

CSCI3360 Introduction to Computational Complexity (2026 Fall)

Xiao Liang

<https://xiao-liang.github.io>

Department of Computer Science and Engineering
The Chinese University of Hong Kong

(Updated on September 27, 2026)

Disclaimer: These lecture notes originate from the 2026-Fall offering of the course. They have been lightly edited for accuracy and completeness, and may still contain typographical or other errors.

Contents

1	The of Computation: Turing Machines	1
1.1	Before Turing Machines: Restricted Models of Computation	1
1.1.1	Finite Automata	1
1.1.2	Pushdown Automata	2
1.2	Turing Machines	4
1.2.1	Informal Description	4
1.2.2	Formal Definition	5
1.2.3	A Turing Machine for Palindromes	8
1.2.4	Running Example: $x = \text{radar}$	8
1.2.5	Going Beyond Pushdown Automata	10
1.3	Robustness of the Turing-Machine Model	12
1.3.1	Three Tapes Versus One Tape	12
1.3.2	One-Way Versus Bidirectional Tapes	14
1.4	Universal Turing Machines	16
1.4.1	Machines as Data	16
1.4.2	Definition and Basic Simulation	17
1.4.3	Efficient Universal Simulation	17
1.4.4	*Proof of Theorem 1.4.2 (Part 1/2)	19
1.4.4.1	Example 1: Computing powers by repeated squaring	20
1.4.4.2	Example 2: Incrementing a binary counter	21
1.4.5	*Proof of Theorem 1.4.2 (Part 2/2)	22
1.4.5.1	The main idea: move the contents, not the simulated heads	22
1.4.5.2	Buffers and their meaning	23
1.4.5.3	Simulating one transition	24
1.4.5.4	Simulating a move to the right	24
1.4.5.5	The cost of one rearrangement	25
1.4.5.6	Why large rearrangements are rare	26
1.4.5.7	Summing the costs	27
1.5	The Church–Turing Thesis	27
1.5.1	The Standard Church–Turing Thesis	27
1.5.2	The Extended Church–Turing Thesis	28
1.5.3	Quantum Computing and the Extended Thesis	29

1.5.4	The Turing Machine as a Final Model	29
1.6	Closing Remarks	30
2	Deterministic Polynomial Time: P	31
2.1	Polynomial Time and the Class P	31
2.1.1	Why Polynomial Time?	31
2.1.2	The Formal Definition	32
2.1.3	Input Encodings and Bit Complexity	33
2.2	Representative Decision Problems in P	34
2.2.1	Relative Primality	34
2.2.2	Directed Reachability	35
2.2.3	Bipartite Matching with a Size Threshold	37
2.3	Decision and Search	41
2.3.1	Witnesses and Search Problems	41
2.3.2	Polynomial-Time Search-to-Decision Reductions	42
2.4	When the Natural Existence Question Is Not Enough	45
2.4.1	The Maximum-Cut Problem	46
2.4.2	Factorization versus Compositeness	47
2.4.3	Further Discussion of Primality Testing	48
2.5	Randomized Algorithms and the Limits of P	53
3	Non-Deterministic Polynomial Time: NP	55
3.1	From Efficient Solution to Efficient Verification	55
3.2	Witnesses, Polynomial-Time Verifiers, and the Definition of NP	56
3.3	Examples of Problems in NP	58
3.3.1	Boolean Satisfiability	59
3.3.2	Hamiltonian Path	64
3.3.3	Clique	65
3.3.4	Independent Set	66
3.3.5	Graph Three-Coloring	67
3.3.6	Knapsack	68
3.3.7	Subset Sum	69
3.3.8	Composite Integers	70
3.4	The Relationship Between P and NP	71
3.4.1	The P Versus NP Question	72
3.5	Nondeterministic Turing Machines	72
3.5.1	Nondeterministic Computation	72
3.5.2	The Guess-and-Check View	74
3.5.3	The Nondeterministic Definition of NP	75
3.6	Equivalence of the Two Definitions of NP	75
3.6.1	From Verifiers to Nondeterministic Machines	75
3.6.2	From Nondeterministic Machines to Verifiers	76
	Bibliography	78

Chapter 1

The of Computation: Turing Machines

Computational complexity asks not only whether a problem can be solved, but also how many computational resources are required to solve it. Before we can measure resources such as time and memory, however, we must specify what counts as a computation. In other words, we need a mathematical model of an algorithm.

We begin by recalling two models from formal-language theory: finite automata and push-down automata, which were already covered in **CSCI3130 Formal Languages and Automata Theory**. These models are useful precisely because their memory is restricted, but the same restrictions prevent them from expressing general computation. We then introduce the Turing machine, work through algorithms for recognizing palindromes and a non-context-free language, and examine the robustness of the model. Finally, we introduce universal Turing machines and discuss the Church–Turing thesis and its complexity-theoretic extension.

To study the complexity of computation, we must first define a model of computation. The model must be precise enough for mathematical proofs and expressive enough to represent general algorithms.

1.1 Before Turing Machines: Restricted Models of Computation

Nothing from [Section 1.1](#) will be tested on quizzes or exams!

1.1.1 Finite Automata

Recall that a deterministic finite automaton has a finite set of states and reads its input from left to right. At every step, its next state is determined by its current state and the next input symbol.

Definition 1.1.1 (Deterministic Finite Automaton). A deterministic finite automaton, or DFA, is a tuple

$$A = (Q, \Sigma, \delta, q_0, F),$$

where:

- Q is a finite set of states;
- Σ is a finite input alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function;
- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accepting states.

The automaton reads its input only once, from left to right. It accepts an input $x \in \Sigma^*$ if the state reached after reading all of x belongs to F . ◇

The current state of a DFA contains all the information that the machine retains about the portion of the input that it has already read. Since Q is finite, a DFA has only a constant amount of memory, independent of the input length.

Example 1.1.1 (A Language Recognized by a DFA). *Consider the language*

$$L_{\text{even}} := \{x \in \{0,1\}^* : x \text{ contains an even number of 1's}\}.$$

A two-state DFA recognizes this language. One state records that the number of 1's seen so far is even, and the other records that it is odd. Reading a 1 switches states, while reading a 0 leaves the state unchanged. This is illustrated in Figure 1.1.

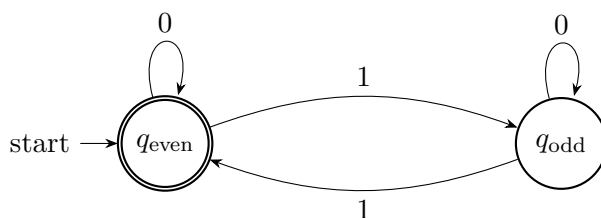


Figure 1.1: A DFA recognizing binary strings containing an even number of 1's.

The boundary of finite-state computation

Finite automata recognize exactly the *regular languages*. They are well suited to problems in which only a bounded amount of historical information is needed.

For example, finite automata are used in the lexical-analysis stage of a compiler. A lexer may recognize identifiers, integer literals, punctuation, and keywords using regular expressions and finite-state machinery.

The limitation is that a DFA cannot store an unbounded counter. Consequently, it cannot recognize the language

$$L_{\text{eq}} := \{0^n 1^n : n \geq 0\}. \quad (1.1)$$

To recognize L_{eq} , a machine must compare an arbitrarily large number of 0's with an arbitrarily large number of 1's.

A finite set of states cannot remember the required number for all possible input lengths.

1.1.2 Pushdown Automata

A pushdown automaton extends a finite automaton by providing it with a stack. The stack can grow with the input, so a pushdown automaton can remember an unbounded amount of information. However, that information can be accessed only in last-in, first-out order.

Definition 1.1.2 (Deterministic Pushdown Automaton). *A deterministic pushdown automaton (DPDA) is a 6-tuple*

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

where:

1. Q is a finite set of states;
2. Σ is the finite input alphabet;

3. Γ is the finite stack alphabet;
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow Q \times \Gamma_\varepsilon$ is the transition function; Here,

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\} \quad \text{and} \quad \Gamma_\varepsilon = \Gamma \cup \{\varepsilon\},$$

where ε denotes the empty string.

5. $q_0 \in Q$ is the start state; and
6. $F \subseteq Q$ is the set of accept states.

The automaton reads its input only once, from left to right. It accepts an input $x \in \Sigma^*$ if the state reached after reading all of x belongs to F .

◇

Remark 1.1.1 (On Nondeterminism). There is a more powerful version called a nondeterministic pushdown automaton (NPDA), in which the transition function is allowed to return a set of possible next states and stack symbols. Deterministic and nondeterministic pushdown automata do not have the same expressive power. Every language recognized by a DPDA can be recognized by an NPDA, but the converse does not hold. More precisely,

$$\text{DCFL} \subsetneq \text{CFL}.$$

Thus, NPDAs recognize all context-free languages, whereas DPDAs recognize only the proper subclass of deterministic context-free languages.

Think of the stack as restricted working memory. The machine can access the most recently stored symbol, but it cannot directly inspect a symbol buried in the middle of the stack.

Example 1.1.2 (Recognizing $0^n 1^n$ with a Stack). A DPDA can recognize L_{eq} from [Equation \(1.1\)](#) as follows:

1. For each 0 in the first part of the input, push one marker onto the stack.
2. When the first 1 is encountered, begin popping one marker for each 1.
3. Accept if the input ends exactly when the stack returns to its initial marker.

Such a DPDA is shown in [Figure 1.2](#). For the input 000111, the stack first grows by three markers and then shrinks by three markers. For 00111, the machine attempts to pop a third marker after only two have been stored, so it rejects.

Nondeterministic pushdown automata recognize exactly the *context-free languages*. In compiler design, they provide the mathematical basis for parsing nested syntactic structures, such as balanced parentheses, arithmetic expressions, and recursively nested blocks.

The boundary of stack-based computation

Although a stack is more powerful than finite-state memory, one stack does not provide general-purpose random-access memory. A standard example beyond the power of pushdown automata is

$$L_{\text{three}} := \{0^n 1^n 2^n : n \geq 0\}. \quad (1.2)$$

A machine recognizing this language must verify that three independently delimited blocks have the same length.

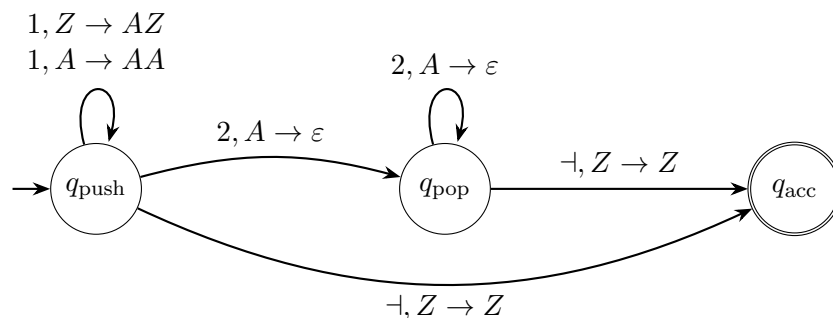


Figure 1.2: A deterministic pushdown automaton for $L = \{1^n 2^n \mid n \geq 0\}$. Each transition has the form “input, stack top $\rightarrow$ replacement.” The automaton pushes one A for every 1 and pops one A for every 2. Here, Z is the bottom-of-stack symbol, ϵ means that the top symbol is removed, and ϵ marks the end of the input. Thus, $\epsilon, Z \rightarrow Z$ means that the input has ended, only Z remains on the stack, and Z is left unchanged.

A stack can conveniently match the 0’s against the 1’s by pushing for the first block and popping for the second. After those comparisons, however, the stored count has been destroyed, leaving no copy with which to check the 2’s.

Fact 1.1.1 (Pushdown-Automaton Boundary). *The language L_{three} in Equation (1.2) is not context-free. Therefore, no pushdown automaton recognizes it.*

The statement in Fact 1.1.1 can be proved using the pumping lemma for context-free languages or Ogden’s lemma. The important point for us is conceptual: the stack provides unbounded but highly structured memory, and that structure imposes a genuine limit on the languages a PDA can recognize.

Takeaway

Finite automata and pushdown automata remain important because their restrictions make them suitable for particular applications and allow precise classifications of languages. They are not, however, expressive enough to represent every algorithm that we would normally regard as executable by a computer. This motivates the Turing-machine model.

1.2 Turing Machines

Our next goal is to formalize a general-purpose model of computation. To this end, we introduce *Turing machines*, which may be viewed roughly as finite automata augmented with unbounded memory and the ability to move back and forth over their input, rather than being restricted to a single left-to-right pass.

1.2.1 Informal Description

An informal algorithm consists of a fixed collection of mechanical rules that can be applied repeatedly. A typical step may perform the following operations:

- read symbols from the input or working memory;
- write symbols to working memory;
- update a finite control state;

- move to nearby memory locations; and
- halt with an output.

A Turing machine turns this description into a precise mathematical object. Each tape is divided into cells, and each cell stores one symbol from a finite alphabet. A tape head reads the symbol in the current cell and may move one position left or right.

The tapes in our definition are unbounded to the right and have a leftmost cell. This is a mathematical abstraction: an execution lasting T steps can visit at most $T + 1$ cells on any tape, so every finite computation uses only a finite portion of each tape.

1.2.2 Formal Definition

Definition 1.2.1 (Three-Tape Turing Machine). *A three-tape Turing machine is a tuple*

$$M = (\Gamma, Q, \delta),$$

with the following components:

- Γ is a finite set of symbols that M 's tape can contain. We call it the alphabet.
We assume that it contains a blank symbol $\sqcup$, a left-end marker $\triangleright$, and a set of some characters¹ (e.g., the English alphabet, or the binary bits $\{0, 1\}$).
- Q is a finite set of states containing a start state q_{start} and a halting state q_{halt} .
- The transition function is

$$\delta : Q \times \Gamma^3 \longrightarrow Q \times \Gamma^2 \times \{L, S, R\}^3. \quad (1.3)$$

The first tape is a read-only input tape. The second tape is a read/write work tape, and the third tape is a read/write output tape. $\diamond$

Let us unpack the transition function in [Equation \(1.3\)](#). Suppose

$$\delta(q, a, b, c) = (q', b', c', d_1, d_2, d_3).$$

This transition instructs the machine to:

1. change its state from q to q' ;
2. leave the input symbol a unchanged; (recall that the input tape is read-only;)
3. replace the work-tape symbol b by b' ;
4. replace the output-tape symbol c by c' ; and
5. move the three heads according to

$$d_1, d_2, d_3 \in \{L, S, R\},$$

where L , S , and R mean move left, stay in place, and move right, respectively.

If a head is at the left-end marker and is instructed to move left, it remains in place.

¹Note that the choice of characters is arbitrary. But it is without loss of generality that we only consider binary alphabets, because any finite alphabet can be encoded as a binary string. On potential issue is the size of encoding may grow. But as we will see, this does not affect the computational power of Turing machines, at least to the extent that this course is concerned.

Definition 1.2.2 (Start Configuration). *On input $x = x_1x_2 \cdots x_n \in \Gamma^*$, the start configuration of a three-tape Turing machine is defined as follows:*

- the input tape contains

$$\triangleright x_1x_2 \cdots x_n \sqcup \sqcup \cdots ;$$

- the work and output tapes each contain

$$\triangleright \sqcup \sqcup \sqcup \cdots ;$$

- all three heads are placed at their left-end markers; and
- the finite control is in state q_{start} .

◇

The machine proceeds by repeatedly applying its transition function. If it enters q_{halt} , its computation stops. The finite string written immediately to the right of the output tape's left-end marker is interpreted as the machine's output.

Definition 1.2.3 (Computing a Function and Running Time). *Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$, and let $T : \mathbb{N} \rightarrow \mathbb{N}$. A Turing machine M computes f in time $T(n)$ if, for every input $x \in \{0,1\}^*$, the machine:*

1. starts in the start configuration on x ;
2. halts after at most $T(|x|)$ transitions; and
3. has $f(x)$ written on its output tape when it halts.

The running time of M is the maximum number of transitions it performs over all inputs of a given length.

◇

The quantity $n = |x|$ is the input length. Complexity theory studies how resource usage grows as a function of n , rather than how long a machine takes on only one particular input.

Definition 1.2.4 (Deciding a Language). *Let $L \subseteq \{0,1\}^*$. A Turing machine M decides L if it halts on every input x and outputs*

$$M(x) = \begin{cases} 1, & x \in L, \\ 0, & x \notin L. \end{cases} \quad (1.4)$$

Equivalently, M computes the characteristic function of L .

◇

Remark 1.2.1. *In complexity theory, we normally study machines that halt on every input. A machine that accepts members of a language but may run forever on nonmembers is called a recognizer rather than a decider. Unless stated otherwise, the algorithms in these notes are deciders.*

Notice that [Definition 1.2.4](#) use the concept of a language to formalize the notion of a decision problem. Do not worry if you are not familiar with the concept of a language. Let us explain it in the following example.

Example 1.2.1 (Representing a Property as a Language). *A language is a set of strings satisfying a specified property. For example, consider the property*

“The number of 1’s in the binary string is a multiple of 3.”

This property determines the language

$$L_{\text{mult-3}} := \{x \in \{0,1\}^* : \text{the number of 1's in } x \text{ is a multiple of 3}\}.$$

If $\#_1(x)$ denotes the number of occurrences of 1 in x , then the same language can be written more concisely as

$$L_{\text{mult-3}} = \{x \in \{0,1\}^* : \#_1(x) \equiv 0 \pmod{3}\}.$$

For instance,

$$\varepsilon, 0, 000, 111, 10101 \in L_{\text{mult-3}},$$

because these strings contain either zero or three occurrences of 1. On the other hand,

$$1, 10, 110, 1111 \notin L_{\text{mult-3}}.$$

Notice that 0 is a multiple of 3, so every string containing no 1’s, including the empty string ε , belongs to the language.

This language corresponds to the following decision problem:

Input: A binary string x .

Question: Is the number of 1’s in x a multiple of 3?

The answer is “yes” exactly when $x \in L_{\text{mult-3}}$, and it is “no” exactly when $x \notin L_{\text{mult-3}}$.

Remark 1.2.2 (Decision Problems as Languages). *Every decision problem whose instances have finite descriptions can be formalized as a language. First, encode each problem instance I as a binary string $\langle I \rangle$. The language associated with the problem is then*

$$L = \{\langle I \rangle \in \{0,1\}^* : \text{the answer for instance } I \text{ is “yes”}\}.$$

Thus, solving the decision problem is equivalent to deciding whether the encoding $\langle I \rangle$ belongs to L .

Strings in L are called yes-instances, while strings not in L are called no-instances.

Why Turing Machines?

The Turing machine provides a simple but powerful mathematical model of general computation. Its finite control represents the machine’s fixed program: the set of states and transition rules is chosen in advance and does not depend on the particular input. The work tape represents memory whose size may grow as the computation proceeds. Thus, unlike a finite automaton, a Turing machine can store an amount of information that is not bounded by a constant.

Despite having potentially unbounded memory, the machine operates locally. During one transition, it can inspect only the three currently scanned symbols, overwrite only the scanned tape cells, and move each head by at most one cell. A complicated computation must therefore be carried out as a sequence of elementary local operations. Consequently, the number of transitions gives a meaningful measure of sequential running time, while the number of tape cells used gives a meaningful measure of memory consumption.

This model is also sufficiently general to express ordinary algorithms. Data structures, intermediate results, and program configurations can all be encoded as strings on the tapes. Although such encodings may be less convenient than the data types and instructions of a programming language, they allow a Turing machine to simulate the essential steps of a computer

program. In particular, a universal Turing machine can receive the description of another machine together with its input and simulate that machine, capturing the idea of a stored-program computer.

Finally, the simplicity of the model makes mathematical analysis possible. Questions about whether a problem is computable can be stated precisely in terms of whether some Turing machine decides the corresponding language. Questions about efficiency can likewise be studied by counting the machine's transitions and tape usage. Moreover, standard variants of the model—such as machines with one tape, several tapes, or different tape alphabets—compute the same class of languages and can simulate one another with controlled overhead. The Turing machine therefore abstracts away implementation details while retaining the fundamental notions of algorithm, memory, running time, and computability.

1.2.3 A Turing Machine for Palindromes

We now use the formal model to design a familiar algorithm. The purpose of this example is not to argue that palindrome testing is difficult. Rather, it illustrates how an ordinary algorithm can be implemented using tapes, head movements, and a finite collection of states.

Definition 1.2.5 (Palindrome Language). *For an alphabet Σ , define*

$$\text{PAL}_\Sigma := \{x \in \Sigma^* : x = x^R\},$$

where x^R denotes the reversal of x .

For example,

$$\text{radar}^R = \text{radar}, \quad \text{complexity}^R = \text{ytixelpmoc}.$$

Thus, $\text{radar} \in \text{PAL}_\Sigma$, while $\text{complexity} \notin \text{PAL}_\Sigma$. ◇

Algorithm 1.2.1: Three-Tape Palindrome Algorithm

On input $x = x_1x_2 \cdots x_n$:

1. Scan the input from left to right, copying each symbol to the work tape.
2. When the input head reaches the first blank after x , move the work head left so that it scans x_n .
3. Return the input head to the beginning of x , while leaving the work head in place.
4. Repeatedly compare the input symbols from left to right with the work-tape symbols from right to left.
5. If any pair differs, write 0 on the output tape and halt.
6. If all pairs agree, write 1 on the output tape and halt.

1.2.4 Running Example: $x = \text{radar}$

Let us follow the machine through its major phases.

Phase 1: Copy the input. Initially, the relevant portions of the tapes are

input	:	▷ r a d a r □,
work	:	▷ □ □ □ □ □ □,
output	:	▷ □.

where the **red color** indicates the current position of each head.

After the copying phase, the work tape contains a copy of the input:

input : ▷ **r** a d a r ⊐,
work : ▷ **r** a d a r ⊐,
output : ▷ ⊐.

