An Introduction to Computer Science

Jingde Cheng

Southern University of Science and Technology

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)

The Question: What Is Computation/Computing ?

What is computation or computing ?

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 of "computation" 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)*.
- Logic is the father/mother of Computer Science
  - It is the "decision problem" that leads to the notion and/or concept of "computation", and motivates researches on computability.

Computation: What Is It?

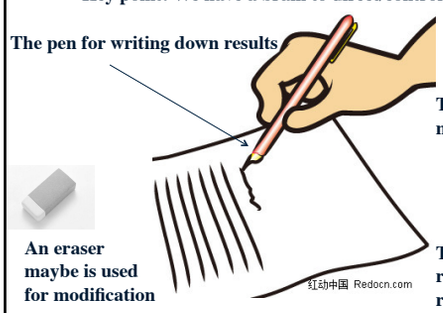
- The state-of-the-art of computation
  - Early equivalent models of computation: *Recursive functions* (Herbrand, Godel, Kleene, 1931-1936), *Lambda calculus* (Church, Kleene, 1933-1935), ***Turing machine*** **(Turing, 1936-1937),** *Post systems* (Post, 1936-1943).
  - Some *Turing-complete* and/or *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?

The Question: How Do We Human Compute ?

How do we human compute ?

How Do We Human Compute?

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

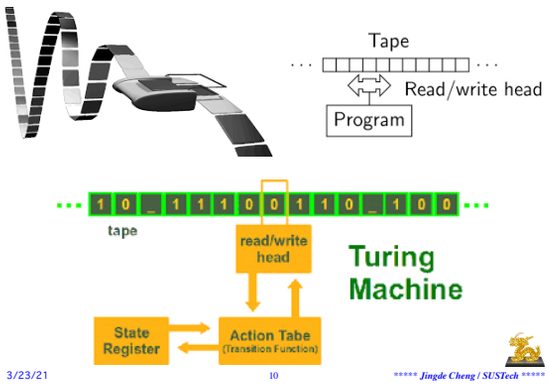


Other ?

The Question: How to Simulate Human Computing ?

How to use a abstract machine (Turing machine) to simulate human computing ?

Turing Machine: A Theoretical Computation Model



Turing Machine: The Origins

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 Machine: A Theoretical Computation Model

- **Tape** 纸带
  - The TM model uses *an infinite tape* 无限长纸带 that is divided into *infinite cells*, each of which can contain any one of *a finite set of symbols* 有限字符集, as its unlimited memory.
  - The type can be infinite at one end or infinite at both ends.
- **Read/Write head** 读写头
  - 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
  - The TM model has *a control unit* that controls actions performed by the read/write head.

Turing Machine: A Theoretical Computation Model

Figure 12.2 The components of a Turing machine

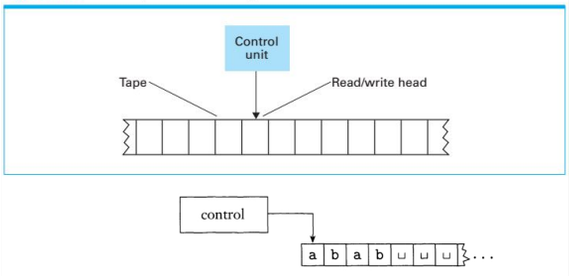


FIGURE **3.1**
Schematic of a Turing machine

Turing Machine: A Theoretical Computation Model

♣ **States of a TM**
- The position of the read/write head and the contents of the tape of a TM consists a *state* of that TM.

♣ **Initial state**
- 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**
- A TM has two specified states as *outputs*.
- 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.

Turing Machine: A Theoretical Computation Model

♣ **Computation process**
- 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.
- 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.

Alphabets, Strings, and Languages

♣ **Alphabet**
- An *alphabet* is a non-empty finite set of symbols.
- We generally use capital Greek letters Σ and Γ to designate alphabets.

♣ **Examples of alphabet**
- $\Sigma_1 = \{0, 1\}$
- $\Sigma_2 = \{a, b, c, \ldots, z\}$
- $\Gamma = \{0, 1, x, y, z\}$

Alphabets, Strings, and Languages

♣ **String**
- A *string* over an alphabet is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.
- For a given alphabet Σ, Σ^* denotes all strings over Σ (The reflexive transitive closure of symbol connection relation).

♣ **The length of a string**
- If w is string, the length of w, written $|w|$, is the number of its symbols.
- Note: A symbol may appear in a strings many times.

♣ **The empty string**
- The string of length zero is called *the empty string* and is written ε.

Alphabets, Strings, and Languages

♣ **The reverse of string**
- For string $w = w_1w_2w_3\ldots w_n$, *the reverse* of w, written w^R, is the string obtained by writing w in the opposite order, i.e., $w^R = w_n\ldots w_3w_2w_1$.

♣ **The concatenation of strings**
- For string x of length m and string y of length n, *the concatenation* of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1x_2x_3\ldots x_my_1y_2y_3\ldots y_n$.
- To concatenate a string with itself many time, say k times, we use the superscript notation x^k.

Alphabets, Strings, and Languages

♣ **Prefix of string**

- For string y, we say that string x is *a prefix* of y if a string z exists where $xz = y$, and that x is *a proper prefix* of y if in addition $x \neq y$.
- Note: Any string is a prefix of itself.

♣ **Language**

- For a given alphabet Σ, *A language over Σ*, denoted by L_Σ, is a set of strings over Σ.
- For any Σ, $L_\Sigma \subseteq \Sigma^*$.
- A language is *prefix-free* if no member is a proper prefix of another member.

Turing Machine: 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 Machine: Formal Definition

♣ **State set, Input alphabet, and Tape alphabet**

- *State set* Q, *Input alphabet* Σ, and *Tape alphabet* Γ all are finite sets.
- $\Sigma \subset \Gamma$, $\sqcup \notin \Sigma$, $\sqcup \in \Gamma$.

♣ **Transition function**

- *Transition function* $\delta = (Q \times \Gamma) \rightarrow ((Q \times \Gamma) \times \{\text{L}, \text{R}\})$ is a function from the Cartesian (direct) product of Q and Γ to the Cartesian (direct) product of $(Q \times \Gamma)$ and {L, R}.
- $(q_i, a_j) \rightarrow (q_{i+1}, a_{j+1}, \text{L|R})$ 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 Machine: Formal Definition [L-ToC-17]

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.

Turing Machine: Formal Definition

♣ **Question**

- How to represent a transition function δ by a table?

♣ **Hint example**

- Is the following table sufficient to representing a transition function?

A table representation of transition function

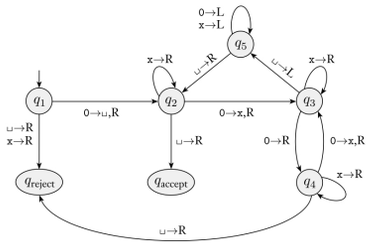
	0	1	x	
q_1	1, q_2			
q_2		q_{accept}		
q_3			q_{reject}	

Turing Machine: Formal Definition

♣ **Question**

- How to represent a transition function δ by a directed graph?

♣ **Hint example**



Turing Machine Example: 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 zig-zag to the corresponding places
  - 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.

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

- Let M_1 work in the following way

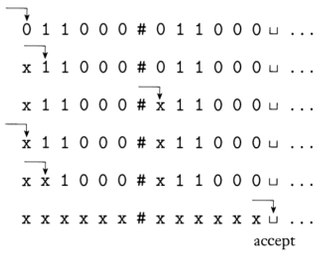


FIGURE **3.2**
Snapshots of Turing machine M_1 computing on input 011000#011000

Turing Machine Example: 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*."

Turing Machine Example: A Variant of M_1 (CS101A)

- 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.
  - Show a state transition table/diagram to representing the transition function of M_1'.

Turing Machine Example: 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.

Turing Machine Example: 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
- $\Gamma = \{0, \text{x}, \sqcup\}$.
- 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}.

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

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

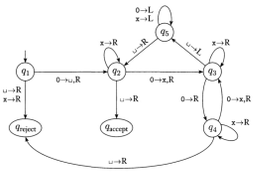


FIGURE **3.8**
State diagram for Turing machine M_2

In this state diagram, the label 0→␣,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 ␣, and moves the head to the right. In other words, $\delta(q_1, 0) = (q_2, ␣, R)$. For clarity we use the shorthand 0→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 $\delta(q_3, 0) = (q_4, 0, R)$.

Turing Machine: 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.

Computation of Turing Machine: 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.**

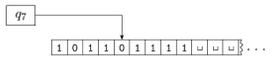


FIGURE **3.4**
A Turing machine with configuration $1011q_701111$

Computation of Turing Machine: 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 $\delta(q_i, b) = (q_j, c, \text{L})$, and
  $ua\ q_i\ bv$ yields $uac\ q_j\ v$, if $\delta(q_i, b) = (q_j, c, \text{R})$.

♣ **Note**

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

Computation of Turing Machine: 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 $\delta(q_i, b) = (q_j, c, \text{L})$
  (because we prevent the machine from going off the left-hand end of the tape),
  $q_i\ bv$ yields $c\ q_j\ v$, if $\delta(q_i, b) = (q_j, c, \text{R})$.

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

- **For the special case of right-hand end,**
  "$ua\ q_i$" is equivalent to "$ua\ q_i$ □"
  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 Machine: 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.**

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

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

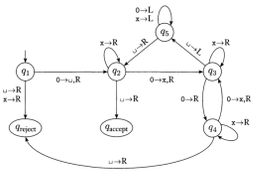


FIGURE **3.8**
State diagram for Turing machine M_2

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 $\delta(q_3, 0) = (q_4, 0, R)$.

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

- **Acceptable strings (n = 0, 1, 2, 3, …)**
  - **0 ($2^0 = 1$), 00 ($2^1 = 2$), 0000 ($2^2 = 4$), 00000000 ($2^3 = 8$), …**
- **Input: 0 (n = 0)**

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

- **Input: 00 (n = 1)**

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

- **Input: 000**

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

Turing Machine Example: 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$0000	␣q_5x0x␣	␣x$q_5$xx␣
␣$q_2$000	q_5␣x0x␣	␣$q_5$xxx␣
␣x$q_3$00	␣q_2x0x␣	q_5␣xxx␣
␣x0$q_4$0	␣x$q_2$0x␣	␣$q_2$xxx␣
␣x0xq_3␣	␣xxq_3x␣	␣x$q_2$xx␣
␣x0q_5x␣	␣xxxq_3␣	␣xxq_2x␣
␣x$q_5$0x␣	␣xxq_5x␣	␣xxxq_2␣
		␣xxx␣q_{accept}

Turing Machine Example: 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 $\Gamma = \{0,1,\#,\text{x},\sqcup\}$.
- 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.

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

- **Homework: Show a table representation of the state transition diagram of M_1.**

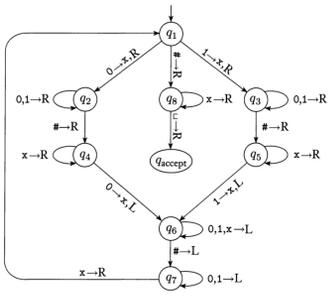


FIGURE **3.10**
State diagram for Turing machine M_1

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

- **Note: The reject state and the transitions going to the reject state are not shown in the Figure 3.10. The transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol.**
- **Homework: Make up all transitions that have been omitted.**

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.

Turing Machine Example: M_3 [S-ToC-13] (CS101A only)

- **Homework for CS101A: 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*."

Turing Machine Example: M_4 [S-ToC-13] (CS101A only)

- **Homework for CS101A: 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."

Variants of Turing Machines

♣ **Multi-tape Turing Machine**

- **A *multi-tape Turing machine* is like an ordinary TM with several tapes; each tape has its own read/write head.**
- **The transition function of a multi-tape TM can be defined as**
  $\delta =_{df} (Q \times \Gamma^k) \rightarrow ((Q \times \Gamma^k) \times \{L, R\}^k)$ such that:
  $(q_i, a_1, a_2, \ldots, a_k) \rightarrow$
  $(q_j, b_1, b_2, \ldots, b_k, (L|R)_1, \ldots, (L|R)_k)$.

♣ **Fact about multi-tape Turing machines**

- **Every multi-tape TM has an equivalent single-tape TM.**

Variants of Turing Machines

♣ **Nondeterministic Turing Machine**

- **A *nondeterministic Turing machine* is like an ordinary TM but for any configuration in a computation, the next configuration may be any one of several possibilities.**
- **The transition function of a nondeterministic TM can be defined as**
  $\delta =_{df} (Q \times \Gamma) \rightarrow P((Q \times \Gamma) \times \{L, R\})$.

