Funzioni calcolabili e linguaggi decidibili

Indice

Inizieremo questa dispensa parlando di *linguaggio*, correlato ad una macchina di Turing di tipo riconoscitore, ed introducendo i concetti di accettabilità e decidibiltà di un linguaggio. Subito dopo, nel Paragrafo 3.2, studieremo il concetto di calcolabilità di una funzione, correlato al concetto di macchina di Turing di tipo trasduttore. Successivamente, nel Paragrafo 3.3 enunceremo la tesi di Church-Turing, che svincola il concetto di calcolabilità dal particolare modello di calcolo utilizzato e, presentando un linguaggio di programmazione dotato di un insieme minimale di istruzioni, dimostreremo, a sotegno della tesi di Church-Turing, la sua equivalenza con il modello di calcolo delle macchine di Turing. Utilizzeremo, poi, il linguaggio di programmazione appena descritto nel Paragrafo 3.4 presentando un algoritmo che simula il comportamento di una macchina di Turing non deterministica mediante la tecnica della coda di rondine con ripetizioni. Infine, nel Paragrafo 3.5 introdurremo un modello di calcolo puramente teorico che non rispetta le specifiche entro le quali la tesi di Church-Turing è valida e che risulta strettamente più potente del modello di calcolo macchina di Turing.

3.1 Linguaggi e macchine di Turing

Come sappiamo, una macchina di Turing di tipo riconoscitore calcola una funzione booleana. Più in particolare, dato un input $x \in \Sigma^*$, verifica se x soddisfa una certa proprietà, o predicato, $\pi(\cdot)$. Se indichiamo con $o_T(x)$ l'esito della computazione $T(x)$, ovvero, lo stato raggiunto dalla computazione $T(x)$ nel caso essa termini, allora possiamo dire che

$$o_T(x) = q_A \Leftrightarrow \pi(x)$$

o, equivalentemente,

$$\{x \in \Sigma^* : o_T(x) = q_A\} = \{x \in \Sigma^* : \pi(x)\}.$$

Quindi, possiamo parlare dell'*insieme delle parole accettate da una macchina di Turing di tipo riconoscitore*.

Proviamo ora a ribaltare il punto di vista, e concentriamo la nostra attenzione su un qualche sottoinsieme di Σ^*.

Definizione 3.1: Un *linguaggio* L è un sottoinsieme di Σ^*: $L \subseteq \Sigma^*$.

Definizione 3.2: Il *linguaggio complemento* L^c di un linguaggio $L \subseteq \Sigma^*$ è l'insieme delle parole non contenute in L: $L^c = \Sigma^* - L$.

Dato un qualsiasi linguaggio $L \subseteq \Sigma^*$, non è, in generale, banale progettare una macchina di Turing che accetti tutte e sole le parole di L. Anzi, in generale, non è nemmeno detto che una tale macchina di Turing esista.

Definizione 3.3: Un linguaggio $L \subseteq \Sigma^*$ è *accettabile* se esiste una macchina di Turing T tale che

$$\forall x \in \Sigma^* \; [\; o_T(x) = q_A \;\Leftrightarrow\; x \in L \;].$$

La macchina T è detta *accettare* L. Osserviamo che la definizione di accettabilità non fornisce alcuna indicazione circa l'esito della computazione $T(x)$ nel caso in cui $x \notin L$: se un linguaggio L è accettato da una macchina T e $x \notin L$ potrebbe accadere che $o_T(x) = q_R$ oppure che $T(X)$ non raggiunga mai uno stato finale e, dunque, non termini.
In altre parole, la accettabilità di un linguaggio L non dà alcuna indicazione circa la accettabilità di L^c.

Per vincolare alla terminazione il comportamento di tutte le computazioni, anche di quelle con un input non appartenente ad un dato linguaggio, è necessaria una ulteriore definizione.

Definizione 3.4: Un linguaggio $L \subseteq \Sigma^*$ è *decidibile* se esiste una macchina di Turing T che *termina per ogni input* $x \in \Sigma^*$ e tale che

$$\forall x \in \Sigma^* \; [\; o_T(x) = q_A \;\Leftrightarrow\; x \in L \;].$$

La macchina T è detta *decidere* L. Osserviamo esplicitamente che, se una macchina T decide un linguaggio $L \in \Sigma^*$,

allora

$$o_T(x) = \begin{cases} q_A & \text{se } x \in L, \\ q_R & \text{se } x \in L^c. \end{cases}$$

Il prossimo teorema si occupa della relazione fra la accettabilità di un linguaggio e quella del suo complemento.

Teorema 3.1: *Un linguaggio $l \subseteq \Sigma^*$ è decidibile se e soltanto se L e L^c sono accettabili.*

Dimostrazione: Se L è decidibile, allora esiste una macchina di Turing che, per ogni $x \in \Sigma^*$, con input x termina nello stato q_A se $x \in L$ e termina nello stato q_R se $x \in L^c$.
Dunque, $o_T(x) = q_A$ se e soltanto se $x \in L$, ossia, T accetta L.
Deriviamo, ora, da T una nuova macchina T'. Gli stati di T' sono gli stati di T più due ulteriori stati, q'_A e q'_R; gli stati finali di T' sono q'_A (accettazione) e q'_R (rigetto). Le quintuple di T' sono le stesse quintuple di T più le due ulterionri quintuple seguenti:

$$\langle q_A, u, u, q'_R, ferma \rangle \qquad \langle q_R, u, u, q'_A, ferma \rangle \qquad \forall u \in \Sigma \cup \{\Box\}.$$

Quindi, per ogni $x \in \Sigma^*$, la computazione $T'(x)$ coincide con la computazione $T(x)$ tranne che per l'utima istruzione eseguita da $T'(x)$: se $T(x)$ termina in q_A allora $T'(x)$ esegue una ulteriore istruzione che la porta nello stato di rigetto, mentre se $T(x)$ termina in q_R allora $T'(x)$ esegue una ulteriore istruzione che la porta nello stato di accettazione. Allora, T' accetta x se e soltanto se t rigetta x, ossia, se e soltanto se $x \in L^c$. E, quindi, T' accetta L^c.

Viceversa, se L è accettabile e L^c è accettabile, allora esistono due macchine di Turing, T_1 e T_2 tali che, per ogni $x \in \Sigma^*$, $T_1(x)$ accetta se e soltanto se $x \in L$ e $T_2(x)$ accetta se e soltanto se $x \in L^c$. Ricordiamo che non è specificato l'esito della computazione $T_1(x)$ per $x \in L^c$ né quello di $T_2(x)$ per $x \in L$. Componendo opportunamente T_1 e T_2, definiamo, ora, una nuova macchina T che, simulando le computazioni eseguite da T_1 e da T_2 su input $x \in \Sigma^*$, decide L. Osserviamo esplicitamente che, se T simulasse prima l'intera computazione di T_1 e poi l'intera computazione di T_2, non vi sarebbe garanzia di terminazione. Pertanto, la simulazione delle due macchine deve avvenire in modo diverso.
T dispone di due nastri, su ciascuno dei quali è inizialmente scritto l'input x; la computazione $T(x)$ avviene alternando sui due nastri singole istruzioni di T_1 e di T_2, nel modo di seguito descritto.

