

---

# **Foundations of Computer Science**

***Release 0.5***

**Amir Kamil and Chris Peikert**

**Dec 29, 2024**

# CONTENTS

<b>I Algorithms</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Text Objectives . . . . .	2
1.2 Tools for Abstraction . . . . .	3
1.3 The First Algorithm: Euclid's GCD . . . . .	4
<b>2 The Potential Method</b>	<b>7</b>
2.1 A Potential Function for Euclid's Algorithm . . . . .	10
<b>3 Divide and Conquer</b>	<b>13</b>
3.1 The Master Theorem . . . . .	14
3.2 Integer Multiplication . . . . .	16
3.3 The Closest-Pair Problem . . . . .	19
<b>4 Dynamic Programming</b>	<b>25</b>
4.1 Implementation Strategies . . . . .	25
4.2 Weighted Task Selection . . . . .	28
4.3 Longest Increasing Subsequence . . . . .	31
4.4 Longest Common Subsequence . . . . .	33
4.5 All-Pairs Shortest Paths . . . . .	38
<b>5 Greedy Algorithms</b>	<b>41</b>
<b>II Computability</b>	<b>46</b>
<b>6 Introduction to Computability</b>	<b>47</b>
6.1 Formal Languages . . . . .	47
6.2 Overview of Automata . . . . .	50
<b>7 Finite Automata</b>	<b>53</b>
7.1 Formal Definition . . . . .	57
<b>8 Turing Machines</b>	<b>61</b>
8.1 The Language of a Turing Machine . . . . .	85
8.2 Decidable Languages . . . . .	87
8.3 Equivalent Models . . . . .	89
<b>9 Diagonalization</b>	<b>92</b>
9.1 Countable Sets . . . . .	92
9.2 Uncountable Sets . . . . .	95

9.3	The Existence of an Undecidable Language . . . . .	97
<b>10</b>	<b>“Natural” Undecidable Problems</b>	<b>100</b>
10.1	Code as Input . . . . .	100
10.2	The Barber Language . . . . .	101
10.3	The Acceptance Language and Simulation . . . . .	102
10.4	The Halting Problem . . . . .	104
<b>11</b>	<b>Turing Reductions</b>	<b>106</b>
11.1	The Halts-on-Empty Problem . . . . .	108
11.2	More Undecidable Languages and Turing Reductions . . . . .	110
11.3	Wang Tiling . . . . .	113
<b>12</b>	<b>Recognizability</b>	<b>123</b>
12.1	Unrecognizable Languages . . . . .	126
12.2	Dovetailing . . . . .	127
<b>13</b>	<b>Rice’s Theorem</b>	<b>130</b>
13.1	Rice’s Theorem and Program Analysis . . . . .	133
<b>III</b>	<b>Complexity</b>	<b>136</b>
<b>14</b>	<b>Introduction to Complexity</b>	<b>137</b>
14.1	Polynomial Time and the Class P . . . . .	138
14.2	Examples of Efficient Verification . . . . .	140
14.3	Efficient Verifiers and the Class NP . . . . .	143
14.4	P Versus NP . . . . .	147
<b>15</b>	<b>Satisfiability and the Cook-Levin Theorem</b>	<b>149</b>
<b>16</b>	<b>Proof of the Cook-Levin Theorem</b>	<b>152</b>
16.1	Configurations and Tableaus . . . . .	152
16.2	Constructing the Formula . . . . .	153
16.3	Conclusion . . . . .	160
<b>17</b>	<b>NP-Completeness</b>	<b>162</b>
17.1	Polynomial-Time Mapping Reductions . . . . .	162
17.2	NP-Hardness and NP-Completeness . . . . .	164
17.3	Resolving P versus NP . . . . .	165
<b>18</b>	<b>More NP-Complete Problems</b>	<b>169</b>
18.1	3SAT . . . . .	169
18.2	Clique . . . . .	174
18.3	Vertex Cover . . . . .	177
18.4	Set Cover . . . . .	182
18.5	Hamiltonian Cycle . . . . .	184
18.6	Concluding Remarks . . . . .	187
<b>19</b>	<b>Search Problems and Search-to-Decision Reductions</b>	<b>188</b>
<b>20</b>	<b>Approximation Algorithms</b>	<b>192</b>
20.1	Minimum Vertex Cover . . . . .	193
20.2	Maximum Cut . . . . .	196
20.3	Knapsack . . . . .	199
20.4	Other Approaches to NP-Hard Problems . . . . .	201

<b>IV Randomness</b>	<b>202</b>
<b>21 Randomized Algorithms</b>	<b>203</b>
21.1 Review of Probability . . . . .	204
21.2 Randomized Approximation Algorithms . . . . .	214
21.3 Quick Sort . . . . .	219
21.4 Skip Lists . . . . .	222
<b>22 Monte Carlo Methods and Concentration Bounds</b>	<b>228</b>
22.1 Variance and Chebyshev's Inequality . . . . .	229
22.2 Hoeffding's Inequality . . . . .	234
22.3 Polling . . . . .	236
22.4 Load Balancing . . . . .	238
<b>V Cryptography</b>	<b>241</b>
<b>23 Introduction to Cryptography</b>	<b>242</b>
23.1 Review of Modular Arithmetic . . . . .	243
23.2 One-time Pad . . . . .	248
<b>24 Diffie-Hellman Key Exchange</b>	<b>253</b>
<b>25 RSA</b>	<b>256</b>
25.1 RSA Signatures . . . . .	260
25.2 Quantum Computers and Cryptography . . . . .	261
<b>VI Supplemental Material</b>	<b>262</b>
<b>26 Supplemental: Algorithms</b>	<b>263</b>
26.1 Non-master-theorem Recurrences . . . . .	263
<b>27 Supplemental: Computability</b>	<b>265</b>
27.1 Applying Rice's Theorem . . . . .	265
27.2 Computable Functions and Kolmogorov Complexity . . . . .	267
<b>28 Supplemental: Randomness</b>	<b>270</b>
28.1 Primality Testing . . . . .	270
28.2 Multiplicative Chernoff Bounds . . . . .	274
28.3 Probabilistic Complexity Classes . . . . .	282
28.4 Amplification for Two-Sided-Error Algorithms . . . . .	287
<b>VII Appendix</b>	<b>290</b>
<b>29 Appendix</b>	<b>291</b>
29.1 Proof of the Master Theorem . . . . .	291
29.2 Alternative Analysis of Quick Sort . . . . .	296
29.3 Proof of the Simplified Multiplicative Chernoff Bounds . . . . .	300
29.4 Proof of the Upper-Tail Hoeffding's Inequality . . . . .	302
29.5 General Case of Hoeffding's Inequality . . . . .	306

