Capitolo 5

La tesi di Church-Turing

SOMMARIO

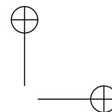
Sebbene il concetto di algoritmo abbia avuto una lunga storia nel campo della matematica, il concetto formale di algoritmo non fu introdotto prima dell'inizio del diciannovesimo secolo: prima di allora, i matematici avevano una nozione intuitiva di cosa fosse un algoritmo e su di essa facevano affidamento per usare e descrivere algoritmi. Tuttavia, tale nozione intuitiva era insufficiente per comprendere appieno le potenzialità del calcolo algoritmico: per questo motivo, come già detto nell'introduzione, diversi ricercatori nel campo della logica matematica proposero una definizione formale del concetto di algoritmo. In questo capitolo, presenteremo tre di queste definizioni formali, dimostrando, per ciascuna di esse, come essa non introduca un modello di calcolo più potente delle macchine di Turing ed enunciando (senza dimostrarla) la sua equivalenza con il modello di calcolo della macchina di Turing: in base a tali risultati, enunceremo infine quella che tutt'ora è nota coma la tesi di Church-Turing.

5.1 Algoritmi di Markov

All'inizio degli anni cinquanta il matematico e logico russo Markov, propose un modello di calcolo basato sull'elaborazione di stringhe, che dimostrò essere equivalente alle macchine di Turing. A differenza delle macchine di Turing, gli algoritmi di Markov operano esclusivamente in funzione del contenuto della stringa e non in funzione di uno stato interno dell'algoritmo.

In particolare, un **algoritmo di Markov** è definito mediante una serie di **regole di riscrittura** che permettono di trasformare, in modo univoco, il contenuto di una stringa in un'altra stringa. Formalmente, esso consiste di:

- un alfabeto Σ, di cui un sotto-insieme Γ costituisce l'alfabeto dei caratteri di cui può essere costituita la stringa iniziale o di input;



- un insieme finito e ordinato R di regole del tipo $x \rightarrow y$ oppure del tipo $x \Rightarrow y$, dove x e y sono stringhe su Σ (le regole del secondo tipo sono anche dette regole **terminali**).

Una regola $x \rightarrow y$ o $x \Rightarrow y$ si dice essere **applicabile** a una stringa z su Σ, se $z = z_1 x z_2$, dove z_1 e z_2 sono due stringhe su Σ tali che $z_1 x$ contiene una sola occorrenza della stringa x: **applicare** la regola a z, significa sostituire la prima occorrenza di x in z con y.

Partendo da una stringa $x \in \Gamma^*$ di input, un algoritmo di Markov opera nel modo seguente.

1. Scandisce tutte le regole in R nell'ordine stabilito, alla ricerca di una che sia applicabile alla stringa corrente: se nessuna regola viene trovata, l'algoritmo termina producendo in output la stringa corrente.
2. Se r è la regola trovata al passo precedente, applica r alla stringa corrente: la nuova stringa diviene la stringa corrente.
3. Se r non è terminale, torna al primo passo, altrimenti termina producendo in output la stringa corrente.

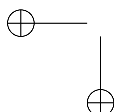
Esempio 5.1: algoritmo di Markov per il successore

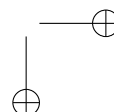
Dato un numero naturale n codificato in unario mediante la stringa 1^{n+1}, un semplice algoritmo di Markov che produce la codifica del suo successore è costituito dalla sola regola terminale $\epsilon \Rightarrow 1$. Tale regola è sempre applicabile e inserisce un simbolo 1 in testa alla stringa di input.

Esempio 5.2: algoritmo di Markov per la somma

Dati due numeri naturali n e m codificati in unario e separati da un simbolo 0, un semplice algoritmo di Markov che produce la codifica della loro somma è costituito dalla sola regola terminale $10 \Rightarrow \epsilon$. Tale regola elimina il separatore tra i due numeri insieme al simbolo 1 di troppo.

Il prossimo esempio illustra come l'ordine con cui le regole sono presenti nell'insieme R sia determinante per il corretto funzionamento dell'algoritmo. In particolare, osserviamo che se R contiene due regole $x_1 \rightarrow y_1$ e $x_2 \rightarrow y_2$ (eventualmente terminali) tali che x_1 è un prefisso di x_2, allora la regola $x_2 \rightarrow y_2$ deve precedere la regola $x_1 \rightarrow y_1$, perché altrimenti essa non verrebbe mai selezionata durante l'esecuzione dell'algoritmo: in particolare, se R include una regola $\epsilon \rightarrow y$ (eventualmente terminale), questa deve essere la sua ultima regola.