1) Esegui *una singola istruzione* di T_1 sul nastro 1: se T_1 entra nello stato di accettazione allora T accetta, altrimenti esegui il passo 2).

2) Esegui *una singola istruzione* di T_2 sul nastro 2: se T_2 entra nello stato di accettazione allora T rigetta, altrimenti esegui il passo 1).

Sia ora $x \in \Sigma^*$. Se $x \in L$, allora prima o poi, al passo 1) T_1 entrerà nello stato di accettazione, portando T ad accettare. Viceversa, se $x \in L^c$, allora prima o poi, al passo 2) T_2 entrerà nello stato di accettazione, portando T a rigettare.
Quindi, T decide L. □

Appare naturale, a questo punto, domandarsi se le definizioni di accettabilità e di decidibilità di un linguaggio siano realmente significative, ovvero:

- esistono davvero linguaggi non accettabili oppure non decidibili?
- Le due definizioni sono realmente distinte? Ossia, non potrebbe accadere che l'insieme dei linguaggi decidibili coincida con l'insieme dei linguaggi accettabili?
- Le due definizioni sono realmente significative? Ossia, non potrebbe esistere un modello di calcolo *fisicamente realizzabile* (così come la macchina di Turing) in grado di decidere linguaggi non decisi da macchine di Turing?

La maggior parte di questo dispensa è dedicata a fornire una risposta all'ultimo quesito. Ai primi due sono, invece, dedicate le successive due dispense.

3.2 Funzioni calcolabili e linguaggi decidibili

In questo paragrafo torniamo a considerare macchine di Turing di tipo trasduttore. In quanto segue, assumeremo che le macchine di tipo trasduttore cui faremo riferimento siano dotate di un nastro output, ossia, di un nastro che, al temine di ogni computazione che termina, contiene il valore di output.

Ricordiamo che, una *funzione parziale* da un dato insieme A ad un insieme B è una qualunque funzione $f : A \to B$. Se il dominio di f coincide con A, f si dice *funzione totale*.

Dato un qualsiasi alfabeto finito Σ, indichiamo con Σ^* l'insieme delle *parole* su Σ, ossia, l'insieme delle stringhe di lunghezza finita costituite da caratteri in Σ.

Definizione 3.5: Siano Σ e Σ_1 due alfabeti finiti; una funzione (parziale) $f : \Sigma^* \to \Sigma_1^*$ è una funzione *calcolabile* se esiste una macchina di Turing T di tipo trasduttore che, dato in input $x \in \Sigma^*$, termina con la stringa $f(x)$ scritta sul nastro output se e soltanto se $f(x)$ è definita.

Osserviamo che la precedente definizione non chiarisce il comportamento della macchina di Turing nel caso in cui il suo input non appartenga al dominio della funzione. Questa ambiguità nella definizione è simile ad una analoga ambiguità nella definizione di accettabilità di un linguaggio.

Ci proponiamo ora di studiare le relazioni che intercorrono fra accettabilità e decidibilità di linguaggi e calcolabilità di funzioni. A questo scopo, dobbiamo associare una opportuna funzione a ciascun linguaggio, e viceversa.

Sia Σ un alfabeto finito ed $L \subseteq \Sigma^*$ un linguaggio. La *funzione caratteristica* $\chi_L : \Sigma^* \to \{0,1\}$ di L è una funzione totale tale che, per ogni $x \in \Sigma^*$,

$$\chi_L(x) = \begin{cases} 1 & \text{se } x \in L \\ \\ 0 & \text{se } x \notin L. \end{cases}$$

Teorema 3.2: *Un linguaggio L è decidibile se e soltanto se la funzione χ_L è calcolabile.*

Dimostrazione: Supponiamo che $L \subseteq \Sigma^*$ sia decidibile: allora, esiste una macchina di Turing di tipo riconoscitore T, con stato di accettazione q_A e stato di rigetto q_R, tale che

$$o_T(x) = \begin{cases} q_A & \text{se } x \in L \\ \\ q_R & \text{se } x \notin L. \end{cases}$$

Senza perdita di generalità, supponiamo che T utilizzi un unico nastro. A partire da T, definiamo una macchina di Turing di tipo trasduttore T', a 2 nastri, che, con input $x \in \Sigma^*$, opera nella maniera seguente:

1) sul primo nastro, sul quale è scritto l'input x, esegue la computazione $T(x)$;

2) se $T(x)$ termina nello stato q_a allora scrive sul nastro di output il valore 1, altrimenti scrive il valore 0. Successivamente, termina.

Osserviamo innanzi tutto che, poiché L è decidibile, il passo 1) sopra termina per ogni input x. Se $x \in L$, allora $T(x)$ termina nello stato di accettazione e, quindi, al passo 2) $T'(x)$ scrive 1 sul nastro output. Se, invece, $x \notin L$, allora $T(x)$ non termina nello stato di accettazione e, quindi, al passo 2) $T'(x)$ scrive 0 sul nastro output. Questo dimostra che χ_L è calcolabile.

Supponiamo, ora, che χ_L sia calcolabile. Osserviamo che, per costruzione, χ_L è una funzione totale. Allora, esiste una macchina di Turing T di tipo trasduttore che, per ogni $x \in \Sigma^*$, calcola $\chi_L(x)$. A partire da T, definiamo una macchina di Turing di tipo riconoscitore T' a due nastri che, con input $x \in \Sigma^*$, opera nella maniera seguente:

1) sul primo nastro, sul quale è scritto l'input x, esegue la computazione $T(x)$, scrivendo il risultato sul secondo nastro;

2) se sul secondo nastro è stato scritto 1 allora la computazione $T'(x)$ termina nello stato di accettazione, altrimenti termina nello stato di rigetto.

