Fondamenti di Programmazione: elementi di calcolabilità

Pierpaolo Degano
Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo, 3, I-56127 Pisa, Italia
degano@di.unipi.it

7 novembre 2006

Sommario

AVVERTENZA: queste note sono estratte dalle dispense preparate per il corso di Calcolabiltà e Complessità della laurea magistrale e raccolgono gli elementi minimi e basilari della teoria della calcolabilità che tutti gli informatici dovrebbero conoscere. Le note coprono gli argomenti svolti nei corsi di Fondamenti di programmazione nelll' anno accademico 2006/2007.

Indice

0.1 Idea intuitiva di algoritmo 2
0.2 Macchina di Turing 3
0.3 Problemi e Funzioni 10
0.4 Alcuni risultati classici 16
0.5 Problemi insolubili e riducibilità 18

0.1 Idea intuitiva di algoritmo

Ci sono moltisimi formalismi che sono stati proposti per esprimere algoritmi, tra cui ricordiamo Macchine di Turing, funzioni ricorsive, λ-calcolo, Random Access Machine, algoritmi di Markov, grammatiche, 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*;

iii) Una computazione è eseguita per *passi discreti* (singoli), senza ricorrere a sistemi analogici o metodi continui;

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, nè 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 raggiunto uno stato finale e abbiamo trovato il risultato, 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 le future.

0.2 Macchina di Turing

Introdurremo di seguito uno dei formalismi piú importanti e piú diffusi per esprimere algoritmi: le Macchine di Turing, che ricordano con straordinaria verosimiglianza i comuni elaboratori à la 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 (MARK IV, COLOSSUS o ENIAC).

Definizione 0.2.1 (Macchina di Turing)**.** Si dice Macchina di Turing (in breve MdT) una quadrupla

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

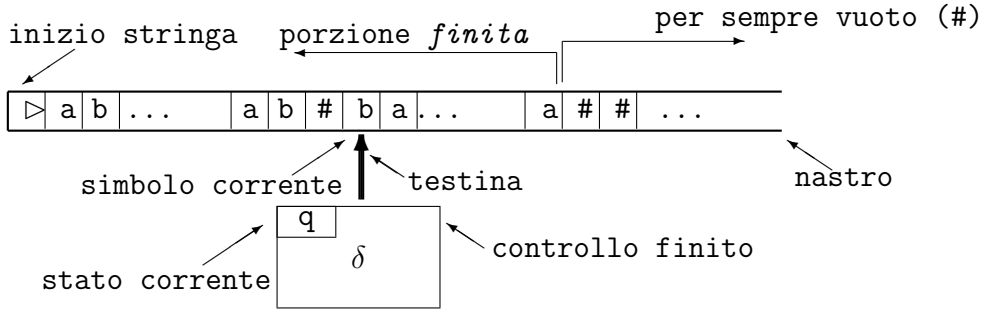
dove:

- $Q(\ni q_i)$ è l'insieme *finito* degli *stati*, con $h \notin Q$
  (con lo stato speciale h indicheremo il caso di un arresto "corretto" di un calcolo di M)
- $\Sigma(\ni \sigma)$ è l'insieme *finito* dei simboli (*alfabeto*) con $\# \in \Sigma$ (*carattere bianco*), $\rhd \in \Sigma$ (*marca di inizio stringa*) e $L, R, - \notin \Sigma$
- $q_0 \in Q$ è lo *stato iniziale*
- $\delta \subseteq (Q \times \Sigma) \times (Q \cup \{h\}) \times \Sigma \times \{L, R, -\}$ è la *relazione di transizione*. In questa parte restringeremo la relazione δ in modo che sia una funzione, imponendo che sia tale che:
  - $(q, \sigma, q', \sigma', D'), (q, \sigma, q'', \sigma'', D'') \in \delta \Rightarrow\ q' = q'', \sigma' = \sigma'', D' = D''$
    (questa condizione ci permette di scrivere $\delta(q, \sigma) = (q', \sigma', D')$ al posto della quintupla $(q, \sigma, q', \sigma', D')$)
  - se $\delta(q, \rhd) = (q', \sigma, D)$ allora $\sigma = \rhd, D = R$
    (questa condizione dice che il carattere corrente, ovvero il cursore (vedi Figura piú sotto) non può mai trovarsi a sinistra della marca di inizio stringa, $\rhd$)

È facile verificare che la condizione (i) dell'idea intuitiva di algoritmo elencata dianzi è soddisfatta, cosí 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;

maioris causa la sua funzione di transizione δ contiene un numero finito di elementi.

