i, x, y)$ e una funzione $U(y)$ calcolabili totali tali che $\forall i, x.\ \varphi_i(x) = U(\mu y.\ T(i, x, y))$. Inoltre, T e U sono primitive ricorsive.*

Dimostrazione. Definisci $T(i, x, y)$, detto comunemente *predicato di Kleene*, vero se e solamente se y è la codifica di una computazione terminante di M_i con dato iniziale x. Per calcolare T, dato i recupera M_i dalla lista e comincia a scandire i valori y. Decodifica ognuno di essi uno alla volta e, avendo come ingresso x, controlla se il risultato è una computazione terminante della forma $M_i(x) = c_0, c_1, \ldots, c_n$. Questa sequenza di passi termina sempre e quindi T è totale. Se y è la codifica di una computazione terminante di $M_i(x)$, allora $c_n = (h, \triangleright z\underline{\#})$ e definisci U in modo che $U(y) = z$ e i passi necessari a questo calcolo sono finiti e terminano tutti: quindi U è totale. L'intero procedimento seguito è effettivo, e quindi T e U sono calcolabili per la tesi di Church-Turing.

Inoltre U e T sono ricorsivi primitivi perché le codifiche che usiamo lo sono e perché lo sono i controlli effettuati. □

Si noti che la versione del teorema di forma normale sulle MdT deterministiche è tale per cui se esiste un y tale che $T(i, x, y)$ risulta vero, allora tale y è anche unico. Invece, se le MdT considerate fossero non-deterministiche (vedi Def. 2.3.1), ci potrebbero essere più computazioni che terminano nello stato h, per cui l'operatore di minimizzazione darebbe come risultato il minimo intero che codifica una di esse (si veda la discussione sulle regole di valutazione fatta nel capitolo 1.6).

Un'immediata conseguenza del teorema di forma normale è che ogni funzione calcolata da una MdT ammette una definizione μ-ricorsiva. In altre parole, vale il seguente corollario.

Corollario 1.9.1. *Le funzioni T-calcolabili sono μ-ricorsive.*

Lemma 1.9.1. *Le funzioni μ-calcolabili sono T-calcolabili.*

Ora è facile concludere l'equivalenza tra MdT e funzioni μ-ricorsive.

Teorema 1.9.4. *Una funzione è T-calcolabile se e solo se è μ-calcolabile.*

Il teorema di forma normale e quello d'equivalenza tra MdT e funzioni μ-ricorsive ha il seguente corollario interessante dal punto di vista informatico. La sua rilevanza nel nostro campo è legata al fatto che le funzioni primitive ricorsive si possono rappresentare con un programma nel linguaggio *FOR* e quelle μ-ricorsive con uno nel linguaggio *WHILE* (v. capitolo precedente); pertanto, si deduce che *ogni programma* può essere scritto (in forma normale) usando un comando di tipo `while` e due di tipo `for` (ciò è particolarmente rilevante quando il programma in questione sia l'interprete di un linguaggio, come vedremo più avanti).

Corollario 1.9.2. *Ogni funzione calcolabile parziale può essere ottenuta da due funzioni primitive ricorsive e una sola applicazione dell'operatore μ.*

Adesso arriva un teorema molto importante. Dice che un formalismo universale, cioè uno che esprima *tutte* le funzioni calcolabili, è così potente da riuscire a esprimere *l'interprete dei propri programmi*. Vedremo più avanti che questa capacità può essere usata "alla rovescia", cioè per mostrare che un formalismo è universale. Vediamo il semplice caso con funzioni a una variabile e la sua dimostrazione; l'estensione al caso generale per funzioni a n variabili è immediata.

Teorema 1.9.5 (Enumerazione)**.** *Esiste una funzione calcolabile parziale $\varphi_z(i,x)$ tale che $\varphi_z(i,x) = \varphi_i(x)$.*

(Si noti l'ordine dei quantificatori: $\exists z$ tale che $\forall i, x$ si ha $\varphi_z(i,x) = \varphi_i(x)$.)

Dimostrazione. Poiché la funzione $U(\mu y.\ T(i,x,y))$ usata nel teorema di forma normale è definita per composizione e μ-ricorsione a partire da funzioni (meglio da una funzione e un predicato) primitive ricorsive, essa stessa è una funzione calcolabile in due argomenti i e x. Avrà quindi un indice che chiamiamo z, cioè sia $\varphi_z(i,x) = U(\mu y.\ T(i,x,y))$. Applichiamo allora il teorema di forma normale, da cui $U(\mu y.\ T(i,x,y)) = \varphi_i(x)$. Per ottenere la tesi basta la transitività dell'uguaglianza.

Più intuitivamente e informalmente: M_z recupera la descrizione di M_i e la applica a x. □

Il teorema di enumerazione garantisce che esiste la MdT Universale, M_z, che ha in ingresso la descrizione di una MdT M_i, il dato x e "si comporta come M_i". È esattamente quanto avviene ogni giorno con i nostri programmi: li diamo in pasto a una macchina realizzata in parte H/W, in parte S/W (si tratta di una macchina reale, e quindi solo "quasi-universale", perché, tra l'altro, ha memoria limitata). La macchina in questione esegue i nostri programmi, ovvero si comporta esattamente come dettato dai suoi dati, i programmi in ingresso: finché l'istruzione corrente non è STOP prende i

suoi argomenti, esegue l'istruzione, ne memorizza il risultato e aggiorna il puntatore all'istruzione corrente.

Intuitivamente, il teorema di enumerazione ci "libera" dalla necessità di avere un esecutore umano delle MdT, così come previsto da Alan Turing: *esiste una macchina che esegue gli algoritmi.*

Teorema 1.9.6 (Parametro, s-1-1)**.** *Esiste una funzione calcolabile totale (iniettiva) s_1^1 con due argomenti, tale che $\forall i, x$*

$$\varphi_{s_1^1(i,x)} = \lambda y.\ \varphi_i(x, y)$$

Intuitivamente, la macchina $M_{s(i,x)}$, o più liberamente l'algoritmo o il programma $P_{s(i,x)}$ (omettiamo d'ora in poi l'indice e l'apice 1, per leggibilità) opera su y soltanto, mentre P_i opera su x e y. Quindi x è un *parametro* di P_i. Ad esempio, sia $\varphi_i(x, y)$ la funzione $x \times f(y)$ (con f qualunque); allora, a partire da i e da 2, mediante la s trovo in *modo effettivo* l'indice del programma che calcola la funzione $2 \times f(y)$, cioè determino $\varphi_{s(i,2)}$.

Il teorema *s-m-n* è importante in informatica perché è la base per la tecnica di "valutazione parziale" secondo la quale si specializza via via un programma generale per ottenerne versioni più efficienti in casi particolari (compilatori/interpreti per architetture parametriche, ecc.). Questo teorema è inoltre utilissimo nella teoria della calcolabilità sia perché ci offre uno strumento potentissimo da usare nelle dimostrazioni, sia per le ragioni espresse dal teorema di espressività, riportato più avanti. Vediamo la dimostrazione della sua versione *s-1-1*, cioè con $m, n = 1$, e poi l'enunciato generale.

Dimostrazione intuitiva del teorema s-1-1. Per calcolare $\varphi_{s_1^1(i,x)}(y)$ si prenda M_i decodificando i e si predisponga lo stato iniziale, cioè $M_i(x, y)$, dove x è fissato in anticipo. (Se vi fosse più familiare, prendete il programma P_i che avrà un'istruzione per leggere il valore k del parametro x: allora rimuovetela e inserite al suo posto l'assegnamento $x := k$.) Quella delineata (in entrambi i casi) è una procedura effettiva, cioè algoritmica, che termina sempre, quindi per la tesi di Church-Turing esiste una funzione calcolabile totale $s = s_1^1$. Se tale funzione non fosse iniettiva, allora si costruisca s' con $\varphi_{s(i,x)} = \varphi_{s'(i,x)}$ in modo tale che $s'(i, x)$ generi indici (che esistono perché gli indici delle MdT che calcolano $s(i, x)$ sono $\#(\mathbb{N})$) in modo strettamente crescente, cioè tali che $s'(i_0, x_0) > s'(i_1, x_1)$ se la codifica della coppia (i_0, x_0) è maggiore di quella di (i_1, x_1). A questo punto basta notare che una funzione totale strettamente crescente è iniettiva. □

Teorema 1.9.7 (Parametro, s-m-n)**.** $\forall m, n > 0$ *esiste una funzione calcolabile totale (iniettiva) s_n^m con $m + 1$ argomenti tale che $\forall i, x_1, \ldots, x_m$*

$$\varphi^{(n)}_{s_n^m(i,x_1,\ldots x_m)} = \lambda y_1, \ldots y_n.\ \varphi_i^{(m+n)}(x_1, \ldots, x_m, y_1, \ldots, y_n)$$

Si noti come il teorema del parametro e quello di enumerazione siano in un certo senso l'inverso l'uno dell'altro. Infatti uno "abbassa" un argomento nella posizione di indice, mentre l'altro "innalza" un indice nella posizione di argomento.

L'importanza dei teoremi del parametro e di enumerazione può essere compresa ancora meglio considerando il seguente teorema che non dimostreremo.

Teorema 1.9.8 (Espressività)**.** *Un formalismo è Turing-equivalente (calcola tutte e sole le funzioni T-calcolabili, è universale) se e solamente se*

- *ha un algoritmo universale (cioè vale il teorema di enumerazione),*
- *vale il teorema del parametro.*

Grazie al teorema *s-m-n* si dimostra un teorema molto elegante, che ha un ruolo fondamentale, sia in informatica che nella pura teoria della calcolabilità.

Teorema 1.9.9 (Ricorsione, Kleene II)**.** $\forall f$ *funzione calcolabile totale* $\exists n$ *tale che* $\varphi_n = \varphi_{f(n)}$*.*

(Un tale indice viene detto ***punto fisso*** *di* f*.)*

Prima della dimostrazione, diamo un po' di intuizione: la funzione f "trasforma" programmi in programmi, come fanno i compilatori. Infatti f trasforma indici: dato n, ovvero il programma P_n lo trasforma in $P_{f(n)}$. Quando consideriamo il punto fisso, la trasformazione operata da f *non cambia* la funzione calcolata, ovvero trasforma un programma P_n nel programma $P_{f(n)}$ con la *stessa* semantica. Si noti che l'accezione "punto fisso" usata qui è diversa da quella solita in cui il punto fisso x di un funzione g è tale che $g(x) = x$: qui il punto fisso riguarda la *funzione*, e non l'indice delle macchine che la calcolano.

Questo teorema fornisce in un certo senso la "base" della semantica denotazionale; garantisce la realizzabilità di macchine che eseguono programmi ricorsivi o delle funzioni di crittografia o di molte altre diavolerie infernatiche. Nella dimostrazione del teorema si fa uso del fatto che le funzioni calcolabili sono chiuse rispetto alla trasformazioni di indici introdotte dal teorema del parametro.

Dimostrazione. Definiamo la seguente funzione calcolabile "diagonale"

$$\psi(u, z) = \begin{cases} \varphi_{\varphi_u(u)}(z) & \text{se } \varphi_u(u) \downarrow \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

Poiché ψ è calcolabile, per Church-Turing avrà un indice $\varphi_i(u,z) = \psi(u,z)$. A questo punto per il teorema del parametro si ha $\varphi_i(u,z) = \varphi_{s(i,u)}(z)$, ma $s(i,u)$ dipende *solo* da u, quindi ponendo $d(u) = \lambda u.\ s(i,u)$ si ottiene che

$$\psi(u,z) = \varphi_i(u,z) = \varphi_{s(i,u)}(z) = \varphi_{d(u)}(z) = \begin{cases} \varphi_{\varphi_u(u)}(z) & \text{se } \varphi_u(u)\downarrow \\ \text{indefinita} & \text{altrimenti} \end{cases} \tag{1}$$

Per il teorema *s-m-n*, $d(u)$ è totale e iniettiva (e *non* dipende da f). Data f, $f \circ d$ è calcolabile e sia v proprio un indice tale che

$$\varphi_v(x) = f(d(x)) \tag{2}$$

Tale funzione è totale (perché sia d che f lo sono), e quindi $\varphi_v(v)\downarrow$. Pertanto, in accordo con la definizione (1) abbiamo $\varphi_{d(v)} = \varphi_{\varphi_v(v)}$. Calcoliamo adesso $d(v)$ e supponiamo che il risultato sia n, cioè poniamo

$$n = d(v) \tag{3}$$

Dimostriamo che n è un punto fisso di f. Infatti

$$\varphi_n \overset{(3)}{=} \varphi_{d(v)} \overset{(1)}{=} \varphi_{\varphi_v(v)} \overset{(2)}{=} \varphi_{f(d(v))} \overset{(3)}{=} \varphi_{f(n)}$$

Si noti come nell'eguaglianza più a sinistra si sfrutti l'iniettività della funzione d garantitaci dal teorema del parametro.

□

Ci sono due fatti interessanti che sono correlati con il teorema di ricorsione.

Proprietà 1.9.1. *Nelle ipotesi del teorema di ricorsione,*

- *il punto fisso è calcolabile mediante una funzione totale (iniettiva) g e a partire da(ll'indice di) f;*
- *ci sono $\#(\mathbb{N})$ punti fissi di f.*

C'è un altro modo per dimostrare il teorema di ricorsione, o meglio per specificare come deve essere implementata la ricorsione nei linguaggi di programmazione. Supponete di avere una procedura ricorsiva P il cui corpo sia C, all'interno del quale ovviamente appare la chiamata a P stessa. La tecnica usata per definire la semantica operazionale consiste nel memorizzare in un componente, di solito chiamato *ambiente* e rappresentato da una funzione ρ, l'associazione tra P e C, cioè $\rho(P) = C$. Al momento della chiamata si cerca nell'ambiente il significato di P, che appunto è C, e si trasferisce il controllo all'inizio di C, dopo aver ovviamente legato i parametri formali con

quelli attuali. Il significato di P viene mantenuto nell'ambiente, cosicché alla successiva chiamata si possa recuperare $\rho(P) = C$.

Questa tecnica a volte viene chiamata *copy rule*, perché in effetti si copia il corpo C della procedura P al posto di P stesso, tante volte quanto è necessario. Si noti che non si tratta di una macro-espansione, né si potrebbe macro-espandere per sempre la chiamata a P, perché non è noto a priori quante volte si debba chiamare la P stessa. Per rendersene conto, si consideri di nuovo la funzione fattoriale. Il suo corpo viene "usato" $n + 1$ volte se n è l'argomento: di fatto si approssima la definizione della funzione dove la macro-espansione è ripetuta per sempre, avendo definito il suo comportamento sull'intervallo $[0 \dots n]$, mentre per valori maggiori di n il risultato è lasciato indefinito.

1.10 Problemi Insolubili e Riducibilità

In questo capitolo studieremo i problemi di appartenenza o di non appartenenza di un elemento ad un dato insieme, affrontando così il modo di risolvere problemi alternativo a quello considerato fino ad ora, in cui l'oggetto del discorso era calcolare funzioni. Ovviamente queste due visioni di un problema matematico sono strettamente correlate; in seguito vedremo *esattamente* come lo sono.

Ricordiamo la definizione di insieme ricorsivo.

Un insieme I è *ricorsivo* (ovvero *decidibile*) se e solo se la sua funzione caratteristica

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

è calcolabile totale.

Vediamo quali insiemi vengono fuori se non poniamo alcun limite al numero dei passi consentiti a una macchina. Significa che andremo a vedere se una MdT termina (su un particolare dato) e se c'è un algoritmo per determinare tale proprietà – ovviamente no! Il gioco si fa definendo una classe di insiemi più ampia di quella degli insiemi ricorsivi.

Definizione 1.10.1. Diciamo che un insieme I è *ricorsivamente enumerabile*

$$I \text{ è } \textit{ricorsivamente enumerabile sse } \exists i.\ I = dom(\varphi_i)$$

Un insieme ricorsivamente enumerabile, detto anche *semi-decidibile* è quindi il dominio di una funzione calcolabile (il più delle volte *parziale*, infatti se fosse totale $I = \mathbb{N}$), che è anche detta *semi-caratteristica* di I.

Ci sono ovvie relazioni tra gli insiemi ricorsivi (decidibili) e quelli ricorsivamente enumerabili (semi-decidibili).

Proprietà 1.10.1.

i) Se I è ricorsivo allora I è ricorsivamente enumerabile.

ii) $I, \bar{I}$ sono ricorsivamente enumerabili se e solo se I (e $\bar{I}$) sono ricorsivi.

Dimostrazione. Il caso (i) è ovvio: la φ_i cercata restituisce 1 su x se $\chi_I(x) = 1$, altrimenti diverge.

(ii) Il caso precedente basta per vedere la parte "*se*". Consideriamo allora la parte "*solo se*": siano φ_i e $\varphi_{\bar{i}}$ le funzioni i cui domini sono rispettivamente I e $\bar{I}$. Adesso si ripeta il seguente ciclo: esegui un passo nel calcolo di $\varphi_i(x)$; se $\varphi_i(x) \downarrow$ allora $x \in I$ e poni $\chi_I(x) = 1$; altrimenti esegui un passo nel calcolo di $\varphi_{\bar{i}}(x)$; se $\varphi_{\bar{i}}(x) \downarrow$ allora $x \notin I$ e poni $\chi_I(x) = 0$. □

In realtà uno vorrebbe poter elencare (enumerare, generare) gli elementi di un insieme mediante una funzione calcolabile. Ecco un teorema che ci permette di fare ciò.

Teorema 1.10.1. *I è ricorsivamente enumerabile se e solamente se è vuoto oppure è l'immagine di una funzione calcolabile totale.*

Dimostrazione. Il primo caso si dimostra banalmente: l'insieme vuoto è il dominio di una funzione ovunque indefinita; d'altra parte, la funzione ovunque indefinita ha dominio vuoto.

Consideriamo adesso il caso in cui $I = dom(\varphi_i)$ sia non vuoto: consiste nella costruzione di una funzione totale calcolabile f tale che $I = imm(f)$ a partire da φ_i. Innanzitutto, si cerca un elemento di I mediante un procedimento a coda di colomba (Figura 1.1), in cui l'indice di riga m rappresenta il numero dei passi del calcolo di φ_i e l'indice di colonna n il suo argomento.

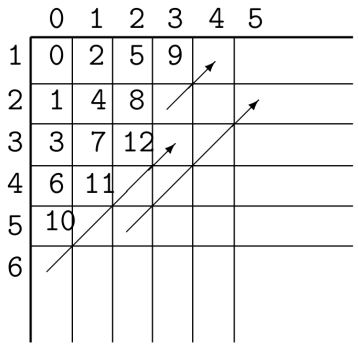


Figura 1.1: Nella tabella a "coda di colomba" si interpreti l'indice di riga come il numero di passi eseguiti da M_i sul valore dell'indice di colonna.

Più precisamente, si eseguono m passi nel calcolo di $\varphi_i(n)$, finché per qualche valore di m e dell'argomento, sia $\bar{n}$, il calcolo si arresta; ovvero $\varphi_i(\bar{n}) \downarrow$ in m passi e quindi $\bar{n} \in I$.

A questo punto, rappresentando con $\langle n, m \rangle$ il valore della codifica della coppia (n, m), si inizia un secondo procedimento a coda di colomba eseguendo $\varphi_i(n)$ per m passi: se tale calcolo si arresta, allora si pone $f(\langle n, m \rangle) = n$ (ovviamente $n \in dom(\varphi_i) = I$), altrimenti si pone $f(\langle n, m \rangle) = \bar{n}$ (per quanto detto prima $\bar{n} \in I$); si itera il procedimento incrementando la codifica $\langle n, m \rangle$, ovvero considerando $\langle n, m \rangle + 1$. Si generano così tutti gli elementi di I.

□

Adesso vediamo un insieme veramente speciale e paradigmatico:

$$K = \{x \mid \varphi_x(x) \downarrow\}$$

Proprietà 1.10.2. *K è ricorsivamente enumerabile.*

Dimostrazione. K è il dominio di

$$\psi(x) = \begin{cases} x & \text{se } \varphi_x(x)\downarrow \\ \text{indefinita} & \text{altrimenti} \end{cases}$$

che è calcolabile: prendi la MdT M_x e applicala a x; se e quando essa si arresta, restituisci x. □

Facciamo adesso vedere che la ψ della dimostrazione precedente genera K, ma *non* è una funzione calcolabile totale, e che nessuna funzione che decida K lo è.

Proprietà 1.10.3. *K non è ricorsivo.*

Dimostrazione. Sia χ_K la funzione caratteristica di K e per assurdo sia totale e calcolabile. Ma allora anche la seguente funzione

$$f(x) = \begin{cases} \varphi_x(x) + 1 & \text{se } x \in K \\ 0 & \text{se } x \notin K \end{cases}$$

sarebbe calcolabile totale. In questo modo otteniamo una contraddizione perché $\forall x.\ f(x) \neq \varphi_x(x)$; quindi non troviamo alcun indice per f che di conseguenza non è calcolabile. □

La proprietà appena vista significa che *non esiste un algoritmo per decidere se $x \in K$* o no. Quindi questo problema è *insolubile*, anche se ovviamente è semi-decidibile. Inoltre *$\overline{K}$ non è ricorsivamente enumerabile* quindi esistono problemi ancora più difficili di K! Infatti, se $\overline{K}$ fosse ricorsivamente enumerabile, sia K che $\overline{K}$ sarebbero ricorsivi, per la proprietà 1.10.1(ii), in quanto K è ricorsivamente enumerabile per la proprietà 1.10.2. Abbiamo così stabilito un piccolo frammento di gerarchia:

$$R \subsetneq RE \subsetneq nonRE$$

dove R è la classe degli insiemi ricorsivi, RE quella degli insiemi ricorsivamente enumerabili e $nonRE$ quella degli insiemi non ricorsivamente enumerabili (la scelta del nome "non ricorsivamente enumerabile" non è molto felice, perché un insieme (ricorsivo è anche) ricorsivamente enumerabile è anche non ricorsivamente enumerabile: in questo, come in mille altri casi, una formula è assai più precisa di una frase nel linguaggio comune! con maggior esattezza si dovrebbe dire che $\overline{K} \in$ *co-RE*, la classe dei problemi i cui complementi sono ricorsivamente enumerabili, ma non ricorsivi).

Finalmente siamo arrivati al problema, che di solito si chiama *problema della fermata*, e che confuta l'affermazione di Hilbert che *tutti* i problemi matematici hanno una caratterizzazione esatta. A costo di essere noioso, vale la pena di ripetere che, come tutti i risultati sulla calcolabilità che discuteremo, anche questo è *indipendente* sia dal formalismo impiegato per scrivere gli algoritmi, ovvero per esprimere le funzioni, sia dall'enumerazione effettiva scelta. In altre parole, *tutti* i formalismi che siano T-equivalenti soffrono del problema della fermata, il che potrebbe, con le dovute ipotesi, essere aggiunto come terzo punto al teorema di espressività.

Potrebbe comunque rimanere il dubbio che K sia un problema artificiale, che non ha alcuna rilevanza pratica; si tratterebbe allora di un risultato negativo, ma di scarso impatto sulla realizzazione di elaboratori, programmi e degli altri aggeggi infernatici che ci stanno a cuore. Infatti a chi mai verrebbe in mente di applicare un programma a sé stesso? A chi realizza compilatori, in particolare il cosiddetto "bootstrapping"! Il problema può essere raccontato così: supponete di aver scritto un compilatore, o meglio un *cross-compiler*, in un certo linguaggio L, che traduce programmi scritti in L in programmi scritti in un altro linguaggio A; rappresentiamo tale compilatore come $C_L^{L\to A}$ — questo ovviamente implica che abbiate già un compilatore per L che gira su una qualche macchina, magari molto potente, ma sulla quale i programmi A non girano. Supponete adesso di voler scrivere il compilatore $C_A^{L\to A}$, cioè un compilatore ancora da L a A, ma stavolta scritto nel liguaggio A, che potrebbe essere l'assembler di una macchina che non sostiene L. Basta allora prendere $C_L^{L\to A}$, applicarlo a sé stesso per ottenere ciò che si desidera:

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

Tuttavia questo può ancora sembrare un caso estremo, e consideriamo allora il seguente problema, la cui soluzione positiva ci aiuterebbe enormemente nel nostro lavoro. Possiamo scrivere un programma P che, dato un altro programma Q (individuato dal suo indice y) e un argomento x, ci assicura che la computazione di Q su x terminerà o meno? Questo è un problema certamente più reale di K, prescinde dalla formalizzazione di algoritmo che stiamo esaminando e ha dunque interesse in sé. Infatti, piacerebbe a ciascuno di noi avere a disposizione il programma guardia P, in modo da non lanciare nemmeno l'esecuzione di $Q(x)$ quando questi non termina. Vista la sua importanza, questo problema si merita un nome:

Problema della fermata: dati x, y. $\varphi_y(x)\downarrow$? cioè $P_y(x)$ si ferma?

Il problema della fermata si formalizza e si studia nei termini usati in questo capitolo introducendo un altro insieme che gode di grande popolarità e

caratterizzandone la natura. Sia

$$K_0 = \{(x, y) \mid \varphi_y(x) \downarrow\}, \text{ ovvero } K_0 = \{(x, y) \mid \exists z.\ T(x, y, z)\}$$

dove T è il predicato di Kleene che abbiamo introdotto nel teorema di forma normale (1.9.3).

Corollario 1.10.1. *K_0 non è ricorsivo.*

Dimostrazione. Si ha che $x \in K$ se e solamente se $(x, x) \in K_0$, quindi se K_0 fosse ricorsivo lo sarebbe anche K. □

Abbiamo appena visto che il problema della fermata, formalizzato da K_0, è strettamente collegato al problema per decidere l'appartenenza o meno di un elemento a K, tanto da risultarne "equivalente". La tecnica di dimostrazione usata per collegare K e K_0 si basa sul concetto di RIDUCIBILITÀ, che è una nozione fondamentale, e non solo nella teoria della calcolabilità e della complessità. Il punto cruciale nella sua definizione è che la *funzione di riduzione* deve essere "*semplice*" (in questa parte del corso useremo funzioni calcolabili totali, nella terza parte funzioni polinomiali in tempo o indifferentemente logaritmiche in spazio).

Visto che la nozione di riducibilità è importantissima, faremo di seguito una digressione per introdurla e caratterizzarla in una forma abbastanza generale.

RIDUCIBILITÀ

Una riduzione è una particolare funzione f che trasforma un problema (ovvero un insieme o una classe) A in un altro problema B, in modo da mantenerne inalterata la caratteristica principale.

Definizione 1.10.2. A si *riduce* a B secondo la *riduzione* f, in simboli $A \leqslant_f B$, tutte e sole le volte che

$$a \in A \Leftrightarrow f(a) \in B, \text{ ovvero } f(A) \subseteq B \text{ e } f(\bar{A}) \subseteq \overline{B}$$

La seguente proprietà è di immediata dimostrazione.

Proprietà 1.10.4. *$A \leqslant_f B$ se e solamente se $\bar{A} \leqslant_f \overline{B}$.*

Dimostrazione. Si ha che $x \in \bar{A}$ se e solamente se $x \notin A$ se e solamente se $f(x) \notin B$ se e solamente se $f(x) \in \overline{B}$. □

Più in generale, si definisce una relazione di riduzioni ($\leqslant_F$) dove F è una particolare classe di funzioni. Allora scriveremo

$$A \leqslant_F B \Leftrightarrow \exists f \in F.\ A \leqslant_f B$$

Ci interessano solo quelle riduzioni $\leqslant_F$ che danno origine a *classi* di problemi in qualche modo "omogenei". Vediamo adesso in maggior dettaglio cosa si debba intendere per omogeneo e come si possano definire quelle relazioni di riduzione che riteniamo interessanti.

