Elementi di Informatica Teorica

Marcello D'Agostino

Dispensa n. 1

Indice

Che cos'è un algoritmo?

Tipi di problemi

Le nostre vite sono circondate da problemi. Acuni di questi problemi sono di facile soluzione, altri richiedono da parte nostra un grande inpegno e, a volte, anche una buona dose di fortuna. In alcuni casi è garantito che esista una soluzione e ciò che è in questione è solo la nostra capacità di trovarla. In altri non è neppure chiaro se i nostri tentativi di risolvere un problema sono destinati a restare vani semplicemente perché si tratta di un problema *insolubile*. Un classico problema insolubile che ha sfidato per secoli generazioni di geometri è il seguente:

QUADRATURA DEL CERCHIO: Costruire un quadrato che abbia la stessa area di un dato cerchio facendo uso esclusivamente di riga e compasso.

Un problema geometrico è risolubile con riga e compasso se la soluzione può sempre essere trovata utilizzando una sequenza finita di operazioni che rientrano tra le seguenti:

1. tracciare una retta per due punti assegnati;
2. costruire una circonferenza di centro e raggio fissati;
3. intersecare due rette;
4. intersecare due circonferenze;
5. intersecare una retta e una circonferenza.

Un numero reale r è costruibile con riga e compasso se lo è un segmento di lunghezza pari al valore assoluto di r, una volta che si sia fissato un segmento di lunghezza unitaria.

Sebbene il problema risalga alle origini della geometria, l'impossibilità di risolverlo venne dimostrata solo nel 1882. Infatti, dato che l'area del cerchio è uguale a $\pi \cdot r^2$, per risolvere il problema bisognerebbe costruire un quadrato di lato $\sqrt{\pi \cdot r^2} = r \cdot \sqrt{\pi}$. Ma, nel 1882, Ferdinand Von Lindemann dimostrò che π è un numero trascendente, cioè non può essere costruito con riga e compasso.

Possiamo pensare a un problema come a a una domanda di carattere generale o *schema di domanda*. Per esempio il problema della quadratura del cerchio consiste nello schema di domanda seguente: "Come si ottiene, usando solo riga e compasso, un quadrato la cui area sia uguale a quella di un cerchio di raggio r?" Si tratta di uno *schema*, e non di una domanda vera e propria, perché non viene specificato il valore assunto dalla variabile r. Un *esempio* del problema è la domanda vera e propria che si ottiene sostituendo la variabile r con un determinato numero, e.g.: "Come si ottiene, usando solo riga e compasso, un quadrato la cui area sia uguale a quella di un cerchio di raggio 1?"

Non vi sarà sfuggita l'analogia con le *proposizioni aperte* di cui abbiamo parlato nel modulo di logica. Naturalmente, un problema può contenere più variabili. I problemi rappresentati dai seguenti schemi di domanda:

Ricordate che una proposizione aperta è l'espressione linguistica che si ottiene da una proposizione sostituendo uno o più nomi con opportuni spazi vuoti o *variabili*.

È x divisibile per y? (1)

Qual è il MCD di k, m e n? (2)

contengono rispettivamente due e tre variabili. Esempi di ciascuno di essi si ottengono sostituendo valori specifici a tutte le variabili (e.g.: "È 15 divisibile per 3?"; "Qual è il MCD di 75, 105 e 450?").

È chiaro che i problemi non riguardano necessariamente variabili numeriche. Per esempio, nei seguenti problemi:

La parola x segue la parola y in ordine alfabetico? (3)

Quali parole italiane si trovano fra x e y in ordine alfabetico? (4)

le variabili sono parole, cioè sequenze di caratteri di un dato alfabeto. Esempi sono: "la parola *zuzzurellone* segue la parola *zwingliano* in ordine alfabetico?"; "Quali parole italiane si trovano fra *zuzzurellone* e *zygion* in ordine alfatetico?".

In generale:

Che cos'è un problema?

Un problema è caratterizzato da uno *schema di domanda* che contiene un certo numero di variabili (i cui valori rappresentano i *dati* del problema) e può essere identificato con l'*insieme di tutti i suoi esempi*, cioè l'insieme di tutte le domande vere e proprie che si ottengono sostituendo alle variabili valori specifici.

Avrete notato che c'è una differenza importante fra i problemi rappresentati dalle domande (1) e (3) da una parte e quelli rappresentati dalle domande (2) e (4) dall'altra. Mentre alla (1) e alla (3) si può rispondere solo con un "sì" oppure con un "no", per rispondere alla (2) e alla (4) bisogna specficare qualcosa — un numero nel primo caso e un elenco di parole nel secondo. I problemi rappresentati da domande del primo tipo si dicono *problemi di decisione*, mentre quelli rappresentati da domande del secondo tipo si dicono *problemi di determinazione.*

Problemi di decisione, problemi di determinazione

Quando un problema non è insolubile, come nel caso del problema della quadratura del cerchio, possiamo sperare di risolverlo caso per caso, sfruttando la nostra intelligenza, la nostra intuizione e contando sulla buona sorte. Ma è certamente meglio se siamo in grado di escogitare un *metodo sistematico* mediante il quale *qualunque agente* possa risolverlo, indipendentemente dal proprio stato mentale e dalle proprie capacità intellettuali. Un metodo di sistematico di questo tipo è quello che comunemente si intende per *algoritmo*.[1]

[1] Il termine "algoritmo" è una corruzione del nome del matematico arabo al-Khuwarizmi vissuto nel IX secolo d.C.

Che cos'è un algoritmo?

Un *algoritmo* è un *procedimento sistematico* per risolvere un determinato *problema* che consiste in una *sequenza finita di istruzioni precise* eseguibili da qualsiasi agente, non importa se umano o meccanico, le quali *infallibilmente* conducono alla soluzione in un *numero finito* di passi.

Spesso gli algoritmi vengono paragonati alle ricette di cucina, ma è un paragone fuorviante. Infatti una tipica ricetta consiste in una sequenza di istruzioni *approssimative*, che lasciano molto spazio all'abilità e alla creatività individuale. In un algoritmo invece, ogni istruzione dovrebbe prescrivere *esattamente* ciò che si deve fare, senza lasciare alcuno spazio al caso o all'intuizione. Vediamo qualche semplice esempio.

Algoritmo per riconoscere i numeri primi

Supponiamo che qualcuno vi chieda se sapete che cos'è un numero primo. La risposta vi è probabilmente nota fin dalle scuole elementari:

> Un numero primo è un numero che è divisibile solo per se stesso e per 1.