Osserviamo innanzi tutto che, poiché χ_L è totale, il passo 1) sopra termina per ogni input x. Se $\chi_L(x) = 1$, allora il passo 1) termina scrivendo 1 sul secondo nastro e, quindi, al passo 2) $T'(x)$ termina nello stato di accettazione. Se, invece, $\chi_L(x) = 0$, allora il passo 1) termina scrivendo 0 sul secondo nastro e, quindi, al passo 2) $T'(x)$ termina nello stato di rigetto. Questo dimostra che L è decidibile. □

Il Teorema 3.2 mostra come ad ogni linguaggio decidibile corrisponda una funzione (totale) calcolabile. Quindi, il concetto di linguaggio decidibile può essere sostituito dal concetto di funzione caratteristica calcolabile. Ci chiediamo ora se sia vero anche l'inverso. Per rispondere a tale domanda, associamo ad ogni funzione $f : \Sigma^* \to \Sigma_1^*$ il linguaggio

$$L_f = \{\langle x, y\rangle : x \in \Sigma^* \ \wedge \ y \in \Sigma_1^* \ \wedge y = f(x)\}.$$

Osserviamo che sussiste una sostanziale asimmetria fra i concetti di decidibilità di un linguaggio e di calcolabilità di una funzione: mentre è definito il comportamento di una macchina di Turing che decide un linguaggio decidibile $L \subseteq \Sigma^*$ *per ogni possibile input* $x \in \Sigma^*$, il comportamento di una macchina di Turing che calcola una funzione calcolabile $f : \Sigma^* \to \Sigma_1^*$ è non definito ogni volta che il suo il suo input non appartiene al dominio della funzione. Nel caso in cui l'input x non appartenga al dominio di f, in effetti, la definizione di funzione calcolabile non chiarisce nemmeno se la macchina non termini oppure termini con un valore scritto sul nastro output che non ha alcuna relazione con f. Per queste ragioni, al fine di garantire la decidibilità di L_f, è necessario vincolare la funzione f ad una ulteriore ipotesi, quella di essere una funzione totale. Tale concetto è illustrato nel seguente teorema.

Teorema 3.3: *Se la funzione $f : \Sigma^* \to \Sigma_1^*$ è totale e calcolabile allora il linguaggio $L_f \subseteq \Sigma^* \times \Sigma_1^*$ è decidibile.*

Dimostrazione: Poiché f è calcolabile e totale, allora esiste una macchina di Turing T di tipo trasduttore che, per ogni $x \in \Sigma^*$, calcola $f(x)$. A partire da T, definiamo una macchina di Turing di tipo riconoscitore T' a due nastri che, con input $\langle x, y\rangle$, con $x \in \Sigma^*$ e $y \in \Sigma_1^*$, opera nella maniera seguente:

1) sul primo nastro è scritto l'input $\langle x, y\rangle$;

2) sul secondo nastro esegue la computazione $T(x)$, scrivendovi il risultato z;

3) esegue un confronto fra z e y: se $z = y$ allora la computazione $T'(x)$ termina nello stato di accettazione, altrimenti termina nello stato di rigetto.

Osserviamo innanzi tutto che, poiché f è totale, il passo 2) sopra termina per ogni input x. Se $f(x) = y$, allora il passo 2) termina scrivendo y sul secondo nastro e, quindi, al passo 3) $T'(x)$ termina nello stato di accettazione. Se, invece, $f(x) = z \neq y$, allora il passo 2) termina scrivendo z sul secondo nastro e, quindi, al passo 3) $T'(x)$ termina nello stato di rigetto. Questo dimostra che L_f è decidibile. □

Il Teorema 3.3 può essere invertito solo parzialmente, ossia, dalla decidibilità di L_f possiamo dedurre la sola calcolabilità di f, come dimostrato nel seguente teorema.

Teorema 3.4: *Sia $f : \Sigma^* \to \Sigma_1^*$ una funzione. Se il linguaggio $L_f \subseteq \Sigma^* \times \Sigma_1^*$ è decidibile allora f è calcolabile.*

Dimostrazione: Poiché $L_f \subseteq \Sigma^* \times \Sigma_1^*$ è decidibile, esiste una macchina di Turing di tipo riconoscitore T, con stato di accettazione q_A e stato di rigetto q_R, tale che, per ogni $x \in \Sigma^*$ e per ogni $y \in \Sigma_1^*$,

$$o_T(x, y) = \begin{cases} q_A & \text{se } y = f(x) \\ q_R & \text{altrimenti.} \end{cases}$$

Senza perdita di generalità, supponiamo che T utilizzi un unico nastro. A partire da T, definiamo una macchina di Turing di tipo trasduttore T', a 4 nastri, che, con input $x \in \Sigma^*$ sul primo nastro, opera nella maniera seguente:

1) scrive il valore $i = 0$ sul primo nastro;

2) enumera tutte le stringhe $y \in \Sigma_1^*$ la cui lunghezza è pari al valore scritto sul primo nastro, simulando per ciascuna di esse le computazione $T(x,y)$; in altri termini, opera come segue:

   (a) sia y la prima stringa di lunghezza i non ancora enumerata; allora, scrive y sul secondo nastro;

   (b) sul terzo nastro, esegue la computazione $T(x,y)$;

   (c) se $T(x,y)$ termina nello stato q_a allora scrive sul nastro di output la stringa y e termina, altrimenti, eventualmente incrementando il valore i scritto sul primo nastro se y era l'ultima stringa di lunghezza i, torna al passo 2).

Osserviamo innanzi tutto che, poiché L_f è decidibile, il passo (b) sopra termina per ogni input $\langle x,y\rangle$. Se x appartiene al dominio di f, allora esiste $\overline{y} \in \Sigma_1^*$ tale che $\overline{y} = f(x)$ e, quindi, $\langle x,\overline{y}\rangle \in L_f$. Allora, in questo caso, prima o poi (ma, comunque, in tempo finito) la stringa $\overline{y}$ verrà scritta sul secondo nastro e la computazione $T(x,\overline{y})$ terminerà nello stato di accettazione e, quindi, al passo (c) $T'(x)$ scriverà $\overline{y}$ sul nastro output e terminerà. Questo dimostra che L_f è calcolabile. □

Notiamo esplicitamente che, nella dimostrazione del Teorema 3.4, se x non appartiene al dominio di f, allora nessuna stringa y generata al passo (b) consente a $T(x,y)$ di terminare nello stato di accettazione e, quindi, la computazione $T'(x)$ non termina. Pertanto, l'ipotesi di decidibilità di L_f non consente di affermare che f sia totale.

3.3 La tesi di Church-Turing ed un nuovo modello di calcolo

Nel precedente paragrafo abbiamo parlato di calcolabilità di funzioni e di decidibilità di linguaggi sempre in relazione alle macchine di Turing. Potremmo, a questo punto, domandarci se, utilizzando modelli di calcolo più potenti non possa essere possibile calcolare anche funzioni non calcolabili dalla semplice macchina di Turing. In effetti, nella teoria della calcolabilità è stata proposta un'ampia varietà di modelli di calcolo. A titolo di esempio, ricordiamo

- le grammatiche di tipo 0
- il modello di Kleene basato sulle equazioni funzionali,
- il λ-calcolo di Church
- la logica combinatoria
- gli algoritmi normali di Markov
- i sistemi combinatori di Post
- le macchine a registri elementari
- i comuni linguaggi di programmazione.

Per quanto riguarda il potere computazionale di *tutti* i modelli di calcolo elencati sopra, è stato dimostrato che essi sono Turing-equivalenti, ossia, qualunque funzione calcolabile mediante uno di tali modelli è calcolabile anche mediante una macchina di Turing.

Questa osservazione ha condotto i due logici Alonzo Church ed Alan Turing ad enunciare la tesi seguente: se la soluzione di un problema può essere descritta da una serie finita di passi elementari, allora esiste una macchina di Turing in grado di calcolarlo. O, in altri termini,

è calcolabile tutto (e solo) ciò che può essere calcolato da una macchina di Turing.