Definizione 1.10.3. Siano $\mathcal{D}$ e $\mathcal{E}$ due classi di problemi con $\mathcal{D} \subseteq \mathcal{E}$ (e anche $\mathcal{E} \subseteq \mathcal{H}$, che però non menzioneremo ulteriormente). Una relazione di riduzione $\leqslant_F$ *classifica* $\mathcal{D}$ ed $\mathcal{E}$ se e solo se per ogni problema A, B, C

i) $A \leqslant_F A$ (*Riflessiva*)

ii) $A \leqslant_F B, B \leqslant_F C$ implica $A \leqslant_F C$ (*Transitiva*)

iii) $A \leqslant_F B, B \in \mathcal{D}$ implica $A \in \mathcal{D}$ ($\mathcal{D}$ *ideale = chiuso all'ingiù per riduzione*)

iv) $A \leqslant_F B, B \in \mathcal{E}$ implica $A \in \mathcal{E}$ ($\mathcal{E}$ *ideale = chiuso all'ingiù per riduzione*)

Vediamo adesso una caratterizzazione differente, ma del tutto equivalente, delle riduzioni che classificano coppie di classi, l'una inclusa nell'altra.

Lemma 1.10.1. *Una relazione di riduzione $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, tali che $\mathcal{D} \subseteq \mathcal{E}$, se e solo se*

i) $id \in F$ *(F ha identità)*

ii) $f, g \in F \Rightarrow f \circ g \in F$ *(F chiusa per composizione)*

iii) $f \in F,\ B \in \mathcal{D} \Rightarrow \{x \mid f(x) \in B\} \in \mathcal{D}$

iv) $f \in F,\ B \in \mathcal{E} \Rightarrow \{x \mid f(x) \in B\} \in \mathcal{E}$

Dimostrazione. Vedi punto per punto la definizione di classificazione. □

Attraverso il concetto di relazione di riduzione che classifica due classi di problemi si possono definire le seguenti nozioni molto importanti. (Si noti che la relazione $\leqslant_F$ è un pre-ordine parziale[16], il che giustifica l'uso del termine "ideale" fatto sopra.)

[16]Cioè un ordinamento parziale riflessivo e transitivo, ma non anti-simmetrico.

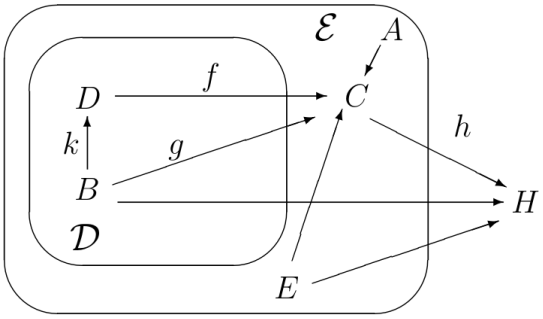
Definizione 1.10.4. Se $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, $\forall A, B, H$ problemi

i) $A \equiv B$ se $A \leqslant_F B$ e $B \leqslant_F A$ (si dice anche che $\{B \mid A \equiv_F B\}$ è il *grado* di A, o anche che A è equivalente a B rispetto a $\leqslant_F$)[17].

ii) H è $\leqslant_F$-*arduo* per $\mathcal{E}$ se $\forall A \in \mathcal{E}.\ A \leqslant_F H$ [18].

iii) H è $\leqslant_F$-*completo* per $\mathcal{E}$ se H è $\leqslant_F$-arduo per $\mathcal{E}$ e $H \in \mathcal{E}$.

Diremo semplicemente $\mathcal{E}$-arduo o $\mathcal{E}$-completo quando la classe di riduzioni F è fissata; talvolta ometteremo anche $\mathcal{E}$, se chiaro nel contesto.



Il disegno di sopra esemplifica quanto scritto. Il problema C è completo per $\mathcal{E}$ e a esso si riducono sia B e D di $\mathcal{D}$, sia A e E di $\mathcal{E}$; non tutte le riduzioni sono state disegnate (come frecce), tuttavia si noti che $g = f \circ k$ e allo stesso modo si compongono tutte le frecce disegnate o meno; infine H è un problema arduo per $\mathcal{E}$, ma non completo: tutti i problemi di $\mathcal{E}$ si riducono ad H, ma $H \notin \mathcal{E}$.

Se un problema è completo per un classe $\mathcal{E}$ e appartiene ad un sottoclasse $\mathcal{D}$, allora le due classi coincidono.

Proprietà 1.10.5. *Se $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$, $\mathcal{D} \subseteq \mathcal{E}$ e C è completo per $\mathcal{E}$, allora $C \in \mathcal{D}$ se e solamente se $\mathcal{D} = \mathcal{E}$.*

Dimostrazione. (se) ovvia.
(solo se) Sia $C \in \mathcal{D}$ e $A \in \mathcal{E}$. Per completezza $A \leqslant_F C$ e $A \in \mathcal{D}$ per la condizione (iii) di $\leqslant_F$ che classifica $\mathcal{D}$ ed $\mathcal{E}$. Quindi $\mathcal{E} \subseteq \mathcal{D}$ e la tesi. □

[17] Se consideriamo, per esempio $\mathcal{D}_{\equiv}$ allora $\leqslant_F$ diventa un ordinamento parziale.
[18] Potrebbe essere $H \in \mathcal{H} \backslash \mathcal{E}$.

Inoltre è facile capire quali siano gli elementi del grado di A, problema $\leqslant_F$-completo per $\mathcal{E}$. Per aiutare l'intuizione guardiamo nuovamente il disegno fatto sopra: se vi fosse una riduzione anche tra C e A, allora componendo le frecce si otterrebbe una riduzione tra ogni elemento di $\mathcal{E}$ e A, cioè anche A sarebbe completo per $\mathcal{E}$.

Proprietà 1.10.6. *Se A è completo per $\mathcal{E}$, $A \leqslant_F B$ e $B \in \mathcal{E}$, allora B è completo per $\mathcal{E}$.*

Dimostrazione. $\forall D \in \mathcal{E}$, $D \leqslant_F A$ per complettezza, ma $\leqslant_F$-classifica $\mathcal{D}$ ed $\mathcal{E}$ e allora $D \leqslant_F A$ e $A \leqslant_F B$ implicano $D \leqslant_F B$ e quindi B è arduo e, poiché appartiene a $\mathcal{E}$, è completo. □

Un problema completo per $\mathcal{E}$ gioca un ruolo rilevantissimo, in quanto "rappresenta la difficoltà" massima dei problemi di $\mathcal{E}$. Infatti, è facile vedere che il grado di un problema A completo per $\mathcal{E}$ è il grado massimo di $\mathcal{E}$ in $\leqslant_F$. Inoltre, valgono le seguenti affermazioni (anche per problemi non completi, a dire il vero):

- se $B \leqslant_F A$ allora B ha *al più* il (o meglio al più appartiene al) grado di A, cioè è più facile o altrettanto difficile di A;
- se $A \leqslant_F B$ allora B ha *almeno* il grado di A, cioè è di difficoltà maggiore o uguale a quella di A;

FINE DIGRESSIONE

Rifrasiamo ora le definizioni 1.10.2 e 1.10.3 per ottenere il concetto di riducibilità che useremo in questa parte del corso; per farlo diamo un nome alla classe delle funzioni calcolabili totali:

$$rec = \{\varphi_x \mid \forall y \in \mathbb{N}.\ \varphi_x(y) \downarrow\}$$

Definizione 1.10.5. A è riducibile a B ($A \leqslant_{rec} B$) se e solamente se esiste una funzione calcolabile totale $f : \mathbb{N} \to \mathbb{N}$ tale che $x \in A$ se e solamente se $f(x) \in B$.

Vediamo ora che queste relazioni di riduzione *conservano la ricorsività e la ricorsiva enumerabilità*. Come già fatto quando abbiamo introdotto il piccolo frammento di gerarchia, d'ora in avanti indichiamo con R e RE le classi di insiemi rispettivamente ricorsivi e ricorsivamente enumerabili. Allora, possiamo dimostrare quanto segue.

Teorema 1.10.2. *La relazione di riduzione $\leqslant_{rec}$ classifica R ed RE.*

Dimostrazione. Sappiamo già che $R \subseteq RE$ grazie alla proprietà 1.10.1(i). Possiamo allora usare il lemma 1.10.1 per dimostrare la tesi. Facciamo allora vedere che tutte le ipotesi del lemma sono soddisfatte.

i) Facile, dalla definizione di μ-ricorsiva.

ii) Ovvio perché la composizione conserva la totalità.

iii) La funzione caratteristica di $\{x \mid f(x) \in B\}$ è $\chi_B \circ f$, che è calcolabile totale perché f e χ_B sono entrambe calcolabili totali.

iv) Analoga al punto precedente, con la semi-caratteristica di B.

□

Nei teoremi e negli esercizi che seguono useremo quasi sempre funzioni di riduzione inietttive, di solito ottenute applicando il teorema del parametro (teorema 1.9.7); più in generale si potrebbero definire relazioni di riduzioni $\leqslant^m_{rec}$, in cui le funzioni (calcolabili totali) di riduzione usate non sono iniettive (cioè sono molti-a-uno nella terminologia nord-americana).

Il fatto che $\leqslant_{rec}$ classifichi R ed RE può essere intuitivamente visto come la capacità che le riduzioni con funzioni calcolabili totali hanno di *separare* i problemi ricorsivi da quelli ricorsivamente enumerabili. Ciò viene fatto giocando sul tempo necessario a decidere un problema: se questo è ricorsivo avremo la risposta in tempo *finito*, altrimenti il tempo necessario è *infinito*. Inoltre, ci basta trovare un problema che sia $\leqslant_{rec}$-completo per R per poter vedere quali problemi sono decidibili e quali no; ancora più interessante è trovare un problema che sia $\leqslant_{rec}$-completo per RE: sapremmo allora quali problemi sono (al più) semi-decidibili e quali nemmeno semi-decidibili. Infatti basta ridurre il problema da studiare a quello completo e sapremo che è ricorsivamente enumerabile oppure ridurre il problema completo a quello da studiare e sapremo che quest'ultimo, ben che ci vada, è ricorsivamente enumerabile. Infatti, come notato nella digressione:

- Se $A \leqslant_{rec} B$ e B è ricorsivamente enumerabile ($B \in RE$), allora A è ricorsivamente enumerabile (e forse anche ricorsivo).
- Se $A \leqslant_{rec} B$ e A non è ricorsivamente enumerabile ($A \notin RE$), allora B non è ricorsivamente enumerabile (e men che meno ricorsivo).

Inoltre, se A è ricorsivo il fatto che si riduca a B non ci consente di dedurre alcunché sulla natura di B, il quale potrebbe essere ricorsivo o ricorsivamente enumerabile o nemmeno ricorsivamente enumerabile; analogalmente nel caso di A ricorsivamente enumerabile.

Prima di vedere il nostro (primo) problema completo per RE, notiamo che né K si riduce con funzioni calcolabili totali a $\overline{K}$, né il viceversa; cioè vediamo che un insieme ricorsivamente enumerabile può essere inconfrontabile con uno non ricorsivamente enumerabile, usando $\leqslant_{rec}$ (si ricordi che la definizione di K a pagina 44). (Questo può apparire bizzarro, perché se scopro che $x \in K$ so anche che $x \notin \overline{K}$ e infatti ci sono riduzioni un po' più astute che permettono di confrontare anche K e $\overline{K}$.) Dobbiamo quindi mostrare che

$$\overline{K} \not\leqslant_{rec} K \qquad \text{e} \qquad K \not\leqslant_{rec} \overline{K}$$

La prima disuguaglianza deve essere vera, perché altrimenti, usando la proprietà 1.10.1(ii), $\overline{K}$ sarebbe ricorsivamente enumerabile e anche ricorsivo così come K, poiché quest'ultimo è ricorsivamente enumerabile per la proprietà 1.10.3; per provare la seconda disuguaglianza basta considerare che $A \leqslant_{rec} B$ se e solamente se $\bar{A} \leqslant_{rec} \overline{B}$ (cf. la proprietà 1.10.4). Quindi se valesse $K \leqslant_{rec} \overline{K}$ avremmo anche $\overline{K} \leqslant_{rec} K$, che è appena stato dimostrato falso.

Naturalmente l'insieme RE-completo non può che essere K!

Teorema 1.10.3. *K è RE-completo, ovvero $\leqslant_{rec}$-completo per RE.*

Dimostrazione. Dobbiamo dimostrare che se $A \in RE$ allora $A \leqslant_{rec} K$. Per definizione A è il dominio di una funzione calcolabile ψ, cioè $A = \{x \mid \psi(x)\downarrow\}$. A partire da ψ si definisca una funzione ψ' a due variabili di cui ignora la seconda, cioè sia $\psi'(x, y) = \psi(x)$, che è a sua volta una funzione calcolabile e quindi avrà un indice, diciamo i; in simboli $\psi' = \varphi_i$. Allora, per il teorema del parametro $\psi'(x, y) = \varphi_i(x, y) = \varphi_{s(i,x)}(y)$, con s calcolabile, iniettiva e totale. Posso riscrivere la definizione di A come segue:

$$\begin{aligned}
A &= \{x \mid \psi(x)\downarrow\} \\
&= \{x \mid \psi'(x, y)\downarrow\} \\
&= \{x \mid \varphi_i(x, y)\downarrow\} \\
&= \{x \mid \varphi_{s(i,x)}(y)\downarrow\} \text{ per il teorema del parametro} \\
&= \{x \mid \varphi_{s(i,x)}(s(i, x))\downarrow\} \text{ ponendo } y = s(i, x) \\
&= \{x \mid s(i, x) \in K\}
\end{aligned}$$

quindi $x \in A$ se e solamente se $f(x) \in K$, con $f(x) = \lambda x.\ s(i, x)$, che è totale, calcolabile e iniettiva perché la $s(i, x)$ lo è (si veda la dimostrazione di pagina 40: prendiamo i tale che $\psi'(x, y) = \varphi_i(x, y) = \varphi_{s(i,x)}(y)$ da cui $f = \lambda x.\ s(i, x)$). □

Esempio 1.10.1. Vediamo il seguente esercizio, affatto banale, con cui si mostra che l'insieme degli indici delle funzioni calcolabili totali, cioè *rec*, è indecidibile. Dimostriamo che

$$K = \{x \mid \varphi_x(x) \downarrow\} \leqslant_{rec} \{x \mid \varphi_x \in rec\} = TOT$$[19]

Definiamo ora questa funzione:

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

La nostra ψ è calcolabile parziale: il programma P_x calcola $\varphi_x(x)$ e se e quando questa converge, restituisce 1 per ogni y. Per il teorema *s-m-n* esiste f calcolabile totale iniettiva tale che $\varphi_{f(x)}(y) = \psi(x,y)$. (Per costruire la f si ricordi la dimostrazione di pagina 40: si scelga i tale che $\psi(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$ da cui $f = \lambda x.\ s(i,x)$.) Adesso

$$x \in K \Rightarrow \varphi_{f(x)} = \psi(x,y) = \lambda y.\ 1 \Rightarrow \varphi_{f(x)} \text{ totale} \Rightarrow f(x) \in TOT$$

$$x \notin K \Rightarrow \varphi_{f(x)} = \lambda y.\ \text{indefinito} \Rightarrow \varphi_{f(x)} \text{ non è totale} \Rightarrow f(x) \notin TOT$$

Di conseguenza, TOT è *ben che ci vada* ricorsivamente enumerabile.

Nell'esercizio appena svolto abbiamo usato un insieme che contiene *tutti e soli* gli indici delle funzioni calcolabili che hanno la proprietà di essere totali. La stessa cosa può essere fatta considerando altre proprietà delle funzioni. Per esempio, si potrebbe definire l'insieme A di *tutti e soli* gli indici dei programmi che calcolano una particolare funzione φ (si confronti questo insieme A con A_x, definito nel teorema 1.9.2 che contiene *solo* indici della stessa funzione, ma *non tutti*); definito un tale insieme A, ci si può riferire indifferentemente ad esso oppure a φ in quanto descrivono entrambi la *medesima* entità. Più formalmente abbiamo la seguente definizione.

Definizione 1.10.6. A è un *insieme di indici che rappresentano le funzioni* se e solamente se

$$\forall x, y.\ \text{se } x \in A \text{ e } \varphi_x = \varphi_y \text{ allora } y \in A$$

Adesso passiamo a studiare quegli insiemi di indici A che rappresentano le funzioni e tali per cui $K \leqslant_{rec} A$ (oppure $K \leqslant_{rec} \bar{A}$). Così sapremo quali classi di funzioni sono al massimo semi-decidibili (ma non decidibili), perché, come detto sopra, un insieme di indici A che rappresentano le funzioni *individua esattamente* le funzioni calcolate dalle macchine che hanno indice in A.

[19] Non si confondano TOT e *rec*: il primo è un insieme di *indici*, cioè di macchine, programmi; il secondo è un insieme di *funzioni*.

Teorema 1.10.4. *Sia A un insieme di indici che rappresentano le funzioni tale che $\emptyset \neq A \neq \mathbb{N}$. Allora $K \leqslant_{rec} A$ oppure $K \leqslant_{rec} \bar{A}$.*

Dimostrazione. Prendi i_0 tale che $\varphi_{i_0}(y)$ sia ovunque indefinita. Supponiamo che $i_0 \in \bar{A}$ e dimostriamo $K \leqslant_{rec} A$ (se $i_0 \in A$ si procede in modo simmetrico). Poiché $A \neq \emptyset$ scegli $i_1 \in A$. Hai $\varphi_{i_0} \neq \varphi_{i_1}$ perché A è un insieme di indici che rappresentano le funzioni. Definiamo adesso la seguente funzione che è calcolabile:

$$\psi(x,y) = \varphi_{f(x)}(y) = \begin{cases} \varphi_{i_1}(y) & \text{se } x \in K \\ \text{indefinita} = \varphi_{i_0}(y) & \text{altrimenti} \end{cases}$$

dove, usando il teorema *s-m-n*, abbiamo determinato la f funzione calcolabile totale iniettiva (come suggerito nella dimostrazione di pagina 40: sia i tale che $\psi(x,y) = \varphi_i(x,y) = \varphi_{s(i,x)}(y)$, allora si pone $f = \lambda x.\ s(i,x)$). Allora

$$x \in K \Rightarrow \varphi_{f(x)} = \varphi_{i_1} \Rightarrow f(x) \in A$$

perché $i_1 \in A$ e A è un insieme di indici che rappresentano le funzioni e quindi anche $f(x) \in A$. Viceversa, dato che $i_0 \in \bar{A}$,

$$x \notin K \Rightarrow \varphi_{f(x)} = \varphi_{i_0} \Rightarrow f(x) \in \bar{A}\ (\Rightarrow f(x) \notin A)$$

□

Nota: esistono insiemi B (per esempio TOT) tali che $K \leqslant_{rec} B$ e *anche* $K \leqslant_{rec} \overline{B}$, cioè esistono f e g calcolabili totali iniettive tali che $x \in K$ se e solo se $f(x) \in B$ e $x \in K$ se e solo se $g(x) \in \overline{B}$.

Il seguente corollario del teorema precedente è di immediata dimostrazione ed è di particolare importanza, tanto da essere chiamato teorema, perché pone dei limiti drastici alle proprietà dimostrabili sulle funzioni calcolabili.

Teorema 1.10.5 (Rice)**.** *Sia $\mathcal{A}$ una classe di funzioni calcolabili. L'insieme $A = \{n \mid \varphi_n \in \mathcal{A}\}$ è ricorsivo se e solo se $\mathcal{A} = \emptyset$ oppure $\mathcal{A}$ è la classe di tutte le funzioni calcolabili.*

Dimostrazione. Si noti che A è un insieme di indici mentre $\mathcal{A}$ è una classe di *funzioni* (anche se la lettera è la stessa, il carattere è *diverso*! a indicare che i primi sono *sintassi*, mentre i secondi sono *semantica*).

La dimostrazione è immediata per i casi banali, cioè quando $\mathcal{A} = \emptyset$ e quando $\mathcal{A}$ è la classe di *tutte* le funzioni calcolabili.

Negli altri casi, basta applicare il teorema 1.10.4, poiché A è un insieme di indici che rappresentano le funzioni, il quale non è vuoto, perché $\mathcal{A}$ contiene almeno una funzione, né coincide con $\mathbb{N}$, perché $\mathcal{A}$ non contiene tutte le funzioni calcolabili. □

Questo importante risultato negativo ovviamente si ripercuote sulle proprietà che si possono dimostrare sui programmi: ogni metodo di prova si scontra inevitabilmente con il problema della fermata. Gli informatici però non si sono arresi e hanno sviluppato svariate tecniche per aggirare il problema. Una famiglia di analizzatori di programmi particolarmente usata è quella che va sotto il nome di *analisi statica*. In breve, il *testo* del programma viene scrutinato e si raccolgono informazioni su come gli oggetti che vi compaiono (per esempio variabili, chiamate di procedura, ecc.) verrano usati a tempo di esecuzione (per esempio se i valori che verranno assegnati alle variabili sono del giusto tipo, se prima di usarle avranno ricevuto un valore di inizializzazione, ecc.). Il gioco ha successo perché il comportamento dei programmi viene *approssimato* in modo sicuro, ovvero ciò che viene predetto è una sovra-approssimazione di ciò che succederà davvero a tempo di esecuzione (per esempio potrà succedere di dire che tra i valori che si potrebbero assegnare a una variabile intera c'è una stringa senza che questo avvenga davvero a tempo di esecuzione, ma *non* capiterà mai di dire che tutti i valori sono interi se a tempo di esecuzione a tale variabile viene assegnata una stringa). In altre parole si dice che si può sbagliare rimanendo sul lato giusto, ovvero senza conseguenze, perché si afferma che un programma corretto non lo è, ma mai si afferma che un programma scorretto è invece corretto. A questa famiglia di analizzatori appartengono vari strumenti spesso incorporati nei compilatori, da cui il nome statici, tra cui i *type-checker*, gli analizzatori *data-flow* o *control-flow* e molti altri ancora.

Un'applicazione immediata del teorema di Rice è che

$$K_1 = \{x \mid \text{dominio}(\varphi_x) \neq \emptyset\}$$

cioè l'insieme (degli indici) delle funzioni che sono definite in almeno un punto *non è ricorsivo*, sebbene sia ricorsivamente enumerabile. Inoltre si può dimostrare facilmente che $K \equiv K_0 \equiv K_1$, cioè i tre insiemi si riducono l'uno all'altro e sono RE-completi.

Altre classi che non sono ricorsive sono

$\text{FIN} = \{x \mid \text{dominio}(\varphi_x) \text{ finito}\}$

$\text{INF} = \{x \mid \text{dominio}(\varphi_x) \text{ è infinito}\} = \mathbb{N} \backslash \text{FIN}$

$\text{TOT} = \{x \mid \varphi_x \text{ totale}\} = \{x \mid \text{dominio}(\varphi_x) = \mathbb{N}\}$

$\text{REC} = \{x \mid \text{dominio}(\varphi_x) \text{ è ricorsivo}\}$

$\text{CONST} = \{x \mid \varphi_x \text{ totale e costante}\}$

$$\text{EXT} = \{x \mid \varphi_x \text{ è estendibile a funzione calcolabile totale}\}$$

Si noti che trova qui la sua giustificazione l'affermazione fatta a suo tempo che non esiste un algoritmo che termini sempre (e nemmeno una funzione di semi-decisione, per quanto detto subito sotto) per trovare tutte le funzioni totali o tutte quelle estendibili.

Inoltre si può vedere che gli insiemi listati sopra *non sono nemmeno* ricorsivamente enumerabili. Infatti, si può dimostrare che

$$\overline{K} \leqslant_{rec} \text{FIN}, \ldots, \text{TOT}, \ldots, \text{EXT}$$

Visto che $\overline{K}$ non è ricorsivamente enumerabile (altrimenti sarebbe ricorsivo così come lo sarebbe K), ne segue immediatamente che questi problemi non si possono nemmeno semi-decidere!

Concludiamo con un paio di osservazioni ovvie. Il complemento di un insieme non ricorsivamente enumerabile può essere a sua volta non ricorsivamente enumerabile, come mostrato da FIN e dal suo complemento INF; si ricordi che ciò non è vero per gli insiemi ricorsivamente enumerabili, a meno che non siano anche ricorsivi. Inoltre, tra i sottoinsiemi di un insieme non ricorsivamente enumerabile (per esempio INF) ve ne sono sia di non ricorsivamente enumerabili (per esempio REC), che di ricorsivamente enumerabili (per esempio K), che di ricorsivi (per esempio $\emptyset$); il che è banalmente vero anche per gli insiemi ricorsivamente enumerabili e per quelli ricorsivi, primo fra tutti $\mathbb{N}$.

Capitolo 2

Complessità

2.1 Una teoria quantitativa degli algoritmi

Nella prima parte abbiamo studiato alcuni aspetti estensionali della teoria della calcolabilità. Infatti, abbiamo caratterizzato i problemi risolubili, rappresentati mediante funzioni calcolabili totali o insiemi ricorsivi, quelli insolubili o semi-decidibili, rappresentati da funzioni calcolabili o insiemi ricorsivamente enumerabili, e infine abbiamo discusso problemi nemmeno semi-decidibili, rappresentati da funzioni non calcolabili o insiemi non ricorsivamente enumerabili. Quindi l'enfasi è stata posta su *cosa* si calcola.

Passiamo ora a studiare *come* si calcola e *quali risorse* siano necessarie per calcolare, ovvero spostiamo l'accento sugli aspetti intensionali dei problemi, cioè introduciamo una *teoria quantitativa* dei problemi analizzando il comportamento degli *algoritmi* durante la loro esecuzione.

Considereremo nel seguito solo problemi *decidibili*, limitandoci a quelli di *decisione*, cioè guarderemo solo quei problemi, o insiemi, per cui esiste un algoritmo che ne calcola la funzione caratteristica. Per fare ciò, useremo alcune opportune varianti delle macchine di Turing, considerandole come automi accettori. In altre parole, le macchine usate finiranno in uno stato di accettazione se il dato in ingresso appartiene all'insieme, e quindi quel caso particolare del problema è risolto positivamente, altrimenti raggiungeranno uno stato di rifiuto.

Le risorse di cui si tiene principalmente conto quando si scrivono o si utilizzano programmi sono relative al *tempo* e allo *spazio*. Tuttavia queste non sono le uniche risorse critiche usate durante il calcolo; per esempio può essere importante valutare il dispendio energetico, che nel caso di macchine mobili può non coincidere esattamente con il tempo di calcolo, oppure il numero delle comunicazioni che avvengono tra i processori, in macchine parallele o distribuite. Nel seguito ci limiteremo a studiare, da un punto di vista astratto, solo quanto tempo e/o spazio sono necessari alla soluzione di un problema, ovvero alla soluzione di tutti i suoi casi; esamineremo anche in gran fretta pregi e manchevolezze di un tentativo di prescindere da tempo e spazio, in un quadro ancor più astratto. In ogni caso, è necessario calcolare le risorse necessarie alla soluzione di un caso del problema in funzione dei suoi dati di ingresso, o meglio della loro *taglia*. La taglia del dato di ingresso x, che esprimeremo come $|x|$, verrà opportunamente misurata in base ad alcuni criteri, quali una stima dell'occupazione di memoria, il numero dei componenti del dato medesimo o altri ancora.

