There is a specific problem that is algorithmically unsolvable. Computers appear to be so powerful that you may believe that all problems will eventually yield to them. However, you will be disappointed. The general problem of software verification is not solvable by computer.

given input string. We call it A_{TM} by analogy with A_{DFA} and A_{CFG}. But, whereas A_{DFA} and A_{CFG} were decidable, A_{TM} is not. Let

$$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

THEOREM 4.11

A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus, this theorem shows that recognizers *are* more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM}.

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

- Abbiamo due insiemi A e B e una funzione $f : A \mapsto B$
- f è iniettiva se non mappa mai elementi diversi nello stesso punto: $f(a) \neq f(b)$ ogniqualvolta che $a \neq b$
- f è suriettiva se tocca ogni elemento di B: per ogni $b \in B$ esiste $a \in A$ tale che $f(a) = b$
- Una funzione iniettiva e suriettiva è chiamata biettiva: è un modo per accoppiare elementi di A con elementi di B

Definition

A e B hanno la stessa cardinalità se esiste una funzione biettiva $f : A \mapsto B$

An Undecidable Problem

- A_{TM} = {<M, w> | M is a TM and M accepts w}.

Universal Turing Machine

U = "On input <M, w>, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, ACCEPT; if M ever enters its reject state, REJECT.

Don't be confused by the notion of running a machine on its own description! That is similar to running a program with itself as input, something that does occasionally occur in practice. For example, a compiler is a program that translates other programs. A compiler for the language Python may itself be written in Python, so running that program on itself would make sense. In summary,

$$D(\langle M\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M\rangle. \end{cases}$$

What happens when we run D with its own description $\langle D\rangle$ as input? In that case, we get

$$D(\langle D\rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D\rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could *reject* in this case. However, an algorithm has no way to make this determination, as we shall see.

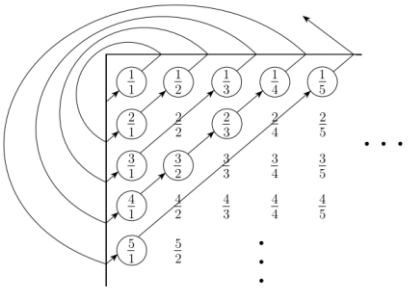


FIGURE 4.16
A correspondence of $\mathcal{N}$ and $\mathcal{Q}$

After seeing the correspondence of $\mathcal{N}$ and $\mathcal{Q}$, you might think that any two infinite sets can be shown to have the same size. After all, you need only demonstrate a correspondence, and this example shows that surprising correspondences do exist. However, for some infinite sets, no correspondence with $\mathcal{N}$ exists. These sets are simply too big. Such sets are called ***uncountable***.

PROOF We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M\rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D.

D = "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs. That is, if H accepts, *reject*; and if H rejects, *accept*."

Where is the diagonalization in the proof of Theorem 4.11? It becomes apparent when you examine tables of behavior for TMs H and D. In these tables we list all TMs down the rows, $M_1, M_2, \ldots$, and all their descriptions across the columns, $\langle M_1\rangle, \langle M_2\rangle, \ldots$. The entries tell whether the machine in a given row accepts the input in a given column. The entry is *accept* if the machine accepts the input but is blank if it rejects or loops on that input. We made up the entries in the following figure to illustrate the idea.

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

In the following figure, we added D to Figure 4.20. By our assumption, H is a TM and so is D. Therefore, it must occur on the list $M_1, M_2, \ldots$ of all TMs. Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$		$\vdots$			$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$		$\vdots$					$\ddots$

Theorem

Un linguaggio è decidibile se solo se è Turing-riconoscibile e co-Turing riconoscibile.

Dimostrazione:

- Dobbiamo dimostrare entrambe le direzioni
- Se A è decidibile, allora sia A che $\overline{A}$ sono Turing-riconoscibili
  - Il complementare di un linguaggio decidibile è decidibile!
- Se A e $\overline{A}$ sono Turing-riconoscibili, possiamo costruire un decisore per A

- Se il complementare di A_{TM} fosse Turing-riconoscibile, allora A_{TM} sarebbe decidibile
- Sappiamo che A_{TM} non è decidibile, quindi il suo complementare non può essere Turing-riconoscibile!