La tesi di Church afferma che non esiste un formalismo più potente della macchina di Turing in termini computazionali. Quindi, tutto ciò che non è calcolabile da una macchina di Turing non può essere calcolato da qualche altro formalismo. La tesi di Church-Turing è ormai universalmente accettata, ma tuttora non esiste per essa una dimostrazione fomale.

Osserviamo che, comunque, sono stati definiti esempi di modelli di calcolo meno potenti delle macchine di Turing. Ricordiamo, fra essi, le grammatiche regolari, gli automi a stati finiti e le macchine che terminano sempre.

Nel seguito di questo corso ci riferiremo più frequentemente a programmi scritti in uno specifico linguaggio che non a macchine di Turing. In particolare, considereremo il linguaggio di programmazione che utilizza, oltre al concetto di variabile e di collezioni di variabili (che, nel seguito, denoteremo sempre con il nome generico di array), le istruzioni seguenti:

- l'istruzione di assegnazione "$x \leftarrow y$;" ove y può essere una variabile, un valore costante, oppure una espressione (con operatori aritmetici, o booleani, oppure insiemistici, o di concatenazione fra stringhe, dipendentemente dai casi) contenente variabili e valori costanti;
- l'istruzione condizionale "**if** (condizione) **then begin** ⟨sequenza di istruzioni⟩ **end else begin** ⟨sequenza di istruzioni⟩ **end**" in cui la parte **else** può essere assente;
- l'istruzione di loop "**while** (condizione) **do begin** ⟨sequenza di istruzioni⟩ **end**";
- l'istruzione di output, che deve essere l'ultima istruzione del programma e che consente di comunicare all'esterno il valore di una variabile oppure un valore costante: "**Output:** ⟨nomeDiVariabile⟩", oppure, **Output:** ⟨valoreCostante ⟩.

Assumiamo anche che, se le sequenze di istruzioni sono istruzioni singole, si possa omettere il **begin**-**end**. Assumiamo, infine, che l'input venga comunicato al programma prima che le istruzioni del programma abbiano inizio mediante una sorta di istruzione denotata dalla parola "**Input:**".

Nel seguito denotermo il linguaggio appena descritto con il termine **PascalMinimo**. Per poterlo usare in sostituzione delle macchine di Turing, dobbiamo dimostrare che il potere computazionale del linguaggio **PascalMinimo** non differisce da quello delle macchine di Turing.

Iniziamo dimostrando (o meglio, accennando alla dimostrazione) che la tesi di Church-Turing rimane valida per il nostro linguaggio.

Teorema 3.5: *Per ogni programma scritto in accordo con il linguaggio di programmazione* **PascalMinimo***, esiste una macchina di Turing T di tipo trasduttore che scrive sul nastro di output lo stesso valore fornito in output dal programma.*

Schema della dimostrazione: Sia $\mathscr{P}$ un programma scritto in accordo con le regole del linguaggio di programmazione **PascalMinimo**. Allora, la macchina di Turing T è definita in accordo a quanto di seguito descritto.

- T utilizza, oltre ai nastri input ed output, un nastro per ciascuna variabile e per ciascun valore costante che compare in una condizione. Ad esempio, se in $\mathscr{P}$ compare l'istruzione **if** $(a = 2)$ **then** $b = 1$, allora T utilizza una nastro per la variabile a, un nastro per la variabile b, un nastro per il valore costante 2 e un nastro per il valore costante 1. I contenuti dei nastri corrispondenti a valori costanti non vengono mai modificati durante le computazioni di T. Infine, T utilizza un nastro di lavoro per la valutazione delle espressioni e delle condizioni contenute nel programma.
- I contenuti dei nastri sono codificati in binario.
- Ad ogni istruzione di assegnazione in $\mathscr{P}$ corrisponde uno stato q_i, $i > 0$, in T.
- Ad ogni istruzione condizionale in $\mathscr{P}$ corrisponde uno stato q_i oppure una coppia di stati q_i, q_j, $i, j > 0$, in T, rispettivamente, nel caso in cui sia assente oppure presente la parte **else**.
- Ad ogni istruzione di loop in $\mathscr{P}$ corrisponde uno stato q_i, $i > 0$, in T.
- Lo stato iniziale di T è q_0.

Illustriamo, ora, come costruire una quintupla a partire da una istruzione in $\mathscr{P}$. A questo scopo, per semplicità, assumiamo di scrivere una singola istruzione in ciascuna linea di codice, come nel programma esempio in Tabella 3.1, in modo tale da avere una corrispondenza fra linee di codice e stati della macchina (ovviamente, alle linee contenenti **end** non corrisponde alcuno stato in T).

Input:	coppia di interi memorizzati nelle variabili n e m
1	$k \leftarrow 2$;
2	**if** $(n > m)$ **then**
3	$\quad p \leftarrow n$;
4	**else**
5	$\quad p \leftarrow m$;
6	**while** $(p \geq 2)$ **do begin**
7	$\quad p \leftarrow p - k$;
8	**end**
9	**Output:** p

Tabella 3.1: Algoritmo che calcola se il massimo fra due interi è pari o dispari.

1. Ad ogni assegnazione di un valore costante oppure di un valore variabile ad una variabile (rispettivamente, linea 1 e linee 3 e 5 in Tabella 3.1) corrisponde una copia del nastro corrispondente alla costante o alla variabile che compare a destra dell'assegnazione sul nastro corrispondente alla variabile che compare a sinistra dell'assegnazione; tale sequenza è indipendente da ciò che viene letto sugli altri nastri. La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita (nell'esempio, prima di eseguire l'istruzione alla linea 1 la macchina si trova nello stato q_0 e dopo averla eseguita entra nello stato q_1).

2. Ad ogni assegnazione di una espressione ad una variabile (linea 7 in Tabella 3.1) corrisponde una sequenza di quintuple che eseguono quella espressione (analoghe alle quintuple che eseguono la somma di due valori descritte nella Dispensa 1) utilizzando il nastro di lavoro e che terminano con una scrittura sul nastro corrispondente alla variabile assegnanda del valore calcolato. La sequenza termina con la macchina che entra nello stato corrispondente all'istruzione eseguita.

3. Ogni condizione viene valutata, utilizzando il nastro di lavoro, confrontando i contenuti dei due nastri interessati: ad esempio, nella linea 2 in Tabella 3.1, vengono confrontati i contenuti dei nastri corrispondenti alle variabili n ed m, mentre nella linea 6 vengono confrontati i contenuti dei nastri corrispondenti alla variabile p ed al valore 2. Dopo aver valutato la condizione, la macchina entra in un nuovo stato che dipende, oltre che dal valore della condizione, dal tipo di istruzione in cui la condizione è utilizzata.