<b>VIII About</b>	<b>310</b>
<b>30 About</b>	<b>311</b>
<b>Index</b>	<b>312</b>

# **Part I**

# **Algorithms**

## INTRODUCTION

Every complex problem has a solution that is clear, simple, and wrong. — H. L. Mencken

Welcome to *Foundations of Computer Science*! This text covers foundational aspects of Computer Science that will help you reason about any computing task. In particular, we are concerned with the following with respect to problem solving:

- What are common, effective approaches to designing an algorithm?
- Given an algorithm, how do we reason about whether it is correct and how efficient it is?
- Are there limits to what problems we can solve with computers, and how do we identify whether a particular problem is solvable?
- What problems are *efficiently* solvable, and how do we determine whether a particular problem is?
- For problems that seem not to be solvable efficiently, can we efficiently find *approximate* solutions, and what are common techniques for doing so?
- Can randomness help us in solving problems?
- How can we exploit problems that are not efficiently solvable to build secure cryptography algorithms?

In order to answer these questions, we must define formal mathematical models and apply a proof-based methodology. Thus, this text will feel much like a math text, but we apply the approach directly to widely applicable problems in Computer Science.

As an example, how can we demonstrate that there is no general algorithm for determining whether or not two programs have the same functionality? A simple but incorrect approach would be to analyze all possible algorithms, and show that none can work. However, there are infinitely many possible algorithms, so we have no hope of this approach working. Instead, we need to construct a model that captures the notion of what is computable by any algorithm, and use that to demonstrate that no such algorithm exists.

### 1.1 Text Objectives

The main purpose of this text is to give you the tools to approach computational problems you've never seen before. Rather than being given a particular algorithm or data structure to implement, approaching a new problem requires reasoning about whether the problem is solvable, how to relate it to problems that you've seen before, what algorithmic techniques are applicable, whether the algorithm you come up with is correct, and how efficient the resulting algorithm is. These are all steps that must be taken *prior* to actually writing code to implement a solution, and these steps are independent of your choice of programming language or framework.

Thus, in a sense, the fact that you will not have to write code (though you are free to do so if you like) is a feature, not a bug. We focus on the prerequisite algorithmic reasoning required before writing any code, and this reasoning is independent of the implementation details. If you were to implement all the algorithms you design in this course, the workload would be far greater, and it would only replicate the coding practice you get in your programming courses.

Instead, we focus on the aspects of problem solving that you have not yet had much experience in. The training we give you in this text will make you a better programmer, as algorithmic design and analysis is crucial to effective programming. This text also provides a solid framework for further exploration of theoretical Computer Science, should you wish to pursue that path. However, you will find the material here useful regardless of which subfields of Computer Science you decide to study.

As an example, suppose your boss tells you that you need to make a business trip to visit several cities, and you must minimize the cost of visiting all those cities. This is an example of the classic [traveling salesperson problem](#)<sup>1</sup>, and you may have heard that it is an intractable problem. What exactly does that mean? Does it mean that it cannot be solved at all? What if we change the problem so that you don't have to minimize the cost, but instead must fit the cost within a given budget (say \$2000)? Does this make the problem any easier? What if we don't require that the total cost be minimized, but that it merely needs to be within a factor of two of the optimal cost?

Before we can reason about the total cost of a trip, we need to know how much it costs to travel between two consecutive destinations in the trip. Suppose we are traveling by air. There may be no direct flight between those two cities, or it might be very expensive. To help us keep our budget down, we need to figure out what the cheapest itinerary between those two cities is, considering intermediate layover stops. And since we don't know *a priori* in what order we will visit all the cities, we need to know the cheapest itineraries between all pairs of cities. This is an instance of the [all-pairs shortest path problem](#)<sup>2</sup>. Is this an “efficiently solvable” problem, and if so, what algorithmic techniques can we use to find a solution?

We will consider both of the problems above in this text. We will learn what it means for a problem to be solvable or not (and how to prove this), what it means for a problem to be tractable or not (and how to prove that it is, or give strong evidence that it is not), and techniques for designing and analyzing algorithms for tractable problems.

## 1.2 Tools for Abstraction

*Abstraction* is a core principle in Computer Science, allowing us to reason about and use complex systems without needing to pay attention to implementation details. As mentioned earlier, the focus of this text is on reasoning about problems and algorithms independently of specific implementations. To do so, we need appropriate abstract models that are applicable to any programming language or system architecture.