Questa definizione caratterizza con precisione l'insieme dei numeri primi come un particolare sottoinsieme degli interi positivi $(1, 2, 3, \ldots)$. In base a questa definizione è facile riconoscere che $2, 5, 7, 11, 13, 17$ e 19 sono tutti numeri primi. Supponete però che vi si chieda: "16433 è un numero primo?".[2] Anche se sapete perfettamente che cos'è un numero primo — ne conoscete la definizione! — probabilmente non sarete in grado di rispondere subito. La definizione di numero primo non comprende, infatti, un algoritmo per "decidere" quali numeri sono primi e quali no. Dunque il fatto che conosciamo la definizione di numero primo, non implica che siamo in grado di risolvere *in generale* il problema di decisione corrispondente allo schema di domanda: "È n un numero primo?", che siamo cioè sempre in grado di rispondere infallibilmente a *tutte* le domande generate da questo schema sostituendo la variabile n con un intero positivo.

[2] La risposta è sì.

In questo caso particolare, tuttavia, è piuttosto banale formulare un algoritmo del genere. Dato un intero positivo n, consideriamo la successione di tutti gli interi da 2 a $n - 1$ e dividiamo n per ciascuno di questi interi. Se a un certo punto il resto della divisione è uguale a 0, possiamo fermarci e rispondere "n non è un numero primo". Altrimenti, se *nessuna* di queste divisioni dà come resto 0, la risposta sarà: "n è un numero primo". Possiamo rappresentare questo algoritmo come una successione di istruzioni nel modo seguente:

0. **INPUT**: un intero positivo n
1. **per tutti** gli interi i compresi fra 2 e $n-1$
   1.1 dividete n per i;
   1.2 **se** il resto è 0, **allora STOP** e **OUTPUT**: "NO" (n non è primo);
2. **STOP** e **OUTPUT**: "SÌ" (n è primo).

Notate che l'istruzione 1 chiede di *ripetere* il blocco di istruzioni 1.1 e 1.2 facendo variare i finché resta vero che il resto della divisione di n per i è uguale a 0, oppure la divisione è stata già eseguita per tutti gli interi compresi fra i e $n-1$. Notate anche che l'istruzione 1.2 è rappresentata da un *condizionale*. Questo significa che l'istruzione specificata nel conseguente (**STOP** e **OUTPUT**: " "NO" (n non è primo)" viene eseguita nel caso in cui si verifica la condizione specificata dall'antecedente (cioè se il resto della divisione è 0), altrimenti non viene eseguita e si passa oltre.

Vediamo come funziona questo algoritmo con due semplici esempio. Consideriamo il caso in cui l'input sia 9 e dunque la risposta deve essre "no", visto che 9 non è un numero primo. I passi eseguiti dall'algoritmo per dare la risposta sono i seguenti:

1. dividete 9 per 2 (esecuzione dell'istruzione 1 con $i = 2$)
2. Il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;
3. dividete 9 per 3 (istruzione 1 con $i = 3$;
4. il resto è 0? Sì, allora l'istruzione nel conseguente della 1.2 deve essere eseguita;
5. **STOP** e **OUTPUT**: "9 non è primo".

Consideriamo ora il caso in cui l'input sia 7 che è invece un numero primo.

1. dividete 7 per 2 (istruzione 1 con $i = 2$);
2. il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;
3. dividete 7 per 3 (istruzione 1 con $i = 3$);
4. il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;
5. dividete 7 per 4 (istruzione 1 con $i = 4$;

6. il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;

7. dividete 7 per 5 (istruzione 1 con $i = 5$);

8. il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;

9. dividete 7 per 6 (istruzione 1 con $i = 6$);

10. il resto è 0? No, dunque l'istruzione nel conseguente della 1.2 non viene eseguita;

A questo punto le isttruzioni 1.1 e 1.2 non vengono più eseguite, perché la variabile i dell'istruzione 1 ha raggiunto il valore massimo $n - 1$. Dunque è terminata l'iterazione delle istruzioni 1.1 e 1.2 prescritta dall'istrtuzione 1 e si esegue l'istruzione successiva, cioè l'istruzione 2. L'algortimo termina con output "7 è un numero primo".

Non è difficile rendersi conto, però, che questa procedura banale è del tutto ridondante. Non c'è infatti alcun bisogno di considerare *tutti* gli interi compresi fra 2 e $n - 1$ come possibili divisori di n. Infatti, se n è divisibile per un intero m (e dunque non è primo) allora deve necessariamente essere divisibile per in intero minore o uguale a $\sqrt{n}$. Lo dimostriamo con un ragionamento per assurdo. Supponiamo che n sia divisibile per un intero, diciamo m, tale che $m > \sqrt{n}$. In tal caso esisterà un intero positivo k tale che $n/m = k$ e dunque $n = m \times k$, per cui n è divisibile anche per k. Ora, questo intero k deve necessariamente essere minore di $\sqrt{n}$. Per dimostrarlo, ragioniamo per assurdo. Supponiamo che $k \geq \sqrt{n}$. In tal caso, dal momento che $m > \sqrt{n}$ e $k \geq \sqrt{n}$, avremmo che $m \times k > \sqrt{n} \times \sqrt{n} > n$. Ma questo è impossibile, perché nessun numero può essere strettamente maggiore di se stesso. Dunque, la nostra supposizione che $k \geq \sqrt{n}$ non può essere vera. Ne segue che, se $n/m = k$ e $m > \sqrt{n}$, deve per forza essere falso che $k \geq \sqrt{n}$ e dunque deve essere vero che $k < \sqrt{n}$. Dato che anche k è un divisore di n (perché $m/k = n$), abbiamo mostrato che se n è divisibile per un intero $m > \sqrt{n}$, deve essere divisibile anche per un intero $k < \sqrt{n}$.

Per i ragionamenti per assurdo, vedi la dispensa n. 4 del modulo di Logica.

In generale, se $x > y$ e $z \geq w$, $x \cdot z > y \cdot w$.

Dunque, per stabilire se n è primo, non è necessario considerare tutti i potenziali divisori compresi fra 2 e $n - 1$ e possiamo limitarci a considerare quelli compresi fra 2 e $\sqrt{n}$. Infatti, se c'è un divisore maggiore $\sqrt{n}$ ce n'è anche uno minore di $\sqrt{n}$ e nell'istruzione 1 possiamo limitarci a considerare gli interi compresi fra 2 e $\sqrt{n}$. Questo ci consente di formulare un nuovo algoritmo *più efficiente* del primo:

0. **INPUT**: un intero positivo n
1. **per tutti** gli interi i compresi fra 2 e $\sqrt{n}$
   1.1 dividete n per i;
   1.2 **se** il resto è 0, **allora OUTPUT**: "NO" (n non è primo);
2. **STOP** e **OUTPUT**: "SÌ" (n è primo).