Esempio 5.3: algoritmo di Markov per il prodotto

Dati due numeri naturali n e m codificati in unario e separati da un simbolo 0, un algoritmo di Markov che produce la codifica del loro prodotto può essere realizzato immaginando di eseguire la moltiplicazione in modo simile a quanto fatto con le macchine di Turing nel terzo capitolo. In particolare, l'algoritmo cancella un simbolo 1 da entrambe le sequenze, quindi, per ogni simbolo 1 della prima sequenza esegue una copia della seconda sequenza in fondo alla stringa corrente e, infine, cancella la seconda sequenza: l'insieme di regole che implementa tale strategia è la seguente.

1	$\pi 1 \to \pi_0$	10	$\alpha 0 \to \lambda$	19	$1\delta_1 \to \delta_1 1$
2	$\pi_0 1 \to 1\pi_0$	11	$\alpha 1 \to \beta$	20	$2\delta_1 \to 2\gamma$
3	$\pi_0 0 \to 0\rho$	12	$\beta 1 \to 1\beta$	21	$2\mu \to \mu 1$
4	$\rho 1 \to \rho_0$	13	$\beta 0 \to 0\gamma$	22	$0\mu \to \nu 0$
5	$\rho_0 1 \to 1\rho_0$	14	$\gamma 1 \to 2\delta$	23	$1\nu \to \nu 1$
6	$\rho_0 \to \sigma{=}$	15	$\delta 1 \to 1\delta$	24	$\nu \to \alpha$
7	$1\sigma \to \sigma 1$	16	$\delta{=} \to \delta_0{=}\,1$	25	$\lambda 1 \to \lambda$
8	$0\sigma \to \sigma 0$	17	$1\delta_0 \to \delta_1 1$	26	$\lambda{=} \to .1$
9	$\sigma \to \alpha$	18	$2\delta_0 \to \mu 1$	27	$\epsilon \to \pi$

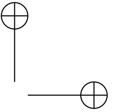
Le regole 1-9 realizzano la cancellazione di un simbolo 1 nelle due sequenze e l'inserimento di un separatore (ovvero il simbolo =) al termine della seconda sequenza, mentre le regole 10-24 eseguono, per ogni simbolo 1 della prima sequenza, una copia della seconda sequenza immediatamente dopo il simbolo =. Infine, le regole 25, 26 e 27 consentono di cancellare la seconda sequenza e il simbolo =.

Una funzione $f : \Sigma^* \to \Sigma^*$ è detta essere **Markov-calcolabile** se esiste un algoritmo di Markov che, partendo da una stringa $x \in \Sigma^*$, produce in output $f(x)$.

Teorema 5.1

Se una funzione $f : \Sigma^* \to \Sigma^*$ è Markov-calcolabile, allora esiste una macchina di Turing che calcola f.

Dimostrazione. Sia R l'insieme delle regole di riscrittura dell'algoritmo di Markov che calcola f, con $n = |R|$, e sia $a_i \to b_i$ o $a_i \Rightarrow b_i$ l'i-esima regola di R, per i compreso tra 1 e n. La macchina di Turing T che calcola f include uno stato $q_0^{[i]}$, per ogni i con $1 \leqslant i \leqslant n$: in particolare, $q_0^{[1]}$ è lo stato iniziale di T. Tale macchina opera nel modo seguente. Trovandosi nello stato $q_0^{[i]}$, cerca la prima occorrenza di a_i: se non la trova, scorre il nastro verso sinistra fino a incontrare un simbolo @, e, se $i < n$, si sposta a destra e passa nello stato $q_0^{[i+1]}$, altrimenti (ovvero $i = n$), si sposta a destra e passa nello stato finale $q_{\mathtt{end}}$. Se, invece, T trova un'occorrenza di a_i, allora trasforma a_i in una sequenza



di $*$, a cui aggiunge o sottrae tanti simboli $*$ quanti ne servono per ottenere una sequenza di lunghezza pari a $|b_i|$ e trasforma la sequenza di $*$ così ottenuta in b_i. Se la i-esima regola è terminale, allora passa nello stato $q_{\mathtt{pre-end}}$ altrimenti passa nello stato $q_{\mathtt{next}}$: in entrambi i casi, scorre il nastro verso sinistra fino a trovare un @. Infine, se si trova nello stato $q_{\mathtt{pre-end}}$, si sposta a destra e passa nello stato finale $q_{\mathtt{end}}$, altrimenti (ovvero si trova nello stato $q_{\mathtt{next}}$) passa nello stato $q_0^{[1]}$. □