Our abstract model for expressing algorithms is *pseudocode*, which describes the steps in the algorithm at a high level without implementation details. As an example, the following is a pseudocode description of the [Floyd-Warshall algorithm](#) (page 38), which we will discuss later:

### Algorithm 1 (Floyd-Warshall)

```

Input: a weighted directed graph
Output: all-pairs (shortest-path) distances in the graph
function FLOYDWARSHALL( $G = (V, E)$ )
    for all  $u, v \in V$  do
         $d_0(u, v) = \text{weight}(u, v)$ 
    for  $k = 1$  to  $|V|$  do
        for all  $u, v \in V$  do
             $d_k(u, v) = \min\{d_{k-1}(u, v), d_{k-1}(u, k) + d_{k-1}(k, v)\}$ 
    return  $d_{|V|}$ 
```

This description is independent of how the graph  $G$  is represented, or the two-dimensional matrices  $d_i$ , or the specific syntax of the loops. Yet it should be clear to the intended reader what each step of the algorithm does. Expressing this algorithm in a real-world programming language like C++ would only add unnecessary syntactic and implementation

<sup>1</sup> [https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem)

<sup>2</sup> [https://en.wikipedia.org/wiki/Shortest\\_path\\_problem#All-pairs\\_shortest\\_paths](https://en.wikipedia.org/wiki/Shortest_path_problem#All-pairs_shortest_paths)

details that are an artifact of the chosen language and data structures, and not intrinsic to the algorithm itself. Instead, pseudocode gives us a simple means of expressing an algorithm that facilitates understanding and reasoning about its core elements.

We also need abstractions for reasoning about the efficiency of algorithms. The actual real-world running time and space/memory usage of a program depend on many factors, including choice of language and data structures, available compiler optimizations, and characteristics of the underlying hardware. Again, these are not intrinsic to an algorithm itself. Instead, we focus not on concrete real-world efficiency, but on how the algorithm's running time (and sometime memory usage) *scales* with respect to the input size. Specifically, we focus on time complexity in terms of:

1. the number of basic operations performed as a function of input size,
2. asymptotically, i.e., as the input size grows,
3. ignoring leading constants,
4. over worst-case inputs.

We measure space complexity in a similar manner, but with respect to the number of memory cells used, rather than the number of basic operations performed. These measures of time and space complexity allow us to evaluate algorithms in the abstract, rather than with respect to a particular implementation. This is not to say that absolute performance is irrelevant. Rather, the asymptotic complexity gives us a “higher-level” view of an algorithm. An algorithm with poor asymptotic time complexity will be inefficient when run on large inputs, regardless of how good the implementation is.

Later in the text, we will also reason about the intrinsic solvability of a problem. We will see how to express problems in the abstract (e.g. as *languages* and *decision problems*), and we will examine a simple model of computation (*Turing machines*) that captures the essence of computation.

## 1.3 The First Algorithm: Euclid’s GCD

One of the oldest known algorithms is Euclid’s algorithm for computing the *greatest common divisor (GCD)* of two integers.

**Definition 2 (Divides, Divisor)** Let  $x \in \mathbb{Z}$  be an integer. We say that an integer  $d$  *divides*  $x$  (and is a *divisor* of  $x$ ) if there exists an integer  $k \in \mathbb{Z}$  such that  $d \cdot k = x$ . (When  $d \neq 0$ , this is equivalent to  $x/d$  being an integer.)

Whether  $d$  divides  $x$  is not affected by their signs (positive, negative, or zero), so from now on we restrict our attention to nonnegative integers and divisors.

Note:  $d = 1$  divides *any* integer  $x$ , by taking  $k = x$  (i.e.,  $1 \cdot x = x$ ), and  $d = x$  is the largest divisor of  $x$  when  $x > 0$ . Take care with the special cases involving zero: *any* integer  $d$  divides  $x = 0$ , because  $d \cdot 0 = 0$ . But  $d = 0$  does not divide anything except  $x = 0$ , because  $d \cdot k = 0 \cdot k = 0$  for every  $k$ .

**Definition 3 (Greatest Common Divisor)** Let  $x, y \in \mathbb{Z}$  be nonnegative integers. A *common divisor* of  $x, y$  is an integer that divides both of them, and their *greatest common divisor*, denoted  $\gcd(x, y)$ , is the largest such integer.

For example,  $\gcd(21, 9) = 3$  and  $\gcd(121, 5) = 1$ . Also,  $\gcd(7, 7) = \gcd(7, 0) = 7$ . (Make sure you understand why!) If  $\gcd(x, y) = 1$ , we say that  $x$  and  $y$  are *coprime*. So, 121 and 5 are coprime, but 21 and 9 are not coprime (nor are 7 and 7, nor are 7 and 0).

Take note: as long as  $x, y$  are not both zero,  $\gcd(x, y)$  is well defined, because there is at least one common divisor  $d = 1$ , and no common divisor can be greater than  $\max(x, y)$ . However,  $\gcd(0, 0)$  is not well defined, because every integer divides zero, and there is no largest integer. In this case, it is convenient to define  $\gcd(0, 0) = 0$ , so that  $\gcd(x, 0) = x$  for all  $x$ .

So far we have just defined the GCD mathematically. Now we consider the computational question: given two integers, can we *compute* their GCD, and how efficiently can we do so? As we will see later, this problem turns out to be very important in cryptography and other fields.

Here is a naïve, “brute-force” algorithm for computing the GCD of given integers  $x \geq y$  (we adopt this requirement for convenience, since we can swap the values without changing the answer): try every integer from  $y$  down to 1, check whether it divides both  $x$  and  $y$ , and return the first (and hence largest) such number that does. The algorithm is clearly correct, because the GCD of  $x$  and  $y$  cannot exceed  $y$ , and the algorithm returns the first (and hence largest) value that actually divides both arguments.