Graficamente, una MdT può essere rappresentata dalla figura seguente.



Esempio 0.2.2. Si consideri la Mdt $M = (Q, \Sigma, \delta, q_0)$, con $Q = \{q_0, q_1\}$ e $\Sigma = \{\#, \triangleright, a\}$ definita dalla seguente tabella:

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, quando introdurremo macchine di Turing, ometteremo per semplicità la definizione esplicita dell'insieme degli stati Q e di quello dei simboli Σ se questi possono essere dedotti dall'apparire dei loro elementi nella tabella che definisce la funzione di transizione δ.

Passiamo ora a descrivere la dinamica delle macchine di Turing, cioè il loro comportamento quando operino su una stringa di caratteri memorizzata 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 con 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" del nastro che contiene almeno la sua parte non bianca. La seguente tabella rappresenta informalmente una computazione della MdT introdotta nell'esempio 0.2.2, che si arresta con

successo in 7 passi. Nelle configurazioni, il carattere sottolineato nella parte *Nastro* sta a indicare la posizione corrente del cursore, o *testina*.

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

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

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

dove q è lo stato corrente, u è la stringa di caratteri a sinistra del simbolo corrente σ e v è quella dei caratteri a destra del simbolo corrente. Per semplicità scriveremo $(q, u\underline{\sigma}v)$ al posto di (q, u, σ, v), cosí come abbiam già fatto sopra. Poiché per ora non ci interessa considerare i simboli # a destra del simbolo $\sigma_n \neq \#$ piú a destra nel nastro (ovvero della porzione v), indichiamo con λ la stringa vuota e conveniamo che $\# \cdot \lambda = \lambda$ e $\sigma \cdot \lambda = \sigma$, se $\sigma \neq \#$. Definiamo allora

$$\Sigma^R = \Sigma^* \cdot (\Sigma \setminus \{\#\}) \cup \{\lambda\}$$

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 $i < n$).

Si noti che la stringa u può essere vuota solo quando il carattere corrente σ è la marca di inizio stringa $\rhd$, a causa della seconda condizione sulla funzione di transizione δ.

Con questa convenzione, il frammento iniziale della computazione di prima viene rappresentato come:

$$(q_0, \rhd\#\#a\#a, a, \lambda) \rightarrow (q_1, \rhd\#\#a\#, a, \lambda) \rightarrow$$
$$(q_0, \rhd, \#\#a\#, a, \lambda) \rightarrow (q_1, \rhd\#\#a, \#, \lambda) \rightarrow \ldots$$

Un altro esempio di computazione della macchina definita nell'esempio 0.2.2 è:

$$(q_0, \rhd\#, a, a) \rightarrow (q_1, \rhd, \#, \#a) \rightarrow (q_0, \lambda, \rhd, \#\#a) \rightarrow$$
$$(q_0, \rhd, \#, \#a) \rightarrow (h, \rhd, \#, \#a)$$

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 piú 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 $v = \lambda$ allora $bv = \lambda$ e infine che nel caso (iii) se $a = \triangleright$ allora anche $b = \triangleright$):

i)		$(q, u\underline{a}v) \to (q', u\underline{b}v)$	se $\delta(q, a) = (q', b, -)$
ii)		$(q, uc\underline{a}v) \to (q', u\underline{c}bv)$	se $\delta(q, a) = (q', b, L)$
iii)	(a)	$(q, u\underline{a}cv) \to (q', ub\underline{c}v)$	se $\delta(q, a) = (q', b, R)$
	(b)	$(q, u\underline{a}) \to (q', ub\underline{\#})$	se $\delta(q, a) = (q', b, R)$

Si noti che 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,[1] e da *un solo stato*, quello corrente (cf. i punti (iii) e (iv) richiesti dall'intuizione).

Una *computazione* è una successione finita di passi

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

cioè $\to^*$ è la chiusura riflessiva e transitiva di $\to$. Come al solito, se vi sono n passi, la computazione è lunga n e la scriveremo così:

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

La computazione $(q_0, w) \to^* (q', w')$ *termina* (converge, $\downarrow$) su w se $q' = h$.

