Introduction to the Theory of Computation

Jingde Cheng

Saitama University

Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility (Turing-Reducibility)
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Computation: What Is It?

♣ Computation [The Oxford English Dictionary, 2nd Ed, OUP]

- "The action or process of computing, reckoning, or counting; a method or system of reckoning; arithmetical or mathematical calculation."

♣ Fact (September 2019)

- No word item in the Encyclopædia Britannica, OUP Dictionary of Computer Science, IEEE Standard Computer Dictionary.
- What can you think of from this fact?

♣ Fact (at present)

- There is no explicit definition for "computation" that is accepted by all disciplines/scholars.

Computation: What Is It?

♣ The historical origin of "computation"

- The "*decision problem*" (*Entscheidungsproblem*), proposed by Hilbert in 1928, asks for an *effective method* to determine the validity of any given formula in classical predicate calculus.
- Effective method: effective procedure, or effectively calculable, means *a FINITE step way (algorithm)*.

判定问题

有限步骤

♣ Logic is the father/mother of Computer Science

- It is the "decision problem" leads to the notion and/or concept of "computation", and motivates research on computability.

Computation: What Is It?

♣ The state-of-the-art of computation

- Early equivalent models of computation: *Recursive functions* (Herbrand, Godel, Kleene, 1931-1936), *Lambda calculus* (Church, Kleene, 1933-1935), ***Turing machine*** **(Turing, 1936-1937),** *Post systems* (Post, 1936-1943).
- Some *Turing-complete* and/or *Turing-equivalent* models of computation: *Combinatory logic*, *Rewriting systems*, *Register machines*, …

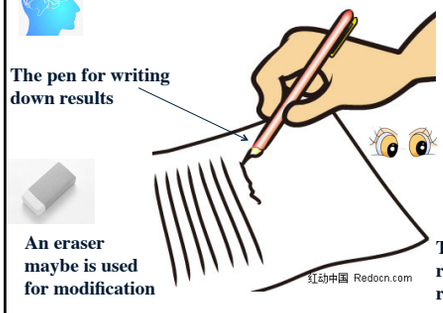
♣ Other models of computation?

- Is there some model of computation that is more "powerful" than Turing-equivalent models?

有存观点的

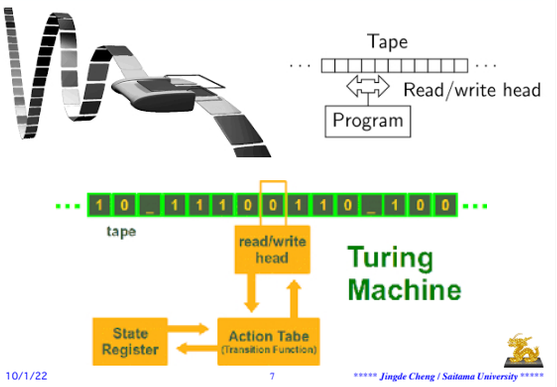
How Do We Human Compute?

Key point: We have a brain to direct/control computation



Other ?

Turing Machines: A Theoretical Computation Model



Turing Machines: The Origins [B-CS-11]

The Origins of Turing Machines

Alan Turing developed the concept of a Turing machine in the 1930s, well before technology was capable of providing the machines we know today. In fact, Turing's vision was that of a human performing computations with pencil and paper. Turing's goal was to provide a model by which the limits of "computational processes" could be studied. This was shortly after the publication in 1931 of Gödel's famous paper exposing the limitations of computational systems, and a major research effort was being directed toward understanding these limitations. In the same year that Turing presented his model (1936), Emil Post presented another model (now known as Post production systems) that has been shown to have the same capabilities as Turing's. As a testimony to the insights of these early researchers, their models of computational systems (such as Turing machines and Post production systems) still serve as valuable tools in computer science research.

Turing Machines: A Theoretical Computation Model

无穷纸带 有穷字符集

- Tape 纸带
  - [D] The TM model uses *an infinite tape* that is divided into *infinite cells*, each of which can contain any one of *a finite set of symbols*, as its unlimited memory.
  - The type can be infinite at one end or infinite at both ends.
- Read/Write head 读/写头
  - [D] The TM model has *a read/write head* that can READ and WRITE symbols and MOVE on the tape, its action is controlled by the control unit.
- Control unit 控制单元
  - [D] The TM model has *a control unit* that controls actions performed by the read/write head.

读写头 一格一格移动

Turing Machines: A Theoretical Computation Model

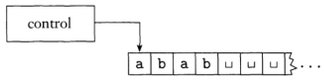
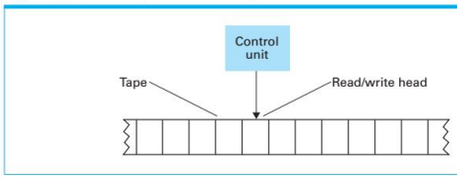


FIGURE **3.1**
Schematic of a Turing machine

Figure 12.2 The components of a Turing machine



Turing Machines: A Theoretical Computation Model

读写头的位置 纸带的内容

- States (Configurations) of a TM 图灵机的状态
  - [D] The position of the read/write head and the contents of the tape of a TM consists a *state (configuration)* of that TM.
- Initial state (configuration) 初态
  - [D] A TM has *an initial state* such that the tape contains only *the input string* and is blank everywhere else. A computation process of a TM starts from its initial state.
- Accept and reject states (configurations) 接受、拒绝态
  - [D] A TM has two specified states as *outputs*.
  - [D] A computation process of a TM either halts when it enters the *accept state* or *reject state* and produces an output ("*accept*" or "*reject*"), or will go on forever, *never halting*.
  - Note: Three possible result cases of a computation process of a TM, accept and halt, reject and halt, and never halting.

图灵机只有三种可能状态：接受、拒绝、永不停止

Turing Machines: A Theoretical Computation Model

- Computation process 计算过程
  - [D] A *computation process* of a TM (with a "program") consists of a (finite or infinite) sequence steps that are executed by the TM's control unit.
  - The computation process starts from the initial state.
  - [D] Each *computation step* consists of reading the symbol in the current tape cell, writing a symbol into that cell, and moving the head one cell to the left or right.
- "Result" of a computation process
  - A computation process of a TM either halts when it enters the accept state or reject state and produces an output ("accept" or "reject"), or will go on forever, never halting.

接受、拒绝 ↑ 这个序列可以有穷也可无穷 永不停止

计算步骤有：①读 ②写 ③移动

Turing Machines vs. Finite Automata

- **The differences between TMs and FAs**
  - A TM can both write on the tape and read from it.
  - The read–write head of a TM can move both to the left and to the right.
  - The tape of a TM is infinite.
  - The special states of a TM for rejecting and accepting take effect immediately.
- **The ability of "computation"**
  - The TMs are more powerful than FAs.

图灵机比有穷自动机更强大

An Example of Turing Machine: M_1 [S-ToC-13]

- **TM M_1**
  - Let M_1 be a TM for testing membership in the language $B = \{ w\#w \mid w \in \{0, 1\}^* \}$, i.e., we want M_1 to accept if its input is a member of B, two identical strings separated by a # (hash) symbol, and to reject otherwise.
  - Ex: 110101#110101, accept; 110101#101011, reject.
- **The working strategy**
  - To move back and forth to the corresponding places on the two sides of the # and determine whether they match.
  - To mark places (with some symbol) on the tape to keep track of which places correspond.

An Example of Turing Machine: M_1 [S-ToC-13]

- **Let M_1 work in the following way**

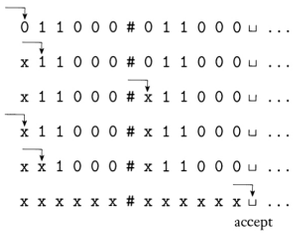


FIGURE **3.2**
Snapshots of Turing machine M_1 computing on input 011000#011000

An Example of Turing Machine: M_1 [S-ToC-13]

We design M_1 to work in that way. It makes multiple passes over the input string with the read–write head. On each pass it matches one of the characters on each side of the # symbol. To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined. If it crosses off all the symbols, that means that everything matched successfully, and M_1 goes into an accept state. If it discovers a mismatch, it enters a reject state. In summary, M_1's algorithm is as follows.

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*."

An Example of Turing Machine: A Variant of M_1

- **TM M_1'**
  - Let M_1' be a TM for testing membership in the language $B = \{ w\#w^R \mid w \in \{0, 1\}^* \}$, i.e., we want M_1' to accept if its input is a member of B, a string w and its reverse separated by a # (hash) symbol, and to reject otherwise.
  - Ex: 110101#101011, accept; 110101#110101, reject.
- **Homework**
  - Consider a working strategy of M_1'.
  - Investigate how many different working strategies might exist.

Turing Machines: Formal Definition [S-ToC-13]

DEFINITION **3.3**

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

- **A more accurate/exact definition of transition function**
  - $\delta =_{\text{df}} ((Q - \{q_{accept}, q_{reject}\}) \times \Gamma) \rightarrow ((Q \times \Gamma) \times \{\text{L}, \text{R}\})$ (Why?)

Turing Machines: Formal Definition

- **State set, Input alphabet, and Tape alphabet**
  - *State set* Q, *Input alphabet* Σ, and *Tape alphabet* Γ all are finite sets.
  - $\Sigma \subset \Gamma$, $\square \notin \Sigma$, $\square \in \Gamma$.
- **Transition function**
  - *Transition function* $\delta = (Q \times \Gamma) \rightarrow ((Q \times \Gamma) \times \{L, R\})$ is a function from the Cartesian (direct) product of Q and Γ to the Cartesian (direct) product of $(Q \times \Gamma)$ and $\{L, R\}$.
  - $(q_i, a_j) \rightarrow (q_{i+1}, a_{j+1}, L|R)$ means that for a state q_i and the current symbol a_j, the next state and symbol will be q_{i+1} and a_{j+1}, and the next position will be one cell left (L) or right (R) of the current position.
  - It is the transition function δ that represents the computation process.

Turing Machines: Formal Definition

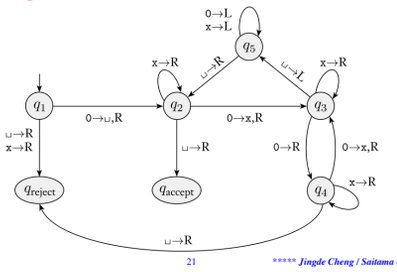
- **Question**
  - How to represent a transition function δ by a state transition table?
- **Hint example**
  - Is the following table sufficient to represent a transition function?

A table representation of transition function

	0	1	*x*	
q_1	1, q_2			
q_2		q_{accept}		
q_3			q_{reject}	

Turing Machines: Formal Definition

- **Question**
  - How to represent a transition function δ by a directed graph?
- **Hint example**



Turing Machines: Languages of Turing Machines

- **Definition: Language of Turing machine**
  - [D] The set of strings that M accepts is called *the language of M*, or *the language recognized by M*, denoted *$L(M)$*.
  - $L(M) =_{df} \{ \omega \mid \omega \in \Sigma^* \text{ and } M \text{ accepts } \omega \}$
- **Notes**
  - The language of a TM is an intrinsic property of that TM.
  - $L(M)$ may be an infinite (countable) set.

Turing Machines: Formal Definition [L-ToC-17]

DEFINITION 9.1

A Turing machine M is defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F),$$

where

Q	is the set of internal states,
Σ	is the input alphabet,
Γ	is a finite set of symbols called the **tape alphabet**,
δ	is the transition function,
$\square \in \Gamma$	is a special symbol called the **blank**,
$q_0 \in Q$	is the initial state,
$F \subseteq Q$	is the set of final states.

In the definition of a Turing machine, we assume that $\Sigma \subseteq \Gamma - \{\square\}$, that is, that the input alphabet is a subset of the tape alphabet, not including the blank. Blanks are ruled out as input for reasons that will become apparent shortly. The transition function δ is defined as

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}.$$

Turing Machines: Formal Definition [B-CS-11]

EXAMPLE 9.1

Figure 9.2 shows the situation before and after the move

$$\delta(q_0, a) = (q_1, d, R).$$

FIGURE 9.2 The situation (a) before the move and (b) after the move.

Languages of Turing Machines [L-ToC-17]

♣ **Language of Turing machine**

DEFINITION 9.3

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine. Then the language accepted by M is

$$L(M) = \left\{ w \in \Sigma^+ : q_0 w \overset{*}{\vdash} x_1 q_f x_2 \text{ for some } q_f \in F, x_1, x_2 \in \Gamma^* \right\}.$$

The symbol $\overset{*}{\vdash}$ has the usual meaning of an arbitrary number of moves.

♣ **The equivalence between two classes of machines**

DEFINITION 10.1

Two automata are equivalent if they accept the same language. Consider two classes of automata C_1 and C_2. If for every automaton M_1 in C_1 there is an automaton M_2 in C_2 such that

$$L(M_1) = L(M_2),$$

we say that C_2 is at least as powerful as C_1. If the converse also holds and for every M_2 in C_2 there is an M_1 in C_1 such that $L(M_1) = L(M_2)$, we say that C_1 and C_2 are equivalent.

Turing Machines: Formal Definition [HS-ToC-11]

Formally, a Turing machine is a system

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, q_{\text{accept}}, q_{\text{reject}} \rangle,$$

where

Q is the finite set of states,
Γ is the finite tape alphabet,
$B \in \Gamma$, the *blank*,
Σ is the input alphabet, $\Sigma \subseteq \Gamma - \{B\}$,
δ is the transition function,
$q_0 \in Q$ is the *initial state*,
q_{accept} is the accepting state, and
q_{reject} is the rejecting state.

The next move is determined by a *transition* function

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

To avoid confusion, we usually take $Q \cap \Gamma = \emptyset$.

Turing Machines: Formal Definition [HS-ToC-11]

Definition 2.1. Let M be a Turing machine. The language *accepted* by M is

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

Definition 2.3. Two Turing machines are *equivalent* if they accept the same language.

Turing Machines: Computation [S-ToC-13]

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ computes as follows. Initially M receives its input $w = w_1 w_2 \ldots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank (i.e., filled with blank symbols). The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states at which point it halts. If neither occurs, M goes on forever.

图灵机 $M=(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$的计算方式如下：开始时，$M$以最左边的$n$个带方格接收输入$w=w_1 w_2 \cdots w_n \in \Sigma^*$，带的其余部分保持空白(即填以空白符)，读写头从最左边的带方格开始运行，注意Σ不含空白符，故出现在带上的第一个空白符表示输入的结束。M开始运行后，计算根据转移函数所描述的规则进行，如果M试图将读写头从带的最左端再向左移出，即使转移函数指示的是L，读写头也停在原地不动。计算一直持续到它进入接受或拒绝状态，此时停机，如果二者都不发生，则M将永远运行下去。

Computation of Turing Machines: Formal Definition

♣ **Configuration**

- **As a TM computes, changes occur in the current state, the current tape contents, and the current head location.**
- **A setting of these three items is called *a configuration* of the TM.**

♣ **Representation of configuration**

- **For a state *q* and two string *u* and *v* over the tape alphabet Γ, we write *u q v* for the configuration where the current state is *q*, the current tape contents is *uv*, and the current head location is the first symbol of *v*.**
- **The tape contains only blanks following the last symbol of *v*.**

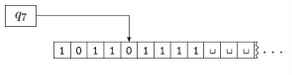


FIGURE 3.4
A Turing machine with configuration $1011q_701111$

Computation of Turing Machines: Formal Definition

♣ **Configuration transition: regular cases**

- **We say that configuration C_1 *yields* configuration C_2 if the TM can legally go from C_1 to C_2 in a single step.**
- **For *a*, *b*, and *c* in Γ, *u* and *v* in Γ^*, and states q_i and q_j,**
  ***ua* q_i *bv* yields *u* q_j *acv*, if δ(q_i, *b*) = (q_j, *c*, L), and**
  ***ua* q_i *bv* yields *uac* q_j *v*, if δ(q_i, *b*) = (q_j, *c*, R).**

♣ **Note**

- **The TM can legally go from C_1 to C_2 in a "SINGLE" step.**

Computation of Turing Machines: Formal Definition

♣ **Configuration transition: special case of the left-hand end**

- For the special case of left-hand end,
  $q_i\, bv$ yields $q_j\, cv$, if $\delta(q_i, b) = (q_j, c, \mathrm{L})$
  (because we prevent the machine from going off the left-hand end of the tape),
  $q_i\, bv$ yields $c\, q_j\, v$, if $\delta(q_i, b) = (q_j, c, \mathrm{R})$.