At this point, both heads have reached the blank immediately following the word.

Phase 2: Position the two heads. The work head moves left once and therefore scans the last symbol, **r**. The input head moves left until it reaches ▷, and then moves right once to the first symbol. The heads now scan opposite ends of the word:

input : ▷ **r** a d a r ⊐,
work : ▷ **r** a d a **r** ⊐,
output : ▷ ⊐.

Phase 3: Compare in opposite directions. The sequence of comparisons is shown in Table 1.1.

Comparison	Input-tape symbol	Work-tape symbol	Result
1	r	r	Match
2	a	a	Match
3	d	d	Match
4	a	a	Match
5	r	r	Match

Table 1.1: The comparison phase of the palindrome machine on input **radar**. The input head moves right while the work head moves left.

After the fifth successful comparison, the input head reaches the blank following the input and the work head reaches the left-end marker. The machine writes 1 to the output tape and halts.

Theorem 1.2.1 (Correctness and Running Time of Palindrome Testing). *The machine in Algorithm 1.2.1 decides PAL_Σ in time $3n + O(1)$, where n is the input length.* ◇

Proof. Let the input be $x = x_1x_2 \cdots x_n$.

The copying phase writes x_i into the i -th work-tape cell for every $i \in \{1, \dots, n\}$. Therefore, after copying, the work tape contains exactly $x_1x_2 \cdots x_n$.

The machine then positions the input head over x_1 and the work head over x_n . After $i - 1$ successful comparisons, the input head scans x_i , while the work head scans x_{n-i+1} . Thus, the i -th comparison checks whether

$$x_i = x_{n-i+1}.$$

The machine outputs 1 exactly when this equality holds for every i . This condition is equivalent to $x = x^R$, so the algorithm is correct.

The machine makes $3n + O(1)$ transitions in total:

- $n + O(1)$ transitions to copy the input;
- $n + O(1)$ transitions to return the input head to the beginning; and
- $n + O(1)$ transitions to compare the two copies.

In each phase, the n term accounts for the transitions that process or traverse the n input symbols. The $O(1)$ term accounts for a fixed number of additional transitions, such as detecting an endmarker, changing states, repositioning a head by a fixed number of cells, and entering an accepting or rejecting state. It is not hard to see that the number of these additional transitions is bounded by a constant independent of n . ■

Remark 1.2.3. *We remark that palindromes can actually be recognized by a nondeterministic pushdown automaton. Therefore, palindrome recognition is not being used here to separate Turing machines from all pushdown automata. It is used as a simple example of Turing-machine design. The separation from pushdown automata will instead use L_{three} .*

Takeaway

The work tape allows the machine to store a copy of its input and revisit that information in a different order. This is an elementary example of a general pattern: a Turing machine can reorganize and repeatedly access information rather than being restricted to one left-to-right pass or one stack discipline.

1.2.5 Going Beyond Pushdown Automata

We now return to the language

$$L_{\text{three}} = \{0^n 1^n 2^n : n \geq 0\},$$

which no pushdown automaton recognizes. We show that a Turing machine can decide this language.

Algorithm 1.2.2: Deciding L_{three}

On input x :

1. Verify that x has the form $0^*1^*2^*$. Reject if the symbols appear in any other order.
2. While scanning the block of 0's, write one marker symbol "X" on the work tape for each 0.
3. At the beginning of the block of 1's, return the work head to its first marker.
4. For each input symbol 1, move the work head right by one marker. Reject if the input has more 1's than there are markers.
5. At the end of the 1-block, reject unless the work head has passed exactly all the markers.
6. Return the work head to its first marker.
7. For each input symbol 2, move the work head right by one marker. Reject if the input has more 2's than there are markers.
8. At the end of the input, accept if and only if the work head has passed exactly all the markers.

Although [Algorithm 1.2.2](#) is stated at a high level, each instruction can be expanded into ordinary Turing-machine transitions. We illustrate this for two representative steps. Let $\triangleright$ denote a left-endmarker and let $\sqcup$ denote the blank symbol.

- First, the machine can verify that the input has the form $0^*1^*2^*$ using only its finite control. It uses three states, corresponding to the three possible blocks. In the first state it reads 0's; upon seeing the first 1, it enters the second state; and upon seeing the first 2, it enters the third state. A 0 seen in either the second or third state, or a 1 seen in the third state, causes immediate rejection. Reaching the end of the input without such a violation confirms that the symbols occur in the required order.
- Second, consider Steps 3–5. After Step 2, the relevant part of the work tape has the form

$$\triangleright X^n \sqcup \sqcup \cdots .$$

To return to the first marker, the work head repeatedly moves left until it sees $\triangleright$, and then moves one cell to the right. For each input symbol 1, the machine checks that the work head is scanning an X and then moves the work head one cell to the right. If it sees $\sqcup$ while another 1 remains, then there are more 1's than 0's, so it rejects. When the input head reaches the first 2, the machine checks the work-tape symbol currently under the work head. If that symbol is X , then some markers remain and there are fewer 1's than 0's; if it is $\sqcup$, then exactly all n markers have been traversed. Thus, these elementary tests implement the high-level instruction that the numbers of 0's and 1's must be equal. The comparison with the 2's is implemented in exactly the same way: the work head first returns to the left-endmarker and then traverses one X -cell for each 2.

These examples illustrate a general principle: phrases such as “verify the order,” “return to the first marker,” and “check that all markers have been passed” abbreviate finite sequences of transitions that inspect one symbol, change state, and move a tape head by one cell at a time.

Example 1.2.2 (Input 000111222). *The machine first records three markers:*

$$\triangleright XXX \sqcup \sqcup \cdots .$$

It then uses these markers to count the three 1's. After returning to the beginning of the markers, it uses the same three markers again to count the three 2's. Both comparisons succeed, so the machine accepts.

For the input 0011122, the machine encounters a third 1 after it has already passed its two stored markers. It therefore rejects.

Proposition 1.2.1. *Algorithm 1.2.2 is an $O(n)$ -time (i.e., linear-time) Turing machine that decides L_{three} .*

Proof. The algorithm in Algorithm 1.2.2 accepts only strings of the form $0^a 1^b 2^c$. It writes exactly a markers and accepts only if both the 1-block and the 2-block contain exactly one symbol per marker. Therefore, it accepts exactly when

$$a = b = c.$$

This is precisely the condition for membership in L_{three} .

The algorithm is in linear time: each input symbol is scanned a constant number of times, and the work-tape head makes only a constant number of complete traversals of the n stored markers. ■

Turing machines are strictly more expressive than finite automata and pushdown automata: they can decide languages, such as $\{0^n 1^n 2^n : n \geq 0\}$, that neither of the earlier models can recognize.

1.3 Robustness of the Turing-Machine Model

Many details of Definition 1.2.1 may appear arbitrary. Why three tapes rather than one? Why are tapes unbounded only to the right? Why is the input tape read-only? We now show that several natural variations produce the same class of computable functions. The simulations may change running time, but they do not change what can ultimately be computed.

1.3.1 Three Tapes Versus One Tape

Definition 1.3.1 (Single-Tape Turing Machine). *A single-tape Turing machine has one read/write tape that initially contains*

$$\triangleright x \sqcup \sqcup \cdots .$$

The same tape is used for the input, working memory, and output. ◇

Lemma 1.3.1 (A Three-Tape Machine Simulates a Single-Tape Machine). *If a single-tape Turing machine M computes a function in time $T(n)$, then a three-tape Turing machine computes the same function in time $O(T(n))$.* ◇

Proof. The three-tape machine simply uses the work tape exactly as M uses its single tape, storing M 's simulated state in its own finite control.

Each transition of M is simulated by a constant number of transitions. When M halts, the simulator copies the designated output to its output tape. Thus, apart from the initial and final linear-time copies, the simulation has constant overhead per transition. ■

The reverse direction requires more work because one physical tape must represent three independently moving tapes.

Lemma 1.3.2 (A Single-Tape Machine Simulates a Three-Tape Machine). *Suppose a three-tape Turing machine M runs in time $T(n)$. There is a single-tape Turing machine $\widetilde{M}$ that computes the same function in time*

$$O(T(n)^2).$$

◇

Proof. Let Γ be the tape alphabet of M , including the blank symbol $\sqcup$, and write $g = |\Gamma|$. We assume that tape 1 initially contains the input, tapes 2 and 3 are initially blank, and all three heads begin at their designated starting cells.

Encoding the three tapes. The machine $\widetilde{M}$ stores the three simulated tapes using its single physical tape. An encoded physical cell has the form

$$((a_1, h_1), (a_2, h_2), (a_3, h_3)),$$

where $a_i \in \Gamma$ is the symbol stored in that cell of simulated tape i , and

$$h_i \in \{0, 1\}$$

indicates whether the head of tape i is currently scanning that cell.

Each component (a_i, h_i) within the cell $((a_1, h_1), (a_2, h_2), (a_3, h_3))$ is called a *track*. Conceptually, the three tracks are three aligned sequences laid on top of one another, with the i -th track recording the contents and head position of the i -th simulated tape.

Thus, for example, the encoded cell

$$((a_1, 0), (a_2, 1), (a_3, 0))$$

stores the symbols $(a_1, 0)$, $(a_2, 1)$, $(a_3, 0)$ on the first, second, and third tracks, respectively. The head bits indicate that the second simulated head is located at this position, whereas the first and third simulated heads are not.

For each track there are $2g$ possible symbol-marker pairs:

$$|\Gamma \times \{0, 1\}| = 2g.$$

Consequently, the number of possible three-track (i.e., one-cell) symbols is

$$(2g)^3 = 8g^3.$$

The tape alphabet may also retain a separate copy of the original input alphabet for input cells that have not yet been encoded, adding at most g symbols. In addition, the simulator may use a constant number of special symbols for bookkeeping. Hence $\widetilde{M}$ can be given a tape alphabet of size at most

$$8g^3 + g + O(1).$$

In particular, the simulated alphabet remains finite. Notice that the head markers must be attached independently to the three components; merely adding one marked copy of each tuple would not suffice, since several simulated heads may occupy the same physical position.

At every point in the simulation, exactly one cell has $h_i = 1$ for each $i \in \{1, 2, 3\}$. The finite control of $\widetilde{M}$ also stores the current state of M .

To simulate one transition of M , the machine $\widetilde{M}$ performs two sweeps:

1. It sweeps from left to right, finds the three marked symbols, and stores their values in its finite control (see [Remark 1.3.1](#)). Once it has the current state and the three scanned symbols, it can determine the next state, the symbols to write, and the directions to move each head; this is done by invoking M 's transition function.
2. It then sweeps back from right to left, updating the three symbols and moving the three head markers as required.

The main take way is: to simulate one transition of M , the machine $\widetilde{M}$ must sweep across the entire encoded region of its tape twice. Since M is a $T(n)$ -time machine, the encoded region has length $O(T(n))$. Therefore, simulating one transition of M takes $O(T(n))$ time. Simulating all $T(n)$ transitions consequently takes

$$T(n) \cdot O(T(n)) = O(T(n)^2).$$

Remark 1.3.1 (On the Finite Control of a Turing Machine). *The finite control of a Turing machine refers to its finite set of states. Thus, saying that M stores information in its finite control means that this information can be incorporated into its current state. In the simulation above, the finite control does not store the positions of the three simulated heads, since these positions may be arbitrarily large. Instead, the head positions are recorded on the tape by the marker bits h_1, h_2, h_3 . The finite control stores only the current state q of M and the three scanned symbols a_1, a_2, a_3 . This is possible because*

$$(q, a_1, a_2, a_3) \in Q_M \times \Gamma^3,$$

which is a finite set. Equivalently, $\widetilde{M}$ may have a distinct state for each possible tuple (q, a_1, a_2, a_3) . More generally, storing information in finite control is possible whenever that information ranges over a fixed finite set independent of the input length; unbounded information, such as arbitrary head positions, must instead be stored on the tape. ■

Combining [Lemma 1.3.1](#) and [Lemma 1.3.2](#) shows that the two models compute the same functions, up to a polynomial slowdown:

Theorem 1.3.1 (Tape-Number Robustness). *A function is computable by a three-tape Turing machine if and only if it is computable by a single-tape Turing machine. The efficiency is up to a quadratic relationship in running time.* ◇

Remark 1.3.2. *The quadratic overhead in [Lemma 1.3.2](#) is an upper bound for a straightforward simulation, not a claim that every multi-tape computation inherently requires a quadratic slowdown. More sophisticated simulations can achieve smaller overhead.*

1.3.2 One-Way Versus Bidirectional Tapes

Definition 1.3.2 (Bidirectional Turing Machine). *A bidirectional Turing machine is a Turing machine whose tape cells are indexed by all integers:*

$$\dots, -3, -2, -1, 0, 1, 2, 3, \dots$$

Thus, its tape is unbounded in both directions. ◇

A standard one-way tape has cells indexed by $0, 1, 2, \dots$. To simulate a bidirectional tape, we fold it around position 0. See [Figure 1.3](#) for an illustration.

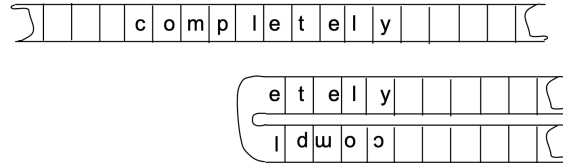


Figure 1.3: Folding a bidirectional tape between positions -1 and 0 to simulate it using a one-way-infinite tape. Adapted from [\[AB09\]](#).

Lemma 1.3.3 (Bidirectional Tapes Can Be Folded). *Let M be a Turing machine whose tapes are infinite in both directions. If M computes a function f in time $T(n)$, then there is a standard Turing machine $\widetilde{M}$, whose tapes are infinite only to the right, that computes f in time $O(T(n))$.* ◇

Proof. We describe how to replace each bidirectional tape of M with a standard tape that has a left endpoint. Because the number of tapes is fixed, it suffices to explain the construction for one tape.

Index the cells of M 's bidirectional tape by the integers:

$$\dots, -3, -2, -1, 0, 1, 2, 3, \dots,$$

with the tape head initially at cell 0. We “fold” this tape between cells -1 and 0 . Cell $j \geq 0$ of

the corresponding standard tape stores the pair

$$(a_j, a_{-j-1}),$$

where a_i denotes the symbol in cell i of M 's tape. Thus, the first few physical cells of $\widetilde{M}$'s tape represent

Cell of $\widetilde{M}$	0	1	2	3
First component	a_0	a_1	a_2	a_3
Second component	a_{-1}	a_{-2}	a_{-3}	a_{-4} .

The first components encode the nonnegative half of M 's tape in its usual order, while the second components encode the negative half in reverse order. Again, see [Figure 1.3](#) for an illustration.

If M has tape alphabet Γ , then each encoded cell contains an element of

$$\Gamma \times \Gamma.$$

The simulator therefore uses Γ^2 , together with a constant number of special bookkeeping symbols, as part of its tape alphabet. This is still a finite alphabet.

The finite control of $\widetilde{M}$ records, for each simulated tape, which component is currently active:

$$\mathbf{P} \quad \text{or} \quad \mathbf{N}.$$

Mode **P** means that the simulated head is on the nonnegative half of the tape and is therefore reading the first component. Mode **N** means that it is on the negative half and is reading the second component. The physical head of $\widetilde{M}$ always scans the cell containing the currently simulated tape cell.

Suppose first that the simulator is in mode **P** and its physical head is at position j . It is then simulating the head of M at position j . Its movements are translated as follows:

Move by M	Move by $\widetilde{M}$	New mode
R	R	P
$L, j > 0$	L	P
$L, j = 0$	remain at cell 0	N .

The last line represents crossing the fold: moving left from simulated cell 0 leads to simulated cell -1 , and both cells are represented in physical cell 0.

Now suppose that the simulator is in mode **N** and its physical head is at position j . It is then simulating the head of M at position $-j-1$. Since the negative half was stored in reverse order, left and right movements are interchanged:

Move by M	Move by $\widetilde{M}$	New mode
L	R	N
$R, j > 0$	L	N
$R, j = 0$	remain at cell 0	P .

The final line represents crossing the fold from simulated cell -1 to simulated cell 0.

The leftmost physical cell can be given a specially marked version of the paired alphabet, so $\widetilde{M}$ can recognize when its head is at the fold.

In summary, on the nonnegative half, head movements are simulated normally; On the negative half, left and right movements are reversed because that half is stored in reverse order. When the simulated head crosses between positions 0 and -1 , the physical head remains at the leftmost cell, and the simulator switches between the two components.

If M has tape alphabet Γ , the simulator uses paired symbols from Γ^2 . Each transition of

M can therefore be simulated using only a constant number of transitions: the simulator reads and updates the appropriate component of the pair and moves its head as described above. The same construction is applied independently to each tape of M . Therefore, a $T(n)$ -step computation of M is simulated in $O(T(n))$ time. Hence, the resulting one-way-tape machine computes the same function as M . ■

What robustness does and does not say

These simulations establish that the models compute the same functions. They do not say that every choice of model gives exactly the same running time. For example, a direct three-tape palindrome algorithm runs in linear time, while some restricted single-tape implementations may require substantially more time.

Reasonable variants of the Turing-machine model agree on computability and usually agree on broad complexity distinctions such as polynomial versus exponential time. This robustness is one reason Turing machines are useful as a foundation for complexity theory.

1.4 Universal Turing Machines

1.4.1 Machines as Data

A Turing machine has a finite description: it has a finite alphabet, finitely many states, and a finite transition table.² Therefore, the description of a machine can itself be encoded as a binary string. This simple observation allows one machine to receive another machine as input and simulate it.

Let M be a Turing machine. We write

$$\langle M \rangle \in \{0, 1\}^*$$

for a binary encoding of M . The encoding contains enough information to recover:

- the tape alphabet and an encoding of each tape symbol;
- the states, including the start and halting states;
- the number and roles of the tapes; and
- every entry of the transition function.

Because all these sets are finite, such an encoding can be constructed using ordinary binary representations and delimiters.

Definition 1.4.1 (Machine Represented by a String). *For a binary string α , let M_α denote the Turing machine represented by α . Strings that are not valid encodings may be assigned to a fixed default machine, such as a machine that immediately outputs 0 and halts. ◇*

This convention ensures that every binary string denotes some machine. It is also convenient to permit harmless padding, so one machine may have more than one valid encoding.

²Note that the infinite tape is not part of the *description*; only the sentence that “a Turing machine has an infinite tape” is part of the description; this sentence has finite number of words, and thus can be encoded as a finite string.

Remark 1.4.1 (Programs as Data). *The distinction between program and data is not absolute. A machine description is data when it is read by another machine, but it specifies a program when interpreted as a transition table. This principle underlies interpreters, compilers, stored-program computers, and many later arguments in complexity theory.*

1.4.2 Definition and Basic Simulation

Definition 1.4.2 (Universal Turing Machine). *A universal Turing machine is a fixed Turing machine U such that, for every machine description α and every input x ,*

$$U(\alpha, x) = M_\alpha(x), \quad (1.5)$$

whenever $M_\alpha(x)$ halts. If $M_\alpha(x)$ does not halt, then the simulation by U also need not halt. $\diamond$

The notation $U(\alpha, x)$ means that U receives an encoding of the pair (α, x) . A standard delimiter or self-delimiting encoding can be used to distinguish the two components.

The universal machine maintains encoded versions of:

- the simulated machine's transition table;
- the simulated machine's current state;
- the contents of its tapes; and
- the positions of its tape heads.

To simulate one transition, U reads the simulated state and scanned symbols, looks up the appropriate transition in the encoded table, and updates the simulated configuration.

Analogy: Virtual Machines

A useful real-world analogy is a *virtual machine*. A virtual machine (e.g., a VMWare® instance) may behave like a complete computer, with its own processor, memory, disks, and operating system. Nevertheless, everything needed to specify it—its configuration, virtual hardware, and software—can be stored as finite data on a physical computer.

A physical computer can read this description and simulate the virtual computer step by step. The same physical computer can simulate many different virtual machines; which machine it simulates depends on the description it is given. Similarly, a universal Turing machine can take an encoding $\langle M \rangle$ as input and simulate the particular Turing machine M described by that encoding.

This analogy also illustrates the idea of *programs as data*. To the host computer, a virtual-machine image is simply a file containing bits. When those bits are interpreted by virtualization software, however, they describe a computer and determine its behavior. In the same way, $\langle M \rangle$ is an ordinary binary string when treated as input, but it becomes a program when interpreted as the description of a Turing machine.

1.4.3 Efficient Universal Simulation

The existence of a universal Turing machine does not immediately imply that the simulation is efficient. In this subsection, we discuss the accurate cost of simulating a machine M_α on input x using a universal machine U . We will first show that a naive simulation may introduce quadratic overhead, and then we will present a more efficient simulation that incurs only logarithmic overhead.

Theorem 1.4.1 (Quadratic Universal Turing Machine). *There exists a fixed Turing machine U such that, for every $\alpha, x \in \{0, 1\}^*$,*

$$U(\alpha, x) = M_\alpha(x)$$

whenever $M_\alpha(x)$ halts.

Furthermore, if M_α halts on x within T steps, then U halts within

$$C_\alpha T^2 \tag{1.6}$$

steps, where C_α depends on the fixed machine M_α , including its number of states, alphabet size, and number of tapes, but does not depend on x or T . $\diamond$

Before proving the theorem, let us emphasize its main strength: the input machine M_α is completely arbitrary. For example, M_α may have any fixed number of tapes and may use any finite tape alphabet. In contrast, the universal machine U is fixed once and for all; indeed, U may be taken to be a standard three-tape Turing machine with binary tape alphabet. Nevertheless, U can simulate every machine M_α with at most quadratic overhead in its running time. Thus, although the constants involved may depend on the description α , the quadratic form of the bound is uniform over all input machines.

