Introduction to the Theory of Computation

Jingde Cheng

Saitama University

Introduction to the Theory of Computation

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

Computation 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!"

The Central Question of Computational Complexity Theory

What makes some problems computationally hard and others easy?

- Note: At present, we do not know the answer to the question.

Computational Complexity Theory: 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?

♣ 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 Theory: What Is It and Why Study It?

♣ 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.

Computational Complexity Theory

- **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.
- **Computational complexity theory**
  - A theoretical investigation of the time, memory (space), or other resources required for solving computational problems.
- **The fundamental questions**
  - What resources should be measure?
  - How the resources should be measure?

Computational Complexity Theory

- **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.

An Introduction to the Theory of Computation

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

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]

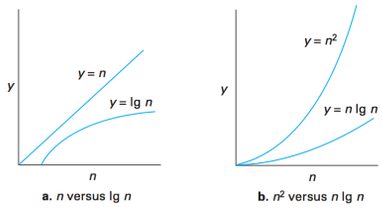
我们来看一个有关“可计算”问题所需演算步骤的数量化比较。我们考虑A,B,C三个“可计算”问题，假设为了算出最终结果A问题需要n步演算、B问题需要n2（n的2次方）步演算、而C问题需要log2n（以2为底n的对数）步演算，并且还假设这些问题在某种现代通用计算机上被实施计算时每个演算步骤花费的时间为0.001秒，那么关于这三个问题之演算步骤及花费时间的一个数量化对比如下：

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。

Bounded Relationship among Mathematical Expressions

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



a. n versus $\lg n$ **b.** n^2 versus $n \lg n$

Measuring Time Complexity: An Example [S-ToC-13]

♣ An example: The language $A = \{0^k1^k \mid k \geq 0\}$ and its TM M_1

- Take the language $A = \{0^k1^k \mid k \geq 0\}$ that is a decidable language.
- How much time does a single-tape TM need to decide A?
- We examine the following single-tape TM M_1 for A.
- 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."

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 [S-ToC-13]

DEFINITION **7.1**

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.

定义 7.1 令 M 是一个在所有输入上都停机的确定型图灵机。M 的运行时间或者时间复杂度，是一个函数 $f: \mathcal{N} \to \mathcal{N}$，其中 $\mathcal{N}$ 是非负整数集合，$f(n)$ 是 M 在所有长度为 n 的输入上运行时所经过的最大步数。若 $f(n)$ 是 M 的运行时间，则称 M 在时间 $f(n)$ 内运行，M 是 $f(n)$ 时间图灵机。通常使用 n 表示输入的长度。

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]

DEFINITION 7.2

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

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

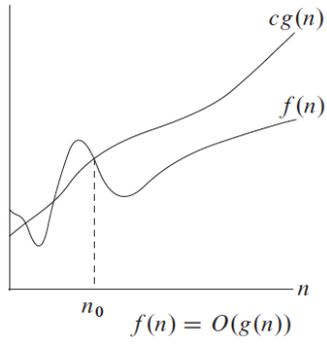
When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

定义 7.2 设 f 和 g 是两个函数 $f, g: \mathcal{N} \to \mathcal{R}^+$。称 $f(n)=O(g(n))$，若存在正整数 c 和 n_0，使得对所有 $n \geqslant n_0$ 有

$$f(n) \leqslant cg(n)$$

当 $f(n)=O(g(n))$ 时，称 $g(n)$ 是 $f(n)$ 的**上界**(upper bound)，或更准确地说，$g(n)$ 是 $f(n)$ 的渐近上界，以强调没有考虑常数因子。

Definition of Asymptotic Upper Bound (Big-*O*) [CLRS-12A-09]



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

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

♣ **Intuitive meaning**

- ***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.**

♣ **In practice**

- **Most functions *f* that you are likely to encounter have an obvious highest order term *h*.**
- **In that case, write *f*(*n*) = *O*(*g*(*n*)), where *g* is *h* without its coefficient.**

Asymptotic Upper Bound (Big-*O*): Examples [S-ToC-13]

♣ **An example**

- **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).**
- **For $n = 6$, $5n^3+2n^2+22n+6 = 1290$, $6n^3 = 1296$.**
- **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.**

Asymptotic Upper Bound (Big-*O*): Examples [S-ToC-13]

♣ **Big-*O* notation with logarithms**

- **Usually when we use logarithms, we must specify the base, as in $x = \log_2 n$. The base 2 here indicates that this equality is equivalent to the equality $2^x = n$.**
- **Changing the value of the base b changes the value of $\log_b n$ by a constant factor, owing to the identity $\log_b n = \log_2 n / \log_2 b$.**
- **Thus, when we write $f(n) = O(\log n)$, specifying the base is no longer necessary because we are suppressing constant factors anyway.**
- **Let $f_2(n)$ be the function $3n\log_2 n + 5n\log_2\log_2 n + 2$. In this case, we have $f_2(n) = O(n\log n)$ because $\log n$ dominates $\log\log n$.**

Asymptotic Upper Bound (Big-*O*): Examples [S-ToC-13]

♣ **Big-*O* notation in arithmetic expressions**

- **Big-*O* notation also appears in arithmetic expressions such as the expression $f(n) = O(n^2)+O(n)$. In that case, each occurrence of the *O* symbol represents a different suppressed constant. Because the $O(n^2)$ term dominates the $O(n)$ term, that expression is equivalent to $f(n) = O(n^2)$.**
- **When the *O* symbol occurs in an exponent, as in the expression $f(n) = 2^{O(n)}$, the same idea applies. This expression represents an upper bound of 2^{cn} for some constant c.**
- **The expression $f(n) = 2^{O(\log n)}$ occurs in some analyses. Using the identity $n = 2\uparrow(\log_2 n)$ and thus $n^c = 2\uparrow(c\log_2 n)$, we see that $2^{O(\log n)}$ represents an upper bound of n^c for some c. The expression $n^{O(1)}$ represents the same bound in a different way because the expression $O(1)$ represents a value that is never more than a fixed constant.**

Polynomial Bound and Exponential Bound [S-ToC-13]

♣ **Polynomial bound**

- **[D] Bounds of the form $f(n) = O(n^c)$ for real number c greater than 0 are called *polynomial bounds*.**

♣ **Exponential bound**

- **[D] 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]

DEFINITION **7.5**

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

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

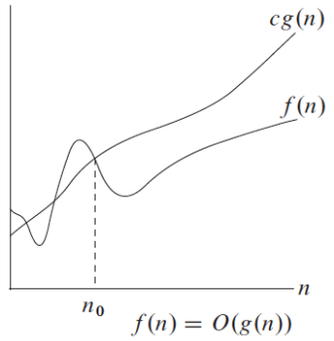
In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

定义 7.5 设 f 和 g 是两个函数 $f, g: \mathcal{N} \to \mathcal{R}^+$。如果

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

则称 $f(n) = o(g(n))$。换言之，$f(n) = o(g(n))$ 意味着对于任何实数 $c > 0$，存在一个数 n_0，使得对所有 $n \geq n_0$，$f(n) < cg(n)$。

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



Big-O Notation vs. Small-o Notation

♣ **Big-O notation**

- **Big-O notation is used to represent that one function is asymptotically no more than another function.**

♣ **Small-o notation**

- **Small-o notation is used to represent that one function is asymptotically less 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 ≤ and < .**

Comparison of Asymptotic Notations [CLRS-I2A-09]

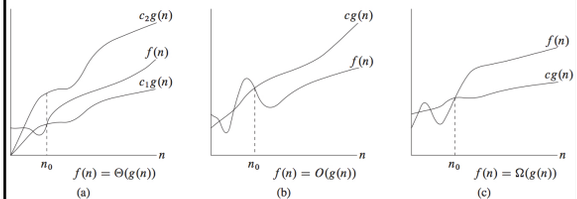


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (a) Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and 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 $cg(n)$.

Comparison of Asymptotic Notations [CLRS-I2A-09]

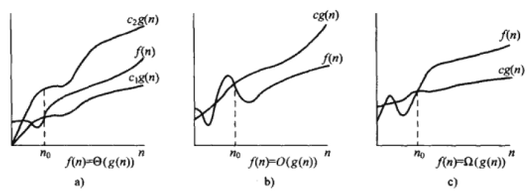


图 3-1 记号 Θ、O 和 Ω 的图例。在每个部分中，n_0 是最小的可能值；大于 n_0 的值也有效。a)Θ 记号限制一个函数在常数因子内。如果存在正常数 n_0，c_1 和 c_2 使得在 n_0 右边 $f(n)$ 的值永远在 $c_1 g(n)$ 与 $c_2 g(n)$ 之间，那么可以写成 $f(n) = \Theta(g(n))$。b)O 记号给出一个函数在常数因子内的上限。如果存在正常数 n_0 和 c 使得在 n_0 右边 $f(n)$ 的值永远等于或小于 $cg(n)$，那么可以写成 $f(n) = O(g(n))$。c)Ω 符号给出一个函数在常数因子内的下限。如果存在正常数 n_0 和 c 使得在 n_0 右边 $f(n)$ 的值永远等于或大于 $cg(n)$，那么可以写成 $f(n) = \Omega(g(n))$

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

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

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

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 $n \ge n_0$, 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 Θ-Notation [CLRS-I2A-09]

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 $n \ge n_0$, 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 Θ-Notation [CLRS-I2A-09]

$\Theta(g(n)) = \{f(n)$：存在正常数 c_1，c_2 和 n_0，使对所有的 $n \ge n_0$，有 $0 \le c_1 g(n) \le f(n) \le c_2 g(n)\}$[2] 对任一个函数 $f(n)$，若存在正常数 c_1，c_2，使当 n 充分大时，$f(n)$ 能被夹在 $c_1 g(n)$ 和 $c_2 g(n)$ 中间，则 $f(n)$ 属于集合 $\Theta(g(n))$。因为 $\Theta(g(n))$ 是一个集合，可以写成"$f(n) \in \Theta(g(n))$"表示 $f(n)$ 是 $\Theta(g(n))$ 的元素。不过，通常写成"$f(n) = \Theta(g(n))$"来表示相同的意思。这种用于表示集合成员关系的等号越规使用初看起来可能有点令人迷惑，但在本节稍候就会看到它的好处了。

图 3-1a 给出函数 $f(n)$ 与 $g(n)$ 的直观图示，其中 $f(n) = \Theta(g(n))$。对所有位于 n_0 右边的 n 值，$f(n)$ 的值落在 $c_1 g(n)$ 和 $c_2 g(n)$ 之间。换句话说，对所有的 $n \ge n_0$，$f(n)$ 在一个常因子范围内与 $g(n)$ 相等。我们说 $g(n)$ 是 $f(n)$ 的一个渐近确界。

$\Theta(g(n))$ 的定义需要每个成员 $f(n) \in \Theta(g(n))$ 都是渐近非负，也就是说当 n 足够大时 $f(n)$ 是非负值（渐近正函数则是当 n 足够大时总为正值）。这就要求函数 $g(n)$ 本身也必须是渐近非负的，否则集合 $\Theta(g(n))$ 就是空集。因此，假定 Θ 记号中用到的每个函数都是渐近非负的。这个假设对本章中定义的其他渐近记号也是成立的。

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

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

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 $c g(n)$.