Questo algoritmo esegue, in generale, molte meno divisioni dell'algoritmo precedente. Se n è 16433 (che è un numero primo), l'algoritmo precedente deve eseguire tutte le divisioni di n per i prescritte dall'istruzione 1 per tutti i valori interi di i compresi fra 2 e 16433, cioè deve eseguire 16431 divisioni. Invece il secondo algoritmo deve eseguire solo le divisioni di n per i per tutti gli interi i compresi fra 2 e $\sqrt{16433} = 128,1912...$, dunque deve eseguire solo 128 divisioni.

Algoritmi espressi da formule

Qualunque formula matematica per calcolare una certa incognita è un algoritmo. Considerate, per esempio, la formula per calcolare la somma dei primi n numeri interi è $\frac{n\times(n+1)}{2}$. Questo significa che il problema di determinazione:

Qual è la somma dei primi n numeri interi?

è risolto dal semplice algoritmo che consiste solo delle due istruzioni seguenti:

0. **INPUT**: un intero positivo n;
1. Moltiplicate n per $n+1$;
2. dividete il risultato per 2;
3. **OUTPUT**: scrivete il risultato.

Un altro esempio è la formula per calcolare l'interesse composto. Se ho un capitale iniziale C che investo a un rendimento annuo i, quale sarà il valore dell'investimento dopo t anni? Supponiamo che C sia 100 euro e che il tasso di interesse sia del 3 per cento. È chiaro che il valore dell'investimento dopo 1 anno è dato da $C_1 = 100 + 100 \times 0,03 = 103$ euro. Il valore dell'investimento dopo 2 anni sarà

$$C_1 + C_1 \times 0,003 = 103 + 103 \times 0,03 = 209,09 \text{ euro.}$$

E così via. L'algoritmo è:

0. **INPUT**: Il capitale iniziale C, il tasso di interesse (in forma decimale) i, la durata dell'investimento (in numero di anni) t;
1. Ponete $C_0 = C$;
2. **Per tutti** i k compresi fra 1 e t
   2.1 ponete $C_k = C_{k-1} + C_{k-1} \times i = C_{k-1} \times (i+1)$.
3. **OUTPUT**: Il valore dell'investimento dopo t anni è uguale a C_t.

In questo caso non è difficile vedere che l'esecuzione di questo algoritmo corrisponde alla sequenza di moltiplicazioni:

$$C \times \underbrace{(i+1) \times \cdots \times (i+1)}_{t \text{ volte}}.$$

Dunque l'algoritmo è catturato dalla seguente formula:

$$C_t = C \times (i+1)^t.$$

Algoritmo per il massimo comun divisore

Come ultimo esempio di algoritmo matematico consideriamo il problema di calcolare il massimo comun divisore di due numeri interi. Il massimo comun divisore di due interi a e b (che non siano entrambi uguali a 0), indicato da $\text{MCD}(a,b)$, è il più grande numero naturale per il quale possono essere divisi entrambi. In linea di principio, è possibile calcolare sempre il MCD eseguendo la scomposizione in fattori primi dei due numeri a e b e prendendo il prodotto di tutti i fattori comuni (considerati una sola volta) con il minimo esponente. L'algoritmo consiste dunque delle seguenti istruzioni:

Algoritmo per il massimo comun divisore

MASSIMO COMUN DIVISORE:

0. **INPUT**: due interi a e b tali che $\neg(a = 0 \wedge b = 0)$ (a e b non sono entrambi uguali a 0);
1. scomponete a e b in fattori primi;
2. sia P uguale al prodotto dei fattori comuni (ciascuno preso una sola volta) con il minimo esponente;
3. **OUTPUT**: $\text{MCD}(a,b) = P$.

Per esempio, per calcolare $\text{MCD}(105, 450)$ si esegue prima la scomposizione in fattori primi di ciascun numero:

$$105 = 3 \times 5 \times 7 \qquad 450 = 2 \times 3^2 \times 5^2.$$

Il MCD è dato dal prodotto dei fattori comuni presi ciascuno con il minimo esponente, in questo caso $3 \times 5 = 15$.

Le istruzioni contenute in questo algoritmo chiedono di eseguire compiti che possono essere considerati complessi: scomporre un numero in fattori primi, prendere il prodotto dei fattori comuni con il minimo esponente. Non sempre si può assumere che l'agente che esegue questo algoritmo sia necessariamente in grado di eseguirle direttamente. Questo è del tutto evidente se l'agente che deve eseguire l'algoritmo è una *macchina*. A meno che la macchina non sia già stata programmata per eseguire questi compiti specifici, non possiamo aspettarci che sia in grado di eseguire l'algoritmo senza ulteriori istruzioni. In generale, il *livello di dettaglio* delle istruzioni contenute in un algoritmo dipende essenzialmente dalle operazioni che riteniamo possano essere eseguite *direttamente* dall'agente che deve utilizzarlo, senza che siano necessarie ulteriori istruzioni. Altrimenti, per ciascuna delle istruzioni che non possono essere eseguite direttamente dall'agente che deve utlizzare l'algoritmo bisogna specificare un *nuovo algoritmo* che spieghi come eseguirle. In questo caso, se l'agente non è in grado di eseguire direttamente la scomposizione in fattori primi, possiamo specificare un *algoritmo ausiliare* per risolvere questo sottoproblema. Questo algoritmo è descritto nella Tabella 1. Non vi preoccupate se non comprendete immediatamente le istruzioni. Tutto risulterà più chiaro quando faremo "girare" l'algoritmo su un esempio concreto. Notate che l'istruzione 4 introduce un blocco di istruzioni, la 4.1 e la 4.2, che devono essere *ripetute* finché resta vero che $N > 1$. E l'istruzione 4.2, a sua volta, chiede di ripetere un blocco di istruzioni, la 4.2.1 e la 4.2.2, finché resta vero che la divisione di N per I dà come resto 0. Questo tipo di istruzioni si chiamano *cicli* o *loop*:

Un *ciclo* o *loop* è un blocco di istruzioni che devono essere ripetute finché resta vera una certa condizione relativa alle variabili introdotte nell'algoritmo.