♣ **Fact about nondeterministic Turing machines**

- **Every nondeterministic TM has an equivalent deterministic TM.**

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.**
- **Question: 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 [Cheng, 2015]

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。

Large Numbers

These numbers are order-of-magnitude estimates, and have been culled from a variety of sources. Many of the astrophysics numbers are explained in Pournan

TABLE 1.1
Large Numbers

Physical Analogue	Number
Odds of being killed by lightning (per day)	1 in 9 billion (2^{33})
Odds of winning the top prize in a U.S. state lottery	1 in 4,000,000 (2^{22})
Odds of winning the top prize in a U.S. state lottery and being killed by lightning in the same day	1 in 2^{55}
Odds of drowning (in the U.S. per year)	1 in 59,000 (2^{16})
Odds of being killed in an automobile accident(in the U.S. in 1993)	1 in 6100 (2^{13})
Odds of being killed in an automobile accident(in the U.S. per lifetime)	1 in 88 (2^7)
Time until the next ice age	14,000 (2^{14}) years
Time until the sun goes nova	10^9 (2^{30}) years
Age of the planet	10^9 (2^{30}) years
Age of the Universe	10^{10} (2^{34}) years
Number of atoms in the planet	10^{51} (2^{170})
Number of atoms in the sun	10^{57} (2^{190})
Number of atoms in the galaxy	10^{68} (2^{227})
Number of atoms in the Universe (dark matter excluded)	10^{77} (2^{265})
Volume of the Universe	10^{84} (2^{280}) cm^3
If the Universe is Closed:	
Total lifetime of the Universe	10^{17} (2^{57}) years 10^{26} (2^{87}) seconds
If the Universe is Open:	
Time until low-mass stars cool off	10^{14} (2^{47}) years
Time until planets detach from stars	10^{15} (2^{50}) years
Time until stars detach from galaxies	10^{19} (2^{63}) years
Time until orbits decay by gravitational radiation	10^{20} (2^{67}) years
Time until black holes decay by the Hawking process	10^{64} (2^{213}) years
Time until all matter is liquid at zero temperature	10^{65} (2^{216}) years
Time until all matter decays to iron	10^{1000} years
Time until all matter collapses to black holes	10^{1076} years

Dyson's paper, "Time Without End: Physics and Biology in an Open Universe," in *Reviews of Modern Physics*, v. 52, n. 3, July 1979, pp. 447–460. Automobile accident deaths are calculated from the Department of Transportation's statistic of 163 deaths per million people in 1993 and an average lifespan of 69.7 years.

Large Numbers

表中这些数是估计的数量级，并且是从各种资料中精选得到的，天体物理学中许多大数见 Freeman Dyson 的文章 Time Without End: Physics and Biology in an Open Universe, Reviews of Modern Physics, v. 52, n. 3, 1979.7, 447—460。汽车事故的死亡人数根据据 1993 年交通部统计数据每百万人中有 163 起死亡事故和人均寿命为 69.7 年计算出来的。

表 1-1 大数

物理模拟量	大 数
每天被闪电击死的可能性	90 亿 (2^{33}) 分之一
赢得新泽西州彩票头等奖的可能性	4 百万 (2^{22}) 分之一
赢得新泽西州彩票头等奖并且当天一大意因电击而死的可能性	2^{55}
每年溺死的可能性	59 000 (2^{16}) 分之一
1993 年在美国交通事故中死亡的可能性	6 100 (2^{13}) 分之一
一生里美国用于交通事故的可能性	88 (2^7) 分之一
到下一个冰川年代的时间	14 000 (2^{14}) 年
到太阳变成新星的时间	10^9 (2^{30}) 年
行星的年龄	10^9 (2^{30}) 年
宇宙的年龄	10^{10} (2^{34}) 年
行星中的原子数	10^{51} (2^{170})
太阳中的原子数	10^{57} (2^{190})
银河系中的原子数	10^{68} (2^{227})
宇宙中的原子数（黑洞子除外）	10^{77} (2^{265})
宇宙的体积	10^{84} (2^{280}) cm^3
如果宇宙是封闭的：	
宇宙的生命期	10^{17} (2^{57}) 年 10^{26} (2^{87}) 秒
如果宇宙是开放的：	
到小质量星冷却下来的时间	10^{14} (2^{47}) 年
到行星脱离恒星系的时间	10^{15} (2^{50}) 年
到行星脱离星际系的时间	10^{19} (2^{63}) 年
到由引力辐射造成轨道衰变的时间	10^{20} (2^{67}) 年
到霍金物理过程黑洞蒸发的时间	10^{64} (2^{213}) 年
到所有物质在零时都为液体的时间	10^{65} (2^{216}) 年
到所有物质都蜕变成铁的时间	10^{1000} 年
到所有物质都收缩为黑洞的时间	10^{1076} 年

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)

The Question: What Is Computability ?

What is computability ?

(From the viewpoint of Turing Machine)

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 investigate the nature of computation.
  - In order to find limits of computation.
  - 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.

Finite Automata: An Example [S-ToC-13] [CS101A only]

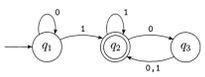


FIGURE **1.4**
A finite automaton called M_1 that has three states

Figure 1.4 is called the ***state diagram*** of M_1. It has three ***states***, labeled q_1, q_2, and q_3. The ***start state***, q_1, is indicated by the arrow pointing at it from nowhere. The ***accept state***, q_2, is the one with a double circle. The arrows going from one state to another are called ***transitions***.

For example, when we feed the input string 1101 to the machine M_1 in Figure 1.4, the processing proceeds as follows.

1. Start in state q_1.
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. *Accept* because M_1 is in an accept state q_2 at the end of the input.

Finite Automata: Formal Definition [S-ToC-13] [CS101A only]

DEFINITION **1.5**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

- **1 Transition function δ is a function from the Cartesian (direct) product ($Q \times \Sigma$) of Q and Σ to Q.**
- **2 Accept states are also called final states.**

Finite Automata: Formal Representation [S-ToC-13] [CS101A only]

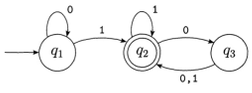


FIGURE **1.6**
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

Finite Automata: Regular Language [S-ToC-13] [CS101A only]

- **Language of finite automata**
  - If A is the set of all strings that finite automata M accepts, we say that A is *the language of M* and write $L(M) = A$, also say that M *recognizes* A.
- **Regular language**
  - A language is called a *regular language* if some finite automaton recognizes it.
- **Note**
  - Finite automata can be regarded as a simplification of Turing machines.

Finite Automata: Formal Definition of Computation [S-ToC-13] [CS101A only]

♣ **Formal definition of computation by a finite automata**

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w | M \text{ accepts } w\}$.

Language of Turing Machine

♣ **Language of Turing machine**

- **TM *M* *accepts* input string *w* if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where**
  1. C_1 is the start configuration of *M* on input *w*,
  2. each C_i yields C_{i+1}, and
  3. C_k is an accepting configuration.
- **The set of strings that *M* accepts is called *the language of M*, or *the language recognized by M*, denoted *L(M)*.**

♣ **Fact**

- ***L*(*M*) may be an infinite (*recursively enumerable*) set.**

♣ **Notes**

- **The language of a TM is an intrinsic property of that TM.**
- **Any TM accepts only one language. (Why?)**

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 "infinite loop" (does not halt, not necessarily repeat the same actions).**

♣ **Notes**

- **It is often difficult to distinguish a TM that is infinitely looping 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.**

Turing-recognizable Language

♣ **Turing-recognizable language**

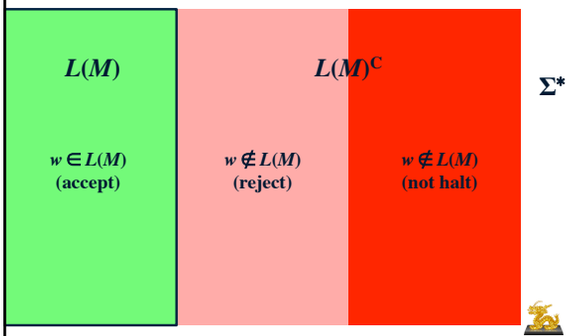
- **A language is called *Turing-recognizable* (*recursively enumerable*) if some TM recognizes it, i.e., it may be the language of a TM.**
- **A language is called *Turing-unrecognizable* if there is no TM recognizes it.**

♣ **Notes**

- **The "*Turing-recognizability*" of a language is an intrinsic property of that language.**
- **For any $w \in L(M)$, *M* must halt and accept *w*; but for any $w \notin L(M)$ (but $w \in \Sigma^*$), *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*) ⊂ Σ*. For *w* ∈ Σ*, *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

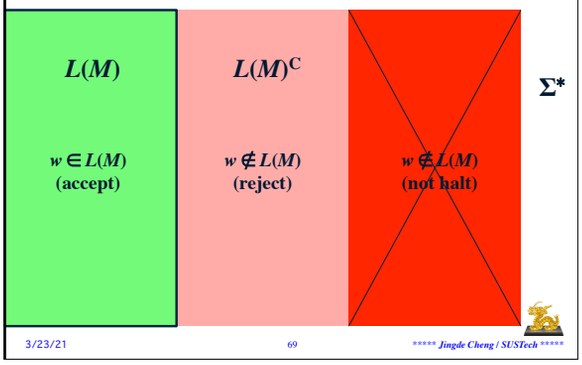
- **Turing machines as deciders**
  - A TM is called *a decider* if it halts on all inputs.
  - A decider that recognizes some language also is said to *decide* that language.
- **Notes**
  - Every decider must be a TM, but some TMs are not deciders.
  - Whether a TM is a decider or not is an intrinsic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

Turing-decidable Language

- **Turing-decidable language**
  - A language is called *Turing-decidable* (*recursive*) or simply *decidable* if some TM M decides 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".

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

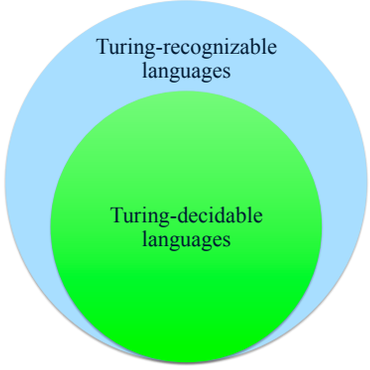
For L, there is a TM M, $L = L(M) \subset \Sigma^*$. For $w \in \Sigma^*$, w may be:



Turing-recognizable Language vs. Turing-decidable 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-decidable language**
  - The "Turing-decidability" of a language is an intrinsic property of that language.
  - Every decidable language must be Turing-recognizable, but some Turing-recognizable languages are not decidable, i.e., they have no decider.

Turing-recognizable Language vs. Turing-decidable Language



The Question: What Is Computability ?

What is computability ?

(From the viewpoint of Turing Machine)

The Notion of Algorithm

- **The notion of algorithm**
  - 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 an "*algorithm*" ("a process according to which it can be determined by a finite number of operations") 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*".
- **Necessity of the notion of algorithm**
  - Proving that an algorithm does not exist requires having a clear definition of algorithm.

The Church-Turing Thesis (1936) [S-ToC-13]

- **Church's Lambda calculus (λ-calculus) (1930s)**
  - *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**

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.

The Question: What Is Computable ?

A problem is computable

(algorithmically solvable, i.e., has an algorithm)

IFF

it is Turing-decidable

[The Church-Turing thesis]

Hilbert's 10th Problem as a Decision Problem [S-ToC-13]

- **Hilbert's 10th problem as a decision problem**
  - Let $D = \{ p \mid p$ is a polynomial Diophantine equation with integer coefficients with an integral root $\}$.
  - The problem asks in essence whether the set D is decidable.
- ***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.

From Turing Machine to Algorithm

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

From Turing Machine to Algorithm

- **Encoding of objects**
  - 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.
- **Format and 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 $\langle O_1, O_2, \ldots, O_k \rangle$.

Describing Turing Machine Algorithm

- **Format**
  - **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 TM algorithm**
  - **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]

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.