Quando avremo bisogno di 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 $(q_0, \underline{\triangleright}w)$, cioè avendo come dato iniziale la stringa w, ovvero che *applicheremo* M a w.

Domanda 0.2.3. Ma c'è un limite ai passi di computazione (punto 5 della definizione intuitiva di algoritmo)?

[1] 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.

NO. Vediamolo sul seguente esempio.

Esempio 0.2.4.

Si consideri la seguente MdT:

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

Un esempio di computazione che non termina è
$(q_0, \underline{\trianglerighteq}, a\#a\#) \to (q_0, \rhd, \underline{a}\#a\#) \to^*$
$(q_0, \rhd a\#a\# \ldots \underline{\#}\#) \to \ldots$

Ovviamente, le macchine di Turing sono state introdotte per formalizzare la nozione di calcolo. Lo faremo tra poco e tuttavia negli esempi che seguono ci sentiamo liberi di usare le parole "calcolare" e "decidere".

Esempio 0.2.5.

Consideriamo la MdT che calcola $n + m$ dove n ed m sono interi positivi rappresentati in notazione unaria con il simbolo | e separati dal simbolo + (stipuliamo che il nastro iniziale abbia sempre questa forma). In notazione unaria, vi sono tanti | giustapposti quanto è il numero da rappresentare; cioè n è rappresentato da $|^n$; ad esempio $|^3 = |||$ sta per 3.

q	σ	δ
q_0	$\rhd$	$q_0, \rhd, R$
q_0	$\|$	$q_0, \|, R$
q_0	$+$	$q_1, \|, R$
q_1	$\|$	$q_1, \|, R$
q_1	$\#$	$q_2, \#, L$
q_2	$\|$	$h, \#, -$

La computazione per il calcolo di $1 + 2$ è la seguente:

$(q_0, \underline{\trianglerighteq} \mid + \mid\mid) \to (q_0, \rhd \underline{\mid} + \mid\mid) \to (q_0, \rhd \mid \underline{+} \mid\mid)$
$\to (q_1, \rhd \mid\mid \underline{\mid} \mid) \to (q_1, \rhd \mid\mid\mid \underline{\mid}) \to (q_1, \rhd \mid\mid\mid\mid \underline{\#})$
$\to (q_2, \rhd \mid\mid\mid \underline{\mid}) \to (h, \rhd \mid\mid\mid \underline{\#})$

Esempio 0.2.6.

Vediamo una macchina che, data una stringa palindroma su $\{a, b\}^*$ si arresta con il simbolo *SI* nella casella scritta più a destra e il cursore sul primo carattere del nastro bianco. Se la stringa in ingresso non è palindroma, la macchina va in stallo entrando in uno stato speciale q_N; non sarebbe difficile estendere questa macchina in modo che si arrestasse nello stato h anche in questo caso, lasciando però sul nastro carattere *NO*. Avremmo cosí rappresentato un algoritmo che *decide* se la stringa in ingresso appartiene all'insieme

delle stringhe palindrome o meno. Il lettore avrà certamente notato una leggera differenza di stile tra questa e la macchina precedente che invece *calcola* una funzione.

Come abbreviazione, nella tabella seguente che descrive la funzione di transizione δ, scriviamo solo $(q_{a/b}, \sigma, q'_{a/b}, \sigma', D)$ al posto delle due quintuple $(q_a, \sigma, q'_a, \sigma', D)$ e $(q_b, \sigma, q'_b, \sigma', D)$ (cioè $q_{a/b}$ non è uno stato singolo, ma rappresenta gli stati q_a e q_b; analogamente per $q'_{a/b}$ e per q'_a, q'_b).

q	σ	δ
q_0	$\triangleright$	$q_0, \triangleright, R$
q_0	a	$q_a, \triangleright, R$
q_0	b	$q_b, \triangleright, R$
q_0	$\#$	$q_2, \#, -$
$q_{a/b}$	a	$q_{a/b}, a, R$
$q_{a/b}$	b	$q_{a/b}, b, R$
$q_{a/b}$	$\#$	$q'_{a/b}, \#, L$
q'_a	$\triangleright$	$q_2, \triangleright, R$
q'_a	a	$q_1, \#, L$
q'_a	b	$q_N, b, -$
q'_b	$\triangleright$	$q_2, \triangleright, R$
q'_b	a	$q_N, a, -$
q'_b	b	$q_1, \#, L$
q_1	$\triangleright$	$q_0, \triangleright, R$
q_1	a	q_1, a, L
q_1	b	q_1, b, L
q_2	$\#$	h, SI, R