- Una riduzione è un modo per trasformare un problema in un altro problema
- Una soluzione al secondo problema può essere usata per risolvere il primo problema
- Se A è riducibile a B, e B è decidibile, allora A è decidibile
- Se A è riducibile a B, e A è indecidibile, allora A è indecidibile

Sono usate per dimostrare che un problema è indecidibile:

1. Assumi che B sia decidibile
2. Riduci A al problema B
   - costruisci una TM che usa B per risolvere A
3. Se A è indecidibile, allora questa è una contraddizione
4. L'assunzione è sbagliata e B è indecidibile

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}.$$

PROOF Let's assume for the purposes of obtaining a contradiction that TM R decides $HALT_{\mathsf{TM}}$. We construct TM S to decide A_{TM}, with S operating as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Clearly, if R decides $HALT_{\mathsf{TM}}$, then S decides A_{TM}. Because A cidable, $HALT_{\mathsf{TM}}$ also must be undecidable.

Instead, use the assumption that you have TM R that decides $HALT_{\mathsf{TM}}$. With R, you can test whether M halts on w. If R indicates that M doesn't halt on w, reject because $\langle M, w\rangle$ isn't in A_{TM}. However, if R indicates that M does halt on w, you can do the simulation without any danger of looping.

Thus, if TM R exists, we can decide A_{TM}, but we know that A_{TM} is undecidable. By virtue of this contradiction, we can conclude that R does not exist. Therefore, $HALT_{\mathsf{TM}}$ is undecidable.

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

M_1 = "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1\rangle$.
3. If R accepts, *reject*; if R rejects, *accept*."

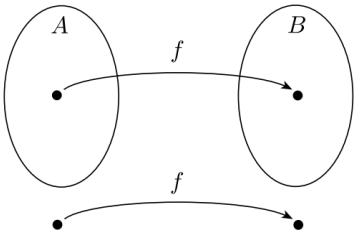
Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it needs only add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

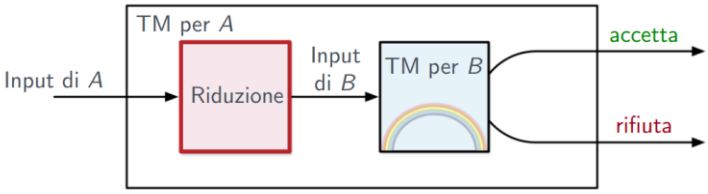
Instead of running R on $\langle M \rangle$, we run R on a modification of $\langle M \rangle$. We modify $\langle M \rangle$ to guarantee that M rejects all strings except w, but on input w it works as usual. Then we use R to determine whether the modified machine recognizes the empty language. The only string the machine can now accept is w, so its language will be nonempty iff it accepts w. If R accepts when it is fed a description of the modified machine, we know that the modified machine doesn't accept anything and that M doesn't accept w.

Un linguaggio A è riducibile mediante funzione al linguaggio B ($A \leq_m B$), se esiste una funzione calcolabile $f : \Sigma^* \mapsto \Sigma^*$ tale che

per ogni w : $w \in A$ se e solo se $f(w) \in B$



f è la riduzione da A a B



2. (12 punti) Data una Turing Machine M, definiamo

$$\text{HALTS}(M) = \{w \mid M \text{ termina la computazione su } w\}.$$

Considera il linguaggio

$$I = \{\langle M \rangle \mid \text{HALTS}(M) \text{ è un insieme infinito}\}.$$

Dimostra che I è indecidibile.

Soluzione. La seguente macchina F calcola una riduzione mediante funzione $A_{TM} \leq_m I$:

F = "su input $\langle M, w \rangle$, dove M è una TM e w una stringa:

1. Costruisci la seguente macchina M':

   M' = "Su input x:
   1. Esegue M su input w.
   2. Se M accetta, *accetta*.
   3. Se M rifiuta, va in loop."

2. Ritorna $\langle M' \rangle$."

Mostriamo che F calcola una funzione di riduzione f da A_{TM} a I, cioè una funzione tale che

$$\langle M, w \rangle \in A_{TM} \text{ se e solo se } M' \in I.$$

THEOREM 5.22

If $A \leq_{\text{m}} B$ and B is decidable, then A is decidable.

COROLLARY 5.23

If $A \leq_{\text{m}} B$ and A is undecidable, then B is undecidable.