Definizione di *loop*

Dato che il valore di N si riduce ad ogni esecuzione dell'istruzione 4.2.1, prima o poi $N \mod I$ dovrà per forza diventare diverso da 0, perché, annche nel caso estremo in cui $N = I^k$ per qualche k, a un certo punto N diventerà uguale a 1 e, dato che dalla 4.1 in poi il valore di I è sempre maggiore o uguale a 2, avremo comunque che $1 \mod I = 1$ (il risultato della divisione di 1 per un numero maggiore di 1 è sempre 0 e dunque il resto è uguale a 1). Dunque è garantito che primo o poi la condizione $N \mod I = 0$ diventerà *falsa* e l'algoritmo uscirà dal *loop* lanciato dall'istruzione 4.2. Analogamente, dato che I si incrementa ad ogni applicazione della 4.1 e N si riduce

SCOMPOSIZIONE IN FATTORI PRIMI:

0. **INPUT**: un intero positivo $n \geq 2$;

1. introducete una variabile N e assegnate ad essa il valore n (scriviamo $N := n$);

2. introducete una variabile F e assegnate ad essa la *lista vuota*, cioè la lista che non contiene nessun elemento (scriviamo $F := [\,]$, dove $[\,]$ rappresenta la lista vuota);

3. introducete una variabile I e assegnate ad essa il valore 0 ($I := 0$));

4. **finché** $N > 1$, **ripetete** le seguenti istruzioni:

   4.1 assegnate ad I il nuovo valore costituito dal più piccolo numero primo maggiore del valore corrente di I e minore o uguale a N ($I := \min\{k \mid k \text{ è primo } \wedge k > I \wedge k \leq N\}$);

   4.2 **finché** $N \bmod I = 0$, **ripetete**:

      4.2.1 assegnate a N il valore N/I ottenuto dividendo il valore precedente per I (scriviamo $N := N/I$);

      4.2.2 assegnate ad F la lista ottenuta appendendo il numero I alla fine della lista precedente ($F := F + +[I]$);

5. **OUTPUT**: scrivete il prodotto di tutti i numeri elencati nella lista F.

Tabella 1: Algoritmo per la scomposizione in fattori primi. In generale, con la notazione $\{x \mid x \text{ ha la proprietà } P\}$ si indica l'insieme di tutti gli oggetti x che hanno la proprietà P. Ricordiamo anche che $N \bmod i$ è uguale al resto della divisione di N per i.

ad ogni applicazione della 4.2.1, N prima o poi diventerà uguale a 1. Dunque la condizione dell'istruzione 4 diventerà *falsa* e l'algoritmo uscirà anche dal *loop* lanciato da questa istruzione. (Notate che, nel caso in cui N è primo, l'istruzione 4, nel suo complesso, ci porta a dividere N per N ottenendo 1.) Dunque abbiamo informalmente verificato che il nostro procedimento *termina sempre* dopo un numero finito di passi, una condizione necessaria perché si possa parlare di un algoritmo.

Vediamo adesso come questo algoritmo funziona con un esempio concreto. Consideriamo come valore n di input il numero 450. Abbiamo allora la seguente successione di passi:

0. **INPUT**: 450
1. $N := 450$ (istruzione 1)
2. $F = [\,]$ (istruzione 2)
3. $I := 0$ (istruzione 3)
4. verifichiamo che $N > 1$ (istruzione 4); la risposta è SÌ;
5. $I := 2$ (istruzione 4.1: assegniamo ad I il più piccolo numero primo maggiore di 0 e minore o uguale a 450, cioè 2);
6. verifichiamo che $N \bmod 2 = 0$ (istruzione 4.2); la risposta è SÌ;
7. $N := 450/2 = 225$ (istruzione 4.2.1)
8. $F := [\,] + +[2] = [2]$ (istruzione 4.2.2)
9. verifichiamo che $N \bmod 2 = 0$ (istruzione 4.2); la risposta è NO;
10. verifichiamo che $N > 1$ (istruzione 4); la risposta è SÌ;
11. assegniamo a I il più piccolo numero primo maggiore di 2 e minore o uguale a 225, cioè 3 (istruzione 4.1);
12. verifichiamo che $N \bmod 3 = 0$ (istruzione 4.2); la risposta è SÌ;
13. $N := 225/3 = 75$ (istruzione 4.2.1)
14. $F := [2] + +[3] = [2, 3]$ (istruzione 4.2.2)
15. verifichiamo che $N \bmod 3 = 0$ (istruzione 4.2); la risposta è SÌ;
16. $N := 75/3 = 25$ (istruzione 4.2.1)
17. $F := [2, 3] + +[3] = [2, 3, 3]$ (istruzione 4.2.2)
18. verifichiamo che $N \bmod 3 = 0$ (istruzione 4.2); la risposta è NO;
19. verifichiamo che $N > 1$ (istruzione 4); la risposta è SÌ;

20. assegniamo a I il più piccolo numero primo maggiore di 3 e minore o uguale a 25, cioè 5 (istruzione 4.1);
21. verifichiamo che $N \bmod 5 = 0$ (istruzione 4.2); la risposta è SÌ;
22. $N := 25/5 = 5$ (istruzione 4.2.1)
23. $F := [2,3,3] + +[5] = [2,3,3,5]$ (istruzione 4.2.2)
24. verifichiamo che $N \bmod 5 = 0$ (istruzione 4.2); la risposta è SÌ;
25. $N := 5/5 = 1$ (istruzione 4.2.1)
26. $F := [2,3,3,5] + +[5] = [2,3,3,5,5]$ (istruzione 4.2.2)
27. verifichiamo che $N \bmod 5 = 0$ (istruzione 4.2); la risposta è NO;
28. verifichiamo che $N > 1$ (istruzione 4); la risposta è NO
29. **OUTPUT**: $2 \times 3 \times 3 \times 5 \times 5 = 2 \times 3^2 \times 5^2$.

L'algoritmo che abbiamo illustrato può essere utilizzato per eseguire l'istruzione 1 dell'algoritmo per il massimo comun divisore. Allora quest'ultimo assume la forma seguente:

MASSIMO COMUN DIVISORE:

0. **INPUT**: due interi a e b tali che $\neg(a = 0 \wedge b = 0)$ (a e b non sono entrambi uguali a 0);
1. usate l'algoritmo per la scomposizione in fattori primi con a e b come input; siano P_a e P_b gli output ottenuti;
2. sia P uguale al prodotto dei fattori comuni a P_a e P_b;
3. **OUTPUT**: $\mathrm{MCD}(a,b) = P$.

In questo caso si dice che l'algoritmo per la scomposizione in fattori primi viene *chiamato* dall'algoritmo per il massimo comun divisore come *subroutine*, cioè come algoritmo ausiliare per eseguire un compito che il programma originario riassume in una singola istruzione.