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

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 O-Notation [CLRS-I2A-09]

为表示一个函数 $f(n)$ 是集合 $O(g(n))$ 的一个元素，记 $f(n) = O(g(n))$。注意 $f(n) = \Theta(g(n))$ 隐含着 $f(n) = O(g(n))$，因为 Θ 记号强于 O 记号。按集合论中的写法，有 $\Theta(g(n)) \subseteq O(g(n))$。我们已经证明任意的二次函数 $an^2 + bn + c$（其中 $a > 0$）属于 $\Theta(n^2)$，这也就证明任意二次函数也属于 $O(n^2)$。更令人惊讶的是任一个线性函数 $an + b$ 也在 $O(n^2)$ 中，这可由设 $c = a + |b|$ 和 $n_0 = 1$ 来证明。

有些式子如 $n = O(n^2)$，对某些曾见过 O 记号的读者来说可能有些奇怪。在文献中，O 记号有时会被非正式地用来描述渐近确界，而这在前面是用 Θ 记号来定义的。在本书中，当我们写 $f(n) = O(g(n))$ 时，只是说明 $g(n)$ 的某个常数倍是 $f(n)$ 的渐近上界，而不反映该上界如何接近。现今有关算法的文献中，都将渐近上界与渐近确界加以区分。

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

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

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 $cg(n)$.

正如 O 记号给出一个函数的渐近上界，Ω 记号给出函数的渐近下界。给定一个函数 $g(n)$，用 $\Omega(g(n))$ 表示一个函数集合

$\Omega(g(n)) = \{f(n)$：存在正常数 c 和 n_0，使对所有的 $n \ge n_0$，有 $0 \le cg(n) \le f(n)\}$

读作 $g(n)$ 的大 Ω。图 3-1c 说明了 Ω 记号的直观意义。对所有在 n_0 右边的 n 值，函数 $f(n)$ 的数值等于或大于 $cg(n)$。

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

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

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for *some* constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ 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 o-Notation [CLRS-I2A-09]

O 记号所提供的渐近上界可能是也可能不是渐近紧确的。界 $2n^2 = O(n^2)$ 是渐近紧确的，但 $2n = O(n^2)$ 却不是的。我们使用 o 记号来表示非渐近紧确的上界。$o(g(n))$ 的形式定义为集合

$o(g(n)) = \{f(n)$：对任意正常数 c，存在常数 $n_0 > 0$，使对所有的 $n \ge n_0$，有 $0 \le f(n) \le cg(n)\}$

例如，$2n = o(n^2)$，但是 $2n^2 \ne o(n^2)$。

O 记号与 o 记号的定义是类似的。主要区别在于对 $f(n) = O(g(n))$，界 $0 \le f(n) \le cg(n)$ 对某个常数 $c > 0$ 成立；但对 $f(n) = o(g(n))$，界 $0 \le f(n) \le cg(n)$ 对所有常数 $c > 0$ 成立。从直觉上看，在 o 表示中当 n 趋于无穷时，函数 $f(n)$ 相对于 $g(n)$ 来说就不重要了，即

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad (3.1)$$

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

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

For example, $n^2/2 = \omega(n)$, but $n^2/2 \ne \omega(n^2)$. 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.

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

ω 记号与 Ω 记号的关系就好像 o 记号与 O 记号的关系一样。我们用 ω 记号来表示非渐近紧确的下界。它的一种定义为

$$f(n) \in \omega(g(n)) \text{ 当且仅当 } g(n) \in o(f(n))$$

$\omega(g(n))$ 的形式定义为集合

$\omega(g(n)) = \{f(n)$：对任意正常数 $c > 0$，存在常数 $n_0 > 0$，使对所有的 $n \ge n_0$，有 $0 \le cg(n) < f(n)\}$。

例如，$n^2/2 = \omega(n)$，但 $n^2/2 \ne \omega(n^2)$。关系 $f(n) = \omega(g(n))$ 意味着

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

如果这个极限存在。也就是说当 n 趋于无穷时，$f(n)$ 相对 $g(n)$ 来说变得任意大了。

Analyzing Complexity of Algorithm: Examples [S-ToC-13]

Let's analyze the TM algorithm we gave for the language $A = \{0^k 1^k \mid k \ge 0\}$. 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*."

Analyzing Complexity of Algorithm: Examples [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.

Analyzing Complexity of Algorithm: Examples [S-ToC-13]

本节分析语言 $A=\{0^k1^k \mid k\geqslant 0\}$ 对应的图灵机算法。为了便于阅读，这里重述一遍此算法。

M_1 ="对输入串 w：

1）扫描带，如果在 1 的右边发现 0，就拒绝。

2）如果带上既有 0 也有 1，就重复下一步。

3） 扫描带，删除一个 0 和一个 1。

4）如果所有 1 都被删除以后还有 0，或者所有 0 都被删除以后还有 1，就拒绝。否则，如果在带上既没有剩下 0 也没有剩下 1，就接受。

为了分析 M_1，把它的四个步骤分开来考虑。步骤 1 中，机器扫描带以验证输入的形式是 0*1*。执行这次扫描需要 n 步。如前面约定的，通常用 n 表示输入的长度。将读写头重新放置在带的左端另外需要 n 步，所以这一步骤总共需要 $2n$ 步。用大 O 记法，称这一阶段需要 $O(n)$ 步。注意，在机器描述中没有提及重新放置读写头的操作。渐近记法允许在机器描述中忽略那些对运行时间的影响不超过常数倍的操作细节。

在步骤 2 和 3 中，机器反复扫描带，在每一次扫描中删除一个 0 和一个 1。每一次扫描需要 $O(n)$ 步。因为每一次扫描删除两个符号，所以最多扫描 $n/2$ 次。于是步骤 2 和 3 需要的全部时间是 $(n/2)O(n)=O(n^2)$ 步。

在步骤 4 中，机器扫描一次来决定是接受还是拒绝。这一步需要的时间最多是 $O(n)$。

所以，M_1 在长度为 n 的输入上总共耗时为 $O(n)+O(n^2)+O(n)$，或 $O(n^2)$。换言之，它的运行时间是 $O(n^2)$。这就完成了对该机器的时间分析。

Analyzing Complexity of Algorithm: Examples [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*."

Analyzing Complexity of Algorithm: Examples [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 $O(n) + O(n \log n) = O(n \log n)$.

Analyzing Complexity of Algorithm: Examples [S-ToC-13]

M_2 ="对输入串 w：

1）扫描带，如果在 1 的右边发现 0，就拒绝。

2）只要在带上还有 0 和 1，就重复下面的步骤。

3） 扫描带，检查剩余的 0 和 1 的总数是偶数还是奇数。若是奇数，就拒绝。

4） 再次扫描带，从第一个 0 开始，隔一个删除一个 0；然后从第一个 1 开始，隔一个删除一个 1。

5）如果带上不再有 0 和 1，就接受。否则，拒绝。"

在分析 M_2 之前，先来验证它的确可以判定 A。在步骤 4 中，每执行一次扫描，剩余的 0 的总数就减少一半，其他的 0 被删除了。因此，如果开始时有 13 个 0，那么在步骤 4 执行一次以后就只剩下 6 个 0。随后每次执行分别剩下 3 个、1 个和 0 个 0。该步骤对 1 的数目有同样的效果。

现在来检查在每次执行步骤 3 时 0 和 1 的数目的奇偶性。再次假定开始时有 13 个 0 和 13 个 1。第一次执行步骤 3 时，有奇数个 0（因为 13 是奇数）和奇数个 1，以后每次执行时分别只剩下偶数个（6 个）、然后是奇数个（3 个）、又是奇数个（1 个）0 和 1。由于步骤 2 指定的循环条件，当剩下 0 个 0 或 0 个 1 时，步骤 3 不再被执行。对于得到的奇偶序列（奇、偶、奇、奇），如果把偶替换成 0，把奇替换成 1，并且反排这个序列，就得到 1101，即 13，这是开始时 0 和 1 的数目的二进制表示。该奇偶序列总是以反排的方式给出二进制表示。

当步骤 3 检查剩下的 0 和 1 的总数是偶数时，实际上是在检查 0 数目的奇偶性与 1 数目的奇偶性是否一致。如果这两个奇偶性始终一致，那么 0 和 1 的数目的二进制表示就一致，从而这两个数目就相等。

为了分析 M_2 的运行时间，首先注意，每一步骤都消耗 $O(n)$ 的时间。然后确定每一步骤需要执行的次数。步骤 1 和 5 执行一次，共需要 $O(n)$ 的时间。步骤 4 在每一次执行时至少删除一半的 0 和 1，所以至多 $1+\log_2 n$ 次循环就可以把全部字符删除。于是步骤 2、3 和 4 总共消耗时间 $(1+\log_2 n)\,O(n)$，即 $O(n\log n)$。M_2 的运行时间是 $O(n)+O(n\log n)=O(n\log n)$。

Analyzing Complexity of Algorithm: Examples [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.

Analyzing Complexity of Algorithm: Examples [S-ToC-13]

如果图灵机有第二条带，就可以在 $O(n)$ 时间(也称为**线性时间**)内判定语言 A。下面的两带 TM M_3 在线性时间内判定 A。机器 M_3 的运转方式和上面那些判定 A 的机器不同。它只是简单地将所有 0 复制到第二条带上，然后拿来和 1 进行匹配。

M_3 ="对输入串 w：

1) 扫描带，如果在 1 的右边发现 0，就**拒绝**。

2) 扫描带 1 上的 0，直到第一个 1 时停止，同时把 0 复制到带 2 上。

3) 扫描带 1 上的 1 直到输入的末尾。每次从带 1 上读到一个 1，就在带 2 上删除一个 0，如果在读完 1 之前所有的 0 都被删除，就**拒绝**。

4) 如果所有的 0 都被删除，就**接受**。如果还有 0 剩下，就**拒绝**。"

这个机器分析起来很简单。四个步骤的每一步明显地需要 $O(n)$ 步，所以全部运行时间是 $O(n)$，因而是线性的。注意，这是可能的最好运行时间，因为光是读输入就需要 n 步。

Analyzing Complexity of Algorithm: Examples [S-ToC-13]

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

- **The single-tape deterministic TM M_1: the total time of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$, i.e., its running time is $O(n^2)$.**
- **The single-tape deterministic TM M_2: the total time of M_2 on an input of length n is $O(n) + O(n\log n) = O(n\log n)$, i.e., its running time is $O(n\log n)$.**
- **The two-tape deterministic TM M_3: the total time of M_3 on an input of length n is $O(n)$ and thus is linear, i.e., its running time is $O(n)$ (the best one).**

♣ **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.**

Time Complexity Classes of Languages/Problems [S-ToC-13]

♣ **Time complexity class**

- **Let t: $N \rightarrow R^+$ be a function where R^+ is the set of non-negative real numbers.**
- **[D] The *time complexity class*, TIME($t(n)$) is the collection of all languages/problems that are decidable by an $O(t(n))$ time (steps) single-tape deterministic TM.**

DEFINITION 7.7

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

♣ **Note**

- **The time complexity class is a class of languages/problems but not TMs.**

Time Complexity Classes of Languages/Problems

♣ **Linear time complexity**

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

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

- **The example M_1 shows $A \in$ TIME(n^2).**
- **The example M_2 shows $A \in$ TIME($n\log n$).**
- **The example M_3 shows $A \in$ TIME(n) [two-tape TM].**
- **Note: A language/problem may belong to different time complexity classes, because the complexity of a language/problem depends on the algorithm/model of computation.**

♣ **An important fact about regular languages**

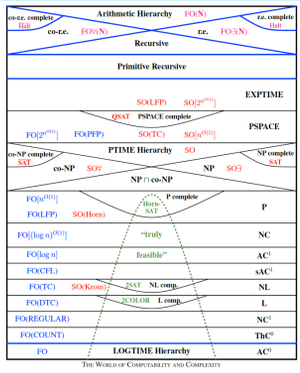
- **Any language that can be decided in $o(n\log n)$ time on a single-tape deterministic TM is regular.**

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 $n \times n$ 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 $\varepsilon > 0$	$O(2^{\log n \log \log n})$	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]

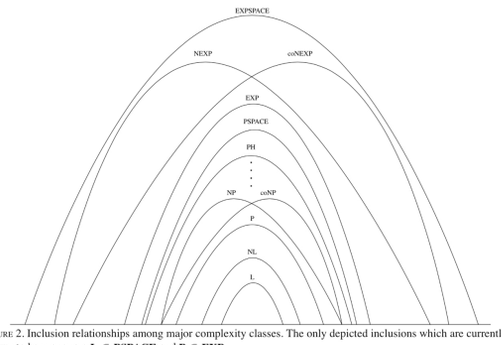


FIGURE 2. Inclusion relationships among major complexity classes. The only depicted inclusions which are currently known to be proper are **L** ⊊ **PSPACE** and **P** ⊊ **EXP**.