La sua computazione su *aba* è la seguente:

$(q_0, \underline{\triangleright} aba\#) \rightarrow (q_o, \triangleright \underline{a} ba\#) \rightarrow$
$(q_a, \triangleright \triangleright \underline{b} a\#) \rightarrow (q_a, \triangleright \triangleright b \underline{a} \#) \rightarrow$
$(q_a, \triangleright \triangleright ba\underline{\#}) \rightarrow (q'_a, \triangleright \triangleright b \underline{a} \#) \rightarrow$
$(q_1, \triangleright \triangleright \underline{b} \# \#) \rightarrow (q_1, \triangleright \underline{\triangleright} b \# \#) \rightarrow$
$(q_0, \triangleright \triangleright \underline{b} \# \#) \rightarrow (q_b, \triangleright \triangleright \triangleright \underline{\#} \#) \rightarrow$
$(q'_b, \triangleright \triangleright \underline{\triangleright} \# \#) \rightarrow (q_2, \triangleright \triangleright \triangleright \underline{\#} \#) \rightarrow$
$(h, \triangleright \triangleright \triangleright SI \underline{\#})$

La computazione di sopra si arresta con successo, perché *aba* è palindroma. Si noti che la marca di inizio stringa $\triangleright$ si sposta a destra, ma non viene mai oltrepassata. Non è difficile modificare la funzione di transizione in modo che ciò non accada: basta che il simbolo sostituito da $\triangleright$ venga semplicemente cancellato, spostando l'intera stringa residua a sinistra di una casella.

Vediamo adesso un altro esempio di computazione che si arresta con successo, perché il suo argomento *abba* è una striga palindroma. Per comprendere meglio il comportamento della macchina, abbiamo raggruppato i suoi passi in fasi, che operano in maniera "omologa". Inoltre, in questo esempio e in altri nel capitolo, abbiamo mantenuto traccia nelle configurazioni di tutte le posizioni che la macchina visita, per valutare, nella seconda parte del corso, lo spazio necessario alle computazioni di questa macchina.

$(q_0, \underline{\triangleright} abba\#) \rightarrow$
$(q_o, \triangleright \underline{a} bba\#) \rightarrow (q_a, \triangleright \triangleright \underline{b} ba\#) \rightarrow (q_a, \triangleright \triangleright b \underline{b} a\#) \rightarrow (q_a, \triangleright \triangleright bb \underline{a} \#) \rightarrow (q_a, \triangleright \triangleright bba \underline{\#}) \rightarrow$

$(q'_a, \rhd \rhd bb\underline{a}\#) \to (q_1, \rhd \rhd b\underline{b}\#\#) \to (q_1, \rhd \rhd \underline{b}b\#\#) \to (q_1, \rhd \trianglerighteq bb\#\#) \to$
$(q_0, \rhd \rhd \underline{b}b\#\#) \to (q_b, \rhd \rhd \rhd \underline{b}\#\#) \to (q_b, \rhd \rhd \rhd b\underline{\#}\#) \to$
$(q'_b, \rhd \rhd \rhd \underline{b}\#\#) \to (q_1, \rhd \rhd \trianglerighteq \#\#\#) \to (q_0, \rhd \rhd \rhd \underline{\#}\#\#) \to$
$(q_2, \rhd \rhd \rhd \underline{\#}\#\#) \to (h, \rhd \rhd \rhd \textsc{si}\underline{\#}\#)$

La seguente computazione si arresta perché la funzione δ non è definita per lo stato q_N; quindi, non lascia la macchina in una configurazione terminale e riflette il fatto che la stringa ba non è palindroma.

$(q_0, \trianglerighteq ba\#) \to (q_0, \rhd \underline{b}a\#) \to (q_b, \rhd \rhd \underline{a}\#) \to (q_b, \rhd \rhd a\underline{\#}) \to (q'_b, \rhd \rhd \underline{a}\#) \to$
$(q_N, \rhd \rhd \underline{a}\#)$

Come detto sopra, non è difficile definire delle quintuple che, a partire dallo stato q_N, cancellino la parte di nastro contenente simboli diversi dal $\#$ e dal $\rhd$, scrivano nella prima casella utile $\textsc{no}$ in modo che la macchina si arresti con successo.