   (a) In una istruzione **if-then-else**, se la condizione è vera allora la macchina entra in uno stato che permette di eseguire le istruzioni della parte **if**, se la condizione è falsa allora la macchina entra in uno stato che permette di eseguire le istruzioni della parte **else**. Ad esempio, possiamo trascrivere le linee 2-5 del codice in Tabella 3.1 nell'insieme di pseudo-quintuple che segue, nelle quali viene mostrato il solo contenuto dei nastri di interesse in quelle istruzioni, il confronto $n > m$ non viene trascritto in una sequenza di contolli di coppie di caratteri binari sui nastri n ed m, e non viene specificato il movimento delle testine:

   $$\langle q_1, n < m, (\ldots, n, \ldots, m, \ldots), q_2, \cdot \rangle$$

   $$\langle q_1, n \geq m, (\ldots, n, \ldots, m, \ldots), q_3, \cdot \rangle$$

   $$\langle q_2, p \leftarrow n, (\ldots), q_4, \cdot \rangle$$

   $$\langle q_3, p \leftarrow m, (\ldots), q_4, \cdot \rangle.$$

   (b) In una istruzione **while**, se la condizione è vera allora la macchina entra in uno stato che permette di eseguire la prima istruzione del corpo del loop, altrimenti entra in uno stato che permette di eseguire la prima istruzione successiva a quelle che costituiscono il corpo del loop. Una volta eseguita l'ultima istruzione del corpo del loop, la macchina entra nuovamente nello stato in cui testa la condizione del loop. Di nuovo, a titolo di esempio, trascriviamo in pseudo-quintuple le linee 6-8 del codice in Tabella 3.1:

   $$\langle q_4, p > 2, (\ldots, p, \ldots, 2, \ldots), q_5, \cdot \rangle$$

   $$\langle q_4, p \leq 2, (\ldots, p, \ldots, 2, \ldots), q_6, \cdot \rangle$$

   $$\langle q_5, p \leftarrow p - k, (\ldots), q_4, \cdot \rangle.$$

4. L'istruzione di output corrisponde ad una scrittura sul nastro di output con la macchina che entra nello stato finale.

5. Resta da chiarire come vengano collegate le quintuple le une con le altre. Da quanto sopra esposto, si intuisce che *lo stato con il quale la macchina termina una istruzione è lo stato che le consente di iniziare l'istruzione successiva*: sempre riferendoci all'esempio in Tabella 3.1 ed alle pseudo-quintuple descritte sino ad ora, osserviamo che lo stato q_1, con il quale termina l'istruzione alla linea 1, è lo stato con il quale inizia l'istruzione **if** alla linea 2. Questa regola è valida con due sole eccezioni:

   (a) lo stato con il quale la macchina termina l'esecuzione dell'ultima istruzione della sequenza che costituisce un blocco **if** deve passare il controllo alla prima istruzione successiva al blocco **else**: nell'esempio, a partire dallo stato q_4 dovrà essere possibile eseguire l'istruzione **while**, e solo essa;

   (b) lo stato con il quale la macchina termina l'esecuzione dell'ultima istruzione della sequenza che costituisce il corpo di un **while** deve passare il controllo all'istruzione che testa la condizione del loop (nell'esempio, la macchina, dopo avere eseguito l'istruzione alla linea 7, entra nuovamente nello stato q_4).

Non possiamo, in questa sede, formalizzare maggiormente la codifica della macchina di Turing (ad esempio, esplicitandone le quintuple) perché essa dipende dal numero di nastri utilizzato. Per tale motivo, non procediamo neppure oltre con la prova formale della correttezza della trasformazione. Lasciamo come (utile) esercizio la progettazione delle quintuple corrispondenti all'algoritmo in Tabella 3.1. □

Il Teorema 3.5 mostra che le funzioni che possono essere calcolate da un programma scritto nel linguaggio **Pascal-Minimo** possono essere calcolate anche da una macchina di Turing, ossia, che il linguaggio **PascalMinimo** non è un sistema di calcolo più potente delle macchine di Turing. Il prossimo teorema prova la proposizione inversa, ossia, che le macchine di Turing non sono un sistema di calcolo più potente del linguaggio **PascalMinimo**.

Teorema 3.6: *Per ogni macchina di Turing deterministica T di tipo riconoscitore ad un nastro esiste un programma $\mathscr{P}$ scritto in accordo alle regole del linguaggio* **PascalMinimo** *tale che, per ogni stringa x, se $T(x)$ termina nello stato finale $q_F \in \{q_A, q_R\}$ allora $\mathscr{P}$ con input x restituisce q_F in output.*

Schema della dimostrazione: Siano $\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle$, $\langle q_{2_1}, s_{2_1}, s_{2_2}, q_{2_2}, m_2\rangle$, ..., $\langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle$, le quintuple che descrivono la macchina T (ove qualche q_{i_2} sarà lo stato di accettazione q_A di T e qualche q_{j_2} sarà lo stato di rigetto q_R).
In quanto segue, assumeremo sempre che il movimento della testina di T sia rappresentato da un intero in $\{-1, 0, +1\}$, dove -1 rappresenta il movimento della testina a sinistra, 0 la testina che rimane ferma, e $+1$ il movimento della testina a destra.

Il programma $\mathscr{P}_T$ è illustrato in Tabella 3.2. Esso utilizza l'array N per memorizzare il contenuto del nastro di T, la variabile q per memorizzare lo stato attuale di T e la variabile t per memorizzare la cella scandita della testina; inoltre, due variabili vengono utilizzate per delimitare la prorzione dell'array N effettivamente utilizzata in ogni istante: ad ogni istante, *primaCella* memorizza l'indirizzo della cella del nastro più a sinistra utilizzata fino ad allora dalla computazione $T(x)$, mentre *ultimaCella* memorizza l'indirizzo della cella del nastro più a destra utilizzata fino ad allora dalla computazione $T(x)$.

$\mathscr{P}_T$ non è altro che un loop **while** il cui corpo è una sequenza di istruzioni condizionali: fino a quando la macchina non entra in uno stato finale, viene eseguita l'istruzione corrispondente alla quintupla i cui primi due simboli sono q e $N[t]$. Ogni volta che viene eseguita una quintupla, inoltre, se nell'eseguire una quintupla la testina viene spostata a sinistra o a destra della porzione di nastro utilizzata fino a quel momento (questo accade, rispettivamente, quando $t < primaCella$ e quando $t > ultimaCella$), allora viene aggiornato il valore di *primaCella* o di *ultimaCella* e viene inizializzato a □ il valore di $N[t]$. □

La stessa tecnica utilizzata nella dimostrazione del Teorema 3.6 permette di scrivere un programma $\mathscr{P}$ che si comporta come la macchina di Turing universale. L'input del programma $\mathscr{P}$ è, in questo caso, costituito, oltre che dalla parola x, anche dalla descrizione della macchina T che deve essere simulata, ossia dalle descrizioni dell'alfabeto Σ, dell'insieme degli stati Q, delle quintuple $\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle$, $\langle q_{2_1}, s_{2_1}, s_{2_2}, q_{2_2}, m_2\rangle$, ..., $\langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle$ e degli stati iniziale e finali di T. Vengono utilizzati, allo scopo, le variabili di seguito descritte:

- per insieme degli stati ed alfabeto, rispettivamente, gli array $Q = \{q_1, q_2, \ldots, q_m\}$ e $\Sigma = \{s_1, s_2, \ldots, s_n\}$

Input: stringa $x_1x_2\ldots x_n$ memorizzata nell'array N, con $N[i] = x_i$ per $i = 1,\ldots,n$.