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.**

Complexity Relationship Among Models [S-ToC-13]

- **Single-tape TMs vs. multi-tape TMs**

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

PROOF IDEA The idea behind the proof of this theorem is quite simple. Recall that in Theorem 3.13, we showed how to convert any multitape TM into a single-tape TM that simulates it. Now we analyze that simulation to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

Complexity Relationship Among Models [S-ToC-13]

- **The running time of a nondeterministic TM**

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

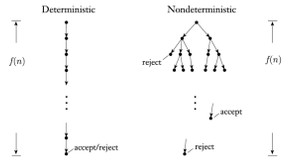


FIGURE 7.10
Measuring deterministic and nondeterministic time

Complexity Relationship Among Models [S-ToC-13]

- **Deterministic single-tape TMs vs. nondeterministic single-tape TMs**

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

- **Notes**
  - **Theorems 7.8 and 7.11 illustrate an important distinction.**
  - **We demonstrated at most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and multi-tape TMs.**
  - **We showed at most an exponential difference between the time complexity of problems on deterministic and nondeterministic TMs.**

Polynomial Time Complexity vs. Exponential Time Complexity [S-ToC-13]

- **Polynomial time complexity**
  - **Polynomial time algorithms are fast enough for many purposes.**
  - **Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large.**
  - **There is the dramatic difference between the growth rate of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n.**
  - **For example, let n be 1000, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe.**

Polynomial Time Complexity vs. Exponential Time Complexity [S-ToC-13]

♣ Exponential time complexity

- Exponential time algorithms rarely are useful.
- [D] Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called *brute-force search*.
- For example, one way to factor a number into its constituent primes is to search through all potential divisors. The size of the search space is exponential, so this search uses exponential time.

Polynomial Equivalence [S-ToC-13]

♣ Polynomial equivalence

- [D] All reasonable deterministic computational models are *polynomially equivalent*. That is, any one of them can simulate another with only a polynomial increase in running time.

♣ Notes

- When we say that all reasonable deterministic models are polynomially equivalent, we do not attempt to define reasonable. However, we have in mind a notion broad enough to include models that closely approximate running times on actual computers.
- We focus on aspects of time complexity theory that are unaffected by polynomial differences in running time. Ignoring these differences allows us to develop a theory that doesn't depend on the selection of a particular model of computation.

The Class P: Problems with Polynomial Time Complexity [S-ToC-13]

♣ The class P: Problems with polynomial time complexity

DEFINITION 7.12

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

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

♣ Time complexity class

- Let t: $N \rightarrow R^+$ be a function where R^+ is the set of non-negative real numbers.
- [D] The *time complexity class*, TIME($t(n)$) is the collection of all languages/problems that are decidable by an $O(t(n))$ time (steps) single-tape deterministic TM.

The Class P: Problems with Polynomial Time Complexity [S-ToC-13]

♣ Why the class P is intrinsically important?

- 1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape TM.
- 2. P roughly corresponds to the class of problems that are realisitically solvable on a computer.

♣ Notes

- The above item 1 indicates that P is a mathematical robust (stable) class.
- The above item 2 indicates that P is relevant (rational) from a practical standpoint.

Algorithms with Polynomial Time Complexity [S-ToC-13]

♣ Analizing algorithms with polynomial time complexity

- When we analyze an algorithm to show that it runs in polynomial time, we need to do two things.
- First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n.
- Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model.
- We choose the stages when we describe the algorithm to make this second part of the analysis easy to do.
- When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial.

Encoding a Graph [S-ToC-13]

♣ Encoding a graph

- One reasonable encoding of a graph is a list of its nodes and edges.
- Another is the *adjacency matrix*, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not.
- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation.
- In reasonable graph representations, the size of the representation is a polynomial in the number of nodes.
- Thus, if we analyze an algorithm and show that its running time is polynomial (or exponential) in the number of nodes, we know that it is polynomial (or exponential) in the size of the input.

Examples of Problems in P [S-ToC-13]

♣ **Finding a path in a directed graph**

- A directed graph G contains nodes s and t. The PATH problem is to determine whether a directed path exists from s to t. Let PATH = { $\langle G, s, t \rangle$ | G is a directed graph that has a directed path from s to t }.
- Theorem: PATH ∈ P.

♣ **A brute-force algorithm for PATH**

- Examining all potential paths in G and determining whether any is a directed path from s to t.
- A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a node never is necessary.) But the number of such potential paths is roughly m^m, which is exponential in the number of nodes in G. Therefore, this brute-force algorithm uses exponential time.

Examples of Problems in P [S-ToC-13]

♣ **A polynomial time algorithm M for PATH**

- M = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:
  1. Place a mark on node s.
  2. Repeat the following until no additional nodes are marked:
  3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
  4. If t is marked, accept. Otherwise, reject."
- Stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus, the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.
- Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for PATH.

Examples of Problems in P [S-ToC-13]

♣ **Testing relatively prime numbers**

- Two numbers are relatively prime if 1 is the largest integer that evenly divides them both. Let RELPRIME be the problem of testing whether two numbers are relatively prime.
- Let RELPRIME = { $\langle x, y \rangle$ | x and y are relatively prime }.
- Theorem: RELPRIME ∈ P.

♣ **A brute-force algorithm for RELPRIME**

- Searching through all possible divisors of both numbers and accepts if none are greater than 1.
- The magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation. Therefore, this brute-force algorithm searches through an exponential number of potential divisors and has an exponential running time.

Examples of Problems in P [S-ToC-13]

♣ **A polynomial time algorithm R for RELPRIME**

- E = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:
  1. Repeat until $y = 0$:
  2. Assign $x := x \bmod y$.
  3. Exchange x and y.
  4. Output x." -- Euclidean algorithm
- R = "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:
  1. Run E on $\langle x, y \rangle$.
  2. If the result is 1, accept. Otherwise, reject ."
- Algorithm R solves RELPRIME, using E as a subroutine.

Examples of Problems in P [S-ToC-13]

♣ **A polynomial time algorithm R for RELPRIME**

- To analyze the time complexity of E, we first show that every execution of stage 2 (except possibly the first) cuts the value of x by at least half. After stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently executed, $x > y$. If $x/2 \geq y$, then $x \bmod y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $x \bmod y = x - y < x/2$ and x drops by at least half.
- The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop. Thus, the maximum number of times that stages 2 and 3 are executed is the lesser of $2\log_2 x$ and $2\log_2 y$. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as $O(n)$. Each stage of E uses only polynomial time, so the total running time is polynomial.

Examples of Problems in P [S-ToC-13]

♣ **Context-free languages**

- Theorem: Every context-free language is a member of P.

PROOF The following algorithm D implements the proof idea. Let G be a CFG in Chomsky normal form generating the CFL L. Assume that S is the start variable. (Recall that the empty string is handled specially in a Chomsky normal form grammar. The algorithm handles the special case in which $w = \varepsilon$ in stage 1.) Comments appear inside double brackets.

D = "On input $w = w_1 \cdots w_n$:

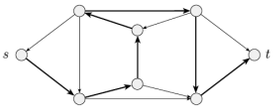
1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. [[$w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \to \mathrm{b}$ is a rule, where $\mathrm{b} = w_i$.
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: [[i is the start position of the substring]]
8. Let $j = i + l - 1$. [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$: [[k is the split position]]
10. For each rule $A \to BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

Now we analyze D. Each stage is easily implemented to run in polynomial time. Stages 4 and 5 run at most nv times, where v is the number of variables in G and is a fixed constant independent of n; hence these stages run $O(n)$ times. Stage 6 runs at most n times. Each time stage 6 runs, stage 7 runs at most n times. Each time stage 7 runs, stages 8 and 9 run at most n times. Each time stage 9 runs, stage 10 runs r times, where r is the number of rules of G and is another fixed constant. Thus stage 11, the inner loop of the algorithm, runs $O(n^3)$ times. Summing the total shows that D executes $O(n^3)$ stages.

Hamiltonian Cycle/Path Problem

♣ Hamiltonian cycle/path problem

- [D] For any given undirected or directed graph G, determine whether a Hamiltonian cycle/path (a cycle/path that visits each vertex exactly once) exists.
- Let HAMPATH = { <G, s, t> | G is a directed graph with a Hamiltonian path from s to t }.



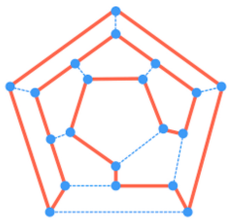


FIGURE 7.17
A Hamiltonian path goes through every node exactly once

Hamiltonian Cycle/Path Problem

♣ Hamiltonian cycle/path problem

- We can easily obtain an exponential time algorithm for the HAMPATH problem by modifying the brute-force algorithm for PATH problem: add a check to verify that the potential path is Hamiltonian. At present, no one knows whether HAMPATH is solvable in polynomial time.
- However, for any given G and path from s to t, to verify whether the path is a Hamiltonian path, is solvable in polynomial time.

♣ An example of may not polynomially verifiable language

- The complement of the HAMPATH problem: HAMPATH^c.
- Even if we could determine (somehow) that a graph did not have a Hamiltonian path, we do not know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place.

A Verifier for a Language [S-ToC-13]

♣ Verifier

- [D] A *verifier* for a language A is an algorithm 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.
- [D] A language A is *polynomially verifiable* if it has a polynomial time verifier.

♣ Note

- A verifier uses additional information, represented by the string 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.

Nondeterministic Polynomial Time Complexity [S-ToC-13]

♣ The class NP

- [D] NP is the class of languages that have polynomial time verifiers.
- Theorem: A language is in NP IFF it is decided by some nondeterministic polynomial time TM.

♣ NP problem examples

- HAMPATH ∈ NP.
- Let COMPOSITES = { x | $x = pq$, for integers $p, q > 1$ }.
- COMPOSITES ∈ NP.

Nondeterministic Polynomial Time Complexity [S-ToC-13]

♣ The class NP

- [D] NP is the class of languages that have polynomial time verifiers.
- Theorem: A language is in NP IFF it is decided by some nondeterministic polynomial time TM.

THEOREM 7.20

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

PROOF IDEA We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate.

Nondeterministic Polynomial Time Complexity [S-ToC-13]

PROOF For the forward direction of this theorem, let $A \in$ NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows.

N = "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, *accept*; otherwise, *reject*."

To prove the other direction of the theorem, assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows.

V = "On input $\langle w, c \rangle$, where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in the proof of Theorem 3.16).
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

Nondeterministic Polynomial Time Complexity [S-ToC-13]

♣ The class NP

DEFINITION **7.21**

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

COROLLARY **7.22**

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

♣ Note

- The class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent.