**Algorithm 4 (Naïve GCD)**

**Input:** integers  $x \geq y \geq 0$ , not both zero  
**Output:** their greatest common divisor  $\gcd(x, y)$

```
function NAIVEGCD(x, y)
    for d = y down to 1 do
        if d divides both x and y then return d
```

Here, the mod operation computes the remainder of the first operand divided by the second. For example,  $9 \bmod 6 = 3$  since  $9 = 6 \cdot 1 + 3$ , and  $9 \bmod 3 = 0$  since  $9 = 3 \cdot 3 + 0$ . So, the result of  $x \bmod d$  is an integer in the range  $[0, d-1]$ , and in particular  $x \bmod 1 = 0$ . The result is not defined when  $d$  is 0.

How efficient is the above algorithm? In the worst case, it performs two mod operations for every integer in the range  $[1, y]$ . Using asymptotic notation, the worst-case number of mod operations is therefore  $\Theta(y)$ . (Recall that this  $\Theta(y)$  notation means: between  $cy$  and  $c'y$  for some positive constants  $c, c'$ , for all “large enough”  $y$ .)

Can we do better?

Here is a key observation: if  $d$  divides both  $x$  and  $y$ , then it also divides  $x - my$  for any integer  $m \in \mathbb{Z}$ . Here is the proof: since  $x = d \cdot a$  and  $y = d \cdot b$  for some integers  $a, b \in \mathbb{Z}$ , then  $x - my = da - mdb = d \cdot (a - mb)$ , so  $d$  divides  $x - my$  as well. By the same kind of reasoning, the converse holds too: if some  $d'$  divides both  $x - my$  and  $y$ , it also divides  $x$ . Thus, the common divisors of  $x$  and  $y$  are exactly the common divisors of  $x - my$  and  $y$ , and hence the *greatest* common divisors of these two pairs are equal. We have just proved the following:

**Lemma 5** For all  $x, y, m \in \mathbb{Z}$ , we have  $\gcd(x, y) = \gcd(x - my, y)$ .

Since any  $m \in \mathbb{Z}$  will do, let’s choose  $m$  to minimize  $x - my$ , without making it negative. As long as  $y \neq 0$ , we can do so by taking  $m = \lfloor x/y \rfloor$ , the integer quotient of  $x$  and  $y$ . Then  $r = x - my = x - \lfloor x/y \rfloor y$  is simply the remainder of  $x$  when divided by  $y$ , i.e.,  $r = x \bmod y$ .

The above results in the following corollary (where the second equality holds because the GCD is symmetric):

**Corollary 6** For all  $x, y \in \mathbb{Z}$  with  $y \neq 0$ , we have  $\gcd(x, y) = \gcd(x \bmod y, y) = \gcd(y, x \bmod y)$ .

(The rightmost expression maintains our convention that the first argument of gcd should be greater than or equal to the second.)

We have just derived the key *recurrence relation* for gcd, which we will use as the heart of an algorithm. However, we also need base cases. As mentioned previously,  $x \bmod y$  is defined only when  $y \neq 0$ , so we need a base case for  $y = 0$ . As we have already seen,  $\gcd(x, 0) = x$  (even when  $x = 0$ ). (We can also observe that  $\gcd(x, 1) = 1$  for all  $x$ . This latter base case is not technically necessary; some descriptions of Euclid’s algorithm include it, while others do not.)

This leads us to the Euclidean algorithm:

**Algorithm 7 (Euclid’s Algorithm)**

**Input:** integers  $x \geq y \geq 0$ , not both zero  
**Output:** their greatest common divisor  $\gcd(x, y)$

```
function EUCLID(x, y)
    if y = 0 then return x
    return EUCLID(y, x mod y)
```

Here are some example runs of this algorithm:

**Example 8**

```
EUCLID(21,9)
= EUCLID(9,3)
= EUCLID(3,0)
= 3
```

**Example 9**

```
EUCLID(30,19)
= EUCLID(19,11)
= EUCLID(11,8)
= EUCLID(8,3)
= EUCLID(3,2)
= EUCLID(2,1)
= EUCLID(1,0)
= 1
```

**Example 10**

```
EUCLID(376281,376280)
= EUCLID(376280,1)
= EUCLID(1,0)
= 1
```

How efficient is this algorithm? Clearly, it does one mod operation per iteration—or more accurately, recursive call—but it is no longer obvious how many iterations it performs. For instance, the computation of  $\text{EUCLID}(30, 19)$  does six iterations, while  $\text{EUCLID}(376281, 376280)$  does only two. There doesn't seem to be an obvious relationship between the form of the input and the number of iterations.

This is an extremely simple algorithm, consisting of just a few lines of code. But that does not make it simple to reason about. We need new techniques to analyze code like this and determine its time complexity.