Una *subroutine* è un algoritmo ausiliare che viene utilizzato per eseguire un'istruzione contenuta in un algoritmo principale.

Subroutine

Ma l'analisi non termina qui. Notate che, nell'istruzione 4.1 il nostro algoritmo per la scomposizione in fattori primi assume che l'agente che esegue il calcolo sia in grado di generare il più piccolo numero primo maggiore di un certo numero dato dal valore corrente di I.

Così, anche la subtoutine richiede, a sua volta, un'altra subroutine, specificamente un algoritmo per eseguire l'istruzione 4.1, cioè un metodo per trovare il più piccolo numero primo maggiore di un dato I e minore di un dato N. Un algoritmo di questo tipo verrà discusso nel prossimo paragrafo.

Questo esempio illustra ciò che si intende per approccio *modulare* alla risoluzione dei problemi mediante algoritmi. Progettiamo un algoritmo in cui certe istruzioni danno per scontato che determinati *sottoproblemi* — in questo caso la scomposizione in fattori primi — siano già stati risolti. Poi utilizziamo gli algoritmi per risolvere questi sottoproblemi come *subroutines* chiamate dall'algoritmo principale. All fine l'algoritmo principale sarà simile a un puzzle a incastro dove le subroutine rappresentano i pezzi da sistemare in modo appropriato per comporre la figura desiderata. Ma, dato che anche le subroutine possono essere costruite seguendo questo approccio modulare, cioè chiamando a loro volta altre subroutine, è come se i pezzi stessi del puzzle fossero essi stessi delle figure da comporre con altri pezzi più piccoli.

Approccio modulare

Il crivello di Eratostene

Algoritmi di questo tipo sono noti fin dall'antichità. Uno dei più semplici (anche se piuttosto inefficiente) è il cosiddetto *Crivello di Eratostene* inventato da Eratostene di Cirene (275 a.C., 195 a.C.).

CRIVELLO DI ERATOSTENE

0. **INPUT**: due interi positivi a e b tali che $2 \leq a < b$;
1. scrivete l'elenco di tutti i numeri naturali compresi fra a e b;
2. assegnate questo elenco a una variabile E;
3. **per tutti** gli i compresi fra 2 e $\sqrt{b}$, eseguite la seguente istruzione
   3.1 assegnate ad E il nuovo elenco ottenuto cancellando dal valore corrente di E tutti i multipli di i tranne i stesso;
4. **OUTPUT**: scrivete E.

Crivello di Eratostene

Per esempio, se vogliamo ottenere tutti i numeri primi compresi fra 5 e 30, procediamo nel modo seguente:

0. **INPUT**: $5, 30$;
1. $E := [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]$ (istruzioni 1–2)
2. $E := [5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]$ (istruzione 3.1 con $i = 2$)

3. $E := [5, 7, 11, 13, 17, 19, 23, 25, 29]$ (istruzione 3.1 con $i = 3$)

4. $E := [5, 7, 11, 13, 17, 19, 23, 25, 29]$ (istruzione 3.1 con $i = 4$)

5. $E := [5, 7, 11, 13, 17, 19, 23, 29]$ (istruzione 3.1 con $i = 5$)

6. **OUTPUT**: $[5, 7, 11, 13, 17, 19, 23, 29]$.

Se il Crivello di Eratostene viene utilizzato come subroutine nell'algoritmo per la scomposizione in fattori primi, consente di individuare subito il più piccolo numero primo compreso fra il valore corrente di I e il valore corrente di N. Abbiamo così ottenuto un algoritmo completo? Non necessariamente. Potrebbero esserci altre istruzioni (per esempio l'istruzione 2 della subroutine per il MCD oppure l'istruzione 3.1 del Crivello di Eratostene) che non possono essere eseguite direttamente dall'agente che deve usare l'algoritmo. In tal caso sarà necessario specificare altre subroutine che spieghino come eseguire queste istruzioni, e così via.

Ma questo processo di analisi ha un termine? O c'è il rischio che prosegua all'infinito? È possibile che *ogni* algoritmo, per quanto dettagliato possa essere, richieda la specificazione di algoritmi ausiliari che spieghino come eseguire le sue istruzioni? Questo problema, che possiamo chiamare il *problema del regresso infinito* nell'analisi degli algoritmi, verrà discusso nel prossimo paragrafo.

Problema del regresso infinito

Le macchine di Turing

Il problema del regresso infinito

Nel paragrafo precedente abbiamo brevemente discusso il *problema del regresso infinito* nell'analisi degli algoritmi: è possibile che un algoritmo contenga almento un'istruzione che richiede di eseguire un compito complesso che certi utenti dell'algoritmo non sono in grado di eseguire direttamente. Per esempio è possibilissimo che alcuni agenti non siano in grado di eseguire, senza ulteriori istruzioni, la scomposizione in fattori primi richiesta dall'istruzione 1 dell'algoritmo per il MCD. In tal caso, perché l'algoritmo possa essere eseguito da *qualsiasi* agente, è necesario specificare algoritmi ausiliari, detti *subroutine*, per eseguire ciascuna di queste istruzioni complesse. Ma queste subroutine potrebbero a loro volta contenere istruzioni che non possono essere eseguite direttamente da alcuni agenti e dunque richiederanno la specificazione di ulteriori subroutine, etc. In questo modo, il concetto intuitivo di algoritmo diventa *relativo* all'agente che lo deve eseguire. Una certa procedura sarà un algoritmo per quegli agenti che sono in grado di eseguire direttamente tutte le sue istruzioni e non lo sarà invece per tutti gli altri. Se parliamo di macchine,

il concetto di algoritmo diventerebbe relativo alle operazioni elementari che ciascuna macchina è in grado di eseguire direttamente in virtù della sua struttura fisica. Ma noi vorremmo invece un concetto *assoluto* di algoritmo che non dipenda da specifiche assunzioni sulle operazioni che un certo agente (o una certa macchina) è in grado di eseguire. Questo si può ottenere solo se siamo in grado di specificare un insieme finito di *operazioni elementari* — cioè operazioni talmente semplici che non è possibile "spiegare" la loro esecuzione in termini di operazioni più semplici — che siano (i) eseguibili meccanicamente (cioè possano essere eseguite anche da una macchina) e (ii) sufficienti a definire algoritmi per eseguire qualsiasi istruzione complessa.