Ricapitolando: dato un problema I (sotto forma di un insieme) cercheremo una funzione $f(|x|)$ che esprima la quantità di risorse in spazio o tempo necessaria al calcolo della soluzione di $x \in I$, che spesso chiameremo il caso x di I. Quella che andremo a delineare risulta quindi una teoria della *com-*

plessità asintotica, perché si studia come cresce l'uso delle risorse al crescere dei dati di ingresso.

La giustificazione concreta per lo sviluppo di una teoria della complessità astratta è, come già detto, quello di misurare l'efficienza degli algoritmi e, in ultima analisi, rispondere quando possibile alla domanda seguente: *qual è il modo più efficiente di risolvere un problema*? Noi non risponderemo a questa domanda dando le tecniche e i metodi per progettare e realizzare algoritmi — argomento di altri corsi — anche se considereremo alcuni problemi paradigmatici e alcuni algoritmi per deciderli. Ci limiteremo piuttosto a capire, seppur superficialmente, le caratteristiche che tali problemi e quegli algoritmi hanno. Detto in maniera molto informale e intuitiva, cercheremo di individuare una parte della struttura *profonda* che accomuna molti problemi, o meglio quella del modo di risolverli, in un senso molto simile a quanto fatto nella prima parte del corso, quando abbiamo raggruppato i problemi decidibili tra loro e li abbiamo separati da quelli indecidibili. Ancora: cercheremo di evidenziare per quanto possibile la struttura matematica comune a gruppi di problemi, manifestata dal fatto che i loro algoritmi risolutori si "trasformano" gli uni negli altri, mediante opportune funzioni di riduzione.

Per far ciò è necessario prima studiare come si possono esprimere limiti alle risorse necessarie al calcolo: come si definisce e quali proprietà ha una funzione f (dai naturali ai naturali) che sia in grado di stimare la quantità minima di risorse necessarie per risolvere un certo problema I. Ovviamente, vogliamo far sì che un algoritmo M che risolve il caso $x \in I$ richieda una quantità di risorse spazio/temporali inferiore, o meglio ancora uguale a $f(|x|)$: f è quindi la minima delle funzioni che maggiorano la quantità di risorse necessarie al calcolo di $x \in I$. Si noti che se un algoritmo ha bisogno di una certa quantità di risorse, funzionerà benissimo con risorse maggiori e che invece se fornissimo a tale algoritmo una quantità di risorse inferiore a $f(|x|)$, allora potremmo non essere in grado di decidere con esso il caso $x \in I$. Trovata una tale funzione che stima la quantità di risorse necessarie, diremo piuttosto imprecisamente che il problema I ha *complessità* in tempo o spazio f. Si noti che f deve stimare le risorse necessarie alla soluzione di *tutti* i casi del problema, incluso il caso più difficile che ne richiede in misura maggiore; stiamo dunque parlando di complessità nel caso *pessimo*.

È importante osservare che determinare questa funzione f, se e quando ciò è possibile, richiede di considerare tutti gli algoritmi che risolvono il problema I, i quali sono tanti quanti i numeri naturali (v. il teorema 1.9.1). Infatti, per quanto appena detto, per stabilire che I ha complessità $f(n)$ *bisogna costruire un* programma che risolve il caso $x \in I$ in tempo o spazio inferiore o uguale a $f(|x|)$. Ecco allora che, data una certa funzione f, si può definire una *classe di complessità*, dipendente da f, cui appartengono tutti quei problemi per

cui esiste almeno un algoritmo che li risolve e che richiede risorse in misura inferiore o uguale a $f(n)$.

Allora viene naturale chiedersi: se il limite $f(n)$ alle risorse disponibili viene aumentato fino a un certo $g(n)$, cioè $g(n) > f(n)$, la classe di problemi che si possono risolvere diviene più ampia? Ad esempio, aumentando le risorse di memoria di una certa macchina da $f(n) = 2n$ a $g(n) = n^2$, riusciamo a risolvere problemi che prima non potevamo risolvere? In altre parole, esistono delle classi i cui problemi sono *risolubili* con dei limiti alle risorse *prefissati*? ovvero esistono davvero *classi di complessità* diverse tra loro? e se sì, quali sono le relazioni che sussistono tra esse? Vedremo che in alcuni casi interessanti si possono stabilire delle gerarchie precise tra classi di problemi, e quindi classificare i problemi in base alla quantità di risorse necessarie per risolverli. Tuttavia, vedremo anche che vi sono dei casi in cui non è noto se vi sia una gerarchia in senso stretto, o addirittura che non è possibile stabilirne alcuna, a causa di funzioni di stima bizzarre.

Nel caso in cui si possano definire precisamente delle classi di complessità, queste hanno dei *problemi completi*? (Cioè problemi che appartengono a una classe e che sono i più *ardui* da risolvere con quelle limitazioni spazio/temporali? cf. la definizione 1.10.4.)

E inoltre: esistono *classi particolarmente interessanti* dal punto di vista computazionale? e siccome una classe non è interessante senza almeno un problema completo che sia noto prima dell'introduzione della classe medesima, la stessa domanda implica anche: *vi sono problemi completi particolarmente interessanti*? Avete certamente già incontrato nei corsi precedenti le seguenti due classi di complessità

- **Ptime**, o semplicemente $\mathcal{P}$, la classe dei problemi risolubili in tempo polinomiale *deterministico*.
- **NPtime**, o semplicemente $\mathcal{NP}$, la classe dei problemi risolubili in tempo polinomiale *non-deterministico*.

Queste due classi sono particolarmente rilevanti e spesso vengono associate rispettivamente alla classe dei problemi "trattabili" e a quella dei problemi "intrattabili". Questa associazione è talmente forte, sebbene non scevra da critiche che riporteremo nel seguito, da essersi meritata il nome di TESI DI COOK E KARP, in analogia con la tesi di Church e Turing.

Prima di descrivere la teoria che abbiamo intuitivamente delineato, ci sono tre condizioni preliminari sulle quali ci soffermiamo.

Innanzitutto, bisogna mettersi d'accordo su *come si misura* lo spazio o il tempo necessario per risolvere un problema. Di solito, il costo di una computazione è la somma dei costi dei suoi passi elementari. Ad esempio, se il

modello di calcolo sono le macchine di Turing, il tempo è misurato di solito dal numero dei passi necessari per arrivare alla configurazione terminale; se il modello è quello di un linguaggio funzionale, può essere il numero delle chiamate di funzione o delle riduzioni; se è un linguaggio di tipo *WHILE* può essere il numero dei confronti, o delle operazioni di somma o di moltiplicazione o il numero degli accessi in memoria, opportunamente pesati. Nel seguito non studieremo in dettaglio i vari modi per definire le funzioni di costo, ma ci limiteremo a osservare che bisogna aver molta cura nella loro scelta. Inoltre, le funzioni di costo così scelte devono consentire una buona formalizzazione di ciò che sembra accadere in realtà, quale ad esempio l'intuizione forte che al crescere delle risorse cresca anche la classe dei problemi risolvibili con esse. Infine, riporteremo anche un paio di fatti, intuitivamente paradossali, che si verificano quando si scelgano funzioni di costo in una certa misura arbitrarie, seppur apparentemente soddisfacenti.

Altra condizione preliminare è quella di garantire che la complessità di un problema non cambi al variare del modello di calcolo in cui la funzione che decide il problema è espressa. In altre parole, vogliamo essere sicuri che i risultati della teoria della complessità che andremo a enunciare siano *invarianti rispetto al modello di calcolo* scelto. Ci poniamo cioè l'obiettivo di avere una teoria *robusta*. Fortunatamente, sotto ragionevolissime ipotesi, la scelta del modello di calcolo *non influenza* le classi di complessità che esso induce al variare delle risorse spazio/temporali disponibili.[1] Quindi potremo parlare di classi di complessità in generale, senza specificare da quale modello siano state indotte. Non studieremo approfonditamente il problema di garantire che tali classi siano effettivamente chiuse rispetto a "ragionevoli" trasformazioni di modelli, né caratterizzeremo in dettaglio quali trasformazioni siano ragionevoli. Ci basterà mostrare, mediante un esempio, come modifiche apparentemente innocue apportate a un modello possono mutare drasticamente le classi di complessità, se il modo in cui si definiscono le funzioni di costo non viene anch'esso opportunamente modificato. Infine, ci limiteremo a enunciare, senza dimostrarli, alcuni teoremi che assicurano che le classi di complessità, in particolare $\mathcal{P}$ e $\mathcal{NP}$, sono sostanzialmente invarianti rispetto al modello di calcolo.

La terza condizione per costruire una teoria quantitativa degli algoritmi accettabile, è che la complessità di un problema sia *invariante rispetto alla rappresentazione* del problema stesso. Ad esempio, la complessità di un problema sui grafi *non deve* dipendere dal fatto che questi siano rappresen-

[1] Per far ciò si ricorre a riduzioni da un modello di calcolo all'altro, che sono "semplici" in un senso diverso da quello visto nella prima parte, ma altrettanto preciso; ne faremo solo un breve cenno.

tati come matrici o come liste di adiacenza o ancora come coppie di insiemi (N, A), dove N sono i nodi e A sono gli archi. In altre parole, la richiesta di invarianza rispetto alla rappresentazione significa che le classi di complessità sono chiuse rispetto al cambio di rappresentazione dei dati. Abbiamo già incontrato la stessa richiesta quando abbiamo affermato che i risultati di calcolabilità *non* dipendono dal formato dei dati e allora abbiamo usato funzioni di codifica (calcolabili e) totali per passare da una rappresentazione a un'altra; qui chiederemo inoltre che le funzioni che trasformano i dati da un formato a un altro siano *facili*, in un senso che sarà più chiaro in seguito. Naturalmente ci possono essere casi in cui la complessità cambia al cambiare della rappresentazione, ma sono casi degeneri, seppure molto interessanti da indagare perché ammettono algoritmi che li risolvono in modo particolarmente efficiente, sfruttando la loro struttura matematica. Ad esempio, si considerino i casi in cui si debbano moltiplicare matrici sparse o di altra forma speciale. Non si dimentichi che una notevole parte dell'ingegno dell'infernatico si esercita proprio nel trovare le giuste strutture dei dati!

Concludiamo questa breve introduzione ripetendo una domanda che ci siamo già posti: *le classi di complessità formano una gerarchia*? La letteratura riporta, sotto ragionevolissime ipotesi, un gran numero di risultati che stabiliscono le relazioni tra varie classi di complessità. Noi vedremo solamente un piccolissimo frammento di questa gerarchia, riassumibile nella seguente catena di inclusioni tra classi, la cui definizione posponiamo

$$\text{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subset \mathcal{R} \subset RE$$

Oltre alle inclusioni proprie esplicitamente rappresentate nella parte destra della formula, è noto anche che LOGSPACE $\subset$ PSPACE. Non è ancora noto invece quali altre inclusioni siano proprie. In particolare, non è noto se $\mathcal{P} \subset \mathcal{NP}$ o se piuttosto $\mathcal{P} = \mathcal{NP}$ e nemmeno se ciò sia dimostrabile o meno: non si è tuttora riusciti a ottenere quel risultato di separazione tra le due classi che ci è riuscito nella prima parte considerando R ed RE – là il gioco di separarle riesce perché decidere la non appartenenza di un elemento a un insieme ricorsivo richiede un tempo *finito*, mentre questa limitazione di finitezza non può essere soddisfatta quando l'insieme è ricorsivamente enumerabile e non ricorsivo; qui il gioco non riesce affatto, non tanto perché sia i problemi in $\mathcal{P}$ che quelli in $\mathcal{NP}$ sono decidibili e quindi lo sono *per definizione* in tempo finito, ma soprattutto perché, e lo vedremo più in dettaglio, tali insiemi sono per l'appunto decidibili in tempo *limitato*.

2.2 Misure di complessità deterministiche

In questo capitolo studieremo brevemente come associare a una macchina di Turing una funzione che stimi il tempo che le è necessario per risolvere un caso $x \in I$ del problema I che decide. Poi vedremo un paio di teoremi che mostrano come e di quanto questo tempo possa essere ridotto, al prezzo di usare macchine con un hardware più "efficiente". Preliminarmente introdurremo una variante "parallela" delle macchine di Turing, permettendo alla macchina di operare simultaneamente su molti nastri. Questa estensione naturalmente non modifica la classe dei problemi decidibili, tuttavia ci consente in alcuni casi un trattamento più agevole. Inoltre è un buon modello delle attuali macchine parallele sincrone (ma non di quelle concorrenti e distribuite e men che meno di quelle impiegate nel paradigma chiamato "mobile computing"!), nella stessa misura in cui le macchine di Turing a un nastro lo sono delle macchine sequenziali mono-processore. Vedremo infine quale sia il prezzo in termini di tempo che si deve pagare per simulare una macchina a molti nastri su una a un nastro solo, dando così un preciso limite teorico ai vantaggi ottenibili dall'introduzione di calcolatori paralleli.

Lo stesso schema verrà seguito per misure che riguardano lo spazio necessario al calcolo. Introdurremo un'ulteriore variante delle macchine di Turing in cui si identificano i nastri di lavoro (ignorando quelli destinati ai dati in ingresso e in uscita), in modo da definire lo spazio necessario per risolvere un problema. Anche in questo caso, la potenza espressiva non cambia. Infine, mostreremo che lo spazio può essere compresso in modo analogo a quanto fatto per ridurre il tempo.

L'aggettivo *deterministico* che compare nel titolo è legato al fatto che le macchine di Turing che impieghiamo usano una *funzione* di transizione, come quelle usate nella prima parte del corso. Ciò garantisce che ogni passo di computazione è univocamente determinato dallo stato e dal simbolo correnti, in altri termini, la macchina è *deterministica*. In seguito vedremo cosa succede impiegando *relazioni* di transizione piuttosto che funzioni, il che rende le macchine di Turing *non deterministiche*, in un senso che sarà precisato.

2.2.1 Macchine di Turing a k-nastri

Ricordiamo che per le macchine di Turing introdotte nella definizione 1.2.1 si postula l'esistenza di un nastro semi-infinito e di una funzione di transizione δ che opera su di esso. Adesso arricchiamo l'hardware delle macchine di Turing fornendole di k nastri. Poiché trattiamo solo problemi di decisione, ci prendiamo la libertà di spezzare lo stato di arresto h in due nuovi stati

di arresto *SI*, *NO*, a rappresentare che la macchina si ferma con successo nel primo caso e con insuccesso nel secondo. Formalmente:

Definizione 2.2.1 (MdT a k nastri)**.** Dato un numero naturale k, una *Macchina di Turing a k nastri* è una quadrupla $M = (Q, \Sigma, \delta, q_0)$, con

- $\#, \triangleright \in \Sigma$ e $L, R, - \notin \Sigma$
- *SI*, *NO* $\notin Q$
- $\delta : Q \times \Sigma^k \to Q \cup \{SI, NO\} \times (\Sigma \times \{L, R, -\})^k$ è la funzione di transizione, soggetta alle stesse condizioni della definizione 1.2.1 sull'unicità della stringa in $(\Sigma \times \{L, R, -\})^k$, in modo che δ sia una funzione, e sull'uso del carattere di inizio stringa $\triangleright$.

La funzione di transizione δ per uno stato q e k simboli $\sigma_1, \ldots, \sigma_k$ ha allora la forma seguente

$$\delta(q, \sigma_1, \ldots, \sigma_k) = (q', (\sigma'_1, D_1), (\sigma'_2, D_2), \ldots, (\sigma'_k, D_k))$$

Una configurazione di una macchina a k nastri ha la forma:

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

dove il carattere corrente sull'i-esimo nastro è σ_i, che abbiamo evitato di sottolineare per non appesantire la notazione; le sue computazioni lunghe n saranno rappresentate da

$$(q, w_1, w_2, \ldots, w_k) \to^n (q', w'_1, w'_2, \ldots, w'_k)$$

le cui mosse, derivabili in accordo con la funzione di transizione δ, hanno la forma

$$(q, u_1\sigma_1 v_1, u_2\sigma_2 v_2, \ldots, u_k\sigma_k v_k) \to (q, u_1\sigma_1 v'_1, u_2\sigma_2 v'_2, \ldots, u_k\sigma_k v'_k)$$

Si noti che se si volesse rappresentare una situazione in cui ci sono davvero k processori che evolvono in sincronia, ciascuno con il suo carattere corrente e con il suo stato preso da un insieme Q_i, basterebbe definire l'insieme della macchina a k nastri come $Q = Q_1 \times Q_2 \times \cdots \times Q_k$ e interpretare nel modo ovvio la funzione di transizione in modo da tenerne conto.

Vediamo adesso un esempio di macchina di Turing con due nastri.

Esempio 2.2.1. La seguente MdT con 2 nastri riconosce le stringhe palindrome costruite sull'alfabeto $\{a, b\}$. La funzione di transizione può essere suddivisa in tre "blocchi omogenei". Nel primo blocco ci sono le istruzioni

che ricopiano la stringa in ingresso sul secondo nastro; nel secondo la testina del primo nastro vien portata sul simbolo di inizio nastro, mentre quella del secondo nastro è lasciata sul primo carattere # dopo la stringa. Il terzo blocco di istruzioni effettua il controllo vero e proprio, cancellando dal secondo nastro a partire da *destra* i caratteri della stringa di ingresso se e solamente se corrispondono a quelli della stringa originale, incontrati muovendo il cursore del primo nastro da *sinistra*. Ovviamente, la stringa è palindroma se le due testine si trovano su caratteri sempre uguali e il secondo nastro viene svuotato.

q	σ_1	σ_2	$\delta(q,\sigma_1,\sigma_2)$		
q_0	$\triangleright$	$\triangleright$	q_0	$(\triangleright, R)$	$(\triangleright, R)$
q_0	a	$\#$	q_0	(a, R)	(a, R)
q_0	b	$\#$	q_0	(b, R)	(b, R)
q_0	$\#$	$\#$	q_0	$(\#, L)$	$(\#, -)$
q_1	a	$\#$	q_1	(a, L)	$(\#, -)$
q_1	b	$\#$	q_1	(b, L)	$(\#, -)$
q_1	$\triangleright$	$\#$	q_2	$(\triangleright, R)$	$(\#, L)$
q_2	a	a	q_2	(a, R)	$(\#, L)$
q_2	b	b	q_2	(b, R)	$(\#, L)$
q_2	a	b	*NO*	(a, R)	$(a, -)$
q_2	b	a	*NO*	(b, R)	$(b, -)$
q_2	$\#$	$\triangleright$	*SI*	$(\#, -)$	$(\triangleright, R)$

Come esempio di calcolo applichiamo la macchina alla stringa *abba*, raggruppando per quanto possibile i passi della computazione in blocchi omogenei.

$(q_0, \underline{\triangleright}abba, \underline{\triangleright}\#) \to (q_0, \triangleright\underline{a}bba, \triangleright\underline{\#}) \to$

$(q_0, \triangleright a\underline{b}ba, \triangleright a\underline{\#}) \to^3 (q_0, \triangleright abba\underline{\#}, \triangleright abba\underline{\#}) \to$

$(q_1, \triangleright ab\underline{b}a, \triangleright abba\underline{\#}) \to (q_1, \triangleright abb\underline{a}\#, \triangleright abba\underline{\#}) \to^4 (q_1, \underline{\triangleright}abba, \triangleright abba\underline{\#})$

$(q_2, \triangleright\underline{a}bba, \triangleright abb\underline{a}\#) \to (q_2, \triangleright a\underline{b}ba, \triangleright ab\underline{b}\#) \to (q_2, \triangleright ab\underline{b}a, \triangleright a\underline{b}\#) \to$

$(q_2, \triangleright abb\underline{a}, \triangleright\underline{a}\#) \to (q_2, \triangleright abba\underline{\#}, \underline{\triangleright}) \to (SI, \triangleright abba\underline{\#}, \triangleright\underline{\#})$

2.2.2 Complessità in tempo deterministico

Introduciamo adesso il modo che useremo per determinare il tempo necessario alla soluzione di un problema, ricordando che per problema qui intendiamo

l'appartenenza o meno a un insieme, ovvero a un linguaggio di cui le macchine di Turing sono gli automi accettori; poiché le macchine usate sono *deterministiche*, anche le misure che introdurremo sono tali e spesso ometteremo tale aggettivo, dandolo per inteso.

Definizione 2.2.2. Diciamo che t è il *tempo richiesto* da una MdT M a k nastri[2] per *decidere* il caso $x \in I$ se

$$(q_0, \trianglerighteq x, \trianglerighteq, \ldots, \trianglerighteq) \to^t (H, w_1, w_2, \ldots, w_k), \text{ con } H \in \{SI, NO\}$$

In realtà vorremmo ottenere una misura del tempo necessario a risolvere un problema mediante la macchina M come una funzione della taglia dei suoi possibili dati di ingresso x; cioè, indicando la taglia di x con $|x|$, vorremmo una funzione $f(|x|)$. La definizione della taglia dei dati è arbitraria, ma spesso è molto naturale. Per esempio, spesso la taglia di un grafo è il numero dei suoi nodi (e/o dei suoi archi), quella di una stringa o di un vettore la sua lunghezza, indipendentemente dagli elementi costitutivi. La funzione *taglia* deve essere ovviamente calcolabile totale e *facile*; in ogni caso deve restituire un numero naturale. In queste note useremo funzioni di taglia spesso senza definirle e nel modo che ci sarà più conveniente. Nella maggior parte dei casi e senza avvertenza contraria, misureremo i dati di ingresso x in relazione alle caselle della MdT necessarie a contenerli.

Nel seguito, non pretenderemo che la funzione $f(|x|)$ dia il numero *esatto* di passi necessari al calcolo di $M(x)$, perché ciò potrebbe rivelarsi troppo complicato; ci contenteremo allora di approssimare tale numero per eccesso: la macchina non richiederà un tempo maggiore di quello stimato. In conclusione, la funzione che determina la complessità di M è una funzione calcolabile totale $f : N \to N$, la quale limita superiormente il numero dei passi che M compie per risolvere il problema in questione — torneremo sulle caratteristiche delle funzioni di misura nella definizione 2.4.1. Si noti che questo non contrasta con la richiesta di avere classi di complessità, indotte da funzioni di misura, che esprimono la quantità *minima* di risorse necessarie alla decisione dei problemi in esse contenuti: basta trovare la minima funzione che limita superiormente i passi di M. L'aggettivo *deterministico* che compare nella definizione dipende dal fatto che le macchine di Turing usate sono deterministiche, nel senso che sarà più chiaro dopo la definizione 2.3.1 (la componente δ è una funzione e non una relazione); quando sarà chiaro dal contesto che parliamo di questo tipo di macchine, ometteremo tale aggettivo.

[2]Se volessimo considerare il tipo di macchine viste in precedenza, cioè se $H = \{h\}$, allora si potrebbe definire che il *tempo richiesto* da M su x è t. Inoltre, questa definizione sarebbe accettabile anche per la complessità di problemi *tout court* e non solo di problemi decidibili come facciamo, ponendo ∞ il tempo richiesto, se $M(x) \uparrow$.

Definizione 2.2.3. *M decide I in tempo deterministico f* se per ogni dato di ingresso $x \in I$ il tempo t richiesto da M per decidere x è $\leq f(|x|)$.

Adesso possiamo introdurre il concetto di *classe di complessità* in tempo *deterministico*.

Definizione 2.2.4 (Classe di complessità in tempo deterministico)**.**

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

La classe appena introdotta contiene tutti e soli i problemi risolvibili in tempo deterministico f, ovvero affinché un problema vi appartenga occorre e basta che vi sia una macchina M che lo decide in tempo deterministico f.

Prendiamo ad esempio la macchina M dell'esempio 2.2.1 e calcoliamo la sua complessità in tempo, suppondendo che la stringa di ingresso sia lunga n. Come abbiamo visto dianzi, il funzionamento della macchina può essere suddiviso in tre blocchi di operazioni "omogenee":

1. copia il dato in ingresso sul secondo nastro in	$n+1$ passi
2. rimette la prima testina sul $\triangleright$ in	$n+1$ passi
3. sposta le due testine se i simboli sono uguali in	$\underline{n+1}$ passi
	$3n+3$ passi

cui va aggiunto il passo per l'accettazione. Quindi il problema di verificare se una stringa è palindroma appartiene a $\text{TIME}(3n+4)$ o, scordandosi le costanti, è dell'ordine di n.

NOTAZIONE Nella discussione precedente abbiamo menzionato l'ordine di una funzione. Poiché questo concetto viene ripetutamente usato in seguito, in quanto in questa porzione di teoria della complessità si preferisce ignorare le costanti (ne discuteremo più avanti le ragioni), vale la pena di introdurre esplicitamente la seguente abbreviazione, dove "quasi ovunque" significa per ogni argomento, eccetto che per un insieme finito di essi:

$$\mathcal{O}(f) = \{g \mid \exists r \in \mathbb{R}^+.\ g(n) < r \times f(n) \text{ quasi ovunque}\}$$

a indicare che la funzione f cresce allo stesso modo o più velocemente delle funzioni g appartenenti alla classe $\mathcal{O}(f)$, la quale viene quindi chiamata *ordine* di f.[3] (Inutile notare che "più velocemente" dipende solo dal fattore moltiplicativo r.)

[3]Di solito si introducono anche le classi

$\Omega(f) = \{g \mid f \in \mathcal{O}(g)\}$ — f cresce più lentamente di g e
$\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$ — f cresce come g.

In effetti, il calcolo di complessità fatto sopra non tiene conto del fatto che, se la stringa non è palindroma, il numero di passi necessari è minore. Abbiamo infatti definito una misura della complessità nel *caso pessimo*. Ci sono altri modi per misurare la complessità che tengono conto della distribuzione dei dati di ingresso. Un esempio particolarmente rilevante è quello della complessità nel *caso medio*, che tuttavia non tratteremo in queste note.

Può essere interessante confrontare adesso, dal punto di vista della complessità, le macchine che decidono se una stringa è palindroma nelle loro versioni "parallela", appena vista, e "sequenziale". Il tempo delle computazioni di quest'ultima è in $\mathcal{O}(n^2)$, perché servono n passi per controllare se il primo simbolo è uguale all'ultimo e questo controllo va ripetuto per $n/2$ volte. Una prima osservazione che possiamo fare è che il problema di decidere se una stringa è palindroma sta *anche* in $\mathcal{O}(n^2)$, il che non sorprende perché abbiamo appena visto che sta in $\mathcal{O}(n)$ e se una funzione è superiormente dominata da $g \in \mathcal{O}(n)$ lo è a maggior ragione da $h \in \mathcal{O}(n^2)$ — se avendo poco tempo a disposizione risolvo un problema, lo risolverò a maggior ragione se ne avessi di più.

Un'altra osservazione, forse più interessante, è che, usando una macchina parallela, abbiamo "guadagnato" tempo in modo quadratico (meglio: perduto da parallelo a sequenziale). Questo è un fatto vero in generale.