♣ **Configuration transition: special case of the right-hand end**

- For the special case of right-hand end,
  "$ua\ q_i$" is equivalent to "$ua\ q_i\ \square$"
  because we assume that blanks follow the part of the tape represented in the configuration. Thus we can handle this case as the regular case, with the head no longer at the right-hand end.

Computation of Turing Machines: Formal Definition

♣ **Start configuration**

- *The start configuration* of a TM on input w is the configuration q_0w, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape.

♣ **Accepting configuration and Rejecting configuration**

- In *an accepting configuration* of a TM, the state of the configuration is q_{accept}.
- In *a rejecting configuration* of a TM, the state of the configuration is q_{reject}.
- Accepting configuration and rejecting configuration are *halting configurations* and do not yield further configurations.

Computation of Turing Machines: Formal Definition [L-ToC-17]

DEFINITION 9.2

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine. Then any string $a_1 \cdots a_{k-1} q_i a_k a_{k+1} \cdots a_n$, with $a_i \in \Gamma$ and $q_i \in Q$, is an instantaneous description of M. A move

$$a_1 \cdots a_{k-1} q_i a_k a_{k+1} \cdots a_n \vdash a_1 \cdots a_{k-1} b q_2 a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_i, a_k) = (q_2, b, R).$$

A move

$$a_1 \cdots a_{k-1} q_i a_k a_{k+1} \cdots a_n \vdash a_1 \cdots q_2 a_{k-1} b a_{k+1} \cdots a_n$$

is possible if and only if

$$\delta(q_i, a_k) = (q_2, b, L).$$

M is said to halt starting from some initial configuration $x_1 q_i x_2$ if

$$x_1 q_i x_2 \overset{*}{\vdash} y_1 q_j a y_2$$

for any q_j and a, for which $\delta(q_j, a)$ is undefined. The sequence of configurations leading to a halt state will be called a **computation**.

The symbol $\overset{*}{\vdash}$ has the usual meaning of an arbitrary number of moves.

Computation of Turing Machines: TMs as Transducers [L-ToC-17]

The input for a computation will be all the nonblank symbols on the tape at the initial time. At the conclusion of the computation, the output will be whatever is then on the tape. Thus, we can view a Turing machine transducer M as an implementation of a function f defined by

$$\widehat{w} = f(w),$$

provided that

$$q_0 w \overset{*}{\vdash}_M q_f \widehat{w},$$

for some final state q_f.

DEFINITION 9.4

A function f with domain D is said to be **Turing-computable** or just **computable** if there exists some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ such that

$$q_0 w \overset{*}{\vdash}_M q_f f(w), \qquad q_f \in F,$$

for all $w \in D$.

An Example of Turing Machine: Formal Description of M_1 [S-ToC-13]

The following is a formal description of $M_1 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$, the Turing machine that we informally described (page 167) for deciding the language $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_8, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, and $\Gamma = \{0,1,\#,\mathtt{x},\sqcup\}$.
- We describe δ with a state diagram (see the following figure).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}, respectively.

An Example of Turing Machine: State Transition of M_1 [S-ToC-13]

- **Homework: Show a state transition table of the state transition diagram of M_1.**
- **Note: To simplify the figure, we do not show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol.**

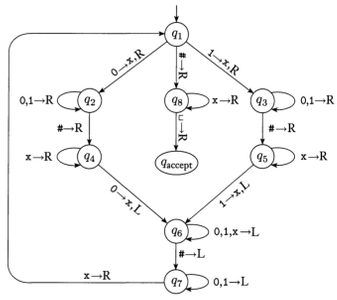


FIGURE 3.10
State diagram for Turing machine M_1

An Example of Turing Machine: State Transition of M_1 [S-ToC-13]

In Figure 3.10, which depicts the state diagram of TM M_1, you will find the label 0,1→R on the transition going from q_3 to itself. That label means that the machine stays in q_3 and moves to the right when it reads a 0 or a 1 in state q_3. It doesn't change the symbol on the tape.

Stage 1 is implemented by states q_1 through q_6, and stage 2 by the remaining states. To simplify the figure, we don't show the reject state or the transitions going to the reject state. Those transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol. Thus, because in state q_5 no outgoing arrow with a # is present, if a # occurs under the head when the machine is in state q_5, it goes to state q_{reject}. For completeness, we say that the head moves right in each of these transitions to the reject state.

An Example of Turing Machine: M_1

- **Acceptable strings**
  - **0#0, 1#1, 01#01, …**
- **Computation process examples**

Input: 0#0	Input: 1#1	Input: 01#01	
$q_1$0#0□	$q_1$1#1□	$q_1$01#01□	xx#x$q_5$1□
xq_2#0□	xq_3#1□	x$q_2$1#01□	xx#$q_6$xx□
x#$q_4$0□	x#$q_5$1□	x1q_2#01□	xxq_6#xx□
xq_6#x□	xq_6#x□	x1#$q_4$01□	xq_7x#xx□
q_7x#x□	q_7x#x□	x1q_6#x1□	xxq_1#xx□
xq_1#x□	xq_1#x□	x$q_7$1#x1□	xx#$q_8$xx□
x#q_8x□	x#q_8x□	q_7x1#x1□	xx#xq_8x□
x#xq_8□	x#xq_8□	x$q_1$1#x1□	xx#xxq_8□
x#x□q_{accept}	x#x□q_{accept}	xxq_3#x1□	xx#xx□q_{accept}
		xx#q_5x1□	

An Example of Turing Machine: M_1

- **Strings that must be rejected**
  - **0#1, 1#0, 01#10, …**
- **Computation process examples**

Input: 0#1	Input: 1#0	Input: 01#10
$q_1$0#1□	$q_1$1#0□	$q_1$01#10□
xq_2#1□	xq_3#0□	x$q_2$1#10□
x#$q_4$1□	x#$q_5$0□	x1q_2#10□
x#1q_{reject}□	x#0q_{reject}□	x1#$q_4$10□
		x1#1q_{reject}0□

An Example of Turing Machine: M_2 [S-ToC-13]

Here we describe a Turing machine (TM) M_2 that decides $A = \{0^{2^n} \mid n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

M_2 = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

Each iteration of stage 1 cuts the number of 0s in half. As the machine sweeps across the tape in stage 1, it keeps track of whether the number of 0s seen is even or odd. If that number is odd and greater than 1, the original number of 0s in the input could not have been a power of 2. Therefore the machine rejects in this instance. However, if the number of 0s seen is 1, the original number must have been a power of 2. So in this case the machine accepts.

An Example of Turing Machine: M_2

- **Acceptable strings (n = 0, 1, 2, 3, …)**
  - **0 (2^0 = 1), 00 (2^1 = 2), 0000 (2^2 = 4), 00000000 (2^3 = 8), …**
  - **Numbers of zero: 1, 2, 4, 8, 16, 32, …**
  - **Decision method: halve the input string, halve the remaining string, and halve the remaining string, …, until only one 0.**
- **Strings that must be rejected**
  - **000, 00000, 000000, 0000000, 000000000, 0000000000, …**
  - **Numbers of zero: 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, …**
  - **Decision method: The input string 000 should be rejected. Delete 0s at all even positions of the input string, if the length of the remaining string is an odd number greater than 2, then the input string should be rejected; if the length of the remaining string is an even number greater than 2, continue to delete 0s at all even positions of the remaining string; …**

An Example of Turing Machine: M_2 [S-ToC-13]

Now we give the formal description of $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{\text{accept}}, q_{\text{reject}})$:

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, x, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.

An Example of Turing Machine: M_2 [S-ToC-13]

- Homework: Show a state transition table of the state transition diagram of M_2.
- Note: $(q_1, x) \rightarrow (q_{reject}, x, R)$ is a misprint.

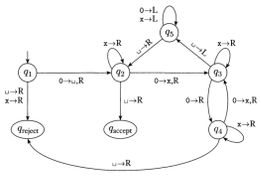


FIGURE 3.8
State diagram for Turing machine M_2

In this state diagram, the label $0 \rightarrow \sqcup, R$ appears on the transition from q_1 to q_2. This label signifies that, when in state q_1 with the head reading 0, the machine goes to state q_2, writes $\sqcup$, and moves the head to the right. In other words, $\delta(q_1, 0) = (q_2, \sqcup, R)$. For clarity we use the shorthand $0 \rightarrow R$ in the transition from q_3 to q_4, to mean that the machine moves to the right when reading 0 in state q_3 but doesn't alter the tape, so $\delta(q_3, 0) = (q_4, 0, R)$.

An Example of Turing Machine: M_2

- Acceptable strings (n = 0, 1, 2, 3, …)
  - 0 ($2^0 = 1$), 00 ($2^1 = 2$), 0000 ($2^2 = 4$), 00000000 ($2^3 = 8$), …
  - Numbers of zero: 1, 2, 4, 8, 16, 32, …
- Computation process examples

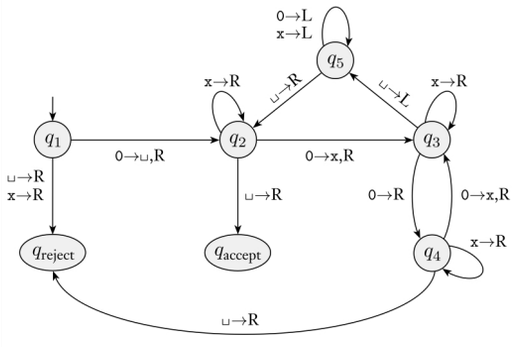
Input: 0 (n = 0)

$q_1 0 \square$
$\square q_2 \square$
$\square\square q_{accept}$

Input: 00 (n = 1)

$q_1 00 \square$
$\square q_2 0 \square$
$\square x q_3 \square$
$\square q_5 x \square$
$q_5 \square x \square$
$\square q_2 x \square$
$\square x q_2 \square$
$\square x \square q_{accept}$

An Example of Turing Machine: M_2 [S-ToC-13]



An Example of Turing Machine: M_2

- Strings that must be rejected
  - Numbers of zero: 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, …
- Computation process examples

Input: 000

$q_1 000 \square$
$\square q_2 00 \square$
$\square x q_3 0 \square$
$\square x 0 q_4 \square$
$\square x 0 \square q_{reject}$

Input: 00000

$q_1 00000 \square$
$\square q_2 0000 \square$
$\square x q_3 000 \square$
$\square x 0 q_4 00 \square$
$\square x 0 x q_3 0 \square$
$\square x 0 x 0 q_4 \square$
$\square x 0 x 0 \square q_{reject}$

Input: 000000

$q_1 000000 \square$
$\square q_2 00000 \square$
$\square x q_3 0000 \square$
$\square x 0 q_4 000 \square$
$\square x 0 x q_3 00 \square$
$\square x 0 x 0 q_4 0 \square$
$\square x 0 x 0 x q_3 \square$
$\square x 0 x 0 q_5 x \square$
$\square x 0 x q_5 0 x \square$
:
$q_5 \square x 0 x 0 x \square$
$\square q_2 x 0 x 0 x \square$
:

An Example of Turing Machine: M_2 [S-ToC-13]

This machine begins by writing a blank symbol over the leftmost 0 on the tape so that it can find the left-hand end of the tape in stage 4. Whereas we would normally use a more suggestive symbol such as # for the left-hand end delimiter, we use a blank here to keep the tape alphabet, and hence the state diagram, small. Example 3.11 gives another method of finding the left-hand end of the tape.

Next we give a sample run of this machine on input 0000. The starting configuration is $q_1 0000$. The sequence of configurations the machine enters appears as follows; read down the columns and left to right.

$q_1 0000$	$\sqcup q_5 x 0 x \sqcup$	$\sqcup x q_5 x x \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup x 0 x \sqcup$	$\sqcup q_5 x x x \sqcup$
$\sqcup x q_3 00$	$\sqcup q_2 x 0 x \sqcup$	$q_5 \sqcup x x x \sqcup$
$\sqcup x 0 q_4 0$	$\sqcup x q_2 0 x \sqcup$	$\sqcup q_2 x x x \sqcup$
$\sqcup x 0 x q_3 \sqcup$	$\sqcup x x q_3 x \sqcup$	$\sqcup x q_2 x x \sqcup$
$\sqcup x 0 q_5 x \sqcup$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_2 x \sqcup$
$\sqcup x q_5 0 x \sqcup$	$\sqcup x x q_5 x \sqcup$	$\sqcup x x x q_2 \sqcup$
		$\sqcup x x x \sqcup q_{accept}$

An Example of Turing Machine: M_3 [S-ToC-13]

- Show the state transition diagram of the following M_3.

EXAMPLE 3.11

Here, a TM M_3 is doing some elementary arithmetic. It decides the language $C = \{a^i b^j c^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$.

M_3 = "On input string w:

1. Scan the input from left to right to determine whether it is a member of $a^+b^+c^+$ and *reject* if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, *reject*.
4. Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, *accept*; otherwise, *reject*."

An Example of Turing Machine: M_4 [S-ToC-13]

- **Show the state transition diagram of the following M_4.**

EXAMPLE 3.12

Here, a TM M_4 is solving what is called the *element distinctness problem*. It is given a list of strings over $\{0,1\}$ separated by #s and its job is to accept if all the strings are different. The language is

$$E = \{\#x_1\#x_2\#\cdots\#x_l \mid \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}.$$

Machine M_4 works by comparing x_1 with x_2 through x_l, then by comparing x_2 with x_3 through x_l, and so on. An informal description of the TM M_4 deciding this language follows.

M_4 = "On input w:

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, *accept*. If that symbol was a #, continue with the next stage. Otherwise, *reject*.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so *accept*.
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, *reject*.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so *accept*.
5. Go to stage 3."

Turing Machine Examples [L-ToC-17]

EXAMPLE 9.2

Consider the Turing machine defined by

$$Q = \{q_0, q_1\},$$
$$\Sigma = \{a, b\},$$
$$\Gamma = \{a, b, \Box\},$$
$$F = \{q_1\},$$

and

$$\delta(q_0, a) = (q_0, b, R),$$
$$\delta(q_0, b) = (q_0, b, R),$$
$$\delta(q_0, \Box) = (q_1, \Box, L).$$

As before, we can use transition graphs to represent Turing machines. Now we label the edges of the graph with three items: the current tape symbol, the symbol that replaces it, and the direction in which the read-write head is to move. The Turing machine in Example 9.2 is represented by the transition graph in Figure 9.4.

FIGURE 9.4

If this Turing machine is started in state q_0 with the symbol a under the read-write head, the applicable transition rule is $\delta(q_0, a) = (q_0, b, R)$. Therefore, the read-write head will replace the a with a b, then move right on the tape. The machine will remain in state q_0. Any subsequent a will also be replaced with a b, but b's will not be modified. When the machine encounters the first blank, it will move left one cell, then halt in final state q_1.

Figure 9.3 shows several stages of the process for a simple initial configuration.

FIGURE 9.3 A sequence of moves.

Turing Machine Examples [L-ToC-17]

EXAMPLE 9.3

Look at the Turing machine in Figure 9.5. To see what will happen, we can trace a typical case. Suppose that the tape initially contains $ab\ldots$, with the read-write head on the a. The machine then reads the a, but does not change it. Its next state is q_1 and the read-write head moves right, so that it is now over the b. This symbol is also read and left unchanged. The machine goes back into state q_0 and the read-write head moves left. We are now back exactly in the original state, and the sequence of moves starts again. It is clear from this that the machine, whatever the initial information on its tape, will run forever, with the read-write head moving alternately right then left, but making no modifications to the tape. This is an instance of a Turing machine that does not halt. In analogy with programming terminology, we say that the Turing machine is in an **infinite loop**.