Figura 1: Alan Mathison Turing

L'inglese *Alan Mathison Turing* (Figura 1) fu il primo a prendere sul serio questo problema. Turing è considerato il padre dell'informatica e dell'Intelligenza Artificiale in quanto, grazie al concetto di Macchina di Turing, fu in grado di definire con precisione sia le *possibilità* sia i *limiti* delle macchine calcolatrici già intorno alla metà degli anni '30 del XX secolo, prima ancora che fosse costruito un solo computer. La sua tragica biografia si intreccia con le vicende più drammatiche della prima metà del XX secolo. Turing venne perseguitato per la sua omosessualità che nella società inglese fortemente omofobica dell'epoca era considerata un reato (e per questo fu condannato alla castrazione chimica). Durante la seconda guerra mondiale venne arruolato dal *Department of Communications* inglese e contribuì in modo decisivo a decifrare il sistema crittografico *Enigma* usato dai tedeschi per le comunicazioni segrete. Nel 1942, sfruttando l'idea della macchina di Turing, Max Newman progettò il *Colossus*, una macchina in grado di decifrare i codici del sistema crittografico *Lorenz SZ 40/42*, usato nelle comunicazioni tra Hitler e i suoi ufficiali. Turing morì suicida all'età di 44 anni, in circostanze ancora misteriose, ingerendo una mela avvelenata col cianuro.[3]

[3] Per saperne di i potete leggere il libro di Andrew Hodges, *Storia di un Enigma*, Bollati-Boringhieri 1991 e consultare il sito http://www.turing.org.uk/ gestito dalla stesso Hodges.

Che cos'è una macchina di Turing?

Turing definì un classe di macchine astratte — che vennero poi dette *macchine di Turing* — in grado di eseguire *qualsiasi* algoritmo per mezzo delle operazioni più semplici che sia possibile immaginare. La sua idea era quella di simulare le operazioni eseguite da un essere umano quando esegue calcoli con carta e penna. Che cosa facciamo quando eseguiamo un calcolo?

Essenzialmente:

(i) leggiamo dei simboli (cifre o lettere dell'alfabeto) che sono contenuti in determinate "caselle" (i quadretti di un quaderno)

(ii) scriviamo dei simboli in determinate caselle;

(iii) cancelliamo qualche simbolo scritto in precedenza;

(iv) spostiamo il nostro sguardo da una casella all'altra;

(v) ci troviamo, ad ogni passo, in un certo "stato mentale" che indirizza le nostre azioni successive.

Questa idea di fondo è sintetizzata nel concetto di *Macchina di Turing*:

MACCHINA DI TURING

Una *macchina di Turing* consiste di:

- un *nastro infinito* suddiviso in caselle occupate da simboli presi da un *alfabeto finito* $\mathcal{A}$; si assume che questo alfabeto comprenda anche un simbolo speciale "$-$" corrispondente al *carattere vuoto*; (cancellare un simbolo vuol dire sovrascriverlo con il carattere vuoto);
- un *dispositivo di lettura/scrittura* (in breve DLS) che è in grado di
  - leggere il simbolo contenuto nella casella sulla quale è posizionato;
  - sovrascrivere il simbolo contenuto nella casella sulla quale è posizionato con un altro simbolo;
  - spostarsi a destra o a sinistra di una casella alla volta.
- un'*unità di controllo* che contiene le istruzioni per eseguire il calcolo (quello che oggi si dice il "programma").

Macchina di Turing. Vedi Figura 2

In ogni dato stadio del calcolo la macchina si trova in un certo *stato* che appartiene a un certo insieme *finito* di stati possibili che comprendono uno *stato iniziale* (o stato 0) in cui la macchina si trova prima di cominciare. Ciascuna *istruzione* eseguita dalla macchina ha la forma seguente:

> Se la macchina si trova nello stato S e legge il simbolo x, allora entra nello stato S', scrive il simbolo y al posto del simbolo x e si sposta di una casella a destra (oppure di una casella a sinistra).

Una tipica istruzione di una macchina di Turing viene rappresentata da una *quintupla* di questa forma:

(`stato_attuale`, `simbolo_attuale`, `nuovo_stato`, `nuovo_simbolo`, `direzione`)

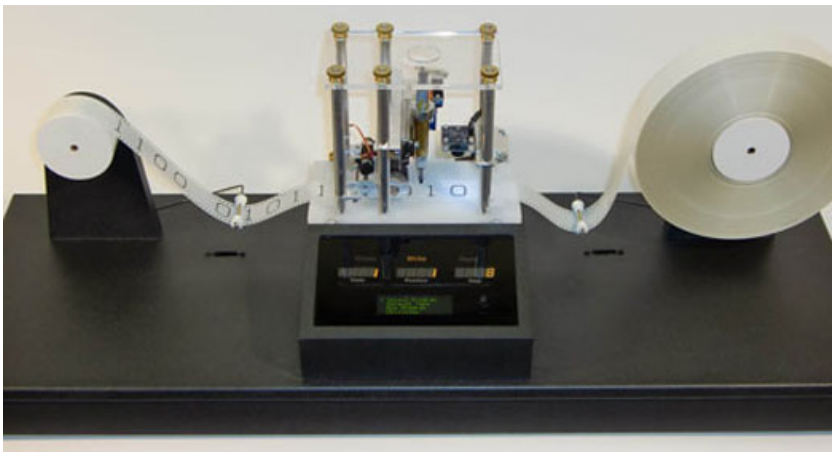

Figura 2: Modello fisico di macchina di Turing. Immagine tratta dal sito A Turing Machine.

Conveniamo di rappresentare la direzione dello spostamento del DLS con "$>$" per indicare che il dispositivo si sposta di una casella a destra e con "$<$" per indicare che si sposta di una casella a sinistra. Così, per esempio, l'istruzione:

$$(0, A, 1, -, >) \tag{5}$$

significa:

> Se il DLS si trova nello stato 0 e legge il simbolo "A", allora entra nello stato 1, cancella il simbolo "A" (lo sovrascrive con il carattere vuoto "$-$") e si sposta di una casella a destra.

Semplici esempi di macchine di Turing

Questa definizione astratta risulterà più comprensibile con qualche esempio. Nella Tabella 2 è definita una semplicissima macchina di Turing che scrive la frase "HELLO" su un nastro inizialmente vuoto (cioè tutti i quadretti sono occupati dal carattere "spazio vuoto"). Qui l'alfabeto finito $\mathcal{A}$ consiste dei simboli "H", "E", "L", "O", "$-$" (il simbolo "$-$" per lo spazio vuoto deve sempre essere presente).

Notate che ciascuna istruzione ha la forma di un condizionale. Le

MT PER SCRIVERE "HELLO".