Proof of Theorem 1.4.1. We construct the required universal Turing machine U . In fact, U will be a standard three-tape Turing machine with binary tape alphabet $\{0, 1\}$.

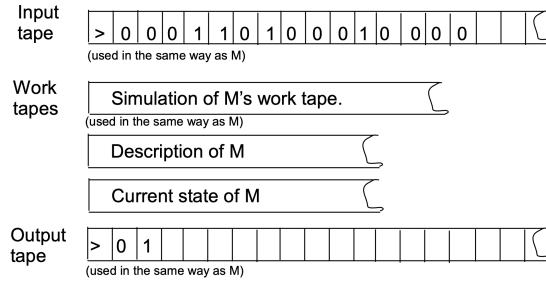


Figure 1.4: A fixed universal Turing machine U simulating an arbitrary machine M_α . The machine U is a standard three-tape Turing machine with binary tape alphabet, whereas M_α may have any fixed number of tapes and any finite alphabet. Adapted from [AB09, Figure 1.6].

The basic idea is straightforward. Given the description α of a Turing machine M_α and an input x , the machine U maintains an encoding of the current configuration of M_α . It then repeatedly consults α to determine the next transition and updates the encoded configuration accordingly.

This process is illustrated in Figure 1.4, where the “Description of M ” and “Current state of M ” segments can be placed at the beginning of the work tape (immediately preceding the “Simulation of M ’s work tape” segment), and occupy only a constant number of cells.

The main point is that this simulation remains efficient even though M_α may have multiple tapes and an arbitrary finite alphabet. This follows from two observations:

1. Let Γ be the tape alphabet of M_α . Each symbol of Γ can be represented by a binary block of length

$$\lceil \log_2 |\Gamma| \rceil.$$

Since Γ is fixed once M_α is fixed, operations on these blocks incur only constant overhead relative to the running time of M_α .

2. Suppose that M_α has k tapes. Its k tapes can be encoded on a single tape by placing their contents on separate tracks and marking the locations of their heads. As in the proof of [Lemma 1.3.2](#), one transition of M_α can be simulated by scanning the portion of the encoded tape visited so far. After t steps, this portion has length $O(t)$, so simulating the first T transitions requires

$$\sum_{t=1}^T O(t) = O(T^2)$$

time. Although k may be arbitrarily large across different machines, it is a constant for each fixed machine M_α .

The universal machine U performs these operations directly from the binary description α . Consequently, if M_α halts on x within T steps, then U simulates the computation and produces the same output in $O(T^2)$ time, with constants that may depend on α .

This finishes the proof of [Theorem 1.4.1](#). ■

By a more involved construction, one can achieve a more efficient simulation of a universal Turing machine. The following [Theorem 1.4.2](#) shows that the overhead can be reduced to $O(T \log T)$ time. However, the proof of this theorem is much more technical and out of the scope of this undergraduate-level course. We present the proof in [Sections 1.4.4](#) and [1.4.5](#) as optional readings for interested students.

Theorem 1.4.2 (Efficient Universal Turing Machine). *There exists a fixed Turing machine U such that, for every $\alpha, x \in \{0, 1\}^*$,*

$$U(\alpha, x) = M_\alpha(x)$$

whenever $M_\alpha(x)$ halts.

Furthermore, if M_α halts on x within $T \geq 2$ steps, then U halts within

$$C_\alpha T \log T \tag{1.7}$$

steps, where C_α depends on the fixed machine M_α , including its number of states, alphabet size, and number of tapes, but does not depend on x or T . ◇

1.4.4 *Proof of [Theorem 1.4.2](#) (Part 1/2)

Nothing from [Section 1.4.4](#) will be tested on quizzes or exams!

Before studying the $O(T \log T)$ -time universal Turing machine simulation, we examine two simpler examples involving powers of two. They illustrate two related but distinct ideas:

- Organizing a computation into exponentially increasing scales produces only logarithmically many levels.
- An operation can be expensive occasionally without making a long sequence of operations expensive on average.

Repeated squaring illustrates the first idea. A binary counter illustrates the second.

These ideas will become crucial in the real proof as we will show in [Section 1.4.5](#), where we use them in a more complicated way than in these examples. Thus, we believe it is helpful to understand them in a simpler context first, as we will do now in the current subsection.

1.4.4.1 Example 1: Computing powers by repeated squaring

Suppose we want to compute a^n , where n is a nonnegative integer. Multiplying a by itself repeatedly takes $n - 1$ multiplications when $n \geq 1$. Can we use fewer multiplications?

The key identities are

$$a^{2m} = (a^m)^2, \quad a^{2m+1} = a(a^m)^2.$$

Thus, to compute a^n , we can first compute $a^{\lfloor n/2 \rfloor}$, square the result, and multiply by a once more if n is odd.

More explicitly, the following recursive procedure computes a^n :

1. If $n = 0$, return 1.
2. If $n = 1$, return a .
3. Recursively compute

$$y = a^{\lfloor n/2 \rfloor}.$$

4. If n is even, return $y \cdot y$. If n is odd, return $a \cdot (y \cdot y)$.

The important point is that we compute y only once and then reuse it.

For example, to compute a^{13} , we can calculate

$$a^2 = a \cdot a, \quad a^3 = a \cdot a^2, \quad a^6 = a^3 \cdot a^3, \quad a^{12} = a^6 \cdot a^6, \quad a^{13} = a \cdot a^{12}.$$

This uses five multiplications rather than twelve.

Why the number of multiplications is logarithmic. Each recursive call roughly halves the exponent:

$$n, \quad \left\lfloor \frac{n}{2} \right\rfloor, \quad \left\lfloor \frac{n}{4} \right\rfloor, \quad \dots, \quad 1.$$

For $n \geq 1$, there are $\lfloor \log_2 n \rfloor$ reductions before reaching 1, and each reduction contributes at most two multiplications. Hence the procedure uses

$$O(\log(n + 1))$$

multiplications.

An equivalent way to understand the method is through the binary representation of the exponent. For instance,

$$13 = 8 + 4 + 1, \quad \text{so} \quad a^{13} = a^8 a^4 a.$$

Repeated squaring constructs

$$a, \quad a^2, \quad a^4, \quad a^8, \quad \dots,$$

and the binary digits of n tell us which powers to multiply together.

A warning about the cost model. This is a bound on the *number of multiplications*, not necessarily on the number of Turing machine steps. Multiplication is not a constant-time operation when its operands are arbitrarily long integers. For example, if $a \geq 2$ is a fixed integer, the binary representation of a^n already has $\Theta(n)$ bits.

The lesson we want here is structural:

Doubling the scale at each level gives logarithmically many levels.

1.4.4.2 Example 2: Incrementing a binary counter

Consider a binary counter initially equal to 0. We increment it T times, where $T \geq 1$. Allocate

$$b = \lceil \log_2(T + 1) \rceil$$

bits, which is enough to represent every value from 0 through T .

To increment the counter, start at the least significant bit:

1. Change each consecutive 1 into a 0, carrying to the next bit.
2. Change the first 0 into a 1, and stop.

For example,

$$0111 \longrightarrow 1000$$

changes four bits, whereas

$$1000 \longrightarrow 1001$$

changes only one.

We measure the cost of an increment by the number of bits it changes.

A worst-case bound is not the whole story. One increment can change all $b = O(\log(T + 1))$ bits. Multiplying this worst-case cost by the number of increments gives the valid upper bound

$$O(T \log(T + 1)).$$

However, this overestimates the total cost: increments with long carry chains are rare.

To obtain a sharper bound, we count how often each bit changes over the entire sequence.

Number the bits $0, 1, \dots, b - 1$, starting with the least significant bit. Then:

- Bit 0 changes on every increment.
- Bit 1 changes on every second increment.
- Bit 2 changes on every fourth increment.
- In general, bit i changes on every 2^i -th increment.

Consequently, the total number of bit changes is

$$\begin{aligned} \sum_{i=0}^{b-1} \left\lfloor \frac{T}{2^i} \right\rfloor &\leq T \sum_{i=0}^{b-1} \frac{1}{2^i} \\ &< T \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2T. \end{aligned}$$

Thus all T increments together cost $O(T)$, even though a single increment can cost $\Theta(\log(T + 1))$.

What amortized analysis means. We say that incrementing the counter has $O(1)$ *amortized cost*: the total cost of T increments, starting from zero, is $O(T)$.

This is not a probabilistic statement. We are not assuming that the counter values are random, nor that every increment is cheap. Instead, we prove a bound on the total cost of the entire sequence.

The useful accounting principle is

$$\text{Total cost} = \sum_{\text{levels}} (\text{cost per event}) (\text{number of events}).$$

For the counter, a level- i event is a change to bit i . It costs 1 and occurs at most $T/2^i$ times.

Connecting these examples to tape simulation. As we will see in [Section 1.4.5](#), the universal Turing machine simulation uses buffers whose sizes grow geometrically:

$$2, \quad 4, \quad 8, \quad 16, \quad \dots$$

As in repeated squaring, there are only $O(\log(T+1))$ relevant levels during a computation of T steps.

As with a binary counter, activity at high levels is infrequent. The buffer construction will ensure that reorganizing level i costs $O(2^i)$, but occurs only $O(T/2^i)$ times. Therefore, each level contributes at most

$$O(2^i) \cdot O(T/2^i) = O(T)$$

work. Summing over all relevant levels gives

$$O(T) \cdot O(\log(T+1)) = O(T \log(T+1)).$$

Notice the difference from the binary counter: changing a high-order bit costs only 1, whereas reorganizing a large buffer costs proportionally to its size. This is why the counter has $O(T)$ total cost, while the buffer simulation has an $O(T \log(T+1))$ bound.

The proof that follows makes this accounting argument precise by constructing the buffers and showing why a large rearrangement cannot happen again until sufficiently many simulated head moves have occurred.

1.4.5 *Proof of [Theorem 1.4.2](#) (Part 2/2)

Nothing from [Section 1.4.5](#) will be tested on quizzes or exams!

We now proceed to the real proof of [Theorem 1.4.2](#).

We use the input/output-tape convention of the theorem: U simulates M 's input and output tapes directly. The main task is to simulate M 's k work tapes using a fixed number of work tapes.

1.4.5.1 The main idea: move the contents, not the simulated heads

Let Γ be the work-tape alphabet of M . We view one work tape of U as having k parallel *tracks*. A cell of this tape stores one symbol for each track, so it represents a tuple of k symbols.

These tracks are not independent physical tapes: they share a single physical head. Nevertheless, we will arrange that, between simulated steps, the symbol under each of M 's work-tape heads is stored at position 0 of its corresponding track.

Thus, when a simulated head moves, we move the represented tape contents relative to position 0:

- A simulated move to the right brings the next symbol on the right into position 0.
- A simulated move to the left brings the next symbol on the left into position 0.

If we physically shifted an entire track after every move, one simulated step could cost $O(T)$, giving an $O(T^2)$ simulation. Instead, we introduce unused space into each track and use it as a collection of buffers.

Implementation convention. For readability, we describe cells as storing tuples from an alphabet that depends on M . The actual universal machine has a fixed alphabet: it encodes each such tuple by a block of fixed-alphabet symbols. For a fixed M , the block length is constant, so this changes the running time by only a factor depending on M .

1.4.5.2 Buffers and their meaning

Introduce a padding symbol $\diamond$, distinct from every symbol of Γ , including the ordinary tape blank.

The distinction is important:

- An ordinary blank represents an actual cell of M 's tape.
- A $\diamond$ represents unused storage and is ignored.

For example, the strings

$$010, \quad 0 \diamond 10, \quad 0 \diamond \diamond 1 \diamond 0$$

represent the same sequence of simulated tape symbols.

We use a two-sided physical tape, with positions indexed by the integers. Position 0 is the *home position*. For each $i \geq 1$, define a right zone and a left zone by

$$R_i = \{2^i - 1, \dots, 2^{i+1} - 2\}, \quad L_i = \{-p : p \in R_i\}.$$

Both zones have 2^i cells. The layout is therefore

$$\cdots L_3 L_2 L_1 \boxed{0} R_1 R_2 R_3 \cdots,$$

where the zone sizes, moving outward from the center, are

$$2, 4, 8, 16, \dots$$

Starting the indexing at 1 ensures that every zone has an integer half-capacity.

For each track separately, we maintain these invariants:

1. Position 0 contains the symbol currently under the corresponding head of M .
2. Reading the track and ignoring all $\diamond$'s gives the simulated tape contents, in the correct order.
3. Each zone of size 2^i is either empty, half full, or full of non-padding symbols.
4. The total number of non-padding symbols in L_i and R_i is exactly 2^i .

The last two conditions mean that the occupancies of L_i and R_i must be one of

$$(0, 2^i), \quad (2^{i-1}, 2^{i-1}), \quad (2^i, 0).$$

Initially, every zone is conceptually half full of ordinary blank symbols, and position 0 also contains a blank. There is no need to initialize infinitely many cells: a zone is initialized only when it is first reached. Beyond the initialized region, all simulated symbols are still blank.

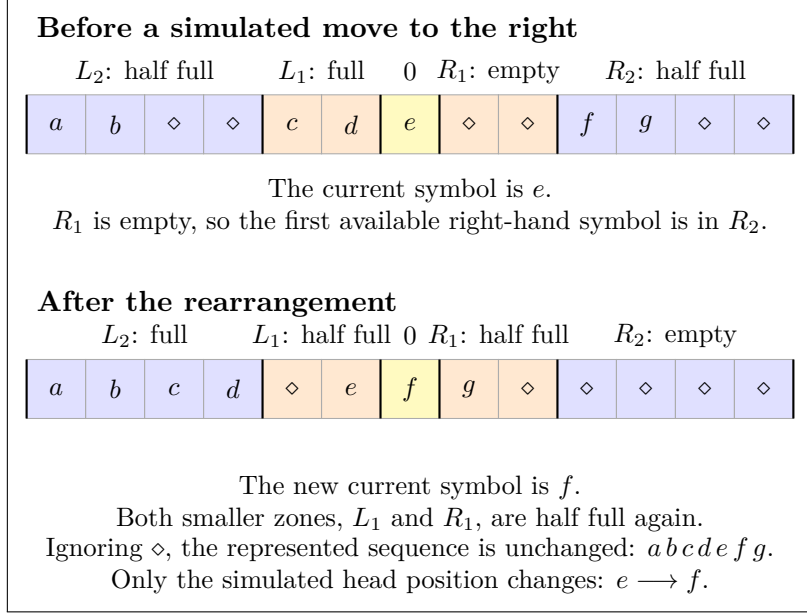


Figure 1.5: A rightward head move on one track, assuming the transition does not change the symbol e . The simulator brings f to position 0, places g in R_1 , moves c, d into L_2 , and leaves e in L_1 . The symbol ◇ denotes unused storage, not an ordinary tape blank. Only zones through level 2 are rearranged; all farther zones are unchanged.

1.4.5.3 Simulating one transition

At the beginning of a simulated step, U 's physical head is at position 0. It reads the tuple of current work-tape symbols, consults M 's transition table, and writes the updated symbols at position 0.

It then handles the prescribed head movement on each track separately. A stationary head requires no rearrangement. We explain a move to the right (see also Figure 1.5 for an illustration); a move to the left is symmetric.

1.4.5.4 Simulating a move to the right

A move to the right must bring the nearest represented symbol on the right into position 0, while placing the old position-0 symbol on the left.

Find the smallest i such that R_i is nonempty. Then

$$R_1, \dots, R_{i-1} \text{ are empty,}$$

and, by the paired-occupancy invariant,

$$L_1, \dots, L_{i-1} \text{ are full.}$$

Write

$$h = 2^{i-1}.$$

The total capacity of the smaller zones on either side is

$$\sum_{j=1}^{i-1} 2^j = 2^i - 2 = 2h - 2,$$

and their total half-capacity is

$$\sum_{j=1}^{i-1} 2^{j-1} = h - 1.$$

We now rearrange only

$$L_i, \dots, L_1, 0, R_1, \dots, R_i.$$

Right side. Take the first h non-padding symbols of R_i , in order starting from the end nearest position 0.

- Put the first symbol at position 0.
- Distribute the remaining $h - 1$ symbols among $R_1, \dots, R_{i-1}$, making each of these zones half full.

Preserve the order of the represented symbols. The occupancy of R_i decreases by h .

Left side. The smaller left zones contain $2h - 2$ symbols. Together with the old symbol at position 0, there are $2h - 1$ symbols to place on the left.

- Move the h symbols farthest from the new head position into L_i , placing them after its existing symbols in left-to-right order.
- Distribute the remaining $h - 1$ symbols among $L_1, \dots, L_{i-1}$, making each of these zones half full.

Again, preserve the order of all represented symbols. The occupancy of L_i increases by h .

Why the invariants are preserved. Before the rearrangement, the occupancies of (L_i, R_i) were either

$$(h, h) \quad \text{or} \quad (0, 2h),$$

because R_i was nonempty. Afterward, they are respectively

$$(2h, 0) \quad \text{or} \quad (h, h).$$

Thus the allowed occupancies and their sum are preserved.

All smaller zones are now half full. Position 0 contains the next simulated tape symbol, and the old position-0 symbol is immediately to its left in the represented sequence. Consequently, this rearrangement correctly simulates one rightward head move.

For $i = 1$, there are no smaller zones. The operation simply transfers the old central symbol into L_1 and the nearest symbol of R_1 into the center.

1.4.5.5 The cost of one rearrangement

Call an operation whose outermost affected zones are L_i, R_i a *level- i rearrangement*.

Its entire affected region has length

$$1 + 2 \sum_{j=1}^i 2^j = O(2^i).$$

Using a scratch tape, U can copy the represented symbols in order and redistribute them with a constant number of scans of this region. Zone boundaries and half-capacities can be marked on auxiliary tracks.

Therefore, a level- i rearrangement costs

$$O_M(2^i).$$

This includes returning the physical head to position 0. Initializing a newly reached zone also fits within this bound.

The crucial point is that expensive rearrangements occur rarely.

1.4.5.6 Why large rearrangements are rare

Fix one track and a level i . Consider any rearrangement that reaches level i or farther.

Immediately afterward, every smaller zone

$$L_1, R_1, \dots, L_{i-1}, R_{i-1}$$

is half full. Hence the smaller zones contain exactly

$$s_i = \sum_{j=1}^{i-1} 2^{j-1} = 2^{i-1} - 1$$

represented symbols on each side of position 0.

Until another rearrangement reaches level i , each rightward move decreases the number of symbols in the smaller right zones by one, and each leftward move increases it by one. The analogous statement holds on the left.

To reach level i again, all smaller zones on one side must first be empty. Starting with s_i symbols on each side, this requires at least s_i moves, followed by the move that actually reaches level i . Thus two rearrangements reaching level i or farther are separated by at least

$$s_i + 1 = 2^{i-1}$$

head moves.

The same lower bound applies before the first such rearrangement, because initially all zones are half full.

Since a work-tape head makes at most T moves during T steps, the number of rearrangements reaching level i or farther is at most

$$\left\lfloor \frac{T}{2^{i-1}} \right\rfloor.$$

In particular, this also bounds the number of level- i rearrangements.

Moreover, a level can be reached only if $2^{i-1} \leq T$. Thus the largest possible level is

$$1 + \lfloor \log_2 T \rfloor.$$

1.4.5.7 Summing the costs

For one track, the total rearrangement time is at most

$$\begin{aligned} \sum_{i=1}^{1+\lfloor \log_2 T \rfloor} O_M(2^i) \left\lfloor \frac{T}{2^{i-1}} \right\rfloor &\leq O_M \left(\sum_{i=1}^{1+\lfloor \log_2 T \rfloor} 2^i \frac{T}{2^{i-1}} \right) \\ &= O_M \left(\sum_{i=1}^{1+\lfloor \log_2 T \rfloor} 2T \right) \\ &= O_M(T \log(T+1)). \end{aligned}$$

There are k tracks, where k depends only on M , so their combined cost has the same asymptotic bound. Reading the transition table, updating the state, and handling the input and output tapes add only $O_M(T)$ time.

Therefore U correctly simulates M within

$$O_M(T \log(T+1))$$

steps.

The idea to remember. A buffer at level i costs $O(2^i)$ to reorganize, but it is involved only once per $\Omega(2^i)$ simulated head moves. Each level therefore contributes only $O(T)$ total work. There are $O(\log T)$ levels, giving $O(T \log T)$ time overall.

This finishes the proof of [Theorem 1.4.2](#).

Takeaway

A single fixed machine can simulate every other Turing machine with only modest overhead. Thus, computational universality does not require building a different physical device for every algorithm; the algorithm itself can be supplied as encoded data.

1.5 The Church–Turing Thesis

We have shown that several formal models can simulate one another, but a broader question remains: does the Turing-machine model capture every computation that could be carried out by any effective mechanical procedure? The Church–Turing thesis proposes that it does.

1.5.1 The Standard Church–Turing Thesis

Church–Turing thesis: Every function that can be computed by a physically realizable computational device can be computed by a Turing machine.

The phrase *a physically realizable computational device* refers informally to a finite collection of unambiguous rules that can be followed step by step without requiring creativity, insight, or infinitely precise information.

Two important requirements of an effective mechanical procedure are *precision* and *executability*.

- *Precision:* Every instruction must specify exactly what is to be done. For example, a cooking recipe may instruct the reader to add “a pinch of salt” or “sugar to taste.” Such instructions may be perfectly adequate for cooking, but they are not mechanically precise:

different people may interpret them differently. By contrast, an instruction such as “add 5 grams of salt” specifies a definite quantity.

- *Executability*: Every instruction must also describe an operation that can actually be carried out by the intended computational agent. Consider the familiar joke:

How do you put an elephant into a refrigerator?

Answer: In three steps: (1) Open the door, (2) put the elephant inside, and (3) close the door.