FIGURE 9.5

Turing Machine Examples [L-ToC-17]

EXAMPLE 9.9

Given two positive integers x and y, design a Turing machine that computes $x + y$.

We first have to choose some convention for representing positive integers. For simplicity, we will use unary notation in which any positive integer x is represented by $w(x) \in \{1\}^+$, such that

$$|w(x)| = x.$$

We must also decide how x and y are placed on the tape initially and how their sum is to appear at the end of the computation. We will assume that $w(x)$ and $w(y)$ are on the tape in unary notation, separated by a single 0, with the read-write head on the leftmost symbol of $w(x)$. After the computation, $w(x+y)$ will be on the tape followed by a single 0, and the read-write head will be positioned at the left end of the result. We therefore want to design a Turing machine for performing the computation

$$q_0 w(x)\,0\,w(y) \overset{*}{\vdash} q_f w(x+y)\,0,$$

where q_f is a final state. Constructing a program for this is relatively simple. All we need to do is to move the separating 0 to the right end of $w(y)$, so that the addition amounts to nothing more than the coalescing of the two strings. To achieve this, we construct $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$, with $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, and

$$\delta(q_0, 1) = (q_0, 1, R),$$
$$\delta(q_0, 0) = (q_1, 1, R),$$
$$\delta(q_1, 1) = (q_1, 1, R),$$
$$\delta(q_1, \Box) = (q_2, \Box, L),$$
$$\delta(q_2, 1) = (q_3, 0, L),$$
$$\delta(q_3, 1) = (q_3, 1, L),$$
$$\delta(q_3, \Box) = (q_4, \Box, R).$$

Note that in moving the 0 right we temporarily create an extra 1, a fact that is remembered by putting the machine into state q_1. The transition $\delta(q_2, 1) = (q_3, 0, R)$ is needed to remove this at the end of the computation. This can be seen from the sequence of instantaneous descriptions for adding 111 to 11:

$$\begin{aligned} q_0 11\,0\,111 &\vdash 1q_0 1\,0\,111 \vdash 11q_0\,0\,111 \vdash 111q_1\,1\,11 \\ &\vdash 1111q_1\,1\,11 \vdash 11111q_1\,1 \vdash 111111q_1\,\Box \\ &\vdash 11111q_2 1 \vdash 1111q_3 10 \\ &\overset{*}{\vdash} q_3 \Box 11110 \vdash q_4 11110. \end{aligned}$$

Unary notation, although cumbersome for practical computations, is very convenient for programming Turing machines. The resulting programs are much shorter and simpler than if we had used another representation, such as binary or decimal.

Turing Machine Examples [L-ToC-17]

EXAMPLE 9.10

Design a Turing machine that copies strings of 1's. More precisely, find a machine that performs the computation

$$q_0 w \overset{*}{\vdash} q_f ww,$$

for any $w \in \{1\}^+$.

To solve the problem, we implement the following intuitive process:

1. Replace every 1 by an x.
2. Find the rightmost x and replace it with 1.
3. Travel to the right end of the current nonblank region and create a 1 there.
4. Repeat Steps 2 and 3 until there are no more x's.

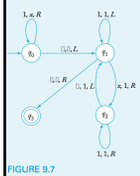


FIGURE 9.7

The solution is shown in the transition graph in Figure 9.7. It may be a little hard to see at first that the solution is correct, so let us trace the program with the simple string 11. The computation performed in this case is

$$\begin{aligned} q_0 11 &\vdash xq_0 1 \vdash xxq_0 \Box \vdash xq_1 x \\ &\vdash x1q_2 \Box \vdash xq_1 11 \vdash q_1 x11 \\ &\vdash 1q_2 11 \vdash 11q_2 1 \vdash 111q_2 \Box \\ &\vdash 11q_1 11 \vdash 1q_1 111 \\ &\vdash q_1 1111 \vdash q_1 \Box 1111 \vdash q_3 1111. \end{aligned}$$

Turing Machine Examples [L-ToC-17]

EXAMPLE 9.11

Let x and y be two positive integers represented in unary notation. Construct a Turing machine that will halt in a final state q_y if $x \geq y$, and that will halt in a nonfinal state q_n if $x < y$. More specifically, the machine is to perform the computation

$$q_0 w(x)\,0\,w(y) \overset{*}{\vdash} q_y w(x)\,0\,w(y) \qquad \text{if } x \geq y,$$
$$q_0 w(x)\,0\,w(y) \overset{*}{\vdash} q_n w(x)\,0\,w(y) \qquad \text{if } x < y.$$

To solve this problem, we can use the idea in Example 9.7 with some minor modifications. Instead of matching a's and b's, we match each 1 on the left of the dividing 0 with the 1 on the right. At the end of the matching, we will have on the tape either

$$xx\cdots 110xx\cdots x\Box$$

or

$$xx\cdots xx0xx\cdots x11\Box,$$

depending on whether $x > y$ or $y > x$. In the first case, when we attempt to match another 1, we encounter the blank at the right of the working space. This can be used as a signal to enter the state q_y. In the second case, we still find a 1 on the right when all 1's on the left have been replaced. We use this to get into the other state q_n. The complete program for this is straightforward and is left as an exercise.

This example makes the important point that a Turing machine can be programmed to make decisions based on arithmetic comparisons. This kind of simple decision is common in the machine language of computers, where alternate instruction streams are entered, depending on the outcome of an arithmetic operation.

Variants of TMs: Multi-tape Turing Machines

♣ Multi-tape Turing machine

- [D] A *multi-tape Turing machine* is like an ordinary TM with several tapes; each tape has its own read/write head.
- Initially the input appears on tape 1, and the others start out blank.
- The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously.
- [D] The transition function of a multi-tape TM can be defined as $\delta: (Q \times \Gamma^k) \rightarrow ((Q \times \Gamma^k) \times \{L, R\}^k)$ such that: $(q_i, a_1, a_2, \ldots, a_k) \rightarrow (q_j, b_1, b_2, \ldots, b_k, (L|R)_1, \ldots, (L|R)_k)$.

Variants of TMs: Multi-tape Turing Machines

♣ Multi-tape Turing machine

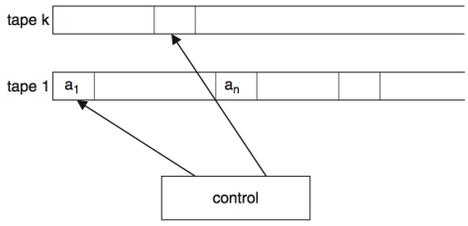


Fig. 2.2 Diagram of a k-tape Turing machine

Variants of TMs: Multi-tape Turing Machines [S-ToC-13]

♣ The equivalence of Multi-tape TM and single-tape TM

- Every multi-tape TM has an equivalent single-tape TM.
- For any multi-tape TM, there is a single-tape TM that recognizes the language of that multi-tape TM.

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

Variants of TMs: Non-deterministic Turing Machines

♣ Non-deterministic Turing machine

- [D] A *non-deterministic Turing machine* is like an ordinary TM but for any configuration in a computation, the next configuration may be any one of several possibilities.
- [D] The transition function of a nondeterministic TM can be defined as $\delta: (Q \times \Gamma) \rightarrow P((Q \times \Gamma) \times \{L, R\})$.
- The computation of a non-deterministic Turing machine is a tree whose branches correspond to different possibilities for the machine.
- If some branch of the computation leads to the accept state, the machine accepts its input.

Variants of TMs: Non-deterministic Turing Machines [S-ToC-13]

♣ The equivalence of non-deterministic TM and deterministic TM

- Every non-deterministic TM has an equivalent deterministic TM.
- For any non-deterministic TM, there is a deterministic TM that recognizes the language of that non-deterministic TM.

THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Variants of TMs: Non-deterministic Turing Machines [L-ToC-17]

♣ Non-deterministic Turing machine

DEFINITION 10.2

A nondeterministic Turing machine is an automaton as given by Definition 9.1, except that δ is now a function

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L,R\}}.$$

As always when nondeterminism is involved, the range of δ is a set of possible transitions, any of which can be chosen by the machine.

Variants of TMs: TMs with a Stay-option

♣ **Turing machines with a stay-option**

- Sometimes it is convenient to provide a third option, to have the read-write head stay in place after rewriting the cell content.
- [D] We can define a *Turing machine with a stay-option* by replacing δ in the definition of a standard TM by $\delta: (Q \times \Gamma) \rightarrow ((Q \times \Gamma) \times \{L, R, S\})$ such that $(q_i, a_j) \rightarrow (q_{i+1}, a_{j+1}, L \mid R \mid S)$.

♣ **The equivalence of TMs with a stay-option and standard TM**

- Every TM with a stay-option has an equivalent standard TM.

THEOREM 10.1

The class of Turing machines with a stay-option is equivalent to the class of standard Turing machines.

The Universal Turing Machine

♣ **The universal (reprogrammable) TM [Turing, 1936]**

- The *universal (reprogrammable) Turing machine* M_u is an automaton that, given as input the description of any TM M and a string w, can simulate the computation of M on w.

♣ **Q and Γ of M_u**

- We assume that $Q = \{q_1, q_2, ..., q_n\}$, with q_1 the initial state, q_2 the single final state, and $\Gamma = \{a_1, a_2, ..., a_m\}$, where a_1 represents the blank.
- We then select an encoding in which q_1 is represented by 1, q_2 is represented by 11, and so on. Similarly, a_1 is encoded as 1, a_2 as 11, etc. The symbol 0 will be used as a separator between the 1's.

The Universal Turing Machine

♣ **The transition function δ of M_u**

- With the initial and final state and the blank defined by this convention, any TM can be described completely with transition function δ only.
- The transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence.
- For example, $\delta(q_1, a_2) = (q_2, a_3, L)$ might appear as ⋯10110110111010⋯.
- It follows from this that any TM M has a finite encoding as a string on $\{0, 1\}^+$ and that, given any encoding of M, we can decode it uniquely.

The Universal Turing Machine

♣ **Structure of M_u**

- A universal Turing machine M_u then has an input alphabet that includes {0, 1} and the structure of a multi-tape TM.

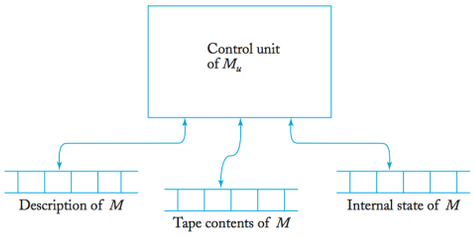


FIGURE 10.16

The Universal Turing Machine

♣ **Simulating the computation of M on w by M_u**

- For any input M and w, tape 1 will keep an encoded definition of M. Tape 2 will contain the tape contents of M, and tape 3 the internal state of M.
- M_u looks first at the contents of tapes 2 and 3 to determine the configuration of M.
- It then consults tape 1 to see what M would do in this configuration.
- Finally, tapes 2 and 3 will be modified to reflect the result of the move.

The Universal Turing Machine [W-ToC-12]

3.5. A Universal Turing Machine

From the enumeration of all Turing machines and the pairing function we can define a *universal* Turing machine U; that is, a machine that will emulate *every* other machine. Simply set

$$U(\langle e, x\rangle) = \varphi_e(x).$$

U decodes the single number it receives into a pair (e, x), decodes e into the appropriate set of quadruples, and uses x as input, acting according to the quadruples it decoded. This procedure is computable because decoding the pairing function, decoding Turing machines from indices, and executing quadruples on a given input are computable procedures. Turing [85] gives an explicit, full construction of a universal machine.

Note that of course there are infinitely many universal Turing machines, as there are for any program via padding, and that a universal machine exists for any collection of functions that may be indexed. Although we will use U and analogously defined universal machines for other indexings in this section, typically we simply refer to the individual indexed functions, using $\varphi_e(x)$ instead of $U(\langle e, x\rangle)$.

The Universal Turing Machine [HS-ToC-11]

Since every word is a number and vice versa, we may think of a Turing-machine code as a number. The code for a Turing machine M is called the *Gödel number* of M. If e is a Gödel number, then M_e is the Turing machine whose Gödel number is e.

Let U be a Turing machine that computes on input e and x and that implements the following algorithm:

if e is a code
  then simulate M_e on input x
  else output 0.

(Why are you convinced that a Turing machine with this behavior exists?) U is a *universal* Turing machine. To put it differently, U is a general-purpose, stored-program computer: U accepts as input two values: a "stored program" e, and "input to e," a word x. If e is the correct code of a program M_e, then U computes the value of M_e on input x.

Early computers had their programs hard-wired into them. Several years after Turing's 1936 paper, von Neumann and co-workers built the first computer that stored instructions internally in the same manner as data. Von Neumann knew Turing's work, and it is believed that von Neumann was influenced by Turing's universal machine. Turing's machine U is the first conceptual general-purpose, stored-program computer.

The Universal Turing Machine [F-ToC-09]

2.4.2 The universal Turing machine

It is possible to give a code for each Turing machine, so that from the code we can retrieve the machine. An easy way of doing this is as follows.

Assume the machine has n states $q_0, \ldots, q_{n-1}$, where each q_i is a number, q_0 is the initial state, and the last m states are final. Also assume that the input alphabet is $\{1\}$ and the tape alphabet is $\{0, 1\}$ (where 0 plays the role of a blank symbol). It is well known that a binary alphabet is sufficient to encode any kind of data, so there is no loss of generality in making this assumption. The transition function can be represented as a table, or equivalently a list of 5-tuples of the form (q, s, q', s', d), where q represents the current state, s the symbol on the tape under the head, q' the new state, s' the symbol written on the tape, and d the direction of movement, which we will write as $0, 1$. The order of the tuples is not important here. Thus, we can assume without loss of generality that the transition function is represented by a list l of tuples. The full description of the machine under these assumptions is given by the tuple (n, m, l), where l is the list representing the transition function, n the number of states, and m the number of final states, as indicated above. We will say that the tuple is the *code* for the machine since from it we can recover the original machine. In fact, the code for the machine is not unique since we can reorder the list l and still obtain an equivalent machine.

Now we can see the codes of Turing machines as words, and as such they can be used as input for a Turing machine. It is then possible to define a Turing machine U such that, when the code of a machine A is written on the tape, together with an input word w for A, U decodes it and simulates the behaviour of the machine A on w. The machine U is usually called the *universal Turing machine.*

The Universal Turing Machine [LP-ToC-98]

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine, and let i and j be the smallest integers such that $2^i \geq |K|$, and $2^j \geq |\Sigma| + 2$. Then each state in K will be represented as a q followed by a binary string of length i; each symbol in Σ will be likewise represented as the letter a followed by a string of j bits. The head directions $\leftarrow$ and $\rightarrow$ will also be treated as "honorary tape symbols" (they were the reason for the "+2" term in the definition of j). We fix the representations of the special symbols $\sqcup, \triangleright, \leftarrow$, and $\rightarrow$ to be the lexicographically four smallest symbols, respectively: $\sqcup$ will always be represented as $a0^j$, $\triangleright$ as $a0^{j-1}1$, $\leftarrow$ as $a0^{j-2}10$, and $\rightarrow$ as $a0^{j-2}11$. The start state will always be represented as the lexicographically first state, $q0^i$. Notice that we require the use of leading zeros in the strings that follow the symbols a and q, to bring the total length to the required level.

We shall denote the representation of the whole Turing machine M as "M". "M", consists of the transition table δ. That is, it is a sequence of strings of the form (q, a, p, b), with q and p representations of states and a, b of symbols, separated by commas and included in parentheses. We adopt the convention that the quadruples are listed in *increasing lexicographic order*, starting with $\delta(s, \sqcup)$. The set of halting states H will be determined indirectly, by the absence of its states as first components in any quadruple of "M". If M decides a language, and thus $H = \{y, n\}$, we will adopt the convention that y is the lexicographically smallest of the two halt states.