Nel seguito, una situazione di arresto "anomalo", come quella appena vista verrà talvolta chiamata *stallo*.

Complichiamoci adesso un po' la vita e sostituiamo alla quintupla $\delta(q'_a, b) = (q_N, b, -)$ la quintupla $\delta(q'_a, b) = (q'_a, b, -)$ e similmente alla quintupla $\delta(q'_b, a) = (q_N, a, -)$ sostituiamo $\delta(q'_b, a) = (q'_b, a, -)$. È immediato verificare che la macchina, applicata a una stringa non palindroma, non si arresterà mai.

0.3 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.

Un esempio banale di problema è la domanda: "qual è il massimo comun divisore di x e y?" Meno interessante è uno dei casi di questo problema: "qual è il massimo comun divisore di 42 e 56?", i cui parametri x e y hanno valori numerici.

Per esempio, la MdT per la somma introdotta nell'esempio 0.2.5 calcola la funzione $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (quando applicata alla stringa iniziale $|^n + |^m$); la MdT per le stringhe palindrome dell'esempio 0.2.6 se termina con successo dice che la stringa corrispondente al dato iniziale appartiene a $\{w \in \{a, b\}^* \mid w = w^R\}$ (con w^R si intende la stringa w letta da destra a sinistra, quindi w può essere letta indifferentemente da destra a sinistra o viceversa); se va in stallo dice che non vi appartiene. Si noti tuttavia che nell'ultima variante introdotta ciò non è piú vero: la macchina non si arresta mai se la stringa non è palindroma e quindi abbiamo un algoritmo che dà risultato *solo se* il dato iniziale è una stringa palindroma. Nel primo caso, abbiamo una funzione che opera sui naturali, o meglio su una loro codifica nell'alfabeto $\{|, +, \#\}$. Più precisamente abbiamo la seguente definizione:

Definizione 0.3.1. Siano $\Sigma, \Sigma_0, \Sigma_1$ alfabeti, con $\# \notin \Sigma_0 \cup \Sigma_1$, $\Sigma_0 \cup \Sigma_1 \subset \Sigma$, 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

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

Domanda 0.3.2. 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*, ovvero funzioni biunivoche che siano *effettive e facili* [2] dei dati, nel modo seguente

[2] Preciseremo in seguito cosa intendiamo per *effettivo* e per *facile*. Lo faremo considerando solo i numeri naturali, loro coppie ecc., quindi imbroglieremo un bel po', nella speranza di essere perdonati — tanto l'esame lo fate voi!

i) Dato x in formato A codificalo come y in formato B.

ii) Applica la MdT a y e se e quando ottieni z in formato B,

iii) traduci z dal formato B al formato A.

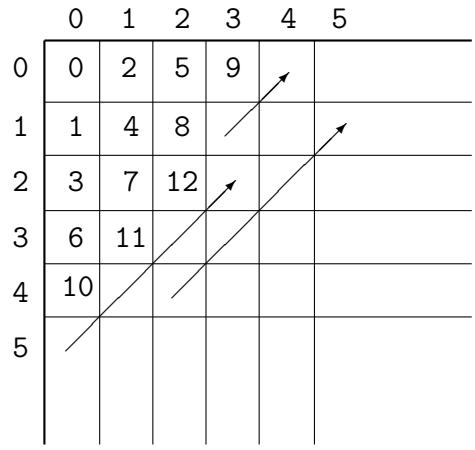
Quindi con il rischio, ma non il timore di banalizzare, considereremo da qui in avanti solo i *numeri naturali* come nostri dati

A titolo esemplificativo, vediamo adesso una codifica "classica".

Esempio 0.3.3 (Coda di colomba)**.** La seguente funzione codifica coppie di naturali come un singolo naturale

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

ed è graficamente rappresentata come segue:



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

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

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

Domanda 0.3.4. Esistono modelli di calcolo con dati più ricchi?

Certamente! ogni linguaggio di programmazione lo è (ma *NON lo sono le sue implementazioni.* Perché?). Ovviamente la presenza di dati con struttura piú ricca permette di migliorare la comprensibilità della rappresentazione delle funzioni — in termini informatici, la concisione e la leggibilità dei programmi, la loro modificabilità e molte altre proprietà interessanti dal punto di vista della produzione e dell'uso del software. Ma allora bisogna subito chiedersi se le codifiche mutino la natura dei problemi risolvibili e delle proprietà delle funzioni che consideriamo come calcolabili. Ovvero

Domanda 0.3.5. Le proprietà basilari dei formalismi e della classe delle funzioni calcolate cambiano al variare dei dati su cui esse operano?