Abbiamo dunque dimostrato che gli algoritmi di Markov non sono un modello più potente delle macchine di Turing e, quindi, che tutto ciò che non può essere calcolato da una macchina di Turing non può essere calcolato da un algoritmo di Markov. In particolare, tutti i linguaggi non decidibili che abbiamo mostrato nel precedente capitolo non sono decidibili facendo uso degli algoritmi di Markov.

5.2 Funzioni ricorsive

La teoria delle funzioni ricorsive fu sviluppata dal logico matematico Kleene, sulla base di quanto fatto in precedenza da Gödel per definire in modo "ragionevole" il concetto di funzione calcolabile. Una funzione ricorsiva è una funzione che può essere ottenuta, a partire da tre funzioni di base, applicando uno di tre possibili operatori, ovvero l'operatore di composizione, quello di ricorsione primitiva e quello di minimalizzazione.

In particolare, una funzione $f : N^n \to N$ è detta essere ricorsiva se una delle seguenti condizioni è vera:

1. f è la funzione **successore** S, tale che, per ogni numero naturale x, $S(x) = x + 1$;

2. f è la funzione **zero** Z^n, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$, $Z^n(x_1, \ldots, x_n) = 0$;

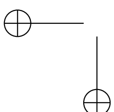
3. f è la funzione **selettore** U^n_k con $1 \leqslant k \leqslant n$, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$, $U^n_k(x_1, \ldots, x_n) = x_k$;

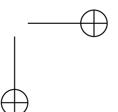
4. esistono m funzioni ricorsive $g_1, \ldots, g_m : N^n \to N$ e una funzione ricorsiva $h : N^m \to N$, tali che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

5. esiste una funzione ricorsiva $b : N^n \to N$ e una funzione ricorsiva $i : N^{n+1} \to N$, tali che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,

$$f(x_1, \ldots, x_n) = \begin{cases} b(x_1, \ldots, x_n) & \text{se } x_1 = 0, \\ i(x_1 - 1, \ldots, x_n, f(x_1 - 1, \ldots, x_n)) & \text{altrimenti} \end{cases}$$





6. esiste una funzione ricorsiva $p : N^{n+1} \rightarrow N$, tale che, per ogni n-tupla di numeri naturali $x_1, \ldots, x_n$,

$$f(x_1, \ldots, x_n) = \min_{t \geqslant 0}[p(x_1, \ldots, x_n, t) = 0]$$

Esempio 5.4: funzione ricorsiva per il calcolo del predecessore

È facile dimostrare che la funzione predecessore è una funzione ricorsiva (assumendo che il predecessore di 0 sia 0 stesso). Infatti, tale funzione può essere definita nel modo seguente:

$$f(x) = \begin{cases} Z^1(x) & \text{se } x = 0, \\ U_1^2(x-1, f(x-1)) & \text{altrimenti} \end{cases}$$

Esempio 5.5: funzione ricorsiva per il calcolo della somma

È anche facile dimostrare che la funzione di somma di due numeri naturali è una funzione ricorsiva. Infatti, tale funzione può essere definita nel modo seguente:

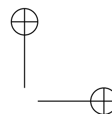
$$f(x, y) = \begin{cases} U_2^2(x, y) & \text{se } x = 0, \\ S(U_3^3(x-1, y, f(x-1, y))) & \text{altrimenti} \end{cases}$$

Teorema 5.2

Data una funzione $f : N^n \rightarrow N$ ricorsiva, esiste una macchina di Turing che calcola f.

Dimostrazione. La dimostrazione consiste anzitutto nel mostrare che le tre funzioni di base sono calcolabili da una macchina di Turing (si vedano gli Esercizi 2.1, 2.2 e 2.3). Facendo poi uso del concetto di sotto-macchina, non è difficile dimostrare che una funzione ottenuta per composizione di funzioni calcolabili da macchine di Turing è essa stessa calcolabile da una macchina di Turing, che una funzione ottenuta per ricorsione primitiva a partire da funzioni calcolabili da macchine di Turing è essa stessa calcolabile da una macchina di Turing e che una funzione ottenuta per minimizzazione a partire da una funzione calcolabile da una macchina di Turing è essa stessa calcolabile da una macchina di Turing (si veda l'Esercizio 5.6). □

Anche in questo caso, abbiamo dunque dimostrato che le funzioni ricorsive non sono un modello più potente delle macchine di Turing e, quindi, che esistono funzioni che non sono ricorsive.