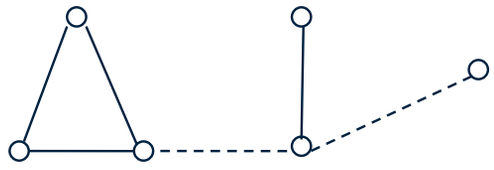
M = "On input $\langle G \rangle$, the encoding of a graph G:

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**
  - **$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$**
  - **$E(G) = \{<v_1,v_2>,<v_1,v_3>,<v_2,v_3>,<v_4,v_5>\}$**
    $E(G)' = \{<v_1,v_2>,<v_1,v_3>,<v_2,v_3>,<v_3,v_5>,<v_4,v_5>,<v_5,v_6>\}$

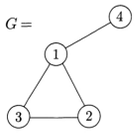
Describing Turing Machine Algorithm: An Example



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 this 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. ■

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

♣ **Undecidable language A_{TM}**

- **The problem of determining whether a TM accepts a given input string or not is undecidable.**
- **A_{TM} = { <*M*, *w*> | *M* is a TM and *M* accepts *w* }.**
- **A_{TM} is undecidable, i.e., there is no TM as a decider that can decide A_{TM}.**

♣ **A_{TM} is Turing-recognizable**

- **The following TM *U* (Turing, 1936) recognizes A_{TM}.**
- ***U* = "On input <*M*, *w*>, where *M* is a TM and *w* is a string:**
  1. Simulate *M* on input *w*.
  2. If *M* ever enters its accept state, accept; if *M* ever enters its reject state, reject. "
- **Note *U* will loop on input <*M*, *w*> if *M* loops on *w*, which is why it does not decide A_{TM}.**

Post Correspondence Problem (PCP): An Example [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.

Undecidability of PCP

♣ **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 $i_1, i_2, \ldots, i_l$, 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.**

♣ **Undecidability of PCP**

- ***PCP* = { <*P*> | *P* is an instance of the PCP with a match }**
- **Language *PCP* is not Turing-decidable, i.e., there is no TM as a decider that decides *PCP*.**

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

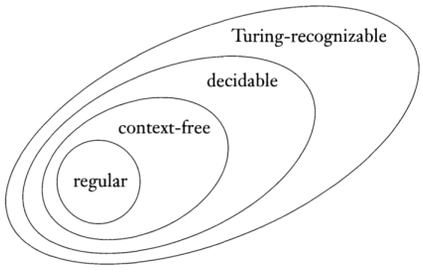


FIGURE **4.10**
The relationship among classes of languages

Turing-unrecognizable Languages

♣ **Countable set**

- **A set *C* is said to be *countable* if either it is finite or there is a bijection between it and the set of natural numbers.**

♣ **Facts**

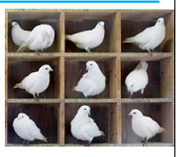
- **The set of all strings Σ^* is countable for any alphabet Σ.**
- **The set of all TMs is countable.**

♣ **Uncountable sets**

- **The set *R* of real numbers is not countable (*uncountable*), i.e., there is no bijection between it and the set of natural numbers.**
- **The set of all languages is uncountable.**

Turing-unrecognizable Languages

♣ The pigeonhole principle



- If n pigeonholes are occupied by $n + 1$ or more pigeons, then at least one pigeonhole is occupied by more than one pigeon.
- For any two finite sets A, B, and $|A| > |B|$, there is no injection f: $A \rightarrow B$.

♣ Turing-unrecognizable languages

- Because the set of real numbers is uncountable (and any subset of real numbers is a language), there are uncountably many languages.
- There are uncountably many languages yet only countably many TMs, therefore, there must be some Turing-unrecognizable languages.

Turing-unrecognizable Languages

♣ The complement of a language

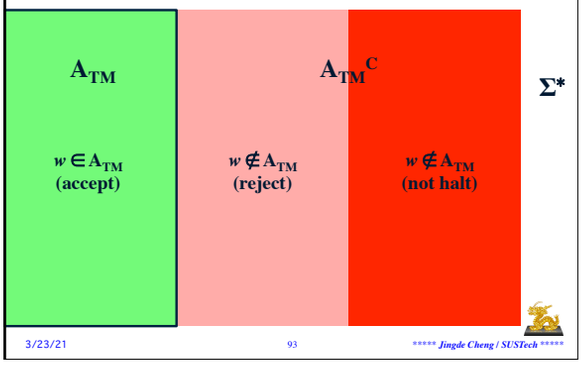
- For a language L over Σ, the *complement* of L is the language (denotes L^C) over Σ consisting of all strings that are not in L.
- $\Sigma^* = L \cup L^C$, $L \cap L^C = \varnothing$

♣ A Turing-unrecognizable language: $A_{TM}{}^C$

- Fact: A language L is decidable IFF both L and its complement L^C are Turing-recognizable.
- Note: There are two different TMs M and M^C such that M recognizes L and M^C recognizes L^C.
- The complement of language A_{TM}, i.e., $A_{TM}{}^C$, is not Turing-recognizable, because A_{TM} is undecidable but Turing-recognizable.

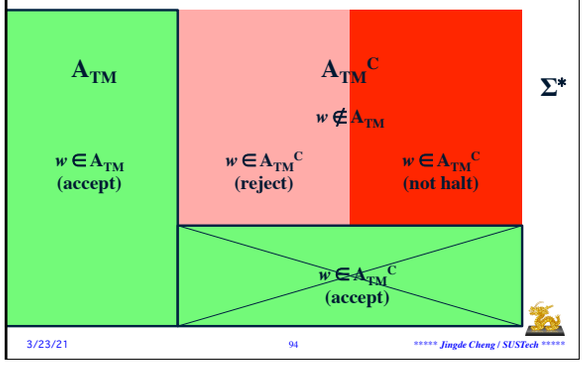
Turing-unrecognizable Language $A_{TM}{}^C$ over Σ

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



Turing-unrecognizable Languages

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

$$\Sigma^* = \{\ \varepsilon,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,001,\ \cdots\ \};$$
$$A = \{\ 0,\ 00,\ 01,\ 000,001,\ \cdots\ \};$$
$$\chi_A = 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1\ \ \cdots\ .$$

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.

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine.

The Turing-decidability of a Language

♣ The Turing-decidability of a language

- Theorem: A language L is decidable IFF both it and its complement L^c 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.

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)

The Question: What Is Computational Complexity ?

What is computational complexity ?

Computation Problem Example: Prime Factorization

♣ Prime factorization

- 8,633 = ? × ? 988,027 = ? × ? 99,400,891 = ? × ?

♣ Multiplication of prime numbers

- 9973 × 9967 = ? 997 × 991 = ? 97 × 89 = ?

♣ Question

- What can you think of from these (computable) examples?

♣ An important fact [S-ToC-13]

- "Here's a big one that remains unsolved: If I give you a large number — say, with 500 digits — can you find its factors (the numbers that divide it evenly) in a reasonable amount of time? Even using a supercomputer, no one presently knows how to do that in all cases within the lifetime of the universe!"

Computational Complexity: What Is It and Why Study It?

♣ The fundamental questions

- What a computation process can "really" compute?
- What computers can "really" do?
- What makes some problems "computationally hard" and others easy?

♣ The theory of computational complexity: What is it?

- It provides qualitative classification methods and quantitative measurement methods for computational difficulty of problems.
- It is one of the most theoretical foundations of CS.
- It is the theory of computational complexity that classifies problems into various explicitly defined classes according to their computational difficulty.

Computational Complexity: What Is It and Why Study It?

♣ The theory of computational complexity: Why study it?

- In order to know what computers can really (effectively and efficiently) do.
- In order to know what computers cannot really (effectively and efficiently) do.
- In order to define criteria for qualitative classification and metrics for quantitative measurement.
- In order to find effective and efficient algorithms for solving computational problems in the real world.
- In order to find computationally difficult problems for designing cryptographic systems.

The Theory of Computational Complexity

♣ Computationally solvable vs. really solvable problems

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory.

♣ The theory of computational complexity

- A theoretical investigation of the time, memory (space), or other resources required for solving computational problems.

♣ The fundamental questions

- What resources should be measured?
- How the resources should be measured?

The Theory of Computational Complexity

- ♣ Time complexity theory
  - ◆ It introduces a way of measuring the time used to solve a problem.
  - ◆ It classifies problems according to the amount of time required.
- ♣ Space complexity theory
  - ◆ It introduces a way of measuring the memory and/or space used to solve a problem.
  - ◆ It classifies problems according to the amount of memory and/or space required.

Measuring Time Complexity: The Fundamental Assumptions

- ♣ Measuring the general time complexity of an algorithm
  - ◆ We want to measure the general time complexity of an algorithm rather then the special time complexity of a special execution of that algorithm on a special computer.
  - ◆ Note: The real running time of a special execution of an algorithm depends on the special computer running the algorithm.
- ♣ Running time is represented by running steps
  - ◆ We compute (measure) the running steps of an algorithm rather then its real running time on some computer.
  - ◆ Note: The running steps of an algorithm is an intrinsic property of that algorithm but do not depend on computers running the algorithm.

Measuring Time Complexity: The Fundamental Assumptions

- ♣ Time complexity is represented by a function of input
  - ◆ The number of running steps that an algorithm uses on a particular input may depend on several parameters.
  - ◆ For simplicity, we compute (measure) the running steps of an algorithm purely as a function of the length of the string representing the input and do not consider any other parameters.
- ♣ Fundamental assumptions underlying the above consideration
  - ◆ The length of the string representing the input is the most important factor affecting the complexity of the computing problem.
  - ◆ The longer the input, the more complex the computing problem is.

Definition of Time Complexity of TM [S-ToC-13]

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Time Complexity Analysis

- ♣ Worst-case analysis
  - ◆ Consider the longest running time (steps) of all inputs of a particular length.
  - ◆ The *worst-case running time* of an algorithm gives us an *upper bound* on the running time for any input.
  - ◆ Knowing it provides a guarantee that the algorithm will never take any longer.
- ♣ Average-case analysis
  - ◆ Consider the *average of all running time* (steps) of all inputs of a particular length.
  - ◆ The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem.

Time Complexity Analysis

- ♣ Asymptotic analysis
  - ◆ Because the exact running time (steps) of an algorithm often is a complex expression, we usually just estimate it by one convenient form of estimation, called *asymptotic analysis*.
- ♣ Asymptotic notation: big-*O* notation
  - ◆ We consider only the highest order term of the expression for the running time (steps) of an algorithm, disregarding both the coefficient of that term and any lower order terms, because the highest order term dominates the order terms on large inputs.
  - ◆ Ex: For $f(n) = 6n^3 + 2n^2 + 20n + 45$, denote $f(n) = O(n^3)$

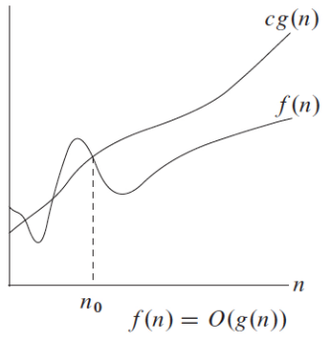
Definition of Asymptotic Upper Bound (Big-*O*) [S-ToC-13]

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Definition of Asymptotic Upper Bound (Big-*O*) [CLRS-I2A-09]



$f(n) = O(g(n))$

Asymptotic Upper Bound (Big-*O*): Examples

- ***f*(*n*) = *O*(*g*(*n*)) means that *f* is less than or equal to *g* if we disregard differences up to a constant factor.**
- **You may think of *O* as representing a suppressed constant.**
- **Let $f_1(n) = 5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.**
- **Let c be 6 and n_0 be 10 (or 9, 8, 7, 6). Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$ (or 9, 8, 7, 6).**
- **In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so is still an asymptotic upper bound on f_1.**
- **However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0, the definition remains unsatisfied in this case.**

Polynomial Bound and Exponential Bound

- **Polynomial bound**
  - **Bounds of the form $f(n) = O(n^c)$ for real number c greater than 0 are called *polynomial bounds*.**
- **Exponential bound**
  - **Bounds of the form $f(n) = O(2^{(n^c)})$ for real number c greater than 0 are called *exponential bounds*.**
  - **Note: 2 may be other integer k.**

Definition of Small-*o* Notation [S-ToC-13]

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < c\, g(n)$ for all $n \geq n_0$.

Small-*o* Notation vs. Big-*O* Notation

- **Small-*o* notation**
  - **Small-*o* notation is used to represent that one function is asymptotically less than another function.**
- **Big-*O* notation**
  - **Big-*O* notation is used to represent that one function is asymptotically no more than another function.**
- **The difference between big-*O* notation and small-*o* notation**
  - **The difference between big-*O* notation and small-*o* notation is analogous to the difference between $\leq$ and $<$.**

Small-o Notation vs. Big-O Notation

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < c\, g(n)$ for all $n \geq n_0$.

Comparison of Asymptotic Notations [CLRS-I2A-09]

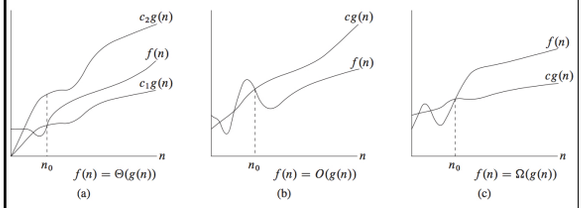


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

Definition of Θ-Notation [CLRS-I2A-09]

we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}\ .^{1}$$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion. You might be confused because we abuse equality in this way, but we shall see later in this section that doing so has its advantages.