1	$q \leftarrow q_0$;
2	$t \leftarrow 1$;
3	$primaCella \leftarrow 1$;
4	$ultimaCella \leftarrow n$;
5	**while** $(q \neq q_A \ \wedge q \neq q_R)$ **do begin**
6	**if** $(q = q_{1_1} \ \wedge N[t] = s_{1_1})$ **then begin**
7	$N[t] \leftarrow s_{1_2}$;
8	$q \leftarrow q_{1_2}$;
9	$t \leftarrow t + m_1$;
10	**end**
11	**else if** $(q = q_{2_1} \ \wedge N[t] = s_{2_1})$ **then begin**
12	$N[t] \leftarrow s_{2_2}$;
13	$q \leftarrow q_{2_2}$;
14	$t \leftarrow t + m_2$;
15	**end**
	.....
$5k+5$	**else if** $(q = q_{k_1} \ \wedge N[t] = s_{k_1})$ **then begin**
$5k+6$	$N[t] \leftarrow s_{k_2}$;
$5k+7$	$q \leftarrow q_{k_2}$;
$5k+8$	$t \leftarrow t + m_k$;
$5k+9$	**end**
$5k+10$	**else** $q \leftarrow q_R$;
$5k+11$	**if** $(t < primaCella)$ **then begin**
$5k+12$	$primaCella \leftarrow t$;
$5k+13$	$N[t] \leftarrow \square$;
$5k+14$	**end**
$5k+15$	**if** $(t > ultimaCella)$ **then begin**
$5k+16$	$ultimaCella \leftarrow t$;
$5k+17$	$N[t] \leftarrow \square$;
$5k+18$	**end**
$5k+19$	**end**;
$5k+20$	**Output:** q

Tabella 3.2: Algoritmo corrispondente ad una Macchina di Turing deterministica ad un nastro T definita dall'insieme di quintuple $P = \{\langle q_{1_1}, s_{1_1}, s_{1_2}, q_{1_2}, m_1\rangle, \langle q_{2_1}, s_{2_1}, s_{2_2}, q_{2_2}, m_2\rangle, \ldots, \langle q_{k_1}, s_{k_1}, s_{k_2}, q_{k_2}, m_k\rangle\}$.

- per le quintuple gli array
  - $Q_1 = \{q_{1_1}, q_{2_1}, \ldots, q_{k_1}\}$ che descrive gli stati di partenza di ciascuna quintupla;
  - $S_1 = \{s_{1_1}, s_{2_1}, \ldots, s_{k_1}\}$ che descrive gli elementi di Σ che devono essere letti per poter eseguire ciascuna quintupla;
  - $S_2 = \{s_{1_2}, s_{2_2}, \ldots, s_{k_2}\}$ che descrive gli elementi di Σ che vengono scritti quando è eseguita ciascuna quintupla;
  - $Q_2 = \{q_{1_2}, q_{2_2}, \ldots, q_{k_2}\}$ che descrive gli stati di arrivo di ciascuna quintupla;
  - $M = \{m_1, m_2, \ldots, m_k\}$ che descrive i movimenti della testina che avvengono quando è eseguita ciascuna quintupla (ricordiamo che $m_i \in \{-1, 0, 1\}$ per ogni $i = 1, \ldots, k$);

  in tal modo la quintupla $\langle q_{j_1}, s_{j_1}, s_{j_2}, q_{j_2}, m_j \rangle$ è univocamente rappresentata da $Q_1[j]$, $S_1[j]$, $S_2[j]$, $Q_2[j]$ e $M[j]$;
- lo stato finale q_0 e i due stati finali q_A e q_R.

Nel programma $\mathscr{P}$, inoltre, vengono utilizzate le seguenti variabili:

- q, che descrive lo stato attuale della macchina;
- N, un array che descrive il contenuto del nastro ad ogni istante della computazione; la dimensione di N, ad ogni istante, è pari alla porzione di nastro utilizzata dalla macchina T in quell'istante;
- t, che è la posizione della testina sul nastro di T;
- *primaCella*, che, ad ogni istante, memorizza l'indirizzo della cella del nastro più a sinistra utilizzata fino ad allora dalla computazione $T(x)$;
- *ultimaCella*, che, ad ogni istante, memorizza l'indirizzo della cella del nastro più a destra utilizzata fino ad allora dalla computazione $T(x)$;
- i e j sono variabili di iterazione;
- *trovata* è una variabile booleana utilizzata per testimoniare dell'avvenuto ritrovamento della corretta quintupla da eseguire.

Il programma $\mathscr{P}$ è mostrato in Tabella 3.3. Intuitivamente, esso consiste in un unico loop (linee 10-35) nel quale, fissati lo stato attuale q e la posizione t della testina sul nastro (e, conseguentemente, il simbolo letto $N[t]$), cerca la quintupla che inizia con la coppia $(q, N[t])$ e, una volta trovata (se esiste), la esegue. Le variabili $t <$ *primaCella* e $t >$ *ultimaCella* vengono gestite allo stesso modo che nell'algoritmo in Tabella 3.2, così come l'inizializzazione a $\square$ del valore di $N[t]$.

Si osservi che sussiste una sostanziale differenza fra il comportamento dell'algoritmo $\mathscr{P}_T$ e il comportamento dell'algoritmo $\mathscr{P}$ nel caso in cui la quintupla che inizia con $(q, N[t])$ non esista: in tal caso, mentre l'esecuzione di $\mathscr{P}_T$ con input x non avrebbe termine, l'esecuzione di $\mathscr{P}$ con input T e x terminerebbe nello stato di rigetto.

Seppure intuitivo, non dimostriamo formalmente, in questa sede, che se $T(x)$ termina in uno stato finale allora l'esecuzione di $\mathscr{P}$ con input x restituisce in output lo stato finale in cui $T(x)$ ha terminato.

I due Teoremi 3.5 e 3.6 ci consentono di utilizzare, ai fini del nostro studio, il formalismo del linguaggio **PascalMinimo** al posto di quello, meno immediato, delle macchine di Turing.

Osserviamo, infine, che è possibile dimostrare che l'espressività del linguaggio **PascalMinimo** non cambia introducendo l'uso di funzioni, ossia di sottoprogrammi che possono essere utilizzati come parti destre nelle istruzioni di assegnazione. Una funzione viene definita mediante una *intestazione*, che ne specifica il nome ed i parametri, ossia, i valori letti da opportune variabili del corpo del programma principale specificate al momento delle sua invocazione, ed un *corpo*, costituito da una sequenza di istruzioni del linguaggio **PascalMinimo**. Il corpo di una funzione termina sempre con l'istruzione "**Output:**" che specifica il valore che deve essere scritto nella variabile che compare a sinistra dell'assegnazione in cui viene invocata. Nel seguito di questo corso, utilizzeremo frequentemente le funzioni insieme con la loro variante in cui non viene restituito alcun valore in output ma che modificano i valori delle variabili che ricevono come parametri: ci riferiremo a tale variante delle funzioni con il termine *procedure*.