5.3 Macchine RAM

La maggior parte dei programmatori sono soliti esprimere i loro algoritmi facendo uso di linguaggi di programmazione di alto livello come `C`, `Java` e `COBOL`, sapendo che un programma (sintatticamente corretto) scritto in uno qualunque di tali linguaggi può essere tradotto in un programma scritto in linguaggio macchina: questo è effettivamente il compito svolto dai compilatori. L'ultimo modello di calcolo che consideriamo in questo capitolo è un modello molto simile a un linguaggio macchina e, quindi, in grado di calcolare tutto quello che possiamo calcolare facendo uso di un linguaggio di programmazione di alto livello. A dire il vero, tale modello risulta essere apparentemente più potente di un linguaggio macchina, non avendo alcun vincolo sulla dimensione di una parola di memoria o di un indirizzo di una cella di memoria: in questo paragrafo, mostreremo che anche questo modello non è più potente delle macchine di Turing, per cui potremo concludere che esistono funzioni che non possono essere calcolate da alcun programma scritto, ad esempio, in `Java`.

Una **Random Access Machine** (in breve, **RAM**) ha un numero potenzialmente infinito di *registri* e un *contatore di programma*: ogni registro è individuato da un *indirizzo*, ovvero da un numero intero non negativo, e può contenere un numero intero non negativo *arbitrariamente* grande.

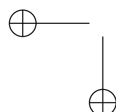
Nel seguito, indicheremo con `n` un numero intero non negativo, con `(n)` il contenuto del registro il cui indirizzo è `n` e con `[n]` il contenuto del registro il cui indirizzo è contenuto nel registro il cui indirizzo è `n` (si tratta del ben noto *indirizzamento indiretto*). Un **programma RAM** è una sequenza di istruzioni (eventualmente dotate di un'etichetta) di uno dei seguenti tipi.

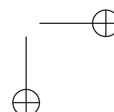
- `accept` oppure `reject`: il programma termina accettando o rifiutando l'input letto.

- `read (n)` oppure `read [n]`: l'input di un programma RAM è un flusso di numeri interi non negativi e queste due istruzioni leggono il prossimo numero del flusso memorizzandolo, rispettivamente, nel registro il cui indirizzo è `n` oppure nel registro il cui indirizzo è contenuto nel registro il cui indirizzo è `n`.

- `(n) : = o1` *operatore* `o2` oppure `[n] := o1` *operatore* `o2` dove *operatore* indica una delle quattro operazioni aritmetiche elementari, ovvero `+`, `-`, `*` e `/` e `o1` (rispettivamente, `o2`) può essere `m1`, `(m1)` oppure `[m1]` (rispettivamente, `m2`, `(m2)` oppure `[m2]`), per un qualunque numero intero non negativo `m1` (rispettivamente, `m2`).

- `if o1` *operatore* `o2 goto` *etichetta* dove *operatore* indica una delle quattro ope-





razioni relazionali =, <>, <= e < e o1 e o2 hanno lo stesso significato del tipo di istruzione precedente.

All'avvio dell'esecuzione di un programma RAM, tutti i registri contengono 0 e il contatore di programma è uguale a 0: a ogni passo, l'istruzione indicata dal contatore di programma viene eseguita. Se tale istruzione non è un `if` oppure è un `if` la cui condizione booleana non è verificata, il contatore di programma aumenta di 1, altrimenti il contatore di programma viene posto uguale all'indice dell'istruzione corrispondente all'etichetta dell'istruzione `if`.

Esempio 5.6: programma RAM per la verifica di sequenze palindrome

Supponiamo di voler decidere se una sequenza di 0 e 1, terminata da un 2, è palindroma. Un programma RAM che risolve tale problema può leggere la sequenza memorizzandola in registri adiacenti per poi scorrere tali registri in una direzione e nell'altra verificando che i numeri corrispondenti siano uguali. Formalmente, un tale programma è composto dalla seguente sequenza di istruzioni.