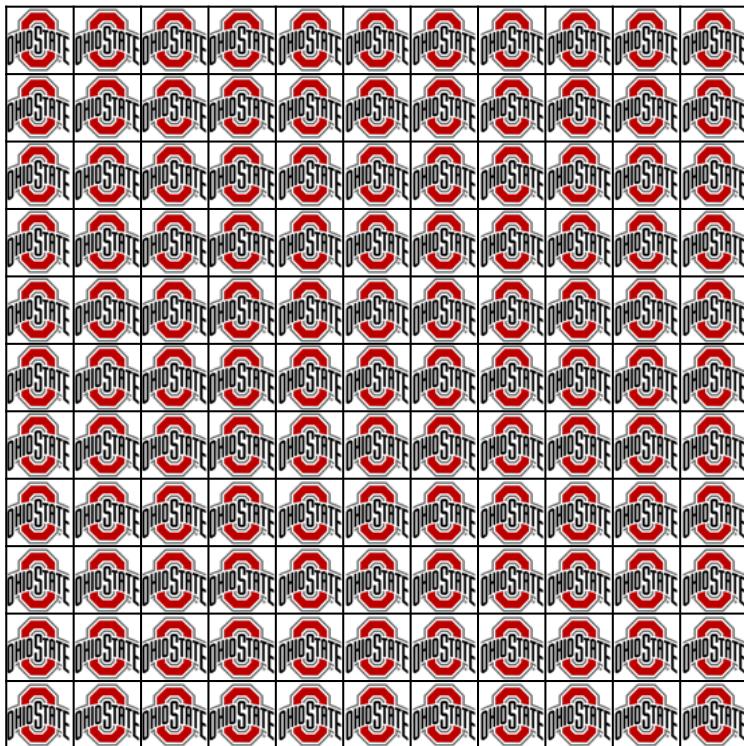
---

**CHAPTER  
TWO**

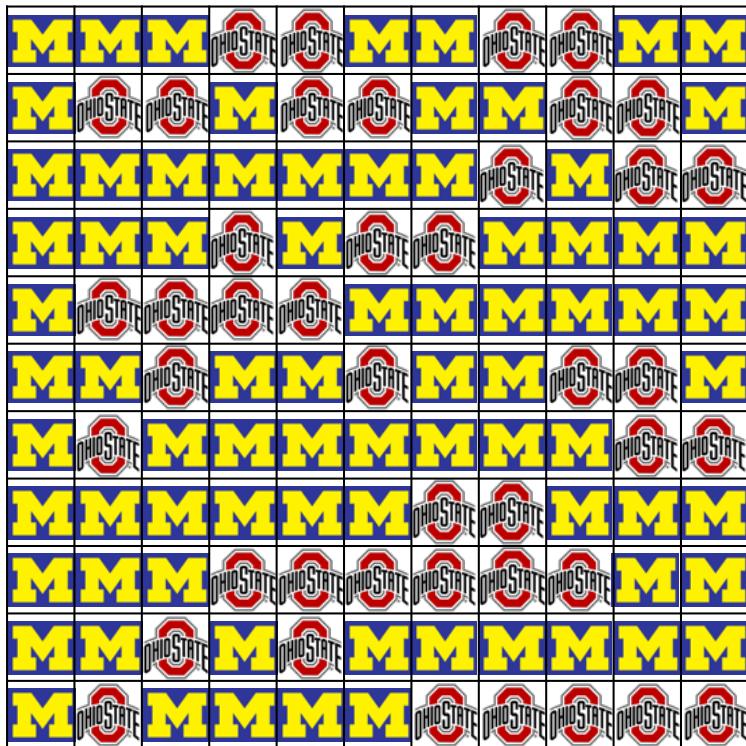
---

## **THE POTENTIAL METHOD**

Let's set aside Euclid's algorithm for the moment and examine a game instead. Consider a "flipping game" that has an  $11 \times 11$  board covered with two-sided chips, say  on the front side and  on the back. Initially, the entirety of the board is covered with every chip face down.



The game pits a row player against a column player, and both take turns flipping an entire row or column, respectively. A row or column may be flipped only if it contains more face-down (OSU) than face-up (UM) chips. The game ends when a player can make no legal moves, and that player loses the game. For example, if the game reaches the following state and it is the column player's move, the column player has no possible moves and therefore loses.



Let's set aside strategy and ask a simpler question: must the game end in a finite number of moves, or is there a way to play the game in a way that continues indefinitely?

An  $11 \times 11$  board is rather large to reason about, so a good strategy is to simplify the problem by considering a  $3 \times 3$  board instead. Let's consider the following game state.



Suppose that it is the row player's turn, and the player chooses to flip the bottom row.



Notice that in general, a move may flip some chips from UM to OSU, and others vice versa. This move in particular flipped the first chip in the row from UM to OSU, and the latter two chips from OSU to UM. The number of chips flipped in each direction depends on the state of a row or column, and it is not generally the case that every move flips three OSU chips to UM in the  $3 \times 3$  board.

Continuing the game, the column player only has a single move available.



Again, one UM chip is flipped to OSU, and two OSU chips are flipped to UM.

It is again the row player's turn, but now no row flips are possible. The game ends, with the victory going to the column player.

We observe that each move flipped both UM and OSU chips, but each move flipped *more* OSU chips to UM than vice versa. Is this always the case? Indeed it is, by rule: a move is legal only if the flipped row or column has more OSU than UM chips. The move flips each OSU chip to a UM one, and each UM chip to an OSU chip. Since there are more OSU than UM chips, more OSU-to-UM flips happen than vice versa. More formally and generally, for an  $n \times n$  board, an individual row or column has  $k$  OSU chips and  $n - k$  UM chips, for some value of  $k$ . A move is legal when  $k > n - k$ . After the flip, the row or column will have  $k$  UM chips and  $n - k$  OSU chips. The net change in the number of UM chips is  $k - (n - k)$ , which is positive because  $k > n - k$ . Thus, each move strictly increases the number of UM chips, and strictly decreases the number of OSU chips.

In the  $3 \times 3$  case, we start with nine OSU chips. No board configuration can have fewer than zero OSU chips. Then because each move decreases the number of OSU chips, no more than nine moves are possible before the game must end. By the same reasoning, no more than 121 moves are possible in the original  $11 \times 11$  game.

Strictly speaking, it will always take fewer than 121 moves to reach a configuration where a player has no moves available, because the first move decreases the number of OSU chips by eleven. But we don't need an exact number to answer our original question. We have proved an upper bound of 121 on the number of moves, so we have established that any valid game must indeed end after a finite number of moves.