- Se $\langle M, w\rangle \in A_{TM}$ allora la macchina M accetta w. In questo caso la macchina M' accetta tutte le parole, quindi $\text{HALTS}(M) = \Sigma^*$ che è un insieme infinito. Di conseguenza $M' \in I$.
- Viceversa, se $\langle M, w\rangle \notin A_{TM}$, allora la macchina M su input w rifiuta oppure va in loop. In entrambi i casi la macchina M' va in loop su tutte le stringhe, quindi $\text{HALTS}(M) = \emptyset$ che è un insieme finito. Di conseguenza $M' \notin I$.

1 Usa una riduzione mediante funzione per dimostrare che $ALL_{TM} = \{\langle M\rangle \mid M \text{ è una TM tale che } L(M) = \Sigma^*\}$ è indecidibile.

Proof. Suppose that ALL_{TM} is decidable by R. Show how to decide A_{TM}.
On input $\langle M, w\rangle$, construct TM M' as follows:

M': "On input x, simulate M on w, accepting if M accepts w".

Now, if M accepts w, then $L(M') = \Sigma^*$; and if M does not accept w, then $L(M') = \emptyset$.
So to decide A_{TM}, do the following:

"On input $\langle M, w\rangle$, construct TM M' defined above. Run R on input $\langle M'\rangle$. If R accepts $\langle M'\rangle$, then Accept; otherwise, Reject."

Since A_{TM} is undecidable, we conclude that R cannot exist. □

3 Se $A \leq_m B$ e B è un linguaggio regolare, allora ciò implica che A è un linguaggio regolare? Perché si o perché no?

Answer: No. For example, define the languages $A = \{\, 0^n 1^n \mid n \geq 0 \,\}$ and $B = \{1\}$, both over the alphabet $\Sigma = \{0, 1\}$. Define the function $f : \Sigma^* \to \Sigma^*$ as

$$f(w) = \begin{cases} 1 & \text{if } w \in A, \\ 0 & \text{if } w \notin A. \end{cases}$$

Observe that A is a context-free language, so it is also Turing-decidable. Thus, f is a computable function. Also, $w \in A$ if and only if $f(w) = 1$, which is true if and only if $f(w) \in B$. Hence, $A \leq_{\text{m}} B$. Language A is nonregular, but B is regular since it is finite.

4. A *useless state* in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a state in a Turing machine is useless. Formulate this problem as a language and show it is undecidable.

Answer: We define the decision problem with the language

$$USELESS_{\text{TM}} = \{\langle M, q\rangle \mid q \text{ is a useless state in TM } M\,\}.$$

We show that $USELESS_{\text{TM}}$ is undecidable by reducing E_{TM} to $USELESS_{\text{TM}}$, where $E_{\text{TM}} = \{\, \langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset \,\}$. We know E_{TM} is undecidable by Theorem 5.2.

Suppose that $USELESS_{\text{TM}}$ is decidable and that TM R decides it. Note that for any Turing machine M with accept state q_{accept}, q_{accept} is useless if and only if $L(M) = \emptyset$. Thus, because TM R solves $USELESS_{\text{TM}}$, we can use R to check if q_{accept} is a useless state to decide E_{TM}. Specifically, below is a TM S that decides E_{TM} by using the decider R for $USELESS_{\text{TM}}$ as a subroutine:

S = "On input $\langle M\rangle$, where M is a TM:
1. Run TM R on input $\langle M, q_{\text{accept}}\rangle$, where q_{accept} is the accept state of M.
2. If R accepts, *accept*. If R rejects, *reject*."

However, because we known E_{TM} is undecidable, there cannot exist a TM that decides $USELESS_{\text{TM}}$.

2. Considera il linguaggio AlwaysDiverge $= \{\langle M \rangle \mid M$ è una TM tale che per ogni $w \in \Sigma^*$ la computazione di M su input w non termina$\}$.

(a) Dimostra che il AlwaysDiverge è indecidibile.

(b) AlwaysDiverge è un linguaggio Turing-riconoscibile, coTuring-riconoscibile oppure né Turing-riconoscibile né coTuring-riconoscibile? Giustifica la tua risposta.