The instructions are short and appear unambiguous, but the crucial step—putting the elephant into the refrigerator—may be physically impossible. Merely describing an action does not make that action executable.

Thus, an effective mechanical procedure must consist of instructions that are both sufficiently precise to determine each step uniquely and sufficiently elementary to be carried out mechanically.

It is only a conjecture! The Church–Turing thesis is not a theorem. One side of the thesis refers to the informal notion of an effective procedure, so the statement cannot be proved using only formal mathematics. It is a hypothesis connecting an informal concept to a formal model.

Evidence for the thesis includes the following:

- Many independently proposed formal models of computation, including Turing machines, lambda calculus, and general recursive functions, define the same class of computable functions.
- Standard programming languages can simulate Turing machines, and Turing machines can simulate programs written in standard programming languages.
- Natural changes to the details of the Turing-machine model do not change the class of computable functions.
- No generally accepted effective procedure has been discovered that computes a function uncomputable by a Turing machine.

Implications for the limits of computation

If we accept the standard Church–Turing thesis, then a proof that no Turing machine computes a particular function establishes a fundamental limit. The conclusion is not merely that present computers lack sufficient memory or that current programmers have not found the correct algorithm. Rather, no algorithmic procedure can compute that function on every input.

This distinction will become important when we study undecidability. Some problems are difficult because known algorithms use many resources. Other problems are uncomputable: no Turing-machine algorithm exists at all.

1.5.2 The Extended Church–Turing Thesis

The standard thesis concerns what is computable, without requiring the simulation to be efficient. Complexity theory motivates a stronger statement.

Extended Church–Turing thesis: Every computation carried out by a physically realizable computational device can be simulated efficiently, usually meaning with at most polynomial overhead, by a probabilistic Turing machine.

The addition of “efficiently” is crucial. An exponential-time simulation still supports the standard Church–Turing thesis, because the same function remains computable. It may nevertheless contradict the extended thesis because exponential overhead is not considered efficient.

Thesis	Main question	Required simulation
Standard	What can be computed?	Any finite overhead is allowed
Extended	What can be computed efficiently?	Polynomial overhead

Table 1.2: The standard and extended Church–Turing theses make different claims. The extended thesis is strictly stronger because it imposes an efficiency requirement.

1.5.3 Quantum Computing and the Extended Thesis

Quantum computers are physical models whose evolution is governed by quantum mechanics. They can be formalized by quantum circuits or quantum Turing machines.

A classical Turing machine can simulate any finite quantum computation to any required finite precision. The classical simulation may, however, require exponential time because a general state of m quantum bits is described using 2^m complex amplitudes.

Let:

- BPP denote the class of decision problems solvable in polynomial time by a “probabilistic” (we will discuss this in later chapters) classical Turing machine with bounded error; and
- BQP denote the class of decision problems solvable in polynomial time by a quantum computer with bounded error.

If quantum computers can efficiently solve a problem that no probabilistic classical Turing machine can solve efficiently, then

$$\text{BQP} \not\subseteq \text{BPP}.$$

Such a separation would contradict the classical probabilistic version of the extended Church–Turing thesis.

It has not been proved that $\text{BQP} \not\subseteq \text{BPP}$. Quantum algorithms provide strong evidence that quantum computation may violate the extended Church–Turing thesis, but this conclusion remains conditional on unresolved complexity-theoretic questions.

Quantum computing does not, according to the standard formal models, violate the standard Church–Turing thesis. A quantum computation can be simulated by a classical Turing machine, although possibly with exponential overhead. It changes our understanding of efficient computation, not the class of functions that are computable in principle.

Remark 1.5.1 (What Quantum Computing Challenges). *Quantum computing challenges the claim that every reasonable physical computation has an efficient classical simulation. It does not provide a method for solving Turing-uncomputable problems. Thus, it is evidence against some formulations of the extended thesis, not against the standard thesis.*

1.5.4 The Turing Machine as a Final Model

If we accept the standard Church–Turing thesis, then introducing a more realistic programming language, a different processor, more tapes, random-access memory, parallel hardware, or quantum mechanics does not enlarge the class of computable functions beyond Turing computability.

This does not mean that all models are equally convenient or equally efficient:

- Different programming languages may make different algorithms easier to express.
- Different machine models may change running time by polynomial or even exponential factors.
- Randomness, parallelism, and quantum computation may change which problems can be solved efficiently.
- Restricted models such as finite automata remain valuable when limited memory is part of the problem being studied.

The main point is more fundamental: under the standard Church–Turing thesis, changing to another physically realizable model does not allow us to compute a genuinely uncomputable function.

If the standard Church-Turing thesis is correct, then the Turing machine is an eventual final model of computability: future models may alter convenience or efficiency, but they will not enlarge the set of functions that are computable by effective procedures.

1.6 Closing Remarks

We emphasize that Turing machines are not the only mathematical model of computation. Other models include the lambda calculus, recursive functions, register machines, and various idealized programming languages. Although these models describe computation in different ways, they are *equivalent* at the level of computability: each can compute exactly the functions computable by a Turing machine. Thus, the Turing machine should not be regarded as the uniquely correct description of computation, but as one particularly useful formalization among several equivalent ones.

We focus on Turing machines because their operation is concrete and easy to visualize: a machine reads and writes symbols, moves its tape heads, and changes its finite state. These elementary actions make it possible to describe algorithms precisely and to measure resources such as running time and memory usage. Moreover, the distinction between the finite control and the unbounded tape closely reflects the familiar distinction between a fixed program and its working memory. Turing machines therefore provide a simple, rigorous, and convenient foundation for studying computability and computational complexity, even though other equivalent models could serve the same fundamental purpose.

We also emphasize that we do not expect you to memorize every detail of the formal definition of a Turing machine or its transition rules. Rather, the purpose of studying Turing machines is to see how computation can be described with mathematical precision by reducing it to a small collection of elementary operations. Much like building with LEGO bricks, each natural step of an algorithm can be decomposed into finitely many simple Turing-machine steps. Despite its simplicity, this model has proved remarkably robust: every standard, reasonable model of general-purpose computation developed so far can compute the same class of functions as a Turing machine. This observation provides the main evidence for the Church-Turing thesis, which asserts that anything computable by an effective algorithm can be computed by a Turing machine. Thus, Turing machines give us a simple and implementation-independent foundation for reasoning rigorously about what can be computed and how efficiently it can be computed.

Chapter 2

Deterministic Polynomial Time: P

Our goal in this chapter is to make precise one of the central ideas of computational complexity: a problem is efficiently solvable if it has a polynomial-time algorithm. Building on our computational model and the notion of running time, we will define the class P and establish that several graph and number-theoretic decision problems belong to it.

We will then examine why complexity theory places so much emphasis on decision problems, even though applications often ask us to find actual solutions. For many natural problems, suitable decision queries allow us to construct a solution. However, this relationship depends on exactly which decision problem we are allowed to ask.

2.1 Polynomial Time and the Class P

Let us begin by connecting running-time analysis to a class of decision problems. The main issues are the choice of polynomial time as a benchmark, the role of input encodings, and the requirement that the algorithm be deterministic.

2.1.1 Why Polynomial Time?

A useful dividing line

An algorithm runs in polynomial time if its worst-case running time is bounded by a polynomial in the input length. Typical polynomial bounds include $O(n)$, $O(n^2)$, and $O(n^5)$, where n denotes the number of symbols in the input.

Polynomial and exponential functions behave very differently as the input length grows. For example, when $n = 100$,

$$n^3 = 1,000,000, \quad 2^n \approx 1.27 \times 10^{30}.$$

This comparison does not tell us the actual running time of a particular implementation. It illustrates why replacing exhaustive search by a polynomial-time method can fundamentally change which input sizes are accessible.

Exponential running times often arise from searching through all possible solutions. If an instance involves n independent binary choices, there are 2^n possible combinations. An efficient algorithm must frequently avoid examining them one by one.

Why not choose a particular polynomial?

Why do we group n^2 and n^5 together, instead of defining efficiency using one particular exponent?

One reason is that polynomial time is stable under several basic operations.

- Running polynomially many polynomial-time subroutines still takes polynomial time, provided their inputs have polynomial length.
- A polynomial slowdown in simulating one computational model by another preserves polynomial-time solvability.

- Changing between encodings that can be translated into each other in polynomial time preserves polynomial-time solvability.

For example, standard deterministic single-tape and multitape Turing machines can simulate one another with polynomial overhead. Their detailed running-time bounds may differ, but they define the same class of polynomial-time decision problems.

Polynomial time is a theoretical benchmark, not a guarantee of practical speed. A running time of n^{100} is polynomial but usually impractical. Conversely, an exponential-time algorithm can be useful on small inputs or restricted families of instances. We use polynomial time to study a robust structural question, not to replace detailed algorithm analysis.

2.1.2 The Formal Definition

Recall that a *decision problem* asks for a yes/no answer. We represent such a problem by a language (see [Definition 1.2.4](#)): the set of encodings of its yes-instances.

Definition 2.1.1 (The class P). *The class P consists of all languages decidable in polynomial time by a deterministic Turing machine. Equivalently,*

$$P := \bigcup_{k \geq 1} \text{TIME}(n^k), \quad (2.1)$$

where k ranges over positive integers and $\text{TIME}(n^k)$ denotes the languages decidable by deterministic machines in time $O(n^k)$. $\diamond$

Let us unpack [Definition 2.1.1](#). A language L belongs to P if there exist a deterministic machine M , a constant $C > 0$, and a fixed positive integer k such that, for every input string x ,

- M accepts x if and only if $x \in L$;
- M halts within $C(|x| + 1)^k$ steps.

Here $|x|$ is the length of x . The constants C and k may depend on the machine, but they must not depend on the particular input.

There must be one algorithm that works for every input. We cannot choose a different algorithm or a different exponent separately for each instance.

Example 2.1.1 (Recognizing a polynomial bound). *Suppose a deterministic decider has running time*

$$T(n) \leq 5n^3 + 12n + 7$$

for inputs of length n . For $n \geq 1$,

$$\begin{aligned} T(n) &\leq 5n^3 + 12n^3 + 7n^3 \\ &= 24n^3. \end{aligned}$$

The finitely many shorter inputs can be absorbed into the constant in the running-time bound. Therefore, the language decided by this machine belongs to P.

The class P contains decision problems for which there exists a deterministic algorithm that is correct on every input and runs in worst-case polynomial time.

Problems versus algorithms. A language is not itself deterministic or randomized; those words describe algorithms. Saying that $L \in P$ means that at least one deterministic polynomial-time decider for L exists. The same language might also have other algorithms, including slower algorithms or randomized algorithms.

2.1.3 Input Encodings and Bit Complexity

An algorithm's running time is measured against the length of its encoded input, not against the numerical value represented by that input. Consequently, changing the encoding can change whether a running-time bound is polynomial in the input length.

To see this, consider two encodings of the same positive integer N :

- **Binary encoding:** Write N in base two. The input length is

$$\ell(N) := \lfloor \log_2 N \rfloor + 1. \quad (2.2)$$

- **Unary encoding:** Write N as a string of N copies of the symbol 1. The input length is N .

Both strings represent the same integer, but their lengths can differ exponentially.

Example 2.1.2 (The same loop under two encodings). *Consider an algorithm that performs a loop exactly N times. For*

$$N = 2^{20} = 1,048,576,$$

the binary input has only 21 symbols, whereas the unary input has 1,048,576 symbols. The loop performs the same number of iterations in either case, but we compare that number against very different input lengths.

More generally, let n denote the length of the chosen encoding.

- *With **unary encoding**, $n = N$, so the loop performs n iterations. If each iteration and the remaining computation take polynomial time in n , the algorithm runs in polynomial time.*
- *With **binary encoding**, $n = \lfloor \log_2 N \rfloor + 1$, so*

$$2^{n-1} \leq N < 2^n.$$

The loop therefore performs exponentially many iterations in n . Even if each iteration takes only one step, this algorithm is not polynomial-time.

Thus, a loop that is polynomially bounded for unary input can be exponentially long for binary input.

Changing to unary does not make the loop perform less work. It makes the input much longer, so the benchmark against which we measure that work changes. Nor does this example rule out a different, faster algorithm for the binary-encoded problem.

Which encoding changes preserve polynomial time? Polynomial-time solvability is preserved when two encodings can be translated into each other in polynomial time. Binary and unary do not satisfy this condition: converting a binary integer N to unary requires writing N symbols, which can take exponential time in the binary input length.

For graphs, we use standard explicit representations, such as adjacency lists that include all vertices or adjacency matrices. These representations can be converted into one another in polynomial time, so choosing between them does not change whether a graph decision problem

belongs to P. In both representations, the numbers of vertices and edges are bounded by polynomials in the input length.

We do not use compressed graph representations that can describe exponentially many vertices with a short string. Such an encoding could change the complexity measured in terms of input length, just as binary encoding does in comparison with unary encoding.

2.2 Representative Decision Problems in P

We now apply the definition to three problems: relative primality, directed reachability, and bipartite matching with a size threshold. The algorithms use different ideas, but the proof structure is the same: correctness together with a polynomial running-time bound.

2.2.1 Relative Primality

For positive integers d and x , we write $d \mid x$ if $x = dq$ for some integer q . In this case, d is a *divisor* of x .

The greatest common divisor $\gcd(x, y)$ is the largest positive integer that divides both x and y . The integers x and y are *relatively prime* if $\gcd(x, y) = 1$.

Relative primality is a property of a pair of integers. Neither integer must itself be prime: for example, 10 and 21 are relatively prime.

Definition 2.2.1 (The language RELPRIME). *For positive integers encoded in binary, define*

$$\text{RELPRIME} := \{\langle x, y \rangle : \gcd(x, y) = 1\}. \quad (2.3)$$

◇

Testing every possible common divisor would take too long in the worst case, because the values of x and y can be exponential in their bit lengths. Instead, we use Euclid's algorithm.

Lemma 2.2.1 (The Euclidean identity). *For integers $a \geq 0$ and $b > 0$,*

$$\gcd(a, b) = \gcd(b, a \bmod b). \quad (2.4)$$

Here $a \bmod b$ is the remainder when a is divided by b .

◇

Proof. Write $a = qb + r$, where $0 \leq r < b$.

If an integer divides both a and b , it divides $a - qb = r$. Conversely, if it divides both b and r , it divides $qb + r = a$. Thus, the pairs (a, b) and (b, r) have exactly the same common divisors, so their greatest common divisors are equal. ■

Algorithm 2.2.1: Euclid's algorithm

On input positive integers x and y :

1. Set $a := x$ and $b := y$. If $a < b$, exchange them.
2. While $b \neq 0$, replace the pair (a, b) simultaneously by
$$(a, b) := (b, a \bmod b).$$
3. Output a .

Theorem 2.2.1. RELPRIME $\in$ P.

◇

Proof. Run [Algorithm 2.2.1](#) and accept exactly when its output is 1.

By [Lemma 2.2.1](#), every iteration preserves the gcd. The second component strictly decreases until it reaches zero. Since $\gcd(a, 0) = a$ for $a > 0$, the algorithm outputs the original gcd. This proves correctness.

To bound the number of iterations, consider a state with $a \geq b > 0$. Its remainder must satisfy:

$$a \bmod b < \frac{a}{2}.$$

This is because:

- if $b \leq a/2$, the remainder is smaller than b and hence smaller than $a/2$.
- If $b > a/2$, the quotient is 1, so the remainder is $a - b < a/2$.

After two iterations, if the algorithm has not already terminated, the first component is this remainder. Thus, the larger component decreases by at least a factor of two every two iterations. The total number of iterations is $O(\log \max\{x, y\})$, which is $O(n)$ for total input length n .

All intermediate values have at most n bits. Binary division with remainder can be performed in polynomial time; for example, a grade-school implementation uses $O(n^2)$ bit operations per division. This gives a sufficient polynomial bound of $O(n^3)$ for the complete computation. ■

Example 2.2.1 (A yes-instance and a no-instance). For $(x, y) = (55, 34)$, the successive divisions are

$$\begin{aligned} 55 &= 1 \cdot 34 + 21, \\ 34 &= 1 \cdot 21 + 13, \\ 21 &= 1 \cdot 13 + 8, \\ 13 &= 1 \cdot 8 + 5, \\ 8 &= 1 \cdot 5 + 3, \\ 5 &= 1 \cdot 3 + 2, \\ 3 &= 1 \cdot 2 + 1, \\ 2 &= 2 \cdot 1 + 0. \end{aligned}$$

The last nonzero remainder is 1, so $\gcd(55, 34) = 1$ and the decision algorithm accepts.

For $(x, y) = (24, 18)$,

$$\begin{aligned} 24 &= 1 \cdot 18 + 6, \\ 18 &= 3 \cdot 6 + 0. \end{aligned}$$

The gcd is 6, so the decision algorithm rejects.

Takeaway

Computing a gcd is a function problem: the output is an integer. The language RELPRIME is a decision problem: the output is yes or no. We established its membership in P by using the numerical computation as a subroutine.

2.2.2 Directed Reachability

A directed graph $G = (V, E)$ consists of a finite vertex set V and a set of ordered pairs $E \subseteq V \times V$. An edge (u, w) allows movement from u to w , but not necessarily from w to u .

A directed path from s to t is a sequence of distinct vertices

$$s = u_0, u_1, \dots, u_r = t$$

such that $(u_{i-1}, u_i) \in E$ for every $1 \leq i \leq r$. Its length is r , the number of edges. We allow a path of length zero from a vertex to itself.

Definition 2.2.2 (The language PATH). *The directed reachability language is*

$$\text{PATH} := \left\{ \langle G, s, t \rangle : \begin{array}{l} G \text{ is a directed graph, } s, t \in V(G), \\ G \text{ has a directed path from } s \text{ to } t \end{array} \right\}. \quad (2.5)$$

◇

Reachability models questions such as whether one state can be reached from another by a sequence of allowed transitions.

Why not enumerate paths? If a directed route repeats a vertex, we can remove the portion between two occurrences of that vertex. Repeating this process produces a path with no repeated vertices. Thus, whenever t is reachable from s , there is a path of length at most $v - 1$.

However, even sequences of at most v vertices can form a very large (i.e., exponential) search space. The marking algorithm avoids enumerating these sequences.

Algorithm 2.2.2: Directed reachability by marking

On input $\langle G, s, t \rangle$:

1. Set $R := \{s\}$.
2. Repeat the following:
 - (a) Set $R' := R$.
 - (b) Scan every edge $(u, w) \in E$. If $u \in R$, add w to R' .
 - (c) If $R' = R$, stop the repetition.
 - (d) Otherwise, replace R by R' .
3. Accept if $t \in R$; otherwise, reject.

During a scan, membership tests use the old set R . Newly added vertices are used as starting points only in the next scan.

Theorem 2.2.2. $\text{PATH} \in \text{P}$.

◇

Proof. The correctness of Algorithm 2.2.2 is obvious.

For the running time, write $v = |V|$ and $m = |E|$. Associate a Boolean mark with each vertex, indicating whether it belongs to R . Reading or setting a vertex's mark takes constant time. Initially, setting these marks takes $O(v)$ operations.

During a scan, the algorithm examines each edge (u, w) at most once. For each edge, it reads the marks of u and w and, if necessary, sets the mark of w and records that a vertex was added. Thus, each edge requires only a constant number of basic operations. Using adjacency lists, traversing the vertices and their outgoing edges therefore costs $O(v + m)$ operations per scan.

Each scan either marks at least one new vertex or terminates the repetition. Since s is marked initially and there are only $v - 1$ other vertices, at most $v - 1$ scans add vertices, followed

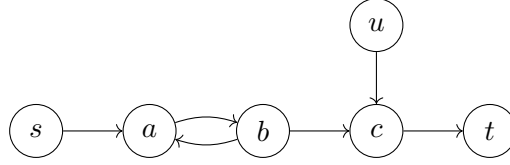


Figure 2.1: A directed reachability instance. The cycle between a and b does not prevent the marking algorithm from terminating. The vertex u is not reachable from s .

by at most one final scan. Hence the total number of operations, including initialization, is

$$O(v) + v \cdot O(v + m) = O(v(v + m)).$$

The final test of whether t is marked takes constant time. Because v and m are polynomially bounded in the length of the explicit graph encoding, the algorithm runs in polynomial time. ■

Example 2.2.2 (Tracing reachability). Consider the directed graph in Figure 2.1, whose edge set is

$$E = \{(s, a), (a, b), (b, a), (b, c), (c, t), (u, c)\}.$$

Let R_i be the marked set after i productive scans. The algorithm produces

$$\begin{aligned} R_0 &= \{s\}, \\ R_1 &= \{s, a\}, \\ R_2 &= \{s, a, b\}, \\ R_3 &= \{s, a, b, c\}, \\ R_4 &= \{s, a, b, c, t\}. \end{aligned}$$

The next scan adds nothing.

The instance $\langle G, s, t \rangle$ is accepted. The instance $\langle G, s, u \rangle$ is rejected: although u has an edge leading to c , there is no directed path from s to u .

Connection to breadth-first search

Breadth-first search explores the same reachable set using a queue. With adjacency lists, it requires $O(v + m)$ elementary graph operations. This familiar bound need not be the exact running time on a single-tape Turing machine; polynomial simulation overhead is enough to establish membership in $\mathbf{P}$.

Takeaway

An exponentially large collection of candidate paths does not imply that reachability requires exponential time. Marking vertices captures the information we need without enumerating all paths.

2.2.3 Bipartite Matching with a Size Threshold

Suppose we have applicants and jobs. An applicant may be compatible with some jobs but not others. We want to assign applicants to jobs so that each applicant receives at most one job and each job is assigned to at most one applicant.

A *bipartite graph* represents this situation. Its vertices are partitioned into disjoint sets U and W , and every edge has one endpoint in each set. We write $G = (U, W, E)$, with an edge (u, w) representing compatibility.