Input:	stringa $x_1x_2\ldots x_n$ memorizzata nell'array N, con $N[i] = x_i$ per $i = 1,\ldots,n$, array Q, Σ, Q_1, S_1, S_2,Q_2, M descritti nel testo, q_0, q_A, q_R.
1	$q \leftarrow q_0$;
2	$t \leftarrow 1$;
3	$primaCella \leftarrow 1$;
4	$ultimaCella \leftarrow n$;
5	**while** $(q \neq q_A \ \wedge \ q \neq q_R)$ **do begin**
6	$j \leftarrow 1$;
7	$trovata \leftarrow$ `falso`;
8	**while** $(j \leq k \wedge trovata =$ `falso`$)$ **do**
9	**if** $(q = Q_1[j] \wedge N[t] = S_1[j])$ **then** $trovata \leftarrow$ `vero`;
10	**else** $j \leftarrow j+1$;
11	**if** $(trovata =$ `vero`$)$ **then begin**
12	$N[t] \leftarrow S_2[j]$;
13	$q \leftarrow Q_2[j]$;
14	$t \leftarrow t + M[j]$;
15	**if** $(t < primaCella)$ **then begin**
16	$primaCella \leftarrow t$;
17	$N[t] \leftarrow \Box$;
18	**end**
19	**if** $(t > ultimaCella)$ **then begin**
20	$ultimaCella \leftarrow t$;
21	$N[t] \leftarrow \Box$;
22	**end**
23	**end**
24	**else** $q \leftarrow q_R$;
25	**end**
26	**Output:** q

Tabella 3.3: Algoritmo corrispondente alla Macchina di Turing universale.

Input:	stringa $x_1x_2\ldots x_n$ memorizzata nell'array N, con $N[i] = x_i$ per $i = 1,\ldots,n$. Il programma utilizza anche le seguenti costanti: q_0, q_A, q_R e inoltre gli array Q, Σ, Q_1, S_1, S_2,Q_2, M descritti nel testo,
1	$i \leftarrow 1$;
2	*primaCella* $\leftarrow 1$;
2	*ultimaCella* $\leftarrow n$;
3	**while** $(q \neq q_A \wedge q \neq q_R)$ **do begin**
4	$q \leftarrow$ `simulaRicorsivo`$(q_0, 1, N, i)$;
5	$i \leftarrow i+1$;
6	**end**
7	**Output:** q

Tabella 3.4: Algoritmo che simula la computazione di una fissata Macchina di Turing non deterministica NT.

3.4 Simulazione di una macchina di Turing non deterministica

Come esempio di utilizzo del linguaggio Pascal minimo, presentiamo ora un algoritmo scritto in tale linguaggio che corrisponde ad una macchina di Turing deterministica che simula il comportamento di una macchina di Turing non deterministica mediante la tecnica della coda di rondine con ripetizioni.

Nell'algoritmo, descritto in Tabella 3.4, in cui simuliamo la computazione di una macchina di Turing non deterministica NT con input x, assumiamo che la macchina NT sia descritta dall'insieme di quintuple $P = \{p_1, p_2, \ldots, p_k\}$ memorizzate negli array Q_1, S_1, S_2, Q_2 e M descritti nel Paragrafo 3.3 Assumiamo, inoltre, che tali array e gli stati iniziale e finale siano codificati nell'algoritmo (ossia, siano costanti).
L'algoritmo consiste di un ciclo **while** controllato da una variabile intera i. Durante la i-esima iterazione del ciclo, l'algoritmo verifica se esiste una computazione di i passi di NT che termina nello stato di accettazione oppure se tutte le computazioni di i passi di NT terminano nello stato di rigetto: se si verifica la prima ipotesi allora il ciclo **while** viene interrotto e l'algoritmo restituisce in output q_A, se si verifica la seconda ipotesi allora il ciclo **while** viene interrotto e l'algoritmo restituisce in output q_R, se non si verifica alcuna delle due ipotesi il valore di i viene incrementato ed inizia l'iterazione successiva (poiché nulla si può concludere in questo caso circa la accettazione o il rigetto di x da parte di NT).

La verifica della possibilità di accettare o rigettare l'input in i passi è compito della funzione `simulaRicorsivo`, descritta in Tabella 3.5. È questa una funzione ricorsiva di tipo Q (ossia, calcola e restituisce in output uno stato di NT) i cui parametri formali sono tutti collegati per valore ai corrispondenti parametri attuali (cioè, ogni invocazione ricorsiva copia nel proprio spazio di indirizzi il valore ricevuto come parametro in modo tale che le modifiche al parametro che avvengono nel corso della sua esecuzione non sono visibili dalla funzione chiamante). In dettaglio, i parametri dela funzione `simulaRicorsivo` sono lo stato globale in cui si trova NT nel momento in cui parte la simulazione (individuato dallo stato attuale q, dal contenuto del nastro N, array di tipo Σ, e dalla posizione della testina t) ed il numero di passi i che devono essere simulati.
La ricorsione termina quando $i = 0$: in questo caso, la funzione non fa altro che restituire lo stato interno in cui si troverebbe la macchina di Turing, ossia, lo stesso stato interno con il quale essa è stata invocata (linea 2).
Se, invece, si vuole verificare se uno stato finale può essere raggiunto in $i > 1$ istruzioni a partire dallo stato globale individuato dai parametri (attuali), allora viene eseguita la parte **else** di tale istruzione (linee 3-27). In questo caso, devono essere eseguite *tutte* le computazioni di i passi che partono dallo stato globale attuale: questo è il compito del ciclo **while** alle linee 6-25 che si occupa di cercare, una alla volta, a partire dalla prima quintupla (inizializzando ad 1 la variabile j), tutte le quintuple che possono essere eseguite. Più in dettaglio, cerca, innanzi tutto, la prima quintupla non ancora considerata che parte dallo stato globale attuale (ciclo **while** della linea 7) e, se la trova,

1. inizia ad eseguirla modificando l'elemento dell'array N corrispondente alla cella di memoria scandita dalla macchina di Turing (linea 10);

2. invoca ricorsivamente la funzione `simulaRicorsivo` per verificare se uno stato finale può essere raggiunto in $i-1$ istruzioni a partire dallo stato globale individuato dall'esecuzione della quintupla (linee 19-21):

   - se l'invocazione ricorsiva ha raggiunto lo stato q_A (linea 20), termina il ciclo **while**, altrimenti

- se l'invocazione ricorsiva non ha raggiunto lo stato q_R (linea 21), non è possibile rigettare in i passi e, quindi, assegna alla variabile *rigetto* il valore `falso`;

3. in ogni caso, prima di terminare il chiclo **while** o di iniziare una nuova iterazione, ripristina lo stato del nastro precedente all'esecuzione della quintupla (linea 22) e, se non ha raggiunto lo stato q_A, si predispone ad eseguire una nuova iterazione del ciclo **while** (linea 23).