Figure 3.1(a) gives an intuitive picture of functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$. For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be ***asymptotically nonnegative***, that is, that $f(n)$ be nonnegative whenever n is sufficiently large. (An ***asymptotically positive*** function is one that is positive for all sufficiently large n.) Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within Θ-notation is asymptotically nonnegative. This assumption holds for the other asymptotic notations defined in this chapter as well.

Definition of O-Notation [CLRS-I2A-09]

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}\ .$$

We use O-notation to give an upper bound on a function, to within a constant factor. Figure 3.1(b) shows the intuition behind O-notation. For all values n at and to the right of n_0, the value of the function $f(n)$ is on or below $cg(n)$.

We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$. Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. Thus, our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any such quadratic function is in $O(n^2)$. What may be more surprising is that when $a > 0$, any *linear* function $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = \max(1, -b/a)$.

If you have seen O-notation before, you might find it strange that we should write, for example, $n = O(n^2)$. In the literature, we sometimes find O-notation informally describing asymptotically tight bounds, that is, what we have defined using Θ-notation. In this book, however, when we write $f(n) = O(g(n))$, we are merely claiming that some constant multiple of $g(n)$ is an asymptotic upper bound on $f(n)$, with no claim about how tight an upper bound it is. Distinguishing asymptotic upper bounds from asymptotically tight bounds is standard in the algorithms literature.

Definition of Ω-Notation [CLRS-I2A-09]

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\ .$$

Figure 3.1(c) shows the intuition behind Ω-notation. For all values n at or to the right of n_0, the value of $f(n)$ is on or above $cg(n)$.

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

Definition of o-Notation [CLRS-I2A-09]

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight. We formally define $o(g(n))$ ("little-oh of g of n") as the set

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}\ .$$

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for *all* constants $c > 0$. Intuitively, in o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\ .$$

Definition of ω-Notation [CLRS-I2A-09]

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty ,$$

if the limit exists. That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

Example of Measuring Complexity and Analyzing Algorithm [S-ToC-13]

Let's analyze the TM algorithm we gave for the language $A = \{0^k1^k | k \geq 0\}$. We repeat the algorithm here for convenience.

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Example of Measuring Complexity and Analyzing Algorithm [S-ToC-13]

To analyze M_1 we consider each of its four stages separately. In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*. Performing this scan uses n steps. As we mentioned earlier, we typically use n to represent the length of the input. Repositioning the head at the left-hand end of the tape uses another n steps. So the total used in this stage is $2n$ steps. In big-O notation we say that this stage uses $O(n)$ steps. Note that we didn't mention the repositioning of the tape head in the machine description. Using asymptotic notation allows us to omit details of the machine description that affect the running time by at most a constant factor.

In stages 2 and 3, the machine repeatedly scans the tape and crosses off a 0 and 1 on each scan. Each scan uses $O(n)$ steps. Because each scan crosses off two symbols, at most $n/2$ scans can occur. So the total time taken by stages 2 and 3 is $(n/2)O(n) = O(n^2)$ steps.

In stage 4 the machine makes a single scan to decide whether to accept or reject. The time taken in this stage is at most $O(n)$.

Thus the total time of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$. In other words, its running time is $O(n^2)$, which completes the time analysis of this machine.

Example of Measuring Complexity and Analyzing Algorithm [S-ToC-13]

only by a factor of 2 and doesn't affect the asymptotic running time. The following machine, M_2, uses a different method to decide A asymptotically faster. It shows that $A \in \text{TIME}(n \log n)$.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Example of Measuring Complexity and Analyzing Algorithm [S-ToC-13]

Before analyzing M_2, let's verify that it actually decides A. On every scan performed in stage 4, the total number of 0s remaining is cut in half and any remainder is discarded. Thus, if we started with 13 0s, after stage 4 is executed a single time only 6 0s remain. After subsequent executions of this stage, 3, then 1, and then 0 remain. This stage has the same effect on the number of 1s.

Now we examine the even/odd parity of the number of 0s and the number of 1s at each execution of stage 3. Consider again starting with 13 0s and 13 1s. The first execution of stage 3 finds an odd number of 0s (because 13 is an odd number) and an odd number of 1s. On subsequent executions an even number (6) occurs, then an odd number (3), and an odd number (1). We do not execute this stage on 0 0s or 0 1s because of the condition on the repeat loop specified in stage 2. For the sequence of parities found (odd, even, odd, odd) if we replace the evens with 0s and the odds with 1s and then reverse the sequence, we obtain 1101, the binary representation of 13, or the number of 0s and 1s at the beginning. The sequence of parities always gives the reverse of the binary representation.

When stage 3 checks to determine that the total number of 0s and 1s remaining is even, it actually is checking on the agreement of the parity of the 0s with the parity of the 1s. If all parities agree, the binary representations of the numbers of 0s and of 1s agree, and so the two numbers are equal.

To analyze the running time of M_2, we first observe that every stage takes $O(n)$ time. We then determine the number of times that each is executed. Stages 1 and 5 are executed once, taking a total of $O(n)$ time. Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + \log_2 n$ iterations of the repeat loop occur before all get crossed off. Thus the total time of stages 2, 3, and 4 is $(1 + \log_2 n)O(n)$, or $O(n \log n)$. The running time of M_2 is $O(n) + O(n \log n) = O(n \log n)$.

Example of Measuring Complexity and Analyzing Algorithm [S-ToC-13]

We can decide the language A in $O(n)$ time (also called ***linear time***) if the Turing machine has a second tape. The following two-tape TM M_3 decides A in linear time. Machine M_3 operates differently from the previous machines for A. It simply copies the 0s to its second tape and then matches them against the 1s.

M_3 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*."

This machine is simple to analyze. Each of the four stages uses $O(n)$ steps, so the total running time is $O(n)$ and thus is linear. Note that this running time is the best possible because n steps are necessary just to read the input.

Time Complexity Classes of Languages (Problems)

♣ **Time complexity class**

- Let t: $N \rightarrow R^+$ be a function where R^+ is the set of non-negative real numbers.
- We may define the *time complexity class*, TIME($t(n)$), to be the collection of all languages that are decidable by an $O(t(n))$ time (steps) Turing machine.

♣ **Notes**

- The time complexity class is a class of languages (problems) but not Turing machines.
- There may be various time complexity classes because there may be a lot of functions $t1$, $t2$, …, .
- A language may be a member of different classes if there are different TMs to decide it.

Time Complexity Classes of Languages (Problems)

♣ **Linear time complexity**

- $O(n)$ time is called *linear time*.

♣ **An important fact about regular languages**

- Any language that can be decided in $o(n \log n)$ time on a single-tape deterministic Turing machine is regular.

♣ **Facts about language $A = \{0^k1^k \mid k \geq 0\}$**

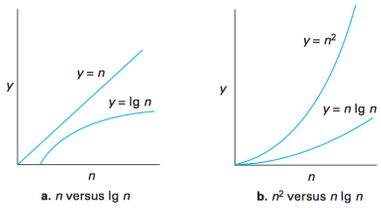
- The example M_1 shows $A \in \text{TIME}(n^2)$.
- The example M_2 shows $A \in \text{TIME}(n \log n)$.
- The example M_3 shows $A \in \text{TIME}(n)$.
- Language A is a member of three different classes, because there are three different TMs to decide it !

The Question: What a Computation Process Can "Really" Compute ?

What a computation process can "really" compute ?

Bounded Relationship among Mathematical Expressions

Figure 12.11 Graphs of the mathematical expressions n, lg n, n lg n, and n^2



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 [Cheng, 2015]

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。

Difference between Computability Theory and Complexity Theory?

Difference between computability theory and complexity theory ?

Difference between Computability Theory and Complexity Theory

♣ **An important difference between computability theory and complexity theory**

- **In computability theory, the Church-Turing thesis implies that all reasonable models of computation are equivalent, i.e., they are all decide the same class of languages.**
- **In complexity theory, the choice of computation model affects the time complexity of languages.**

♣ **With which model do we measure time complexity?**

- **Time requirements do not differ greatly for typical deterministic models.**
- **If our classification system is not very sensitive to relatively small difference in complexity, the choice of deterministic model is not crucial.**

Important Questions

♣ **About difference between computability theory and complexity theory**

- **What can you think of from the difference between computability theory and complexity theory, if you want to be a scientist?**
- **What can you think of from the difference between computability theory and complexity theory, if you want to be an engineer?**

♣ **Science vs Engineering**

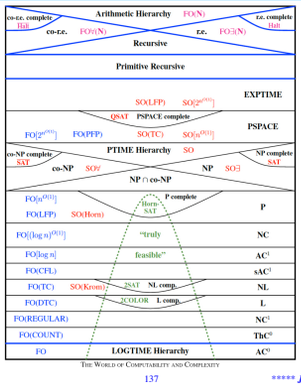
- **What/Why vs How**
- **Theory vs Practice**
- **Methodologies vs Tools**

Time Complexities [Wikiprdia]

The following table summarizes some classes of commonly encountered time complexities. In the table, poly(x) = $x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest or largest item in an unsorted array, Kadane's algorithm
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
quasilinear time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort; Fast Fourier transform.
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort; Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two n×n matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n \cdot \log \log n})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time (with linear exponent)	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

The World of Computability and Complexity [SEP]



The World of Computability and Complexity

Inclusion Relationships among Major Complexity Classes [SEP]

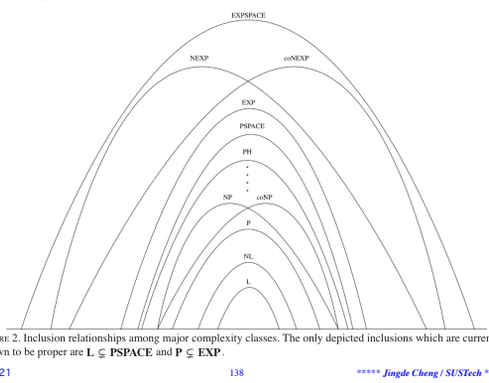


FIGURE 2. Inclusion relationships among major complexity classes. The only depicted inclusions which are currently known to be proper are **L** ⊊ **PSPACE** and **P** ⊊ **EXP**.

P = NP? (the Greatest Problem in TCS and Math)

P = NP ?

(the Greatest Problem in TCS and Math)

The Class P: Problems with Polynomial Time Complexity

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

The Class P: Problems with Polynomial Time Complexity

♣ **Why the class P is intrinsically important?**

- **All reasonable deterministic computational models are *polynomially equivalent*, i.e., any one of them can simulate another with only a polynomial increase in running time.**
- **P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.**
- **P roughly corresponds to the class of problems that are realisitically solvable on a computer.**

♣ **Notes**

- **P is a mathematical robust (stable) class.**
- **P is relevant (rational) from a practical standpoint.**

The Class NP: Problems with Nondeterministic Polynomial Time Complexity

$\mathbf{NTIME}(t(n)) = \{L \mid L$ is a language decided by a $O(t(n))$ time nondeterministic Turing machine$\}$.

$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$.