Teorema 2.2.1 (Riduzione del numero di nastri)**.** *Data una macchina di Turing M con k nastri che decide I in tempo deterministico f, allora $\exists M'$ con un solo nastro che decide I in tempo deterministico $\mathcal{O}(f^2)$.*

Dimostrazione. Riportiamo solo una traccia della dimostrazione, confidando nella diligenza dei lettori che vorranno certamente precisare i passi appena accennati. Costruiamo M' in modo che simuli la data M, in modo analogo a quanto fatto nella costruzione della macchina di Turing universale. Ogni configurazione di M della forma

$$(q, \triangleright w_1\sigma_1u_1, \triangleright w_2\sigma_2u_2, \ldots, \triangleright w_k\sigma_ku_k)$$

viene simulata da:

$$(q', \triangleright \triangleright' \, w_1\overline{\sigma}_1u_1 \triangleleft' \triangleright' w_2\overline{\sigma}_2u_2 \triangleleft' \cdots \triangleright' \, w_k\overline{\sigma}_ku_k\triangleleft')$$

per qualche q', cioè racchiudiamo ciascun nastro $w_i\sigma_iu_i$ tra due nuove parentesi $\triangleright'$ e $\triangleleft'$ e usiamo $\#\Sigma$ nuovi simboli $\overline{\sigma}_i$ per ricordarci la posizione della testina sull'i-esimo nastro.

Per cominciare, la macchina M' applicata a x dovrà generare la configurazione che simula la configurazione iniziale di M, cioè dobbiamo passare dalla configurazione iniziale di M $(q_0, \trianglerighteq x, \triangleright, \ldots, \triangleright)$ a

$$(q', \triangleright \triangleright' \, x \triangleleft' \, (\triangleright'\triangleleft')^{k-1})$$

per qualche q'. Per far ciò, bastano $2k+\#\Sigma$ nuovi stati[4] e un certo numero di passi che, essendo dell'ordine di $|x|$, non influenza la complessità asintotica (consideriamo infatti il caso pessimo, quindi tutto il dato iniziale va letto).

Per simulare una mossa di M, la macchina M' scorre l'intero nastro da sinistra a destra e *viceversa* due volte:

- la prima volta M' determina quali sono i simboli correnti di M, $\overline{\sigma}_i$ (si noti che, per ricordare quale sia la stringa $\overline{\sigma}_1 \dots \overline{\sigma}_k$ sono sufficienti $(\#\Sigma)^k$ nuovi stati);
- la seconda volta M' scrive i nuovi simboli nel posto giusto — attenzione! se un $\triangleleft'$ deve essere spostato a destra, per far posto a un nuovo simbolo da scrivere, si verifica una cascata di spostamenti a destra!

Infine quando M si ferma, anche M' si ferma, eventualmente rimuovendo tutte le parentesi $\triangleright'$ e $\triangleleft'$ e sostituendo i caratteri $\overline{\sigma}_i$ con σ_i.

Adesso ricordiamo un fatto generale e ovvio: una macchina non può toccare un numero di caselle maggiore del numero dei passi che compie. Di conseguenza, la lunghezza totale del nastro scritto è al più $K = k \times (f(n)+2)+1$ (l'addendo 2 è dovuto alle parentesi $\triangleright'$ e $\triangleleft'$, l'addendo 1 al simbolo $\triangleright$). Allora, andare due volte avanti e indietro costa, in termini di tempo, per ogni stringa simulata $4K$ più al massimo $3K$ per gli spostamenti a destra, nel caso in cui la casella corrente sia all'estrema sinistra (è il caso pessimo nel quale ci poniamo sempre): K per arrivare alla fine del nastro scritto e $2K$ per spostare a destra i K caratteri, come descritto nella nota precedente. Poiché né k né le altri costanti sono rilevanti, possiamo concludere che per simulare un *singolo* passo di M la macchina M' richiede $\mathcal{O}(f|x|)$ passi sul dato x. Il numero di passi di M' sull'intera computazione è quindi in $\mathcal{O}(f(|x|)^2)$, perché M richiede tempo $f(|x|)$ e perché M' impiega $\mathcal{O}(f(|x|))$ per simulare ogni passo di M. Infine, per costruzione M' è equivalente a M, e quindi le due macchine decidono lo stesso problema; allora M', che ha un nastro solo, decide tale problema in tempo deterministico $\mathcal{O}(f(|x|)^2)$. □

Il teorema appena dimostrato mostra che le MdT sono molto stabili! Infatti, miglioramenti che siano accettabili "algoritmicamente", come aggiungere nastri e processori che operano in parallelo, non solo non cambiano le funzioni

[4]Supponendo che il carattere # non appaia in x, un modo per farlo è il seguente: arrivare al primo carattere # (il che richiede $|x|+1$ passi e un nuovo stato); cambiare stato; tornare indietro di una casella e ricordarsi, codificandolo in un nuovo stato il simbolo ($\neq \#$) corrente, sia a, scriverci #, spostarsi a destra e scrivere a; poi bisogna ripetere le ultime due mosse per $|x|-1$ volte. In questo modo abbiamo "spostato" x di una casella a destra, impiegando $2 \times |x|$ passi e $2+\#\Sigma$ nuovi stati. A questo punto si scrive sulla casella corrente, che è vuota la parentesi $\triangleright'$; si torna sulla prima casella vuota muovendosi a destra e si scrivono $k-1$ coppie $\triangleright'\triangleleft'$, usando altri $2 \times (k-1)$ nuovi stati.

calcolate, come ci aspettavamo, ma non modificano il tempo deterministico richiesto se *non polinomialmente*, quindi le MdT appaiono stabili anche rispetto la tesi di Cook-Karp (ancora da vedere con precisione!).

Ribadiamo adesso l'osservazione fatta nella dimostrazione di sopra, che mette in relazione il tempo e lo spazio necessari alla soluzione di un problema. Se una macchina di Turing M richiede tempo $f(|x|)$ per decidere $x \in I$, significa che si arresta in un numero di passi inferiore a $f(|x|)$, e quindi non può aver visitato, in alcuno dei suoi nastri, un numero di caselle maggiore di $f(|x|)$. Abbiamo quindi il seguente fatto basilare:

Osservazione 1: non si può usare più spazio che tempo!

Usiamo ancora la dimostrazione appena riportata per dedurre un ulteriore osservazione. Infatti, il ragionamento fatto ci suggerisce come misurare, beninteso solo da un punto di vista teorico, i vantaggi che derivano dall'introduzione di macchine parallele. Negli ultimi passi della dimostrazione abbiamo potuto ottenere l'ordine $\mathcal{O}(f(|x|)^2)$ eliminando il fattore k^2, perché k è indipendente da x. Però l'elevamento al quadrato del fattore $f(|x|)$ non potrà mai essere eliminato! Quindi possiamo dedurre una stima del vantaggio che deriva dall'uso del parallelismo.

Corollario 2.2.1. *Le macchine parallele sono polinomialmente più veloci di quelle sequenziali.*

Finora, nei nostri conti abbiamo impiegato solo gli ordini di crescita *trascurando* le costanti. Ovviamente, quando si cerchino stime più precise, le costanti contano terribilmente, tanto che l'astuzia dei progettisti di algoritmi si dispiega spesso proprio nello scoprire come ridurle. Ciò nonostante, continueremo nel seguito a trascurare le costanti, a meno di casi particolari in cui esse saranno menzionate espressamente. Due sono le ragioni:

i) la teoria che si sviluppa è molto più semplice; inoltre per valori grandi di n, cioè per dati di grandi dimensioni, le costanti tendono a valere "poco";

ii) macchine sempre più potenti tendono a far rimpicciolire le costanti.

L'ultima osservazione è sostenuta da un teorema che riportiamo qui sotto, detto di accelerazione lineare. L'idea è che se $I \in \text{TIME}(f)$, (ovvero se esiste una MdT M che lo risolve in tempo deterministico $f(n)$) allora I appartiene anche a $\text{TIME}(\epsilon \times f)$, qualunque sia la scelta per $\epsilon > 0$ (attenzione: poiché la nostra complessità è nel caso pessimo, quanto detto è impreciso e ci sarà bisogno di una correzione per mantenere lineare la misura del tempo). In

altre parole, dato un algoritmo che decide un problema, se ne può sempre trovare uno equivalente che è più veloce per una costante moltiplicativa ϵ (supponendo ovviamente che questa sia minore di 1). Attenzione però: se p.e. $I \in \text{TIME}(2^n)$, ovvero se il problema I è deciso da un algoritmo in tempo esponenziale, non è possibile trovare un algoritmo che lo risolva in tempo deterministico *polinomiale*, per mezzo del *solo* teorema di accelerazione. Un'analoga osservazione vale ovviamente per lo spazio, come vedremo. Quindi il teorema non inficia una eventuale gerarchia, ancora da stabilire.

Il trucco fondamentale è quello di codificare l'alfabeto Σ in un alfabeto "più ricco" Σ^m, con m arbitrario. In pratica, questo significa avere macchine con parole di dimensioni via via crescenti (32, 64, 128, ..., 2^m bit). Si vede quindi che l'accelerazione è legata al *cambio di hardware*.

Questo non è chiaramente del tutto fattibile in pratica, e mostra che le costanti *sono* importanti quando la macchina sia fissata; inoltre, non si può aumentare a piacere l'efficienza di un tuo programma cambiando semplicemente l'hardware delle macchine!

Nell'enunciato del teorema seguente compare un addendo $n + 2$ il quale dipende unicamente dal tipo di MdT usata (a 1 o a k nastri, con nastro semi-infinito o infinito, ecc.). La presenza dell'addendo n garantisce che, anche se la $f(n)$ fosse lineare, la complessità risultante rimarrebbe tale. In enunciati diversi del teorema si possono trovare diversi addendi, che variano in funzione dei vari tipi delle MdT; in tutti i casi essi garantiscono che il risultato sia una funzione almeno lineare. Si noti anche che in questa dimostrazione e in altre che seguiranno, si misura la taglia del dato con il numero di caselle del nastro di ingresso che servono a memorizzarlo.

Teorema 2.2.2 (Accelerazione lineare MdT)**.**
Se $I \in \text{TIME}(f)$*, allora* $\forall \epsilon < 1 \in \mathbb{R}^+$ *si ha che* $I \in \text{TIME}(\epsilon \times f(n) + n + 2)$.

Dimostrazione. Omessa, perché lunga e piena di dettagli insidiosi: si tratta di simulare una data macchina M con una macchina M', sulla falsariga di quanto fatto nella costruzione della macchina universale o nella dimostrazione 2.2.1. Può essere tuttavia interessante notare un fatto che potrebbe guidare il lettore a una maggiore comprensione del teorema stesso e dell'uso che si può fare degli stati per "ricordare" porzioni di nastro. Quello che faremo è vedere che si può determinare il numero m di simboli di M da compattare in un unico simbolo di M' in funzione del *solo* ϵ.

Il primo passo "condensa il dato di ingresso" (in $n+2$ passi, con $n = |x| \leq m \times \left\lceil \frac{|x|}{m} \right\rceil + 2$): ogni sequenza di m simboli di M origina un singolo simbolo di M', cioè $\sigma_{i_1} \dots \sigma_{i_m}$ viene codificata come il singolo simbolo $[\sigma_{i_1} \dots \sigma_{i_m}]$ (si noti che non c'è alcun problema se esiste $m' > 1$ tale che $\sigma_{k_h} = \#$ per

$h \geq m' > 1$). In maniera analoga, gli stati di M' saranno formati da triple $[q, \sigma_{i_1} \dots \sigma_{i_m}, k]$, con $1 \leq k \leq m$, in modo da "rappresentare" il fatto che M si trova nello stato q e ha il cursore sul k-esimo simbolo della stringa $\sigma_{i_1} \dots \sigma_{i_m}$. Data una configurazione, alla macchina M' bastano 6 passi per simularne m della macchina M. Nei primi 4 passi M' va a sinistra, poi a destra, poi ancora a destra e infine ritorna sul carattere corrente $s = \sigma_{i_1} \dots \sigma_{i_m}$, in modo da raccogliere i simboli che M potrebbe visitare e codificarli nel suo stato. Infatti, M con m mosse può spostare il suo cursore all'interno della stringa di m caratteri che si trova a sinistra del carattere corrente, o di quella a destra o lasciarlo all'interno della stringa s considerata. Quando M compie m mosse, M' le simula "a blocchi" muovendosi a sinistra, oppure a destra del simbolo corrente s, ma in ogni caso ne modifica solo due, incluso s – le altre 2 mosse. Basta quindi "prevedere" il risultato di ciascun blocco di 6 transizioni di M', che dipende *solo* dalla funzione di transizione di M e non dal tipo di mosse fatte e men che meno dalla taglia del dato di ingresso. Allora M' farà $|x| + 2 + 6 \times \left\lceil \frac{f(|x|)}{m} \right\rceil$ passi e la traccia della dimostrazione si conclude scegliendo m in modo tale che $m = \lceil \frac{6}{\epsilon} \rceil$.

□

Prima di introdurre una delle classi di complessità più importanti, quella dei problemi decidibili in tempo polinomiale deterministico, usiamo i teoremi precedenti, per fare alcune osservazioni che giustificano ulteriormente la scelta fatta di usare solo ordini di grandezza trascurando le costanti.

Preliminarmente, notiamo che vi sono misure di complessità in tempo che sono sub-lineari, cioè vi sono macchine che richiedono un tempo $f(n) < n$ per risolvere un problema di taglia n. Per esempio, la ricerca di una parola in un dizionario effettuata con un metodo dicotomico porta a leggere $\log n$ parole, certo non tutte quelle contenute nel dizionario stesso (ma tale misura è invariante rispetto al cambiamento di rappresentazione dei dati o si basa proprio su una caratteristica specifica della rappresentazione?). Non considereremo nel seguito misure in tempo sub-lineari, perché per ipotesi vogliamo ottenere la complessità nel caso pessimo, e quindi le macchine leggono sempre l'intero dato di ingresso x, il che richiede appunto $n = |x|$ passi e quindi ogni funzione di complessità in tempo f è tale che $f(n) \geq n$.

Adesso supponiamo di avere una funzione f. Se $f(n) = c \times n$ (cioè f è lineare), allora il teorema di accelerazione ci consente di "rimpicciolire" la costante fino a renderla uguale 1, ponendo $\epsilon = \frac{1}{c}$. Se invece $f(n) = c_1 n^k + c_2 n^{k-1} + \dots + c_k$ (cioè è un polinomio), ancora una volta il teorema ci dice che possiamo rendere c_1 uguale 1 e inoltre gli addendi con esponente minori di k si possono trascurare perché quello di grado massimo li domina per n sufficientemente grande: ecco allora giustificato l'uso di $\mathcal{O}\left(n^k\right)$. Infine, quanto

detto sopra ci porta a concludere che, se I è decidibile polinomialmente, allora esiste un k tale che $I \in \mathcal{O}\left(n^k\right)$. Analoghe considerazioni si possono applicare quando la funzione f considerata maggiori ogni polinomio, per esempio sia una funzione esponenziale.

Quanto osservato sopra basta per introdurre la classe dei problemi decidibili in tempo polinomiale deterministico, ovvero di quei problemi per cui esiste una macchina deterministica che li decide in tempo deterministico limitato da un polinomio.

Definizione 2.2.5. La classe dei problemi decidibili (da MdT) in tempo polinomiale deterministico è

$$\mathcal{P} = \bigcup_{k \geq 1} \mathrm{TIME}\left(n^k\right)$$

Il prossimo passo dovrebbe essere quello di dimostrare che la classe $\mathcal{P}$ appena introdotta è invariante rispetto al cambio di modelli. Dopo aver fatto questo, potremmo anche eliminare nella definizione precedente il riferimento alle macchine di Turing, ciò che implicitamente abbiamo già fatto. Tuttavia la mancanza di tempo ci impedisce di affrontare il problema della robustezza delle classi di complessità con la dovuta attenzione. Ci limiteremo allora a ritornare di sfuggita su questo punto più avanti, affermando senza dimostrarlo che si può passare *in tempo polinomiale* da un algoritmo rappresentato in un modello a uno equivalente rappresentato in un altro modello. In altre parole, la classe $\mathcal{P}$ è chiusa rispetto a trasformazioni di modelli, il che ne garantisce la robustezza.

Tuttavia, come accennato in precedenza, bisogna far molta attenzione al modo in cui si misura il tempo necessario a risolvere un problema.

2.2.3 Macchine di Turing I/O

Per studiare la complessità in spazio è conveniente usare un'ulteriore ragionevole e ben motivata variante di macchine di Turing, quelle che hanno un nastro dedicato a contenere il dato di ingresso, che sarà di sola lettura, uno destinato a memorizzare il risultato, che sarà di sola scrittura, e $k-2$ nastri di lavoro, gli unici rilevanti ai fini della complessità.

Definizione 2.2.6. Una MdT con k nastri $M = (Q, \Sigma, \delta, q_0)$ è di tipo I/O se e solamente se la funzione di transizione δ è tale che, tutte le volte che $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', (\sigma'_1, D_1), \ldots, (\sigma'_k, D_k))$

- $\sigma'_1 = \sigma_1$ — quindi il primo nastro è a sola lettura;

- o $D_k = R$ o, quando $D_k = -, \sigma'_k = \sigma_k$ — quindi il k-esimo nastro è a sola scrittura;
- se $\sigma_1 = \#$ allora $D_1 \in \{L, -\}$ — la macchina visita al massimo una cella bianca a destra del dato di ingresso (che ipotizziamo non contenere # al suo interno, o che abbia una marca di fine stringa, v. dopo).

Si noti che nulla cambia nella definizione di funzione calcolata, né rispetto alle macchine di Turing usate nella prima parte, né rispetto quelle con k nastri. Inoltre, le relazioni delle macchine con k nastri di tipo I/O con quelle non di tipo I/O sono facili da stabilire.

Proprietà 2.2.1. *Per ogni MdT a k nastri M che decide I in tempo deterministico f, esiste una MdT a $k+2$ nastri M' di tipo I/O che decide I in tempo deterministico $c \times f$, per qualche costante c.*

Dimostrazione. La macchina M' copia il primo nastro di M sul proprio secondo nastro, impiegando $|x| + 1$ passi; opera come M senza più toccare il proprio primo nastro; e quando M si arresta, M' si arresta dopo aver copiato il risultato sul proprio $k+2$-esimo nastro, in al più $f(|x|)$ passi. In totale, la macchina M' ha richiesto su x un numero di passi inferiore o uguale a $2 \times f(|x|) + |x| + 1$. Determinare la costante c è ora immediato. □

Infine, per stabilire l'equivalenza tra le MdT a k nastri di tipo I/O o le MdT usate nella prima parte, basta ricorrere alla simulazione vista nel teorema 2.2.1. Quindi, ancora una volta le macchine di Turing si dimostrano estremamente robuste: modifiche algoritmicamente "ragionevoli" non ne alterano il potere espressivo.

2.2.4 Complessità in spazio deterministico

Al fine di avere una nozione di misura di spazio sensata, modifichiamo la definizione di configurazione in modo da ricordare *tutte* le celle visitate, incluse quelle che erano o sono diventate bianche. A esser pignoli, questa modifica richiederebbe l'introduzione di un nuovo simbolo ausiliario, per esempio $\triangleleft \notin \Sigma$, da usare su ciascun nastro come delimitatore destro della parte scritta, e una semplice modifica alla funzione di transizione perché ne tenga conto e lo sposti, *ma solo a destra*, quando necessario. Per esempio, prendiamo la macchina "parallela" per decidere se una stringa è palindroma ed estendiamola con un nastro di ingresso e uno di uscita. Alcuni passi della computazione su *aba* verranno allora rappresentati così, dove q_i, q_j, q_k e q_h sono stati opportuni:

$$(q_0, \trianglerighteq aba \triangleleft, \triangleright \triangleleft) \to^* (q_i, \triangleright aba \underline{\trianglelefteq}, \triangleright aba \underline{\trianglelefteq}) \to^*$$

$$(q_j, \underline{\triangleright}aba\triangleleft, \triangleright aba\underline{\triangleleft}) \to (q_k, \triangleright\underline{a}ba\triangleleft, \triangleright ab\underline{a}\triangleleft) \to (q_h, \triangleright a\underline{b}a\triangleleft, \triangleright a\underline{b}\#\triangleleft)$$

Ci sentiamo liberi di non apportare queste modifiche e di immaginare che, una volta toccata, una casella del nastro apparirà sempre nella rappresentazione del nastro. Quindi in una configurazione $(q, u_1\sigma_1 v_1, \ldots, u_k\sigma_k v_k)$, u_i comincia per $\triangleright$ e v_i può finire con (molti) #, tutti quelli su cui l'i-mo cursore è venuto a posizionarsi durante il calcolo. In questo modo il numero delle caselle in uso nei nastri non diminuisce mai, né

- nel nastro di ingresso, il primo, perché è di sola lettura;
- nel nastro di uscita, il k-esimo, perché è di sola scrittura;
- nei nastri di lavoro $1 < i < k$, perché i caratteri bianchi a destra non scompaiono mai.

Adesso siamo pronti per definire lo spazio necessario a una computazione come il numero totale delle caselle toccate solamente sui nastri di lavoro. Come per il tempo, anche qui si parla di spazio deterministico perché usiamo macchine di Turing deterministiche.

Definizione 2.2.7. Sia M una MdT a k nastri di tipo I/O tale per cui $\forall x$

$$(q_0, \underline{\triangleright}x, \underline{\triangleright}, \ldots, \underline{\triangleright}) \to^* (H, w_1, w_2, \ldots, w_k) \text{ con } H \in \{\textit{SI, NO}\}$$

Lo *spazio richiesto* da M per decidere x è

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

Inoltre M *decide* I *in spazio deterministico* $f(n)$ se $\forall x$ lo spazio richiesto da M per decidere x è minore o uguale a $f(x)$. Infine, se M decide I in spazio deterministico $f(n)$, allora $I \in \text{SPACE}(f(n))$.

Ovviamente, la definizione appena vista può essere immediatamente adattata nel caso generale, in cui si consideri anche lo stato di arresto h come elemento dell'insieme H.

Alcuni autori preferiscono definire lo spazio richiesto come

$$\max_{2 \le i \le k-1} |w_i|$$

ma la sola differenza con la nostra definizione è un fattore k, il quale viene trascurato quando si considerano solamente ordini di grandezza.

La ragione principale per cui si trascura lo spazio necessario a contenere i dati di ingresso e quelli di uscita è che si vogliono misure abbastanza fini per la complessità in spazio. Infatti, se uno considerasse sempre anche la dimensione dei dati di ingresso, cioè la sommatoria di sopra partisse da 1 anziché da 2, si avrebbero sempre complessità almeno lineari. Ciò perché la misura del primo nastro è proprio $|x| + 1$, in quanto tale nastro è di sola lettura e contiene la stringa $w_1 = \triangleright x$. La dimensione del nastro d'uscita non è rilevante nel caso di problemi di decisione I considerati: il risultato è semplicemente un segnale che il caso $x \in I$ è risolto positivamente o meno. Inoltre, a differenza di quanto accade per le classi di complessità in tempo, ci sono delle classi interessanti e importanti che sono sub-lineari e che rivestono un ruolo rilevante nella trattazione successiva, per esempio, LOGSPACE(n), la classe dei problemi decisi in spazio deterministico logaritmico che definiamo più precisamente qui sotto (la quale potrebbe coincidere con $\mathcal{P}$: si veda il frammento di gerarchia introdotto a pagina 61). In questo caso, sommare anche lo spazio per il dato di ingresso porterebbe a trascurare l'addendo $\log n$ che cresce meno di n e a schiacciare così LOGSPACE(n) su PSPACE(n).

Passiamo ora a vedere che anche lo spazio è suscettibile di essere ridotto linearmente, come già visto per il tempo. Si noti che un teorema analogo a quello di riduzione dei nastri sarebbe banale: avremmo ancora la stessa misura, dopo aver ricopiato fianco a fianco i k nastri di lavoro su un solo nastro.

Teorema 2.2.3 (Compressione Lineare in Spazio)**.**
Se $I \in \text{SPACE}(f(n))$ *allora* $\forall \epsilon \in \mathbb{R}^+.\ I \in \text{SPACE}(2 + \epsilon \times f(n))$

Come fatto precedentemente per $\mathcal{P}$, i problemi decidibili in tempo polinomiale deterministico, introduciamo ora la classe dei problemi decidibili in spazio deterministico polinomiale; poi, come promesso, definiremo quella dei problemi decidibili in spazio deterministico logaritmico, che giocheranno un ruolo importante nel capitolo 2.5.

Definizione 2.2.8. La classe dei problemi decidibili (da MdT) in spazio polinomiale deterministico è

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

La classe dei problemi decidibili (da MdT) in spazio logaritmico deterministico è

$$\text{LOGSPACE} = \bigcup_{k \geq 1} \text{SPACE}(k \times \log n)$$

Per fortuna anche queste classi sono invarianti rispetto al cambio di modelli, e quindi come già fatto per $\mathcal{P}$ possiamo eliminare nella definizione precedente il riferimento alle macchine di Turing. In altre parole, le classi PSPACE e LOGSPACE sono chiuse rispetto a trasformazioni di modelli, il che garantisce la loro robustezza, oltre a quella della teoria che stiamo passando in rassegna.

Concludiamo questo capitolo stabilendo alcune relazioni tra le classi di complessità appena introdotte e facendo un'ulteriore osservazione su come tempo e spazio siano correlati. Iniziamo enunciando senza dimostrazione il teorema che stabilisce la relazione di stretta contenenza tra le due classi in spazio appena viste:

Teorema 2.2.4. LOGSPACE $\subsetneq$ PSPACE

Inoltre, confrontiamo LOGSPACE con $\mathcal{P}$, stabilendo un altro piccolissimo frammento della gerarchia che intercorre tra classi di complessità in spazio e in tempo; ulteriori risultati si trovano nel capitolo 2.4.

Teorema 2.2.5. LOGSPACE $\subseteq \mathcal{P}$

Dimostrazione. Poiché il problema I appartiene a LOGSPACE, c'è una macchina di Turing M che decide ogni sua istanza $x \in I$ in $\mathcal{O}(\log |x|)$ spazio deterministico, basta notare che M può attraversare al massimo $\mathcal{O}(|x| \times \log |x| \times \#Q \times \#\Sigma^{\log |x|})$ configurazioni non terminali diverse. Una computazione non può ripassare su una stessa configurazione, altrimenti va in ciclo, quindi una computazione ha al massimo $\mathcal{O}\left(|x|^k\right)$ passi per qualche k.

□

Infine, dalla dimostrazione di sopra, si può vedere che, seppure in modo meno preciso, vale anche il "duale" di quanto affermato a pagina 69, e cioè che, nel caso di algoritmi per problemi decidibili

Osservazione 2: lo spazio limita il tempo di calcolo!

2.3 Misure di complessità non deterministiche

C'è un modo per risolvere i problemi che è un po' ottuso, ma funziona sempre benissimo, complessità a parte, soprattutto quando uno non conosca o non sappia definire l'algoritmo giusto. Prendiamo come semplicissimo esempio il problema di calcolare il massimo numero naturale che sia minore della radice quadrata di 29 senza sapere né come fare né avere a disposizione la tabellina pitagorica: basterà allora cominciare a moltiplicare tra loro tutti i numeri tra 1 e 6 per scoprire che il risultato è 5 (se lo volete vedere come un problema di decisione, chiedetevi se $n \in \{\lfloor\sqrt{29}\rfloor\}$). In alternativa, si può tirare un dado, calcolare il quadrato del numero uscito e controllare se è una soluzione, magari ripetendo il lancio (avendo per magia rimosso il numero appena uscito dalle facce del dado) e confrontando l'ultimo numero uscito con i precedenti e con 29 — basta quindi che esista un lancio "fortunato" e la soluzione è trovata. Quest'ultima modalità (e anche l'altra!) origina un albero di scelte i cui livelli rappresentano i lanci e nei cui nodi immaginiamo di aver scritto il numero uscito — basta che esista un cammino nell'albero che porta a una soluzione.