NO. I formalismi in questione sono tutti equivalenti se sono sufficientemente potenti, cioè se sono in grado di esprimere *tutte* le funzioni calcolabili (per esempio, le implementazioni dei linguaggi di programmazione *non* lo fanno, v. sopra). Le codifiche, proprio perché sono trasformazioni "facili" di funzioni su certi dati in funzioni "equivalenti" su dati diversi, non allargano né restringono tale classe — bisognerebbe dimostrarlo, però!

Ma dobbiamo ancora metterci d'accordo sul termine *funzione.*

Definizione 0.3.6 (Funzione totale)**.** $f : A \to B$, sottoinsieme di $A \times B$ è una *funzione totale* se e solamente se

i) $\forall a \in A \ \ \exists b \in B : (a, b) \in f$, (*la funzione è definita ovunque*)

ii) $(a, b), (a, c) \in f \Rightarrow b = c$ (*unicità*)

Esempio 0.3.7. $doppio : \mathbb{N} \to \mathbb{N}$ è definita (di solito) come l'insieme delle coppie $doppio = \{(0, 0), (1, 2), (2, 4), \ldots\}$; da cui sappiamo per esempio che $doppio(5) = 10$. Qualche volta, per far prima (!?) scriviamo $doppio(n) = n + n$ oppure ancora $doppio(n) = 2 \times n$, dando per note le funzioni + o $\times$.

Vediamo adesso un esempio di funzione un po' strana, nella cui definizione si menziona la congettura di Goldbach, definita piú 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 0.3.8. $gb : \mathbb{N} \to \mathbb{N}$

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

Definizione 0.3.9 (Congettura di Goldbach)**.**
$\forall m > 1$ si ha $2m = p_1 + p_2$ con p_1, p_2 primi.

Eccovi alcuni esempi di scomposizione à la Goldbach:

$$\begin{aligned} 22 &= 17 + 5 \quad = \quad 11 + 11 \\ 24 &= 17 + 7 \\ 36 &= 17 + 19 \end{aligned}$$

Tuttavia, non tutte le funzioni sono totali. Ecco un esempio.

Esempio 0.3.10. Definiamo, come si fa usualmente, la funzione $mezzo : \mathbb{N} \to \mathbb{N}$, come l'insieme delle coppie $mezzo = \{(0,0), (2,1), (4,2), \cdots\}$ oppure in maniera piú concisa come $mezzo(n) = n/2$. Questa funzione è ovviamente parziale, in quanto è definita solo se n è pari, mentre non ha valore (in $\mathbb{N}$) se n è dispari.

Certamente la funzione *mezzo* è una funzione interessante; e di funzioni parziali ancor piú interessanti di questa se ne incontrano a bizzeffe. Inoltre, vedremo piú avanti che siamo *obbligati* ad avere funzioni che non sono ovunque definite, perciò introduciamo le funzioni parziali.

Definizione 0.3.11 (Funzione parziale)**.**
$f : A \to B$ è una funzione *parziale* se è un sottoinsieme di $A \times B$ tale che

b) $(a,b), (a,c) \in f \Rightarrow b = c$ (*unicità*), ovvero

b′) 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 *divergente* 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 una funzione iniettiva è invertibile sull'immagine.)

Abbiamo gli algoritmi, sia pure sotto forma di macchine di Turing, e abbiamo le funzioni. Allora siamo arrivati a porci una domanda cruciale:

Domanda 0.3.12. Qual è il rapporto tra funzioni e algoritmi?

Una funzione f è un insieme di coppie, cioè f associa l'argomento con il risultato (es. $doppio(n) = n + n$) 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*! Per esempio, consideriamo ancora la funzione *doppio*: esiste uno e un solo insieme $\{(0,0),(1,2),(2,4),\dots\}$.

Un algoritmo invece specifica *come si calcola* il risultato a partire dall'argomento. In altre parole un algoritmo *calcola* o *rappresenta*, in modo *finito*, una funzione.