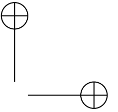
```
        (1) := 2 + 0
 next:  read [1]
        if [1] = 2 goto test
        (1) := (1) + 1
        if 0 = 0 goto next
 test:  (1) := (1) -1
        (0) = 2 + 0
 loop:  if (1) <= (0) goto yes
        if [0] <> [1] goto no
        (0) = (0) + 1
        (1) = (1) - 1
        if 0 = 0 goto loop
  yes:  accept
   no:  reject
```

Una funzione $f : N^n \rightarrow \{0, 1\}$ è **RAM-calcolabile** se esiste un programma RAM che, data in input una sequenza x di n numeri interi, termina con l'istruzione `accept` se $f(x) = 1$ e termina con l'istruzione `reject` se $f(x) = 0$.

Teorema 5.3

Ogni funzione $f : N^n \rightarrow N$ RAM-calcolabile è calcolabile da una macchina di Turing.

Dimostrazione. Definiamo una macchina di Turing T con più nastri, di cui un nastro funge da input e contiene la sequenza di n numeri interi (ad esempio, rappresentati in binario e separati da un @) e un nastro funge da memoria e contiene inizialmente



contenente il solo simbolo $ (i rimanenti nastri sono nastri di lavoro, inizialmente vuoti e il cui utilizzo sarà chiarito nel corso della definizione di T). A regime il nastro di memoria contiene la stringa

$$\$\#a_1:v_1\#a_2:v_2\#a_3:v_3\#\cdots a_m:v_m\#@@@\cdots$$

dove a_i indica l'indirizzo di un registro e v_i ne indica il contenuto (assumiamo che se l'indirizzo di un registro non appare in tale sequenza, il suo contenuto sia uguale a 0). Se l'indirizzo di un registro appare più di una volta nella sequenza, allora il suo contenuto attuale è quello che appare più a destra nella sequenza stessa. Il nastro di memoria viene usato per realizzare le seguenti due operazioni.

- Determinare il contenuto di un registro il cui indirizzo a è memorizzato su un nastro di lavoro: a tale scopo, T scorre verso destra il nastro di memoria fino a trovare un @, per poi spostarsi verso sinistra, cercando la prima occorrenza di a. Se T non trova tale occorrenza, il contenuto del registro è 0, altrimenti il contenuto del registro è racchiuso tra il successivo simbolo : e il successivo simbolo #: in entrambi i casi, tale contenuto viene scritto su un altro nastro di lavoro.

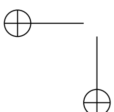
- Modificare il contenuto di un registro il cui indirizzo a è memorizzato su un nastro di lavoro, assegnandogli il valore v memorizzato su un altro nastro di lavoro: a tale scopo, T scorre il nastro di memoria fino a trovare un @ e scrive $a:v\#$.

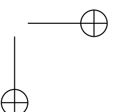
Ogni istruzione del programma RAM che calcola f è realizzata da una sotto-macchina, la cui definizione dipende dal tipo dell'istruzione nel modo seguente (nel seguito, considereremo solo il caso in cui l'indirizzamento sia esclusivamente indiretto, in quanto questo risulta essere il caso più complesso).

- L'istruzione è `accept` (rispettivamente, `reject`): in tal caso, T scrive 1@ (rispettivamente, 0@) su un nastro di output, posiziona la testina sul simbolo 1 (rispettivamente, 0) e termina.

- L'istruzione è `read [n]`: in tal caso, T legge il contenuto a del registro `n`, legge il prossimo numero intero v e assegna al registro a il valore v.

- L'istruzione è `[n] := [m1]` *operatore* `[m2]`: in tal caso, T legge il contenuto a del registro `n`, legge il contenuto b1 del registro `m1`, legge il contenuto o1 del registro b1, legge il contenuto b2 del registro `m2`, legge il contenuto o2 del registro b2, calcola il valore v ottenuto eseguendo o1 *operatore* o2 e assegna al registro a il valore v.





- L'istruzione è `if [n1]` *operatore* `[n2] goto l`: in tal caso, T legge il contenuto a1 del registro `n1`, legge il contenuto o1 del registro a1, legge il contenuto a2 del registro `n2`, legge il contenuto o2 del registro a2. Se o1 *operatore* o2 non è vera, passa il controllo alla sotto-macchina corrispondente all'istruzione successiva, altrimenti passa il controllo alla sotto-macchina corrispondente all'istruzione con etichetta `l`.