This way, any Turing machine can be represented. We shall use the same method to represent *strings* in the alphabet of the Turing machine. Any string $w \in \Sigma^*$ will have a unique representation, also denoted "w", namely, the juxtaposition of the representations of its symbols.

Turing Machines are Countable [L-ToC-17]

THEOREM 10.3

The set of all Turing machines, although infinite, is countable.

Proof: We can encode each Turing machine using 0 and 1. With this encoding, we then construct the following enumeration procedure.

1. Generate the next string in $\{0, 1\}^+$ in proper order.
2. Check the generated string to see if it defines a Turing machine. If so, write it on the tape in the form required by Definition 10.4. If not, ignore the string.
3. Return to Step 1.

Since every Turing machine has a finite description, any specific machine will eventually be generated by this process. ∎

Computation: What Is It and Why Study It?

- **Computation: What is it?**
  - **It is a process consisting of FINITE "computing" steps.**
  - **It is an ORDERED process according to previously specified procedure (algorithm).**
  - **Note: Operational viewpoint.**
- **Computation: Why study it?**
  - **In order to investigate the nature of computation.**
  - **In order to find those principles to design and implement "computers" to perform computation automatically.**
  - **In order to find limits of computation.**
  - **In order to find effectiveness and efficiency of computation.**

Uncomputable, Computable, and "Really" Computable Functions

- **Uncomputable functions**
  - **There is no effective method (finite step way) to compute such a function.**
  - **Note: Never halting TM**
- **Computable functions**
  - **There is an effective method (finite step way) to compute such a function.**
- **"Really" computable functions**
  - **There is a "really" effective method ("task-meaningfully" finite step way) to compute such a function.**
  - **What "really" and/or "task-meaningfully" means?**

A Comparison of Computation Time

n	Time (n)	$\log_2 n$	Time ($\log_2 n$)	n^2	Time (n^2)
2	0.002 sec	1	0.001 sec	4	0.004 sec
16	0.016 sec	4	0.004 sec	256	0.256 sec
64	0.064 sec	6	0.006 sec	4096	4.1 sec
256	0.256 sec	8	0.008 sec	65536	1 min 5 sec
1024	1 sec	10	0.010 sec	1048576	17 min 28 sec
4096	4.1 sec	12	0.012 sec	16777216	4 hours 40 min
16384	16.4 sec	14	0.014 sec	268435456	3 days 2 hours 34 min
65536	1 min 5 sec	16	0.016 sec	4294967296	49 days 17 hours
262144	4 min 22 sec	18	0.018 sec	68719476736	2 years 65 days
1000000	16 min 40 sec	20	0.020 sec	1000000000000	31 years 259 days
6	0.006 sec	3	0.003 sec	36	0.03 sec
30	0.03 sec	5	0.005 sec	900	0.9 sec
1000000000	11 days 14 hours	30	0.03 sec	10^{18}	33,000,000 years

A Comparison of Computation Time [J. Cheng]

我们来看一个有关“可计算”问题所需演算步骤的数量化比较。我们考虑A,B,C三个“可计算”问题，假设为了算出最终结果A问题需要n步演算、B问题需要n2（n的2次方）步演算、而C问题需要log2n（以2为底n的对数）步演算，并且还假设这些问题在某种现代通用计算机上被实施计算时每个演算步骤花费的时间为0.001秒，那么关于这三个问题之演算步骤及花费时间的一个数量化对比如下：

A	n=2	n=4	n=6	n=30	n=109
B	n2=4	n2=16	n2=36	n2=900	n2=1018
C	log2n=1	log2n=2	log2n=3	log2n=5	log2n=30
A	0.002秒	0.004秒	0.006秒	0.030秒	约11天14小时
B	0.004秒	0.016秒	0.036秒	0.9秒	约33,000,000年
C	0.001秒	0.002秒	0.003秒	0.005秒	0.030秒

由上面这个表我们可以看出，以30(36)步和0.03(0.36)秒为基准，三个问题之间的差距是巨大的。

An Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility (Turing-Reducibility)
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Computability: What Is It and Why Study It?

♣ The fundamental questions
- What a computation process can compute?
- What any computation process cannot compute?
- What computers can and cannot do?
- Is there some computational problems that cannot be computed by any computer?

♣ The theory of computability: What is it?
- It is the most theoretical foundation of CS.
- It is the theory of computability that classifies various problems into solvable one and those that are not.
- The theory of computability is one of the foundations of the theory of computational complexity; it introduces several concepts used in the latter.

Computability: What Is It and Why Study It?

♣ The theory of computability: Why study it?
- In order to know what computers can and cannot do in principles.
- We have to study the theory of computational complexity in order to know what computers can really (effectively and efficiently) do, but the theory of computability is one of the foundations of the theory of computational complexity.

Outcomes of Turing Machine

♣ Three possible outcomes of Turing machine
- When we start a TM on an input, three outcomes are possible: the TM may accept, reject, or “not halt”.

♣ Note
- It is often difficult to distinguish a TM that does not halt from one that is merely taking a very long time but will halt at some time.
- Therefore, we need a conceptual/theoretical way to distinguish the two cases.

图灵机只有三个最终态：接受、拒绝、永不停止

图灵可识别(递归可数)语言

最终能接受

Turing-recognizable (Recursively Enumerable) Languages

- **Turing-recognizable (recursively enumerable) languages**
  - **[D] A language is called *Turing-recognizable (recursively enumerable)* if some TM recognizes it, i.e., it may be the language of a TM.**

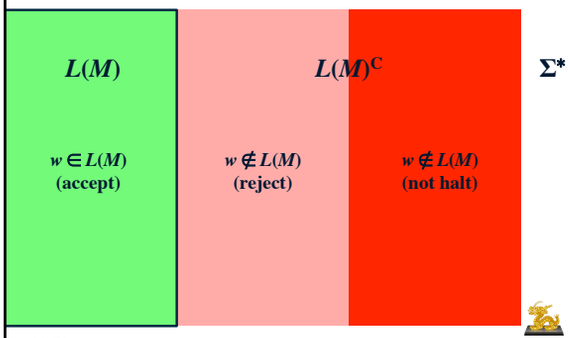
DEFINITION 11.1

A language L is said to be **recursively enumerable** if there exists a Turing machine that accepts it.

  - **[D] A language is called *Turing-unrecognizable* if there is no TM recognizes it.**
- **Notes**
  - **The "*Turing-recognizability*" of a language is an intrinsic property of that language.**
  - **For any $w \in \Sigma^*$, if $w \in L(M)$, M must halt and accept w; but if $w \notin L(M)$, M may halt and reject w or cannot halt.**

Turing-recognizable Language $L(M)$ over Σ

For L, there is a TM M, $L = L(M) \subset \Sigma^*$. For any $w \in \Sigma^*$, w may be:



Language of Turing Machine vs. Turing-recognizable Language

- **Language of a TM**
  - **The language of a TM is an intrinsic property of that TM.**
  - **For a given TM M, we say "$L(M)$ is M's language".**
- **Turing-recognizable language**
  - **The "Turing-recognizability" of a language is an intrinsic property of that language.**
  - **For a given language L, we consider whether there is a TM that accepts L or not:**
    if so, then we say "L is Turing-recognizable",
    if not, then we say "L is Turing-unrecognizable".

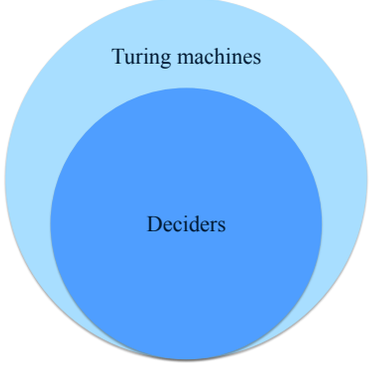
Turing Machines as Deciders

- **Definition: Turing machines as deciders**
  - **[D] A TM is called *a decider* if it halts on all inputs.**
  - **[D] A decider that recognizes some language also is said to *decide* that language.**
- **Notes**
  - **Every decider must be a TM, but some TMs are not deciders.**
  - **Whether a TM is a decider or not is an intrinsic property of that TM.**
  - **A decider decides its language.**

图灵机作为判定器

若图灵机对任何输入都能停止，则称为判定器

Turing Machines vs. Deciders



图灵可判定(递归)语言

Turing-decidable (Recursive) Languages

- **Turing-decidable (recursive) languages**
  - **[D] A language is called *Turing-decidable (recursive)* or simply *decidable* if some TM M decides it.**

DEFINITION 11.2

A language L on Σ is said to be **recursive** if there exists a Turing machine M that accepts L and that halts on every w in Σ^+. In other words, a language is recursive if and only if there exists a membership algorithm for it.

- **Notes**
  - **The "*Turing-decidability*" of a language is an intrinsic property of that language.**
  - **For a given language L, we consider whether there is a TM that decides L or not:**
    if so, then we say "L is Turing-decidable",
    if not, then we say "L is Turing-undecidable".

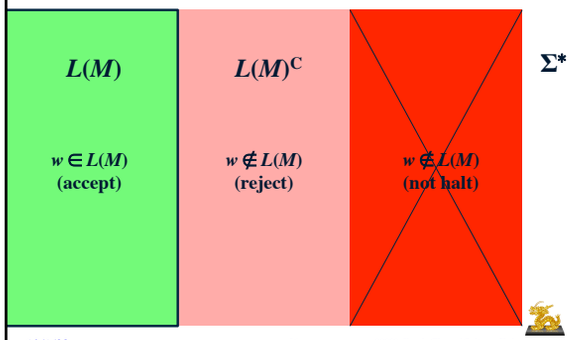
最终能停止

Turing-decidable Languages and Deciders [HS-ToC-11]

Definition 2.2. A language L is *Turing-machine-decidable* if L is accepted by some Turing machine that halts on every input, and a Turing machine that halts on every input and accepts L is called a *decider* for L.

Turing-decidable Language $L(M)$ over Σ

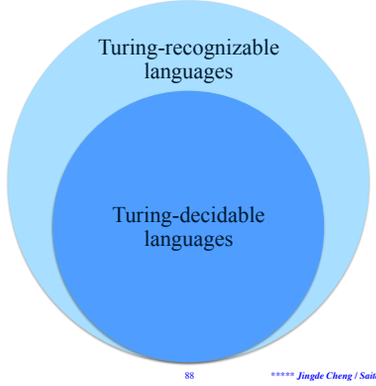
For L, there is a TM M, $L = L(M) \subset \Sigma^*$. For any $w \in \Sigma^*$, w may be:



Turing-recognizable Languages vs. Turing-decidable Languages

- **Turing-recognizable languages**
  - **A Turing-recognizable language may be Turing-decidable or may be not Turing-decidable.**
- **Turing-decidable languages**
  - **Every decider must be a TM, but some TMs are not deciders.**
  - **Every Turing-decidable language must be Turing-recognizable, but some Turing-recognizable languages are not Turing-decidable, i.e., they have no decider.**
- **Note**
  - **You have to understand the above concepts and distinguish them !**

Turing-recognizable Language vs. Turing-decidable Language



Turing-recognizable Languages and Turing-decidable Languages [S-ToC-13]

- **Turing-recognizable languages**

COROLLARY 3.15

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

COROLLARY 3.18

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

- **Turing-decidable languages**

COROLLARY 3.19

A language is decidable if and only if some nondeterministic Turing machine decides it.

Turing-recognizable Languages and Turing-decidable Languages [L-ToC-17]

- **Turing-recognizable (recursively numerable) languages and Turing-decidable (recursive) languages**

THEOREM 11.4

If a language L and its complement $\overline{L}$ are both recursively enumerable, then both languages are recursive. If L is recursive, then $\overline{L}$ is also recursive, and consequently both are recursively enumerable.

THEOREM 11.5

There exists a recursively enumerable language that is not recursive; that is, the family of recursive languages is a proper subset of the family of recursively enumerable languages.

L与$\overline{L}$都是递归可数的 ⇒ L与$\overline{L}$都是递归的

L是递归的⇒L是递归的，且L与$\overline{L}$都是递归可数的.

The Relationship among Classes of Languages [S-ToC-13]

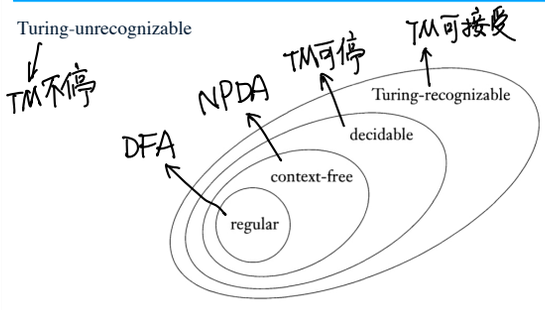


FIGURE **4.10**
The relationship among classes of languages

Decision Problems [HS-ToC-11]

- **Decision problems**
  - **[D] A *decision problem* is a general question to be answered, usually possessing several parameters, or free variables, whose values are left unspecified.**
  - **[D] An *instance* of a problem is obtained by specifying particular values for all of the problem parameters.**
  - **[D] A *solution* to a decision problem is an "*algorithm*" that answers the question that results from each instance.**
  - **[D] A decision problem is *decidable* if a solution exists and is *undecidable* otherwise.**
- **Note**
  - **A "problem" is not the same as a "question." A question refers to a specific instance.**

problem比较宽泛地定义了问题
question则具体提供了细节

Decision Problem Examples [HS-ToC-11]

- **The Hamiltonian Circuit problem**

Example 3.1. The Hamiltonian Circuit problem.

HAMILTONIAN CIRCUIT
instance A graph $G = (V,E)$.
question Does G contain a Hamiltonian circuit?

- **The Program Termination problem**

Example 3.2. The Program Termination problem for Turing machines

PROGRAM TERMINATION
instance A Turing machine M
question Does M eventually halt on every input?

The Notion of Algorithm

- **The notion of algorithm does not exist until the 20th century**
  - **The notion of "*algorithm*" was not defined precisely until the 20th century.**
- **Hilbert's 10th problem (23 problems in ICM, Paris, 1900)**
  - **The problem was to devise "a process according to which it can be determined by a FINITE number of operations" (i.e., an "*algorithm*") that tests whether a polynomial Diophantine equation with integer coefficients has an integral root. (it is undecidable, algorithmically unsolvable)**
  - **The solution of the problem had to wait for a clear definition of "*algorithm*".**
- **The Necessity of the notion of algorithm**
  - **Proving that an algorithm does not exist requires having a clear definition of algorithm.**

Hilbert's 10th Problem as a Decision Problem

- **Hilbert's 10th problem as a decision problem**
  - **Let $D = \{p \mid p$ is a polynomial with an integral root$\}$.**
  - **The problem asks in essence whether the set D is decidable. (The answer is negative)**
- ***D* is Turing-recognizable**
  - **Let $D_1 = \{p \mid p$ is a polynomial over x with an integral root$\}$ (i.e., p is a polynomial that has only a single variable).**

Here is a TM M_1 that recognizes D_1:

M_1 = "On input $\langle p \rangle$: where p is a polynomial over the variable x.
1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, 3, -3, \ldots$. If at any point the polynomial evaluates to 0, *accept*."

If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever. For the multivariable case, we can present a similar TM M that recognizes D. Here, M goes through all possible settings of its variables to integral values.

The Church-Turing Thesis (1936)

- **Church's Lambda calculus (λ-calculus) (1930s)** 丘奇λ演算
  - ***Lambda calculus* is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.**
  - **It is a universal model of computation equivalent to TM.**
- **The Church-Turing thesis (1936)** 丘奇-图灵命题

Intuitive notion of algorithms	equals	*Turing machine algorithms*

FIGURE **3.22**
The Church–Turing Thesis

- **The thesis claims: that a problem is *algorithmically solvable* (i.e., *has an algorithm*) is equivalent to that it is Turing-decidable.**

问题有算法⇔图灵可判定

λ演算能做的⇔图灵可判定的

The Question: What Is Computable?