The core of the above reasoning is that we defined a special measure of the board's state, namely, the number of OSU chips. We observe that in each step, this measure must decrease (by at least one). The measure of the initial state is finite, and there is a lower bound the measure cannot go below, so eventually that lower bound must be reached.

This pattern of reasoning is called the *potential method*. Formally, given some set  $A$  of "states" (e.g., game states, algorithm states, etc.), let  $s: A \rightarrow \mathbb{R}$  be a function that maps states to numbers. The function  $s$  is a *potential function* if:

1. it strictly decreases with every state transition (e.g., turn in a game, step of an algorithm, etc.)<sup>3</sup>
2. it is lower-bounded by some fixed value: there is some  $\ell \in \mathbb{R}$  for which  $s(a) \geq \ell$  for all  $a \in A$ .

By defining a valid potential function and establishing both its lower bound and how quickly it decreases, we can upper bound the number of steps a complex algorithm may take.

<sup>3</sup> We need to be a bit more precise to ensure that  $s$  reaches its lower bound in a finite number of steps. A sufficient condition is that  $s$  decreases by at least some *fixed constant*  $c$  in each step. In the game example, we established that  $s$  decreases by at least  $c = 1$  in each turn.

## 2.1 A Potential Function for Euclid's Algorithm

Let's return to Euclid's algorithm and try to come up with a potential function we can use to reason about its efficiency. Specifically, we want to determine an upper bound on the number of iterations the algorithm takes for a given input  $x$  and  $y$ .

First observe that  $x$  and  $y$  do not increase from one step to the next. For instance,  $\text{EUCLID}(30, 19)$  calls  $\text{EUCLID}(19, 11)$ , so  $x$  decreases from 30 to 19 and  $y$  decreases from 19 to 11. However, the amount each argument individually decreases can vary. In  $\text{EUCLID}(376281, 376280)$ ,  $x$  decreases by only one in the next iteration, while  $y$  decreases by 376279. In  $\text{EUCLID}(7, 7)$ ,  $x$  does not decrease at all in the next iteration, but  $y$  decreases by 7.

Since the algorithm has two arguments, both of which typically change as the algorithm proceeds, it seems reasonable to define a potential function that takes both into account. Let  $x_i$  and  $y_i$  be the values of the two variables in the  $i$ th iteration (recursive call). So,  $x_0 = x$  and  $y_0 = y$ , where  $x$  and  $y$  are the original inputs to the algorithm;  $x_1, y_1$  are the arguments to the first recursive call; and so on. As a candidate potential function, we try a simple sum of the two arguments:

$$s_i = x_i + y_i .$$

Before we look at some examples, let's first establish that this is a valid potential function. Examining the algorithm, when  $y_i \neq 0$  we see that

$$\begin{aligned} s_{i+1} &= x_{i+1} + y_{i+1} \\ &= y_i + r_i , \end{aligned}$$

where  $r_i = x_i \bmod y_i$ . Given the invariant maintained by the algorithm that  $x_i \geq y_i$ , and that  $x_i \bmod y_i \in [0, y_i - 1]$ , we have that  $r_i < y_i \leq x_i$ . Therefore,

$$\begin{aligned} s_{i+1} &= y_i + r_i \\ &< y_i + x_i \\ &= s_i . \end{aligned}$$

Thus, the potential  $s$  always decreases by at least one (because  $s$  is a natural number) from one iteration to the next, satisfying the first requirement of a potential function. We also observe that  $y_i \geq 0$  for all  $i$ ; coupled with  $x_i \geq y_i$ , and the fact that both arguments are not zero, we get that  $s_i \geq 1$  for all  $i$ . Since we have established a lower bound on  $s$ , it meets the second requirement of a potential function.

At this point, we can conclude that Euclid's algorithm  $\text{EUCLID}(x, y)$  performs *at most*  $x + y$  iterations. (This is because the initial potential is  $x + y$ , each iteration decreases the potential by at least one, and the potential is always greater than zero.) However, this bound is no better than the one we derived for the brute-force GCD algorithm. So, have we just been wasting our time here?

Fortunately, we have not! As we will soon show, this  $x + y$  upper bound for  $\text{EUCLID}(x, y)$  is very *loose*, and in fact the actual number of iterations is much smaller. We will prove this by showing that the potential decreases *much faster* than we previously considered.

As an example, let's look at the values of the potential function for the execution of  $\text{EUCLID}(21, 9)$ :

$$\begin{aligned} s_0 &= 21 + 9 = 30 \\ s_1 &= 9 + 3 = 12 \\ s_2 &= 3 + 0 = 3 . \end{aligned}$$

And the following are the potential values for  $\text{EUCLID}(8, 5)$ :

$$\begin{aligned} s_0 &= 8 + 5 = 13 \\ s_1 &= 5 + 3 = 8 \\ s_2 &= 3 + 2 = 5 \\ s_3 &= 2 + 1 = 3 \\ s_4 &= 1 + 0 = 1 . \end{aligned}$$

The values decay rather quickly for  $\text{EUCLID}(21, 9)$ , and somewhat more slowly for  $\text{EUCLID}(8, 5)$ . But the key observation is that they appear to decay *multiplicatively* (by some factor), rather than additively. In these examples, the ratio of  $s_{i+1}/s_i$  is largest for  $s_2/s_1$  in  $\text{EUCLID}(8, 5)$ , where it is 0.625. In fact, we will prove an upper bound that is not far from that value.

**Lemma 11** For all valid inputs  $x, y$  to Euclid's algorithm,  $s_{i+1} \leq \frac{2}{3}s_i$  for every iteration  $i$  of  $\text{EUCLID}(x, y)$ .