Examples of Problems in NP [S-ToC-13]

♣ The clique problem

- [D] A *clique* in an undirected graph is a complete subgraph, wherein every two nodes are connected by an edge. A *k-clique* is a clique that contains k nodes.
- The *clique problem* is to determine whether a graph contains a clique of a specified size. Let CLIQUE = { <G, k> | G is an undirected graph with a k–clique }.
- Theorem: CLIQUE ∈ NP.

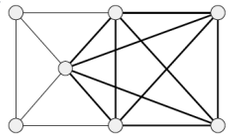


FIGURE **7.23**
A graph with a 5-clique

Examples of Problems in NP [S-ToC-13]

♣ A verifier V for CLIQUE

- V = "On input <<G, k>, c>:
  1. Test whether c is a subgraph with k nodes in G.
  2. Test whether G contains all edges connecting every two nodes in c.
  3. If both pass, accept; otherwise, reject."

♣ A nondeterministic polynomial time TM for CLIQUE

- N = "On input <G, k>, where G is a graph:
  1. Nondeterministically select a subset c of k nodes of G.
  2. Test whether G contains all edges connecting every two nodes in c.
  3. If yes, accept; otherwise, reject."
-

Examples of Problems in NP [S-ToC-13]

♣ The subset-sum problem

- [D] We are given a collection (bag, multiset) of numbers x_1, …, x_k and a target number t. We want to determine whether the collection contains a subcollection that adds up to t.
- Let SUBSET-SUM = {<S, t> | S = {x_1, …, x_k}, and for some {y_1, …, y_l} ⊆ {x_1, …, x_k}, we have $\Sigma y_i = t$ }.
- Theorem: SUBSET-SUM ∈ NP.

Examples of Problems in NP [S-ToC-13]

♣ A verifier V for SUBSET-SUM

- V = "On input <<S, t>, c>:
  1. Test whether c is a collection of numbers that sum to t.
  2. Test whether S contains all the numbers in c.
  3. If both pass, accept; otherwise, reject."

♣ A nondeterministic polynomial time TM for SUBSET-SUM

- N = "On input <S, t>:
  1. Nondeterministically select a subset c of the numbers in S.
  2. Test whether c is a collection of numbers that sum to t.
  3. If yes, accept; otherwise, reject."
-

The Class coNP and the Class EXPTIME [S-ToC-13]

♣ The class coNP

- [D] *coNP* contains the languages that are complements of languages in NP.
- At present, we do not know whether coNP is different from NP.

♣ The class EXPTIME: Problems with EXPonential TIME complexity

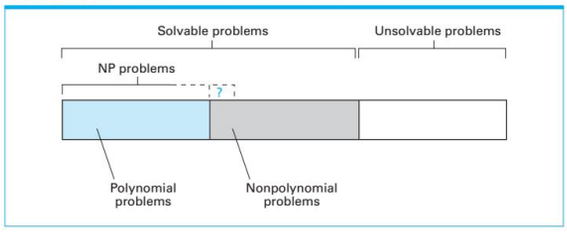
- [D] EXPTIME is the class of languages that are decidable in exponential time on a deterministic single-tape Turing machine.

$$\text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}).$$

- Theorem: NP ⊆ EXPTIME

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]

♣ **Fact**

- **At present, we are unable to prove the existence of a single language in NP that is not in P.**

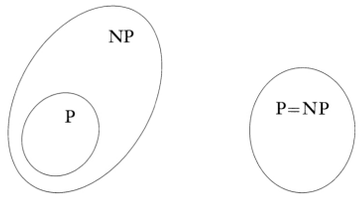


FIGURE **7.26**
One of these two possibilities is correct

NP-Completeness [S-ToC-13]

♣ **NP-completeness**

- **There is a certain set of problems in NP class whose individual complexity is related to that of the entire class.**
- **[D] 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 problem**

- **[D] A logical formula is satisfiable if some truth-value assignment makes it true.**
- **[D] The *satisfiability (SAT) 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 [S-ToC-13]

♣ **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 [S-ToC-13]

♣ **Polynomial time computable function**

- **[D] 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 reduction**

- **[D] 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* ∈ *A* IFF *f*(*w*) ∈ *B*.**
- **[D] The function *f* is called the *polynomial time reduction* of *A* to *B*.**
- **Theorem: If $A \leq_P B$ and *B* ∈ P, then *A* ∈ P.**

Polynomial Time Reducibility [S-ToC-13]

DEFINITION **7.28**

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

DEFINITION **7.29**

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ 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.

Polynomial Time Reducibility [S-ToC-13]

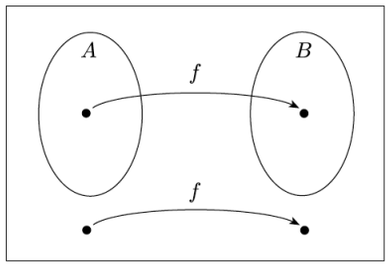


FIGURE **7.30**
Polynomial time function f reducing A to B

Polynomial Time Reducibility [S-ToC-13]

THEOREM **7.31**

If $A \leq_P B$ and $B \in P$, then $A \in P$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

N = "On input w:

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

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

3SAT Problem [S-ToC-13]

♣ **3CNF formulas**

- **[D] A CNF(conjunctive normal form)-formula is a 3CNF-formula if all the clauses have three literals.**
- **Ex:** $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$
- **3SAT = { <*wff*> | *wff* is a satisfiable 3CNF formula }**

♣ **Polynomial time reduction: an example**

- **Theorem: 3SAT is polynomial time reducible to CLIQUE.**

NP-Completeness [S-ToC-13]

♣ **Definition of NP-completeness**

- **[D] 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.**

The Cook-Levin Theorem [S-ToC-13]

♣ **The Cook-Levin theorem (The first NP-complete problem)**

- **SAT is NP-complete.**

PROOF IDEA Showing that *SAT* is in NP is easy, and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to *SAT*.

To do so, we construct a polynomial time reduction for each language A in NP to *SAT*. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore, w is in A if and only if ϕ is satisfiable.

Actually constructing the reduction to work in this way is a conceptually simple task, though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn't surprising. The details are in the implementation of this idea.

NP-Complete Problems [S-ToC-13]

♣ **Corollaries of the Cook-Levin theorem**

- **3SAT is NP-complete.**
- **Proof idea: We can modify the proof of the Cook-Levin so that it directly produces a formula in conjunctive normal form with three literals per clause.**
- **CLIQUE is NP-complete. (because 3SAT is polynomial time reducible to CLIQUE.)**
- **HAMPATH is NP-complete.**
- **SUBSET-SUM is NP-complete.**
- **We can show that 3SAT is polynomial time reducible to HAMPATH and SUBSET-SUM.**

NP-Complete Problems [S-ToC-13]

♣ The graph vertex cover problem

- [D] If G is an undirected graph, a *vertex cover* of G is a subset of the nodes where every edge of G touches one of those nodes.
- [D] The vertex cover problem asks whether a graph contains a vertex cover of a specified size: VERTEX-COVER = { $<G, k>$ | G is an undirected graph that has a k-node vertex cover }.
- Theorem: VERTEX-COVER is NP-complete.

NP-Complete Problems [S-ToC-13]

♣ The long path problem of graph

- Let LPATH = { $<G, a, b, k>$ | G contains a simple path of length at least k from a to b }, where G is an undirected graph and a simple path is a path that does not repeat any nodes.
- LPATH is NP-complete.
- Note: Let SPATH = { $<G, a, b, k>$ | G contains a simple path of length at most k from a to b }. SPATH $\in$ P.

♣ The double SAT problem

- Let DOUBLE-SAT = { $<wff>$ | wff has at least two satisfying assignments }
- DOUBLE-SAT is NP-complete.

NP-Complete Problems [S-ToC-13]

♣ The half clique problem of graph

- Let HALF-CLIQUE = { $<G>$ | G is an undirected graph having a clique (complete subgraph) with at least $m/2$ nodes, where m is the number of nodes in G }.
- HALF-CLIQUE is NP-complete.

♣ The CNF_k SAT problem

- Let CNF_k = { $<wff>$ | wff is a satisfiable CNF-formula where each variable appears in at most k places }.
- CNF_3 is NP-complete.
- Note: $CNF_2 \in P$

NP-Complete Problems [S-ToC-13]

♣ The max cut problem of graph

- A cut in an undirected graph is a separation of the vertices V into two disjoint subsets S and T.
- The size of a cut is the number of edges that have one endpoint in S and the other in T.
- Let MAX-CUT = { $<G, k>$ | G has a cut of size k or more }.
- MAX-CUT is NP-complete.

♣ The graph coloring problem

- A coloring of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color.
- Let 3COLOR = { $<G>$ | G is colorable with 3 colors }.
- 3COLOR is NP-complete.

NP-Complete Problems [S-ToC-13]

♣ The set splitting problem

- Let SET-SPLITTING = { $<S, C>$ | S is a finite set and C = $\{C_1, \ldots, C_k\}$ is a collection of subsets of S, for some $k > 0$, such that elements of S can be colored red or blue so that no C_i has all its elements colored with the same color }.
- SET-SPLITTING is NP-complete.

♣ The dominating set problem

- A subset of the nodes of a graph G is a dominating set if every other node of G is adjacent to some node in the subset.
- Let DOMINATING-SET = { $<G, k>$ | G has a dominating set with k nodes}.
- DOMINATING-SET is NP-complete.

NP-Hard Problems [S-ToC-13]

♣ Definition of NP-hardness

- [D] 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.

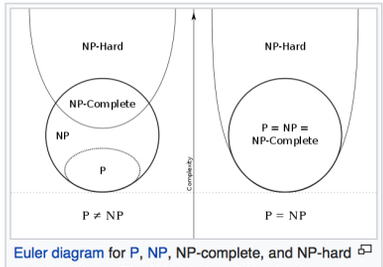
♣ Difference between NP-completeness and NP-Hardness

- 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 has an integral root.
- D = { $<p>$ | p is a polynomial in several variables having an integral root } (Note: 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)

Introduction to the Theory of Computation

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

Measuring Space Complexity [S-ToC-13]

♣ The model for measuring the space used by an algorithm

- We need to select a model of computation for measuring the space used by an algorithm.
- We continue with the Turing machine model for the same reason that we used it to measure time.
- Turing machines are mathematically simple and close enough to real computers to give meaningful results.

♣ Estimating space complexity by using asymptotic notation

- We typically estimate the space complexity of Turing machines by using asymptotic notation.

Definition of Space Complexity [S-ToC-13]

♣ The definition of space complexity

DEFINITION 8.1

Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

对所有输入都能停止的确定性图灵机

Definitions of Space Complexity Classes [S-ToC-13]

♣ The definitions of space complexity class

DEFINITION 8.2

Let $f: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. The ***space complexity classes***, $\mathbf{SPACE}(f(n))$ and $\mathbf{NSPACE}(f(n))$, are defined as follows.

$$\mathrm{SPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$$

$$\mathrm{NSPACE}(f(n)) = \{L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Turing machine}\}.$$

Linear Space Complexity Algorithm: SAT [S-ToC-13]

♣ An example of linear space complexity algorithm: SAT

- SAT can be solved with a linear space algorithm.
- SAT cannot be solved with a polynomial time algorithm, because SAT is NP-complete.
- Space appears to be more powerful than time because space can be reused, whereas time cannot.