È evidente che T termina producendo come output il valore 1 se il programma RAM termina eseguendo l'istruzione `accept` e termina producendo come output il valore 0 se il programma RAM termina eseguendo l'istruzione `reject`: pertanto, la funzione T calcola la funzione f e il teorema risulta essere dimostrato. □

Anche nel caso delle RAM, abbiamo dunque dimostrato che tale modello di calcolo non è più potente delle macchine di Turing e, quindi, che esistono funzioni che non sono RAM-calcolabili.

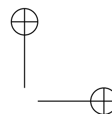
5.4 La tesi di Church-Turing

I risultati ottenuti nei precedenti paragrafi mostrano come modelli di calcolo alternativi alle macchine di Turing, introdotti all'incirca nello stesso periodo di quello introdotto da Turing oppure simili agli odierni calcolatori, non siano in grado di calcolare funzioni che non sono calcolabili mediante una macchina di Turing. Pertanto, possiamo ragionevolmente assumere che tutti i risultati negativi ottenuti nel precedente capitolo valgano *indipendentemente* dal modello di calcolo utilizzato, mostrando la difficoltà computazionale intrinseca ad alcuni linguaggi (come, ad esempio, quello della terminazione). È d'altra parte possibile dimostrare il seguente risultato.

Teorema 5.4

Ogni funzione calcolabile da una macchina di Turing è Markov-calcolabile, ricorsiva e RAM-calcolabile.

La dimostrazione di tale teorema risulta non troppo complicata nel caso degli algoritmi di Markov e dei programmi RAM (si vedano gli Esercizi 5.11 e 5.12). Nel caso delle funzioni ricorsive, invece, la dimostrazione fa uso di una tecnica introdotta dal logico Kurt Gödel nella dimostrazione del suo famoso teorema e, per questo, chiamata *tecnica di goedilizzazione*: tale tecnica consente di mettere in corrispondenza biunivoca e in modo costruttivo l'insieme dei numeri naturali con quello delle configurazioni della computazione di una macchina di Turing, simulando così attraverso funzioni ricorsive il processo di produzione di una configurazione a partire da un'altra configurazione.



Il Teorema 5.4, insieme a quelli dimostrati nei paragrafi precedenti, giustificano la tesi di Church-Turing già esposta nell'introduzione del corso e che può essere formulata nel modo seguente.

> È CALCOLABILE TUTTO CIÒ CHE PUÒ ESSERE CALCOLATO DA UNA MACCHINA DI TURING.

In altre parole, possiamo concludere questa parte del corso affermando che il concetto di calcolabilità secondo Turing cattura il concetto intuitivo di calcolabilità, e può a tutti gli effetti essere usato in sua vece.

Esercizi

Esercizio 5.1. Dimostrare che la funzione successore, definita su numeri interi rappresentati in decimale, è Markov-calcolabile.
Esercizio 5.2. Dimostrare che, per ogni n e per ogni k, la funzione selettore U^n_k è Markov-calcolabile.
Esercizio 5.3. Dimostrare che la funzione di prodotto di due numeri naturali è una funzione ricorsiva.
Esercizio 5.4. Dimostrare che la funzione $r : N \to N$ tale che, per ogni numero naturale x, $p(x) = \lfloor\sqrt{x}\rfloor$ è una funzione ricorsiva.
Esercizio 5.5. Dimostrare che la funzione $r : N \to N$ tale che, per ogni numero naturale x, $p(x)$ è uguale all'x-esimo numero primo, è una funzione ricorsiva.
Esercizio 5.6. Completare la dimostrazione del Teorema 5.2.
Esercizio 5.7. Dimostrare che la funzione $f : N \to N$, tale che $f(n) = 1$ se e solo se n è un numero primo, è RAM-calcolabile.
Esercizio 5.8. Dimostrare che la funzione $f : N \to N$, tale che $f(n) = 1$ se e solo se n è un numero primo, è RAM-calcolabile.
Esercizio 5.9. Dimostrare che la funzione $f : N^2 \to N$, tale che $f(n, m) = 1$ se e solo se n e m sono relativamente primi (ovvero se il massimo comune divisore di n e m è uguale a 1), è RAM-calcolabile.
Esercizio 5.10. Dimostrare che la funzione $f : N^4 \to N$, tale che $f(a, b, c, n) = 1$ se e solo se $a^n + b^n = c^n$, è RAM-calcolabile.
Esercizio 5.11. Dimostrare che una funzione calcolabile da una macchina di Turing è Markov-calcolabile.
Esercizio 5.12. Dimostrare che una funzione calcolabile da una macchina di Turing è RAM-calcolabile.