Terminato il ciclo **while**, verifica se uno stato finale è stato raggiunto (linea 26): in caso ciò non sia accaduto, restituisce lo stato iniziale q_0, altrimenti restituisce lo stato finale raggiunto (che coincide con q_R se la variabile *rigetto* ha valore `vero`). Si osservi che l'**if** alla linea 26 viene eseguito nel caso in cui la sotto-computazione di $NT(x)$ che parte dallo stato globale individuato dai parametri attuali non accetta né rigetta in i passi: informazione, questa, che viene comunicata al chiamante mediante lo stato iniziale q_0. Nel caso in cui il chiamante sia l'algoritmo principale (descritto in Tabella 3.4), esso lancia la simulazione di $NT(x)$ per $i+1$ passi a partire, nuovamente, dallo stato iniziale q_0.

3.5 Macchine di Turing con oracolo

In questo paragrafo introduciamo un modello di calcolo strettamente più potente della macchina di Turing. Tale modello è esclusivamente teorico e totalmente irrealistico e, quindi, la sua definizione non confuta la tesi di Church-Turing.

Sia Σ un alfabeto e $L \subseteq \Sigma^*$ un qualsiasi linguaggio. Una macchina di Turing (deterministica o non deterministica) con oracolo L è una macchina T^L ad uno o più nastri e dotata di un nastro supplementare, il *nastro oracolo*, e di tre stati supplementari, lo stato *interroga-oracolo*, lo stato *oracolo-sì* e lo stato *oracolo-no*.
Su un dato input x, T^L esegue una serie di istruzioni, scrive una parola y sul nastro oracolo ed entra nello stato *interroga-oracolo*. A questo punto, entra in gioco l'oracolo che, *in una singola istruzione*, decide l'appartenenza di y ad L: se $y \in L$ la macchina entra nello stato *oracolo-sì*, se $y \notin L$ la macchina entra nello stato *oracolo-no*. Successivamente, T^L continua la sua computazione eventualmente ponendo nuove domande all'oracolo.

Ovviamente, il potere di calcolo di una macchina di Turing con oracolo è conseguenza diretta delle caratteristiche di decidibilità del linguaggio oracolo.
Così, se L è un linguaggio decidibile, è molto semplice simulare T^L mediante una macchina T' che, ogni volta che T^L scrive una stringa y sul nastro oracolo, esegue le istruzioni necessarie a decidere se $y \in L$.
Di contro, se L è un linguaggio non accettabile, è molto semplice definire una macchina T^L che decide L: con input x, tale macchina non farà altro che scrivere x sul nastro oracolo ed attendere la risposta dell'oracolo. Se l'oracolo accetta, allora $x \in L$ e quindi T^L accetta; viceversa, se l'oracolo rigetta allora T^L rigetta. Dunque, tutti i linguaggi sono accettabili da qualche opportuna macchina di Turing con oracolo.

Da quanto sopra osservato, appare evidente che le macchine con oracolo riescono a decidere anche linguaggi non decidibili, ossia, di calcolare anche ciò che non è calcolabile da una normale macchina di Turing. D'altro canto, questo è reso possibile dall'ipotesi di disporre di oracoli aventi potere di calcolo illimitato. In effetti, la descrizione della soluzione di un problema mediante una macchina di Turing con oracolo, benché finita, non è una sequenza che contiene soltanto *passi elementari*. Ricordiamo, infatti, che il concetto di passo elementare, così come definito da Turing, coincide con l'esecuzione di una quintupla di una macchina di Turing, ossia, con una decisione da prendere sulla base dello stato interno e di *un singolo elemento dell'alfabeto* letto sul nastro[1]. Invece, lo stato *interroga-oracolo* consente di interrogarsi su una intera parola, di lunghezza non costante, in una singola istruzione e, dunque, l'interrogazione dell'oracolo non può essere considerata un passo elementare. Per questa ragione, il modello è totalmente irrealistico ed esula dalle premesse di validità della tesi di Church-Turing.

Osserviamo, infine, che il potere di calcolo potenzialmente infinito di cui dispone tale modello lo rende pressoché inutilizzabile nell'ambito della Teoria della Calcolabilità.

[1] In realtà, è possibile generalizzare il concetto di passo elementare ad una sequenza di lunghezza *costante* di caratteri letti (ad esempio, 4 o 5), ossia, nota al momento della definizione della macchina.

```
    Q funzione simulaRicorsivo(Q q, int t, Σ[ ] N, int i)
1   begin
2       if (i = 0) then ρ ← q;
                // la precedente istruzione gestisce il caso in cui la ricorsione termina
3       else begin
4           j ← 1;
5           rigetto ← vero;
6           while (j ≤ k ∧ q ≠ q_A) do begin
7               while (j ≤ k ∧ [ q ≠ Q_1[j] ∨ N[t] ≠ S_1[j] ]) do j ← j + 1;
8               if (j = k + 1) then ρ ← q_R;
                        // nessuna (ulteriore) quintupla da eseguire è stata trovata
9               else begin
                        // ora i primi due elementi di p_j sono q e N[t]
10                      N[t] ← S_2[j];
11                      if (t + M[j] < primaCella) then begin
12                          primaCella ← t + M[j];
13                          N[t + M[j]] ← □;
14                      end
15                      if (t + M[j] > ultimaCella) then begin
16                          ultimaCella ← t + M[j];
17                          N[t + M[j]] ← □;
18                      end
19                      ρ ← simulaRicorsivo(Q_2[j], t + M[j], N, i − 1);
20                      if (ρ = q_A) then q ← q_A;
21                      else if (ρ ≠ q_R) then rigetto ← falso;
22                      N[t] ← S_1[j];
                            // lo stato di N viene ripristinato alla situazione precedente l'esecuzione
                            della quintupla p_j
23                      j ← j + 1;
                            // si predispone ad iniziare la ricerca di una nuova quintupla da eseguire:
                            tale ricerca avrà luogo solo se nessuna computazione deterministica
                            precedente (iniziata da una quintupla p_h con h < j) ha accettato
24              end;
25          end;
26          if (ρ ≠ q_A ∧ rigetto = falso) then ρ ← q_0;
                    // la precedente istruzione gestisce il caso in cui il ciclo while alle linee 6-25
                        termina senza aver trovato lo stato q_A e senza poter decidere circa il rigetto;
                        si osservi che, se ρ ≠ q_A e rigetto = vero, allora tutte le quintuple eseguite
                        hanno portato a rigettare e, quindi, ρ = q_R
27      end;
28      return: ρ;
29  end
```

Tabella 3.5: La funzione ricorsiva che simula la computazione di una fissata Macchina di Turing non deterministica NT.