Infine, riprendiamo la funzione $gb(n)$ dell'esempio 0.3.8. Non sappiamo a tutt'oggi se la congettura di Goldbach vale. Eppure un algoritmo per calcolarla esiste! ma non sappiamo quale esso (o meglio essi) sia: quello che da come risultato 0 o quello che da come risultato 1? [3]
Finalmente siamo arrivati al nocciolo:

- Quali sono le funzioni calcolabili? Per ora sappiamo quali sono le Turing-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 la calcoli?). Tra questi problemi non calcolabili, ce n'è di interessanti?

[3] 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.

In questa 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.

Abbiamo definito le funzioni T-calcolabili. In realtà l'approccio di Turing non è stato l'unico tentativo di caratterizzare formalmente la nozione intuitiva di funzione calcolabile. Altri formalismi sono stati proposti negli anni '30, tra cui le funzioni μ-ricorsive, il λ-calcolo, i sistemi di Post. Il fatto interessante è che si è potuto dimostrare che tutti questi formalizzano esattamente lo stesso insieme di funzioni, che chiameremo senza fare riferimento al formalismo, insieme delle funzioni calcolabili. Queste dimostrazioni di equivalenza rendono quindi molto solida l'ipotesi che non ci siano funzioni calcolabili al di fuori di quelle calcolate dalle macchine di Turing (o dai formalismi equivalenti). Pertanto possiamo, o meglio vogliamo assumere 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 facciam girare i nostri programmi, purché si abbia a disposizione memoria e tempo illimitati: ciò che possiamo calcolare *non* cambia.

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 sí, ne vedremo una interessante?

0.4 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 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.

Poiché la specifica di una MdT è costituita da un insieme *finito* di quintuple, possiamo codificare le MdT associando a ciascuna di esse un indice.

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 piú ci aggrada. Per saperne di più e avere una definizione formale delle enumerazioni effettive, si veda, ad esempio il [Rogers]. 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. L'osservazione appena fatta ci consente anche di fissare una volta per tutte un elenco di MdT e di indicare con M_i la MdT che vi appare in posizione i-ma. 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 la funzione (parziale in n variabili) che la macchina M_i calcola con $\varphi_i^{(n)}$ e chiameremo i *indice* (non della funzione, bensí della macchina!).

Nota Bene: I risultati che riportiamo nel seguito sono tutti *invarianti* rispetto all'enumerazione scelta.

Cominciamo con un semplice risultato sulla cardinalità [4] dell'insieme delle funzioni calcolabili, da cui segue che vi sono funzioni *non calcolabili*.

Teorema 0.4.1.

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

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 dal nastro vuoto scrivono $|^i$ e si arrestano (sono le funzioni costanti). Che non siano piú di $\#(\mathbb{N})$ segue dal fatto che le MdT si possono enumerare, ovvero ad esse si puó associare un indice come appena discusso.

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 $2^{\#(\mathbb{N})}$. □

Il seguente teorema 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.

Teorema 0.4.2. *Ogni funzione calcolabile φ_x ha $\#(\mathbb{N})$ indici, e $\forall x$ si può costruire, mediante una funzione calcolabile totale 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 $(q_{k+2}, \#, q_{k+2}, \#, -)$, ... □

0.5 Problemi insolubili e riducibilità

In questo capitolo studieremo i problemi di appartenenza o di non appartenenza di un elemento ad un dato insieme, affrontando cosí 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 prima la definizione di insieme ricorsivo:
I è *ricorsivo* (ovvero *decidibile*) 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.

Un esempio di insieme ricorsivo è costituito da quei numeri che soddisfano la condizione di Goldbach, ovvero quelli che sono somma di due numeri primi: $\{n = p + q \mid p, q \text{ primi}\}$. La funzione caratteristica di tale insieme può esser calcolata sommando tra loro tutti i primi minori o uguali a $\frac{n}{2}$, ma si può far meglio ...

Vediamo che razza di insiemi si possono definire quando limitiano il numero di passi nel calcolo di una MdT (e le dimensioni di ingressi e uscite). Scriviamo

$\varphi_{i,s}(x) = y$ sse
1) $i, x, y < s$ e
2) $\varphi_i(x) = y$ è calcolato dalla macchina M_i in
n passi, con n¡s.

Si noti che $\varphi_i(x) \downarrow$ se esiste s *qualsiasi* tale che $\varphi_{i,s}(x) = y$. Non è difficile verificare le seguenti proprietà.