The recursive case of  $\text{EUCLID}(x_i, y_i)$  invokes  $\text{EUCLID}(y_i, x_i \bmod y_i)$ , so  $x_{i+1} = y_i$  and  $y_{i+1} = r_i = x_i \bmod y_i$  (the remainder of dividing  $x_i$  by  $y_i$ ). By definition of remainder, we can express  $x_i$  as

$$x_i = q_i \cdot y_i + r_i,$$

where  $q_i = \lfloor x_i/y_i \rfloor$  is the integer quotient of  $x_i$  divided by  $y_i$ . Since  $x_i \geq y_i$ , we have that  $q_i \geq 1$ . Then:

$$\begin{aligned} s_i &= x_i + y_i \\ &= q_i \cdot y_i + r_i + y_i \quad (\text{substituting } x_i = q_i \cdot y_i + r_i) \\ &= (q_i + 1) \cdot y_i + r_i \\ &\geq 2y_i + r_i \quad (\text{since } q_i \geq 1). \end{aligned}$$

We are close to what we need to relate  $s_i$  to  $s_{i+1} = y_i + r_i$ , but we would like a common multiplier for both the  $y_i$  and  $r_i$  terms. Let's split the difference by adding  $r_i/2$  and subtracting  $y_i/2$ :

$$\begin{aligned} s_i &\geq 2y_i + r_i \\ &> 2y_i + r_i - \frac{y_i - r_i}{2} \quad (\text{since } r_i < y_i). \end{aligned}$$

The latter step holds because  $r_i$  is the remainder of dividing  $x_i$  by  $y_i$ , so  $y_i - r_i > 0$ . (And subtracting a positive number makes a quantity smaller.)

Continuing onward, we have:

$$\begin{aligned} s_i &\geq 2y_i + r_i - \frac{y_i - r_i}{2} \\ &= \frac{3}{2}(y_i + r_i) \\ &= \frac{3}{2}s_{i+1}. \end{aligned}$$

Rearranging the inequality, we conclude that  $s_{i+1} \leq \frac{2}{3}s_i$ .

By repeated applications of the above lemma (i.e., induction), starting with  $s_0 = x_0 + y_0 = x + y$ , we can conclude that:

**Corollary 12** For all valid inputs  $x, y$  to Euclid's algorithm,  $s_i \leq (\frac{2}{3})^i(x + y)$  for all iterations  $i$  of  $\text{EUCLID}(x, y)$ .

We can now prove the following:

**Theorem 13 (Time Complexity of Euclid's Algorithm)** For any valid inputs  $x, y$ ,  $\text{EUCLID}(x, y)$  performs  $O(\log(x + y))$  iterations (and mod operations).

We have previously shown that  $1 \leq s_i \leq (\frac{2}{3})^i(x + y)$  for the  $i$ th iteration of  $\text{EUCLID}(x, y)$ . Therefore,

$$\begin{aligned} 1 &\leq (\frac{2}{3})^i(x + y) \\ (\frac{3}{2})^i &\leq x + y \\ i &\leq \log_{3/2}(x + y) \quad (\text{taking the base-(3/2) log of both sides}). \end{aligned}$$

We have just established an upper bound on  $i$ , which means that the number of iterations cannot exceed  $\log_{3/2}(x+y) = O(\log(x+y))$ . Indeed, in order for  $i$  to exceed this quantity, it would have to be the case that  $1 > (2/3)^i(x+y)$ , leaving no possible value for the associated potential  $s_i$ —an impossibility! (Also recall that we can change the base of a logarithm by multiplying by a suitable constant, and since  $O$ -notation ignores constant factors, the base in  $O(\log(x+y))$  does not matter. Unless otherwise specified, the base of a logarithm is assumed to be 2 in this text.) Since each iteration does at most one mod operation, the total number of mod operations is also  $O(\log(x+y))$ , completing the proof.

Under our convention that  $x \geq y$ , we have that  $O(\log(x+y)) = O(\log 2x) = O(\log x)$ . Recall that the naïve algorithm did  $\Theta(x)$  iterations and mod operations (in the worst case). This means that Euclid's algorithm is *exponentially faster* than the naïve algorithm!

We have seen that the potential method gives us an important tool in reasoning about the complexity of algorithms, enabling us to establish an upper bound on the runtime of Euclid's algorithm.

## DIVIDE AND CONQUER

The *divide-and-conquer* algorithmic paradigm involves subdividing a large problem instance into smaller instances of the same problem. The subinstances are solved recursively, and then their solutions are combined in some appropriate way to construct a solution for the original larger instance.

Since divide and conquer is a recursive paradigm, the main tool for analyzing divide-and-conquer algorithms is induction. When it comes to complexity analysis, such algorithms generally give rise to recurrence relations expressing the time or space complexity. While these relations can be solved inductively, certain patterns are common enough that higher-level tools have been developed to handle them. We will see one such tool in the form of the master theorem.