♣ The Church-Turing thesis

A problem is computable
(algorithmically solvable, i.e., has an algorithm)

IFF

it is Turing-decidable

The Turing Thesis [L-ToC-17]

♣ The Turing thesis as a conjecture/hypothesis

Arguments of this type led A. M. Turing and others in the mid-1930s to the celebrated conjecture called the **Turing thesis**. This hypothesis states that any computation that can be carried out by mechanical means can be performed by some Turing machine.

This is a sweeping statement, so it is important to keep in mind what Turing's thesis is. It is not something that can be proved. To do so, we would have to define precisely the term "mechanical means." This would require some other abstract model and leave us no further ahead than before. The Turing thesis is more properly viewed as a definition of what constitutes a mechanical computation: A computation is mechanical if and only if it can be performed by some Turing machine.

The Turing Thesis [L-ToC-17]

♣ The Turing thesis as a definition of mechanical computation

If we take this attitude and regard the Turing thesis simply as a definition, we raise the question as to whether this definition is sufficiently broad. Is it far-reaching enough to cover everything we now do (and conceivably might do in the future) with computers? An unequivocal "yes" is not possible, but the evidence in its favor is very strong. Some arguments for accepting the Turing thesis as the definition of a mechanical computation are

1. Anything that can be done on any existing digital computer can also be done by a Turing machine.
2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.
3. Alternative models have been proposed for mechanical computation, but none of them is more powerful than the Turing machine model.

These arguments are circumstantial, and Turing's thesis cannot be proved by them. In spite of its plausibility, Turing's thesis is still an assumption. But viewing Turing's thesis simply as an arbitrary definition misses an important point. In some sense, Turing's thesis plays the same role in computer science as do the basic laws of physics and chemistry.

The Turing Thesis [L-ToC-17]

♣ Defining algorithm based on the Turing thesis

- **Having accepted Turing's thesis, we are in a position to give a precise definition of an algorithm.**

DEFINITION 9.5

An **algorithm** for a function $f : D \to R$ is a Turing machine M, which given as input any $d \in D$ on its tape, eventually halts with the correct answer $f(d) \in R$ on its tape. Specifically, we can require that

$$q_0 d \vdash_M^* q_f f(d), q_f \in F,$$

for all $d \in D$.

The Church-Turing Thesis [LP-ToC-98]

We therefore propose to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm." Nothing will be considered an algorithm if it cannot be rendered as a Turing machine that is guaranteed to halt on all inputs, and all such machines will be rightfully called algorithms. This principle is known as the **Church-Turing thesis**. It is a thesis, not a theorem, because it is not a mathematical result: It simply asserts that a certain informal concept (algorithm) corresponds to a certain mathematical object (Turing machine). Not being a mathematical statement, the Church-Turing thesis cannot be proved. It is theoretically possible, however, that the Church-Turing thesis could be *dis*proved at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as a plausible and reasonable model of computation, and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

From Turing Machines to Algorithms

♣ A turning point in the study of the theory of computation

- **We continue to speak of Turing machines, but our real focus from now on is on algorithms.**
- **The Turing machine merely serves as a precise model for the definition of algorithm.**
- **We skip over the extensive theory of Turing machines themselves and do not spend much time on the low-level programming of Turing machines. We need only to be comfortable enough with Turing machines to believe that they capture all algorithms.**

♣ A fundamental problem

- **What is the right level of detail to give when we describe algorithms?**

Describing Turing Machine Algorithm: Three Levels of Detail

- **Formal description (lowest)**
  - Spelling out in full the TM's states, transition function, and so on; this is the most detailed level of description.
- **Implementation description**
  - Using English prose to describe the way that the TM moves its head and the way that it stores data on its tape; at this level we do not give details of states or transition function.
- **High-level description (highest)**
  - Using English prose to describe an algorithm, ignoring the implementation details; at this level we do not need mention how the TM manages its tape or head.

Describing Turing Machine Algorithm: Encoding of Objects

- **Encoding of objects** 对象的编码
  - The standard input to a TM is always a string. Therefore, if we want to provide an object other than a string as input, we must first represent that object as a string.
  - Strings can represent polynomials, graphs, grammars, automata, and any combination of those objects.
  - A TM can be programmed to encode the representations.
- **Languages as problem representations**
  - The encoding of an object into its representation is a string.
  - Various computational problems can be represented by languages.

Describing Turing Machine Algorithm: Notation and Format

- **Notation for describing TMs**
  - The notation for the encoding of an object O into its representation as a string is $\langle O \rangle$.
  - If we have several objects $O_1, O_2, \ldots, O_k$, we denote their encoding into a single string $\langle O_1, O_2, \ldots, O_k \rangle$.
- **Format for describing TMs**
  - We describe TM algorithms with an indented segment of text within quotes.
  - We break the algorithm into stages, each usually involving many individual steps of the TM's computation.
  - We indicate the block structure of the algorithm with further indentation.

Describing Turing Machine Algorithm: Description

- **Description**
  - The first line of the algorithm describes the input to the TM.
  - If the input description is simply w, the input is taken to be a string.
  - If the input description is the encoding of an object as in $\langle A \rangle$, the TM first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't.

Describing Turing Machine Algorithm: An Example [S-ToC-13]

EXAMPLE 3.23

Let A be the language consisting of all strings representing undirected graphs that are connected. Recall that a graph is ***connected*** if every node can be reached from every other node by traveling along the edges of the graph. We write

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}.$$

The following is a high-level description of a TM M that decides A.

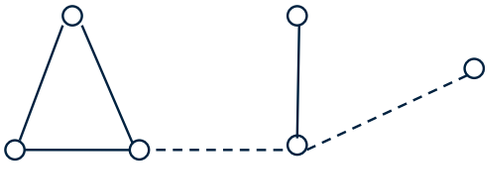
M = "On input $\langle G \rangle$, the encoding of a graph G:

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

Describing Turing Machine Algorithm: An Example

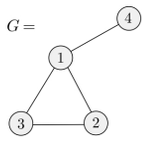
- **Undirected graph**
  - $G =_{df} (V(G), E(G))$, where $V(G)$ is a non-empty set of vertexes, and $E(G)$ is a finite set of edges represented as unordered-pairs.
- **Examples**
  - $V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$
  - $E(G) = \{\{v_1,v_2\},\{v_1,v_3\},\{v_2,v_3\},\{v_4,v_5\}\}$
  - $E(G') = \{\{v_1,v_2\},\{v_1,v_3\},\{v_2,v_3\},\{v_3,v_5\},\{v_4,v_5\},\{v_5,v_6\}\}$
  - $\langle G \rangle = \{1,2,3,4,5,6\}\{\{1,2\},\{1,3\},\{2,3\},\{4,5\}\}$
  - $\langle G' \rangle = \{1,2,3,4,5,6\}\{\{1,2\},\{1,3\},\{2,3\},\{3,5\},\{4,5\},\{4,6\}\}$

Describing Turing Machine Algorithm: An Example



Describing Turing Machine Algorithm: An Example [S-ToC-13]

For additional practice, let's examine some implementation-level details of Turing machine M. Usually we won't give this level of detail in the future and you won't need to either, unless specifically requested to do so in an exercise. First, we must understand how $\langle G \rangle$ encodes the graph G as a string. Consider an encoding that is a list of the nodes of G followed by a list of the edges of G. Each node is a decimal number, and each edge is the pair of decimal numbers that represent the nodes at the two endpoints of the edge. The following figure depicts such a graph and its encoding.

$G =$ 

$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

FIGURE 3.24
A graph G and its encoding $\langle G \rangle$

Describing Turing Machine Algorithm: An Example [S-ToC-13]

When M receives the input $\langle G \rangle$, it first checks to determine whether the input is the proper encoding of some graph. To do so, M scans the tape to be sure that there are two lists and that they are in the proper form. The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers. Then M checks several things. First, the node list should contain no repetitions; and second, every node appearing on the edge list should also appear on the node list. For the first, we can use the procedure given in Example 3.12 for TM M_4 that checks element distinctness. A similar method works for the second check. If the input passes these checks, it is the encoding of some graph G. This verification completes the input check, and M goes on to stage 1.

For stage 1, M marks the first node with a dot on the leftmost digit.

For stage 2, M scans the list of nodes to find an undotted node n_1 and flags it by marking it differently—say, by underlining the first symbol. Then M scans the list again to find a dotted node n_2 and underlines it, too.

Describing Turing Machine Algorithm: An Example [S-ToC-13]

Now M scans the list of edges. For each edge, M tests whether the two underlined nodes n_1 and n_2 are the ones appearing in that edge. If they are, M dots n_1, removes the underlines, and goes on from the beginning of stage 2. If they aren't, M checks the next edge on the list. If there are no more edges, $\{n_1, n_2\}$ is not an edge of G. Then M moves the underline on n_2 to the next dotted node and now calls this node n_2. It repeats the steps in this paragraph to check, as before, whether the new pair $\{n_1, n_2\}$ is an edge. If there are no more dotted nodes, n_1 is not attached to any dotted nodes. Then M sets the underlines so that n_1 is the next undotted node and n_2 is the first dotted node and repeats the steps in this paragraph. If there are no more undotted nodes, M has not been able to find any new nodes to dot, so it moves on to stage 4.

For stage 4, M scans the list of nodes to determine whether all are dotted. If they are, it enters the accept state; otherwise, it enters the reject state. This completes the description of TM M. ■

Decidable Language Examples: A_{DFA} [S-ToC-13]

♣ **The acceptance problem for DFAs**

- **Testing whether a particular DFA accepts a given string can be expressed as a language, A_{DFA}.**
- **This language contains the encodings of all DFAs together with strings that the DFAs accept.**
- **$A_{\text{DFA}} =_{\text{df}} \{ \langle B, w \rangle \mid B$ is a DFA that accepts input string $w \}$**
- **$\langle B, w \rangle \in A_{\text{DFA}} =$ "B is a DFA that accepts input string w".**

♣ **A_{DFA} is decidable**

THEOREM 4.1

A_{DFA} is a decidable language.

- **The proof needs to present a TM M that decides A_{DFA}.**

该语言包括了DFA与字符串的组合

A_{DFA}是可判定的

Decidable Language Examples: A_{DFA} [S-ToC-13]

♣ **A_{DFA} is decidable: A proof**

PROOF IDEA We simply need to present a TM M that decides A_{DFA}.

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

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w \rangle$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components: Q, Σ, δ, q_0, and F. When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects.

Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

Decidable Language Examples: A_{NFA} [S-ToC-13]

♣ **The acceptance problem for NFAs**

◆ $A_{NFA} =_{df} \{ \langle B, w \rangle \mid B$ **is a NFA that accepts input string** $w \}$

♣ A_{NFA} **is decidable**

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: Have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

N = "On input $\langle B, w \rangle$, where B is an NFA and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w \rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Running TM M in stage 2 means incorporating M into the design of N as a subprocedure.

A_{NFA}是可判定的

Decidable Language Examples: A_{REX} [S-ToC-13]

♣ **The generation problem for regular expressions**

◆ $A_{REX} =_{df} \{ \langle R, w \rangle \mid R$ **is regular expression that generates string** $w \}$

♣ A_{REX} **is decidable**

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

P = "On input $\langle R, w \rangle$, where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w \rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

A_{REX}是可判定的

Decidable Language Examples: E_{DFA} [S-ToC-13]

♣ **The emptiness testing problem for DFAs**

◆ ***Emptiness testing*** **for the language of a finite automaton: determine whether or not a FA accepts any strings at all.**

◆ $E_{DFA} =_{df} \{ \langle A \rangle \mid A$ **is a DFA and** $L(A) = \phi \}$

♣ E_{DFA} **is decidable**

THEOREM 4.4

E_{DFA} is a decidable language.

PROOF A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. To test this condition, we can design a TM T that uses a marking algorithm similar to that used in Example 3.23.

T = "On input $\langle A \rangle$, where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."

FA是不是没有语言.

E_{DFA}是可判定的

Decidable Language Examples: EQ_{DFA} [S-ToC-13]

♣ **The problem of determining whether two DFAs recognize the same language**

◆ $EQ_{DFA} =_{df} \{ \langle A, B \rangle \mid A$ **and** B **are DFAs and** $L(A) = L(B) \}$

♣ EQ_{DFA} **is decidable**

◆ EQ_{DFA} **is a decidable language.**

EQ_{DFA}是可判定的

Decidable Language Examples: EQ_{DFA} [S-ToC-13]

♣ EQ_{DFA} **is decidable**

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem, we use Theorem 4.4. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

This expression is sometimes called the ***symmetric difference*** of $L(A)$ and $L(B)$ and is illustrated in the following figure. Here, $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines. Once we have constructed C, we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

F = "On input $\langle A, B \rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C \rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

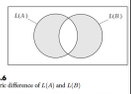


FIGURE 4.6
The symmetric difference of $L(A)$ and $L(B)$

EQ_{DFA}是可判定的

Decidable Language Examples: A_{CFG} [S-ToC-13]

♣ **The string generation problem for CFGs**

◆ **To determine whether a CFG generates a particular string and to determine whether the language of a CFG is empty.**

◆ $A_{CFG} =_{df} \{ \langle G, w \rangle \mid G$ **is a CFG that generates string** $w \}$

♣ A_{CFG} **is decidable**

◆ A_{CFG} **is a decidable language.**

♣ **Notes**

◆ **The problem of determining whether a CFG generates a particular string is related to the problem of compiling programming languages.**

◆ **The algorithm in TM** S **is very inefficient and would never be used in practice.**

上下文无关文法能生成字符串

A_{CFG}是可判定的

Decidable Language Examples: A_{CFG} [S-ToC-13]

♣ A_{CFG} is decidable

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w, we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.

To make this Turing machine into a decider, we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 157) we showed that if G were in Chomsky normal form, any derivation of w has $2n-1$ steps, where n is the length of w. In that case, checking only derivations with $2n-1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

S = "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

Decidable Language Examples: E_{CFG} [S-ToC-13]

♣ The emptiness testing problem for DFAs

- ***Emptiness testing*** for the language of a CFG: determine whether or not a CFG generates any strings at all.
- $E_{CFG} =_{df} \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \phi \}$

♣ E_{CFG} is decidable

- E_{CFG} is a decidable language.

E_{CFG}是可判定的

Decidable Language Examples: E_{CFG} [S-ToC-13]

♣ E_{CFG} is decidable

THEOREM 4.8

E_{CFG} is a decidable language.

PROOF IDEA To find an algorithm for this problem, we might attempt to use TM S from Theorem 4.7. It states that we can test whether a CFG generates some particular string w. To determine whether $L(G) = \emptyset$, the algorithm might try going through all possible w's, one by one. But there are infinitely many w's to try, so this method could end up running forever. We need to take a different approach.

In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines *for each variable* whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols, all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables. The TM R implements this algorithm.

PROOF

R = "On input $\langle G \rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1 U_2 \cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

The Decidability of Content-Free Languages (CFLs) [S-ToC-13]

♣ CFLs are decidable

上下文无关语言都是可判定的

THEOREM 4.9

Every context-free language is decidable.