Proprietà 0.5.1. *Dati* $s, z \in \mathbb{N}$,
$\{(i, x, s) \mid \exists y.\ \varphi_{i,s}(x) = y\}$ *è ricorsivo.*
$\{(i, x, z, s) \mid \varphi_{i,s}(x) = z\}$ *è ricorsivo.*

Dimostrazione. Ovvia: fai andare la MdT M_i su x per $s - 1$ passi: se nel frattempo termina (e vale z) poni $\chi(i, x, s) = 1$, altrimenti poni $\chi(i, x, s) = 0$. □

È interessante notare che la proprietà appena enunciata continua a valere anche se sostituissimo alla costante s una funzione calcolabile f, che dipende

dall'indice i o dal dato x o da entrambi. Adesso vediamo quali insiemi vengono fuori se non poniamo alcun limite al numero dei passi s. 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 piú ampia di quella degli insiemi ricorsivi.

Definizione 0.5.2. Diciamo che un insieme I è *ricorsivamente enumerabile* (oppure *semi-decidibile*) se è il dominio di φ_i, funzione calcolabile (*parziale*), detta funzione semi-caratteristica di I.

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

Proprietà 0.5.3.

i) se I è ricorsivo allora I è ricorsivamente enumerabile

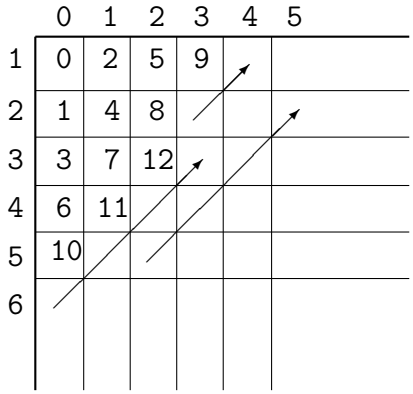
ii) $I, \bar{I}$ sono ricorsivamente enumerabili se e solo se I (e $\bar{I}$) sono ricorsivi.

Dimostrazione. Il caso (i) ovvio.
(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 semi-caratteristiche di I e di $\bar{I}$. 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 0.5.4. *I è ricorsivamente enumerabile se e solamente se è vuoto oppure è l'immagine di una funzione calcolabile* totale.

Dimostrazione. La parte *solo-se* è facile; quella piú complicata riguarda il caso in cui $I = dom(\varphi_i)$ sia non vuoto e consiste nella costruzione di una funzione 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, come rappresentato dalla seguente figura, 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.



Piú precisamente, si eseguono $\langle m, n\rangle$ passi passi nel calcolo di $\varphi_i(n)$ (ovvero tanti quanti vale la codifica della coppia (m, 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 si inizia un secondo procedimento a coda di colomba eseguendo $\varphi_i(n)$ per m passi: se tale calcolo si arresta, allora poni $f(\langle m, n\rangle) = n$,

altrimenti poni $f(\langle m, n\rangle) = \bar{n}$ (che ovviamente appartiene a I); itera il procedimento incrementando la codifica $\langle m, n\rangle$, ovvero considerando $\langle m, n\rangle + 1$. Si generano cosí tutti gli elementi di I. □

Adesso vediamo un insieme veramente speciale e paradigmatico:

$$\boxed{K = \{x | \varphi_x(x) \downarrow\}}$$

Proprietà 0.5.5. *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.

Se vi piace di più K è il dominio di θ con $\theta(x) = \varphi_z(x, x)$ dove φ_z è l'algoritmo (o la macchina) universale del teorema d'enumerazione. □

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

Proprietà 0.5.6. *K non è ricorsivo.*

Dimostrazione. Per assurdo sia χ_K la funzione caratteristica (totale e calcolabile) di K. Ma allora anche la seguente funzione f

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

sarebbe calcolabile totale. 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 0.5.3(ii), in quanto K è ricorsivamente enumerabile per la proprietà 0.5.5. Abbiamo cosí 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.

Finalmente siamo arrivati al problema, che di solito si chiama *problema della fermata*, e che confuta l'affermazione di Hilbert che i problemi matematici hanno una caratterizzazione esatta. A costo di essere noioso, val la pena di ripetere che, come tutti i risultati sulla calcolabilità, 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.

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, sistemi e delle altre diavolerie infernatiche che ci stanno a cuore. 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 piú 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

$$K_0 = \{(x, y) \mid \varphi_y(x) \downarrow\}$$

Corollario 0.5.7. *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. □