A *matching* is a set of edges no two of which share an endpoint. The size of a matching is its number of edges.

Definition 2.2.3 (Bipartite matching decision problem). *Define*

$$\text{BIPARTITE-MATCHING} := \left\{ \langle G, k \rangle : \begin{array}{l} G = (U, W, E) \text{ is bipartite, } k \geq 0, \\ G \text{ has a matching of size at least } k \end{array} \right\}, \quad (2.6)$$

where k is encoded in binary and the partition (U, W) is included in the graph representation. $\diamond$

Maximum versus maximal. A *maximum* matching has the largest possible size. A *maximal* matching is one to which no single edge can be added while retaining the matching property.

These notions are different. A maximal matching may fail to be maximum because increasing its size can require replacing some selected edges.

Example 2.2.3 (Why a greedy choice may need to be revised). *Let*

$$U = \{a, b\}, \quad W = \{1, 2\}, \quad E = \{(a, 1), (a, 2), (b, 1)\}.$$

If we first select $(a, 1)$, neither remaining edge can simply be added. Thus, $\{(a, 1)\}$ is maximal. However,

$$\{(a, 2), (b, 1)\}$$

is a matching of size 2. The first matching is not maximum. For this graph, the threshold $k = 2$ gives a yes-instance, whereas $k = 3$ gives a no-instance.

Augmenting paths

Relative to a matching M , a vertex is *unmatched* if it is not incident to an edge of M .

An *augmenting path* is a simple path whose endpoints are unmatched and whose edges alternate between edges outside M and edges in M , starting and ending with edges outside M .

Then, we can characterize maximum matchings in terms of augmenting paths:

Theorem 2.2.3 (Berge's Theorem [Ber57]). *A matching is maximum if and only if it has no augmenting path.* $\diamond$

Proof. We prove the two directions separately.

First, suppose that M has an augmenting path P . By definition, the endpoints of P are unmatched by M , and the edges of P alternate between edges not in M and edges in M . Since P starts and ends with edges outside M , it has one more edge outside M than inside M .

Now form a new set of edges M' by “flipping” the edges along P : remove from M all edges of P that are in M , and add all edges of P that are not in M . In symbols,

$$M' = M \triangle E(P),$$

where $\triangle$ denotes symmetric difference and $E(P)$ is the set of edges contained in P .

We claim that M' is again a matching. Indeed, every internal vertex of P was incident to exactly one selected edge before the flip, and it is still incident to exactly one selected edge after the flip. The two endpoints of P , which were unmatched by M , become matched after the flip. All vertices not on P are unaffected. Thus no vertex is incident to more than one edge of M' , so M' is a matching.

Moreover, since P contains one more edge outside M than inside M , the flip increases the size of the matching by one:

$$|M'| = |M| + 1.$$

Therefore, if M has an augmenting path, then M is not maximum.

Conversely, suppose that M is not maximum. Then there exists a matching M^* with

$$|M^*| > |M|.$$

We compare M and M^* by looking at the graph whose edge set is their symmetric difference:

$$M \triangle M^* = (M \setminus M^*) \cup (M^* \setminus M).$$

That is, we keep exactly those edges that belong to one of the two matchings, but not both.

In this graph, every vertex has degree at most two. This is because a vertex is incident to at most one edge of M and at most one edge of M^* . Hence each connected component of this graph is either a path, a cycle, or an isolated vertex. Ignoring isolated vertices, every component is a path or a cycle.

Furthermore, along each nontrivial component, the edges must alternate between edges of M and edges of M^* . This follows from the matching property: two distinct edges of M cannot be adjacent, and two distinct edges of M^* cannot be adjacent.

Now consider the number of edges from M and from M^* in each component. Any cycle must contain the same number of edges from M and M^* , since the edges alternate around the cycle. A path may have the same number of edges from the two matchings, or it may have one more edge from one matching than from the other.

Since $|M^*| > |M|$, across all components there are more edges from M^* than from M . Therefore, at least one path component must contain more edges from M^* than from M .

Let P be such a path component. Because the edges of P alternate between M and M^* , and because P has more edges from M^* than from M , the path P must start and end with edges of M^* . Thus P has the form

$$M^*, M, M^*, M, \dots, M^*.$$

It remains to check that the endpoints of P are unmatched by M . Consider one endpoint v of P . The unique edge of P incident to v lies in M^* . If v were also incident to an edge of M , then that M -edge would belong to the symmetric difference and would be adjacent to the M^* -edge at v . Hence it would also be part of the same component, contradicting the fact that v is an endpoint of the path. Therefore v is unmatched by M . The same argument applies to the other endpoint.

Thus P is a simple path whose endpoints are unmatched by M , and whose edges alternate between edges outside M and edges in M , starting and ending with edges outside M . Hence P is an augmenting path for M .

We have shown that if M is not maximum, then M has an augmenting path. Combining this with the first direction proves the theorem. $\blacksquare$

In a bipartite graph, an augmenting path can be found by a directed reachability search. Temporarily orient:

- edges outside M from U to W ;
- edges in M from W to U .

Search from the unmatched vertices in U for an unmatched vertex in W . A directed path found this way is an augmenting path.

Algorithm 2.2.3: Deciding bipartite matching

On input $\langle G, k \rangle$, where $G = (U, W, E)$:

1. If $k = 0$, accept. If $k > \min\{|U|, |W|\}$, reject.
2. Initialize $M := \emptyset$.
3. While $|M| < k$:
 - (a) Search for an augmenting path relative to M .
 - (b) If no augmenting path exists, reject.
 - (c) Otherwise, replace the matching edges along the path by its nonmatching edges.
4. Accept.

Theorem 2.2.4. BIPARTITE-MATCHING $\in P$. ◇

Proof. Every augmentation preserves the matching property and increases the size by one. Thus, acceptance certifies the existence of a matching of size k .

If the algorithm rejects because no augmenting path exists, then [Theorem 2.2.3](#) implies that the current matching is maximum. Its size is less than k , so rejection is correct.

There are at most $\min\{|U|, |W|\}$ augmentations. Each search is a polynomial-time graph search. In the usual adjacency-list model, each search and update uses $O(v + m)$ graph operations, giving $O(v(v + m))$ operations overall. Accounting for representation and simulation costs still gives a deterministic polynomial-time bound. ■

Returning to [Example 2.2.3](#), start with $M = \{(a, 1)\}$. The path

$$b, 1, a, 2$$

is augmenting: its edges are outside M , inside M , and outside M , respectively. Removing $(a, 1)$ and inserting $(b, 1)$ and $(a, 2)$ produces the matching of size 2.

Further Discussion on Matching Algorithms

The algorithm described above is the generic *augmenting-path algorithm* for bipartite matching. But improved algorithms exist, such as the Hopcroft–Karp algorithm. The Hopcroft–Karp algorithm improves [Algorithm 2.2.3](#) by proceeding in phases. In each phase, it finds a maximal collection of vertex-disjoint shortest augmenting paths and augments the matching along all of them simultaneously. Its running time is

$$O((|E| + |V|)\sqrt{|V|}),$$

which is commonly written as $O(|E|\sqrt{|V|})$ after isolated vertices have been discarded [[HK73](#)].

Polynomial-time matching algorithms are not limited to bipartite graphs. Both the decision and search versions of maximum matching are polynomial-time solvable for general graphs. Odd cycles create an additional difficulty because they give rise to structures called *blossoms*, which prevent the straightforward bipartite augmenting-path search from working. Edmonds’s blossom algorithm handles this difficulty by contracting blossoms and finds a maximum matching in polynomial time [[Edm65](#)]. The later Micali–Vazirani algorithm finds a maximum matching in a general graph $G = (V, E)$ in $O(|E|\sqrt{|V|})$ time [[MV80](#)].

2.3 Decision and Search

Applications often require an explicit solution rather than a yes/no answer. In this section, we formalize that distinction and show how a decision procedure can be used to construct a actual solution to the problem. We will use the bipartite matching problem (discussed in [Section 2.2.3](#)) as a running example.

The key point is not that the previously described matching algorithm already produces a matching. We will treat the decision procedure as a subroutine that reveals only one bit per query.

2.3.1 Witnesses and Search Problems

Two formulations of the same task

Recall that a *matching* in a graph $G = (V, E)$ is a subset $M \subseteq E$ such that no two edges in M share an endpoint. Equivalently, each vertex in V is incident to at most one edge in M . A matching is *perfect* if every vertex in V is incident to exactly one edge in M .

For a bipartite graph G , compare:

- **Decision:** Does G have a matching of size at least k ?
- **Search:** Find such a matching, or report that none exists.

A matching demonstrating that the answer is yes is called a *witness*. For PATH, a witness is a path from s to t .

Definition 2.3.1 (Witness relation and associated search problem). *A witness relation is a relation*

$$R \subseteq \{0, 1\}^* \times \{0, 1\}^*,$$

where $R(x, w)$ means that w is a valid solution for instance x .

In the examples considered here:

- validity can be checked deterministically in time polynomial in $|x| + |w|$;
- there is a polynomial q such that every valid witness satisfies $|w| \leq q(|x|)$.

The associated decision language is

$$L_R := \{x : \text{there exists } w \text{ such that } R(x, w)\}. \quad (2.7)$$

The associated search problem asks, on input x , for any w satisfying $R(x, w)$, or for the failure symbol $\perp$ if no such w exists. $\diamond$

The symbol $\perp$ is not a witness. It means that the instance has no valid solution.

The polynomial bound on witness length matters because an algorithm needs time to write its output. We do not ask a polynomial-time algorithm to print an exponentially long object.

Example 2.3.1 (A matching witness). For the graph in [Example 2.2.3](#), take

$$x = \langle G, 2 \rangle, \quad w = \langle \{(a, 2), (b, 1)\} \rangle.$$

To verify $R(x, w)$, check that:

1. both listed edges belong to G ;

2. no two listed edges share an endpoint;
3. the list contains at least two edges.

All three checks succeed, so w certifies that x is a yes-instance. A general matching witness contains at most $v/2$ edges and therefore has polynomial encoding length.

Search gives decision. A correct search algorithm immediately gives a decision algorithm: run it and accept precisely when it outputs a witness rather than $\perp$.

This argument assumes that the search algorithm halts and correctly reports nonexistence on no-instances. An algorithm guaranteed to work only when a solution exists does not automatically give a decider.

The converse requires an argument. A decision algorithm supplies only one bit. It does not directly reveal which edges form a matching or which vertices form a path. To recover a witness, we need a systematic way to ask additional decision questions.

2.3.2 Polynomial-Time Search-to-Decision Reductions

Using a decision procedure as an oracle

An *oracle* for a language L is an idealized subroutine that, on a query string y , answers whether $y \in L$.

The reduction must construct every query and process every answer. The oracle abstraction simply separates that work from the task of answering the decision question itself.

Definition 2.3.2 (Polynomial-time search-to-decision reduction). *A polynomial-time search-to-decision reduction from a search problem to a decision language L is a deterministic algorithm that solves the search problem using an oracle for L , such that:*

- the number of oracle calls is polynomial in the original input length;
- every oracle query has polynomial length;
- the computation outside the oracle calls takes polynomial time.

Later queries may depend on earlier oracle answers. ◇

Such a reduction is a polynomial-time *Turing reduction*, also called an oracle reduction. It is not required to replace the entire task by a single decision instance. This distinction will matter when we study other types of reductions later.

Proposition 2.3.1 (Replacing an oracle by an algorithm). *If a search problem has a polynomial-time search-to-decision reduction to a language $L \in \mathbf{P}$, then the search problem has a deterministic polynomial-time algorithm.*

Proof. Suppose the reduction makes at most $p(n)$ calls, each of length at most $q(n)$, and uses at most $p(n)$ steps outside those calls, for polynomials p and q .

Since $L \in \mathbf{P}$, its oracle can be replaced by a deterministic decider with running time at most $C(t+1)^d$ on queries of length t , for fixed C and d .

The resulting running time is bounded by

$$p(n) + p(n)C(q(n) + 1)^d, \tag{2.8}$$

which is polynomial in n . Every oracle answer is reproduced exactly, so correctness is preserved. ■

Recovering a matching from decision queries

Suppose the only information our subroutine provides is

$$D(G, k) = \begin{cases} 1, & \text{if } G \text{ has a matching of size at least } k, \\ 0, & \text{otherwise.} \end{cases}$$

We are not allowed to inspect how D computes its answer.

To decide whether an edge (u, w) can be included in a matching of size r , delete its endpoints (and all edges incident to them) and ask whether the remaining graph has a matching of size at least $r - 1$.

Let $G - \{u, w\}$ denote the graph obtained by deleting u , w , **and all edges incident to them**. Then, it is obvious that

$$\begin{aligned} &\text{There is a matching of size at least } r \text{ containing } (u, w) \\ &\iff D(G - \{u, w\}, r - 1) = 1. \end{aligned} \tag{2.9}$$

Algorithm 2.3.1: Finding a Size- k matching using a decision oracle

On input $\langle G, k \rangle$, where G is bipartite:

1. Query $D(G, k)$. If the answer is 0, output $\perp$ and halt.

2. Set

$$H := G, \quad M := \emptyset, \quad r := k.$$

3. While $r > 0$:

(a) Examine the edges $e = (u, w)$ of H in a fixed order. For each such edge, query

$$D(H - \{u, w\}, r - 1).$$

(b) If the answer is 0, leave H , M , and r unchanged and continue to the next edge.

(c) On the first answer of 1, add e to M , replace H by $H - \{u, w\}$, decrease r by one, and begin the next iteration of the while-loop.

4. Output M .

Lemma 2.3.1. *Algorithm 2.3.1 is a polynomial-time search-to-decision reduction for*

BIPARTITE-MATCHING.

◇

Proof. If the initial oracle query returns 0, then G has no matching of size at least k , so the algorithm correctly outputs $\perp$. Suppose, therefore, that the initial query returns 1.

We prove correctness using the following invariant, which holds at the beginning of every iteration of the while-loop:

1. M is a matching in the original graph G ;

2. no vertex of H is an endpoint of an edge in M ;
3. $|M| + r = k$;
4. H contains a matching of size at least r .

The invariant holds initially because $M = \emptyset$, $H = G$, $r = k$, and $D(G, k) = 1$.

Suppose that the invariant holds and that $r > 0$. By the fourth condition, H contains a matching N of size r . Let $e = (u, w)$ be any edge of N . Removing e from N leaves a matching of size $r - 1$, none of whose edges is incident to u or w . Consequently,

$$D(H - \{u, w\}, r - 1) = 1.$$

Thus, at least one edge examined during the current iteration produces a yes-answer, so the algorithm cannot become stuck.

Suppose that the algorithm selects an edge $e = (u, w)$ after receiving a yes-answer. By Equation (2.9), the graph $H - \{u, w\}$ contains a matching of size at least $r - 1$. Moreover, the second invariant condition guarantees that e shares no endpoint with any edge already in M . Therefore, after adding e to M , replacing H by $H - \{u, w\}$, and decreasing r by one, all four conditions of the invariant continue to hold.

Each successful iteration decreases r by one. Hence, after exactly k iterations, the algorithm reaches $r = 0$. The third invariant condition then gives

$$|M| = k,$$

and the first condition guarantees that M is a matching in G . Thus, the algorithm correctly outputs a matching of size k .

Running Time. It remains to analyze the running time. If the initial query returns 1, it must hold that

$$k \leq \frac{|V|}{2}.$$

This is because if the initial query $D(G, k)$ returns 1, then G contains a matching of size at least k . Because no two edges in a matching share an endpoint, each of the k edges requires two distinct vertices. Consequently, a matching of size k requires at least $2k$ vertices. Since G has only $|V|$ vertices, we must have

$$2k \leq |V|.$$

Equivalently,

$$k \leq \left\lfloor \frac{|V|}{2} \right\rfloor \leq \frac{|V|}{2}.$$

The algorithm performs k iterations, and each iteration examines at most $|E|$ edges. Including the initial query, it therefore makes at most

$$1 + k|E| \leq 1 + \frac{|V||E|}{2}$$

oracle queries. Also, all computation outside the oracle can be carried out in polynomial time. Hence the algorithm is a polynomial-time search-to-decision reduction for BIPARTITE-MATCHING.

This finishes the proof for Lemma 2.3.1. ■

Example 2.3.2 (Reconstructing the matching). *Use the graph from Example 2.2.3 and request $k = 2$. The initial query returns $D(G, 2) = 1$.*

Suppose the first candidate edge is $(a, 1)$. Deleting a and 1 leaves the vertices b and 2 with no edge, so

$$D(G - \{a, 1\}, 1) = 0.$$

We do not select $(a, 1)$.

Next consider $(a, 2)$. Deleting a and 2 leaves the edge $(b, 1)$, so

$$D(G - \{a, 2\}, 1) = 1.$$

We select $(a, 2)$.

In the remaining graph, selecting $(b, 1)$ leaves a request for a matching of size zero, whose decision answer is yes. The algorithm outputs

$$M = \{(a, 2), (b, 1)\}.$$

The witness was reconstructed using only yes/no answers.

A decision procedure can be enough to construct a solution when we can efficiently express the question: “Can this partial choice be extended to a complete solution?”

Scope of the principle. Many natural problems allow reductions of this kind. However, this is not a theorem asserting that every search problem reduces to its associated existence language. Nor does one yes/no answer necessarily determine a witness. The useful information comes from polynomially many carefully chosen queries.

The next section explains why the exact meaning of the decision query is essential.

2.4 When the Natural Existence Question Is Not Enough

The matching reduction works because deleting endpoints expresses the effect of committing to an edge. But it is totally possible that there are some other search problems that do not have a known reduction to the corresponding decision questions.

We will examine two examples. In both, the associated decision language is in P, but a deterministic polynomial-time algorithm for the search task is not known.

The examples in this section are not claims of proved search lower bounds. “No polynomial-time algorithm is known” and “no polynomial-time algorithm exists” are different statements. We will also specify the target decision language whenever discussing a search-to-decision reduction.

Remark 2.4.1 (On Example Selection). *We emphasize that the examples selected for this course were chosen deliberately to avoid overlap with other related courses such as CSCI3160, CSCI3130, and CSCI2100. Therefore, please do not panic if you find some of these examples unfamiliar. This is intentional: the goal is to broaden your exposure to different types of algorithmic and complexity-theoretical problems.*

2.4.1 The Maximum-Cut Problem

Let $G = (V, E)$ be an undirected graph with nonnegative integer edge weights w_e , encoded in binary. A subset $S \subseteq V$ specifies a cut, whose weight is

$$W_G(S) := \sum_{\substack{e \in E \\ |e \cap S| = 1}} w_e.$$

Thus, an edge contributes to the sum exactly when its endpoints lie on opposite sides of the cut. We allow either side to be empty.

A *maximum cut* is a subset $S^* \subseteq V$ such that

$$W_G(S^*) \geq W_G(T) \quad \text{for every } T \subseteq V.$$

The search task is to find such a subset S^* .

Existence is immediate

Every finite weighted graph has a maximum cut: there are finitely many subsets of V , and at least one attains the largest cut weight. Consequently, the existence language

$$\text{MAXIMUM-CUT-EXISTS} := \left\{ \langle G, w \rangle : \begin{array}{l} (G, w) \text{ is a valid weighted graph} \\ \text{having a maximum cut} \end{array} \right\}$$

contains every valid instance. A decider need only check that the input is a valid encoding, so this language belongs to P.

This existence argument also suggests a construction: enumerate all $2^{|V|}$ subsets, evaluate their cut weights, and return one with the largest weight. That algorithm terminates and returns a maximum cut, but its exhaustive enumeration is not a polynomial-time procedure.

Thus, we have established existence and given an algorithm, but we have not thereby given an efficient algorithm.

This is not the usual MAX-CUT decision problem

The existence question above must not be confused with the usual threshold version of the MAX-CUT decision problem:

$$\text{MAX-CUT} := \left\{ \langle G, w, K \rangle : \begin{array}{l} (G, w) \text{ is a valid weighted graph, and} \\ \text{some } S \subseteq V \text{ satisfies } W_G(S) \geq K \end{array} \right\}.$$

Here K is an integer threshold supplied as part of the input.

These questions are different:

- *Existence of an optimum:* Does the graph have a maximum cut? The answer is always yes on valid inputs.
- *Meeting a threshold:* Does the graph have a cut of weight at least K ? The answer depends on the graph, its weights, and K .

For example, a graph consisting of a single edge of weight 3 has a maximum cut, but it has no cut of weight at least 4.

On the Hardness. Our observation that existence is easy concerns MAXIMUM-CUT-EXISTS, not the usual MAX-CUT decision problem. In particular, the problem of MAX-CUT has been studied extensively, but until now, no polynomial-time algorithm is known for it. In fact, it

is NP-complete as we will discuss in detail in . For now, you just need to know that it is a hard problem and it is widely believed that $\text{MAX-CUT} \notin \text{P}$. Thus, any polynomial-time search-to-decision reduction from the maximum-cut search problem to MAX-CUT would yield a polynomial-time algorithm for MAX-CUT, which is widely believed to be impossible.

Remark 2.4.2 (Worst-Case Guarantee). *We emphasize that when we say “there is an algorithm (or Turing machine) that solves the problem,” we mean that there is a Turing machine that solves the problem for all valid inputs. This one-algorithm-for-all-inputs guarantee is what makes the algorithm design hard. For this example of maximum cut, there are many scenarios where the problem is actually easy to solve, e.g., if $K = 0$, or if the graph is bipartite. However, when we say this problem is hard, we mean that there is no known algorithm that solves the problem **for all valid inputs** in polynomial time. That is, what we want is a worse-case guarantee.*

Takeaway

An existence theorem and an efficient construction are different guarantees. Finiteness ensures that a maximum cut exists, but it does not tell us how to find one in polynomial time. This example illustrates that distinction; it does not prove that efficient construction is impossible or that search-to-decision reductions cannot exist.