The Class NP: Problems with Nondeterministic Polynomial Time Complexity

♣ **Verifier**

- **A *verifier* for a language *A* is an algorithm (TM) *V*, where *A* = { *w* | *V* accepts <*w*, *c*> for some string *c* }**
- **We measure the time of a verifier only in terms of the length of *w*, so a polynomial time verifier runs in polynomial time in the length of *w*.**
- **A language *A* is *polynomially verifiable* if it has a polynomial time verifier.**

♣ **Note**

- **A verifier uses additional information, represented by the symbol *c* in the above definition, to verify that a string *w* is a member of *A*. This information is called a *certificate* or *proof* of membership in *A*.**

The Class NP: Problems with Nondeterministic Polynomial Time Complexity

♣ **The Class NP**

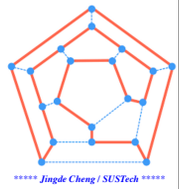
- **NP is the class of languages that have polynomial time verifiers.**

♣ **Fact about the class NP**

- **A language is in NP IFF it is decided by some nondeterministic polynomial time Turing machine.**

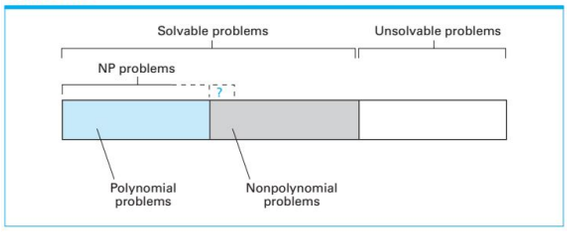
♣ **NP problem example**

- **Hamiltonian path/cycle problem.**
- **For any given undirected or directed graph G, determine whether a Hamiltonian path/cycle (a path/cycle that visits each vertex exactly once) exists.**



Problem (Language) Classification

Figure 12.12 A graphic summation of the problem classification



P = NP? (the Greatest Problem in TCS and Math) [S-ToC-13]

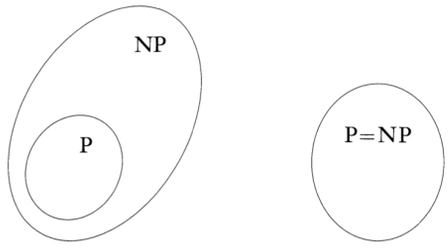


FIGURE **7.26**
One of these two possibilities is correct

NP-Complete Problems

- **NP-completeness**
  - There is a certain set of problems in NP class whose individual complexity is related to that of the entire class.
  - If a polynomial time algorithm exists for any of these problems, all problems in NP class would be polynomial time solvable. These problems are called *NP-complete*.
- **NP-complete problem example: Satisfiability (SAT) problem**
  - A logical formula is satisfiable if some truth-value assignment makes it true.
  - The satisfiability problem (the first NP-complete problem) is to test whether a logical formula is satisfiable.
  - SAT = { <*wff*> | *wff* is a satisfiable logical formula }
  - SAT ∈ P IFF P = NP.

Why NP-completeness is Important

- **From the theoretical viewpoint**
  - To show P ≠ NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete problem does.
  - To show P = NP only needs to find a polynomial time algorithm for an NP-complete problem.
- **From the practical viewpoint**
  - The phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem.
  - Proving that a problem is NP-complete is strong evidence of its nonpolynomiality.

Polynomial Time Reducibility

- **Polynomial time computable function**
  - A function $f: \Sigma^* \rightarrow \Sigma^*$ is a *polynomial time computable function* if some polynomial time deterministic Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.
- **Polynomial time mapping reducible**
  - Language A is *polynomial time mapping reducible*, or simply *polynomial time reducible*, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \rightarrow \Sigma^*$ exists, where for every w, $w \in A$ IFF $f(w) \in B$.
  - The function f is called the *polynomial time reduction* of A to B.
  - If $A \leq_P B$ and $B \in$ P, then $A \in$ P.

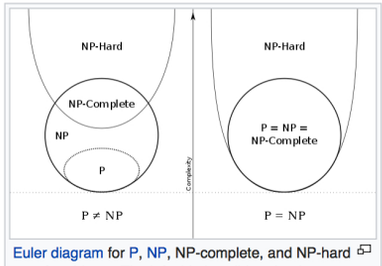
NP-Completeness

- **Definition of NP-completeness**
  - A language B is *NP-complete* if it satisfies two conditions:
    1. B is in NP, and
    2. every A in NP is polynomial time reducible to B.
- **Theorems about NP-completeness**
  - If B is NP-complete and $B \in$ P, then P = NP.
  - If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.
  - Note: Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it.
- **The Cook-Levin theorem (The first NP-complete problem)**
  - SAT is NP-complete.

NP-Hard Problems

- **Definition of NP-hardness**
  - **A problem is *NP-hard* if all problems in NP class are polynomial time reducible to it, even though it may not be in NP class itself.**
- **Difference between NP-completeness and NP-Hardness**
  - **A NP-complete problem must be in NP class, while a NP-hard problem may not be in NP class.**
- **NP-Hard problem example**
  - **Hilbert's 10th problem: to test whether a polynomial Diophantine equation with integer coefficients has an integral root.**
  - **D = { <*p*> | *p* is a polynomial Diophantine equation with integer coefficients an integral root } (D is undecidable)**

Relationship among P, NP, NP-complete, and NP-Hard Classes [Wiki]



Euler diagram for P, NP, NP-complete, and NP-hard set of problems. The left side is valid under the assumption that P≠NP, while the right side is valid under the assumption that P=NP (except that the empty language and its complement are never NP-complete)

An Introduction to Computer Science

- **Computer Science: What Is It and Why Study It?**
- **Computation: What Is It and Why Study It?**
- **Computability**
- **Computational Complexity (CS101A class only)**
- **Algorithms**
- **Data, Information, and Knowledge, and Their Representations**
- **Data Storage**
- **Computer Architecture**
- **Data Manipulation in Computer Systems**
- **Programming Languages and Compilers**
- **Operating Systems**
- **System Software and Application Software**
- **Software Engineering (CS101A class only)**
- **Knowledge Engineering and Artificial Intelligence (CS101A class only)**
- **Information Security Engineering (CS101A class only)**

The Question: What Is an Algorithm ?

What is an algorithm ?

Euclidean Algorithm: The Oldest Algorithm (300 BC)

Figure 0.2 The Euclidean algorithm for finding the greatest common divisor of two positive integers

Description: This algorithm assumes that its input consists of two positive integers and proceeds to compute the greatest common divisor of these two values.

Procedure:

Step 1. Assign M and N the value of the larger and smaller of the two input values, respectively.

Step 2. Divide M by N, and call the remainder R.

Step 3. If R is not 0, then assign M the value of N, assign N the value of R, and return to step 2; otherwise, the greatest common divisor is the value currently assigned to N.

©Bibliocraft Ltd. & Co. KG/Alamy Stock Photo

EUCLID (325 B.C.E.– 265 B.C.E.) Euclid was the author of the most successful mathematics book ever written, *The Elements*, which appeared in over 1000 different editions from ancient to modern times. Little is known about Euclid's life, other than that he taught at the famous academy at Alexandria in Egypt. Apparently, Euclid did not stress applications. When a student asked what he would get by learning geometry, Euclid explained that knowledge was worth acquiring for its own sake and told his servant to give the student a coin "because he must make a profit from what he learns."

Euclidean Algorithm: The Oldest Algorithm (300 BC)

7.2.4 Computing Greatest Common Divisors

The "obvious" way to compute the greatest common divisor of two positive integers n and m is to try all candidate divisors $d \in \{1, 2, \ldots, \min(n, m)\}$ and to return the largest value of d that indeed evenly divides both n and m. This algorithm is slow—very slow!—but there is a faster way to solve the problem. Amazingly, a faster algorithm for computing GCDs has been known for approximately 2300 years: the *Euclidean algorithm*, named after the Greek geometer Euclid, who lived in the 3rd century BCE. (Euclid is also the namesake of the *Euclidean distance* between points in the plane—see Exercise 2.174—among a number of other things in mathematics.) The algorithm is shown in Figure 7.2.

Euclid(n, m):
Input: positive integers n and $m \geq n$
Output: gcd(n, m)
1: **if** m mod $n = 0$ **then**
2: **return** n
3: **else**
4: **return Euclid**$(m \bmod n, n)$

Figure 7.2: The Euclidean algorithm for GCDs.

Euclidean Algorithm: The Oldest Algorithm (300 BC)

Euclidean Algorithm Description

1. Let A and B be integers with $A > B \geq 0$.
2. To find the greatest common divisor of A and B, first check whether $B = 0$. If it is, then $\gcd(A, B) = A$ by Lemma 4.10.1. If it isn't, then $B > 0$ and the quotient-remainder theorem can be used to divide A by B to obtain a quotient q and a remainder r:

$$A = Bq + r \quad \text{where } 0 \leq r < B.$$

By Lemma 4.10.2, $\gcd(A, B) = \gcd(B, r)$. Thus the problem of finding the greatest common divisor of A and B is reduced to the problem of finding the greatest common divisor of B and r.

[What makes this information useful is the fact that the larger number of the pair (B, r) is smaller than the larger number of the pair (A, B). The reason is that the value of r found by the quotient-remainder theorem satisfies

$$0 \leq r < B.$$

And, since by assumption $B < A$, we have that

$$0 \leq r < B < A.]$$

3. Now just repeat the process, starting again at (2), but use B instead of A and r instead of B. The repetitions are guaranteed to terminate eventually with $r = 0$ because each new remainder is less than the preceding one and all are nonnegative.

Euclidean Algorithm: The Oldest Algorithm (300 BC)

Euclidean Algorithm for Finding the gcd.
The input consists of two natural numbers m, n, with $(m, n) \neq (0, 0)$.

begin
  $a := m$; $b := n$;
  if $a < b$ **then**
    $t := b$; $b := a$; $a := t$; (swap a and b)
  while $b \neq 0$ **do**
    $r := a$ mod b; (divide a by b to obtain the remainder r)
    $a := b$; $b := r$
  endwhile;
  $\gcd(m, n) := a$
end

Euclidean Algorithm: The Oldest Algorithm (300 BC)

Algorithm 1: The Euclidean algorithm.
procedure $gcd(a, b$: positive integers)
  $r_0 := a$; $r_1 := b$
  $i := 1$
  while $r_i \neq 0$
    $r_{i+1} := r_{i-1}$ mod r_i
    $i := i + 1$
{gcd(a, b) is r_{i-1}}

ALGORITHM 1 The Euclidean Algorithm.

procedure $gcd(a, b$: positive integers)
$x := a$
$y := b$
while $y \neq 0$
  $r := x$ **mod** y
  $x := y$
  $y := r$
return x {gcd(a, b) is x}

Euclidean Algorithm: The Oldest Algorithm (300 BC)

- **A question**
  - **What can you think of from this algorithm?**
- **What should you have observed**
  - **The algorithm has a goal as its computing object (GCD).**
  - **The algorithm has inputs (two positive integers *M* and *N*).**
  - **The algorithm has a specified ordered sequence of steps.**
  - **The algorithm has three basic control structures: sequence, conditional branch (decision), and loop (iteration).**
  - **The algorithm must halt.**
  - **The algorithm has an output (the GCD of *M* and *N*).**

Algorithm: Historical Notes [Wikipedia]

- **"The word 'algorithm' has its roots in latinizing the name of Muhammad ibn Musa al-Khwarizmi in a first step to algorismus. Al-Khwārizmī (c. 780–850) was a Persian mathematician, astronomer, geographer, and scholar in the House of Wisdom in Baghdad, whose name means 'the native of Khwarezm', a region that was part of Greater Iran and is now in Uzbekistan."**
- **"About 825, al-Khwarizmi wrote an Arabic language treatise on the Hindu–Arabic numeral system, which was translated into Latin during the 12th century under the title Algoritmi de numero Indorum. This title means "Algoritmi on the numbers of the Indians", where "Algoritmi" was the translator's Latinization of Al-Khwarizmi's name. Al-Khwarizmi was the most widely read mathematician in Europe in the late Middle Ages, primarily through another of his books, the Algebra. In late medieval Latin, algorismus, English 'algorism', the corruption of his name, simply meant the "decimal number system". In the 15th century, under the influence of the Greek word ἀριθμός 'number' (cf. 'arithmetic'), the Latin word was altered to algorithmus, and the corresponding English term 'algorithm' is first attested in the 17th century; the modern sense was introduced in the 19th century."**

Algorithm: What Is It?

- **Algorithm is the core of computation**
  - **No computation does not have an algorithm.**
  - **Algorithms are at the core of all technologies used in contemporary computing systems.**
- **Algorithm is the key of solving problems**
  - **The notions and theories of computation, computability, computational complexity only provide conceptual and theoretical frameworks, it is various algorithms that provide concrete solutions for various problems.**
- **Algorithm is the engine driving CS**
  - **Many progresses of CS is leaded by new algorithms.**
  - **Algorithm is the engine driving the progress of CS.**

Definitions of Algorithm

Figure 5.1 The definition of an algorithm

An algorithm is an ordered set
of unambiguous, executable steps
that defines a terminating process.

Definitions of Algorithm [CLRS-I2A-09]

Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified ***computational problem***. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

G. Polya's "How to Solve It" [DL-CS-16]

HOW TO SOLVE IT

UNDERSTANDING THE PROBLEM

First. You have to *understand* the problem. *What is the unknown? What are the data? What is the condition?* Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory? Draw a figure. Introduce suitable notation. Separate the various parts of the condition. Can you write them down?