SAT可以用线性空间算法解决，但不能用多项式时间算法解决。因为空间可复用但时间不可以。

M_1 = "On input $\langle \phi \rangle$, where ϕ is a Boolean formula:

1. For each truth assignment to the variables $x_1, \ldots, x_m$ of ϕ:
2. Evaluate ϕ on that truth assignment.
3. If ϕ ever evaluated to 1, *accept*; if not, *reject*."

Machine M_1 clearly runs in linear space because each iteration of the loop can reuse the same portion of the tape. The machine needs to store only the current truth assignment, and that can be done with $O(m)$ space. The number of variables m is at most n, the length of the input, so this machine runs in space $O(n)$. ■

验证是否一个非确定性有限自动机能接受所有字符串.

Nondeterministic Linear Space Complexity Algorithm for ALL_{NFA}^{C} [S-ToC-13]

♣ **ALL_{NFA} problem: testing whether a nondeterministic finite automaton accepts all strings**

- Let $ALL_{NFA} = \{ \langle A \rangle \mid A \text{ is an NFA and } L(A) = \Sigma^* \}$.

♣ **A nondeterministic linear space algorithm for ALL_{NFA}^{C}**

- We give a nondeterministic linear space algorithm that decides the complement of this language, ALL_{NFA}^{C}.
- The idea behind this algorithm is to use nondeterminism to guess a string that is rejected by the NFA, and to use linear space to keep track of which states the NFA could be in at a particular time.
- Note: This language is not known to be in the time complexity class NP or in the time complexity class coNP.

Nondeterministic Linear Space Complexity Algorithm for ALL_{NFA}^{C} [S-ToC-13]

N = "On input $\langle M \rangle$, where M is an NFA:

1. Place a marker on the start state of the NFA.
2. Repeat 2^q times, where q is the number of states of M:
3. Nondeterministically select an input symbol and change the positions of the markers on M's states to simulate reading that symbol.
4. *Accept* if stages 2 and 3 reveal some string that M rejects; that is, if at some point none of the markers lie on accept states of M. Otherwise, *reject*."

If M rejects any strings, it must reject one of length at most 2^q because in any longer string that is rejected, the locations of the markers described in the preceding algorithm would repeat. The section of the string between the repetitions can be removed to obtain a shorter rejected string. Hence N decides $\overline{ALL_{NFA}}$. (Note that N accepts improperly formed inputs, too.)

The only space needed by this algorithm is for storing the location of the markers and the repeat loop counter, and that can be done with linear space. Hence the algorithm runs in nondeterministic space $O(n)$. Next, we prove a theorem that provides information about the deterministic space complexity of ALL_{NFA}. ■

萨维奇定理

Savitch's Theorem [S-ToC-13]

♣ **Savitch's theorem**

- Savitch's theorem shows that deterministic machines can simulate nondeterministic machines by using a surprisingly small amount of space.
- For time complexity, such a simulation seems to require an exponential increase in time.
- For space complexity, Savitch's theorem shows that any nondeterministic TM that uses $f(n)$ space can be converted to a deterministic TM that uses only $f^2(n)$ space.

只需平方空间即可将非确定图灵机转成确定图灵机

THEOREM 8.5

Savitch's theorem For any[1] function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

Savitch's Theorem [S-ToC-13]

PROOF IDEA We need to simulate an $f(n)$ space NTM deterministically. A naive approach is to proceed by trying all the branches of the NTM's computation, one by one. The simulation needs to keep track of which branch it is currently trying so that it is able to go on to the next one. But a branch that uses $f(n)$ space may run for $2^{O(f(n))}$ steps and each step may be a nondeterministic choice. Exploring the branches sequentially would require recording all the choices used on a particular branch in order to be able to find the next branch. Therefore, this approach may use $2^{O(f(n))}$ space, exceeding our goal of $O(f^2(n))$ space.

Instead, we take a different approach by considering the following more general problem. We are given two configurations of the NTM, c_1 and c_2, together with a number t, and we test whether the NTM can get from c_1 to c_2 within t steps using only $f(n)$ space. We call this problem the *yieldability problem*. By solving the yieldability problem, where c_1 is the start configuration, c_2 is the accept configuration, and t is the maximum number of steps that the nondeterministic machine can use, we can determine whether the machine accepts its input.

We give a deterministic, recursive algorithm that solves the yieldability problem. It operates by searching for an intermediate configuration c_m, and recursively testing whether (1) c_1 can get to c_m within $t/2$ steps, and (2) whether c_m can get to c_2 within $t/2$ steps. Reusing the space for each of the two recursive tests allows a significant savings of space.

This algorithm needs space for storing the recursion stack. Each level of the recursion uses $O(f(n))$ space to store a configuration. The depth of the recursion is $\log t$, where t is the maximum time that the nondeterministic machine may use on any branch. We have $t = 2^{O(f(n))}$, so $\log t = O(f(n))$. Hence the deterministic simulation uses $O(f^2(n))$ space.

Space Complexity Classes: The Class PSPACE [S-ToC-13]

♣ **The definition of space complexity class PSPACE**

DEFINITION 8.6

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

♣ **Notes**

- We define NPSPACE, the nondeterministic counterpart to PSPACE, in terms of the NSPACE classes.
- However, PSPACE = NPSPACE by virtue of Savitch's theorem because the square of any polynomial is still a polynomial.

The Relationships among the Complexity Classes [S-ToC-13]

♣ **The relationships among the complexity classes**

- P ⊆ NP ⊆ PSPACE=NPSPACE ⊆ EXPTIME

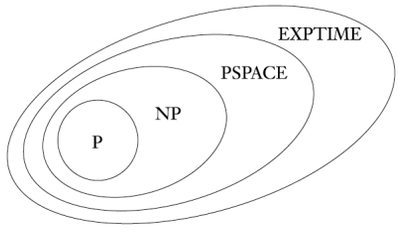


FIGURE 8.7
Conjectured relationships among P, NP, PSPACE, and EXPTIME

这三个⊆中至少有一个真包含关系

The Relationships among the Complexity Classes [S-ToC-13]

♣ The relationships among the complexity classes

- P ⊆ NP ⊆ PSPACE=NPSPACE ⊆ EXPTIME

♣ Facts

- SAT is in SPACE(n). ALL_{NFA} is in coNSPACE(n), and hence, by Savitch's theorem, in SPACE(n^2), because the deterministic space complexity classes are closed under complement. Therefore, both languages are in PSPACE.
- We observe that P ⊆ PSPACE because a machine that runs quickly cannot use a great deal of space. More precisely, for $t(n) \geq n$, any machine that operates in time $t(n)$ can use at most $t(n)$ space because a machine can explore at most one new cell at each step of its computation.
- Similarly, NP ⊆ NPSPACE, and therefore, NP ⊆ PSPACE.

The Relationships among the Complexity Classes [S-ToC-13]

♣ The relationships among the complexity classes

- P ⊆ NP ⊆ PSPACE=NPSPACE ⊆ EXPTIME

♣ Facts

- Conversely, we can bound the time complexity of a TM in terms of its space complexity. For $f(n) \geq n$, a TM that uses $f(n)$ space can have at most $f(n)2^{O(f(n))}$ different configurations. A TM computation that halts may not repeat a configuration. Therefore, a TM that uses space $f(n)$ must run in time $f(n)2^{O(f(n))}$, so PSPACE ⊆ EXPTIME = $\bigcup_k \text{TIME}(2^{n^k})$.
- It is known that P ≠ EXPTIME.
- Therefore, at least one of the preceding containments is proper, but we are unable to say which!

PSPACE完全和PSPACE困难

PSPACE-Complete and PSPACE-Hard [S-ToC-13]

♣ The definition of PSPACE-complete and PSPACE-hard

DEFINITION **8.8**

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

♣ Notes

- We define PSPACE-completeness by polynomial time reducibility.
- Why do not we define a notion of polynomial space reducibility and use that instead of polynomial time reducibility?

用的是多项式时间归约来定义PSPACE完全

On Defining PSPACE-completeness by Polynomial Time Reducibility [S-ToC-13]

- Complete problems are important because they are examples of the most difficult problems in a complexity class. A complete problem is most difficult because any other problem in the class is easily reduced into it.
- So if we find an easy way to solve the complete problem, we can easily solve all other problems in the class. The reduction must be easy, relative to the complexity of typical problems in the class, for this reasoning to apply. If the reduction itself were difficult to compute, an easy solution to the complete problem would not necessarily yield an easy solution to the problems reducing to it.
- Therefore, the rule is: Whenever we define complete problems for a complexity class, the reduction model must be more limited than the model used for defining the class itself.

确定一个满量化布尔式子是真或假

PSPACE-Complete Problems [S-ToC-13]

♣ The TQBF problem: to determine whether a fully quantified Boolean formula is true or false

- TQBF = { <*qbf*> | *qbf* is a true fully quantified Boolean formula }.
- Theorem: TQBF is PSPACE-complete.

♣ A polynomial space algorithm deciding TQBF

- T = "On input <*qbf*>, a fully quantified Boolean formula:
  1. If *qbf* contains no quantifiers, then it is an expression with only constants, so evaluate *qbf* and accept if it is true; otherwise, reject.
  2. If *qbf* equals ∃*x A*, recursively call T on *A*, first with 0 substituted for *x* and then with 1 substituted for *x*. If either result is accept, then accept; otherwise, reject.
  3. If *qbf* equals ∀*x A*, recursively call T on *A*, first with 0 substituted for *x* and then with 1 substituted for *x*. If both results are accept, then accept; otherwise, reject."

PSPACE-Complete Problems [S-ToC-13]

♣ Formula games

- Let $\phi = \exists x_1 \forall x_2 \exists x_3 \ldots Q x_k [\psi]$ be a quantified Boolean formula in prenex normal form. Here, Q represents either a ∀ or an ∃ quantifier. We associate a game with formula ϕ as follows.
- Two players, called Player A and Player E, take turns selecting the values of the variables $x_1, \ldots, x_k$. Player A selects values for the variables that are bound to ∀ quantifiers, and Player E selects values for the variables that are bound to ∃ quantifiers. The order of play is the same as that of the quantifiers at the beginning of the formula.
- At the end of play, we use the values that the players have selected for the variables and declare that Player E has won the game if formula ψ is now TRUE. Player A has won if ψ is now FALSE.

前束范式

PSPACE-Complete Problems [S-ToC-13]

♣ **Formula games: An example**

- Let $\phi_1 = \exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)]$
- In the formula game for ϕ_1, Player E picks the value of x_1, then Player A picks the value of x_2, and finally Player E picks the value of x_3.
- Let's say that Player E picks x_1=1 (TRUE), then Player A picks x_2=0 (FALSE), and finally Player E picks x_3=1. With these values for x_1, x_2, and x_3, the value of subformula $(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is 1, so Player E has won the game.
- In fact, Player E may always win this game by selecting x_1=1 and then selecting x_3 to be the negation of whatever Player A selects for x_2. We say that Player E has a winning strategy for this game.

♣ **Winning strategy**

- [D] A player has a *winning strategy* for a game if that player wins when both sides play optimally.

PSPACE-Complete Problems [S-ToC-13]

♣ **Formula games: Another example**

- Let $\phi_2 = \exists x_1 \forall x_2 \exists x_3 [(x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)]$
- Player A now has a winning strategy because no matter what Player E selects for x_1, Player A may select x_2=0, thereby falsifying the part of the formula appearing after the quantifiers, whatever Player E's last move may be.

♣ **The problem of determining which player has a winning strategy in a formula game**

- [D] A player has a *winning strategy* for a game if that player wins when both sides play optimally.
- Let FORMULA-GAME = { <ϕ> | Player E has a winning strategy in the formula game associated with ϕ }.
- Theorem: FORMULA-GAME is PSPACE-Complete.