2.4.2 Factorization versus Compositeness

Two different questions about an integer

A positive integer $N \geq 2$ is *prime* if its only positive divisors are 1 and N . Otherwise, it is *composite*.

Consider the following tasks for a binary-encoded integer $N \geq 2$:

- **Compositeness decision:** Does there exist an integer d with $1 < d < N$ and $d \mid N$?
- **Factor search:** Find such a d , or report that N is prime.

The corresponding decision languages are

$$\text{PRIMES} := \{\langle N \rangle : N \geq 2 \text{ is prime}\}, \quad (2.10)$$

$$\text{COMPOSITE} := \{\langle N \rangle : N \geq 2 \text{ is composite}\}. \quad (2.11)$$

Theorem 2.4.1 (Agrawal–Kayal–Saxena [AKS04]). $\text{PRIMES} \in \text{P}$. ◇

We use [Theorem 2.4.1](#) as a known result. Its proof and the AKS algorithm require number theory beyond the scope of this chapter.

Corollary 2.4.1. $\text{COMPOSITE} \in \text{P}$. ◇

Proof. Check that the input encodes an integer $N \geq 2$. On such an input, run the deterministic primality algorithm and reverse its answer. Every integer at least 2 is either prime or composite. ■

Example 2.4.1 (Knowing that a factor exists). *For $N = 91$, a compositeness decider returns yes. A factor-search algorithm must instead output an integer such as 7 or 13, because*

$$91 = 7 \cdot 13.$$

The yes-answer certifies existence, but it does not itself identify either factor.

What is not known

No deterministic polynomial-time algorithm is known for finding a nontrivial factor of every composite binary integer.

Consequently, no polynomial-time search-to-decision reduction from this factor-search task to COMPOSITE is known. Such a reduction, combined with [Corollary 2.4.1](#) and [Proposition 2.3.1](#), would yield a deterministic polynomial-time factor-search algorithm.

This does not prove that compositeness queries can never help. It identifies the consequence that a successful polynomial-time reduction would have.

A different decision question does suffice

Now consider a more informative decision language:

$$\text{BOUNDED-DIVISOR} := \left\{ \langle N, b \rangle : \begin{array}{l} N \geq 2, 2 \leq b < N, \\ \text{there exists } d \text{ with } 2 \leq d \leq b \text{ and } d \mid N \end{array} \right\}. \quad (2.12)$$

For a fixed composite N , these answers are monotone in b : once the answer becomes yes, it remains yes for larger admissible thresholds.

An oracle for this language lets us find a factor by binary search. For $N \geq 3$, first query the threshold $N - 1$. A no-answer means that N is prime. A yes-answer lets us search the interval $[2, N - 1]$ for the first threshold with a yes-answer. That threshold is the smallest nontrivial divisor. The special case $N = 2$ is handled directly.

Example 2.4.2 (Finding a divisor by threshold queries). *For $N = 91$, first establish that the threshold 90 gives a yes-answer. Maintain an interval $[\ell, h]$ containing the smallest nontrivial divisor, initially $[2, 90]$.*

At each step, query $b = \lfloor (\ell + h)/2 \rfloor$. On yes, set $h = b$; on no, set $\ell = b + 1$. The resulting steps are

query threshold b	answer	new interval
46	yes	$[2, 46]$
24	yes	$[2, 24]$
13	yes	$[2, 13]$
7	yes	$[2, 7]$
4	no	$[5, 7]$
6	no	$[7, 7]$

The final interval identifies the divisor 7.

Binary search makes $O(\log N)$ queries, each of length $O(\log N)$, so this is a polynomial-time oracle reduction. It does not give an unconditional polynomial-time factoring algorithm, because BOUNDED-DIVISOR is not known to belong to P.

Takeaway

It is inaccurate to say simply that “factoring does not reduce to decision.” Factor search reduces to bounded-divisor decision. What is not known is a polynomial-time reduction to the easier compositeness decision problem.

2.4.3 Further Discussion of Primality Testing

Nothing from [Section 2.4.3](#) will be tested on quizzes or exams!

How difficult is it to recognize a prime number? This question connects more than two thousand years of mathematics with the modern theory of computation. The sieve of Eratosthenes already provided a method for finding primes in ancient Greece. However, having an algorithm is not the same as having an efficient algorithm: for an integer N given in binary, we seek a running time polynomial in its bit length $\ell(N)$, not in its numerical value N .

The modern study of primality testing involved many mathematicians and computer scientists, including Pratt, Miller, Rabin, Solovay, Strassen, Adleman, Pomerance, and Rumely. A major breakthrough came with Miller's work, published in 1976: assuming the Extended Riemann Hypothesis, he obtained a deterministic polynomial-time primality test. Rabin's 1980 paper gave an unconditional randomized polynomial-time test, now called the *Miller–Rabin test*. For an overview of these developments, see the historical discussion in the [AKS paper](#). To appreciate what AKS later accomplished, we first examine this randomized test.

Number Theory behind Miller–Rabin

We start with a number-theoretic lemma that underlies the Miller–Rabin test. We present it [Lemma 2.4.1](#). We emphasize that it is a necessary condition for primality, but it is **not sufficient**: there are composite numbers that satisfy this condition as well.

Lemma 2.4.1 (Miller–Rabin condition for primes). *Let $N \geq 3$ be an odd prime, and write*

$$N - 1 = 2^s d, \quad s \geq 1, \quad d \text{ odd}.$$

For every integer $a \in \{1, \dots, N - 1\}$,

$$a^d \equiv 1 \pmod{N} \quad \text{or} \quad a^{2^j d} \equiv -1 \pmod{N} \quad \text{for some } 0 \leq j < s. \quad (2.13)$$

◇

Proof. The key observation is that we can repeatedly apply the difference-of-squares identity

$$u^2 - 1 = (u - 1)(u + 1).$$

Since $N - 1 = 2^s d$, for $s \geq 2$ this gives

$$\begin{aligned} x^{N-1} - 1 &= x^{2^s d} - 1 \\ &= (x^{2^{s-1} d})^2 - 1 \\ &= (x^{2^{s-1} d} - 1)(x^{2^{s-1} d} + 1) \\ &= (x^{2^{s-2} d} - 1)(x^{2^{s-2} d} + 1)(x^{2^{s-1} d} + 1) \\ &\quad \vdots \\ &= (x^d - 1)(x^d + 1)(x^{2d} + 1) \cdots (x^{2^{s-1} d} + 1) \\ &= (x^d - 1) \prod_{j=0}^{s-1} (x^{2^j d} + 1). \end{aligned}$$

At each step, we factor the remaining expression of the form $x^{2^r d} - 1$, halving its exponent and producing a new factor $x^{2^{r-1} d} + 1$. After s steps, the remaining factor is $x^d - 1$. For $s = 1$, the same identity is simply

$$x^{2d} - 1 = (x^d - 1)(x^d + 1).$$

Now substitute $x = a$. Since N is prime and $1 \leq a < N$, Fermat's little theorem implies

$$0 \equiv a^{N-1} - 1 = (a^d - 1) \prod_{j=0}^{s-1} (a^{2^j d} + 1) \pmod{N}.$$

A prime dividing a product must divide at least one of its factors. Therefore,

$$a^d - 1 \equiv 0 \pmod{N} \quad \text{or} \quad a^{2^j d} + 1 \equiv 0 \pmod{N} \quad \text{for some } 0 \leq j < s,$$

which is precisely [Equation \(2.13\)](#). ■

Witnesses and Nonwitnesses

Definition 2.4.1 (Miller–Rabin witness and nonwitness). *Let $N \geq 3$ be odd, write*

$$N - 1 = 2^s d$$

with d odd, and let $a \in \{1, \dots, N - 1\}$. We call a a Miller–Rabin witness for N if

$$a^d \not\equiv 1 \pmod{N} \quad \text{and} \quad a^{2^j d} \not\equiv -1 \pmod{N} \quad \text{for every } 0 \leq j < s.$$

Thus, a witness is a base for which [Equation \(2.13\)](#) is false.

We call a a Miller–Rabin nonwitness for N if

$$a^d \equiv 1 \pmod{N} \quad \text{or} \quad a^{2^j d} \equiv -1 \pmod{N} \quad \text{for some } 0 \leq j < s.$$

In this case, we also say that N passes the test to base a . ◇

Here, “witness” means a witness of *compositeness*, not of primality. By the contrapositive of [Lemma 2.4.1](#), finding even one Miller–Rabin witness proves that N is composite. In particular, a prime has no Miller–Rabin witnesses.

It is important that, [Equation \(2.13\)](#) is a *necessary but not sufficient* condition for primality. Namely,

$$\text{All } a \in \{1, \dots, N - 1\} \text{ are Miller–Rabin witnesses for } N \implies N \text{ is composite,}$$

but the reverse implication need not hold. A composite number can have both witnesses and nonwitnesses, as demonstrated in the following [Example 2.4.3](#).

Example 2.4.3 (A composite number with witnesses and nonwitnesses). *Let $N = 9$. Since $N - 1 = 8 = 2^3 \cdot 1$, we have $s = 3$ and $d = 1$. The Miller–Rabin sequence for a base a is*

$$(a, a^2, a^4) \pmod{9}.$$

For $a = 2$, this sequence is

$$(2, 4, 7) \pmod{9}.$$

Its first term is not 1, and none of its terms is $-1 \equiv 8 \pmod{9}$. Thus 2 is a Miller–Rabin witness for 9, proving that 9 is composite.

For $a = 8$, however, the sequence is

$$(8, 1, 1) \pmod{9}.$$

Its first term is $-1 \pmod{9}$, so 8 is a Miller–Rabin nonwitness for 9. Therefore, the composite number 9 has both witnesses and nonwitnesses: it fails the test to base 2 but passes the test to base 8.

Efficient Falsifiability of (Non)witnesses. We notice that for any $a \in \{1, \dots, N-1\}$, there is an efficient way to check whether a is a witness. To do that, compute the *Miller–Rabin sequence*

$$a^d, a^{2d}, a^{4d}, \dots, a^{2^{s-1}d} \pmod{N}.$$

Each term is obtained by squaring the preceding term modulo N . The base is a nonwitness exactly when the first term is 1 or some term is -1 . Otherwise, it is a witness.

For example, let $N = 49$. Since $49 - 1 = 2^4 \cdot 3$, the sequence generated by $a = 2$ is

$$(2^3, 2^6, 2^{12}, 2^{24}) \equiv (8, 15, 29, 8) \pmod{49}.$$

The first term is not 1, and no term is -1 modulo 49, so 2 is a witness for the compositeness of 49. In contrast,

$$18^3 = 5832 = 119 \cdot 49 + 1 \equiv 1 \pmod{49}.$$

Thus 18 is a nonwitness: 49 passes the test to base 18, even though $49 = 7^2$ is composite. This explicitly shows why passing one test does not establish primality.

The Nonwitness Bound

The effectiveness of the test rests on the following counting theorem. We state it without proof, since its proof is beyond the scope of this course. Interested students can find a proof in standard references, e.g., [Keith Conrad's notes on the Miller–Rabin test](#).

Theorem 2.4.2 (Miller–Rabin nonwitness bound). *Let $N \geq 3$ be an odd composite integer, and let*

$$L_N = \{a \in \{1, \dots, N-1\} : a \text{ is a Miller–Rabin nonwitness for } N\}.$$

Then

$$|L_N| \leq \frac{N-1}{4}.$$

Consequently, a uniformly random base in $\{1, \dots, N-1\}$ is a nonwitness with probability at most $1/4$. A uniformly random base in $\{2, \dots, N-2\}$ is a nonwitness with probability strictly less than $1/4$. $\diamond$

To see the last assertion, observe that 1 and $N-1$ are always nonwitnesses: $1^d \equiv 1 \pmod{N}$ and, since d is odd, $(N-1)^d \equiv -1 \pmod{N}$. Removing these two bases gives

$$\frac{|L_N| - 2}{N - 3} \leq \frac{(N-1)/4 - 2}{N-3} = \frac{N-9}{4(N-3)} < \frac{1}{4}.$$

Thus we avoid the two uninformative bases by sampling from $\{2, \dots, N-2\}$.

The Miller–Rabin Test

Here is the Miller–Rabin test for deciding whether an odd integer $N \geq 5$ is prime.

Algorithm 2.4.1: Miller–Rabin Primality Test

Write $N - 1 = 2^s d$ with d odd. The error bound in the last step follows from Theorem 2.4.2.

1. Pick an integer $k \geq 1$ to be the number of trials.
2. Choose an integer a uniformly at random from $\{2, \dots, N - 2\}$, independently of all previous choices. Sampling is with replacement.
3. If a is a Miller–Rabin witness for N , stop the test and output COMPOSITE. This conclusion is certainly correct.
4. If a is a nonwitness and fewer than k trials have been completed, return to Step 2.
5. If all k trials are completed without finding a witness, output PROBABLY PRIME. For every fixed composite input, the probability of reaching this output is at most 4^{-k} .

For general integer inputs $N \geq 2$, first handle the small and even cases: output PROBABLY PRIME immediately if $N = 2$ or $N = 3$, and output COMPOSITE if $N > 2$ is even. The remaining inputs are odd and at least 5, so the sampling range above is nonempty.

The error guarantee

Theorem 2.4.3 (Error guarantee for the repeated test). *For every fixed integer $N \geq 2$ and every integer $k \geq 1$, the k -trial Miller–Rabin test satisfies:*

1. *If N is prime, it outputs PROBABLY PRIME with probability 1.*
2. *If N is composite, it outputs PROBABLY PRIME with probability at most 4^{-k} .*

The probabilities are over the independent, uniformly random bases chosen by the algorithm. $\diamond$

Proof. The primes 2 and 3 are handled immediately. For any other prime, Lemma 2.4.1 shows that every base is a nonwitness, so all trials pass.

An even composite input is rejected immediately.

For a fixed odd composite input N , let

$$p_N = \frac{|L_N \cap \{2, \dots, N - 2\}|}{N - 3} = \frac{|L_N| - 2}{N - 3}.$$

By Theorem 2.4.2, $p_N < 1/4$. The algorithm outputs PROBABLY PRIME exactly when all k independently chosen bases are nonwitnesses. Therefore,

$$\Pr(\text{output PROBABLY PRIME}) = p_N^k \leq \left(\frac{1}{4}\right)^k.$$

■

For example, 20 independent trials give the error bound

$$4^{-20} = 2^{-40} < 10^{-12}.$$

This is a *one-sided error* guarantee: COMPOSITE is always correct, whereas PROBABLY PRIME allows a small probability of error.

More precisely, for each fixed composite input, the probability of detecting its compositeness is at least $1 - 4^{-k}$. This does *not* mean that, after an output of PROBABLY PRIME, the input is prime with probability at least $1 - 4^{-k}$. Such a posterior probability requires additional assumptions about how the input was selected.

Why the test is efficient

Let $n = \ell(N)$. Repeated squaring computes $a^d \bmod N$ using $O(n)$ modular multiplications, and the remaining checks require at most $s - 1 < n$ modular squarings. All stored residues have at most n bits, and a product of two such residues has at most $2n$ bits. Using ordinary multiplication and long division, each modular multiplication takes $O(n^2)$ bit operations. Thus, the arithmetic in one round takes $O(n^3)$ bit operations, and k rounds take $O(kn^3)$. See also the repeated-squaring implementation in [these notes on randomized primality testing](#).

In particular, for any fixed number of rounds, the arithmetic cost is polynomial in the binary input length. Increasing k reduces the error exponentially while increasing this cost only linearly.

The AKS breakthrough

Miller–Rabin showed that primality testing could be efficient and extremely reliable without any unproved hypothesis. But a fundamental question remained: was there a deterministic algorithm with a polynomial running-time bound and no possibility of error?

In August 2002, more than twenty years after Rabin’s paper, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena answered this question. Remarkably, Kayal and Saxena had just completed their undergraduate degrees at IIT Kanpur, where they had worked with Agrawal. Their result was published in 2004.

The significance of [Theorem 2.4.1](#) is not simply that primality can be tested efficiently: randomized polynomial-time tests had been known for decades. AKS established that neither randomness nor unproved number-theoretic assumptions are needed. It settled a longstanding problem at the intersection of mathematics and computer science and showed that, for primality testing, deterministic computation can also achieve polynomial time. This is a statement about polynomial-time solvability, not a claim that AKS is faster than Miller–Rabin. Nor does it establish that randomness can be eliminated efficiently from every randomized algorithm.

We use [Theorem 2.4.1](#) as a known result. Its proof and the AKS algorithm require number theory beyond the scope of this chapter.

2.5 Randomized Algorithms and the Limits of P

The class P captures decision problems solvable by deterministic polynomial-time algorithms. A deterministic algorithm has its next step determined by its current configuration. A *randomized algorithm*, in contrast, may consult random bits, so the same input can lead to different computation paths and possibly different outputs. To model this additional resource, we use a *probabilistic Turing machine*, which extends the deterministic model with random choices. We will formally define probabilistic Turing machines in a later chapter, but for now we can think of them as deterministic Turing machines that have access to a source of random bits.

Running time and correctness

Randomness requires us to distinguish two kinds of guarantees. First, does the algorithm run in polynomial time for *every* sequence of random choices, or is its running time polynomial only *in expectation*? Second, is its answer always correct, or can some random choices produce an

incorrect answer? An expected-time bound is not a worst-case bound, and a small probability of error is not the same as correctness on every computation.

These probabilities concern the algorithm's internal randomness, not a distribution over inputs. For example, the guarantee

$$\Pr[\text{the algorithm answers correctly on } x] \geq \frac{2}{3}$$

means that, for each fixed input x , at least two thirds of the probability lies on random choices that produce the correct answer. It is not enough for the algorithm to work on most inputs.

Why we need another complexity class

A randomized polynomial-time algorithm alone does not establish membership in P . Membership in P requires a deterministic polynomial-time decider that is correct on every input.

This does not mean that problems in P cannot have useful randomized algorithms. Primality testing and minimum cut are examples where deterministic and randomized algorithms coexist. Another good example is the famous quick-sort algorithm, which is a randomized algorithm that sorts an array in *expected* $O(n \log n)$ time—although there are deterministic sorting algorithms achieving the same time complexity even in the worst case, quick-sort is often preferred in practice due to its memory efficiency and simplicity. However, the existence of a randomized algorithm does not automatically imply that the problem is in P . Rather, P does not directly express guarantees involving random choices and a bounded probability of error.

The class BPP provides such a framework. Informally, it consists of languages decidable by randomized algorithms that run in polynomial time for every input and every sequence of random choices, and answer correctly with probability at least $2/3$ on each input. A deterministic algorithm can simply ignore the random bits, so

$$P \subseteq BPP.$$

Introducing a different computational model does not, by itself, show that randomness enlarges the class of efficiently solvable problems.

What we defer

In a later chapter, we will formally define probabilistic Turing machines and BPP , explain how repetition reduces error, distinguish one-sided from two-sided error, and study the relationship between deterministic and randomized polynomial time.

We will also address modeling details such as uniform sampling from finite sets using fair random bits. A procedure that retries until sampling succeeds may give an expected-time guarantee rather than a worst-case bound; limiting the retries requires accounting for the probability of failure.

For now, the essential point is that analyzing randomized algorithms requires both a model of random choices and explicit guarantees about running time and correctness, beyond those expressed by P .

Chapter 3

Non-Deterministic Polynomial Time: NP

3.1 From Efficient Solution to Efficient Verification

In earlier lectures, we defined computational problems as decision problems and represented them by languages. Recall that an input string x is a yes-instance of a problem represented by a language L precisely when $x \in L$. We also introduced the class P, consisting of the languages decidable by deterministic Turing machines in polynomial time.

The class P formalizes the idea that a decision problem can be solved efficiently. In this chapter, we study a related but potentially different notion: whether a proposed solution can be verified efficiently.

Finding a solution versus checking a solution

Consider the following informal problem. Given a directed graph G with designated vertices s and t , we want to determine whether there is a path from s to t that visits every vertex exactly once. Such a path is called a *Hamiltonian path* from s to t .

Finding such a path may require considering many possible orders of the vertices. However, suppose someone supplies the sequence

$$s, v_2, v_3, \dots, v_{m-1}, t.$$

We can check efficiently whether this sequence is a valid Hamiltonian path:

1. Check that every vertex of G appears exactly once.
2. Check that the sequence starts at s and ends at t .
3. Check that each consecutive pair in the sequence is connected by a directed edge.

Thus, even if we do not know how to find a Hamiltonian path efficiently, we can efficiently check whether a proposed path is valid.

Remark 3.1.1 (Solving and verifying are different tasks). *A verification algorithm receives both the original input and a proposed solution. A decision algorithm receives only the original input and must determine the answer without being given a proposed solution.*

This distinction motivates the class NP. Informally, NP contains decision problems whose yes-instances have short certificates that can be checked efficiently.

The central idea of NP is efficient verification: a yes-instance has a polynomial-length witness whose validity can be checked in polynomial time.

Roadmap

Our development proceeds in four stages:

- We formalize witnesses and polynomial-time verifiers.

- We prove that several representative problems belong to NP.
- We prove that $P \subseteq NP$ and discuss the P versus NP question.
- We define nondeterministic Turing machines and prove that they give an equivalent definition of NP.

The verifier definition will be our first formal definition of NP. Nondeterminism will then provide a machine-based interpretation of the same class.

3.2 Witnesses, Polynomial-Time Verifiers, and the Definition of NP

Witnesses

Let $L \subseteq \Sigma^*$ be a language, where Σ is the alphabet; throughout this course, we will use the binary alphabet $\Sigma = \{0, 1\}$.

A *witness*, also called a *certificate*, is a string intended to provide evidence that an input x belongs to L .