Questo metodo di soluzione, seppure ottuso, non è così assurdo come può apparire a prima vista e ritorneremo più avanti sui problemi che sembra si possano risolvere solo ricorrendovi, perché non si conoscono o non si sanno sfruttare proprietà matematiche, strutturali del problema. In entrambi i casi delineati nell'esempio, vi è un procedimento di tipo *non deterministico* — l'unica proprietà matematica usata per giustificare l'impiego di un dato a sei facce è che $\sqrt{29} \leq 6$. Nel primo metodo, il non determinismo è meno evidente, essendo stato ridotto a una soluzione operazionalmente accettabile; infatti, si generano tutte le scelte, ovvero l'intero spazio di ricerca, e ciascuna di queste viene esaminata, cioè si fa una ricerca esaustiva della soluzione; questo metodo viene anche detto di *forza bruta*. Seguendo il secondo procedimento si prende una potenziale soluzione *a caso* (si ricordi che basta che ne esista una!) e si controlla se essa lo è davvero; in inglese questo metodo si chiama *guess-and-try* — se c'è una soluzione, ci viene fornita per magia. Si noti che questo metodo non richiede né di generare né di visitare l'intero spazio di ricerca se non *implicitamente*, come sarà forse più chiaro in seguito. Inoltre, l'albero delle possibili scelte è sempre finito, perché stiamo considerando solo problemi decidibili. Quindi, una soluzione appare sempre a profondità finita o, sempre a profondità finita, *tutti* i rami dell'albero di scelta finiscono su nodi che riportano fallimento (non è il caso nel nostro esempio, in cui tutti i rami terminano con successo al massimo dopo sei lanci del dado, cioè a profondità

sei nell'albero; non è difficile immaginare casi in cui vi siano situazioni di successo e situazioni di fallimento e ne vedremo in seguito). Si noti infine che un albero delle possibili scelte, per brevità *albero non deterministico delle computazioni* o semplicemente *albero non deterministico*, può sempre essere visitato in modo deterministico, livello per livello.

Per formalizzare le intuizioni descritte sopra, è opportuno introdurre una variante delle macchine di Turing, dette *non deterministiche*. Essenzialmente, una macchina non deterministica differisce da una deterministica per il fatto che la relazione di transizione δ non è necessariamente una funzione, cioè una configurazione può evolvere in più di una configurazione successiva, originando per così dire un albero (di computazioni) non deterministico (sia ben chiaro però che una computazione continua a essere una *singola successione* di configurazioni). L'osservazione fatta prima, che tale albero può esser visitato per livelli, ci assicura che introdurre le macchine non deterministiche non cambia affatto la classe dei problemi decidibili, né alcuno dei risultati di calcolabilità presentati nella prima parte del corso.

Con la stessa osservazione ci si può facilmente convincere che le macchine di Turing deterministiche simulano quelle non deterministiche con una perdita di efficienza *esponenziale* (si ricordi che i nodi di un albero sono in numero esponenziale rispetto la profondità dell'albero stesso); per una formulazione esatta, si veda il teorema 2.3.1. Quanto detto giustifica, almeno in parte, la tesi di Cook-Karp, cioè che la classe dei problemi decidibili in tempo polinomiale *deterministico*, $\mathcal{P}$, è formata dai problemi *facili*, mentre la classe dei problemi decidibili in tempo polinomiale *non deterministico*, $\mathcal{NP}$, che definiremo precisamente in seguito, è quella dei problemi *difficili*. Torneremo più avanti su questa distinzione, analizzando più in dettaglio $\mathcal{P}$ e $\mathcal{NP}$.

Però a questo punto non possiamo non chiederci se l'introduzione del non-determinismo dia davvero un potere maggiore alle MdT dal punto di vista della complessità del calcolo, ovvero quanto sia fondata la tesi di Cook-Karp. Abbiamo già toccato questo argomento introducendo un frammento di una gerarchia di complessità, concludendo col dire che è ancora irrisolto il famoso problema $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, ovvero se non vi sia differenza tra il tempo polinomiale non deterministico e quello deterministico, nel qual caso aver introdotto il non determinismo non separerebbe i problemi "difficili" da quelli "facili". Per quanto riguarda lo spazio, preannunciamo che la classe dei problemi che richiedono spazio polinomiale *deterministico* PSPACE coincide con NPSPACE, quella dei problemi risolubili in spazio polinomiale *non deterministico*, ancora da definire. Se valesse anche l'eguaglianza $\mathcal{P} = \mathcal{NP}$, il meccanismo del non determinismo, che sembra avere scarso significato computazionale, almeno dal punto di vista della sua concreta realizzabilità, sarebbe completamente

irrilevante anche dal punto di vista della complessità.

2.3.1 Macchine di Turing non deterministiche

Introduciamo di seguito l'estensione non deterministica alle macchine di Turing come definite in 1.2.1, dando esplicitamente conto solo delle cose che cambiano. Come detto sopra, una macchina non deterministica differisce da una deterministica per il fatto che la relazione di transizione δ non è necessariamente una funzione. A differenza di altre estensioni viste, questo nuovo modello non appare però sufficientemente "realistico". Infatti, le estensioni viste nel capitolo precedente si basano su meccanismi che hanno un'interpretazione computazionale immediata; ad esempio, le macchine con molti nastri sono una semplificazione accettabile dei calcolatori paralleli. Invece, non si conoscono, almeno per ora, né ci riesce di immaginare, allo stato presente della tecnologia, macchine che siano davvero non deterministiche. [5]

L'importanza di queste macchine però, sia come le presentiamo qui, e a maggior ragione in versioni assai più elaborate studiate in letteratura, è rilevantissima per organizzare in modo adeguato una teoria quantitativa degli algoritmi, per cui, pur consci della loro astrattezza e apparente (?) irrealizzabilità, non esitiamo a usarle.

Definizione 2.3.1. Una MdT *non deterministica* (a k nastri, di tipo I/O) è una quadrupla $N = (Q, \Sigma, \Delta, q_0)$ dove

[5] Tra le differenti varianti della macchina di Turing, introdotte allo scopo di confermare o confutare la tesi di Church-Turing, vanno citati alcuni modelli di computazione introdotti abbastanza recentemente e ispirati a meccanismi fisici o biologici.

Più precisamente, in alcuni articoli di Deutsch, Feynman ed altri autori, apparsi verso la metà degli anni ottanta, viene esaminata criticamente l'adeguatezza del sistema fisico (computer o agente umano) alla base del modello di macchina di Turing. Senza entrare nei dettagli, appare chiaro che tale sistema fisico obbedisce alle leggi della meccanica classica laddove la comunità fisica è abbastanza concorde che il comportamento di un sistema fisico vada descritto sulla base delle leggi della fisica quantistica [magari facendo riferimento alla teoria delle stringhe]. Una macchina che modella un tale tipo di sistema (macchina di Turing quantistica) è stato formalizzato in un articolo di Deutsch. Meccanismi computazionali basati su fenomeni biologici hanno invece conosciuto un particolare sviluppo soprattutto dopo un esperimento effettuato da Adleman e che consiste essenzialmente nell'utilizzare reazioni biologiche su molecole di DNA per "codificare" un algoritmo per la soluzione del problema del cammino hamiltoniano in un grafo. C'è da dire che né tali meccanismi, né la macchina di Turing quantistica ampliano le capacità espressive delle macchine di Turing: la classe delle funzioni calcolate è quella delle funzioni T- [o μ- o WHILE-] calcolabili. Dato però il parallelismo estremo di tali dispositivi, quello che esse alterano è piuttosto la classificazione di alcuni problemi nelle varie classi di complessità; ad esempio, l'algoritmo quantistico di Shor che, con complessità di tempo polinomiale, fattorizza un intero in numeri primi.

- Q, Σ, q_0 sono come nella definizione 1.2.1 (nella 2.2.1 delle macchine con k nastri, nella 2.2.6 di quelle I/O);
- $\Delta \subseteq (Q \times \Sigma) \times ((Q \cup \{SI, NO\}) \times \Sigma \times \{L, R, -\})$ è la *relazione* di transizione (estesa nel modo ovvio nel caso delle macchine a k nastri e di tipo I/O).

Le configurazioni non vengono affatto modificate: esse hanno la stessa forma di quelle già viste: $(q, w\sigma u)$. Allo stesso modo non cambia il passo di computazione $(q, w\sigma v) \to_N (q', w'\sigma'v')$, e quindi le computazioni sono anche qui una *successione* (*non* un albero!) di configurazioni; infine continueremo a usare gli apici n e $*$ per indicare computazioni di lunghezza n o qualunque.

A rimarcare che nelle macchine di Turing non deterministiche si usa una relazione piuttosto che una funzione, abbiamo usato la lettera maiuscola Δ al posto di quella minuscola δ. Poiché la relazione di transizione Δ può contenere più quintuple associate allo stesso stato e allo stesso simbolo, ci possono essere molte configurazioni $(q', w'\sigma'u')$ che sono raggiungibili da $(q, w\sigma u)$ in un *solo* passo. Dovrebbe essere adesso più chiaro che le computazioni possono venir organizzate in un albero non deterministico del genere visto sopra. Inoltre, la vera potenza del non determinismo appare nel modo in cui si accetta (o se preferite si calcola, il che richiede una semplice e ovvia estensione alla definizione seguente).

Definizione 2.3.2. La macchina non deterministica N (a k nastri) decide I tutte e sole le volte che

> $x \in I$ se e solamente se
>
> esiste una computazione tale che $(q_0, \trianglerighteq x, \triangleright, \ldots, \triangleright) \to_N^* (SI, w_1, \ldots, w_k)$

Per accettare una stringa di ingresso, *basta che esista* una computazione che porti a una configurazione il cui stato sia SI – ecco il pizzico di magia: basta che ci sia una configurazione che accetta e non è affatto rilevante che ci siano altre computazioni che rifiutano, finendo in configurazioni con stato NO (o nel caso generale che non terminano). [6] Si noti come questa definizione di accettazione introduca un'asimmetria rispetto a quella di non accettazione. In quest'ultimo caso, infatti, per rifiutare una stringa di ingresso bisogna che *tutte* le computazioni della macchina portino a configurazioni con stato NO o siano non terminanti. La situazione di non terminazione verrà esclusa

[6] Per questa ragione il non determinismo che abbiamo introdotto si chiama anche angelico; la versione demoniaca prevede che tutte le computazioni raggiungano uno stato di successo.

nel seguito, perché parliamo solo di problemi decidibili; per avere un'idea di come ci possano essere computazioni non terminanti, si consideri di nuovo il banale esempio fatto all'inizio del capitolo, e si cerchi di determinare che 5 è una soluzione lanciando ripetutamente il dado da cui però nessun mago ha sottratto le facce che portano i numeri già usciti: vi saranno allora sequenze di lanci in cui esce sempre lo stesso numero. Si noti tuttavia che, anche senza magia, le soluzioni si trovano *sempre* a profondità finita. Si noti anche che una computazione è *completamente determinata* da una sequenza di scelte tra le varie configurazioni raggiungibili passo dopo passo.

2.3.2 Complessità in tempo e in spazio non deterministici

Introduciamo ora i corrispettivi non deterministici delle classi di complessità definite nel precedente capitolo, i cui nomi inizieranno tutti con la lettera N, per non determinismo. Non si dimentichi che nel seguito considereremo *solo* problemi decidibili, quindi le macchine che useremo terminano per ogni ingresso; ciò implica che se una di queste macchine N decide I, non solo si ha che $\forall x \in I$ esistono $w_1, w_2, \ldots, w_k$ tali che $N(x) = (q_0, \trianglerighteq x, \triangleright, \ldots, \triangleright) \rightarrow_N^* (SI, w_1, w_2, \ldots, w_k)$, ma anche che $\forall x \notin I$, $\forall w_1, w_2, \ldots, w_k$ tali che $N(x) \rightarrow_N^* (q, w_1, w_2, \ldots, w_k) \not\rightarrow_N$ si ha che $q = NO$, cioè si raggiunge lo stato di rifiuto.

Definizione 2.3.3. La macchina non deterministica N *decide* I *in tempo non deterministico* $f(n)$ se e solamente se

- N decide I e
- $\forall x \in I$, $\exists t$ tale che $(q_0, \trianglerighteq x, \triangleright, \ldots, \triangleright) \rightarrow_N^t (SI, w_1, \ldots, w_k)$, $t \leq f(|x|)$.

Intuitivamente, una MdT non deterministica N accetta x in tempo non deterministico $f(|x|)$ se e solo se c'è *almeno una* computazione che termina in uno stato SI in t passi, $t \leq f(|x|)$; di fatto, occorre e basta che la più breve delle computazioni accettanti ci metta meno passi di $f(|x|)$, o altrettanti. Invece, N non accetta x se *tutte* le sue computazioni lunghe al massimo $f(|x|)$ conducono allo stato NO. Ecco di nuovo l'asimmetria: non basta che per un elemento $x \notin I$ ci sia una computazione che porta allo stato NO in meno di $f(|x|)+1$ passi, ma si richiede che lo debbano fare *tutte* in meno di $f(|x|)+1$ passi. Quindi l'asimmetria tra il decidere I e il decidere $\bar{I}$ che abbiamo visto nel caso generale viene rafforzata dal richiedere che ciò venga svolto in tempo $f(|x|)$. Ripetiamo: affinché N decida il problema I basta che per *ogni* suo elemento x ci sia *una* computazione che lo accetta. Invece, affinché N decida $\bar{I}$ bisogna che *tutte* le computazioni sul dato di ingresso $x \notin I$ conducano allo stato NO in meno di $f(|x|) + 1$ passi.

Come fatto per TIME($f(n)$) possiamo ora definire la classe dei problemi decidibili da MdT (e da altri modelli "ragionevolmente" equivalenti) in tempo *non deterministico* $f(n)$.

Definizione 2.3.4. Se una macchina di Turing non deterministica decide il problema I in tempo f, allora $I \in \text{NTIME}(f)$

Adesso possiamo finalmente introdurre la classe dei problemi decidibili in tempo polinomiale non determistico.

Definizione 2.3.5. La classe dei problemi decidibili in tempo non deterministico polinomiale è

$$\mathcal{NP} = \bigcup_{k \geq 1} \text{NTIME}\left(n^k\right)$$

Ovviamente $\mathcal{P} \subseteq \mathcal{NP}$, perché una macchina deterministica M è anche non deterministica. Sarebbe bello se fosse vero anche il contrario, cioè $\mathcal{P} \supseteq \mathcal{NP}$, da cui si dedurrebbe che $\mathcal{P} = \mathcal{NP}$. Se così fosse, ci sarebbe un modo per trasformare un algoritmo non deterministico polinomiale in uno deterministico polinomiale, quindi "facile e fattibile".

Come abbiamo già più volte detto, quello che si sa fare per ora è di simulare una macchina non deterministica N che decide un problema in tempo polinomiale con una macchina deterministica M con una perdita di efficienza in tempo *esponenziale*. L'idea è di generare prima le potenziali soluzioni e poi verificare se lo sono davvero; vedremo anche che il primo passo è difficile, nel senso che (solitamente) richiede un numero di passi esponenziale, mentre il secondo è facile e può essere eseguito in tempo deterministico polinomiale.

Teorema 2.3.1. *Se I è deciso in tempo non deterministico $f(n)$ dalla macchina non deterministica N (a k nastri), allora, con una perdita esponenziale, è deciso in tempo $\mathcal{O}\left(c^{f(n)}\right)$ da una macchina deterministica M (a $k+1$ nastri), con $c > 1$ dipendente solo da N. Più precisamente:*

$$\text{NTIME}\left(f(n)\right) \subseteq \text{TIME}\left(c^{f(n)}\right)$$

Dimostrazione. Sia d il grado di non-determinismo di N, cioè poniamo

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

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

(Per semplicità di trattazione, supponiamo nel seguito che la macchina abbia sempre d scelte; non è difficile modificare quanto segue al caso generale).

Per ogni stato $q \in Q$ e ogni simbolo $\sigma \in \Sigma$, ordiniamo totalmente, p.e. lessicograficamente, l'insieme $\Delta(q, \sigma)$. Ogni computazione di N è una sequenza di scelte; se tale sequenza è lunga t, la possiamo vedere come una successione di numeri naturali minori di t nell'intervallo $[0 \dots d-1]$, rappresentando con 0 la prima scelta. La macchina M considera queste successioni una alla volta, in ordine crescente (ovvero visita l'albero per livelli) e per ciascuna di esse simula N. (Nota bene: la costruzione di M deve essere indipendente da f e quindi deve esser fatta a prescindere dal valore corrente di $f(n)$ — se così non fosse, basterebbe generare tutte le computazioni lunghe al più $f(|x|)$).

La macchina M riproduce la successione di scelte $(c_1, \dots, c_t)$, tenendo l'ultima successione sul nastro aggiuntivo, ovvero vi mantiene il numero t' in base d che gli corrisponde. Se durante questa simulazione M arriva in uno stato *SI* allora M termina accettando, altrimenti genera la prossima successione, usando come guida il prossimo numero in base d, cioè $t' + 1$. Se tutte le successioni terminano (ovvero è stato generato il numero t) portando allo stato *NO*, allora M termina rifiutando. È chiaro che M termina con successo se e solamente se N fa altrettanto.

Rimane da verificare che il tempo impiegato da M nella simulazione è quello dell'enunciato. Quante sono le successioni da visitare? al massimo $\mathcal{O}\left(d^{f(n)+1}\right)$, quindi il teorema è dimostrato ponendo $c = d$.

□

Terminiamo con la definizione di spazio non deterministico e della classe di problemi decidibili in spazio non deterministico polinomiale. Si noti il quantificatore esistenziale nel secondo punto.

Definizione 2.3.6. La macchina di Turing N, non deterministica a k-nastri di tipo I/O, *decide* I *in spazio non deterministico* $f(n)$ se e solamente se

- N decide I
- $\forall x \in I \ \exists w_1, \dots, w_k$ tali che $(q_0, \trianglerighteq x, \dots, \trianglerighteq) \to_N^* (SI, w_1, w_2, \dots, w_k)$ e $\sum_{2 \leq i \leq k-1} |w_i| \leq f(n)$

Se N decide I in spazio non deterministico $f(n)$, allora $I \in \mathrm{NSPACE}(f(n))$. Infine, la classe dei problemi decidibili (da MdT) in spazio non deterministico polinomiale è

$$\mathrm{NPSPACE} = \bigcup_{k \geq 1} \mathrm{NSPACE}\left(n^k\right)$$

Abbiamo già preannunciato che l'estensione con il non determinismo non allarga la classe dei problemi trattabili polinomialmente, quando la misura della complessità riguardi lo spazio. Enunciamo ora tale teorema, senza dimostrarlo.

Teorema 2.3.2 (Savitch)**.** NPSPACE = PSPACE.

Nonostante nella soluzione di un problema I ottenuta per forza bruta o procedendo per tentativi non si sfrutti affatto la struttura matematica di I, forse perché essa è troppo complessa, o perché essa sfugge al solutore del problema, vi sono numerosi esempi di problemi per cui quello è l'unico modo conosciuto di arrivare a una soluzione; ne menzioneremo alcuni più avanti e vi ritorneremo ancora nel prossimo capitolo. Per ora limitiamoci a discutere brevemente un esempio paradigmatico per la soluzione del quale si conoscono solo algoritmi polinomiali non deterministici o esponenziali deterministici: il problema del commesso viaggiatore. [7]

Esempio 2.3.1 (Problema del commesso viaggiatore)**.** Vi sono n città, ciascuna individuata da un intero e collegata a tutte le altre da una strada percorribile nei due sensi; sia allora $d(i, j)$ la distanza tra le città i e j. [8] Il problema consiste nel trovare il cammino di costo minimo che attraversa tutte le città una e una volta sola — ovvero dobbiamo trovare una permutazione di indici (o città) $\Pi : [1 \dots n] \to [1 \dots n]$ che minimizza la seguente quantità, in cui intendiamo $i = \Pi(h), i + 1 = \Pi(k)$ per qualche h e k:

$$\sum_{1 \leq i \leq n-1} d(i, i + 1)$$

Ovviamente, il problema si può rappresentare come un grafo (non diretto) i cui nodi sono le città e in cui c'è un arco tra ogni coppia di nodi i e j, pesato da $d(i, j)$.

Per vedere questo problema come un problema di decisione si abbia un valore limite B, da interpretarsi come il rimborso viaggi assegnato al commesso viaggiatore; allora il problema è risolto positivamente se c'è un cammino che tocca tutte le città una e una sola volta di costo minore o uguale a B.

Vediamo adesso di calcolare, in modo assai spiccio, la complessità prima di una (tipica) procedura che impiega il metodo di forza bruta per risolvere il problema del commesso viaggiatore e poi di una (tipica) MdT non deterministica che risolve la sua variante di decisione; entrambe procedono in due fasi

[7] Stiamo parlando di soluzione esatta; nel caso in cui ci si accontenti di una soluzione approssimata vi sono algoritmi molto più furbi, sulla cui natura ritorneremo brevemente più avanti.

[8] Una distanza d è una funzione da coppie di punti, nel nostro caso città, nei reali positivi, tale che gode delle seguenti tre proprietà
- riflessiva: $d(i, j) = 0$ se e solamente se $i = j$;
- simmetrica: $d(i, j) = d(j, i)$;
- triangolare: $d(i, j) \leq d(i, k) + d(k, j)$.

separate. Il dato iniziale delle due macchine è ovviamente la rappresentazione della rete stradale e del costo B (per esempio come matrice di incidenza determinata dalle distanze).

Come tutte le procedure a forza bruta, il primo metodo ha la seguente struttura:

i) genera tutte le potenziali soluzioni come permutazioni di tutti gli interi fino a n il che costa $\frac{(n-1)!}{2}$

ii) scegli tra tutte le permutazioni la prima che ha costo accettabile, ovvero minore di B, e questo controllo può essere fatto in tempo deterministico cubico (bisogna accedere $\mathcal{O}(n)$ volte alle $\mathcal{O}(n^2)$ coppie (i, j) per ottenere la distanza $d(i, j)$ memorizzata nel nastro di ingresso).

(Può essere interessante vedere che lo spazio necessario alla procedura sopra delineata è in $\mathcal{O}(n)$, perché è sufficiente generare una permutazione alla volta e memorizzare in uno spazio di lavoro la permutazione in esame.)

Costruiamo adesso una macchina di Turing non deterministica N che decide il problema in $\mathcal{O}(n^3)$. N procede con le due fasi seguenti:

i) N scrive su un nastro di lavoro una stringa di n numeri naturali compresi tra 1 e n; cioè vi sono n transizioni possibili a partire dalla configurazione iniziale, ciascuna che scrive un numero nell'intervallo $[1 \dots n]$ sul nastro di lavoro e dalle configurazioni così raggiunte vi sono n transizioni, che scrivono un numero in $[1 \dots n]$, e così via — ovviamente la scelta della transizione da effettuare viene fatta a caso (naturalmente la macchina potrebbe esser più furba). Questa fase richiede $\mathcal{O}(n)$ passi;

ii) N verifica se la stringa è un cammino accettabile, cioè è una permutazione degli indici, usando un altro nastro di lavoro, in tempo $\mathcal{O}(n^2)$; N verifica di seguito (o contemporaneamente) se il costo del cammino è minore o uguale a B, di nuovo in tempo $\mathcal{O}(n^3)$, nel qual caso risponde positivamente; altrimenti risponde negativamente.

Si noti come la prima fase che compiono entrambe le procedure consiste nel generare una delle soluzioni, le quali sono in numero esponenziale; la differenza è che nel primo algoritmo si procede costruttivamente, generandole tutte, mentre nel secondo si tira a indovinare e si sfrutta il meccanismo non deterministico della macchina usata. In altre parole, in entrambi i casi si genera esaustivamente tutto lo spazio del problema su cui fare poi la ricerca della soluzione, ma nel primo modo ciò avviene *esplicitamente*, nel secondo *implicitamente*. È importante osservare che la seconda fase è una *certificazione in tempo polinomiale* fatta in modo *deterministico* che la stringa di naturali

sia davvero una soluzione. Uno potrebbe, e spesso viene fatto, definire allora la classe $\mathcal{NP}$ come l'insieme dei problemi che ammettono una certificazione in tempo polinomiale. Tuttavia, dovrebbe a questo punto essere chiaro che i due modi per definire tale classe, o via MdT non deterministiche o via certificazione polinomiale, sono del tutto equivalenti e differiscono solo dal punto di vista con cui si esaminano i problemi che vi appartengono. Infine, notiamo che, quando il cammino in esame non è una soluzione, il metodo esaustivo esplicito ci dice che dobbiamo visitare tutto lo spazio degli stati, che sono in numero esponenziale, quindi la certificazione del fallimento avviene in tempo esponenziale: *tutti* i tentativi falliscono.

Come già detto più volte, questo modo di procedere trova applicazioni in molti campi. Ad esempio, in logica per dimostrare che una formula, decidibile e lunga n, è un teorema, uno può generare "tutte le dimostrazioni" le cui ultime formule sono lunghe n e se ci trova la formula di partenza, allora questa è un teorema. Oppure, per decidere se c'è un assegnamento di valori di verità alle variabili di una formula che la rende vera, si considerano tutti i possibili assegnamenti e poi si calcola il valore della formula in ciascun caso. Strettamente correlato a quest'ultimo problema, c'è, in teoria dei circuiti, il problema di decidere quando in un dato circuito passa un segnale. Nell'area dell'ottimizzazione combinatoria, oltre al problema del commesso viaggiatore e di quelli ad esso correlati, si risolvono in modo esatto con questa tecnica i problemi di assegnazione di risorse, tra i quali l'allocazione dei task (non prerilasciabili) ai processori, compito fondamentale dei sistemi operativi. Infine, nel campo dell'intelligenza artificiale, ci sono tutti i problemi legati ai giochi o alla ricerca e all'estrazione di informazione da grosse collezioni di dati, anche non strutturate.

2.4 Funzioni di misura, un po' di gerarchia e due assiomi

Prima di studiare un po' più in dettaglio le classi $\mathcal{P}$ e $\mathcal{NP}$ facciamo una rapida carrellata su alcuni risultati di complessità, che enunceremo senza dimostrare; discuteremo anche brevemente l'influsso che sulla loro validità hanno la forma e il tipo delle funzioni che limitano le risorse che sono necessarie alle macchine per risolvere un problema, per esempio delle funzioni che stimano il numero di passi e di celle necessari.