DEVISING A PLAN

Second. Find the connection between the data and the unknown. You may be obliged to consider auxiliary problems if an immediate connection cannot be found. You should obtain eventually a *plan* of the solution. Have you seen it before? Or have you seen the same problem in a slightly different form? *Do you know a related problem?* Do you know a theorem that could be useful? *Look at the unknown!* And try to think of a familiar problem having the same or a similar unknown. *Here is a problem related to yours and solved before. Could you use it?* Could you use its result? Could you use its method? Should you introduce some auxiliary element in order to make its use possible? Could you restate the problem? Could you restate it still differently? Go back to definitions. If you cannot solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? *Keep only a part* of the condition, drop the other part; how far is the unknown then determined; how can it vary? Could you derive something useful from the data? Could you think of other data appropriate to determine the unknown? Could you change the unknown or the data, or both if necessary, so that the new unknown and the new data are nearer to each other? Did you use all the data? Did you use the whole condition? Have you taken into account all essential notions involved in the problem?

CARRYING OUT THE PLAN

Third. *Carry out* your plan. Carrying out your plan of the solution, *check each step.* Can you see clearly that the step is correct? Can you prove that it is correct?

LOOKING BACK

Fourth. *Examine* the solution obtained. Can you *check the result?* Can you check the argument? Can you derive the result differently? Can you see it at a glance? Can you use the result, or the method, for some other problem?

FIGURE 7.1 Polya's How to Solve It list

Reproduced from POLYA, G.; HOW TO SOLVE IT. © 1945 Princeton University Press, 1973 renewed PUP. Reproduced with permission of Princeton University Press

G. Polya [DL-CS, 2002]

George Polya

George Polya was born in Budapest on December 13, 1887. Although he became known as a world famous mathematician, he did not show an early interest in mathematics. His lack of interest might be explained by his memory of three high school mathematics teachers: "two were despicable and one was good."

In 1905, Polya entered the University of Budapest, where he studied law at the insistence of his mother. After one very boring semester, he decided to study languages and literature. He earned a teaching certificate in Latin and Hungarian—and never used it. He became interested in philosophy and took courses in math and physics as part of his philosophy studies. He settled on mathematics, commenting that "I am too good for philosophy and not good enough for physics. Mathematics is in between." He received his Ph.D. in mathematics in 1912, which launched his career.

Polya did research and taught at the University of Göttingen, the University of Paris, and the Swiss Federation of Technology in Zurich. While in Zurich he interacted with John von Neumann, about whom he said, "Johnny was the only student I was ever afraid of. If, in the course of a lecture, I stated an unsolved problem, the chances were he'd come to me as soon as the lecture was over, with the complete solution in a few scribbles on a slip of paper."

Like many Europeans of that era, he moved to the United States in 1940 because of the political situation in Germany. After teaching at Brown University for two years, he moved to Palo Alto to teach at Stanford, where he remained for the rest of his career.

Polya's research and publications encompassed many areas of mathematics, including number theory, combinatorics, astronomy, probability, integral functions, and boundary value problems for partial differential equations. The George Polya Prize is given in his honor for notable application of combinatorial theory.

Yet, for all George Polya's contributions to mathematics, it is his contribution to mathematics education for which he was the most proud and for which he will be the most remembered. His book, *How to Solve It*, published in 1945, sold over a million copies and was translated into 17 languages. In this book, Polya outlines a problem-solving strategy designed for mathematical problems. The generality of the strategy makes it applicable to all problem solving, however. Polya's strategy is the basis of the computer problem-solving strategy outlined in this text. *Mathematics and Plausible Reasoning*, published in 1954, was another book dedicated to mathematics education. He not only wrote about mathematics education, but also took an active interest in the teaching of mathematics. He was a regular visitor to the schools in the Bay Area and visited most of the colleges in the western states. The Math Center at the University of Idaho is named for him.

On September 7, 1985, George Polya died in Palo Alto at the age of 97.

Definitions of Algorithm [DL-CS-16]

Algorithms

The last sentence in the second step in Polya's list says that you should eventually obtain a plan of the solution. In computing, this plan is called an **algorithm**. We have used the term many times; here we define it in computing terms. Formally, an algorithm is a set of instructions for solving a problem or subproblem in a finite amount of time using a finite amount of data. Implicit in this definition is the understanding that the instructions are unambiguous.

Algorithm Unambiguous instructions for solving a problem or subproblem in a finite amount of time using a finite amount of data

In computing, we must make certain conditions explicit that are implicit in human solutions. For example, in everyday life we would not consider a solution that takes forever to be much of a solution. We would also reject a solution that requires us to process more information than we are capable of processing. These constraints must be explicit in a computer solution, so the definition of an algorithm includes them.

Definitions of Algorithm [F-CS-18]

Algorithm: An ordered set of unambiguous steps that produces a result and terminates in a finite time.

- **Well-defined**
  - **An algorithm must be a well-defined, ordered set of instructions.**
- **Unambiguous steps**
  - **Each step in an algorithm must be clearly and unambiguously defined.**
- **Produce a result**
  - **An algorithm must produce a result, otherwise it is useless.**
- **Terminate in a finite time**
  - **An algorithm must terminate (halt).**

The Question: Why Study Algorithms ?

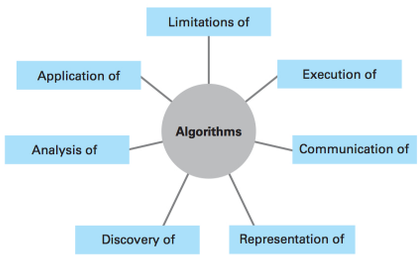
Why study algorithms ?

Algorithm: Why Study It?

- **In order to do computation to solve problems effectively**
  - Any meaningful computation must have an algorithm.
- **In order to do computation to solve problems efficiently**
  - Some computationally difficult problems require efficient algorithms.
  - Computing time is a bounded resource, and so is space in memory. We should use these resources wisely, and algorithms that are efficient in terms of time or space will help us do so.
- **In order to become a professional**
  - Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled professionals from the novices.

The Central Role of Algorithms in Computer Science

Figure 0.5 The central role of algorithms in computer science



Exercise Problem: Comparison of Running Times [CLRS-I2A-09]

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

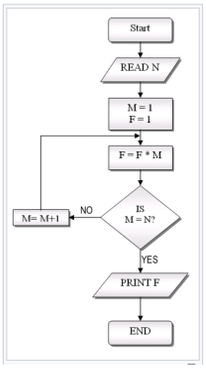
The Question: How to Represent Algorithms ?

How to represent algorithms ?

Algorithm Representation: Flowcharts

- **Flowcharts**
  - A *flowchart* is a digraph that represents an algorithm or process such that its various kinds of vertexes (nodes) are represent computation steps (actions), and its arcs (arrows) represent control transitions.
- **Flowchart primitive graphic symbols**
  - Arrow is used to represent control flow.
  - Oval (ellipse) is used to to represent start/termination.
  - Oblong (rectangle) is used to represent process of operations.
  - Rhombus (diamond) is used to represent conditional branch (decision).
  - Parallelogram is used to represent input/output.

Flowchart: An Example



Algorithm Representation: Flowcharts

- **Notes**
  - **For a complex algorithm/process, its flowchart often become tangled webs of crisscrossing arrows that made understanding the structure of the underlying algorithm difficult.**
  - **It is necessary to decompose a complex problem (algorithm) into some sub-problems (sub-algorithm) to have a hierarchical structure.**

ANSI/ISO Standard Symbols of Flowcharts [Wikipedia]

The American National Standards Institute (ANSI) set standards for flowcharts and their symbols in the 1960s.[13] The International Organization for Standardization (ISO) adopted the ANSI symbols in 1970.[13] The current standard was revised in 1985.[13] Generally, flowcharts flow from top to bottom and left to right.[17]

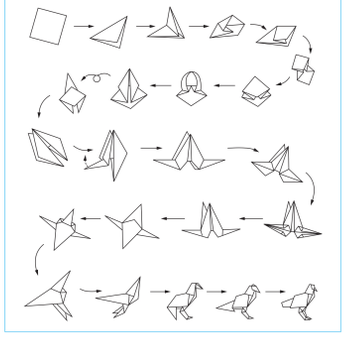
ANSI/ISO Shape	Name	Description
	Flowline (Arrowhead)[18]	Shows the process's order of operation. A line coming from one symbol and pointing at another.[14] Arrowheads are added if the flow is not the standard top-to-bottom, left-to right.[17]
	Terminal[14]	Indicates the beginning and ending of a program or sub-process. Represented as a stadium,[14] oval or rounded (fillet) rectangle. They usually contain the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit inquiry" or "receive product".
	Process[18]	Represents a set of operations that changes value, form, or location of data. Represented as a rectangle.[14]
	Decision[14]	Shows a conditional operation that determines which one of the two paths the program will take.[14] The operation is commonly a yes/no question or true/false test. Represented as a diamond (rhombus).[17][2]
	Input/Output[18]	Indicates the process of inputting and outputting data,[18] as in entering data or displaying results. Represented as a parallelogram.[14]
	Annotation[14] (Comment)[19]	Indicating additional information about a step the program. Represented as an open rectangle with a dashed or solid line connecting it to the corresponding symbol in the flowchart.[19]
	Predefined Process[14]	Shows named process which is defined elsewhere. Represented as a rectangle with double-struck vertical edges.[14]
	On-page Connector[14]	Pairs of labeled connectors replace long or confusing lines on a flowchart page. Represented by a small circle with a letter inside.[14][18]
	Off-page Connector[14]	A labeled connector for use when the target is on another page. Represented as a home plate-shaped pentagon.[14][18]

Algorithm Representation: Primitives

- **Primitives**
  - **CS establishes a well-defined (in syntax and semantics) set of building blocks from which algorithm representations can be constructed. Such a building block is called a *primitive*.**
  - **Assigning precise definitions to these primitives removes many problems of ambiguity, and requiring algorithms to be described in terms of these primitives establishes a uniform level of detail.**
- **Programming languages**
  - **A collection of primitives along with a collection of rules stating how the primitives can be combined to represent more complex ideas constitutes a programming language.**

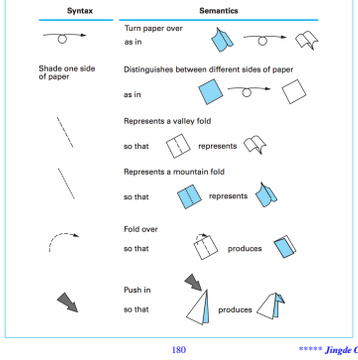
Origami "Algorithm"

Figure 5.2 Folding a bird from a square piece of paper



Origami Primitives

Figure 5.3 Origami primitives



Algorithm Representation: Pseudo-code

♣ **Pseudo-code**

- **In general, a *pseudo-code* is a notational system in which ideas can be expressed informally during the algorithm development process.**
- **One way to obtain a pseudo-code is simply to loosen the rules of the formal language in which the final version of the algorithm is to be expressed.**
- **The goal is to consider the issues of algorithm development and representation without confining our discussion to a particular programming language.**

♣ **Note**

- **A pseudo-code should be "as simple as possible, but not simpler".**

Algorithm Representation: Pseudo-code

♣ **Pseudo-code primitive examples**

- *variable name* ← (**:=**) *expression* **(assignment statement)**
- **";" (sequential execution)**
- **if** (*condition*) **then** (*activity*) **[else** (*activity*)**]** **[end if]** **(conditional branch)**
- **while** (*condition*) **do** (*activity*) **(iteration, loop)**
- **procedure name [**(*parameters*)**]**

♣ **Pseudo-code algorithm example**

- **procedure** *Greetings*
  Count ← 3;
  while (*Count* > 0) **do**
  (**print** *the message "Hello"* **and** *Count* ← *Count* – 1)

Algorithm Example: Sequential Search in Sorted List

♣ **The sequential search algorithm in pseudo-code**

Figure 5.6 The sequential search algorithm in pseudocode

```
procedure Search (List, TargetValue)
if (List empty)
   then
     (Declare search a failure)
   else
     (Select the first entry in List to be TestEntry;
      while (TargetValue > TestEntry and
             there remain entries to be considered)
         do (Select the next entry in List as TestEntry.);
      if (TargetValue = TestEntry)
         then (Declare search a success.)
         else (Declare search a failure.)
     ) end if
```

♣ **Home work**

- **Show the flowchart of the sequential search algorithm.**

Algorithm Example: (Recursive) Binary Search

Figure 5.14 The binary search algorithm in pseudocode

```
procedure Search (List, TargetValue)
if (List empty)
  then
   (Report that the search failed.)
  else
   [Select the "middle" entry in List to be the TestEntry;
    Execute the block of instructions below that is
      associated with the appropriate case.
        case 1: TargetValue = TestEntry
              (Report that the search succeeded.)
        case 2: TargetValue < TestEntry
              (Apply the procedure Search to see if TargetValue
                 is in the portion of the List preceding TestEntry,
                 and report the result of that search.)
        case 3: TargetValue > TestEntry
              (Apply the procedure Search to see if TargetValue
                 is in the portion of List following TestEntry,
                 and report the result of that search.)
   ] end if
```


Algorithm Example: Insertion Sorting [CLRS-I2A-09]

♣ **The definition of sorting problem**

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

- **Ex: Given the input sequence <31, 41, 59, 26, 41, 58>, a sorting algorithm returns as output the sequence <26, 31, 41, 41, 58, 59>. Such an input sequence is called an instance of the sorting problem.**

♣ **Sorting a hand of cards**

- **We start with an empty left hand and the cards face down on the table.**
- **We then remove one card at a time from the table and insert it into the correct position in the left hand.**
- **To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.**

Figure 2.1 Sorting a hand of cards using insertion sort.