2. (a) Dimostriamo che AlwaysDiverge è un linguaggio indecidibile mostrando che $\overline{A_{TM}}$ è riducibile ad AlwaysDiverge. La funzione di riduzione f è calcolata dalla seguente macchina di Turing:

$F =$ "su input $\langle M, w \rangle$, dove M è una TM e w una stringa:
1. Costruisci la seguente macchina M':
   $M' =$ "su input x:
   1. Se $x \neq w$, vai in loop.
   2. Se $x = w$, esegue M su input w.
   3. Se M accetta, *accetta*.
   4. Se M rifiuta, vai in loop."
2. Restituisci $\langle M' \rangle$."

Dimostriamo che f è una funzione di riduzione da $\overline{A_{TM}}$ ad AlwaysDiverge.

- Se $\langle M, w \rangle \in \overline{A_{TM}}$ allora la computazione di M su w non termina oppure termina rifiutando. Di conseguenza la macchina M' costruita dalla funzione non termina mai la computazione per qualsiasi input. Quindi $f(\langle M, w \rangle) = \langle M' \rangle \in$ AlwaysDiverge.
- Viceversa, se $\langle M, w \rangle \notin \overline{A_{TM}}$ allora la computazione di M su w termina con accettazione. Di conseguenza la macchina M' termina la computazione sull'input w e $f(\langle M, w \rangle) = \langle M' \rangle \notin$ AlwaysDiverge.

Per concludere, siccome abbiamo dimostrato che $\overline{A_{TM}} \leq_m$ AlwaysDiverge e sappiamo che $\overline{A_{TM}}$ è indecidibile, allora possiamo concludere che AlwaysDiverge è indecidibile.

(b) AlwaysDiverge è un linguaggio coTuring-riconoscibile. Dato un ordinamento $s_1, s_2, \ldots$ di tutte le stringhe in Σ^*, la seguente TM riconosce il complementare $\overline{\text{AlwaysDiverge}}$:

$D =$ "su input $\langle M \rangle$, dove M è una TM:
1. Ripeti per $i = 1, 2, 3, \ldots$:
2. Esegue M per i passi di computazione su ogni input $s_1, s_2, \ldots s_i$.
3. Se M termina la computazione su almeno uno, *accetta*. Altrimenti continua."

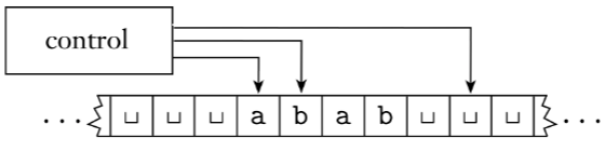
1. Una macchina di Turing a testine multiple è una macchina di Turing con un solo nastro ma con varie testine. Inizialmente, tutte le testine si trovano sopra alla prima cella dell'input. La funzione di transizione viene modificata per consentire la lettura, la scrittura e lo spostamento delle testine. Formalmente,

$$\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R\}^k$$

dove k è il numero delle testine. L'espressione

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli $a_1, \ldots, a_k$ allora la macchina va nello stato q_j, scrive i simboli $b_1, \ldots, b_k$ e muove le testine a destra e a sinistra come specificato.



Un esempio di TM con tre testine.

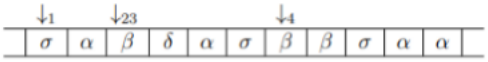
Dimostra che qualsiasi macchina di Turing a testine multiple può essere simulata da una macchina di Turing deterministica a nastro singolo.

SOLUTION We must prove that the k head TM can be simulated by an ordinary TM. Since we already know that a multiple-tape TM can be simulated with an ordinary one, it suffices to simulate the k-head TM with a multiple-tape TM. We call K to the k-tape TM and M to the multiple-tape TM that simulates the former.

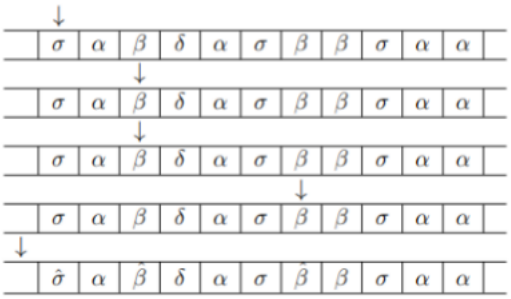
Assume we enumerate each head as h_i with $i = 1, \ldots, k$. We use a $k+1$-tape TM M to do the simulation. The first step is to copy the input tape of K into each tape of M. In the last tape, we call it 'extra', we add a mark to each symbol where there is a head in K (add a mark means that the alphabet of machine M is the union of the alphabet of the machine K plus the set of all the symbols in the alphabet, with a mark).