In principio, una funzione di misura $f : \mathbb{N} \to \mathbb{N}$ può essere una qualunque funzione totale, anche avere essa stessa complessità enorme. Vedremo subito che funzioni di misura arbitrarie portano a risultati che sono quantomeno controintuitivi, e pertanto considereremo di seguito una classe di funzioni di misura, dette *appropriate*, anche se, a prima vista, possono sembrare particolarmente limitate. Intuitivamente, le funzioni appropriate non richiedono più risorse di quanto stimato dal loro stesso valore, cioè l'algoritmo che calcola $f(x)$ deve richiedere tempo $\mathcal{O}(f(|x|) + |x|)$ (l'addendo $|x|$ interviene quando f è sub-lineare, per rispettare l'ipotesi fatta che le funzioni di complessità in tempo siano almeno lineari) e richiedere spazio $\mathcal{O}(f(|x|))$.

Sotto queste ipotesi possiamo enunciare il teorema che risponde a una delle nostre prime domande: esiste davvero una gerarchia di problemi. In altre parole, aumentando le risorse a disposizione delle nostre macchine si risolvono più problemi — ciò che del resto ci dicono intuizione ed esperienza. Successivamente, vedremo anche che ci sono problemi arbitrariamente complessi, e che quindi non vi è una classe che domini tutte le altre e che di conseguenza la gerarchia non è composta da un numero finito di classi.

In effetti, l'ultimo teorema vale anche nel caso in cui si considerino funzioni di misura qualsiasi, purché totali. Ciò suggerirebbe di allargare la classe delle funzioni appropriate a quella delle funzioni totali: mal ce ne incoglierà! Vi sono un paio di teoremi che distruggono la gerarchia appena stabilita e quindi l'estensione fatta non raggiunge lo scopo che ci eravamo prefissato. Il primo teorema dice che, per funzioni di misura arbitrarie, esistono problemi che non hanno un algoritmo ottimo; il secondo che vi sono delle lacune tra classi di complessità.

Infine, a puro titolo di curiosità, discuteremo di una classe di funzioni di misura della complessità, la cui definizione, sottesa da un'intuizione accettabilissima, è molto elegante, in quanto consta di due assiomi e non richiede alcun riferimento esplicito alle risorse di calcolo che esse richiedono per la soluzione del problema. Purtroppo, valgono anche in questo caso i risultati anti-intuitivi citati sopra, e quindi ci tocca contentarci delle funzioni

appropriate che allora useremo nel seguito, salvo che in questo breve capitolo.

Iniziamo con una buona definizione di funzione per misurare il consumo delle risorse: le funzioni appropriate di cui abbiamo parlato sopra. Anche per esse vi sono molte definizioni leggermente diverse e molti nomi diversi: funzioni *oneste*, *costruibili* (in tempo o in spazio) e altre ancora. Per i nostri scopi è sufficiente la seguente definizione.

Definizione 2.4.1. La funzione $f : \mathbb{N} \to \mathbb{N}$ calcolabile totale, è *appropriata* se

i) è monotona crescente (cioè $n \geq m$ implica $f(n) \geq f(m)$);

ii) esiste una MdT M a k nastri, tale che $\forall x \in \Sigma^*$ si arresta (dando come risultato $\diamond^{f(x)}$, con $\diamond \notin \Sigma$ simbolo speciale) in tempo $\mathcal{O}(f(|x|) + |x|)$ e in spazio $\mathcal{O}(f(|x|))$.

Le condizioni poste a una funzione affinché sia appropriata sono piuttosto onerose e sembrano mordersi la coda. Tuttavia le funzioni che si vedono di solito usare per dare la complessità degli algoritmi sono tutte appropriate. Ad esempio sono funzioni appropriate $k, n, n^k, n!, \lfloor \log(n) \rfloor, \lfloor \sqrt{n} \rfloor$ (si noti che le utlime due sono sub-lineari); inoltre se f, g sono funzioni appropriate, lo sono anche $f + g$ e $f \times g$, quindi tutti i polinomi sono funzioni appropriate; infine anche f^g e $g \circ f$ lo sono, assieme a molte altre funzioni d'uso comune.

Adesso andiamo a stabilire che le classi dei problemi decidibili con risorse fissate, misurate con funzioni appropriate, aumentano al crescere delle risorse stesse. Ciò significa che, al variare di queste, si può stabilire una gerarchia tra le classi di complessità.

Teorema 2.4.1 (Gerarchia di tempo e spazio)**.** *Se f è appropriata:*

- $\mathrm{TIME}(f(n)) \subsetneq \mathrm{TIME}((f(2n+1))^3)$
- $\mathrm{SPACE}(f(n)) \subsetneq \mathrm{SPACE}(f(n) \times \log(f(n)))$

Dimostrazione. Omessa. Notiamo soltanto che quella del primo punto si basa sulla dimostrazione che il problema (ancora la diagonalizzazione!):

$$\{x \mid \varphi_x(x) \textit{ converge entro } f(|x|) \textit{ passi}\}$$

appartiene a $\mathrm{TIME}(f^3)$ ma *non* a $\mathrm{TIME}(f)$; la dimostrazione del secondo punto è analoga. □

Naturalmente, accanto a questo teorema ce n'è uno del tutto analogo che riguarda le misure non deterministiche. (Si noti di sfuggita che se $f(n)$ è un polinomio, lo è anche $f(2n+1)^3$.)

In realtà, la gerarchia è ben più densa, perché l'esponente 3 che compare nella formulazione del teorema precedente è determinato dalla particolare dimostrazione scelta. In realtà, vi sono molte classi significative tra TIME(f^3) e TIME(f); analogamente per lo spazio. Questo frammento di gerarchia ci basta comunque per dimostrare il seguente corollario, in cui si afferma che la classe dei problemi decidibili in tempo polinomiale deterministico è strettamente inclusa in quella dei problemi decidibili in tempo deterministico esponenziale, che abbiamo già incontrato e che adesso introduciamo esplicitamente:

$$\mathrm{EXP} = \bigcup_{k \geq 1} \mathrm{TIME}\left(2^{n^k}\right)$$

Corollario 2.4.1.

$$\mathcal{P} \subsetneq \mathrm{EXP}$$

Dimostrazione. L'inclusione è ovvia perché 2^n cresce più velocemente di ogni polinomio; è propria perché

$$\mathcal{P} \subseteq \mathrm{TIME}(2^n) \subsetneq \mathrm{TIME}\left(\left(2^{(2n+1)}\right)^3\right) \subseteq \mathrm{TIME}\left(2^{n^2}\right)$$

□

Questo corollario, assieme al fatto che $\mathrm{NTIME}(f(n)) \subseteq \mathrm{TIME}\left(c^{f(n)}\right)$ (teorema 2.3.1), ci permette di concludere che $\mathcal{NP}$ è inclusa in EXP (il che non ci dice ancora nulla se l'inclusione $\mathcal{P} \subseteq \mathcal{NP}$ è propria o meno). Riceve allora giustificazione anche il modo di procedere per forza bruta: genera tutti i candidati a essere una soluzione in tempo deterministico esponenziale e poi certifica che un candidato è davvero una soluzione (in tempo deterministico polinomiale).

Per dare un quadro assai sommario, ma un pochino più completo della gerarchia che si può stabilire tra le classi di complessità sia in tempo che in spazio e nelle loro versioni deterministiche e non deterministiche, riportiamo senza dimostrarli i seguenti risultati.

Teorema 2.4.2. *Siano f una funzione di misura appropriata e k una costante, allora*

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

Inoltre, ricordiamo che avevamo già stabilito i seguenti fatti.

Teorema 2.4.3.

- LOGSPACE $\subseteq \mathcal{P}$
- LOGSPACE $\subsetneq$ PSPACE
- PSPACE = NPSPACE
- $\mathcal{NP} \subseteq$ EXP

Come anticipato, vediamo che la gerarchia non è superiormente limitata, ovvero che ci sono problemi arbitrariamente difficili, indipendentemente dagli algoritmi usati per risolverli. In questo teorema, la cui dimostrazione omettiamo, si può rilasciare senza timori la richiesta che le funzioni di misura siano appropriate.

Teorema 2.4.4. *Per ogni funzione calcolabile totale g esiste un problema $I \in \text{TIME}(f(n))$ e $I \notin \text{TIME}(g(n))$ con $f(n) > g(n)$ quasi ovunque.* [9]

Abbiamo già preannunciato che usare funzioni di misura arbitrarie, per esempio funzioni che crescono vertiginosamente, porta a conclusioni bizzarre. Le enunceremo, cercando di dar loro un minimo di intuizione, mediante i due teoremi che seguono e che non dimostreremo. [10]

Il primo risultato riguarda l'esistenza di algoritmi ottimi per risolvere un dato problema, ovvero algoritmi le cui prestazioni non possono essere migliorate che per una costante moltiplicativa. Una parte importantissima dell'informatica si occupa di trovare algoritmi ottimi per un problema, sia per ragioni di efficienza concreta, sia perché in questo modo si ottiene una misura precisa della difficoltà di un problema e dell'efficienza dei programmi che lo risolvono. Il risultato che riportiamo è negativo: non sempre esiste un algoritmo ottimo. [11]

Teorema 2.4.5 (di accelerazione, Blum)**.** *Per ogni funzione calcolabile totale h, esiste un problema I tale che, per ogni algoritmo M che decide I in tempo f esiste M' che decide I in tempo f' tale che*

$$f(n) > h(f'(n)) \text{ quasi ovunque}$$

[9] Cioè tranne che per dati di ingresso tratti da un insieme finito.

[10] Non sorprenderà che le loro dimostrazioni siano basate su un ragionamento di tipo diagonale.

[11] Una visione intuitiva di questo teorema, suggerita da Börger, è che ci sono dei programmi che sono più veloci su una macchina (universale) vecchia e lenta che su una macchina (universale) nuova e veloce.

In altre parole, si può dimostrare l'esistenza di una successione di algoritmi via via più efficienti per risolvere il problema costruito in funzione di h, che è una funzione calcolabile totale, ma del tutto arbitraria. Per esempio fissata la funzione h, il teorema dice che si può trovare un algoritmo, magari esponenzialmente più efficiente di M, un terzo esponenzialmente più efficiente del secondo, e così via; o peggio se la funzione h cresce ancor più velocemente. Si noti che non poniamo alcun vincolo sulla funzione f. ATTENZIONE: data la funzione h, il problema che si costruisce per dimostrare il teorema è "artificiale", nel senso che non corrisponde ad alcun problema incontrato prima; inoltre, si sa solo che questa successione di algoritmi via via più efficienti *esiste*, ma non come si costruiscono l'uno dall'altro!

Se anche fosse sopportabile la situazione evidenziata dal teorema precedente, quando si usano funzioni di misura che non sono appropriate si scopre che all'aumentare delle risorse non si allarga la classe dei problemi decisi con esse. Di conseguenza si contraddice l'idea intuitiva che ci debba essere una gerarchia di classi di complessità, come espresso dal teorema 2.4.1 che la stabilisce nel caso in cui le funzioni di misura siano *appropriate*. L'enunciato che segue è una forma speciale del teorema generale: invece che prendere una generica funzione calcolabile h, con $h(n) > n$, al variare della quale si costruisce la f usata sotto, consideriamo per semplicità la funzione 2^n. Il risultato netto è che non si trova alcun problema decidibile in tempo deterministico (strettamente) superiore a $f(n)$ e (strettamente) inferiore a $2^{f(n)}$. [12]

Teorema 2.4.6 (della lacuna, Borodin)**.** *Esiste f calcolabile totale tale che* $\mathrm{TIME}(f(n)) = \mathrm{TIME}\left(2^{f(n)}\right)$.

Una formulazione più fine di questo teorema richiede che la funzione f considerata sia monotona; se non lo fosse non sorprenderebbe poi molto che non vi siano problemi la cui soluzione richiede risorse comprese tra due funzioni che oscillano. Inoltre, si può vedere (dalla dimostrazione) che la funzione f che individua la lacuna cresce in maniera estremamente rapida, ciò che è proibito alle funzioni di misura appropriate.

Naturalmente, valgono anche i teoremi di accelerazione e della lacuna che si ottengono sostituendo misure di spazio invece che di tempo nell'enunciato dei due teoremi precedenti.

In un certo senso, l'ultima osservazione fatta ribadisce la sensazione che si potrebbe sviluppare una teoria della complessità che sia indipendente dallo spazio o dal tempo usati, ma dipenda solo da una nozione astratta di risorsa, istanziabile ad esse, ma anche ad altre risorse, quali l'energia consumata o

[12] Di nuovo, una visione intuitiva, anche questa dovuta a Börger, suggerisce che c'è un insieme di programmi sui quali la macchina lenta e quella veloce si equivalgono.

altro. Vediamo allora un'altra caratterizzazione delle funzioni di misura, dovuta a Blum, che si muove in quest'ottica e che sta alla base del cosidetto approccio assiomatico alla complessità. Questa caratterizzazione è particolarmente elegante in quanto è slegata da ogni modello di calcolo e richiede solo due assiomi. Inoltre, molti dei teoremi sulla complessità, in particolare quelli dell'accelerazione e della lacuna, possono venir dimostrati usando solamente tali assiomi.

Definizione 2.4.2. Una funzione ϕ è una misura di complessità se restituisce un naturale a fronte di una funzione ψ e del suo dato di ingresso x e inoltre soddisfa entrambi i seguenti assiomi:

1. $\phi(\psi, x)$ è definita se e solo se $\psi(x)$ lo è;
2. per ogni ψ, x, k, è decidibile se $\phi(\psi, x) = k$.

Il primo assioma ci dice che ϕ misura la complessità del calcolo di $\psi(x)$; il secondo assicura che si può davvero ottenere la complessità del calcolo di $\psi(x)$. Quindi, se la funzione ϕ misurasse il numero dei passi, ri-otterremmo la vecchia definizione di complessità in tempo; se misurasse le celle toccate quella in spazio. [13]

Non è difficile verificare che le funzioni di misura appropriate soddisfino i due assiomi. Tuttavia vi sono funzioni di misura che soddisfano tali assiomi e il cui impiego conduce a risultati ancora una volta sorprendenti. Oltre alle situazioni contro-intuitive appena viste, può capitare che la complessità della composizione sequenziale di due algoritmi risulti *minore* della complessità del primo dei due; in altre parole, fare ulteriori calcoli può ridurre la complessità del problema appena risolto! Non possiamo allora che rassegnarci a usare funzioni appropriate.

[13] Purché si accetti di non saper valutare lo spazio necessario a una macchina non terminante, anche nel caso in cui questa sia in ciclo.

2.5 P e NP

In questo capitolo cercheremo di esaminare un po' più in dettaglio le classi di complessità $\mathcal{P}$ e $\mathcal{NP}$, studiandone seppur superficialmente la struttura. Lo strumento principale che useremo, se non l'unico, è la ricerca di uno o più problemi completi per ciascuna di tali classi, cioè quei problemi che vi appartengono e di questa sono i più ardui, ovvero son tali per cui a loro si riducono tutti gli altri problemi della classe. Più precisamente, considereremo la relazione di riduzioni $\leqslant_{\text{logspace}}$, cioè la relazione che impiega riduzioni, o algoritmi, di complessità $\text{LOGSPACE} = \bigcup_{k\geq 1} \text{SPACE}(k \times \log n)$, la quale classifica $\mathcal{P}$ e $\mathcal{NP}$, andremo poi a cercare problemi H che siano $\mathcal{P}$-completi, cioè tali per cui ogni problema I in $\mathcal{P}$ si riduce a H (cioè H è arduo) e inoltre H appartiene a $\mathcal{P}$ (cioè H è completo) (si veda la definizione 1.10.1); analogamente per $\mathcal{NP}$.

I problemi completi caratterizzano davvero una classe di complessità, perché ne esprimono la struttura profonda e l'essenza e la difficoltà dei suoi problemi. Di conseguenza attraverso un problema completo si esprime anche il potere espressivo di una classe. A questo proposito, si ricordi l'insieme K che è RE-completo per riduzioni calcolabili totali (così come lo sono K_0 e K_1) e il ruolo che esso gioca nel determinare i gradi di solubilità e insolubilità. Inoltre, sia detto per inciso, vale poco una classe che non ha problemi completi *pre-esistenti* alla sua definizione e che siano interessanti dal punto di vista computazionale. Vedremo in questo capitolo alcuni problemi completi per $\mathcal{P}$ e per $\mathcal{NP}$, i quali sono davvero interesssanti e che son stati posti ben prima che nascesse la teoria della complessità come la conosciamo ora. Per ritornare sull'analogia con le classi di problemi calcolabili e di problemi non calcolabili, notiamo che l'insieme K_0 è davvero interessante: vorremmo, anche se da questa ricerca ne usciamo frustrati, uno strumento che dica, per ogni programma, se esso terminerà sempre — in altre parole stiamo dichiarando che l'Entscheidungproblem proposto da Hilbert nel 1900 è interessante.

Prima di procedere, ricordiamo anche che si può stabilire il seguente frammento di gerarchia tra le classi di complessità che andremo a esaminare:

$$\text{LOGSPACE} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE} = \text{NPSPACE}$$

e ricordiamo anche che $\text{LOGSPACE} \subsetneq \text{PSPACE}$. Di conseguenza, almeno una delle inclusioni

$$\text{LOGSPACE} \subseteq \mathcal{P},\ \mathcal{P} \subseteq \mathcal{NP},\ \mathcal{NP} \subseteq \text{PSPACE}$$

deve essere stretta. A tutt'oggi però non sappiamo quale essa sia, e non ci aiuta sapere anche che

$$\mathcal{P} \subsetneq \text{EXP} \quad \text{e} \quad \mathcal{NP} \subseteq \text{EXP}$$

Il problema più interessante in questo momento è risolvere $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$, che, oltre che estremamente affascinante si è rivelato resistentissimo, tanto che non si sa neppure se sia dimostrabile. Naturalmente, se facessimo vedere che un problema $\mathcal{NP}$-completo è risolto da un algoritmo che richiede tempo polinomiale deterministico, o che esso si riduce a un problema, necessariamente completo, che sta in $\mathcal{P}$, avremmo d'un botto dimostrato l'eguaglianza tra la classe dei problemi "trattabili" e dei problemi che divengono tali grazie al non determinismo (in accordo al paradigma: scommetti su una soluzione e dimostra che lo è in tempo polinomiale deterministico). Vi è tuttavia una sensazione diffusa che l'eguaglianza non valga, a ciò concorrono molti indizi. Tra questi segnaliamo soltanto il fatto che, se $\mathcal{P}$ coincidesse con $\mathcal{NP}$ ci sarebbe un modo computazionalmente accettabile di realizzare il meccanismo del non determinismo, cosa che appare per ora non poco sorprendente; a questo proposito si veda anche la nota a pagina 79 su due modelli di calcolo che paiono raggiungere tale obiettivo.

Prima di iniziare discutiamo brevemente la robustezza delle classi $\mathcal{P}$ e $\mathcal{NP}$ e poi vediamo alcune critiche alla scelta di costruire una teoria asintotica che considera sempre il caso pessimo, e come queste si riverberano sulla tesi di Cook-Karp.

2.5.1 Problemi trattabili e intrattabili

Adesso che abbiamo le nozioni di $\mathcal{P}$ e $\mathcal{NP}$ e che abbiamo visto come operano le macchine di Turing non deterministiche, possiamo rivedere l'affermazione che i problemi in $\mathcal{P}$ sono *trattabili* e quelli in $\mathcal{NP}$ *intrattabili*, ovvero discutiamo della fondatezza della *tesi di Cook-Karp*.

A sostegno di questa tesi vi è la robustezza delle classi che stiamo considerando non solo rispetto alla rappresentazione dei dati e dei problemi, ma anche rispetto al cambio dei modelli di calcolo e di trasformazioni "facili" dei problemi stessi; vediamo in un dettaglio appena maggiore di quanto fatto che cosa intendiamo con le ultime due cose.

i) Le classi $\mathcal{P}$ e $\mathcal{NP}$ sono robuste nel senso che non variano al variare dei modelli di calcolo, a patto che le funzioni per misurare il tempo siano "ragionevoli". In altre parole $\mathcal{P}$ è robusta perché è chiusa rispetto la composizione polinomiale sinistra.[14] L'idea qui è che si può sempre trovare un algoritmo T di complessità p che, dato un problema I, *trasforma* una sua procedura di decisione M, espressa in un modello $\mathcal{M}$, in una sua procedura *equivalente* P, espressa in un altro modello $\mathcal{M}'$.

[14] Una classe $\mathcal{C}$ di funzioni è chiusa rispetto la composizione polinomiale sinistra se per tutti i polinomi p il fatto che $f \in \mathcal{C}$ implica $p \circ f \in \mathcal{C}$.

Se M ha complessità polinomiale f, allora P avrà complessità $p \circ f$, che è ancora un polinomio. In termini più generali, robustezza rispetto al cambio di modelli di calcolo significa che le trasformazioni tra essi sono polinomiali (ad esempio, si consideri come un programma *WHILE* può simulare in tempo polinomiale una macchina di Turing).

ii) La classe $\mathcal{P}$ è chiusa rispetto alla somma, al prodotto, alle riduzioni $\leqslant_F$ con F (sotto-)classe dei polinomi. Questo verrà formalizzato definendo appropriate riduzioni che classificano $\mathcal{P}$ e $\mathcal{NP}$ e dimostrando il teorema 2.5.1. Per amore di simmetria rispetto quanto fatto sopra, ciò si può rifrasare prendendo riduzioni polinomiali e dicendo che $\mathcal{P}$ è chiusa rispetto alla composizione polinomiale destra. [15] In altre parole, per risolvere un problema I, trasformalo "facilmente", ovvero *riducilo*, per mezzo di un algoritmo M, che ha come complessità un polinomio p, in $I' = M(I)$ e decidi I'; se $I' \in P$, cioè I' è deciso in tempo deterministico polinomiale f, anche $I \in P$ perché è deciso in tempo deterministico $f \circ p$, che è ancora un polinomio.

Un'ulteriore difesa della tesi di Cook-Karp è che di solito gli algoritmi polinomiali hanno, o almeno se ne cercano in modo che, le costanti moltiplicative e soprattutto gli esponenti siano piccoli, mentre gli algoritmi esponenziali diventano rapidamente inefficienti al crescere dei dati di ingresso.

Tuttavia ci sono alcune critiche all'identificazione di $\mathcal{P}$ con i problemi trattabili:

i) Un algoritmo in $\mathcal{O}(n^{100})$ non è certo efficiente. Lo è molto di più, almeno per n non enormi, uno in $\mathcal{O}\left(2^{\frac{n}{100}}\right)$ o in $\mathcal{O}(n \log n)$.

ii) Ci sono algoritmi che richiedono tempo esponenziale nel caso pessimo, ma sono efficienti nei casi interessanti o almeno in quelli più comuni. Si possono trovare degli esempi di questo comportamento bizzarro in:

   – *programmazione lineare*: l'algoritmo del simplesso è semplice e intuitivo e, benché sia esponenziale nel caso pessimo, di solito è efficiente, mentre il metodo elissoide è polinomiale, ma incredibilmente lento a causa di costanti moltiplicative enormi; vale la pena di notare che recentemente è stato proposto un metodo, assai sofisticato dal punto di vista matematico, detto dei "punti interni" che è polinomiale, su cui sono basate alcune realizzazioni che sono in svariati casi pratici più efficienti del simplesso. Ovvero vale la pena di insistere nella ricerca di algoritmi più efficienti!

[15] Una classe $\mathcal{C}$ di funzioni è chiusa rispetto la composizione polinomiale destra se per tutti i polinomi p, $f \in \mathcal{C}$ implica $f \circ p \in \mathcal{C}$.

– *algoritmi di paginazione*: si supponga di avere un programma che richiede che le pagine vengano continuamente caricate e scaricate; in questo caso tutti gli algoritmi di paginazione hanno la stessa complessità (la peggiore!), ma sappiamo bene che nella pratica ci sono algoritmi migliori di altri.

– *inferenza di tipi di Standard ML*: *ogni* algoritmo è efficiente in pratica, ma esponenziale nel caso pessimo, il quale è costruito *ad hoc* e risulta alquanto artificiale.

Il primo punto suggerisce anche una critica all'aver definito una teoria asintotica della complessità, mentre gli altri due sollevano una critica al calcolo della complessità nel caso pessimo. Queste scelta sono molto forti, anche se permettono una trattazione matematica semplice e consentono di dimostrare buone proprietà di chiusura della teoria.

Ci sono diversi modi per analizzare la complessità di un algoritmo, e qui ne citiamo un paio, rimandando il lettore alla letteratura per una trattazione più accurata e completa.

Una alternativa alla complessità nel caso pessimo è quella nel caso medio: si valuta la complessità del problema quando i suoi dati siano *medi*. Ma come si determinano i dati medi? Farlo è in generale molto difficile e richiede di conoscere la distribuzione dei dati o quanto meno di poterla approssimare. A volte queste approssimazioni sono grossolane e arbitrarie o addirittura non si riescono a definire. A volte invece le approssimazioni proposte funzionano benissimo, soprattutto quando il problema abbia una "buona" struttura. Ad esempio, ci sono metodi accurati per definire il caso medio per il problema della programmazione lineare citato sopra e sotto tale ipotesi l'algoritmo del simplesso diventa efficiente, avendo complessità polinomiale con costanti piccole (in barba al metodo dei punti interni).

L'efficienza del simplesso è inoltre sostenuta da una recente proposta, chiamata *smooth complexity*, che tende a combinare i vantaggi dell'approccio alla complessità dal punto di vista del caso pessimo e di quello medio. Secondo questo tipo di analisi, che misura la complessità di un algoritmo perturbando leggermente i dati nel caso pessimo, il simplesso risulta essere polinomiale, così come accade quasi sempre.

Un ulteriore approccio alla valutazione della complessità è quello che va sotto il nome di *analisi ammortizzata*. Molto rozzamente, si considerano sequenze di k operazioni e il tempo per l'esecuzione di una di esse viene calcolato prendendo la media dei costi di tali operazioni — non tutte si applicheranno al caso più sfavorevole, quindi ci si differenzia dalla complessità nel caso pessimo. Si noti anche che non è una variante dell'analisi del costo medio, in quanto non c'è alcun uso di nozioni probabilistiche o statistiche.

Per la definizione di problemi completi, e quindi nello studio di $\mathcal{P}$ e di $\mathcal{NP}$, abbiamo bisogno di definire quelle che consideriamo riduzioni "facili" e di dimostrare che esse classificano $\mathcal{P}$ e $\mathcal{NP}$, nel senso definito nel capitolo 1.10. Ancora una volta, l'idea è che per risolvere il problema I lo si riduce efficientemente per mezzo di una funzione f a $f(I)$ appartenente a una data classe $\mathcal{D}$ e si risolve il problema ridotto; se la classe è chiusa rispetto a quelle riduzioni, che pertanto possiamo considerare efficienti rispetto $\mathcal{D}$,[16] allora anche il problema I sta in $\mathcal{D}$. Quando si tratta di $\mathcal{P}$ e di $\mathcal{NP}$, nella letteratura si considerano di solito efficienti le riduzioni che sono *polinomiali in tempo deterministico*; noi useremo invece riduzioni *logaritmiche in spazio deterministico*, stante l'uso fatto di macchine di Turing con k nastri che modellano i calcolatori paralleli e la diffusione di questi ultimi. Si noti tuttavia che ogni riduzione logaritmica in spazio è anche polinomiale in tempo, grazie al teorema 2.2.5. Infatti, se $f \in$ LOGSPACE allora $f \in \mathcal{P}$ e quindi le funzioni di riduzione che useremo in seguito sono più "facili" di quelle classiche polinomiali in tempo, o almeno altrettanto difficili.