1. $(0, -, 1, \mathtt{H}, >)$

   se la macchina si trova nello stato 0 e legge il simbolo "$-$", allora entra nello stato 1, scrive il simbolo "H" al posto di "$-$" e si sposta di una casella a destra;

2. $(1, -, 2, \mathtt{E}, >)$

   se la macchina si trova nello stato 1 e legge il simbolo "$-$", allora entra nello stato 2, scrive il simbolo "E" al posto di "$-$" e si sposta di una casella a destra;

3. $(2, -, 3, \mathtt{L}, >)$

   se la macchina si trova nello stato 2 e legge il simbolo "$-$", allora entra nello stato 3, scrive il simbolo "L" al posto di "$-$" e si sposta di una casella a destra;

4. $(3, -, 4, \mathtt{L}, >)$

   se la macchina si trova nello stato 3 e legge il simbolo "$-$", allora entra nello stato 4, scrive il simbolo "L" al posto di "$-$" e si sposta di una casella a destra;

5. $(4, -, 5, \mathtt{O}, >)$

   se la macchina si trova nello stato 4 e legge il simbolo "$-$", allora entra nello stato 5, scrive il simbolo "O" al posto di "$-$" e si sposta di una casella a destra;

Tabella 2: Macchina di Turing per scrivere "HELLO"

azioni descritte nel conseguente vengono eseguite quando sono verificate le condizioni descritte nell'antecedente. Notate anche che non ci sono istruzioni che dicano alla macchina cosa fare quando si trova nello stato 5. Questo significa che, quando ha raggiunto questo stato, la macchina non ha più istruzioni da eseguire e si ferma. In questo caso particolare non c'è nessun input e l'output è costituito da quello che si trova scritto sul nastro quando l'esecuzione del programma è terminata.

Su internet sono disponibili diversi simulatori di macchine di Turing. Una versione che segue le stesse convenzioni che usiamo in queste dispense è quella che si trova alla pagina `http://www.turingsimulator.net/`. Provate a caricare queste istruzioni (una sotto l'altra e nell'ordine dato) nell'apposito riquadro a destra (Figura 3).

Poi basta regolare la velocità di esecuzione usando l'apposito menù a tendina in basso a sinistra e cliccare sul tasto "esegui". Durante

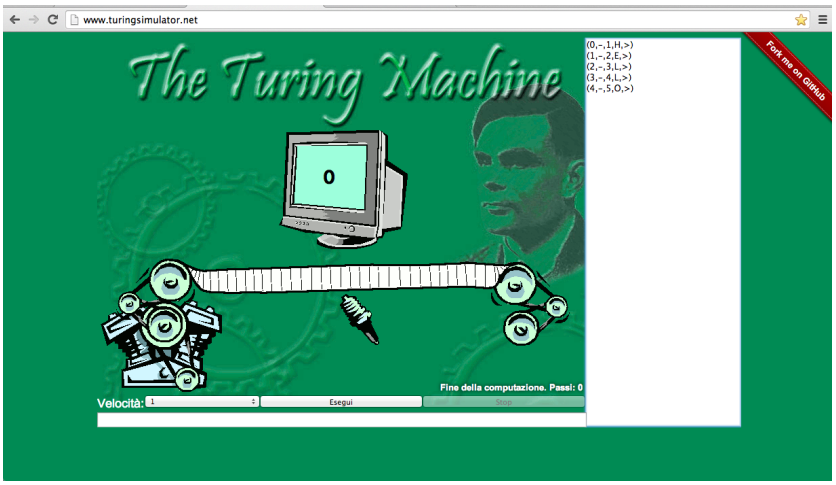

Figura 3: Inserire il programma

l'esecuzione il simulatore evidenzia, ad ogni passo, l'istruzione che viene eseguita in quel momento e lo stato in cui si trova la macchina (sullo schermo del PC al centro della schermata).
Quando la macchina si ferma, sul nastro si trova scritta la parola "HELLO" (Figura 4).

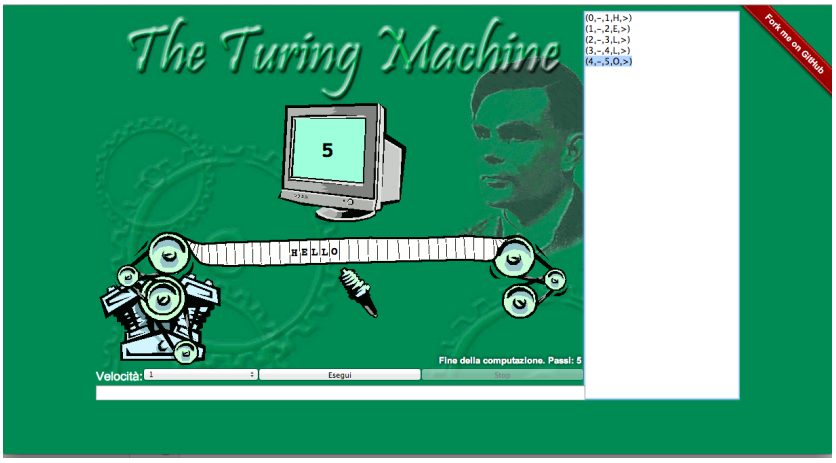

Figura 4: Risultato finale

Un altro semplice esempio è illustrato nella Tabella 3. Definiamo una macchina di Turing che accetti come input una stringa di "A" e "B" e restituisca come output la stringa ottenuta da essa rimpiazzando "A" con "B" e viceversa.
Notate che per definire questa macchina è sufficiente un solo stato.
Notate anche che qui non c'è nessuna istruzione che dice alla macchina cosa fare quando legge il simbolo vuoto "$-$". Questo significa che, quando arriva alla fine della stringa e si trova a leggere il sim-

MT PER INVERTIRE STRINGHE DI "A" E "B"

1. $(0, A, 0, B, >)$

   Se la macchina si trova nello stato 0 e legge il simbolo "A", resta nello stato 0, sostituisce il simbolo "A" con il simbolo "B" e si sposta di una casella a destra;

2. $(0, B, 0, A, >)$

   Se la macchina si trova nello stato 0 e legge il simbolo "B", resta nello stato 0, sostituisce il simbolo "B" con il simbolo "A" e si sposta di una casella a destra;

Tabella 3: MT per invertire stringhe di "A" e "B"

bolo vuoto, la macchina non ha più istruzioni da eseguire e si ferma. In questo caso l'input è una stringa di "A" e "B" (per inserirlo nel simulatore, dopo avere inserito il programma, dovete scriverlo nell'apposito spazio in basso) e l'output è quello che si trova sul nastro quando la macchina si ferma (Figura 5).