So, we get the following picture:

Let the machine K be in the following configuration:



Then the machine M is in the following configuration:



At each move in machine K, we do the same movement in the corresponding tape. Then, we check the last (extra) tape to see what are the changes with respect to each tape. We synchronize those changes one by one in the extra tape. For example, if in the tape 2th the 3th cell has been changed from β to σ and the head moves to the right, then after synchronizing such tape, the extra tape will looks as follows

Then when we have to synchronize the 3th tape we compare all the cells except those already synchronized (all except the 3th cell in our example), so the extra tape remain unchanged.

After having synchronized all the tapes. We replicate the extra tape in all the other tapes, which finish the synchronization process and the movement.

1. **(12 punti)** Una *Macchina di Turing con stack di nastri* possiede due azioni aggiuntive che modificano il suo nastro di lavoro, oltre alle normali operazioni di scrittura di singole celle e spostamento a destra o a sinistra della testina: può salvare l'intero nastro inserendolo in uno stack (operazione di Push) e può ripristinare l'intero nastro estraendolo dallo stack (operazione di Pop). Il ripristino del nastro riporta il contenuto di ogni cella al suo contenuto quando il nastro è stato salvato. Il salvataggio e il ripristino del nastro non modificano lo stato della macchina o la posizione della sua testina. Se la macchina tenta di "ripristinare" il nastro quando lo stack è vuoto, la macchina va nello stato di rifiuto. Se ci sono più copie del nastro salvate nello stack, la macchina ripristina l'ultima copia inserita nello stack, che viene quindi rimossa dallo stack.

   Mostra che qualsiasi macchina di Turing con stack di nastri può essere simulata da una macchina di Turing standard. *Suggerimento:* usa una macchina multinastro per la simulazione.

3. **(9 punti)** Dimostra che un linguaggio è decidibile se e solo se esiste un enumeratore che lo enumera seguendo l'ordinamento standard delle stringhe.

Un linguaggio L è decidibile se e solo se esiste una macchina di Turing che accetta L e si ferma per ogni input. Questo significa che per ogni stringa nel linguaggio, la macchina di Turing decide se la stringa appartiene o meno al linguaggio in un numero finito di passaggi.

Un enumeratore per un linguaggio L è una macchina di Turing che elenca tutte le stringhe in L in ordine lessicografico. Questo significa che l'enumeratore genera tutte le stringhe in L una dopo l'altra, in ordine crescente secondo l'ordinamento standard delle stringhe.

Ora, dimostriamo che un linguaggio L è decidibile se e solo se esiste un enumeratore che lo enumera seguendo l'ordinamento standard delle stringhe.

1. **(⇒) Se L è decidibile, allora esiste un enumeratore che enumera L in ordine standard:**

Se L è decidibile, allora esiste una macchina di Turing M che decide L. Possiamo costruire un enumeratore E per L come segue: E genera tutte le stringhe in ordine lessicografico e per ogni stringa generata, esegue M su di essa. Se M accetta la stringa, allora E la stampa. In questo modo, E enumera tutte le stringhe in L in ordine lessicografico.

2. **(⇐) Se esiste un enumeratore che enumera L in ordine standard, allora L è decidibile:**

Se esiste un enumeratore E che enumera L in ordine lessicografico, allora possiamo costruire una macchina di Turing M che decide L come segue: dato un input x, M esegue E e confronta ogni stringa stampata da E con x. Se E stampa una stringa che è uguale a x, allora M accetta x. Se E stampa una stringa che è maggiore di x nell'ordinamento lessicografico, allora M rifiuta x. In questo modo, M decide se una data stringa appartiene o meno a L.

Quindi, un linguaggio L è decidibile se e solo se esiste un enumeratore che enumera L seguendo l'ordinamento standard delle stringhe. Questa è una dimostrazione ad alto livello e l'implementazione effettiva richiederebbe una gestione accurata delle transizioni di stato e dei movimenti del nastro. Ma dal punto di vista teorico, questa dimostrazione mostra l'equivalenza tra la decidibilità di un linguaggio e l'esistenza di un enumeratore per il linguaggio.