Definizione 2.5.1. (cf. definizione 1.10.3)
Un problema I si *riduce efficientemente* a I' ($I \leqslant_{logspace} I'$) se esiste un algoritmo $f \in$ LOGSPACE tale che

$$x \in I \text{ se e solamente se } f(x) \in I'$$

Adesso vediamo che la classe delle funzioni in LOGSPACE induce una relazione di riduzione che classifica LOGSPACE e $\mathcal{D}$, dove $\mathcal{D}$ è una qualunque delle classi che abbiamo introdotto (si veda la definizione 1.10.3 e si osservi che banalmente LOGSPACE $\subseteq \mathcal{D}$). Nel teorema seguente enunciamo anche l'analogo per le riduzioni in $\mathcal{P}$, che spesso vengono usate al posto di quelle che impieghiamo noi, senza peraltro modificare i risultati principali. Il lettore è anche invitato a riguardare la breve discussione sulla tesi di Cook e Karp nel capitolo 2.5.1 alla luce del seguente teorema.

Teorema 2.5.1. *Siano* $\mathcal{D}, \mathcal{E} \in \{\mathcal{P}, \mathcal{NP}, EXP, PSPACE, NPSPACE\}$ *e* $\mathcal{D} \subseteq \mathcal{E}$

- $\leqslant_{\text{logspace}}$ *classifica* {LOGSPACE} *ed* $\mathcal{E}$
- $\leqslant_{\text{logspace}}$ *e a maggior ragione* $\leqslant_{\mathcal{P}}$ *classificano* $\mathcal{D}$ *e* $\mathcal{E}$

Dimostrazione. Immediata per entrambi gli enunciati, in quanto tutti i punti richiesti dal lemma 1.10.1 sono banalmente soddisfatti. Si noti in particolare che la composizione di due macchine che operano in spazio logaritmico

[16] Ecco perché le riduzioni devono essere "facili" nel senso dato dalla tesi di Cook-Karp: comporre un polinomio con una funzione che domina ogni polinomio ci farebbe uscire dalla classe $\mathcal{P}$.

è ancora una macchina in LOGSPACE. Infatti, basta lanciare la seconda macchina e, non appena questa necessita di un carattere di ingresso, farlo calcolare dalla prima macchina, lanciandola sull'ingresso dato e scrivendo i caratteri via via calcolati sulla medesima casella (si noti che l'output può essere di lunghezza polinomiale; naturalmente serve anche una casella che contenga la posizione del carattere di output appena generato); questo modo di procedere spreca un sacco di tempo e una casella per ricordarci quale carattere è necessario alla seconda macchina, ma stiamo misurando lo spazio! □

2.5.2 Alcuni problemi interessanti e riduzioni efficienti tra essi

Prima di cominciare la nostra indagine su problemi che sono $\mathcal{P}$-completi e su quelli $\mathcal{NP}$-completi, vediamone qualche esempio. Abbiamo già incontrato il problema del commesso viaggiatore che è tale; di seguito ne introdurremo altri presi dalla teoria dei grafi; essendo tutti $\mathcal{NP}$-completi, hanno tutti una struttura profonda molto simile, anche se ciò non è evidente a prima vista. Poi passeremo a considerare un problema preso dalla logica: il problema della soddisfacibilità[17] di una proposizione. Ancor maggiore può sembrare la differenza tra questo problema e quello del commesso viaggiatore, anche per l'apparente distanza tra il calcolo proposizionale e la teoria dei grafi. Faremo vedere che tutti i problemi menzionati in precedenza si riducono al problema della soddisfacibilità di una proposizione; in realtà dimostreremo che *tutti* i problemi in $\mathcal{NP}$ si riducono in spazio logaritmico a tale problema: cioè il problema della soddisfacibilità di una proposizione è esso stesso $\mathcal{NP}$-completo. Sempre nel campo della logica, o meglio in quello affine dei circuiti booleani, prenderemo anche il nostro principale problema $\mathcal{P}$-completo.

Vale la pena di osservare che tutti i problemi che introdurremo sono interessanti nel campo in cui sono stati studiati, indipendentemente dall'indagine della loro complessità che facciamo qui; ciò risponde positivamente alla domanda se $\mathcal{P}$ e $\mathcal{NP}$ siano classi di complessità interessanti *in sé*.

Possiamo adesso elencare alcuni problemi paradigmatici per la classe $\mathcal{NP}$ e presentare delle funzioni di riduzione tra loro; poi faremo lo stesso per $\mathcal{P}$. Come già accennato, sono problemi ben noti in calcolo proposizionale e in teoria dei grafi.

[17] Sebbene alcuni autori prevalentemente pisani propendano per un forse più elegante *soddisfattibilità*, noi continueremo a usare *soddisfacibilità* in quanto tale termine è in uso nella comunità dei logici fin dagli anni trenta e, parafrasando Giacomo Leopardi, nella lingua l'uso è ragione.

Nel resto del capitolo, ci limiteremo a considerare solo espressioni booleane in forma normale congiuntiva. Inoltre rappresenteremo un grafo o come si fa di solito con un disegno, oppure come una coppia $(N, A \subseteq N \times N)$, dove con $i \in N$ $(1 \leq i \leq \#N = n)$ indicheremo i nodi e con la coppia (ordinata se il grafo è orientato) $(i, j) \in A$ l'arco dal nodo i al nodo j.

Problema SAT Il problema SAT, o di *soddisfacibilità*, consiste nel decidere se, data un'espressione booleana B esiste un assegnamento $\mathcal{V}$ tale che $\mathcal{V} \vDash B$ (a volte si dice semplicemente B è soddisfacibile).

Chiaramente SAT appartiene a $\mathcal{NP}$: basta scegliere a caso un assegnamento per le variabili di B e lasciar scegliere al non-determinismo la soluzione buona, se c'è, ovvero uno di quelli che rendono vera l'espressione; vedremo più avanti, ma non è difficile farlo, che la *certificazione*, ovvero l'eventuale controllo che l'assegnamento proposto sia davvero una soluzione, richiede tempo deterministico polinomiale.

Problema HAM Il problema HAM consiste nel decidere se in un grafo (orientato) c'è un cammino, detto *hamiltoniano*, che tocca tutti i nodi una e una sola volta.[18]

Finalmente vediamo che HAM si può ridurre in spazio logaritmico a SAT, cioè SAT è un problema almeno tanto difficile quanto lo è HAM, visto che la riduzione usata è efficiente.

Proprietà 2.5.1. $HAM \leqslant_{\text{logspace}} SAT$

Dimostrazione. Dato un grafo G, dobbiamo costruire $f \in$ LOGSPACE tale che G ha un cammino hamiltoniano se e solamente se $f(G)$ è soddisfacibile. Prima introdurremo un'espressione booleana in forma congiuntiva che "rappresenta" i cammini hamiltoniani (non nella forma più economica: ci sono alcune ridondanze, infatti); poi facciamo vedere che questa è soddisfacibile se e solo se G ha un tale cammino; infine delineamo una macchina di Turing che costruisce a partire da G proprio l'espressione booleana e che usa uno spazio di lavoro logaritmico rispetto al numero dei nodi di G.

Sotto l'ipotesi che G abbia n nodi, $f(G)$ ha n^2 variabili booleane $x_{i,j}$, $1 \leq i, j \leq n$ (si noti che la coppia (i, j) è sufficiente a individuare la variabile $x_{i,j} \in X$). Ciascuna variabile è usata per "rappresentare" che "il nodo j è

[18] Se si volesse considerare il problema del ciclo hamiltoniano basterebbe richiedere la presenza di un arco dall'ultimo nodo del cammino hamiltoniano al primo; la presenza di tale arco è necessaria anche nella dimostrazione della proprietà che segue: bisognerà allora introdurre nuovi congiunti a tale scopo.

l'i-esimo di un cammino (hamiltoniano)". Poiché $f(G)$ sarà in forma congiuntiva, basta costruire i congiunti che la compongono. Esprimiamo con ciascuna delle seguenti formule il fatto che la f è una permutazione sull'intervallo $[1 \dots n]$ (i punti 1 e 2 dicono che f è davvero una funzione e che è definita su ogni elemento di $[1 \dots n]$, il punto 3 che è surgettiva e il 4 che è iniettiva). Infine scriveremo un'espressione che risulta vera se e solamente se una sequenza di nodi (ovvero di numeri $1 \leq j \leq n$) rappresenta un cammino in G (punto 5).

1. lo stesso nodo j non può apparire in due posizioni diverse nello stesso cammino, cioè $\neg(x_{i,j} \wedge x_{k,j})$, ovvero applicando la legge di De Morgan:

$$(\neg x_{i,j} \vee \neg x_{k,j}) \quad k \neq i$$

2. ogni nodo j deve apparire in un cammino:

$$(x_{1,j} \vee x_{2,j} \vee \cdots \vee x_{n,j}), \quad 1 \leq j \leq n$$

3. qualche nodo deve essere l'i-esimo di un cammino:

$$(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n}), \quad 1 \leq i \leq n$$

4. due nodi non possono essere contemporaneamente l'i-esimo nello stesso cammino (ancora De Morgan):

$$(\neg x_{i,j} \vee \neg x_{i,k}) \quad j \neq k$$

5. se (i, j) non è un arco di G, i e j non possono apparire in sequenza in un cammino (hamiltoniano o no):

$$(\neg x_{k,i} \vee \neg x_{k+1,j}) \;\; \forall k.\, 1 \leq k \leq n-1 \text{ e } \forall (i,j) \notin A$$

Adesso dimostriamo che $f(G)$, la congiunzione delle formule ottenute in accordo con i punti 1-5, ha un assegnamento $\mathcal{V}$ tale che lo soddisfa se e solamente se G ha un cammino hamiltoniano.

Prima vediamo frettolosamente che G ha una cammino hamiltoniano se $\mathcal{V} \vDash f(G)$. In questo caso, per ogni j esiste unico i tale che $\mathcal{V}(x_{ij}) = tt$ altrimenti le clausole definite nei punti 1 e 2 non potrebbero essere soddisfatte entrambe (p.e., $\mathcal{V}(x_{1,1}) = tt$ e $\mathcal{V}(x_{2,1}) = tt$ fan sì che $(\neg x_{1,1} \vee \neg x_{2,1}) = ff$). Allo stesso modo per ogni i esiste unico j tale che $\mathcal{V}(x_{i,j}) = tt$ (p.e., $\mathcal{V}(x_{1,1}) = tt$ e $\mathcal{V}(x_{1,2}) = tt$ fan sì che $(\neg x_{1,1} \vee \neg x_{1,2}) = ff$).

Quindi $\mathcal{V}$ rappresenta una permutazione $\Pi(1)\dots\Pi(n)$ dei nodi di G e le clausole definite nel punto 5 garantiscono che la permutazione sia effettivamente un cammino. Del resto un cammino hamiltoniano non è altro che una permutazione dei nodi del grafo, a due a due connessi da un arco. Quindi l'espressione booleana $f(G)$ rappresenta un cammino hamiltoniano se è soddisfatta, il che era la nostra ipotesi.

Adesso sia $H = (\Pi(1), \dots, \Pi(n))$, un cammino hamiltoniano. Allora è immediato verificare che

$$\mathcal{V}(x_{i,j}) = \begin{cases} tt & \text{se } \Pi(j) = i \\ ff & \text{altrimenti} \end{cases}$$

soddisfa $f(G)$, perché il nodo j è l'i-esimo di un cammino hamiltoniano.

Rimane da verificare che $f \in$ LOGSPACE. Costruiamo una MdT di tipo I/O nel modo seguente. L'alfabeto contiene l'insieme $\{tt, ff, \neg, \wedge, \vee, (,)\} \cup \{0, 1\}$ (così gli indici delle variabili $x_{i,j}$, o meglio le variabili stesse, vengono rappresentati in binario). Descriviamo i passi di M raggruppandoli così:

i) M scrive il numero dei nodi n in binario su un nastro di lavoro;

ii) poi M scrive sul nastro di output le clausole definite nei punti da 1 a 4, che dipendono *solo* dal numero n dei nodi di G; nel fare questo legge il nastro di lavoro su cui è memorizzato n; si noti che tutto quello che serve sono 3 contatori che contengono i, j, k, i quali scorrono sulla rappresentazione di n in binario. Quindi quattro nastri di lavoro sono sufficienti — anzi, ne basterebbero tre perché i nastri possono venir inizializzati ad n per poi procedere a ritroso scalando il loro valore fino a 0.

iii) M genera le clausole $(\neg x_{k,i} \vee \neg x_{k+1,j})$ definite nel punto 5 una alla volta e le scrive sul nastro di output se $(i, j) \notin A$ (si ricordi che la descrizione degli archi è memorizzata sul nastro di input).

Tutto quello che ci serve sono quattro (o meglio tre) copie di n, il che richiede $\mathcal{O}(\log n)$ bit perché gli indici sono in binario. Allora $f \in$ LOGSPACE perché lo spazio necessario è la somma delle caselle visitate sui nastri di lavoro.

□

Il risultato netto della funzione f costruita nella dimostrazione precedente è che HAM è stato "tradotto" in SAT — il risultato sorprendente è che il linguaggio logico del calcolo proposizionale, sia pur povero, è abbastanza potente da rappresentare un problema "difficile" come HAM e in un formalismo completamente diverso.

Vediamo adesso un altro problema e un'altra riduzione.

Problema Cricca Il problema CRICCA consiste nel decidere se in un dato grafo (non orientato) $G = (N, A)$ esiste $C \subseteq N$ detto cricca (di grado k, funzione del numero di nodi), tale che $\forall i, j \in C$, con $i \neq j$, l'arco $(i, j) \in A$.

Ora vediamo che SAT si può ridurre in spazio logaritmico a CRICCA, cioè CRICCA è un problema almeno tanto difficile quanto lo è SAT.

Proprietà 2.5.2. *SAT* $\leqslant_{\text{logspace}}$ *CRICCA*

Dimostrazione. Diamo solo l'idea della costruzione. Data un'espressione booleana $B = \bigwedge_{1 \leq k \leq n} C_k$, costruisci il grafo $f(B) = (N, A)$ così:

i) N è l'insieme delle occorrenze dei letterali in B;

ii) A è l'insieme $\{(i, j) \mid i \in C_k \Rightarrow (j \notin C_k \wedge i \neq \neg j)\}$.

Essendo l'espressione B in forma normale congiuntiva, essa è soddisfacibile se e solamente se c'è almeno un letterale vero in ogni suo congiunto. Quindi B è soddisfacibile se e solamente se $f(B)$ ha una cricca di ordine pari al numero di congiunti: basta attribuire *tt* ai letterali corrispondenti ai nodi della cricca. La costruzione infatti garantisce che un nodo, cioè un letterale, non può essere connesso ad un arco al nodo originato dallo stesso letterale negato, né può esserlo a un nodo corrispondente a un letterale che compare nello stesso congiunto.

La riduzione è logaritmica in spazio perché basta mantenere sui nastri di lavoro due indici, rappresentati in binario, che scorrono i letterali.

□

Esempio 2.5.1. Per esemplificare la dimostrazione della proprietà precedente, si consideri l'espressione booleana $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z)$ che dà origine al grafo rappresentato in figura 2.1. Esso ha tre archi uscenti e tre entranti in ciascun nodo. Una cricca di grado tre è l'insieme di nodi $\{\neg x, \neg y, z\}$ che corrisponde all'assegnamento $\mathcal{V}(x) = \mathit{ff}, \mathcal{V}(y) = \mathit{ff}, \mathcal{V}(z) = \mathit{tt}$.

In questo esempio tutti i letterali sono diversi, ma se volessimo aggiungere il congiunto $x \vee z$, la costruzione originerebbe due nuovi nodi, etichettati come quelli all'estrema destra e all'estrema sinistra nella figura.

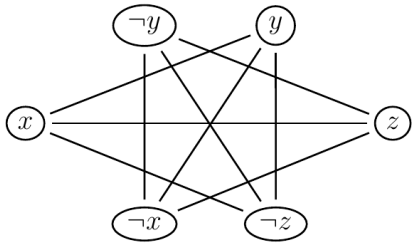


Figura 2.1: Un grafo per l'espressione $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z)$; l'insieme $\{\neg x, \neg y, z\}$ è una cricca di grado 3 e mostra come l'espressione sia soddisfacibile.

Ci sono ancora un paio di problemi interessanti per gli sviluppi futuri. Per descriverli useremo una rappresentazione speciale delle funzioni booleane (totali) $f : \{ff, tt\}^n \to \{ff, tt\}$.

Innanzitutto, dovrebbe già essere noto che tutte e sole le funzioni booleane sono rappresentabili da espressioni booleane; per i nostri scopi, basta dire che l'espressione booleana con n variabili B rappresenta $f(x_1, \ldots, x_n)$ quando dato l'assegnamento $\mathcal{V}(x_i) = t_i \in \{ff, tt\}$,

$$f(x_1, \ldots, x_n) = tt \text{ se e solamente se } \mathcal{V} \vDash B$$

Introduciamo adesso i circuiti booleani, che sono un altro modo ancora di rappresentare, o forse realizzare, le espressioni booleane e, per quanto detto, anche le funzioni booleane.

Definizione 2.5.2 (Circuito Booleano)**.** Un *circuito booleano* è un grafo diretto aciclico (N, A), i cui nodi $1, \ldots, n$ sono detti *porte*, e i cui archi sono di solito rappresentati come coppie ordinate: un arco orientato da i a j è rappresentato da (i, j).

Le porte hanno 0, 1, o 2 ingressi e sono di sorta $s(i) \in \{tt, ff, \neg, \wedge, \vee\} \cup X$, dove X è l'insieme delle *variabili*;

Gli *ingressi* i del circuito sono le porte di sorta $s(i) \in \{tt, ff\} \cup X$, e non hanno ingressi; l'*uscita* del circuito è, per convenzione, la porta n senza uscite[19]; tutte le altre porte hanno un'uscita e quando $s(i) = \neg$, la porta i ha un solo ingresso e quando $s(i) \in \{\wedge, \vee\}$, allora la porta i ha 2 ingressi.

Come visto prima, per rappresentare le funzioni booleane ci serve un assegnamento $\mathcal{V}$ di valori di verità alle variabili di un circuito C, che postuliamo essere buono, cioè tale da legare ogni variabile di C a un valore in $\{tt, ff\}$.

[19]La restrizione di avere una singola uscita si può facilmente rimuovere, permettendo così di avere circuiti con molte uscite.

Allora il valore di verità $[\![i]\!]_{\mathcal{V}}$ calcolato dalla porta i è definito induttivamente dalle seguenti clausole:

$$\begin{array}{lll}
[\![i]\!]_{\mathcal{V}} & = tt & \text{se } s(i) = tt \\
& = ff & \text{se } s(i) = ff \\
& = \mathcal{V}(x) & \text{se } s(i) = x \\
& = \text{not } [\![j]\!]_{\mathcal{V}} & \text{se } s(i) = \neg \text{ e } (j,i) \in A \\
& = [\![j]\!]_{\mathcal{V}} \text{ or } [\![k]\!]_{\mathcal{V}} & \text{se } s(i) = \vee \text{ e } (j,i),(k,i) \in A \\
& = [\![j]\!]_{\mathcal{V}} \text{ and } [\![k]\!]_{\mathcal{V}} & \text{se } s(i) = \wedge \text{ e } (j,i),(k,i) \in A
\end{array}$$

Infine il valore del circuito è $\mathcal{V}(C) = [\![n]\!]_{\mathcal{V}}$, quello della porta di uscita.

Per comprendere meglio cos'è un circuito, consideriamo l'espressione booleana $(x \vee (x \wedge y)) \vee ((x \wedge y) \wedge \neg(y \vee z))$ che può essere espressa mediante circuito rappresentato in figura 2.2.

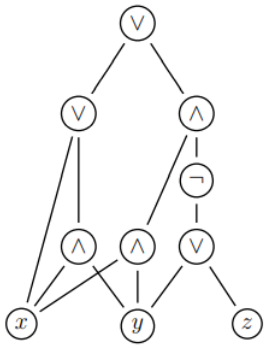


Figura 2.2: Un circuito booleano soddisfacibile.

Possiamo ora descrivere un altro problema, e poi una sua variante, che verranno usati per caratterizzare le classi $\mathcal{P}$ e $\mathcal{NP}$.

Problema CIRCUIT SAT Il problema CIRCUIT SAT, o della *soddisfacibilità dei circuiti*, consiste nel decidere se esiste un assegnamento $\mathcal{V}$ tale che $\mathcal{V}(C) = tt$.

Un modo immediato per risolvere questo problema è di sostituire alle variabili $x, y, z, \ldots$ *tutte le possibili* combinazioni di tt, ff e di *controllare* una ad una le combinazioni per verificare se sono una soluzione. Questa è ancora una volta una ricerca esaustiva per forza bruta. La versione non deterministica mostra che CIRCUIT SAT $\in \mathcal{NP}$ e prevede due passi: nel primo si sceglie in tempo polinomiale non deterministico l'assegnamento "giusto" tra i 2^n possibili assegnamenti, se n sono le variabili; nel secondo, *polinomiale* nel

numero delle porte del circuito, si *certifica* il risultato. Questo secondo passo può essere opportunamente visto come un problema di decisione, visto che risulterà essere particolarmente interessante.

Problema CIRCUIT VALUE Il problema CIRCUIT VALUE consiste nel calcolare il valore di un circuito *senza variabili*, ovvero in cui gli ingressi sono porte di sorta $s(i) \in \{tt, ff\}$.

È immediato vedere che questo problema appartiene a $\mathcal{P}$: basta "arrampicarsi" sull'ordinamento parziale a partire dalle porte di ingresso, valutando i valori di uscita delle n porte livello per livello; non ci si faccia confondere dal fatto che il numero totale delle porte può essere esponenziale rispetto la profondità del circuito, cioè il livello della porta di uscita: la taglia del circuito è data infatti proprio dal numero n delle sue porte.

Per calcolare il valore di un circuito è sufficiente allora al passo i-esimo memorizzare su un nastro di lavoro i valori delle porte di quel livello, usando i valori calcolati ai passi precedenti, ovvero quelli delle porte di livello inferiore, che sono stati via via memorizzati sul nastro di lavoro; i valori delle porte di ingresso stanno sul nastro di ingresso.

2.5.3 Problemi completi per P e NP

Iniziamo con alcune proprietà che ci saranno utili per presentare alcuni problemi e poi dimostrarli completi per $\mathcal{P}$ ed $\mathcal{NP}$.

Il seguente teorema è banale e illustra un tipo speciale di riduzione: quella per *generalizzazione*. Ogni caso particolare di un problema si riduce al problema stesso nella sua piena generalità attraverso la funzione identità.

Proprietà 2.5.3. *CIRCUIT VALUE* $\leqslant_{\text{logspace}}$ *CIRCUIT SAT.*

Correliamo adesso CIRCUIT SAT a un problema appena più complesso e facciamo vedere che quando due problemi sono molto simili è facile trovare una riduzione tra essi (e chi l'avrebbe mai detto?); l'esempio che consideriamo è la riduzione di CIRCUIT SAT a SAT.

Proprietà 2.5.4. *CIRCUIT SAT* $\leqslant_{\text{logspace}}$ *SAT.*

Dimostrazione. Dato il circuito $C = (N, A)$ con variabili in X, dobbiamo trovare una riduzione $f \in$ LOGSPACE tale che l'espressione booleana $f(C)$ sia soddisfacibile se e solamente se C lo è, in simboli tale che se $\exists \mathcal{V}.\ [\![C]\!]_{\mathcal{V}} = tt$ allora e solamente allora $\exists \mathcal{V}'$ tale che $\forall x \in X.\ \mathcal{V}'(x) = \mathcal{V}(x)$ e $\mathcal{V}' \vDash f(C)$. Il gioco è facile, perché sia i circuiti che le espressioni booleane rappresentano funzioni booleane, ma non è affatto banale.

i) Le variabili x di $f(C)$ sono quelle di C unite a un nuovo insieme che contiene una nuova variabile per ogni porta di C; per semplicità, diamo a queste nuove variabili lo stesso nome che hanno le porte, fidandoci del fatto che il contesto ci dice quando sono le une e quando le altre;

ii) per ogni porta g di C costruiamo i congiunti di $f(C)$ così:

   - se g è la porta di uscita, allora genera il congiunto g; ciò garantisce che l'assegnamento $\mathcal{V}'$ manderà g in tt altrimenti l'intera formula $f(C)$ risulterebbe falsa;
   - se $s(g) = tt$ (o ff) allora genera g (oppure $\neg g$), e l'equivalenza è ovvia — si noti che se il circuito avesse solo una porta g di sorta ff la sua immagine sotto f sarebbe $g \wedge \neg g$ che non può essere soddisfacibile (il primo congiunto viene dalla prima condizione, il secondo da quella che stiamo considerando);
   - se $s(g) = x \in X$ allora genera $(\neg g \vee x) \wedge (g \vee \neg x)$, ovvero $g \iff x$ (cioè g *se e solamente se* x, ricordando la definizione di $\iff$ in termini di $\neg, \vee$ e $\wedge$). Quindi entrambi i congiunti sono soddisfatti da $\mathcal{V}'(g) = \mathcal{V}'(x) = \mathcal{V}(x)$;
   - se $s(g) = \neg$ e $(h, g) \in A$ allora genera $(\neg g \vee \neg h) \wedge (g \vee h)$, ovvero $g \iff \neg h$, e l'equivalenza tra le due formulazioni è ancora ovvia;
   - se $s(g) = \vee$ e $(h, g), (k, g) \in A$ allora genera $(\neg h \vee g) \wedge (\neg k \vee g) \wedge (h \vee k \vee \neg g)$, ovvero $g \iff (h \vee k)$, e l'equivalenza tra le due formulazioni è immediata;
   - se $s(g) = \wedge$ e $(h, g), (k, g) \in A$ allora genera $(\neg g \vee h) \wedge (\neg g \vee k) \wedge (\neg h \vee \neg k \vee g)$, ovvero $g \iff (h \wedge k)$, e l'equivalenza tra le due formulazioni è immediata ancora una volta.

La dimostrazione che la trasformazione richiede spazio logaritmico si basa sulle stesse osservazioni fatte per mostrare che HAM si riduce a SAT.

□

Esempio 2.5.2. Per vedere come funziona la riduzione delineata nella dimostrazione precedente, si consideri il semplicissimo circuito illustrato in Figura 2.3, composto da tre porte g, h e k con g porta di uscita e $s(g) = \wedge$ e h, k porte di ingresso di sorta rispettivamente x e ff.

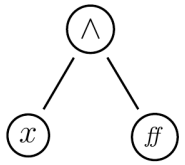


Figura 2.3

Naturalmente C sempre ha valore *ff* e quindi ci aspettiamo un'espressione $f(C)$ non soddisfacibile. La porta di uscita genera i seguenti quattro congiunti, il primo per la prima condizione, gli altri tre per l'ultima: $g \wedge (g \vee \neg h \vee \neg k) \wedge (\neg g \vee h) \wedge (\neg g \vee k)$. La porta h genera i seguenti due congiunti: $(\neg h \vee x) \wedge (h \vee \neg x)$. Infine la porta k genera $\neg k$.

È immediato verificare che la congiunzione di tutte queste sette disgiunzioni non è soddisfacibile poiché a g deve essere assegnato il valore tt e quindi $(\neg g \vee k)$ valuta a tt se e solamente se a k viene assegnato il valore tt, ma in questo caso l'ultimo congiunto $\neg k$ varrebbe *ff*.