PROOF IDEA Let A be a CFL. Our objective is to show that A is decidable. One (bad) idea is to convert a PDA for A directly into a TM. That isn't hard to do because simulating a stack with the TM's more versatile tape is easy. The PDA for A may be nondeterministic, but that seems okay because we can convert it into a nondeterministic TM and we know that any nondeterministic TM can be converted into an equivalent deterministic TM. Yet, there is a difficulty. Some branches of the PDA's computation may go on forever, reading and writing the stack without ever halting. The simulating TM then would also have some nonhalting branches in its computation, and so the TM would not be a decider. A different idea is necessary. Instead, we prove this theorem with the TM S that we designed in Theorem 4.7 to decide A_{CFG}.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

1. Run TM S on input $\langle G, w\rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

The Relationship among Classes of Languages [S-ToC-13]

Turing-unrecognizable

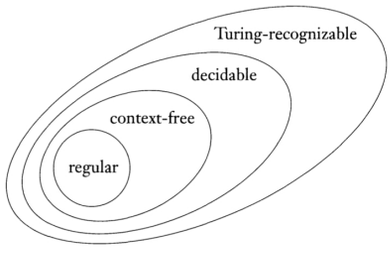
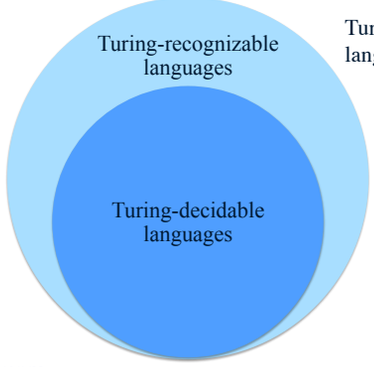


FIGURE 4.10
The relationship among classes of languages

Turing-decidable, Turing-recognizable, Turing-unrecognizable Languages



Turing-undecidable Languages

- **The most philosophically important fact in the theory of computation**
  - There are some computing problems (languages) that are not Turing-decidable, i.e., "algorithmically unsolvable".
- **Implications of the fundamental fact**
  - The computing power of TMs (and all computation models equivalent to TMs) is limited in a fundamental way.
  - The computing power of computers is limited in principle.
- **Note**
  - The fundamental fact is depend on the computation model.

The Pigeonhole Principle 鸽巢原理

- **The pigeonhole principle (Dirichlet drawer principle)**
  - If n pigeonholes are occupied by $n + 1$ or more pigeons, then at least one pigeonhole is occupied by more than one pigeon.
  - If n pigeonholes are occupied by $k*n + 1$ or more pigeons, then at least one pigeonhole is occupied by more than $k + 1$ pigeon.
- **Formal representation of the pigeonhole principle**
  - Let D and R be two finite sets, and $|D| > |R|$.
  - For any total function (mapping) f from D to R, there are $d_1, d_2 \in D$ such that $f(d_1) = f(d_2)$, i.e., there is no total injection (one-to-one).

Turing-unrecognizable Languages

- **Some languages are not recognized by any TM**
  - There are uncountably many languages yet only countably many TMs.
  - Because each TM can recognize a single language and there are more languages than TMs, some languages are not recognized by any TM.
  - Such languages are not Turing-recognizable.
- **Fundamental fact**

COROLLARY 4.18

Some languages are not Turing-recognizable.

Turing-unrecognizable Languages [S-ToC-13]

COROLLARY 4.18

Some languages are not Turing-recognizable.

PROOF To show that the set of all Turing machines is countable, we first observe that the set of all strings Σ^* is countable for any alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An *infinite binary sequence* is an unending sequence of 0s and 1s. Let $\mathcal{B}$ be the set of all infinite binary sequences. We can show that $\mathcal{B}$ is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that $\mathcal{R}$ is uncountable.

Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with $\mathcal{B}$, thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$. The ith bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the *characteristic sequence* of A. For example, if A were the language of all strings starting with a 0 over the alphabet $\{0,1\}$, its characteristic sequence χ_A would be

$$\begin{array}{lccccccccccc} \Sigma^* = \{ & \epsilon, & 0, & 1, & 00, & 01, & 10, & 11, & 000, & 001, & \cdots \}; \\ A = \{ & & 0, & & 00, & 01, & & & 000, & 001, & \cdots \}; \\ \chi_A = & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots \end{array}$$

The function $f: \mathcal{L} \longrightarrow \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto, and hence is a correspondence. Therefore, as $\mathcal{B}$ is uncountable, $\mathcal{L}$ is uncountable as well.

The Undecidable Language A_{TM} [S-ToC-13]

找不到TM能接受任意.w

A_{TM}是不可判定的

- **The undecidable language A_{TM}**
  - The problem of determining whether a TM accepts a given input string or not is undecidable.
  - $A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ accepts } w \}$.
  - A_{TM} is undecidable, i.e., there is no TM as a decider that can decide A_{TM}.

THEOREM 4.11

A_{TM} is undecidable.

- **The idea of proof**
  - Assume that A_{TM} is decidable at first and then obtain a contradiction.

The Recognizability of the Language A_{TM}

A_{TM}是可识别的

- **A_{TM} is Turing-recognizable**
  - The following TM U (an example of the universal Turing machine) [Turing, 1936] 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. "
  - Note U will loop on input $\langle M, w \rangle$ if M loops on w, which is why it does not decide A_{TM}.

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

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

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

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*."

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

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,

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

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

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

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.

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

Let's review the steps of this proof. Assume that a TM H decides A_{TM}. Use H to build a TM D that takes an input $\langle M\rangle$, where D accepts its input $\langle M\rangle$ exactly when M does not accept its input $\langle M\rangle$. Finally, run D on itself. Thus, the machines take the following actions, with the last line being the contradiction.

- H accepts $\langle M, w\rangle$ exactly when M accepts w.
- D rejects $\langle M\rangle$ exactly when M accepts $\langle M\rangle$.
- D rejects $\langle D\rangle$ exactly when D accepts $\langle D\rangle$.

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

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*			
$\vdots$			$\vdots$		

FIGURE 4.19
Entry i, j is *accept* if M_i accepts $\langle M_j\rangle$

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

In the following figure, the entries are the results of running H on inputs corresponding to Figure 4.19. So if M_3 does not accept input $\langle M_2\rangle$, the entry for row M_3 and column $\langle M_2\rangle$ is *reject* because H rejects input $\langle M_3, \langle M_2\rangle\rangle$.

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

FIGURE 4.20
Entry i, j is the value of H on input $\langle M_i, \langle M_j\rangle\rangle$

Proof of the Undecidability of the Language A_{TM} [S-ToC-13]

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*		*accept*	
M_3	*reject*	*reject*	*reject*	*reject*	$\cdots$	*reject*	$\cdots$
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

FIGURE 4.21
If D is in the figure, a contradiction occurs at "?"

The Halting Problem [L-ToC-17] 停机问题

DEFINITION 12.1

Let w_M be a string that describes a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, and let w be a string in M's alphabet. We will assume that w_M and w are encoded as a string of 0's and 1's, as suggested in Section 10.4. A solution of the halting problem is a Turing machine H, which for any w_M and w performs the computation

$$q_0 w_M w \overset{*}{\vdash} x_1 q_y x_2$$

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash} y_1 q_n y_2$$

if M applied to w does not halt. Here q_y and q_n are both final states of H.

THEOREM 12.1

There does not exist any Turing machine H that behaves as required by Definition 12.1. The halting problem is therefore undecidable.

Proof of the Undecidability of the Halting Problem [L-ToC-17]

THEOREM 12.1

There does not exist any Turing machine H that behaves as required by Definition 12.1. The halting problem is therefore undecidable.

Proof: We assume the contrary, namely, that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the string $w_M w$. The requirement is then that, given any $w_M w$, the Turing machine H will halt with either a yes or no answer. We achieve this by asking that H halt in one of two corresponding final states, say, q_y or q_n. The situation can be visualized by a block diagram like Figure 12.1. The intent of this diagram is to indicate that, if H is started in state q_0 with input $w_M w$, it will eventually halt in state q_y or q_n. As required by Definition 12.1, we want H to operate according to the following rules:

$$q_0 w_M w \overset{*}{\vdash}_H x_1 q_y x_2$$

FIGURE 12.1

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash}_H y_1 q_n y_2$$

if M applied to w does not halt.

Proof of the Undecidability of the Halting Problem [L-ToC-17]

Next, we modify H to produce a Turing machine H' with the structure shown in Figure 12.2. With the added states in Figure 12.2 we want to convey that the transitions between state q_y and the new states q_a and q_b are to be made, regardless of the tape symbol, in such a way that the tape remains unchanged. The way this is done is straightforward. Comparing H and H' we see that, in situations where H reaches q_y and halts, the modified machine H' will enter an infinite loop. Formally, the action of H' is described by

$$q_0 w_M w \overset{*}{\vdash}_{H'} \infty$$

if M applied to w halts, and

$$q_0 w_M w \overset{*}{\vdash}_{H'} y_1 q_n y_2$$

if M applied to w does not halt.

FIGURE 12.2

From H' we construct another Turing machine $\widehat{H}$. This new machine takes as input w_M and copies it, ending in its initial state q_0. After that, it behaves exactly like H'. Then the action of $\widehat{H}$ is such that

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} \infty$$

if M applied to w_M halts, and

$$q_0 w_M \overset{*}{\vdash}_{\widehat{H}} q_0 w_M w_M \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$$

if M applied to w_M does not halt.

Proof of the Undecidability of the Halting Problem [L-ToC-17]

Now $\widehat{H}$ is a Turing machine, so it has a description in $\{0,1\}^*$, say, $\widehat{w}$. This string, in addition to being the description of $\widehat{H}$, also can be used as input string. We can therefore legitimately ask what would happen if $\widehat{H}$ is applied to $\widehat{w}$. From the above, identifying M with $\widehat{H}$, we get

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} \infty$$

if $\widehat{H}$ applied to $\widehat{w}$ halts, and

$$q_0 \widehat{w} \overset{*}{\vdash}_{\widehat{H}} y_1 q_n y_2$$

if $\widehat{H}$ applied to $\widehat{w}$ does not halt. This is clearly nonsense. The contradiction tells us that our assumption of the existence of H, and hence the assumption of the decidability of the halting problem, must be false. ■

Proof of the Undecidability of the Halting Problem [L-ToC-17]

THEOREM 12.2

If the halting problem were decidable, then every recursively enumerable language would be recursive. Consequently, the halting problem is undecidable.

Proof: To see this, let L be a recursively enumerable language on Σ, and let M be a Turing machine that accepts L. Let H be the Turing machine that solves the halting problem. We construct from this the following procedure:

1. Apply H to $w_M w$. If H says "no," then by definition w is not in L.
2. If H says "yes," then apply M to w. But M must halt, so it will eventually tell us whether w is in L or not.

This constitutes a membership algorithm, making L recursive. But we already know that there are recursively enumerable languages that are not recursive. The contradiction implies that H cannot exist, that is, that the halting problem is undecidable. ■

The Undecidability of the Program Termination Problem [HS-ToC-11]

♣ **The program termination problem is undecidable**

Theorem 3.1. *The Program Termination problem (Example 3.2) is undecidable. There is no algorithm to determine whether an arbitrary partial computable function is total. Thus, there is no algorithm to determine whether a Turing machine halts on every input.*

♣ **Notes about proof**

- **First, we must encode TMs as words so that TMs can be presented as input strings to other TMs.**
- **Second, we construct a universal TM.**
- **Third, we represent the problem as a partial computational function.**
- **Finally, we use the diagonalization technique to show that no such algorithm exists.**

Encoding Turing Machines [HS-ToC-11]

of the details. Suppose we want to encode Turing machine

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, q_{\text{accept}} \rangle.$$

Assume that $Q = \{q_0, q_1, \ldots, q_{k-1}\}$. Represent the state q_i by $c(q_i) = [C_2(i)]$, and represent the set of states Q by the string $\{c(q_0), c(q_1), \ldots,\}$. Similarly, represent Σ and Γ as strings. Write a move $\delta(q_i, a_j) = (q_k, a_l, D)$, where $D = 1$, for a shift left, and $D = 2$, for a shift right, as a five-tuple (q_i, a_j, q_k, a_l, D). Then, of course, we represent the five-tuple by the string

$$(c(q_i), c(a_j), c(q_k), c(a_l), D).$$

Then δ is a sequence of such five-tuples, so we can represent the sequence as a string as well. Finally, represent the entire seven-tuple that defines M as a string.

Encoding Turing Machines [HS-ToC-11]

Observe that the current representation of a Turing machine M is a word w over the language $\{\mathbf{1}, \mathbf{2}, ,, \{, \}, [,], (,)\}$ consisting of nine symbols. We will make one more refinement. Identify this alphabet with the symbols $\{\mathbf{1}, \ldots, \mathbf{9}\}$. Then, for each word w over the alphabet $\{\mathbf{1}, \ldots, \mathbf{9}\}$, let $T(w) = C_2(N_9(w))$. (Recall that $N_9(w)$ is the natural number n whose 9-adic representation is w, and that $C_2(n)$ is the 2-adic representation of n.) $T(w)$ is the result of inputting w to an algorithm that converts 9-adic notation to 2-adic notation, and $T(w)$ is a word over the two-letter alphabet $\{\mathbf{1}, \mathbf{2}\}$. For a Turing machine M, where w is the representation of M as a word over the nine-letter alphabet, we call the word $T(w)$ the *encoding* of M.

For each Turing machine M, w_M will denote the word that encodes M. We will say that a problem about Turing machines is *decidable* if the set of words corresponding to Turing machines that satisfy the problem is decidable. Thus, the Program Termination decision problem, stated in Example 3.2, is decidable if and only if $\{w_M \mid M \text{ halts on every input}\}$ is a decidable set. Given M one can effectively find w_M; conversely, given a word w one can effectively determine whether $w = w_M$ for any M and if so, then effectively find M.

The Universal Turing Machine [HS-ToC-11]

Since every word is a number and vice versa, we may think of a Turing-machine code as a number. The code for a Turing machine M is called the *Gödel number* of M. If e is a Gödel number, then M_e is the Turing machine whose Gödel number is e.

Let U be a Turing machine that computes on input e and x and that implements the following algorithm:

if e is a code
  then simulate M_e on input x
  else output 0.

(Why are you convinced that a Turing machine with this behavior exists?) U is a *universal* Turing machine. To put it differently, U is a general-purpose, stored-program computer: U accepts as input two values: a "stored program" e, and "input to e," a word x. If e is the correct code of a program M_e, then U computes the value of M_e on input x.

Early computers had their programs hard-wired into them. Several years after Turing's 1936 paper, von Neumann and co-workers built the first computer that stored instructions internally in the same manner as data. Von Neumann knew Turing's work, and it is believed that von Neumann was influenced by Turing's universal machine. Turing's machine U is the first conceptual general-purpose, stored-program computer.

Representing the problem as a Partial Computational Function [HS-ToC-11]

For every natural number e, define

$$\phi_e = \lambda x.\, U(e, x).^1$$

If e is a Gödel number, then ϕ_e is the partial function of one argument that is computed by M_e. If e is not the code of any Turing machine, then by the definition of U, $\phi_e(x) = 0$ for all x.

Let's use the Program Termination problem to illustrate all of this new notation: The Program Termination problem is decidable if and only if

$$\{w_M \mid M \text{ halts on every input}\} = \{e \mid e \text{ is a Gödel number and } \phi_e \text{ halts on every input}\}$$

is decidable, which holds if and only if

$$\{e \mid \phi_e \text{ is total computable}\} = \{e \mid L(M_e) = \Sigma^*\}$$

is decidable. Note that there is an algorithm to determine whether a natural number e is a Gödel number; in case it is not, then, by definition, ϕ_e is total.

Observe that every partial computable function of one argument is ϕ_e for some e, and conversely, every ϕ_e is a partial computable function. Thus, $\{\phi_e\}_{e \geq 0}$ is an *effective enumeration* of the set of all partial computable functions.

Proof of the Undecidability of the Program Termination Problem [HS-ToC-11]

Proof. Suppose there is an algorithm TEST such that, for every i,

TEST(i) = "yes" if ϕ_i halts on every input value, and

TEST(i) = "no" otherwise.

Define a function δ by

$\delta(k) = \phi_k(k) + 1$ if TEST(k) = "yes" and

$\delta(k) = 0$ if TEST(k) = "no."

By definition, δ is defined on every input and δ is computable, so δ is a total computable function. Let e be the Gödel number of a Turing machine that computes δ. Thus, $\delta = \phi_e$ and TEST(e) = "yes." However, in this case, $\delta(e) = \phi_e(e) + 1$, which contradicts the assertion that $\delta = \phi_e$. Thus, the initial supposition must be false. □

Some Undecidable Problems [LP-ToC-98]

Theorem 5.5.1: *Each of the following problems is undecidable.*

(a) For a given grammar G and string w, to determine whether $w \in L(G)$.
(b) For a given grammar G, to determine whether $e \in L(G)$.
(c) For two given grammars G_1 and G_2, to determine whether $L(G_1) = L(G_2)$.
(d) For an arbitrary grammar G, to determine whether $L(G) = \emptyset$.
(e) Furthermore, there is a certain fixed grammar G_0, such that it is undecidable to determine whether any given string w is in $L(G_0)$.