Algorithm Example: Insertion Sorting [CLRS-I2A-09]

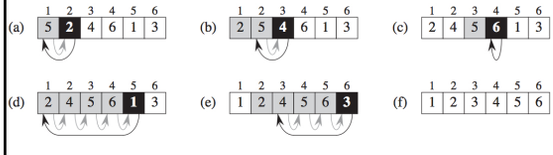


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Algorithm Example: Insertion Sorting [CLRS-I2A-09]

♣ **The insertion sorting algorithm in pseudo-code**

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

A[1], A[2], …, A[i], A[i+1], …, A[length]
A[j] = key (A[i+1] = key)

♣ **Home work**

◆ **Show the flowchart of the intersection sorting algorithm.**

Algorithm Example: Insertion Sorting [CLRS-I2A-09]

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for** loop, which is indexed by j, the subarray consisting of elements $A[1 \,..\, j-1]$ constitutes the currently sorted hand, and the remaining subarray $A[j+1 \,..\, n]$ corresponds to the pile of cards still on the table. In fact, elements $A[1 \,..\, j-1]$ are the elements *originally* in positions 1 through $j-1$, but now in sorted order. We state these properties of $A[1 \,..\, j-1]$ formally as a ***loop invariant***:

> At the start of each iteration of the **for** loop of lines 1–8, the subarray $A[1 \,..\, j-1]$ consists of the elements originally in $A[1 \,..\, j-1]$, but in sorted order.

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Algorithm Example: Insertion Sorting [CLRS-I2A-09]

Let us see how these properties hold for insertion sort.

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.[1] The subarray $A[1 \,..\, j-1]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1 \,..\, j]$ then consists of the elements originally in $A[1 \,..\, j]$, but in sorted order. Incrementing j for the next iteration of the **for** loop then preserves the loop invariant.

A more formal treatment of the second property would require us to state and show a loop invariant for the **while** loop of lines 5–7. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j > A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$, but in sorted order. Observing that the subarray $A[1 \,..\, n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Algorithm Example: Insertion Sorting [CLRS-I2A-09]

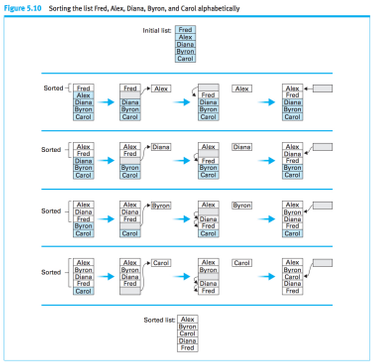
INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time.[6] To compute $T(n)$, the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost* and *times* columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Algorithm Example: Insertion Sorting

Figure 5.10 Sorting the list Fred, Alex, Diana, Byron, and Carol alphabetically



Algorithm Example: Insertion Sorting

Figure 5.11 The insertion sort algorithm expressed in pseudocode

```
procedure Sort (List)
N ← 2;
while (the value of N does not exceed the length of List) do
   (Select the Nth entry in List as the pivot entry;
    Move the pivot entry to a temporary location leaving a hole in List;
    while (there is a name above the hole and that name is greater than the pivot)
       do (move the name above the hole down into the hole
           leaving a hole above the name)
    Move the pivot entry into the hole in List;
    N ← N + 1
   )
```

Algorithm Example: Insertion Sorting [DL-CS-16]

```
Insertion sort
Set current to 1
WHILE (current < length)
    Set index to current
    Set placeFound to FALSE
    WHILE (index > 0 AND NOT placeFound)
        IF (data[index] < data[index – 1])
            Swap data[index] and data[index – 1]
            Set index to index – 1
        ELSE
            Set placeFound to TRUE
    Set current to current + 1
```

Algorithm Example: Insertion Sorting [F-CS-18]

Figure 8.17 *Insertion sort*

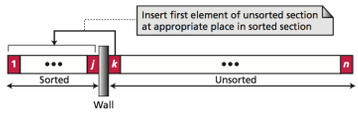
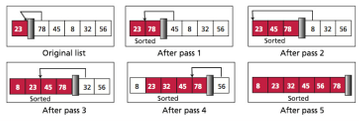


Figure 8.18 *Example of insertion sort*



The Question: How to Design Algorithms ?

How to design algorithms ?

The Art of Problem Solving [DL-CS-16]

- **Problem-solving phases presented by G. Polya in 1945**
  - **Phase 1. Understand the problem.**
    Phase 2. Devise a plan for solving the problem.
    Phase 3. Carry out the plan.
    Phase 4. Evaluate the solution for accuracy and for its potential as a tool for solving other problems.
- **Translate the phases into the context of algorithm/program development**
  - **Phase 1. Understand the problem.**
    Phase 2. Get an idea of how an algorithmic procedure might solve the problem.
    Phase 3. Formulate the algorithm and represent it as a program.
    Phase 4. Evaluate the program for accuracy and for its potential as a tool for solving other problems.

How to Solve Problems by Computing? [DL-CS-16]

- **Analysis and specification process**
  - **The output from the first phase is a clearly written problem statement.**
- **Algorithm development phase**
  - **The output from the algorithm development phase is a plan (algorithm) for a general solution to the problem specified in the first phase.**
- **Implementation phase**
  - **The output from the implementation phase is a working computer program that implements the algorithm, that is, a specific solution to the problem.**
- **Maintenance phase**
  - **There is no output from the maintenance phase, unless errors are detected or changes need to be made. If so, these errors or changes are sent back either to the first phase or second phase, whichever is appropriate.**

How to Solve Problems by Computing? [DL-CS-16]

Analysis and specification phase	
Analyze	Understand (define) the problem.
Specification	Specify the problem that the program is to solve.
Algorithm development phase	
Develop algorithm	Develop a logical sequence of steps to be used to solve the problem.
Test algorithm	Follow the steps as outlined to see if the solution truly solves the problem.
Implementation phase	
Code	Translate the algorithm (the general solution) into a programming language.
Test	Have the computer follow the instructions. Check the results and make corrections until the answers are correct.
Maintenance phase	
Use	Use the program.
Maintain	Modify the program to meet changing requirements or to correct any errors.

FIGURE 7.2 The computer problem-solving process

How to Solve Problems by Computing? [DL-CS-16]

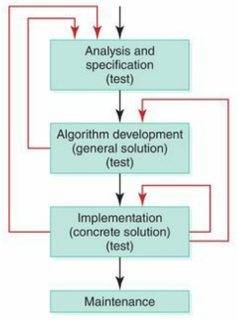
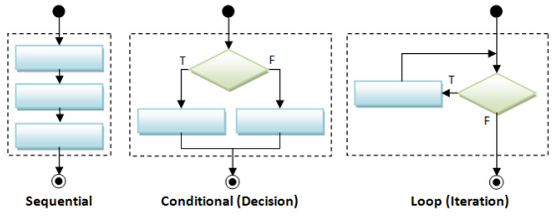


FIGURE 7.3 The interactions among the four problem-solving phases

Three Primary Control Structures in Algorithms

- **Three primary control structures in algorithms**
  - **Sequential execution, Conditional branch (decision), and Loop (iteration).**
  - **Any algorithm/computation can be represented by combination of the three primary control structures.**



Edsger Dijkstra [DL-CS-02,16]

Edsger Dijkstra[a]

Every field of human endeavor has its leading contributors who are acclaimed for their theoretical insights, extensions of fundamental ideas, or innovative changes that have redefined the subject. Just as Beethoven, Schubert, Mozart, and Haydn did during the time in the world of classical music, and the Beatles, Rolling Stones, and the Who stand out in rock 'n' roll, Edsger Dijkstra has a place reserved for him in the computer language hall of fame.

Born to a Dutch chemist in Rotterdam in 1930, Dijkstra grew up with a formalist predilection toward the world. While studying at the University of Leiden in the Netherlands, he attended a summer course on programming in Cambridge, England, and became fascinated with programming. He took a part-time job at the Mathematical Centre in Amsterdam in 1952, and he continued to work there after his graduation. He came to the United States in the early 1970s as a research fellow for Burroughs Corporation, and in September of 1984 he came to The University of Texas at Austin, where he held the Schlumberger Centennial Chair in Computer Sciences. He retired in November of 1999.

One of the Dijkstra's most famous contributions to programming was his strong advocacy of structured programming principles. Dijkstra observed that programs written with goto statements often turned into a rat's nest of jumping back and forth among disorganized, ad hoc sections of programs, making the programs difficult to understand even for the authors—not to mention the colleagues who might later be asked to maintain the program. Dijkstra argued that the goto was not the be-all and end-all of control structures, and he strongly encouraged the use of iterative, or looping, constructs that clearly bracket the scope of branching in a program and effectively self-document the program. Dijkstra claimed that adhering to these structured programming principles would make programs far easier to understand and maintain and less likely to contain errors.

Beyond his clear theoretical contributions, Dijkstra is an interesting character in the computing world. He has developed a reputation for speaking his mind, often in inflammatory or dramatic ways that most of us can't get away with. For example, Dijkstra once remarked that "the use of COBOL cripples the mind; its teaching should therefore be regarded as a criminal offence." Not one to single out only one language for his criticism, he also said that "it is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration." Some people find his message cogent and feel that his manner is politically necessary to make his point. Others, aware of the historical development of languages and the contexts in which they were designed, appreciate his message but find his manner a bit strident.

Besides his work in language design, Dijkstra is also noted for his work in proofs of program correctness. The field of program correctness is an application of mathematics to computer programming. Researchers are trying to construct a language and proof technique that might be used to certify unconditionally that a program will perform according to its specifications—entirely free of bugs. Needless to say, whether your application is customer billing or flight control systems, this claim would be extremely valuable.

In 1972 the Association for Computing Machinery acknowledged Dijkstra's rich contributions to the field by awarding him the distinguished Turing Award. The citation for the award read:

"Edsger Dijkstra was a principal contributor in the late 1950's to the development of the ALGOL, a high level programming language which has become a model of clarity and mathematical rigor. He is one of the principal exponents of the science and art of programming languages in general, and has greatly contributed to our understanding of their structure, representation, and implementation. His fifteen years of publications extend from theoretical articles on graph theory to basic manuals, expository texts, and philosophical contemplations in the field of programming languages."

In 1989 SIGCSE, the Special Interest Group for Computer Science Education, honored him with its award for Outstanding Contributions to Computer Science Education.

In 1989, the Special Interest Group for Computer Science Education (SIGCSE) honored him with its award for Outstanding Contributions to Computer Science Education.

Dijkstra and his wife returned to the Netherlands when he found that he had only months to live. He had always said that he wanted to retire in Austin, Texas, but to die in the Netherlands. Dijkstra died on August 6, 2002.

In March 2003, the following email was sent to the distributed computing community:

> This is to announce that the award formerly known as the "PODC Influential-Paper Award" has been renamed the "Edsger W. Dijkstra Prize in Distributed Computing" after the late Edsger W. Dijkstra, a pioneer in the area of distributed computing. His foundational work on concurrency primitives (such as the semaphore), concurrency problems (such as mutual exclusion and deadlock), reasoning about concurrent systems, and self-stabilization comprises one of the most important supports upon which the field of distributed computing is built. No other individual has had a larger influence on research in principles of distributed computing.

The information on the award can be found at www.podc.org/dijkstra/.

Dijkstra was known for many concise and witty sayings. One that all who study computing should ponder is that "Computer science is no more about computers than astronomy is about telescopes."