Solo per ricordare che la classe LOGSPACE è chiusa per composizione e transitività (vedi il teorema 2.5.1) possiamo enunciare anche le seguenti proprietà.

Corollario 2.5.1.

$$\textit{CIRCUIT VALUE} \leqslant_{\text{logspace}} \textit{SAT}$$
$$\textit{CIRCUIT VALUE} \leqslant_{\text{logspace}} \textit{CRICCA}$$

Possiamo passare adesso a caratterizzare $\mathcal{P}$ e $\mathcal{NP}$ mediante uno o più problemi completi. Ricordiamo anche che, data una classe di funzioni F tale che $\leqslant_F$ classifica $\mathcal{D}$ ed $\mathcal{E}$ (con $\mathcal{D} \subseteq \mathcal{E}$), se un problema A è completo per $\mathcal{E}$ e $A \in \mathcal{D}$ allora $\mathcal{E} = \mathcal{D}$. Quindi studiare i problemi completi per una classe significa davvero caratterizzarla. Prima abbiamo po' di abbreviazioni e di ipotesi di lavoro, per semplificare le dimostrazioni e standardizzare la nozione di computazione. La più importante riguarda un modo di rappresentare le computazioni delle macchine deterministiche come una successione di configurazioni opportunamente organizzate in una matrice. La definizione copre il caso di macchine a 1 nastro, ma se la macchina fosse a k nastri la potremmo sempre ridurre in tempo polinomiale a una macchina a 1 nastro grazie al teorema 2.2.1, riportandoci al caso precedente.

Definizione 2.5.3 (Tabella di Computazione)**.** La *tabella di computazione* T_M di una macchina di Turing M a 1 nastro deterministica, o semplicemente T quando non vi sia ambiguità, è una matrice quadrata il cui indice di riga

i rappresenta l'i-esimo passo di computazione e il cui indice di colonna j rappresenta la j-esima posizione sul nastro. Di conseguenza, la riga i-esima rappresenta la configurazione di M dopo il passo $i-1$ e l'elemento $T(i,j)$ contiene il simbolo contenuto nella cella j-esima del nastro dopo $i-1$ passi di computazione.

Se M decide il problema I in tempo polinomiale deterministico $|x|^k$, la sua tabella di computazione (o semplicemente la sua computazione) su x ha al massimo $|x|^k$ righe e altrettante colonne.[20]

Per semplificarci la vita, facciamo alcune ulteriori standardizzazioni. Data una macchina M, conveniamo che

1. il valore di k sia tale per cui la macchina M si arresta prima di $|x|^k - 2$ passi — quindi k deve essere abbastanza grande per garantirlo (v. anche il punto 5) e ciò è facile quando $|x| \geq 2$; se invece $|x| \leq 1$ allora agisci localmente su questo caso particolare, ovvero fai come se non esistesse. (A dire il vero, sarebbe sufficiente considerare tabelle di dimensione $|x|^k + 2$, ma tale valore risulterebbe noioso da scrivere.)

2. Il nastro è riempito nelle posizioni non significative a destra con tanti # quanti ne servono per arrivare alla casella in colonna $|x|^k$: la testina di M non oltrepasserà mai tale posizione — in una macchina che decide un problema, il tempo limita lo spazio!

3. Arricchiamo l'alfabeto di M in modo che la casella $T(i,j)$ contenga il nuovo simbolo σ_q, per registrare che nella configurazione i-esima la testina viene a trovarsi sulla j-esima casella, il simbolo letto è σ e lo stato corrente è q — per far ciò basta prendere $\Sigma \times Q$ nuovi simboli, che sono la congiunzione del simbolo σ e dello stato q.

4. La configurazione iniziale ha la testina sul carattere immediatamente a destra del simbolo di inizio nastro, cioè M inizia a calcolare da $\triangleright \underline{\sigma}_{q_0} w$ e non da $\underline{\triangleright}_{q_0} \sigma w$. Inoltre trasformiamo la funzione di transizione di M in modo che la testina non si posizioni mai sul simbolo di inizio stringa $\triangleright$ (quando ciò accadesse, basterebbe condensare in un solo passo i due compiuti dalla macchina M: lo spostamento a sinistra sulla casella contenente il simbolo $\triangleright$ e il successivo spostamento a destra). Questa ipotesi fa sì che $\triangleright$ si comporti come un vero respingente! c'è però l'eccezione riportata nel punto seguente.

[20] Con queste dimensioni, una tabella di computazione ha ovviamente abbastanza righe, ed ha anche abbastanza colonne, perché non si può usare più spazio che tempo.

5. Se $T(i,j) \in \{\sigma_{SI}, \sigma_{NO}\}$ allora "sposta" il cursore fino alla seconda colonna (con al massimo $\mathcal{O}\left(|x|^k\right)$ passi, introducendo uno stato ausiliario di finto arresto). Cioè lo stato di accettazione, se è raggiunto, sia sempre in $T(l,2)$ per qualche $l \leq |x|^k$. Si noti che anche queste manovre possono influire sul valore di k, per esempio nel caso in cui M si arresti proprio all'estremità destra del nastro. Inoltre, come eccezione del punto precedente, si ammette che la testina si sposti sopra il simbolo $\triangleright$ quando lo stato sia q_{SI} (abbreviazione di $\sigma_{q_{SI}}$), con il vincolo che non debba *mai* toccare il simbolo $\triangleright$ più a sinistra, che rappresenta l'inizio del nastro (v. ipotesi 4).

6. Se σ_{SI} oppure σ_{NO} appaiono sulla riga $p < |x|^k$ e nella seconda colonna, allora tutte le righe di indice q, $p \leq q \leq |x|^k$, sono uguali alla p-esima.

Per finire, abbiamo l'ovvia condizione di terminazione con successo: M accetta x se e solamente se esiste i tale che $T(i,2) = \sigma_{SI} (= T(|x|^k, 2))$.

Osservazione Sia M una macchina di Turing che decide I in $|x|^k$, e T la sua tabella di computazione su x. Abbiamo che

- la cella $T(1,2)$ contiene lo stato iniziale e il primo carattere di x; inoltre $\forall j.\ 2 \leq j \leq |x|+1$, la casella $T(1,j)$ contiene il $(j-1)$-esimo simbolo di x; infine $\forall j\ |x|+2 \leq j \leq |x|^k$, la cella $T(1,j)$ contiene $\#$;
- $\forall i.\ T(i,1) = \triangleright$;
- $\forall i.\ T\left(i, |x|^k\right) = \#$ — si impiega meno spazio che tempo.

Quindi la prima riga è completamente determinata dal dato di ingresso e la prima e l'ultima colonna sono fisse in *tutte* le tabelle di computazione.

La cosa importante adesso è capire come si determina $T(i,j)$ in tutti gli altri casi, una volta fissata la funzione di transizione δ; il suo valore *dipende solo da tre caselle*: quelle della riga precedente, nella stessa posizione o in quelle immediatamente ai suoi lati, cioè *il valore di $T(i,j)$ è completamente determinato da quelli di $T(i-1,j-1), T(i-1,j), T(i-1,j+1)$*.

Per convincersene si esamini la figura 2.4 e si considerino i seguenti casi:

i) Se le tre caselle contengono $\sigma \in \Sigma$, cioè il cursore non si trova su alcune di esse, allora $T(i,j) = T(i-1,j)$.

ii) Se una di esse contiene σ_q, cioè è la casella su cui c'è il cursore, allora $T(i,j)$, che può essere diverso da $T(i-1,j)$ per effetto di uno spostamento e/o di una scrittura, è completamente determinato da $\delta(\sigma, q)$ e quindi da $T(i-1,j-1), T(i-1,j)$ e $T(i-1,j+1)$.

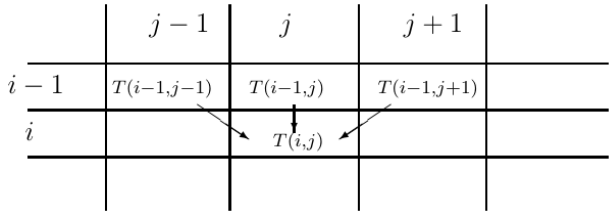


Figura 2.4: *Le tre caselle nella riga $i-1$ e nelle colonne $j-1, j, j+1$ (corrispondenti a quelle della configurazione precedente) determinano la casella $T(i,j)$ (corrispondente a quella della configurazione attuale).*

$\mathcal{P}$-completezza

Siamo pronti per presentare formalmente il primo problema $\mathcal{P}$-completo e a dimostrare che lo è davvero.

Teorema 2.5.2 (CIRCUIT-VALUE è $\mathcal{P}$-completo)**.**
CIRCUIT-VALUE è $\leqslant_{logspace}$-completo per $\mathcal{P}$.

Dimostrazione. Sappiamo già che CIRCUIT-VALUE $\in \mathcal{P}$, quindi prendiamo un qualunque $I \in \mathcal{P}$ e facciamo vedere che c'è una riduzione $f \in$ LOGSPACE che lo trasforma in CIRCUIT-VALUE; ovvero $x \in I$ se e solamente se $f(x)$ è un circuito *senza* variabili il cui valore è tt.

Sia M una macchina di Turing che decide I in n^k, e T la sua tabella di computazione su x e inoltre sia Σ' l'alfabeto di M unito ai nuovi simboli $\sigma_q \in \Sigma \times (Q \cup \{\epsilon\})$ che si usano per costruire la tabella T.

Come primo passo della dimostrazione costruiamo un circuito a partire dalla tabella di computazione di M. Per far ciò codifichiamo ogni simbolo $\rho \in \Sigma'$ con una stringa di bit $(S_1, \ldots, S_m) \in \{tt, ff\}^m$, dove naturalmente $m = \lceil \log \#(\Sigma') \rceil$. Con questa rappresentazione, la tabella di computazione T risulta avere $|x|^k$ righe che sono stringhe di bit, ciascuna lunga $m \times |x|^k$. Possiamo allora rappresentare ciascun elemento della tabella con $S_{i,j,l}$, con $1 \leq i, j \leq |x|^k$ e $l \leq m$.

Innanzitutto, $\forall i$ la m-upla $S_{i,1,1} \ldots S_{i,1,m}$ codifica il simbolo $\triangleright$ e analogamente $\forall i.\ S_{i,|x|^k,1} \ldots S_{i,|x|^k,m}$ codifica il simbolo $\#$.

Inoltre, poiché la funzione di transizione δ_M della macchina M è fissata, possiamo usare l'osservazione fatta appena sopra riguardo al valore della casella $T(i,j)$ nella tabella di computazione: in modo del tutto analogo il valore di $S_{i,j,l}$ dipende solo dal valore delle tre sequenze di m bit nella riga precedente con indice di colonna $j-1, j, j+1$, cioè dai $3m$ bit $S_{i-1,j-1,l'}, S_{i-1,j,l''}, S_{i-1,j+1,l'''}$ con $1 \leq l', l'', l''' \leq m$. Allora ci sono m funzioni

booleane $F_1, \ldots, F_m$, con $3m$ ingressi ciascuna, tali che $\forall i, j > 0$

$$S_{i,j,l} = F_l = \begin{pmatrix} S_{i-1,j-1,1} & \cdots & S_{i-1,j-1,m} \\ S_{i-1,j,1} & \cdots & S_{i-1,j,m} \\ S_{i-1,j+1,1} & \cdots & S_{i-1,j+1,m} \end{pmatrix}$$

A costo di essere noiosi ricordiamo che ciascuna funzione F_l dipende *solamente* dalla funzione di transizione δ_M.

Adesso non è difficile costruire, seguendo lo stesso schema, una funzione che "raggruppi" $F_1, \ldots, F_m$ in una sola funzione e che restituisca m valori a fronte degli stessi $3m$ ingressi. Per ogni funzione booleana, a uno o più valori, esiste un circuito, con lo stesso numero di uscite, che la calcola (come già accennato i circuiti booleani a m valori hanno come porte di uscita m porte, quelle senza archi uscenti). Per ora abbiamo detto come costruire il circuito booleano $\overline{C}$ con $3m$ ingressi e m uscite che codifica $T(i, j)$ dati $T(i-1, j-1), T(i-1, j)$ e $T(i-1, j+1)$; rappresentiamolo graficamente in figura 2.5, dove abbiamo decorato ciascuna copia di $\overline{C}$ con gli indici i e j per rendere chiaro il suo legame con la casella $T(i, j)$; si confronti questa con la figura 2.2, osservando che nelle nostre rappresentazioni le computazioni "procedono" verso il basso, i segnali "fluiscono" verso l'alto. A questo punto facciamo un'osservazione molto importante: poiché il circuito $\overline{C}$ dipende *solo* dalla funzione di transizione δ_M di M, la sua taglia, comunque sia misurata, purché in modo "ragionevole", è *fissata* ed è *indipendente dall'input* x.

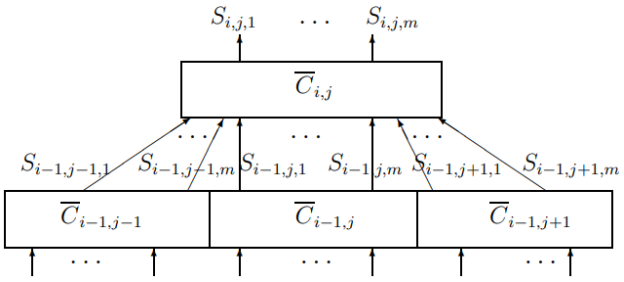


Figura 2.5: Un circuito che calcola $T(i, j)$ dati $T(i-1, j-1), T(i-1, j)$ e $T(i-1, j+1)$. Ciascuno dei circuiti in basso ha $3 \times m$ ingressi.

Siamo pronti per definire la riduzione f da I a CIRCUIT-VALUE. Sostanzialmente, si tratta di trasformare la tabella della computazione T in un circuito C_I, che è composto da una copia di $\overline{C}$ per ogni $T(i, j)$, come rozzamente illustrato illustrato in figura 2.6. (Si noti che bastano $\big(|x|^k - 1\big) \times \big(|x|^k - 2\big)$ copie perché, grazie alle ipotesi fatte sulla forma della tabella di computazione, abbiamo fissato una volta per tutte la prima riga ($\forall j \geq 2.\ T(1, j) = \sigma_{j-1}$ se il dato d'ingresso è $\sigma_1, \ldots, \sigma_{|x|}$) e la prima e l'ultima colonna ($\forall i \geq 1.\ T(i, 1) = \triangleright$

e $T\left(i, |x|^k\right) = \#$, perché la prima macchina si arresta prima di $|x|^k - 2$ passi). Quindi non c'è bisogno di costruire tali copie del circuito corrispondenti a tale riga e a tali colonne: basta prenderle dallo scaffale.) Le uscite di $\overline{C}_{i-1,j-1}, \overline{C}_{i-1,j}, \overline{C}_{i-1,j+1}$ sono gli ingressi di $\overline{C}_{ij}$. Gli ingressi di C_I sono quelli di $T(1,j)$, $T(i,1), T\left(i, |x|^k\right)$, che sono determinati da x o sono fissi. Rappresentiamo adesso i simboli σ_{SI} e σ_{NO} corrispondenti all'accettazione o al rifiuto con stringhe composte rispettivamente da soli tt e da soli ff. Allora, l'uscita di C_I è una delle uscite di $\overline{C}_{|x|^k,2}$, poiché abbiamo convenuto che tali simboli appaiono sempre in posizione $T\left(|x|^k, 2\right)$ (cf. l'ultima ipotesi fatta sulla tabella T).

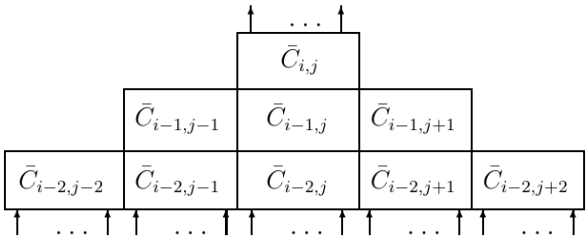


Figura 2.6: *Una parte del circuito C_I, composto dal alcune copie di $\overline{C}$.*

Il prossimo passo consiste nel verificare che $C_I = f(x)$ ha valore tt se e solamente se $x \in I$. Prima di tutto ricordiamo che le uscite di $\overline{C}_{i,j}$ rappresentano in binario il valore della cella $T(i,j)$ della tabella della computazione di M su x. Delineiamo la dimostrazione che procede per induzione sul numero i dei passi della computazione (indice di riga di T).

Se $i = 1$ banale. L'ipotesi induttiva è che il valore per $i-1$ sia ben calcolato (e dipenda solo dai tre precedenti), cioè che $\forall j.\, \overline{C}_{i-1,j}$ ha come uscite $R_1, \ldots, R_m$ se e solamente se $T(i-1,j) = \rho'$ e $\rho' \in \Sigma'$ è codificato proprio come $R_1, \ldots, R_m$. Dobbiamo dimostrare che $\overline{C}_{i,j}$ ha come uscite $S_1 \ldots S_m$ se e solamente se $T(i,j) = \rho$, la cui codifica è $S_1 \ldots S_m$. Le uscite $S_1 \ldots S_m$ sono state calcolate a partire da quelle di $\overline{C}_{i-1,j-1}, \overline{C}_{i-1,j}$ e $\overline{C}_{i-1,j+1}$ in pieno accordo con la funzione di transizione δ_M di M, esattamente allo stesso modo in cui il valore di $T(i,j)$ lo è stato in funzione dei valori di $T(i-1,j-1), T(i-1,j)$ e $T(i-1,j+1)$. Allora basta applicare l'ipotesi induttiva.

A questo punto è immediato dedurre che $\overline{C}_{|x|^k,2} = tt \ldots tt$ se e solamente se $T(n^k, 2) = \sigma_{SI}$, ovvero $x \in I$ se e solamente se $f(x) = tt \ldots tt$; analogamente per il caso $\overline{C}_{|x|^k,2} = ff \ldots ff$ e σ_{NO}.

Vediamo infine che $f \in$ LOGSPACE, cioè che la riduzione può essere calcolata in spazio $\mathcal{O}(\log |x|)$, stimando lo spazio necessario ai nastri di lavoro. Per calcolare f dobbiamo costruire e connettere opportunamente tra loro

i) le porte di ingresso — facile, esamina x e conta fino a $|x|^k$, ricordandosi k in base 2 su un nastro di lavoro, scrivendo la codifica di x (e di # finché serve);

ii) gli elementi della prima e dell'ultima colonna che sono $2 \times |x|^k$ circuiti costanti;

iii) $\left(|x|^k - 1\right) \times \left(|x|^k - 2\right)$ copie del circuito $\overline{C}$ (che dipende, ricordiamolo, solo da M e ha costo *fisso*). A ciascuna copia associamo gli indici appropriati per metterla in relazione con la casella di computazione corrispondente; tali indici sono tutti minori di $|x|^k$, e averli rappresentati in binario ci consente di manipolarli in spazio $\mathcal{O}(\log(n))$.

□

Ritorniamo adesso rapidamente sulla distinzione tra approcci hardware e software ai modelli di calcolo fatta nel capitolo 1.5. Per quelli hardware, si suggeriva di vedere ciascun algoritmo come una macchina, le cui dimensioni potevano addirittura crescere al crescere dei dati di ingresso. Tale osservazione trova giustificazione nel circuito C_I costruito nella dimostrazione precedente: al crescere del dato x cresce la tabella di computazione della macchina di Turing M e così crescono le *dimensioni* del circuito corrispondente, ma non cambiano i suoi componenti $\overline{C}$ — né cambia, per così dire, la sua forma.

Vediamo adesso una variante di CIRCUIT-VALUE, chiamato MONOTONE CIRCUIT VALUE: il problema di verificare se un circuito *senza* $\neg$ calcola il valore *tt*. Si sa bene che le espressioni booleane e quindi i circuiti con soli $\wedge, \vee$ sono meno espressivi di quelli che hanno anche il $\neg$, tuttavia *la difficoltà nel valutare gli uni e gli altri è la stessa*! Infatti non è difficile costruire una riduzione facile da CIRCUIT-VALUE a MONOTONE CIRCUIT VALUE.

Corollario 2.5.2. *MONOTONE CIRCUIT VALUE è $\mathcal{P}$-completo.*

Dimostrazione. Facciamo vedere che CIRCUIT-VALUE $\leqslant_{\text{logspace}}$ MONOTONE CIRCUIT VALUE, cioè trasformiamo un circuito qualunque (con ingressi assegnati) in uno monotono equivalente. Ciò viene fatto applicando le regole di De Morgan partendo dalle porte di uscita a scendere fino alle porte di ingresso. Le porte di ingresso *tt* rimpiazzano se necessario quelle *ff* e viceversa, e se ne aggiungono di nuove se servissero. Tutto questo si fa ovviamente in LOGSPACE: basta visitare una sola volta le porte, rappresentate come coppie (i, j), dove i e j sono indici di livello e di "colonna", rappresentati a loro volta in binario.

□

Ecco un paio di problemi $\mathcal{P}$-completi, l'uno preso dalla teoria dei linguaggi formali e l'altro dalla logica; poiché appartengono allo stesso grado, vi sono trasformazioni sia dall'uno all'altro che da e in (MONOTONE) CIRCUIT VALUE.

Esempio 2.5.3.

- $CFL \neq \emptyset$ è il problema di verificare se il linguaggio generato da una data grammatica libera da contesto[21] è vuoto, cioè:

$$\{L(G) \neq \emptyset \mid G \text{ è una grammatica libera dal contesto}\}$$

- dato un insieme di variabili X e una congiunzione di un numero finito di formule di Horn $\mathtt{H}$,[22] HORN è il problema di decidere se una variabile $x \in X$ è deducibile da $\mathtt{H}$, cioè

$$\text{HORN} = \{\mathtt{H}, x \mid \mathtt{H} \vdash x\}$$

[21]Una *grammatica libera* è una quadrupla $G = \langle N, \Sigma, S, P\rangle$ dove N è l'alfabeto dei simboli *non terminali*, Σ è l'alfabeto dei simboli *terminali*, $S \in N$ è chiamato simbolo *distinto*, $P \subseteq (N \times (N \cup \Sigma)^+)$ è l'insieme *finito* delle *produzioni*, usualmente rappresentate come $A \to \alpha$.

[22]Una formula booleana è un'espressione booleana nella forma

$$x_1 \wedge x_2 \wedge \cdots \wedge x_k \Longrightarrow y,\ y, x_i \in X,\ i \geq 0$$

Inoltre, diciamo che x è deducibile da una congiunzione di formule di Horn $\mathtt{H}$, in simboli $\mathtt{H} \vdash x$, se e solamente se $x \in \mathtt{H}$ oppure $x_1 \wedge x_2 \wedge \cdots \wedge x_k \Longrightarrow x$ e $\forall i.\ \mathtt{H} \vdash x_i$

$\mathcal{NP}$-completezza

Finalmente veniamo ai problemi intrattabili $\mathcal{NP}$ e al suo massimo campione, SAT, il problema della soddisfacibilità delle espressioni booleane.

Teorema 2.5.3 (Cook)**.** *SAT è $\mathcal{NP}$-completo.*

Dimostrazione. Sappiamo già che SAT $\in \mathcal{NP}$, perché abbiamo visto una procedura non deterministica che realizza una ricerca esaustiva, seppur implicita e lo decide: assegna *a caso* un valore di verità alle variabili dell'espressione booleana (fase che richiede tempo *non deterministico* polinomiale) e valuta l'espressione (o il circuito) booleano chiuso che si ottiene (fase che richiede tempo *deterministico* polinomiale); il problema è risolto positivamente se esiste un'espressione chiusa che valuta a tt.

Poiché CIRCUIT-SAT $\leqslant_{\text{logspace}}$ SAT per la proprietà 2.5.4, ci basta dimostrare che CIRCUIT-SAT è $\mathcal{NP}$-completo, ovvero che $\forall I \in \mathcal{NP}$ si ha che $I \leqslant_{\text{logspace}}$ CIRCUIT-SAT.

Sia allora $I \in \mathcal{NP}$. Costruiamo $f \in$ LOGSPACE tale che $x \in I$ se e solamente se $f(x)$ è soddisfacibile. Per ipotesi c'è una macchina di Turing non deterministica N che decide I in tempo n^k. In altri termini, dato x esiste una computazione di lunghezza minore o uguale a $|x|^k$ che accetta x se e solamente se $x \in I$; ora, questa computazione può essere vista come una successione di scelte non deterministiche, anch'essa di lunghezza minore o uguale a $|x|^k$. Per semplicità conveniamo che il grado di non determinismo sia *esattamente* uguale a 2, cioè che ad ogni passo vi siano sempre 2 scelte possibili.[23] Codifichiamo la prima scelta con il bit ff e la seconda con il bit tt. Una computazione è allora una successione di bit della forma

$$B = b_0 b_1 \dots b_{|x|^k - 1}, \text{ con } b_i \in \{tt, ff\}$$

Ovviamente nel caso della macchina non deterministica non abbiamo una tabella della computazione che rappresenti l'albero della computazione non-deterministica; tuttavia, se fissiamo una successione di scelte B sappiamo costruire la tabella T in funzione della macchina N, del dato x e anche della successione di scelte B. Quindi come fatto per CIRCUIT-VALUE, la prima riga e la prima e ultima colonna sono fissate. Inoltre, $T(i, j)$ dipende, oltre che da

[23] Possiamo sempre ricondurci a questa situazione: se la macchina N ha

i) solo una scelta, consideriamo due scelte coincidenti, il che richiede uno stato ausiliario;

ii) $m > 2$ scelte, si sceglie la prima o le rimanenti $m - 1$ e si ripete il procedimento $m - 2$ volte, il che richiede di aggiungere $m - 2$ nuovi stati.

$T(i-1,j-1), T(i-1,j)$ e $T(i-1,j+1)$ *anche* da b_{i-1}, cioè dalla scelta fatta al passo precedente. Quindi il circuito $\overline{C}_N$ che possiamo costruire in perfetta analogia a quanto fatto nella dimostrazione della $\mathcal{P}$-completezza di CIRCUIT VALUE (teorema 2.5.2), ha $3m+1$ ingressi e m uscite. Di nuovo possiamo costruire in LOGSPACE un circuito $f(x)$ che ha come porte di ingresso le scelte non deterministiche codificate nel vettore B e i valori della riga $T(1,j)$ ottenuti dal dato di ingresso x.

Infine, la dimostrazione che $f(x)$ è soddisfacibile se e solamente se $x \in I$ è del tutto simile a quella del teorema 2.5.2.

□

Vediamo alcuni esempi di altri problemi che sono $\mathcal{NP}$-completi, cioè tali per cui vi sono trasformazioni dall'uno all'altro e da e in SAT. La letteratura riporta migliaia di problemi di questo tipo, la gran parte dei quali è significativa dal punto di vista computazionale — quindi la classe $\mathcal{NP}$ è importante.

Esempio 2.5.4. I seguenti problemi sono $\leqslant_{\text{logspace}}$-completi per $\mathcal{NP}$.

- HAM;
- CRICCA;
- Problema del Commesso Viaggiatore;
- Programmazione Intera: dato un sistema di disequazioni a coefficienti interi in n variabili trovare una soluzione intera.[24]

[24] Si ricordi che se si tralascia la parola "intero" il problema è in $\mathcal{P}$: basta usare il metodo elissoide o quello dei punti interni.