是否有必胜策略

PSPACE-Complete Problems [S-ToC-13]

♣ **Generalized geography**

- In generalized geography, we take an arbitrary directed graph with a designated start node. One player starts by selecting the designated start node and then the players take turns picking nodes that form a simple (i.e., does not use any node more than once) directed path in the graph. The first player unable to extend the path loses the game.

♣ **Generalized geography: An example**

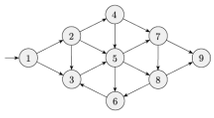


FIGURE 8.13
A sample generalized geography game

第一个走到重复节点的玩家会输

PSPACE-Complete Problems [S-ToC-13]

♣ **Generalized geography: An example**

- Player I is the one who moves first and Player II second.
- Player I has a winning strategy as follows.
- Player I starts at node 1, the designated start node. Node 1 points only at nodes 2 and 3, so Player I's first move must be one of these two choices. He chooses 3. Now Player II must move, but node 3 points only to node 5, so she is forced to select node 5. Then Player I selects 6, from choices 6, 7, and 8. Now Player II must play from node 6, but it points only to node 3, and 3 was previously played. Player II is stuck and thus Player I wins.

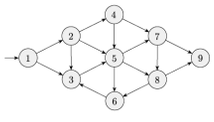


FIGURE 8.13
A sample generalized geography game

PSPACE-Complete Problems [S-ToC-13]

♣ **Generalized geography: An example**

- If we change the example by reversing the direction of the edge between nodes 3 and 6, Player II has a winning strategy.
- If Player I starts out with node 3 as before, Player II responds with 6 and wins immediately, so Player I's only hope is to begin with 2. In that case, however, Player II responds with 4. If Player I now takes 5, Player II wins with 6. If Player I takes 7, Player II wins with 9. No matter what Player I does, Player II can find a way to win, so Player II has a winning strategy.

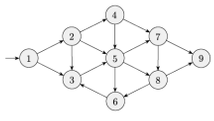


FIGURE 8.13
A sample generalized geography game

PSPACE-Complete Problems [S-ToC-13]

♣ **The problem of determining which player has a winning strategy in a generalized geography game**

- Let GG = { <*G*, *b*> | Player I has a winning strategy for the generalized geography game played on graph *G* starting at node *b* }.
- Theorem: GG is PSPACE-Complete.
- Note: The theorem showed that no polynomial time algorithm exists for optimal play in generalized geography unless P = PSPACE.

PSPACE-Complete Problems [S-ToC-13]

♣ A polynomial space complexity algorithm deciding GG

- M = "On input <*G*, *b*>, where *G* is a directed graph and *b* is a node of *G*:
  1. If *b* has outdegree 0, reject because Player I loses immediately.
  2. Remove node *b* and all connected arrows to get a new graph *G'*.
  3. For each of the nodes $b_1, b_2, \ldots, b_k$ that *b* originally pointed at, recursively call M on <*G'*, b_i>.
  4. If all of these accept, Player II has a winning strategy in the original game, so reject. Otherwise, Player II does not have a winning strategy, so Player I must; therefore, accept."

PSPACE-Complete Problems [S-ToC-13]

♣ Linear bounded automaton problem

- Let A_{LBA} = { <*M*, *w*> | *M* is an LBA that accepts input *w* }.
- Theorem: A_{LBA} is PSPACE-complete.

线性有界自动机M能接受输入w.

Sublinear Space Complexity Bounds [S-ToC-13]

♣ Sublinear space complexity bounds 亚线性空间复杂度界

- In sublinear space complexity, the machine is able to read the entire input but it does not have enough space to store the input.
- To consider this situation meaningfully, we must modify our computational model.
- Note: In time complexity, sublinear bounds are insufficient for reading the entire input, so we do not consider them here.

Sublinear Space Complexity Bounds [S-ToC-13]

♣ TMs with a read-only input tape and a read/write work tape

- On the read-only tape, the input head can detect symbols but not change them.
- We provide a way for the machine to detect when the head is at the left-hand and right-hand ends of the input.
- The input head must remain on the portion of the tape containing the input.
- The work tape may be read and written in the usual way.
- Only the cells scanned on the work tape contribute to the space complexity of this type of Turing machine.

The Complexity Classes L and NL [S-ToC-13]

♣ The definitions of complexity classes L and NL

DEFINITION 8.17

L is the class of languages that are decidable in logarithmic space on a deterministic Turing machine. In other words,

$$\text{L} = \text{SPACE}(\log n).$$

NL is the class of languages that are decidable in logarithmic space on a nondeterministic Turing machine. In other words,

$$\text{NL} = \text{NSPACE}(\log n).$$

- Note: For space bounds that are at least linear, the two-tape TM model is equivalent to the standard one-tape TM model. For sublinear space bounds, we use only the two-tape model.

The Complexity Classes L and NL [S-ToC-13]

♣ An example in class L

- The language A = { 0^k1^k | k ≥ 0} is a member of L.
- We know a TM that decides A by zig-zagging back and forth across the input, crossing off the 0s and 1s as they are matched.
- That algorithm uses linear space to record which positions have been crossed off, but it can be modified to use only log space.

♣ An example in class NL

- The language PATH = { <*G*, *s*, *t*> | *G* is a directed graph that has a directed path from s to t } is a member of NL. (Note: We do not know whether PATH is a member of L.)
- The nondeterministic log space TM deciding PATH operates by starting at node *s* and nondeterministically guessing the nodes of a path from *s* to *t*. The machine records only the position of the current node at each step on the work tape, not the entire path (which would exceed the logarithmic space requirement).

NL-Completeness [S-ToC-13]

♣ L = NL?

- In fact, we do not know of any problem in NL that can be proven to be outside L. Analogous to the question of whether P = NP, we have the question of whether L = NL.
- As a step toward resolving the L versus NL question, we can exhibit certain languages that are NL-complete.
- As with complete languages for other complexity classes, the NL-complete languages are examples of languages that are, in a certain sense, the most difficult languages in NL.
- If L and NL are different, all NL-complete languages do not belong to L.

若L与NL不同，则NL完全语言不属于L

NL-Completeness [S-ToC-13]

♣ Defining NL-completeness

- As with our previous definitions of completeness, we define an NL–complete language to be one that is in NL and to which any other language in NL is reducible.
- However, we do not use polynomial time reducibility here because all problems in NL are solvable in polynomial time.
- Therefore, every two problems in NL except $\varnothing$ and Σ^* are polynomial time reducible to one another.
- Hence polynomial time reducibility is too strong to differentiate problems in NL from one another.
- Instead we use a new type of reducibility called log space reducibility.

多项式时间归约 对NP太强了

使用log空间归约

NL-Completeness [S-ToC-13]

♣ Log space transducer and log space reducibility

DEFINITION 8.21

A ***log space transducer*** is a Turing machine with a read-only input tape, a write-only output tape, and a read/write work tape. The head on the output tape cannot move leftward, so it cannot read what it has written. The work tape may contain $O(\log n)$ symbols. A log space transducer M computes a function $f: \Sigma^* \longrightarrow \Sigma^*$, where $f(w)$ is the string remaining on the output tape after M halts when it is started with w on its input tape. We call f a ***log space computable function***. Language A is ***log space reducible*** to language B, written $A \leq_L B$, if A is mapping reducible to B by means of a log space computable function f.

- Note: A log space transducer is a special TM with 3 types.

有三条带：输入、输出、处理

NL-Completeness [S-ToC-13]

♣ NL-Complete languages

DEFINITION 8.22

A language B is ***NL-complete*** if

1. $B \in$ NL, and
2. every A in NL is log space reducible to B.

THEOREM 8.23

If $A \leq_L B$ and $B \in$ L, then $A \in$ L.

COROLLARY 8.24

If any NL-complete language is in L, then L = NL.

NL-Complete Problems [S-ToC-13]

♣ PATH problem is NL-complete

- Theorem: PATH is NL-complete.
- Corollary: NL ⊆ P.
- Proof: The above theorem shows that any language in NL is log space reducible to PATH. Recall that a TM that uses space $f(n)$ runs in time $n2^{O(f(n))}$, so a reducer that runs in log space also runs in polynomial time. Therefore, any language in NL is polynomial time reducible to PATH, which in turn is in P. We know that every language that is polynomial time reducible to a language in P is also in P, so the proof is complete.

NL-Complete Problems [S-ToC-13]

♣ Strongly connected graph problem

- A directed graph is strongly connected if every two nodes are connected by a directed path in each direction.
- Let STRONGLY-CONNECTED = { <*G*> | *G* is a strongly connected graph }.
- Theorem: STRONGLY-CONNECTED is NL-complete.

♣ Directed cycle problem

- Let CYCL = { <*G*> | *G* is a directed graph that contains a directed cycle }.
- Theorem: CYCL is NL-complete.

NL-Complete Problems [S-ToC-13]

♣ $BOTH_{NFA}$ problem

- Let $BOTH_{NFA}$ = { $<M_1, M_2>$ | M_1 and M_2 are NFAs where $L(M_1) \cap L(M_2) \neq \emptyset$ }.
- Theorem: $BOTH_{NFA}$ is NL-complete.

♣ The acceptance problem for NFAs

- $A_{NFA} =_{df}$ { $<B, w>$ | B is a NFA that accepts input string w }
- Theorem: A_{NFA} is NL-complete.

♣ The emptiness testing problem for DFAs

- $E_{DFA} =_{df}$ { $<A>$ | A is a DFA and $L(A) = \emptyset$ }
- Theorem: E_{DFA} is NL-complete.

NL-Complete Problems [S-ToC-13]

♣ CNF_{H1} problem

- Let CNF_{H1} = { $<cnf>$ | *cnf* is a a satisfiable CNF-formula where each clause contains any number of positive literals and at most one negated literal. Furthermore, each negated literal has at most one occurrence in *cnf* }.
- Theorem: CNF_{H1} is NL-complete.

♣ 2SAT problem

- A 2CNF-formula is an AND of clauses, where each clause is an OR of at most two literals.
- Let 2SAT = { $<2\text{-}cnf>$ | 2-*cnf* is a satisfiable 2CNF-formula }.
- Theorem: 2SAT is NL-complete.

NL and coNL [S-ToC-13]

♣ The class coNL

- [D] *coNL* contains the languages that are complements of languages in NL.

♣ NL = coNL !

- Theorem: NL = coNL.
- This is one of the most surprising results known concerning the relationships among complexity classes.
- The classes NP and coNP are generally believed to be different.

♣ The relationships among the complexity classes

- L ⊆ NL=coNL ⊆ P ⊆ NP ⊆ PSPACE=NPSPACE ⊆ EXPTIME

Introduction to the Theory of Computation

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

Intractability [S-ToC-13]

♣ Intractable problems

- [D] Certain computational problems are solvable in principle, but the solutions require so much time or space that they can't be used in practice. Such problems are called *intractable*.

♣ Some intractable problems

- Most people believe the SAT problem and all other NP-complete problems are intractable, although we do not know how to prove that they are.

Hierarchy Theorems: Space Constructible Functions [S-ToC-13]

♣ Space constructible functions

DEFINITION **9.1**

A function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is at least $O(\log n)$, is called ***space constructible*** if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$.[1]

- Note: 1^n means a string of n 1s.
- *f* is space constructible if some $O(f(n))$ space TM exists that always halts with the binary representation of $f(n)$ on its tape when started on input 1^n.

♣ Space constructible function examples

- All commonly occurring functions that are at least $O(\log n)$ are space constructible, including the functions $\log_2 n$, $n\log_2 n$, and n^2.