For example:

- For a graph containing a Hamiltonian path, a witness may be an ordered list of vertices.
- For a composite integer, a witness may be a nontrivial factor.

A witness does not need to explain how it was found. Its purpose is only to provide enough information for a deterministic algorithm to verify the yes-answer.

Definition 3.2.1 (Verifier). *A verifier for a language $L \subseteq \Sigma^*$ is a deterministic Turing machine V such that*

$$x \in L \iff \exists w \in \Sigma^* \text{ such that } V(x, w) = 1. \quad (3.1)$$

The string x is the problem instance, the string w is a witness, and $V(x, w) = 1$ means that the verifier accepts the pair (x, w) . $\diamond$

The two directions of [Equation \(3.1\)](#) express two different correctness requirements:

- **Completeness:** If $x \in L$, then at least one witness w causes V to accept.
- **Soundness:** If $x \notin L$, then no witness w causes V to accept.

A yes-instance may have many valid witnesses. The definition requires only that at least one valid witness exist. By contrast, a no-instance must have no valid witness at all.

Example 3.2.1 (A witness for membership in PATH). *Recall that*

$$\text{PATH} = \{\langle G, s, t \rangle : \text{there is a directed path from } s \text{ to } t \text{ in } G\}.$$

Consider the instance

$$x = \langle G, s, t \rangle,$$

where

$$V(G) = \{s, a, b, t\}$$

and

$$E(G) = \{(s, a), (s, b), (a, b), (b, t), (a, t)\}.$$

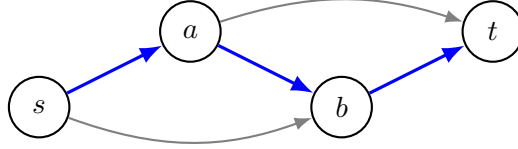
The statement being verified is

$$x \in \text{PATH};$$

that is, G contains a directed path from s to t . A witness for this statement is a sequence of vertices describing such a path. For example, consider

$$w = (s, a, b, t).$$

The graph and the path described by w are illustrated below. The edges of the proposed path are highlighted in blue.



The witness $w = (s, a, b, t)$ is shown in blue.

To verify w , the verifier checks that w begins at s , ends at t , and that every consecutive pair in w is an edge of G :

$$(s, a) \in E(G), \quad (a, b) \in E(G), \quad (b, t) \in E(G).$$

All these checks succeed, so w certifies that

$$\langle G, s, t \rangle \in \text{PATH}.$$

In contrast, consider the proposed witness

$$w' = (s, b, a, t).$$

The verifier rejects w' because

$$(b, a) \notin E(G).$$

Rejecting w' does not show that $\langle G, s, t \rangle \notin \text{PATH}$. It shows only that w' is not a valid witness. Indeed, the valid witness $w = (s, a, b, t)$ establishes that this instance belongs to PATH .

Polynomially Bounded Witnesses

To capture efficient verification, we must impose two efficiency requirements:

1. The witness must have polynomial length.
2. The verifier must run in polynomial time.

The witness-length requirement is necessary because an algorithm cannot efficiently process a witness that is exponentially longer than the original input.

Definition 3.2.2 (Polynomially bounded witness). *A verifier V uses polynomially bounded witnesses if there is a polynomial p such that every relevant witness w satisfies*

$$|w| \leq p(|x|),$$

where $|x|$ and $|w|$ denote the lengths of the input and witness, respectively. $\diamond$

For instance, if $|x| = n$, a witness of length $n^3 + 2n$ is polynomially bounded. A witness of length 2^n is not polynomially bounded.

The running time of a verifier is naturally measured in terms of the total input length $|x| + |w|$. If

$$|w| \leq p(|x|)$$

and the verifier runs in time polynomial in $|x| + |w|$, then its running time is also polynomial in $|x|$. Indeed, if the running time is bounded by a polynomial q , then

$$q(|x| + |w|) \leq q(|x| + p(|x|)),$$

which is polynomial in $|x|$.

The Verifier Definition of NP

We can now give our first formal definition of NP.

Definition 3.2.3 (The class NP: verifier definition). *The class NP consists of all languages $L \subseteq \Sigma^*$ for which there exist a deterministic polynomial-time Turing machine (dubbed a verifier) V and a polynomial p such that, for every $x \in \Sigma^*$,*

$$x \in L \iff \exists w \in \Sigma^* \text{ such that } |w| \leq p(|x|) \text{ and } V(x, w) = 1. \quad (3.2)$$

◇

The verifier V and polynomial p are fixed for the language L . They cannot be chosen separately for each input.

Let us unpack [Equation \(3.2\)](#):

- x is the original instance of the decision problem.
- w is a proposed witness for the claim $x \in L$.
- The existential quantifier means that at least one accepted witness must exist for each yes-instance.
- The condition $|w| \leq p(|x|)$ ensures that the witness is not excessively long.
- The verifier V checks the witness deterministically in polynomial time.
- If $x \notin L$, every possible witness must be rejected.

What the definition does not provide. The definition of NP does not give an efficient method for finding a witness. It says only that if a suitable witness is supplied, then its validity can be checked efficiently.

3.3 Examples of Problems in NP

The examples in this section illustrate several kinds of witnesses. A witness may be a truth assignment, an ordering of vertices, a subset of vertices or numbers, or an arithmetic factor. In each case, the central task is to verify a proposed solution rather than to find one.

3.3.1 Boolean Satisfiability

Motivation

Many problems in computer science ask whether a collection of constraints can be satisfied simultaneously. For example:

- A software configuration tool may need to select features subject to requirements such as “if feature *A* is enabled, then feature *B* must also be enabled” and “features *C* and *D* cannot both be enabled.” See [Supplementary Box 3.3.1](#) for a concrete example of this type of problem.
- A package manager (e.g., *pip*, *conda*, *brew*, *npm*, *yarn*, *apm*) may need to determine whether packages can be installed without violating dependency and incompatibility constraints.
- The model checker *CBMC* can analyze a C program to determine whether some input causes an assertion to fail, an array access to go out of bounds, or an arithmetic overflow to occur.
- The AI planning system *Fast Downward* can plan actions for a warehouse robot, such as moving between locations, picking up a package, and delivering it to a packing station while respecting the preconditions of each action. See [Supplementary Box 3.3.2](#) for a concrete example of this type of problem.

In each of these applications, a binary choice can be represented by a Boolean variable. Logical formulas can then express the constraints that valid choices must satisfy. The resulting computational question is:

Is there an assignment of values to the variables that makes all the constraints true?

This is the Boolean satisfiability problem. It is one of the most fundamental problems in theoretical computer science and also forms the basis of practical tools called *SAT solvers*.

Supplementary Box 3.3.1: Configuring a Linux Kernel as a Satisfiability Problem

Consider configuring a custom Linux kernel for an embedded device. Linux provides a real configuration system, called *Kconfig*, through which a developer chooses which kernel capabilities and device drivers will be included in the compiled system.

For example, introduce Boolean variables with the following meanings:

N = “general networking support is enabled,”

W = “wireless-networking support is enabled,”

U = “USB support is enabled,”

K = “the USB-keyboard driver is enabled.”

These choices cannot be made independently. Wireless networking requires general networking support, so the configuration must satisfy

$$W \rightarrow N.$$

Similarly, the USB-keyboard driver is useful only when USB support is enabled, giving the constraint

$$K \rightarrow U.$$

Using the logical equivalence

$$A \rightarrow B \equiv \neg A \vee B,$$

these requirements can be written as the clauses

$$(\neg W \vee N) \quad \text{and} \quad (\neg K \vee U).$$

Suppose that the embedded device must support both Wi-Fi and a USB keyboard. We then add the requirements

$$W \quad \text{and} \quad K.$$

The complete configuration formula becomes

$$(\neg W \vee N) \wedge (\neg K \vee U) \wedge W \wedge K.$$

A satisfying assignment is

$$N = 1, \quad W = 1, \quad U = 1, \quad K = 1.$$

This assignment describes a valid kernel configuration containing all the infrastructure required by the selected functionality.

In a realistic configuration system, there may be thousands of such options and dependencies. Determining whether all requested capabilities can be enabled simultaneously is therefore a Boolean satisfiability problem.

Supplementary Box 3.3.2: Controlling a Warehouse Robot

Consider using an AI planning system such as *Fast Downward* to control a warehouse robot. Fast Downward accepts a description of the available actions, their preconditions and effects, the initial state, and the desired goal. It then searches for a sequence of actions that reaches the goal.

Suppose a package is initially in a storage room and must be delivered to a packing station. The robot can move between the two locations, pick up the package, and put it down. For each time step t , introduce Boolean variables with the following meanings:

$$\begin{aligned} R_{S,t} &= \text{“the robot is in the storage room at time } t\text{,”} \\ R_{P,t} &= \text{“the robot is at the packing station at time } t\text{,”} \\ B_t &= \text{“the robot is carrying the package at time } t\text{,”} \\ L_{S,t} &= \text{“the package is in the storage room at time } t\text{,”} \\ L_{P,t} &= \text{“the package is at the packing station at time } t\text{.”} \end{aligned}$$

We also introduce variables representing the actions that the robot may perform:

$$\begin{aligned} Pick_t &= \text{“the robot picks up the package at time } t\text{,”} \\ Move_{S,P,t} &= \text{“the robot moves from storage to packing at time } t\text{,”} \\ Drop_t &= \text{“the robot puts down the package at time } t\text{.”} \end{aligned}$$

The initial state is described by

$$R_{S,0} \wedge L_{S,0} \wedge \neg B_0.$$

Thus, the robot and package are both in the storage room, and the robot is not yet carrying the package.

Each action has preconditions. For example, the robot can pick up the package only if both

the robot and the package are in the storage room:

$$Pick_t \rightarrow (R_{S,t} \wedge L_{S,t}).$$

Similarly, the robot can move from storage to the packing station only if it is currently in storage:

$$Move_{S,P,t} \rightarrow R_{S,t}.$$

The robot can put down the package only if it is carrying the package and is at the packing station:

$$Drop_t \rightarrow (B_t \wedge R_{P,t}).$$

Actions also have effects. Picking up the package causes the robot to carry it:

$$Pick_t \rightarrow B_{t+1}.$$

Moving from storage to packing changes the robot's location:

$$Move_{S,P,t} \rightarrow R_{P,t+1}.$$

Putting down the package at the packing station completes the delivery:

$$Drop_t \rightarrow L_{P,t+1}.$$

If the robot must complete the delivery within three actions, the goal condition is

$$L_{P,3}.$$

A satisfying assignment may select the actions

$$Pick_0 = 1, \quad Move_{S,P,1} = 1, \quad Drop_2 = 1.$$

These true variables describe the plan

$$Pick ; Move_{S,P} ; Drop.$$

In a realistic planning problem, the formula would also include constraints describing unchanged facts, mutually exclusive actions, battery capacity, blocked paths, multiple robots, and delivery deadlines. Finding a satisfying assignment then corresponds to finding a valid plan that achieves the goal without violating any of these constraints.

Boolean Formulas

A *Boolean variable* is a variable that takes a value in

$$\{0, 1\},$$

where 1 represents *true* and 0 represents *false*. We use variables such as

$$x_1, x_2, \dots, x_m.$$

Boolean formulas are constructed from Boolean variables using logical connectives. The

three basic connectives used here are:

$$\begin{aligned}\neg\varphi & : \text{ negation, meaning “not } \varphi\text{”,} \\ \varphi \wedge \psi & : \text{ conjunction, meaning “}\varphi \text{ and } \psi\text{”,} \\ \varphi \vee \psi & : \text{ disjunction, meaning “}\varphi \text{ or } \psi\text{”}.\end{aligned}$$

The disjunction $\varphi \vee \psi$ is inclusive: it is true when at least one of φ and ψ is true, including when both are true.

Definition 3.3.1 (Literal). *A literal is either a Boolean variable x , called a positive literal, or the negation $\neg x$ of a Boolean variable, called a negative literal.* $\diamond$

For example,

$$x_1, \quad \neg x_2, \quad x_3$$

are literals.

Definition 3.3.2 (Clause). *A clause is a disjunction of one or more literals. Thus, a clause has the form*

$$\ell_1 \vee \ell_2 \vee \cdots \vee \ell_t,$$

where each ℓ_i is a literal. $\diamond$

For example,

$$x_1 \vee \neg x_2 \vee x_4$$

is a clause. It is true whenever at least one of its literals is true.

Definition 3.3.3 (Conjunctive Normal Form). *A Boolean formula is in conjunctive normal form, abbreviated CNF, if it is a conjunction of clauses. A CNF formula therefore has the form*

$$C_1 \wedge C_2 \wedge \cdots \wedge C_s,$$

where each C_i is a clause. $\diamond$

For example,

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee x_3)$$

is a CNF formula consisting of three clauses. A CNF formula is true exactly when every one of its clauses is true.

A general Boolean formula need not be in CNF. For example,

$$\neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$$

is a Boolean formula, but it is not written in CNF.

Definition 3.3.4 (Truth assignment). *Let φ be a Boolean formula whose variables belong to*

$$\{x_1, x_2, \dots, x_m\}.$$

A truth assignment for φ is a function

$$\alpha : \{x_1, x_2, \dots, x_m\} \longrightarrow \{0, 1\}.$$

Thus, $\alpha(x_i)$ specifies whether x_i is assigned false or true. $\diamond$

The value of a formula under an assignment is determined recursively. For example,

$$\begin{aligned}\neg x_i \text{ is true} &\iff \alpha(x_i) = 0, \\ \varphi \wedge \psi \text{ is true} &\iff \varphi \text{ and } \psi \text{ are both true,} \\ \varphi \vee \psi \text{ is true} &\iff \text{at least one of } \varphi \text{ and } \psi \text{ is true.}\end{aligned}$$

If φ evaluates to true under α , we say that α *satisfies* φ , and write

$$\alpha \models \varphi.$$

Such an assignment is called a *satisfying assignment* for φ . A formula is *satisfiable* if it has at least one satisfying assignment; otherwise, it is *unsatisfiable*.

Definition 3.3.5 (Boolean satisfiability). *Define*

$$\text{SAT} := \left\{ \langle \varphi \rangle \mid \begin{array}{l} \varphi \text{ is a Boolean formula and there exists} \\ \text{a truth assignment } \alpha \text{ such that } \alpha \models \varphi \end{array} \right\}.$$

Equivalently, SAT is the problem of deciding whether a given Boolean formula has a satisfying assignment. $\diamond$

The restriction of this problem to formulas in conjunctive normal form is commonly called CNF-SAT. In this subsection, however, SAT permits arbitrary Boolean formulas.

Proposition 3.3.1. SAT $\in$ NP.

Proof. The proof of [Proposition 3.3.1](#) is straightforward: Given a candidate witness, the verifier evaluates the formula under the specified assignment. If the formula evaluates to true, the verifier accepts; otherwise, it rejects. The evaluation can be performed in polynomial time, and the witness is polynomially bounded in length. $\blacksquare$

Example 3.3.1 (Verifying a satisfying assignment). *Consider the CNF formula*

$$\varphi = \underbrace{(x_1 \vee \neg x_2)}_{C_1} \wedge \underbrace{(x_2 \vee x_3)}_{C_2} \wedge \underbrace{(\neg x_1 \vee x_3)}_{C_3}.$$

The formula consists of the three clauses

$$\begin{aligned}C_1 &= x_1 \vee \neg x_2, \\ C_2 &= x_2 \vee x_3, \\ C_3 &= \neg x_1 \vee x_3.\end{aligned}$$

The expressions

$$x_1, \quad \neg x_2, \quad x_2, \quad x_3, \quad \neg x_1$$

appearing inside these clauses are literals.

Suppose that the witness specifies the assignment

$$\alpha(x_1) = 1, \quad \alpha(x_2) = 0, \quad \alpha(x_3) = 1.$$

Equivalently, the witness is the bit string

$$w = 101.$$

Evaluating the three clauses under this assignment gives

$$\begin{aligned} C_1 &= x_1 \vee \neg x_2 = 1 \vee \neg 0 = 1 \vee 1 = 1, \\ C_2 &= x_2 \vee x_3 = 0 \vee 1 = 1, \\ C_3 &= \neg x_1 \vee x_3 = \neg 1 \vee 1 = 0 \vee 1 = 1. \end{aligned}$$

Therefore,

$$\varphi = C_1 \wedge C_2 \wedge C_3 = 1 \wedge 1 \wedge 1 = 1.$$

Thus,

$$\alpha \models \varphi,$$

so $w = 101$ is a certificate that

$$\langle \varphi \rangle \in \text{SAT}.$$

3.3.2 Hamiltonian Path

A directed path in a directed graph is a sequence of vertices in which every consecutive pair is connected by a directed edge. A Hamiltonian path is a path that visits every vertex exactly once.

Remark 3.3.1. We want to distinguish between a Hamiltonian path and a Hamiltonian cycle. A Hamiltonian cycle is a Hamiltonian path with an additional directed edge from its last vertex back to its first vertex. Thus, both a Hamiltonian path and a Hamiltonian cycle visit every vertex exactly once, but a Hamiltonian cycle returns to its starting vertex, whereas a Hamiltonian path need not do so. Consequently, a graph may contain a Hamiltonian path without containing a Hamiltonian cycle.

Definition 3.3.6 (Hamiltonian path). Define

$$\text{HAMPATH} := \left\{ \langle G, s, t \rangle \mid \begin{array}{l} G \text{ is a directed graph containing a Hamiltonian} \\ \text{path that begins at } s \text{ and ends at } t \end{array} \right\}.$$

◇

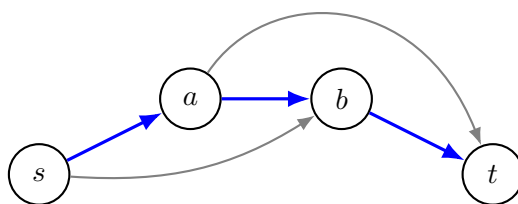


Figure 3.1: A directed graph containing the Hamiltonian path (s, a, b, t) , highlighted in blue.

Proposition 3.3.2. $\text{HAMPATH} \in \text{NP}$.

Proof. Let G have m vertices. The witness is an ordered sequence

$$w = (v_1, v_2, \dots, v_m)$$

that claims to be a Hamiltonian path from s to t .

The verifier performs the following checks:

1. Check that the sequence contains exactly m vertices.

2. Check that every vertex of G appears exactly once.
3. Check that $v_1 = s$ and $v_m = t$.
4. For each $i \in \{1, \dots, m-1\}$, check that

$$(v_i, v_{i+1}) \in E(G).$$

The witness contains m vertex names, each of which has a representation of length polynomial in the size of the graph encoding. Hence the total witness length is polynomially bounded.

If G has a Hamiltonian path from s to t , listing its vertices in order produces an accepted witness. Conversely, if the verifier accepts, the listed vertices form a directed path from s to t and include every vertex exactly once. Therefore, an accepted witness describes a Hamiltonian path.

There are $m-1$ consecutive pairs to check. Duplicate detection and edge lookup can also be implemented in polynomial time under any reasonable graph encoding. Thus the verifier runs in polynomial time, and HAMPATH $\in$ NP. ■

For the graph in [Figure 3.1](#), the sequence

$$(s, a, b, t)$$

is accepted because all four vertices appear exactly once and

$$(s, a), (a, b), (b, t) \in E(G).$$

3.3.3 Clique

Let $G = (V, E)$ be an undirected graph. A set $C \subseteq V$ is a *clique* if every pair of distinct vertices in C is connected by an edge.

Remark 3.3.2 (Why is it called a clique?). *A clique is nothing but a “fully connected” set of vertices. The term clique comes from social groups. In everyday language, a clique is a small, exclusive group of people who spend time together and do not welcome outsiders. Thus, informally, a clique is a group of vertices that are all “friends” with one another.*

Definition 3.3.7 (Clique). *Define*

$$\text{CLIQUE} := \{ \langle G, k \rangle \mid G \text{ contains a clique of size at least } k \}.$$

◇

Supplementary Box 3.3.3: Maximum Course Load

Suppose a student wants to know the maximum number of courses she can take this semester without any timetable conflicts. Each course may have several sessions, and sessions from different courses may meet at overlapping times.

Construct a graph in which:

- each vertex represents a course; and
- two vertices are connected by an edge if the corresponding courses’ meeting times do not conflict.

A clique in this graph is a collection of courses that are pairwise compatible. Therefore, every clique represents a conflict-free schedule. The size of a maximum clique gives the largest

number of courses the student can take without any timetable conflicts.

Proposition 3.3.3. CLIQUE $\in$ NP.

Proof. For an input $\langle G, k \rangle$, the witness is a list

$$w = (v_1, v_2, \dots, v_k)$$

of vertices claimed to form a clique.

The verifier operates as follows:

1. Check that the witness lists k distinct vertices of G .
2. For every pair i, j with $1 \leq i < j \leq k$, check that

$$\{v_i, v_j\} \in E(G).$$

3. Accept if and only if all checks succeed.

If G contains a clique of size at least k , any k vertices from that clique form an accepted witness. Conversely, if the verifier accepts, every pair of listed vertices is connected, so the listed vertices form a clique of size k .

The verifier checks

$$\binom{k}{2} = \frac{k(k-1)}{2}$$

pairs. Because a graph with m vertices cannot contain a clique larger than m , we need only consider $k \leq m$. Thus the number of checks is $O(m^2)$, which is polynomial in the input size. The witness length is also polynomially bounded. Therefore, CLIQUE $\in$ NP. ■

Example 3.3.2 (Checking a proposed clique). Suppose an undirected graph contains vertices a, b, c, d , and its edges include

$$\{a, b\}, \quad \{a, c\}, \quad \{b, c\}.$$

Then the witness

$$w = (a, b, c)$$

certifies the existence of a clique of size 3, because all three required pairs are edges.