Three Primary Constructs in Algorithms [F-CS-18]

Figure 8.6 *Three constructs*

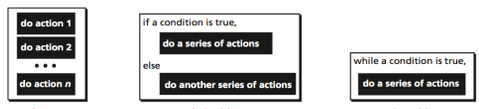
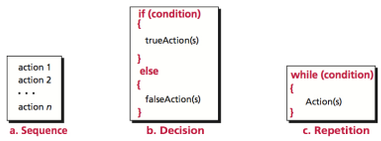


Figure 8.8 *Pseudocode for three constructs*



The While Loop and the Repeat Loop

- **Difference between the while loop and the repeat loop**
  - **The *while loop* (*pre-test loop*) first test the condition and then execute the activity, but the *repeat loop* (*post-test loop*) first execute the activity and then test the condition.**
  - **The while loop may not execute the activity, but the repeat loop must execute the activity at least one time.**

Figure 5.8 The while loop structure

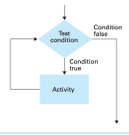
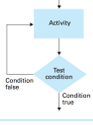
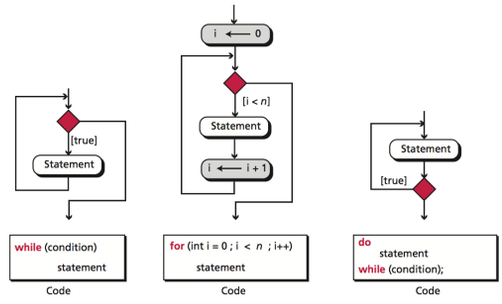


Figure 5.9 The repeat loop structure



Three Types of Repetition [F-CS-18]

Figure 9.10 *Three types of repetition*



The Question: How to Ensure the Correctness of Algorithms ?

How to ensure the correctness of algorithms ?

Algorithm Correctness [CLRS-I2A-09]

- **Correct algorithms**
  - An algorithm is said to be correct if, for every input instance, it halts with the correct output, i.e., for invalid input instance, it halts and reports the invalidity of input, and for valid input instance, it halts and outputs correct result.
  - We say that a correct algorithm solves the given computational problem.
- **Incorrect algorithms**
  - An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect output.
  - Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate.

Algorithm/Program Correctness

- **Correct algorithms/programs**
  - For any valid input, the algorithm/program must halt and output result satisfying the requirement of that algorithm/ program at required time point(s).
  - For any invalid input, the algorithm/program must halt and output an error message to report that input is invalid.
- **Algorithm/program correctness criteria**
  - Halting, giving required output at required time point, rejecting invalid input.

How to Ensure Algorithm/Program Correctness

- **Engineering approaches**
  - Testing, … .
  - All engineering approaches cannot ensure 100% correctness, but can ensure some degree of correctness.
- **Formal methods (formal representation and verification)**
  - Represent a verification object (algorithm/program) and/or its specification into some formal representations (logical formulas), and then verify whether it satisfies the requirement based on a formal logic system.
  - Formal methods can ensure 100% correctness, only if formalization of objects and their requirements are correct.
  - Note: "based on a formal logic system".

Testing vs. Verification (E. W. Dijkstra, 1970)

"Program testing can be used to show the presence of bugs, but never to show their absence!"

-- E. W. Dijkstra, "Notes on Structured Programming," 1970.

Edsger W. Dijkstra

Dijkstra in 2002

Born	11 May 1930 Rotterdam, Netherlands
Died	6 August 2002 (aged 72) Nuenen, Netherlands
Citizenship	Netherlands

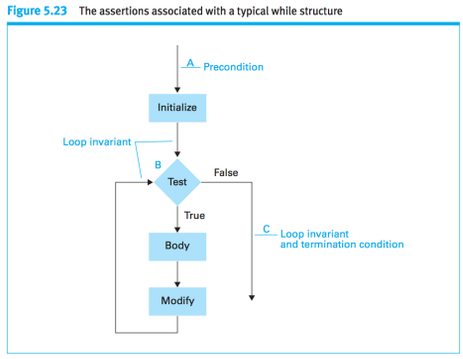
Testing vs. Verification (E. W. Dijkstra, 1972)

- "Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."
- "But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grows hand in hand."

  -- E. W. Dijkstra, "The Humble Programmer," ACM Turing Award Lecture, 1972.

Algorithm/Program Correctness Verification Example

Figure 5.23 The assertions associated with a typical while structure

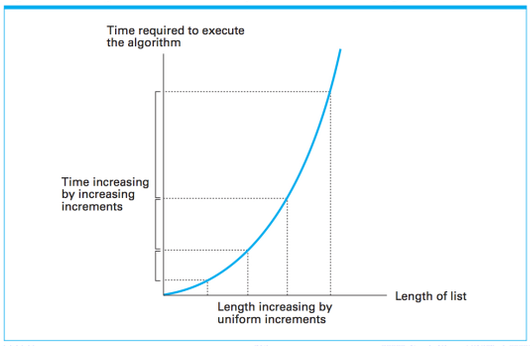


Algorithm/Program Efficiency

- ♣ Why algorithms/program efficiency is important
  - ◆ The most intrinsic power of computers are high-speed and high-performance computing ability.
  - ◆ The choice between efficient and inefficient algorithms can make the difference between a practical solution to a problem and an impractical one.
- ♣ Algorithm/program efficiency criteria
  - ◆ Best-case, average-case, worst-case, typical-case.
- ♣ Algorithm/program efficiency analysis
  - ◆ For given problem and its solution algorithm(s)/program(s), to analyze their efficiency based on various criteria.

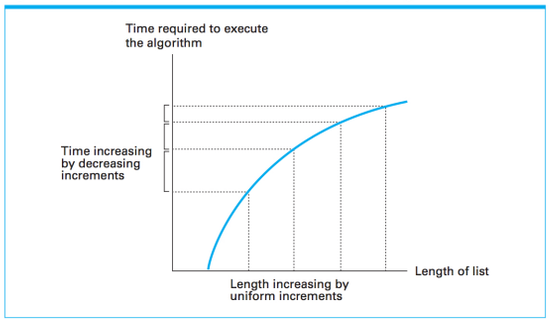
The Parabolic Shape (n^2) vs the Logarithmic Shape (lg$n$)

Figure 5.19 Graph of the worst-case analysis of the insertion sort algorithm



The Parabolic Shape (n^2) vs the Logarithmic Shape (lg$n$)

Figure 5.20 Graph of the worst-case analysis of the binary search algorithm



The Question: How to Verify …… ?

How to verify …… ? Formal methods based on logic

"Logic is the science of sciences, and the art of arts."

"Logic is the science of sciences, and the art of arts."

-- John Duns Scotus, 13th century.

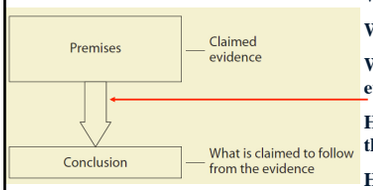
"Nothing can be more important than the art of formal reasoning according to true logic."

-- Gottfried Wilhelm Leibniz

"Logic is the basis for all other sciences"

- "There is a special discipline, called logic, which is considered to be the basis for all other sciences."
  "Logic evolved into an independent science long ago, earlier even than arithmetic and geometry."
  -- A. Tarski, 1941.
- "Mathematical Logic, it is a science prior to all others, which contains the ideas and principles underlying all sciences."
  -- K. Gödel, 1944.

Logic: What Is It?



[From P. J. Hurley, "A Concise Introduction to Logic"]

What entails what?

What follows from what?

Why? What are the evaluation criteria?

How to establish/define the evaluation criteria?

How to evaluate arguments/reasoning?

It is LOGIC to answer these fundamental questions.

Formal Logic Systems

♣ Two indispensable parts of any formal logic system

- A *formal language* (call the '*object language*') for representing things formally.
- A *logical consequence relation* among the formulas of formal language for defining the logical correctness/validity of formal reasoning.

♣ Notes

- Different formal logic systems may have different formal languages; different formal languages have different symbol set.
- Different formal logic systems may have different logical consequence relations.

Formal Logic Systems

♣ Formal logic system

- *Formal logic system* $L =_{df} (F(L), \vdash_L)$ where $\vdash_L =_{df} 2^{F(L)} \rightarrow F(L)$.
- $F(L)$: The *formal language* (call the '*object language*') of L, which is the set of all *well-formed formulas* of L
- $\vdash_L$: The *logical consequence relation* defined among the formulas of $F(L)$, such that for $P \subseteq F(L)$ as the premises and $C \in F(L)$ as a conclusion, $P \vdash_L C$ means that within the framework of L, *C validly follows from P*, or equivalently, *P validly entails C*.

♣ Notes

- '$\vdash_L$' is read as 'the turnstile with subscript L'.
- Both $F(L)$ and $\vdash_L$ have to be defined in detail.

Formal Logic Systems

♣ Logic theorems

- For a formal logic system $L = (F(L), \vdash_L)$, a *logical theorem* t of L is a formula such that $\varnothing \vdash_L t$ where $\varnothing$ is the empty set.
- A logical theorem of a formal logic system validly holds, without premises, in its logic system.
- $Th(L)$: the set of all logical theorems of L.

♣ Notes

- $Th(L)$ is completely determined by $\vdash_L$.
- It is $Th(L)$ that characterizes the logic system L, i.e. if $Th(L) = Th(L')$, then we consider the L and L' to be the same logic system.

Formal Theory

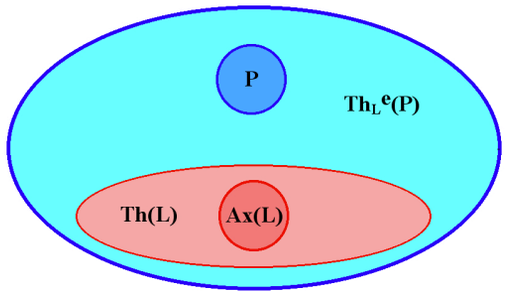
♣ L-theory with premises P

- Let L = (F(L), $\vdash_L$) be a formal logic system and $P \subseteq F$(L) be a non-empty set of *propositions / closed well-formed formulas*.
- A *formal theory* with premises P based on L, called a *L-theory with premises P* and denoted by $T_L(P)$, is defined as

$$T_L(P) =_{df} Th(\mathrm{L}) \cup Th_L^c(P)$$
$$Th_L^c(P) =_{df} \{ et \mid P \vdash_L et \text{ and } et \notin Th(\mathrm{L}) \}$$

- P: The *empirical premises / axioms*.
- Th(L): The *logical part* of the formal theory, any element of Th(L) is called a *logical theorem* of that formal theory.
- $Th_L^c(P)$: The *empirical part* of the formal theory, any element of $Th_L^c(P)$ is called an *empirical theorem* of that formal theory.

Formal Theory



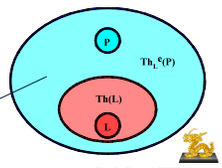
Formal Verification by (Deductive) Theorem Proving

♣ Formal Theorem Proving: What is it?

- For a formal theory $T_L(P)$ with premises P based on a formal logic system (F(L), $\vdash_L$), $P \subseteq F$(L), and a given property $t \in F$(L), find a deduction (proof) of t from P in the formal theory, i.e., find a deduction (proof) for $P \vdash_L t$ or $t \in T_L(P)$.
- The concept is general and applies to all kinds of logics and suitable formal theories.

♣ Theorem proving based on L-theory with premises P

$t \in Th_L^c(P)$?



Formal Verification by (Deductive) Theorem Proving

♣ Formal verification by theorem proving: How do it?

- (1) Select a formal logic system (F(L), $\vdash_L$),
- (2) Determine a set of premises $P \subseteq F$(L) (representation/specifications of a target algorithm/ program),
- (3) Determine a requirement/property $t \in F$(L), the target algorithm/program should satisfy,
- (4) Find a deduction (proof) for $P \vdash_L t$ or $t \in T_L(P)$, usually by an automated theorem prover.

♣ Various automated theorem provers

- ACL2, Coq, HOL, Isabelle, Otter, ……

The Question: How to Ensure the Correctness of Algorithms ?

Because algorithms/programs are logical products, the only correct/effective way to ensure their correctness is the formal verification based on logic!

Ask logic for help!

A Question: What should be Necessary to CS&SE?

- **The appropriate branch of applied mathematics is a necessary part of the education of all (other) engineers.**
- **What should be part of the education of every computer scientist and software engineer?**

The Answer by FAA and NASA to the Question

"Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied mathematics is a necessary part of the education of all other engineers."

-- FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration), USA

The Next Question: How to Automate Computing?

How to automate computing ?

(in the electronic digital way)

An Introduction to Computer Science

- Computer Science: What Is It and Why Study It?
- Computation: What Is It and Why Study It?
- Computability
- Computational Complexity (CS101A class only)
- Algorithms
- Data, Information, and Knowledge, and Their Representations
- Data Storage
- Computer Architecture
- Data Manipulation in Computer Systems
- Programming Languages and Compilers
- Operating Systems
- System Software and Application Software
- Software Engineering (CS101A class only)
- Knowledge Engineering and Artificial Intelligence (CS101A class only)
- Information Security Engineering (CS101A class only)