Hierarchy Theorems: Space Hierarchy Theorem [S-ToC-13]

♣ **Space hierarchy theorem**

THEOREM 9.3

Space hierarchy theorem For any space constructible function $f: \mathcal{N} \longrightarrow \mathcal{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

♣ **The role of space constructibility in the space hierarchy theorem**

- **If *f*(*n*) and *g*(*n*) are two space bounds, where *f*(*n*) is asymptotically larger than *g*(*n*), we would expect a machine to be able to decide more languages in *f*(*n*) space than in *g*(*n*) space.**
- **However, suppose that *f*(*n*) exceeds *g*(*n*) by only a very small and hard to compute amount. Then, the machine may not be able to use the extra space profitably because even computing the amount of extra space may require more space than is available. In this case, a machine may not be able to compute more languages in *f*(*n*) space than it can in *g*(*n*) space.**
- **Stipulating that *f*(*n*) is space constructible avoids this situation and allows us to prove that a machine can compute more than it would be able to in any asymptotically smaller bound.**

Hierarchy Theorems: Space Hierarchy Theorem [S-ToC-13]

♣ **Space hierarchy theorem**

THEOREM 9.3

Space hierarchy theorem For any space constructible function $f: \mathcal{N} \longrightarrow \mathcal{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.

PROOF The following $O(f(n))$ space algorithm D decides a language A that is not decidable in $o(f(n))$ space.

D = "On input w:

1. Let n be the length of w.
2. Compute $f(n)$ using space constructibility and mark off this much tape. If later stages ever attempt to use more, *reject*.
3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, *reject*.
4. Simulate M on w while counting the number of steps used in the simulation. If the count ever exceeds $2^{f(n)}$, *reject*.
5. If M accepts, *reject*. If M rejects, *accept*."

Hierarchy Theorems: Space Hierarchy Theorem [S-ToC-13]

♣ **Corollaries of the space hierarchy theorem**

COROLLARY 9.4

For any two functions $f_1, f_2: \mathcal{N} \longrightarrow \mathcal{N}$, where $f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible, $\mathrm{SPACE}(f_1(n)) \subsetneq \mathrm{SPACE}(f_2(n))$.[2]

COROLLARY 9.5

For any two real numbers $0 \leq \epsilon_1 < \epsilon_2$,

$$\mathrm{SPACE}(n^{\epsilon_1}) \subsetneq \mathrm{SPACE}(n^{\epsilon_2}).$$

COROLLARY 9.6

$\mathrm{NL} \subsetneq \mathrm{PSPACE}$.

PROOF Savitch's theorem shows that $\mathrm{NL} \subseteq \mathrm{SPACE}(\log^2 n)$, and the space hierarchy theorem shows that $\mathrm{SPACE}(\log^2 n) \subsetneq \mathrm{SPACE}(n)$. Hence the corollary follows.

COROLLARY 9.7

$\mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE}$.

Hierarchy Theorems: Time Constructible Functions [S-ToC-13]

♣ **Time constructible functions**

DEFINITION 9.8

A function $t: \mathcal{N} \longrightarrow \mathcal{N}$, where $t(n)$ is at least $O(n \log n)$, is called ***time constructible*** if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.

- **Note: 1^n means a string of n 1s.**
- ***t* is space constructible if some *O*(*t*(*n*)) space TM exists that always halts with the binary representation of *t*(*n*) on its tape when started on input 1^n.**

♣ **Time constructible function examples**

- **All commonly occurring functions that are at least *n*log*n* are time constructible, including the functions *n*log*n*, $nn^{1/2}$, n^2, 2^n.**

Hierarchy Theorems: Time Hierarchy Theorem [S-ToC-13]

♣ **Time hierarchy theorem**

THEOREM 9.10

Time hierarchy theorem For any time constructible function $t: \mathcal{N} \longrightarrow \mathcal{N}$, a language A exists that is decidable in $O(t(n))$ time but not decidable in time $o(t(n)/\log t(n))$.

♣ **The space hierarchy theorem vs. the time hierarchy theorem**

- **For technical reasons, the time hierarchy theorem is slightly weaker than the one we proved for space.**
- **Whereas any space constructible asymptotic increase in the space bound enlarges the class of languages decidable therein, for time we must further increase the time bound by a logarithmic factor in order to guarantee that we can obtain additional languages.**
- **Conceivably, a tighter time hierarchy theorem is true; but at present, we do not know how to prove it. This aspect of the time hierarchy theorem arises because we measure time complexity with single-tape TMs.**

Hierarchy Theorems: Time Hierarchy Theorem [S-ToC-13]

♣ **Time hierarchy theorem**

THEOREM 9.10

Time hierarchy theorem For any time constructible function $t: \mathcal{N} \longrightarrow \mathcal{N}$, a language A exists that is decidable in $O(t(n))$ time but not decidable in time $o(t(n)/\log t(n))$.

PROOF The following $O(t(n))$ time algorithm D decides a language A that is not decidable in $o(t(n)/\log t(n))$ time.

D = "On input w:

1. Let n be the length of w.
2. Compute $t(n)$ using time constructibility and store the value $\lceil t(n)/\log t(n) \rceil$ in a binary counter. Decrement this counter before each step used to carry out stages 4 and 5. If the counter ever hits 0, *reject*.
3. If w is not of the form $\langle M \rangle 10^*$ for some TM M, *reject*.
4. Simulate M on w.
5. If M accepts, then *reject*. If M rejects, then *accept*."

Hierarchy Theorems: Time Hierarchy Theorem [S-ToC-13]

♣ **Corollaries of the time hierarchy theorem**

THEOREM **9.10**

Time hierarchy theorem For any time constructible function $t: \mathcal{N} \longrightarrow \mathcal{N}$, a language A exists that is decidable in $O(t(n))$ time but not decidable in time $o(t(n)/\log t(n))$.

COROLLARY **9.11**

For any two functions $t_1, t_2: \mathcal{N} \longrightarrow \mathcal{N}$, where $t_1(n)$ is $o(t_2(n)/\log t_2(n))$ and t_2 is time constructible, $\text{TIME}(t_1(n)) \subsetneq \text{TIME}(t_2(n))$.

COROLLARY **9.12**

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$, we have $\text{TIME}(n^{\epsilon_1}) \subsetneq \text{TIME}(n^{\epsilon_2})$.

COROLLARY **9.13**

$P \subsetneq \text{EXPTIME}$.

Exponential Space Completeness [S-ToC-13]

♣ **Generalized regular expressions**

- **[D] Let ↑ be the exponentiation operation. If R is a regular expression and k is a non-negative integer, writing $R{\uparrow}k$ is equivalent to the concatenation of R with itself k times: $R{\uparrow}k = R^k = R \bullet R \bullet \ldots \bullet R$ (k times)**
- **[D] *Generalized regular expressions* allow the exponentiation operation in addition to the usual regular operations.**
- **Obviously, these generalized regular expressions still generate the same class of regular languages as do the standard regular expressions because we can eliminate the exponentiation operation by repeating the base expression.**

♣ **Generalized regular expression equivalence problem**

- **Let $EQ_{REX\uparrow} = \{ \langle Q, R \rangle \mid Q$ and R are equivalent regular expressions with exponentiation $\}$.**

Exponential Space Completeness [S-ToC-13]

♣ **EXPSPACE-completeness**

DEFINITION **9.14**

A language B is ***EXPSPACE-complete*** if

1. $B \in$ EXPSPACE, and
2. every A in EXPSPACE is polynomial time reducible to B.

♣ **EXPSPACE-complete languages: An example**

- **Let $EQ_{REX\uparrow} = \{ \langle Q, R \rangle \mid Q$ and R are equivalent regular expressions with exponentiation $\}$.**

THEOREM **9.15**

$EQ_{\text{REX}\uparrow}$ is EXPSPACE-complete.

Exponential Space Completeness [S-ToC-13]

♣ **EXPSPACE-complete languages: An example**

- **Theorem: $EQ_{REX\uparrow}$ is EXPSPACE-complete.**

PROOF First, we present a nondeterministic algorithm for testing whether two NFAs are inequivalent.

N = "On input $\langle N_1, N_2 \rangle$, where N_1 and N_2 are NFAs:

1. Place a marker on each of the start states of N_1 and N_2.
2. Repeat $2^{q_1+q_2}$ times, where q_1 and q_2 are the numbers of states in N_1 and N_2:
3. Nondeterministically select an input symbol and change the positions of the markers on the states of N_1 and N_2 to simulate reading that symbol.
4. If at any point a marker was placed on an accept state of one of the finite automata and not on any accept state of the other finite automaton, *accept*. Otherwise, *reject*."

Exponential Space Completeness [S-ToC-13]

♣ **EXPSPACE-complete languages: An example**

- **Theorem: $EQ_{REX\uparrow}$ is EXPSPACE-complete.**

Algorithm N runs in nondeterministic linear space. Thus, Savitch's theorem provides a deterministic $O(n^2)$ space algorithm for this problem. Next, we use the deterministic form of this algorithm to design the following algorithm E that decides $EQ_{\text{REX}\uparrow}$.

E = "On input $\langle R_1, R_2 \rangle$, where R_1 and R_2 are regular expressions with exponentiation:

1. Convert R_1 and R_2 to equivalent regular expressions B_1 and B_2 that use repetition instead of exponentiation.
2. Convert B_1 and B_2 to equivalent NFAs N_1 and N_2, using the conversion procedure given in the proof of Lemma 1.55.
3. Use the deterministic version of algorithm N to determine whether N_1 and N_2 are equivalent."

Introduction to the Theory of Computation

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

Approximation Algorithms [S-ToC-13]

♣ **Optimization problems**

- **In certain problems called *optimization problems*, we seek the best solution among a collection of possible solutions.**

♣ **Approximation algorithms**

- **In practice, we may not need the absolute best or optimal solution to a problem. A solution that is nearly optimal may be good enough and may be much easier to find.**
- **[D] An *approximation algorithm* is designed to find such approximately optimal solutions.**

Approximation Algorithms: An Example [S-ToC-13]

♣ **The graph vertex cover problem**

- **VERTEX-COVER = { <*G*, *k*> | *G* is an undirected graph that has a *k*-node vertex cover } [NP-complete].**
- **In the optimization version of the problem, called MIN-VERTEX-COVER, we aim to produce one of the smallest vertex covers among all possible vertex covers in the input graph. [an example of a *minimization problem*]**

♣ **A polynomial time algorithm that approximately solves MIN-VERTEX-COVER**

- **A = "On input <*G*>, where *G* is an undirected graph:**
  1. Repeat the following until all edges in *G* touch a marked edge:
  2. Find an edge in G untouched by any marked edge.
  3. Mark that edge.
  4. Output all nodes that are endpoints of marked edges."

Approximation Algorithms: An Example [S-ToC-13]

♣ **A polynomial time algorithm that approximately solves MIN-VERTEX-COVER**

THEOREM **10.1**

A is a polynomial time algorithm that produces a vertex cover of G that is no more than twice as large as a smallest vertex cover.

PROOF *A* obviously runs in polynomial time. Let X be the set of nodes that it outputs. Let H be the set of edges that it marks. We know that X is a vertex cover because H contains or touches every edge in G, and hence X touches all edges in G.

To prove that X is at most twice as large as a smallest vertex cover Y, we establish two facts: X is twice as large as H, and H is not larger than Y. First, every edge in H contributes two nodes to X, so X is twice as large as H. Second, Y is a vertex cover, so every edge in H is touched by some node in Y. No such node touches two edges in H because the edges in H do not touch each other. Therefore, vertex cover Y is at least as large as H because Y contains a different node that touches every edge in H. Hence X is no more than twice as large as Y.