If $\{b, d\} \notin E(G)$, then the proposed witness

$$w' = (a, b, d)$$

is rejected. As in [Example 3.2.1](#), rejecting one witness does not establish that the instance is a no-instance; another witness may still succeed.

3.3.4 Independent Set

Let $G = (V, E)$ be an undirected graph. A set $I \subseteq V$ is an *independent set* if no pair of distinct vertices in I is connected by an edge.

Definition 3.3.8 (Independent Set). Define

$$\text{INDEPENDENT-SET} := \{\langle G, k \rangle \mid G \text{ contains an independent set of size at least } k\}.$$

◇

Supplementary Box 3.3.4: Seating Guests at a Wedding

Suppose a couple is planning the seating arrangement for their wedding. Unfortunately, some guests do not get along and should not be seated at the same table. For example, the guest list may include former couples who had unpleasant breakups, such as Alice and her ex-boyfriend Bob, or Carol and her ex-girlfriend Diana.

Construct a graph in which:

- each vertex represents a wedding guest; and
- two vertices are connected by an edge if the corresponding guests should not be seated together.

An independent set in this graph is a group of guests with no conflicts between them. Therefore, all guests in an independent set can be seated at the same table without placing any incompatible pair together. The size of a maximum independent set gives the largest number of mutually compatible guests who can safely be seated at one table.

Proposition 3.3.4. $\text{INDEPENDENT-SET} \in \text{NP}$.

Proof. Omitted. ■

3.3.5 Graph Three-Coloring

Let $G = (V, E)$ be an undirected graph. A *three-coloring* of G is an assignment of one of three colors to each vertex such that no two adjacent vertices receive the same color. A graph is *three-colorable* if it has such a coloring.

Definition 3.3.9 (Graph Three-Coloring). *Define*

$$\text{G3C} := \{ \langle G \rangle \mid G \text{ has a three-coloring} \}.$$

◇

Supplementary Box 3.3.5: Applications of Graph Coloring

Graph coloring naturally models situations in which conflicting objects must be assigned different labels, times, or resources.

Coloring a map. Suppose we want to color a map using at most three colors so that neighboring regions receive different colors. Construct a graph in which:

- each vertex represents a region of the map; and
- two vertices are connected by an edge if the corresponding regions share a boundary.

A three-coloring of this graph assigns one of three colors to each region while ensuring that neighboring regions have different colors. Thus, the map can be properly colored with at most three colors if and only if its corresponding graph is three-colorable.

Scheduling final examinations. Suppose a university wants to arrange its centralized final examinations using three time slots. Two courses may have examinations in the same time slot only if they have no students in common.

Construct a graph in which:

- each vertex represents a course; and

- two vertices are connected by an edge if the corresponding courses have at least one student in common.

Assigning a color to a vertex represents scheduling the corresponding examination in one of the three time slots. Because adjacent vertices must receive different colors, courses that have a student in common are assigned different examination times. Therefore, all examinations can be scheduled using three time slots without conflicts if and only if the resulting graph is three-colorable.

Proposition 3.3.5. $G3C \in NP$.

Proof. Omitted. ■

3.3.6 Knapsack

Suppose there are n items. Each item i has a positive integer weight w_i and a positive integer value v_i . Given a weight capacity W and a target value V , the knapsack problem asks whether some subset of the items has total weight at most W and total value at least V .

Definition 3.3.10 (Knapsack). *Define*

$$\text{KNAPSACK} := \left\{ \langle (w_1, v_1), \dots, (w_n, v_n), W, V \rangle \mid \begin{array}{l} \text{there exists } S \subseteq \{1, \dots, n\} \text{ such that} \\ \sum_{i \in S} w_i \leq W \quad \text{and} \quad \sum_{i \in S} v_i \geq V \end{array} \right\}.$$

◇

Supplementary Box 3.3.6: A Burglar in a Jewelry Store

Imagine that a burglar breaks into a jewelry store carrying a bag that can hold at most W kilograms. The store contains n pieces of jewelry (e.g., necklaces, bracelets, rings, diamonds, gold bars). Each piece i has:

- a weight w_i ; and
- a monetary value v_i .

The burglar cannot carry every piece because the bag has a limited capacity. Suppose the burglar wants to know whether it is possible to steal jewelry worth at least V dollars without exceeding the bag's weight capacity.

A selection S of jewelry is feasible precisely when

$$\sum_{i \in S} w_i \leq W,$$

and it achieves the burglar's target precisely when

$$\sum_{i \in S} v_i \geq V.$$

Thus, the question

Can the burglar select jewelry with total value at least V while carrying total weight at most W ?

is an instance of the decision version of the knapsack problem.

Proposition 3.3.6. KNAPSACK $\in$ NP.

Proof. For an input

$$x = \langle (w_1, v_1), \dots, (w_n, v_n), W, V \rangle,$$

the witness is a subset $S \subseteq \{1, \dots, n\}$ identifying the items to be placed in the knapsack.

The verifier operates as follows:

1. Compute the total weight

$$w(S) = \sum_{i \in S} w_i.$$

2. Compute the total value

$$v(S) = \sum_{i \in S} v_i.$$

3. Accept if and only if

$$w(S) \leq W \quad \text{and} \quad v(S) \geq V.$$

If the input is a yes-instance, there exists a subset S satisfying both inequalities, and that subset is an accepted witness. Conversely, if the verifier accepts a witness S , then the selected items have total weight at most W and total value at least V . Therefore, the input is a yes-instance.

The witness can be represented by an n -bit string indicating which items are selected. The required additions and comparisons can be performed in time polynomial in the length of the input. Therefore,

$$\text{KNAPSACK} \in \text{NP}.$$

■

3.3.7 Subset Sum

The subset-sum problem asks whether some of the integers in a given list add up to a specified target.

Definition 3.3.11 (Subset sum). *Define*

$$\text{SUBSET-SUM} := \left\{ \langle a_1, \dots, a_m, t \rangle \mid \exists I \subseteq \{1, \dots, m\} \text{ such that } \sum_{i \in I} a_i = t \right\}.$$

◇

Proposition 3.3.7. SUBSET-SUM $\in$ NP.

Proof. The witness is an m -bit string

$$w = w_1 w_2 \cdots w_m,$$

where

$$w_i = \begin{cases} 1, & \text{if } a_i \text{ is selected,} \\ 0, & \text{if } a_i \text{ is not selected.} \end{cases}$$

The witness has length m , which is at most the length of the input encoding.

The verifier computes

$$S = \sum_{i=1}^m w_i a_i \tag{3.3}$$

and accepts if and only if $S = t$.

If the input is a yes-instance, the indicator string of a subset whose sum is t is accepted. Conversely, an accepted indicator string identifies a subset whose sum is exactly t .

The integers are represented in binary. Adding polynomially many binary integers, each having at most the input length in bits, can be done in polynomial time. Therefore, the verifier runs in polynomial time, and SUBSET-SUM $\in$ NP. ■

Example 3.3.3 (A subset-sum witness). *Consider the instance*

$$\langle 3, 7, 10, 14, 17 \rangle,$$

where the final number 17 is the target. The witness

$$w = 1001$$

selects 3 and 14. Using Equation (3.3), the verifier computes

$$1 \cdot 3 + 0 \cdot 7 + 0 \cdot 10 + 1 \cdot 14 = 17.$$

It therefore accepts.

By contrast, the witness 0110 selects 7 and 10, which also sums to 17, so a yes-instance may have more than one accepted witness.

3.3.8 Composite Integers

An integer $n > 1$ is composite if it can be expressed as a product of two smaller positive integers, neither of which is 1.

Definition 3.3.12 (Composite integers). *Define*

$$\text{COMPOSITES} := \{ \langle n \rangle \mid n = ab \text{ for some integers } a, b \text{ with } 1 < a < n \text{ and } 1 < b < n \}.$$

◇

Proposition 3.3.8. COMPOSITES $\in$ NP.

Proof. The witness is a pair of binary integers $\langle a, b \rangle$. The verifier checks that

$$1 < a < n, \quad 1 < b < n,$$

computes ab , and accepts if and only if

$$ab = n.$$

If n is composite, its nontrivial factors provide an accepted witness. Conversely, if the verifier accepts, then it has found a nontrivial factorization of n , so n is composite.

Because $a < n$ and $b < n$, each factor uses at most as many binary digits as n . Binary comparison and multiplication can be performed in polynomial time. Hence the witness has polynomial length, the verifier runs in polynomial time, and COMPOSITES $\in$ NP. ■

Remark 3.3.3 (More than one verifier for the same language). *The language COMPOSITES is a special example because it illustrates that the same language may have more than one polynomial-time verification algorithm.*

The verifier in the proof above uses a nontrivial factorization $\langle a, b \rangle$ as its witness. However, a different verifier can use the empty string

$$\varepsilon$$

as its witness. On input $(\langle n \rangle, \varepsilon)$, this verifier operates as follows:

1. If $n \leq 1$, reject.
2. Run the [AKS04] primality-testing algorithm on n .
3. Accept if and only if the primality-testing algorithm reports that n is not prime.

Since every integer $n > 1$ is either prime or composite, this verifier accepts exactly when n is composite. It runs in polynomial time because primality testing can be performed in polynomial time.

Thus, COMPOSITES has at least two natural verification algorithms:

- one that verifies a nontrivial factorization supplied as a witness; and
- one that ignores the witness and directly decides whether the input is composite.

This illustrates an important point: witnesses and verifiers are not unique. Membership in NP requires only that some polynomial-time verifier with polynomially bounded witnesses exists.

3.4 The Relationship Between P and NP

We have described P as the class of efficiently solvable decision problems and NP as the class of decision problems with efficiently verifiable yes-instances. Every efficiently solvable problem is also efficiently verifiable.

Theorem 3.4.1.

$$P \subseteq NP.$$

◇

Proof. Let $L \in P$. By the definition of P, there is a deterministic polynomial-time Turing machine M that decides L .

Construct a verifier V that operates as follows on input (x, w) :

1. Ignore the witness w .
2. Run M on x .
3. Accept if and only if M accepts.

We may use the empty string ε as the witness. Its length is

$$|\varepsilon| = 0,$$

so it is polynomially bounded.

If $x \in L$, then M accepts x , and hence $V(x, \varepsilon)$ accepts. If $x \notin L$, then M rejects x , so $V(x, w)$ rejects for every possible witness w .

The verifier has the same polynomial running time as M . Thus $L \in NP$. Because L was an arbitrary language in P, we conclude that

$$P \subseteq NP.$$

■

The proof of [Theorem 3.4.1](#) has a simple interpretation: if we can solve a problem efficiently, then we can verify its answer efficiently by solving the problem again. A witness is unnecessary in this case.

3.4.1 The P Versus NP Question

We know from [Theorem 3.4.1](#) that every language in P is in NP. The major unresolved question is whether the reverse inclusion also holds:

$$P \stackrel{?}{=} NP. \tag{3.4}$$

There are two possibilities:

- If $P = NP$, every decision problem whose yes-instances can be verified efficiently can also be solved efficiently.
- If $P \neq NP$, some decision problems have efficiently verifiable yes-instances but cannot be solved by any deterministic polynomial-time algorithm.

At present, no proof of either equality or inequality is known. It is widely conjectured that

$$P \neq NP,$$

but a conjecture is not a theorem.

3.5 Nondeterministic Turing Machines

The verifier definition describes NP in terms of witnesses. We now introduce a machine model that captures the same existential behavior.

3.5.1 Nondeterministic Computation

A deterministic Turing machine has at most one legal next move from any configuration. A nondeterministic Turing machine may have several possible next moves.

Definition 3.5.1 (Nondeterministic Turing machine). *A nondeterministic Turing machine, abbreviated NTM, is a Turing machine whose transition relation may specify multiple possible next moves for a given state and tape symbol.*

In a standard single-tape model, its transition relation may be written as

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, S, R\}),$$

where:

- Q is the finite set of states;
- Γ is the tape alphabet;
- L , S , and R denote left, stay, and right head movements;
- $\mathcal{P}(S)$ denotes the power set of a set S .

◇

Thus, $\delta(q, a)$ is a set of legal transitions rather than a single transition. If this set contains several transitions, the computation branches into several possible computations.

An inaccurate-yet-helpful analogy

It is sometimes helpful to imagine that a nondeterministic machine explores all of its choices simultaneously. Under this analogy, one branch tries one candidate witness, another branch tries a different candidate witness, and so on.

This analogy helps us visualize the computation tree, but it is not the formal definition. A nondeterministic Turing machine is a mathematical model. It does not assert that an ordinary physical computer can execute exponentially many computations for free.

Computation Trees

A computation of an NTM can be represented as a rooted tree:

- The root is the initial configuration.
- Each child of a configuration represents one legal next move.
- Each root-to-leaf path is a computation branch.
- A leaf may be an accepting configuration or a rejecting configuration.

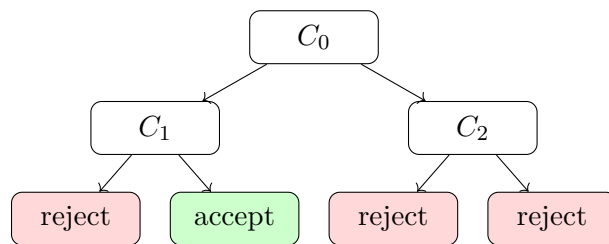


Figure 3.2: A nondeterministic computation accepts because at least one branch reaches an accepting configuration.

Definition 3.5.2 (Acceptance by an NTM). *A nondeterministic Turing machine N accepts an input x if at least one computation branch of N on x reaches an accepting state.*

The machine rejects x if every computation branch reaches a rejecting state. $\diamond$

Equivalently,

$$x \in L(N) \iff \exists \text{ an accepting computation branch of } N \text{ on } x. \quad (3.5)$$

The existential quantifier in Equation (3.5) is essential. In Figure 3.2, the machine accepts even though three of its four branches reject.

Remark 3.5.1 (Nondeterminism is not randomness). *A nondeterministic machine does not choose a branch according to a probability distribution. It accepts whenever at least one branch accepts, regardless of how many other branches reject. A randomized machine, by contrast, is analyzed using the probability that it accepts.*

Running Time of an NTM

A nondeterministic computation may have many branches of different lengths. Its running time must account for all of them.

Definition 3.5.3 (Nondeterministic running time). *The running time of a nondeterministic Turing machine N on input x is the maximum number of steps used by any computation branch of N on x .*

The machine N runs in time $T(n)$ if, for every input x of length n , every computation branch halts within at most $T(n)$ steps. $\diamond$

Using the maximum branch length ensures that a polynomial-time NTM cannot hide an extremely long or nonhalting branch. For a nondeterministic polynomial-time decider, every branch must halt in polynomial time.

3.5.2 The Guess-and-Check View

High-level descriptions of nondeterministic algorithms often use the word *guess*. A typical nondeterministic algorithm has the following form:

1. Nondeterministically guess a candidate witness w .
2. Deterministically check whether w is valid.
3. Accept if the check succeeds; otherwise, reject this branch.

Here, guessing does not mean making a random choice. It means branching over the possible choices. The machine accepts if one branch guesses a valid witness.

You can think of “guess” as a magical command with the following behavior: if a valid witness exists, the command automatically finds one. If no valid witness exists, however, the command cannot magically create one and will only return an invalid candidate. In either case, the command does not tell you whether the returned candidate is valid. You must therefore design a mechanism that verifies its validity.

Example 3.5.1 (A nondeterministic algorithm for CLIQUE). *On input $\langle G, k \rangle$, a nondeterministic machine N operates as follows:*

1. *Nondeterministically choose (or “guess”) a witness candidate:*

$$v_1, v_2, \dots, v_k.$$

2. *Reject if the chosen vertices are not distinct.*

3. *For every $i < j$, check whether*

$$\{v_i, v_j\} \in E(G).$$

4. *Accept if every required edge exists.*

If G contains a clique of size k , at least one branch chooses its vertices and accepts. If G contains no clique of size k , every branch chooses a set that fails at least one check, so every branch rejects.

Each individual branch chooses only k vertices and checks at most

$$\binom{k}{2}$$

edges. Therefore, each branch runs in polynomial time.

3.5.3 The Nondeterministic Definition of NP

We can now state the machine-based definition of NP.

Definition 3.5.4 (The class NP: nondeterministic definition). *The class NP consists of all languages decidable by a nondeterministic Turing machine in polynomial time.* $\diamond$

If $\text{NTIME}(T(n))$ denotes the class of languages decidable by nondeterministic Turing machines in time $O(T(n))$, then

$$\text{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k). \quad (3.6)$$

At first, Definition 3.5.4 appears different from the verifier definition in Definition 3.2.3. One definition refers to witnesses, while the other refers to branching computations. We next prove that the two definitions describe exactly the same class.

3.6 Equivalence of the Two Definitions of NP

The connection between Definition 3.5.4 and Definition 3.2.3 is that a witness can describe an accepting branch, and a nondeterministic branch can guess a witness.

Theorem 3.6.1 (Verifier–nondeterminism equivalence). *For every language L , the following statements are equivalent:*

1. L has a deterministic polynomial-time verifier using polynomially bounded witnesses.
2. L is decidable by a nondeterministic polynomial-time Turing machine.

$\diamond$

3.6.1 From Verifiers to Nondeterministic Machines

We first show that efficient verification implies efficient nondeterministic computation.

Lemma 3.6.1. *If a language L has a deterministic polynomial-time verifier using polynomially bounded witnesses, then L is decidable by a nondeterministic polynomial-time Turing machine.* $\diamond$

Proof. Suppose that V is a deterministic polynomial-time verifier for L . Let p be a polynomial such that

$$x \in L \iff \exists w, |w| \leq p(|x|) \text{ and } V(x, w) = 1.$$

Construct a nondeterministic Turing machine N that operates as follows on input x :

1. Nondeterministically generate a string w of length at most $p(|x|)$.
2. Run the deterministic verifier V on (x, w) .
3. Accept if and only if $V(x, w)$ accepts.

We prove correctness in both cases.

Case 1: $x \in L$

By the verifier property, there is a witness w satisfying

$$|w| \leq p(|x|) \quad \text{and} \quad V(x, w) = 1.$$

One branch of N generates this particular witness. On that branch, V accepts. Therefore, N accepts x .

Case 2: $x \notin L$

By soundness of the verifier, no string w of the allowed length satisfies $V(x, w) = 1$. Hence every branch of N generates a witness that V rejects. Therefore, every branch rejects, and N rejects x .

It remains to analyze the running time. Generating at most $p(|x|)$ symbols takes polynomial time. Because V runs in polynomial time and $|w|$ is polynomially bounded in $|x|$, running $V(x, w)$ also takes polynomial time in $|x|$. Thus every branch of N halts in polynomial time.

Therefore, N is a nondeterministic polynomial-time decider for L . ■

The main idea of [Lemma 3.6.1](#) is that the branches of N correspond to the possible witnesses. The NTM accepts exactly when one of those witnesses passes verification.

3.6.2 From Nondeterministic Machines to Verifiers

We now prove the reverse direction. An accepting branch of a nondeterministic computation will become the witness.

Lemma 3.6.2. *If a language L is decidable by a nondeterministic polynomial-time Turing machine, then L has a deterministic polynomial-time verifier using polynomially bounded witnesses.* ◇

Proof. Suppose that a nondeterministic Turing machine N decides L in time $p(n)$, where p is a polynomial. Thus, on every input x , every computation branch halts within

$$p(|x|)$$

steps.

Because N has a finite transition relation, there is a constant $b \leq |\mathcal{P}(Q \times \Gamma \times \{L, R, S\})|$ such that each configuration has at most b legal next moves. Note that this b depends only on the TM N and not on the input x , and thus it is a constant for the purposes of this proof.

Number these choices

$$1, 2, \dots, b.$$

A computation branch can be described by a sequence

$$w = c_1 c_2 \cdots c_t, \quad t \leq p(|x|),$$

where c_i specifies which transition to follow at the i -th nondeterministic step.

Each choice can be encoded using at most

$$\lceil \log_2 b \rceil$$

bits. Because b is a constant depending only on N , the witness length is at most

$$p(|x|) \lceil \log_2 b \rceil, \tag{3.7}$$

which is polynomial in $|x|$.

Construct a deterministic verifier V as follows:

1. Interpret w as a sequence of nondeterministic choices.

2. Begin simulating N on input x .
3. Whenever N has several legal transitions, use the next choice encoded in w .
4. Reject if w specifies an illegal transition or fails to specify a required choice.
5. Accept if and only if the simulated branch reaches an accepting state.

We again prove correctness in both cases.

Case 1: $x \in L$

Because N accepts x , at least one computation branch is accepting. Let w encode the sequence of choices made along that branch. The verifier follows exactly those choices and reaches the accepting state. Therefore, $V(x, w) = 1$.

Case 2: $x \notin L$

Because N decides L , every computation branch rejects x . Any well-formed witness describes one of these rejecting branches, and a malformed witness is rejected by the verifier. Hence no witness causes V to accept.

Finally, the verifier simulates at most $p(|x|)$ steps of N . A deterministic Turing machine can simulate each selected transition with at most polynomial overhead. Therefore, V runs in polynomial time. By [Equation \(3.7\)](#), its witnesses also have polynomial length.

Thus V is a deterministic polynomial-time verifier for L . ■

Bibliography

- [AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009. [14](#), [18](#)
- [AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of mathematics*, pages 781–793, 2004. [47](#), [71](#)
- [Ber57] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842–844, 1957. [doi:10.1073/pnas.43.9.842](#). [38](#)
- [Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. [doi:10.4153/CJM-1965-045-4](#). [40](#)
- [HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. [doi:10.1137/0202019](#). [40](#)
- [MV80] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|} |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, pages 17–27. IEEE, 1980. [doi:10.1109/SFCS.1980.12.40](#)