Theorem 5.5.2: *Each of the following problems is undecidable.*

(a) Given a context-free grammar G, is $L(G) = \Sigma^*$?
(b) Given two context-free grammars G_1 and G_2, is $L(G_1) = L(G_2)$?
(c) Given two pushdown automata M_1 and M_2, do they accept precisely the same language?
(d) Given a pushdown automaton M, find an equivalent pushdown automaton with as few states as possible.

The Complement of a Language 语言的补

♣ The complement of a language

- For a language L over Σ, the *complement* of L is the language (denotes L^c) over Σ consisting of all strings that are not in L.
- $\Sigma^* = L \cup L^c$, $L \cap L^c = \varnothing$

♣ Co-Turing-recognizable language 协-图灵可识别语言

- We say that a language is *co-Turing-recognizable* if it is the complement of a Turing-recognizable language.

The Turing-decidability of a Language

♣ The sufficient and necessary conditions for the Turing-decidability of a language

- A language L is decidable IFF it is Turing-recognizable and co-Turing-recognizable.
- A language L is decidable IFF both it and its complement L^c are Turing-recognizable.

等价说法

♣ Note

- There are two different TMs M and M^c such that M recognizes L and M^c recognizes L^c.

The Turing-decidability of a Language [S-ToC-13]

THEOREM 4.22

A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

In other words, a language is decidable exactly when both it and its complement are Turing-recognizable.

PROOF We have two directions to prove. First, if A is decidable, we can easily see that both A and its complement $\overline{A}$ are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For the other direction, if both A and $\overline{A}$ are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for $\overline{A}$. The following Turing machine M is a decider for A.

M = "On input w:

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, *accept*; if M_2 accepts, *reject*."

Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2. In this case, M takes turns simulating one step of each machine, which continues until one of them accepts.

Now we show that M decides A. Every string w is either in A or $\overline{A}$. Therefore, either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A. So M is a decider for A, and thus A is decidable.

A Turing-unrecognizable Language [S-ToC-13]

♣ A Turing-unrecognizable language

- The complement of language A_{TM} is not Turing-recognizable, because A_{TM} is undecidable but Turing-recognizable.

COROLLARY 4.23

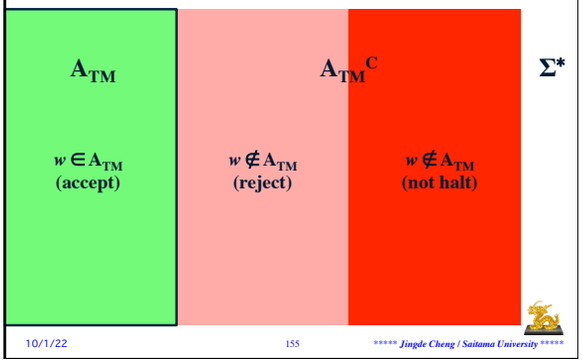
$\overline{A_{TM}}$ is not Turing-recognizable.

A_{TM}是图灵不可识别的

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{TM}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{TM}}$ must not be Turing-recognizable.

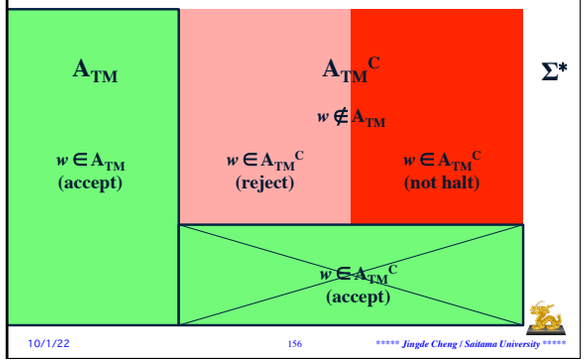
Turing-unrecognizable Language A_{TM}^C over Σ

There is a TM U such that $L(U) = A_{TM} \subset \Sigma^*$. For any $w \in \Sigma^*$, w may be:



Turing-unrecognizable Language A_{TM}^C over Σ

There is a TM U such that $L(U) = A_{TM} \subset \Sigma^*$. For any $w \in \Sigma^*$, w may be:



Not Recursively Enumerable (Turing-unrecognizable) Languages [L-ToC-17]

THEOREM 11.1

Let S be an infinite countable set. Then its powerset 2^S is not countable.

THEOREM 11.2

For any nonempty Σ, there exist languages that are not recursively enumerable.

THEOREM 11.3

There exists a recursively enumerable language whose complement is not recursively enumerable.

An Introduction to the Theory of Computation

- Enumerability and Diagonalization
- Finite Automata and Regular Languages
- Pushdown Automata and Context-Free Languages
- Computation: Turing Machines
- Computation: Turing-Computability (Turing-Decidability)
- Computation: Reducibility (Turing-Reducibility)
- Computation: Recursive Functions
- Computation: Recursive Sets and Relations
- Equivalent Definitions of Computability
- Advanced Topics in Computability Theory
- Computational Complexity
- Time Complexity
- Space Complexity
- Intractability
- Advanced Topics in Complexity Theory

Reducibility

♣ **Reduction**

- **[D] A *reduction* is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.**

♣ **Reducibility**

- **[D] *Reducibility* always involves two problems, which we call A and B. If A reduces to B, we can use a solution to B to solve A.**

♣ **Note**

- **Reducibility says nothing about solving A or B alone, but only about the solvability of A in the presence of a solution to B.**

(Turing-)Reducibility 图灵可归约

♣ **(Turing-)Reducibility**

- **(Turing-)Reducibility plays an important role in classifying problems by Turing-decidability in the computability theory, and in the computational complexity theory as well.**
- **When A is reducible to B, (algorithmically) solving A cannot be harder than solving B because a (algorithmic) solution to B gives a solution to A.**
- **In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable.**
- **Equivalently, if A is undecidable and reducible to B, B also is undecidable.**
- **This last version is key to proving that various problems are undecidable.**

Proving the Undecidability of the Halting Problem by Reduction [S-ToC-13]

♣ **The halting problem**

- **$HALT_{TM}$ = { $\langle M, w\rangle$ | M is a TM and M halts on input w }**

♣ **The Undecidability of the halting problem**

- **$HALT_{TM}$ is undecidable.**

♣ **Proof idea**

- **The proof is by contradiction.**
- **We assume that $HALT_{TM}$ is decidable and use that assumption to show that A_{TM} is decidable, but this is contradictory to the fact of the undecidability of A_{TM}.**
- **The key is to show that A_{TM} is reducible to $HALT_{TM}$.**

THEOREM 5.1

$HALT_{TM}$ is undecidable.

Proving the Undecidability of the Halting Problem by Reduction [S-ToC-13]

Let's assume that we have a TM R that decides $HALT_{TM}$. Then we use R to construct S, a TM that decides A_{TM}. To get a feel for the way to construct S, pretend that you are S. Your task is to decide A_{TM}. You are given an input of the form $\langle M, w\rangle$. You must output *accept* if M accepts w, and you must output *reject* if M loops or rejects on w. Try simulating M on w. If it accepts or rejects, do the same. But you may not be able to determine whether M is looping, and in that case your simulation will not terminate. That's bad because you are a decider and thus never permitted to loop. So this idea by itself does not work.

Instead, use the assumption that you have TM R that decides $HALT_{TM}$. 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, $HALT_{TM}$ is undecidable.

Proving the Undecidability of the Halting Problem by Reduction [S-ToC-13]

PROOF Let's assume for the purpose of obtaining a contradiction that TM R decides $HALT_{TM}$. 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 $HALT_{TM}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

Undecidable Problems from Language Theory [S-ToC-13]

♣ **The empty language testing problem**

- $E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \phi \}$

THEOREM 5.2

E_{TM} is undecidable.

♣ **The regular language testing problem**

- $REGULAR_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular language} \}$

THEOREM 5.3

$REGULAR_{TM}$ is undecidable.

♣ **The equivalence testing problem**

- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

THEOREM 5.4

EQ_{TM} is undecidable.

Reductions by Computation Histories

♣ **The computation history method**

- **The computation history method is an important technique for proving that A_{TM} is reducible to certain languages.**
- **This method is often useful when the problem to be shown undecidable involves testing for the existence of something.**

♣ **The computation history for a TM**

- **The computation history for a TM on an input is simply the sequence of configurations that the machine goes through as it processes the input.**
- **It is a complete record of the computation of this machine.**

The computation history for a TM: Formal Definition [S-ToC-13]

♣ **The computation history for a TM**

DEFINITION 5.5

Let M be a Turing machine and w an input string. An ***accepting computation history*** for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A ***rejecting computation history*** for M on w is defined similarly, except that C_l is a rejecting configuration.

♣ **Notes**

- **Computation histories are finite sequences.**
- **If M does not halt on w, no accepting or rejecting computation history exists for M on w.**

Linear Bounded Automata (LBAs) [S-ToC-13]

♣ **Linear bounded automata (LBAs)**

DEFINITION 5.6

A ***linear bounded automaton*** is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is—in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.

♣ **Notes**

- **A *linear bounded automaton (LBA)* is a TM with a limited amount of memory. It can only solve problems requiring memory that can fit within the tape used for the input.**
- **Using a tape alphabet larger than the input alphabet allows the available memory to be increased up to a constant factor. For an input of length n, the amount of memory available is linear in n.**

Linear Bounded Automata (LBAs) [S-ToC-13]

♣ **The power of LBAs**

- **Despite their memory constraint, LBAs are quite powerful.**
- **The deciders for A_{DFA}, A_{CFG}, E_{DFA}, and E_{CFG} all are LBAs.**
- **Every CFL can be decided by an LBA.**

♣ **A_{LBA}: The acceptance problem of determining whether an LBA accepts its input**

- **$A_{LBA} = \{ \langle M, w \rangle \mid M \text{ is an LBA that accepts input } w \}$**
- **Even though A_{LBA} is the same as the undecidable problem A_{TM} where the TM is restricted to be an LBA, we can show that A_{LBA} is decidable.**
- **The difference between the decidability of A_{LBA} and the undecidability of A_{TM} shows an intrinsic difference between LBAs and TMs.**

The Decidability of A_{LBA} [S-ToC-13]

♣ **A limited number of configurations of a LBA**

- **An LBA can have only a limited number of configurations when a string of length n is the input.**

LEMMA 5.8

Let M be an LBA with q states and g symbols in the tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.

♣ **A_{LBA} is decidable**

- **$A_{\text{LBA}} = \{ \langle M, w \rangle \mid M$ is an LBA that accepts input $w \}$**
- **A_{LBA} is decidable.**

THEOREM 5.9

A_{LBA} is decidable.

Proof of the Decidability of A_{LBA} [S-ToC-13]

PROOF IDEA In order to decide whether LBA M accepts input w, we simulate M on w. During the course of the simulation, if M halts and accepts or rejects, we accept or reject accordingly. The difficulty occurs if M loops on w. We need to be able to detect looping so that we can halt and reject.

The idea for detecting when M is looping is that as M computes on w, it goes from configuration to configuration. If M ever repeats a configuration, it would go on to repeat this configuration over and over again and thus be in a loop. Because M is an LBA, the amount of tape available to it is limited. By Lemma 5.8, M can be in only a limited number of configurations on this amount of tape. Therefore, only a limited amount of time is available to M before it will enter some configuration that it has previously entered. Detecting that M is looping is possible by simulating M for the number of steps given by Lemma 5.8. If M has not halted by then, it must be looping.

PROOF The algorithm that decides A_{LBA} is as follows.

$L =$ "On input $\langle M, w \rangle$, where M is an LBA and w is a string:

1. Simulate M on w for qng^n steps or until it halts.
2. If M has halted, *accept* if it has accepted and *reject* if it has rejected. If it has not halted, *reject*."

If M on w has not halted within qng^n steps, it must be repeating a configuration according to Lemma 5.8 and therefore looping. That is why our algorithm rejects in this instance.

The Undecidability of E_{LBA} [S-ToC-13]

♣ **E_{LBA} is undecidable**

- **$E_{\text{LBA}} = \{ \langle M \rangle \mid M$ is an LBA and $L(M) = \phi \}$**
- **E_{LBA} is undecidable.**

THEOREM 5.10

E_{LBA} is undecidable.

♣ **Proof idea**

PROOF IDEA This proof is by reduction from A_{TM}. We show that if E_{LBA} were decidable, A_{TM} would also be. Suppose that E_{LBA} is decidable. How can we use this supposition to decide A_{TM}?

For a TM M and an input w, we can determine whether M accepts w by constructing a certain LBA B and then testing whether $L(B)$ is empty. The language that B recognizes comprises all accepting computation histories for M on w. If M accepts w, this language contains one string and so is nonempty. If M does not accept w, this language is empty. If we can determine whether B's language is empty, clearly we can determine whether M accepts w.

The Undecidability of E_{LBA} [S-ToC-13]

Now we describe how to construct B from M and w. Note that we need to show more than the mere existence of B. We have to show how a Turing machine can obtain a description of B, given descriptions of M and w.

As in the previous reductions we've given for proving undecidability, we construct B only to feed its description into the presumed E_{LBA} decider, but not to run B on some input.

We construct B to accept its input x if x is an accepting computation history for M on w. Recall that an accepting computation history is the sequence of configurations, $C_1, C_2, \ldots, C_l$ that M goes through as it accepts some string w. For the purposes of this proof, we assume that the accepting computation history is presented as a single string with the configurations separated from each other by the # symbol, as shown in Figure 5.11.

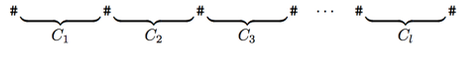


FIGURE 5.11
A possible input to B

The Undecidability of E_{LBA} [S-ToC-13]

The LBA B works as follows. When it receives an input x, B is supposed to accept if x is an accepting computation history for M on w. First, B breaks up x according to the delimiters into strings $C_1, C_2, \ldots, C_l$. Then B determines whether the C_i's satisfy the three conditions of an accepting computation history.

1. C_1 is the start configuration for M on w.
2. Each C_{i+1} legally follows from C_i.
3. C_l is an accepting configuration for M.

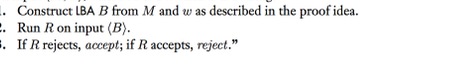


FIGURE 5.11
A possible input to B

The start configuration C_1 for M on w is the string $q_0 w_1 w_2 \cdots w_n$, where q_0 is the start state for M on w. Here, B has this string directly built in, so it is able to check the first condition. An accepting configuration is one that contains the q_{accept} state, so B can check the third condition by scanning C_l for q_{accept}. The second condition is the hardest to check. For each pair of adjacent configurations, B checks on whether C_{i+1} legally follows from C_i. This step involves verifying that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i. These positions must be updated according to the transition function of M. Then B verifies that the updating was done properly by zig-zagging between corresponding positions of C_i and C_{i+1}. To keep track of the current positions while zig-zagging, B marks the current position with dots on the tape. Finally, if conditions 1, 2, and 3 are satisfied, B accepts its input.

By inverting the decider's answer, we obtain the answer to whether M accepts w. Thus we can decide A_{TM}, a contradiction.

The Undecidability of E_{LBA} [S-ToC-13]

PROOF Now we are ready to state the reduction of A_{TM} to E_{LBA}. Suppose that TM R decides E_{LBA}. Construct TM S to decide A_{TM} as follows.

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

1. Construct LBA B from M and w as described in the proof idea.
2. Run R on input $\langle B \rangle$.
3. If R rejects, *accept*; if R accepts, *reject*."