K-optimal Approximation Algorithms [S-ToC-13]

♣ ***K*-optimal approximation algorithms for minimization problems**

- **[D] An approximation algorithm for a minimization problem is *k-optimal* if it always finds a solution that is not more than *k* times optimal.**
- **The preceding algorithm A is 2-optimal for the vertex cover problem.**

♣ ***K*-optimal approximation algorithms for maximization problems**

- **[D] An approximation algorithm for a maximization problem is *k-optimal* if it always finds a solution that is at least 1/k times the size of the optimal.**

Approximation Algorithms: An Example [S-ToC-13]

♣ **The max cut problem of graph**

- **MAX-CUT = { <*G*, *k*> | *G* has a cut of size *k* or more } [NP-complete].**
- **In the optimization version of the problem, called LARGEST-CUT, we asks for a largest cut in a graph *G*. [an example of a *maximization problem*]**

♣ **A polynomial time algorithm that approximately solves LARGEST-CUT**

- **B = "On input <*G*>, where *G* is an undirected graph with nodes *V*:**
  1. Let *S* := ∅; and *T* := *V*.
  2. If moving a single node, either from S to T or from T to S, increases the size of the cut, make that move and repeat this stage.
  3. If no such node exists, output the current cut and halt."
- **B is 2-optimal for LARGEST-CUT.**

Approximation Algorithms: An Example [S-ToC-13]

♣ **A polynomial time algorithm that approximately solves LARGEST-CUT (MAX-CUT)**

THEOREM **10.2**

B is a polynomial time, 2-optimal approximation algorithm for *MAX-CUT*.

PROOF *B* runs in polynomial time because every execution of stage 2 increases the size of the cut to a maximum of the total number of edges in G.

Now we show that *B*'s cut is at least half optimal. Actually, we show something stronger: *B*'s cut edges are at least half of all edges in G. Observe that at every node of G, the number of cut edges is at least as large as the number of uncut edges, or *B* would have shifted that node to the other side. We add up the numbers of cut edges at every node. That sum is twice the total number of cut edges because every cut edge is counted once for each of its two endpoints. By the preceding observation, that sum must be at least the corresponding sum of the numbers of uncut edges at every node. Thus, G has at least as many cut edges as uncut edges. Therefore, the cut contains at least half of all edges.

Probabilistic Algorithms [S-ToC-13]

♣ **Probabilistic algorithms**

- **[D] A *probabilistic algorithm* is an algorithm designed to use the outcome of a random process.**
- **Typically, such an algorithm would contain an instruction to "flip a coin" and the result of that coin flip would influence the algorithm's subsequent execution and output.**

♣

Probabilistic Turing Machines [S-ToC-13]

♣ **Probabilistic Turing machines**

DEFINITION **10.3**

A ***probabilistic Turing machine*** M is a type of nondeterministic Turing machine in which each nondeterministic step is called a ***coin-flip step*** and has two legal next moves. We assign a probability to each branch b of M's computation on input w as follows. Define the probability of branch b to be

$$\Pr[b] = 2^{-k},$$

where k is the number of coin-flip steps that occur on branch b. Define the probability that M accepts w to be

$$\Pr[M \text{ accepts } w] = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr[b].$$

Probabilistic Turing Machines [S-ToC-13]

♣ **Probabilistic Turing machines**

- **[D] The probability that probabilistic TM *M* accepts *w* is the probability that we would reach an accepting configuration if we simulated *M* on *w* by flipping a coin to determine which move to follow at each coin-flip step.**
- **[D] We let Pr[M rejects w] = 1 − Pr[M accepts w].**

♣ **A probabilistic TM decides a language**

- **[D] When a probabilistic TM *M* decides a language *A*, it must accept all strings in *A* and reject all strings out of *A* as usual, except that now we allow *M* a small probability of error.**
- **[D] For $0 \le \varepsilon < 1/2$, we say that *M decides language A with error probability* ε if $w \in A$ implies $\Pr[M \text{ accepts } w] \ge 1 - \varepsilon$, and $w \notin A$ implies $\Pr[M \text{ rejects } w] \ge 1 - \varepsilon$.**
- **In other words, the probability that we would obtain the wrong answer by simulating *M* is at most ε.**

The BPP Class of Languages [S-ToC-13]

♣ **The BPP class of languages**

DEFINITION **10.4**

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with an error probability of $\frac{1}{3}$.

- **We defined this class with an error probability of 1/3, but any constant error probability would yield an equivalent definition as long as it is strictly between 0 and 1/2 by virtue of the following amplification lemma.**

♣ **Amplification lemma**

LEMMA **10.5**

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-p(n)}$.

Amplification Lemma [S-ToC-13]

♣ **Amplification lemma**

LEMMA **10.5**

Let ϵ be a fixed constant strictly between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time Turing machine M_1 that operates with error probability ϵ has an equivalent probabilistic polynomial time Turing machine M_2 that operates with an error probability of $2^{-p(n)}$.

PROOF IDEA M_2 simulates M_1 by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of M_1 made.

- **The amplification lemma gives a simple way of making the error probability exponentially small.**
- **A probabilistic algorithm with an error probability of 2^{-100} is far more likely to give an erroneous result because the computer on which it runs has a hardware failure than because of an unlucky toss of its coins.**

Primality [S-ToC-13]

♣ **A exponential time algorithm to determine whether a number is prime**

- **One way to determine whether a number is prime is to try all possible integers less than that number and see whether any are divisors/factors.**
- **The algorithm has exponential time complexity because the magnitude of a number is exponential in its length.**

♣ **Probabilistic polynomial time algorithm for finding factors**

- **At present, no probabilistic polynomial time algorithm for finding factors is known to exist.**

Primality [S-ToC-13]

♣ Some notations from number theory

- [D] For any p greater than 1, we say that two numbers are *equivalent modulo p* if they differ by a multiple of p.
- [D] If numbers x and y are equivalent modulo p, we write $x \equiv y \pmod{p}$.
- [D] We let x mod p be the smallest non-negative y where $x \equiv y \pmod{p}$.
- Every number is equivalent modulo p to some member of the set $Z_p = \{0, \ldots, p-1\}$.
- For convenience, we let $Z^+_p = \{1, \ldots, p-1\}$.
- We may refer to the elements of these sets by other numbers that are equivalent modulo p, as when we refer to $p-1$ by -1.

Primality [S-ToC-13]

♣ Fermat's little theorem

THEOREM 10.6

If p is prime and $a \in Z^+_p$, then $a^{p-1} \equiv 1 \pmod{p}$.

♣ Fermat test

- Because the theorem states a necessary condition, it provides a type of "test" for primality, called a *Fermat test*.
- When we say that p passes the Fermat test at a, we mean that $a^{p-1} \equiv 1 \pmod{p}$.
- The Fermat's little theorem states that primes pass all Fermat tests for $a \in Z^+_p$.

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ Pseudo-primes

- [D] We call a number *pseudo-prime* if it passes Fermat tests for all smaller a's relatively prime to it.
- With the exception of the infrequent *Carmichael numbers*, which are composite yet pass all Fermat tests, the pseudo-prime numbers are identical to the prime numbers.

♣ Algorithm for determining pseudo-primality by Fermat test

- A pseudo-primality algorithm that goes through all Fermat tests would require exponential time.
- The key to the probabilistic polynomial time algorithm is that if a number is not pseudo-prime, it fails at least half of all tests.
- The algorithm works by trying several tests chosen at random. If any fail, the number must be composite.

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ A probabilistic polynomial time algorithm for determining pseudo-primality

- The algorithm contains a parameter k ($k \geq 1$) that determines the error probability.
- If p is pseudo-prime, it passes all tests and the algorithm accepts with certainty.
- If p is not pseudo-prime, it passes at most half of all tests. In that case, it passes each randomly selected test with probability at most 1/2. The probability that it passes all k randomly selected tests is thus at most 2^{-k}.
- The algorithm operates in polynomial time because modular exponentiation is computable in polynomial time.

PSEUDOPRIME = "On input p:

1. Select $a_1, \ldots, a_k$ randomly in Z^+_p.
2. Compute $a_i^{p-1} \bmod p$ for each i.
3. If all computed values are 1, *accept*; otherwise, *reject*."

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ A probabilistic polynomial time algorithm for determining primality

- The algorithm contains a parameter k ($k \geq 1$) that determines the maximum error probability to be 2^{-k}.

PRIME = "On input p:

1. If p is even, *accept* if $p = 2$; otherwise, *reject*.
2. Select $a_1, \ldots, a_k$ randomly in Z^+_p.
3. For each i from 1 to k:
4. Compute $a_i^{p-1} \bmod p$ and *reject* if different from 1.
5. Let $p - 1 = s \cdot 2^l$ where s is odd.
6. Compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, a_i^{s \cdot 2^2}, \ldots, a_i^{s \cdot 2^l}$ modulo p.
7. If some element of this sequence is not 1, find the last element that is not 1 and *reject* if that element is not -1.
8. All tests have passed at this point, so *accept*."

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ A probabilistic polynomial time algorithm for determining primality

- The principle underlying the algorithm is as follows:
- The number 1 has exactly two square roots, 1 and -1, modulo any prime p. For many composite numbers, including all the Carmichael numbers, 1 has four or more square roots. (For example, ±1 and ±8 are the four square roots of 1, modulo 21.)
- If a number passes the Fermat test at a, the algorithm finds one of its square roots of 1 at random and determines whether that square root is 1 or -1. If it is not, we know that the number is not prime.
- We can obtain square roots of 1 if p passes the Fermat test at a because $a^{p-1} \bmod p = 1$, and so $a^{(p-1)/2} \bmod p$ is a square root of 1. If that value is still 1, we may repeatedly divide the exponent by 2, so long as the resulting exponent remains an integer, and see whether the first number that is different from 1 is -1 or some other number.

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ **A probabilistic polynomial time algorithm for determining primality**

- **The following two lemmas show that algorithm PRIME works correctly.**
- **Obviously the algorithm is correct when p is even, so we only consider the case when p is odd.**
- **We say that a_i is a *(compositeness) witness* if the algorithm rejects at either stage 4 or 7, using a_i.**

LEMMA **10.7**

If p is an odd prime number, $\Pr[PRIME \text{ accepts } p] = 1$.

LEMMA **10.8**

If p is an odd composite number, $\Pr[PRIME \text{ accepts } p] \leq 2^{-k}$.

Algorithms for Determining Primality by Fermat Test [S-ToC-13]

♣ **A probabilistic polynomial time algorithm for determining primality**

- **Let PRIMES = { n | n is a prime number in binary }**

THEOREM **10.9**

$PRIMES \in$ BPP.

- **The algorithm PRIME has one-sided error. When the algorithm outputs reject, we know that the input must be composite. When the output is accept, we know only that the input could be prime or composite.**
- **Thus, an incorrect answer can only occur when the input is a composite number.**

The RP Class of Languages [S-ToC-13]

♣ **The RP class of languages**

DEFINITION **10.10**

RP is the class of languages that are decided by probabilistic polynomial time Turing machines where inputs in the language are accepted with a probability of at least $\frac{1}{2}$, and inputs not in the language are rejected with a probability of 1.

- **The one-sided error feature is common to many probabilistic algorithms, so the special complexity class RP is designated for it.**
- **Let COMPOSITES = { x | $x = pq$, for integers $p, q > 1$ }.**
- **COMPOSITES $\in$ RP.**

Introduction to the Theory of Computation

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