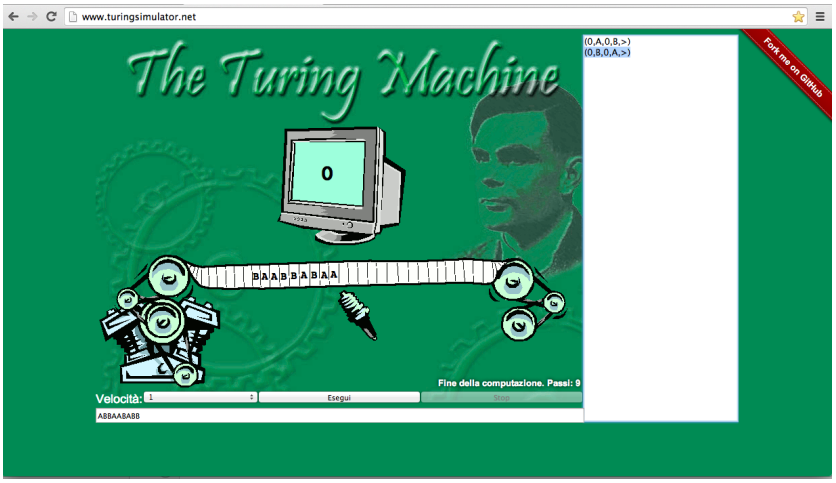

Figura 5: MT per invertire una stringa di "A" e "B"

Come ultimo esempio, nella Tabella 4 è definita una MT in grado di riconoscere le stringhe palindrome composte di "A" e "B" (una stringa è palindroma quando si legge allo stesso modo sia da sinistra a destra, sia da destra a sinistra).

La macchina funziona in questo modo:

1. parte dallo stato 0, legge il primo simbolo, entra in uno stato in cui "si ricorda" quale simbolo ha letto (1 se ha letto "A" e 2 se ha letto "B"), lo cancella e si sposta di una casella a destra (istruzioni 1–2);

2. se ha letto una "A" (e si trova dunque nello stato 1), resta in questo stato, e si sposta a destra di una casella alla volta senza alterare i simboli che legge, finché non arriva alla fine della stringa, cioè fino a quando non legge il simbolo vuoto (istruzioni 3-4); allora entra nello stato 3, e torna indietro di una casella, posizionandosi così sull'ultimo simbolo della stringa (istruzione 5);

   2.1 se l'ultimo simbolo è una "A", allora

      2.1.1 la macchina entra nello stato 5, cancella il simbolo e si sposta di una casella a sinistra (istruzione 9);

      2.1.2 poi si sposta a sinistra di una casella alla volta, rimanendo nello stato 5 e senza alterare i simboli che legge, fino a che non legge il simbolo vuoto (istruzioni 13–14);

      2.1.3 a questo punto torna nello stato 0 e si sposta di una casella a destra, posizionandosi così sul primo simbolo della stringa che è rimasta sul nastro (istruzione 15);

   2.2 se invece l'ultimo simbolo è una "B", allora

      2.2.1 la macchina entra nello stato 6, lascia il simbolo inalterato (cioè lo sovrascrive con lo stesso simbolo) e si sposta di una casella a destra (istruzione 10);

      2.2.2 a questo punto la procedura si arresta perché la macchina non ha nessuna istruzione che le dica cosa fare nello stato 6;

3. se ha letto una "B" (e si trova dunque nello stato 2), resta in questo stato e si sposta a destra di una casella alla volta senza alterare i simboli che legge, finché non arriva alla fine della stringa, cioè fino a quando non legge il simbolo vuoto (istruzioni 6-7); allora entra nello stato 4, e torna indietro di una casella, posizionandosi così sull'ultimo simbolo della stringa (istruzione 8);

   3.1 se l'ultimo simbolo è una "A", allora

      3.1.1 la macchina entra nello stato 7, lascia il simbolo inalterato e si sposta di una casella a destra (istruzione 11);

      3.1.2 a questo punto la procedura si arresta perché la macchina non ha nessuna istruzione che le dica cosa fare nello stato 7;

   3.2 se invece l'ultimo simbolo è una "B", allora

      3.2.1 la macchina entra nello stato 8, cancella il simbolo, e si sposta di una casella sinistra (istruzione 12);

      3.2.2 continua a spostarsi a sinistra, rimanendo nello stato 8 e senza alterare i simboli che legge, fino a che non legge il simbolo vuoto (istruzioni 16–17);

      3.2.3 a questo punto torna nello stato 0 e si sposta di una casella a destra, posizionandosi così sul primo simbolo della stringa che è rimasta sul nastro (istruzione 18);

Il calcolo termina quando la macchina non ha più nessuna istruzione da eseguire, cioè in una delle situazioni seguenti:

- quando si trova nello stato 0 e legge il simbolo vuoto, cioè quando il nastro è rimasto vuoto; in tal caso la stringa è palindroma;
- quando si trova nello stato 3 e legge il simbolo vuoto, cioè quando il nastro è rimasto vuoto; in tal caso la stringa è palindroma;
- quando si trova nello stato 4 e legge il simbolo vuoto, cioè quando il nastro è rimasto vuoto; in tal caso la stringa è palindroma;
- quando si trova nello stato 6; in tal caso rimangono lettere sul nastro e la stringa non è palindroma;
- quando si trova nello stato 7; anche in questo caso rimangono lettere sul nastro e la parola non è palindroma;

Tabella 4: MT per le stringhe palindrome

MT PER RICONOSCERE STRINGHE PALINDROME:

1. $(0, \mathtt{A}, 1, -, >)$
2. $(0, \mathtt{B}, 2, -, >)$
3. $(1, \mathtt{A}, 1, \mathtt{A}, >)$
4. $(1, \mathtt{B}, 1, \mathtt{B}, >)$
5. $(1, -, 3, -, <)$
6. $(2, \mathtt{A}, 2, \mathtt{A}, >)$
7. $(2, \mathtt{B}, 2, \mathtt{B}, >)$
8. $(2, -, 4, -, <)$
9. $(3, \mathtt{A}, 5, -, <)$
10. $(3, \mathtt{B}, 6, \mathtt{B}, >)$
11 $(4, \mathtt{A}, 7, \mathtt{A}, >)$
12. $(4, \mathtt{B}, 8, -, <)$
13. $(5, \mathtt{A}, 5, \mathtt{A}, <)$
14. $(5, \mathtt{B}, 5, \mathtt{B}, <)$
15. $(5, -, 0, -, >)$
16. $(8, \mathtt{A}, 8, \mathtt{A}, <)$
17. $(8, \mathtt{B}, 8, \mathtt{B}, <)$
18. $(8, -, 0, -, >)$