If R accepts $\langle B \rangle$, then $L(B) = \emptyset$. Thus, M has no accepting computation history on w and M doesn't accept w. Consequently, S rejects $\langle M, w \rangle$. Similarly, if R rejects $\langle B \rangle$, the language of B is nonempty. The only string that B can accept is an accepting computation history for M on w. Thus, M must accept w. Consequently, S accepts $\langle M, w \rangle$. Figure 5.12 illustrates LBA B.

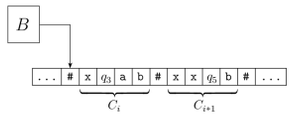


FIGURE 5.12
LBA B checking a TM computation history

The Undecidability of ALL_{CFG} [S-ToC-13]

♣ **The problem of determining whether a CFG generates all possible strings**

- ◆ $ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(M) = \Sigma^* \}$

♣ **ALL_{CFG} is undecidable**

- ◆ **ALL_{CFG} is undecidable.**

THEOREM 5.13

ALL_{CFG} is undecidable.

The Post Correspondence Problem (PCP) [S-ToC-13]

We can describe this problem easily as a type of puzzle. We begin with a collection of dominos, each containing two strings, one on each side. An individual domino looks like

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]$$

and a collection of dominos looks like

$$\left\{\left[\frac{\texttt{b}}{\texttt{ca}}\right], \left[\frac{\texttt{a}}{\texttt{ab}}\right], \left[\frac{\texttt{ca}}{\texttt{a}}\right], \left[\frac{\texttt{abc}}{\texttt{c}}\right]\right\}$$

The task is to make a list of these dominos (repetitions permitted) so that the string we get by reading off the symbols on the top is the same as the string of symbols on the bottom. This list is called a ***match***. For example, the following list is a match for this puzzle.

$$\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{b}}{\texttt{ca}}\right]\left[\frac{\texttt{ca}}{\texttt{a}}\right]\left[\frac{\texttt{a}}{\texttt{ab}}\right]\left[\frac{\texttt{abc}}{\texttt{c}}\right].$$

Reading off the top string we get `abcaaabc`, which is the same as reading off the bottom. We can also depict this match by deforming the dominos so that the corresponding symbols from top and bottom line up.

The Undecidability of PCP [S-ToC-13]

♣ **The Post Correspondence Problem (PCP)**

- ◆ **To determine whether a collection of dominos has a match.**
- ◆ **An instance of the PCP is a collection *P* of dominos**

$$P = \left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\},$$

and a match is a sequence $t_1, t_2, \ldots, t_l$, where $t_{i1}t_{i2}\ldots t_{il} = b_{i1}b_{i2}\ldots b_{il}$.

- ◆ **The problem is to determine whether *P* has a match.**

♣ **The Undecidability of PCP**

- ◆ ***PCP* = { <*P*> | *P* is an instance of the PCP with a match }**
- ◆ ***PCP* is undecidable.**

THEOREM 5.15

PCP is undecidable.

To Reduce the Halting Problem to the State-entry Problem [L-ToC-17]

EXAMPLE 12.1

The **state-entry problem** is as follows. Given any Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$ and any $q \in Q$, $w \in \Sigma^+$, decide whether or not the state q is ever entered when M is applied to w. This problem is undecidable.

To reduce the halting problem to the state-entry problem, suppose that we have an algorithm A that solves the state-entry problem. We could then use it to solve the halting problem. For example, given any M and w, we first modify M to get $\widehat{M}$ in such a way that $\widehat{M}$ halts in state q if and only if M halts. We can do this simply by looking at the transition function δ of M. If M halts, it does so because some $\delta(q_i, a)$ is undefined. To get $\widehat{M}$, we change every such undefined δ to

$$\delta(q_i, a) = (q, a, R),$$

where q is a final state. We apply the state-entry algorithm A to $(\widehat{M}, q, w)$. If A answers yes, that is, the state q is entered, then (M, w) halts. If A says no, then (M, w) does not halt.

Thus, the assumption that the state-entry problem is decidable gives us an algorithm for the halting problem. Because the halting problem is undecidable, the state-entry problem must also be undecidable.

To Reduce the Halting Problem to the Blank-tape Halting Problem [L-ToC-17]

EXAMPLE 12.2

The **blank-tape halting problem** is another problem to which the halting problem can be reduced. Given a Turing machine M, determine whether or not M halts if started with a blank tape. This isundecidable.

To show how this reduction is accomplished, assume that we are given some M and some w. We first construct from M a new machine M_w that starts with a blank tape, writes w on it, then positions itself in a configuration q_0w. After that, M_w acts like M. Clearly M_w will halt on a blank tape if and only if M halts on w.

Suppose now that the blank-tape halting problem were decidable. Given any (M, w), we first construct M_w, then apply the blank-tape halting problem algorithm to it. The conclusion tells us whether M applied to w will halt. Since this can be done for any M and w, an algorithm for the blank-tape halting problem can be converted into an algorithm for the halting problem. Since the latter is known to be undecidable, the same must be true for the blank-tape halting problem.

A Common Approach in Establishing Undecidability Results [L-ToC-17]

♣ **A common approach to establish undecidability results**

- ◆ **The construction in Example 12.2 is summarized in a block diagram in Figure 12.3.**
- ◆ **We first use an algorithm that transforms (M,w) into Mw; such an algorithm clearly exists.**
- ◆ **Next, we use the algorithm for solving the blank-tape halting problem, which we assume exists.**
- ◆ **Putting the two together yields an algorithm for the halting problem.**
- ◆ **But this is impossible, and we can conclude that A cannot exist.**

A Common Approach in Establishing Undecidability Results [L-ToC-17]

♣ **A common approach to establish undecidability results**

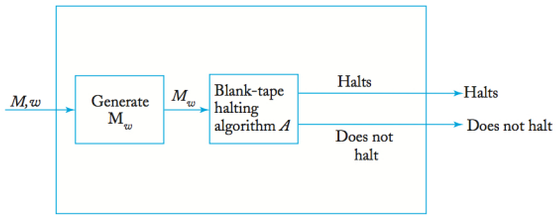


FIGURE 12.3 Algorithm for the halting problem.

Reducing an Undecidable Problem to Computing a Function

♣ **A decision problem as a function**

- **A decision problem is effectively a function with a range {0,1}, that is, a true or false answer.**

♣ **Reducing an undecidable problem to computing a function**

- **We can reduce the halting problem (or any other problem known to be undecidable) to the problem of computing the function in question.**
- **Because of Turing's thesis, we expect that functions encountered in practical circumstances will be computable, so for examples of uncomputable functions we must look a little further.**
- **Most examples of uncomputable functions are associated with attempts to predict the behavior of Turing machines.**

Reducing an Undecidable Problem to Computing a Function [L-ToC-17]

EXAMPLE 12.3

Let $\Gamma = \{0, 1, \square\}$. Consider the function $f(n)$ whose value is the maximum number of moves that can be made by any n-state Turing machine that halts when started with a blank tape. This function, as it turns out, is not computable.

Before we set out to demonstrate this, let us make sure that $f(n)$ is defined for all n. Notice first that there are only a finite number of Turing machines with n states. This is because Q and Γ are finite, so δ has a finite domain and range. This in turn implies that there are only a finite number of different δ's and therefore a finite number of different n-state Turing machines.

Of all of the n-state machines, there are some that always halt, for example machines that have only final states and therefore make no moves. Some of the n-state machines will not halt when started with a blank tape, but they do not enter the definition of f. Every machine that does halt will execute a certain number of moves; of these, we take the largest to give $f(n)$.

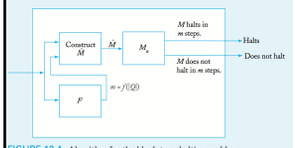


FIGURE 12.4 Algorithm for the blank-tape halting problem.

Take any Turing machine M and positive number m. It is easy to modify M to produce $\widetilde{M}$ in such a way that the latter will always halt with one of two answers: M applied to a blank tape halts in no more than m moves, or M applied to a blank tape makes more than m moves. All we have to do for this is to have M count its moves and terminate when this count exceeds m. Assume now that $f(n)$ is computable by some Turing machine F. We can then put $\widetilde{M}$ and F together as shown in Figure 12.4. First we compute $f(|Q|)$, where Q is the state set of M. This tells us the maximum number of moves that M can make if it is to halt. The value we get is then used as m to construct $\widetilde{M}$ as outlined, and a description of $\widetilde{M}$ is given to a universal Turing machine for execution. This tells us whether M applied to a blank tape halts or does not halt in less than $f(|Q|)$ steps. If we find that M applied to a blank tape makes more than $f(|Q|)$ moves, then because of the definition of f, the implication is that M never halts. Thus we have a solution to the blank-tape halting problem. The impossibility of the conclusion forces us to accept that f is not computable.

Undecidable Problems for Recursively Enumerable Languages [L-ToC-17]

THEOREM 12.3

Let G be an unrestricted grammar. Then the problem of determining whether or not

$$L(G) = \varnothing$$

is undecidable.

THEOREM 12.4

Let M be any Turing machine. Then the question of whether or not $L(M)$ is finite is undecidable.

The Post Correspondence Problem (PCP) [L-ToC-17]

The Post correspondence problem can be stated as follows. Given two sequences of n strings on some alphabet Σ, say

$$A = w_1, w_2, ...w_n$$

and

$$B = v_1, v_2, ..., v_n,$$

we say that there exists a Post correspondence solution (PC-solution) for pair (A, B) if there is a nonempty sequence of integers $i, j, ..., k$, such that

$$w_i w_j \cdots w_k = v_i v_j \cdots v_k.$$

The Post correspondence problem is to devise an algorithm that will tell us, for any (A, B), whether or not there exists a PC solution.

THEOREM 12.7

The Post correspondence problem is undecidable.

The Post Correspondence Problem (PCP): An Example [L-ToC-17]

EXAMPLE 12.5

Let $\Sigma = \{0, 1\}$ and take A and B as

$$w_1 = 11, w_2 = 100, w_3 = 111,$$
$$v_1 = 111, v_2 = 001, v_3 = 11.$$

For this case, there exists a PC solution as Figure 12.7 shows.

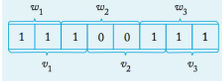


FIGURE 12.7

If we take

$$w_1 = 00, w_2 = 001, w_3 = 1000,$$
$$v_1 = 0, v_2 = 11, v_3 = 011,$$

there cannot be any PC solution simply because any string composed of elements of A will be longer than the corresponding string from B.

Mapping (Many-One) Reducibility

♣ **Mapping (Many-one) reducibility**

- The notion of reducing one problem to another may be defined formally in one of several ways. The choice of which one to use depends on the application.
- A simple type of reducibility is ***mapping reducibility***.
- Being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B.
- If we have such a conversion function, called a reduction, we can solve A with a solver for B.
- The reason is that any instance of A can be solved by first using the reduction to convert it to an instance of B and then applying the solver for B.

Computable Functions [S-ToC-13]

♣ **Computable functions**

- A TM computes a function by starting with the input to the function on the tape and halting with the output of the function on the tape.

DEFINITION 5.17

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

♣ **Computable function examples**

- All usual, arithmetic operations on integers are computable functions.
- Computable functions may be transformations of machine descriptions.

Mapping (Many-One) Reducibility: Formal Definition [S-ToC-13]

DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f: \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

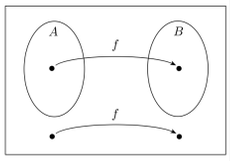


FIGURE 5.21
Function f reducing A to B

Mapping (Many-One) Reducibility: Its Roles

♣ **Converting questions**

- A mapping reduction of A to B provides a way to convert questions about membership testing in A to membership testing in B.
- To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.
- The term ***mapping reduction*** comes from the function or mapping that provides the means of doing the reduction.

♣ **Obtaining solutions**

- If one problem is mapping reducible to a second, previously solved problem, we can thereby obtain a solution to the original problem.

Mapping (Many-One) Reducibility: Applications [S-ToC-13]

- The corollary 5.23 is a main tool for proving undecidability.

THEOREM 5.22

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

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired.

COROLLARY 5.23

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

Mapping (Many-One) Reducibility: Applications [S-ToC-13]

THEOREM 5.28

If $A \leq_{\mathrm{m}} B$ and B is Turing-recognizable, then A is Turing-recognizable.

COROLLARY 5.29

If $A \leq_{\mathrm{m}} B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

Turing Reducibility

♣ **Why Turing reducibility?**

- **Does mapping reducibility capture our intuitive concept of reducibility in the most general way? It does not.**
- **For example, consider the two languages A_{TM} and its complement A_{TM}^C.**
- **Intuitively, they are reducible to one another because a solution to either could be used to solve the other by simply reversing the answer.**
- **However, we know that A_{TM}^C is not mapping reducible to A_{TM} because A_{TM} is Turing-recognizable but A_{TM}^C is not.**
- **A very general form of reducibility, called *Turing reducibility*, captures our intuitive concept of reducibility more closely.**

Oracle Turing Machines [S-ToC-13]

♣ **Oracle Turing machines**

DEFINITION 6.18

An ***oracle*** for a language B is an external device that is capable of reporting whether any string w is a member of B. An ***oracle Turing machine*** is a modified Turing machine that has the additional capability of querying an oracle. We write M^B to describe an oracle Turing machine that has an oracle for language B.

♣ **Note**

- **We are not concerned with the way the oracle determines its responses.**
- **We use the term oracle to connote a magical ability and consider oracles for languages that are not decidable by ordinary algorithms.**

The Relationship of "Decidable Relative to" [S-ToC-13]

EXAMPLE 6.19

Consider an oracle for A_{TM}. An oracle Turing machine with an oracle for A_{TM} can decide more languages than an ordinary Turing machine can. Such a machine can (obviously) decide A_{TM} itself, by querying the oracle about the input. It can also decide E_{TM}, the emptiness testing problem for TMs with the following procedure called $T^{A_{TM}}$.

$T^{A_{TM}}$ = "On input $\langle M \rangle$, where M is a TM:

1. Construct the following TM N.
   N = "On any input:
   1. Run M in parallel on all strings in Σ^*.
   2. If M accepts any of these strings, *accept*."
2. Query the oracle to determine whether $\langle N, 0 \rangle \in A_{TM}$.
3. If the oracle answers NO, *accept*; if YES, *reject*."

If M's language isn't empty, N will accept every input and, in particular, input 0. Hence the oracle will answer YES, and $T^{A_{TM}}$ will reject. Conversely, if M's language is empty, $T^{A_{TM}}$ will accept. Thus $T^{A_{TM}}$ decides E_{TM}. We say that E_{TM} is ***decidable relative to*** A_{TM}. That brings us to the definition of Turing reducibility. ■

Turing Reducibility: Definition [S-ToC-13]

♣ **The definition of Turing reducibility**

DEFINITION 6.20

Language A is ***Turing reducible*** to language B, written $A \leq_T B$, if A is decidable relative to B.

♣ **The Turing reducibility and the mapping reducibility**

- **Turing reducibility is a generalization of mapping reducibility.**
- **If $A \leq_m B$, then $A \leq_T B$ because the mapping reduction may be used to give an oracle Turing machine that decides A relative to B.**

Turing Reducibility: Application [S-ToC-13]

♣ **The Application of Turing reducibility**

THEOREM 6.21

If $A \leq_T B$ and B is decidable, then A is decidable.

PROOF If B is decidable, then we may replace the oracle for B by an actual procedure that decides B. Thus, we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A.

Introduction to the Theory of Computation

- **Enumerability and Diagonalization**
- **Finite Automata and Regular Languages**
- **Pushdown Automata and Context-Free Languages**
- **Computation: Turing Machines**
- **Computation: Turing-Computability (Turing-Decidability)**
- **Computation: Reducibility (Turing-Reducibility)**
- **Computation: Recursive Functions**
- **Computation: Recursive Sets and Relations**
- **Equivalent Definitions of Computability**
- **Advanced Topics in Computability Theory**
- **Computational Complexity**
- **Time Complexity**
- **Space Complexity**
- **Intractability**
- **Advanced Topics in Complexity Theory**