As an example of a divide-and-conquer algorithm, the following is a description of the *merge sort* algorithm for sorting mutually comparable items:

**Algorithm 14 (Merge Sort)**

**Input:** an array of elements that can be ordered

**Output:** a sorted array of the same elements

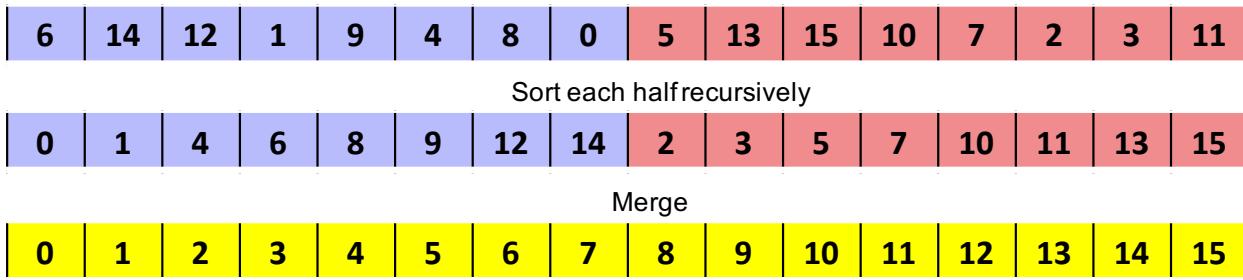
```
function MERGESORT( $A[1, \dots, n]$ )
    if  $n = 1$  then return  $A$ 
     $m = \lfloor n/2 \rfloor$ 
     $L = \text{MERGESORT}(A[1, \dots, m])$ 
     $R = \text{MERGESORT}(A[m + 1, \dots, n])$ 
    return MERGE( $L, R$ )
```

**Input:** two sorted arrays

**Output:** a sorted array of the same elements

```
function MERGE( $L[1, \dots, \ell], R[1, \dots, r]$ )
    if  $\ell = 0$  then return  $R$ 
    if  $r = 0$  then return  $L$ 
    if  $L[1] \leq R[1]$  then
        return  $L[1] : \text{MERGE}(L[2, \dots, \ell], R[1, \dots, r])$ 
    else
        return  $R[1] : \text{MERGE}(L[1, \dots, \ell], R[2, \dots, r])$ 
```

The algorithm sorts an array by first recursively sorting its two halves, then combining the sorted halves with the *merge* operation. Thus, it follows the pattern of a divide-and-conquer algorithm.



A naïve algorithm such as insertion sort<sup>4</sup> has a time complexity of  $\Theta(n^2)$ . How does merge sort compare?

Define  $T(n)$  to be the total number of basic operations (array indexes, element comparisons, etc.) performed by MERGESORT on an array of  $n$  elements. Similarly, let  $S(n)$  be the number of basic operations apart from the recursive calls themselves: testing whether  $n = 1$ , splitting the input array into halves, the cost of MERGE on the two halves, etc. Then we have the following recurrence for  $T(n)$ :

$$T(n) = 2T(n/2) + S(n).$$

This is because on an array of  $n$  elements, MERGESORT makes two recursive calls to itself on some array of  $n/2$  elements, each of which takes  $T(n/2)$  time by definition, and all its other non-recursive work takes  $S(n)$  by definition. (For simplicity, we ignore the floors and ceilings for  $n/2$ , which do not affect the ultimate asymptotic bounds.)

Observe that the merge step does  $n$  comparisons and  $\sim n$  array concatenations. Assuming that each of these operations takes a constant amount of time (that does not grow with  $n$ )<sup>5</sup>, we have that  $S(n) = O(n)$ . So,

$$T(n) = 2T(n/2) + O(n).$$

How can we solve this recurrence, i.e., express  $T(n)$  in a “closed form” that depends only on  $n$  (and does not refer to  $T$  itself)? While we can do so using induction or other tools for solving recurrence relations, this can be a lot of work. Thankfully, there is a special tool called the Master Theorem that directly yields a solution to this recurrence and many others like it.

## 3.1 The Master Theorem

Suppose we have some recursive divide-and-conquer algorithm that solves an input of size  $n$ :

- recursively solving some  $k \geq 1$  smaller inputs,
- each of size  $n/b$  for some  $b > 1$  (as before, ignoring floors and ceilings),
- where the total cost of all the “non-recursive work” (splitting the input, combining the results, etc.) is  $O(n^d)$ .

Then the running time  $T(n)$  of the algorithm follows the recurrence relation

$$T(n) = kT(n/b) + O(n^d).$$

The *Master Theorem* provides the solutions to such recurrences.<sup>6</sup>

<sup>4</sup> [https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)

<sup>5</sup> Using a linked-list data structure, adding an element to the front does indeed take a constant amount of time; for arrays, with care it is also possible to implement MERGE in linear time. On the other hand, the assumption of constant-time comparisons typically holds only for fixed-size data types, such as 32-bit integers or 64-bit floating-point numbers. For arbitrary-size numbers or other variable-length data types like strings whose sizes might grow with  $n$ , this assumption does not hold, and the cost of comparisons needs to be considered more carefully.

<sup>6</sup> Refer to the *appendix* (page 291) for proofs of this master theorem, as well as a more general version with log factors.

**Theorem 15 (Master Theorem)** Let  $k \geq 1, b > 1, d \geq 0$  be constants that do not vary with  $n$ , and let  $T(n)$  be a recurrence with base case  $T(1) = O(1)$  having the following form, ignoring ceilings/floors on (or more generally, addition/subtraction of any constant to) the  $n/b$  argument on the right-hand side:

$$T(n) = k \cdot T(n/b) + O(n^d).$$

Then this recurrence has the solution

$$T(n) = \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d. \end{cases}$$

In addition, the above bounds are tight: if the  $O$  in the recurrence is replaced with  $\Theta$ , then it is in the solution as well.

Observe that the test involving  $k$ ,  $b$ , and  $d$  can be expressed in logarithmic form, by taking base- $b$  logarithms and comparing  $\log_b k$  to  $d$ .

In the case of merge sort, we have  $k = b = 2$  and  $d = 1$ , so  $k = b^d$ , so the solution is  $T(n) = O(n \log n)$  (and this is tight). Thus, merge sort is much more efficient than insertion sort! (As always in this text, this is merely an *asymptotic* statement, for large enough  $n$ .)

We emphasize that in order to apply (this version of) the Master Theorem, the values  $k, b, d$  must be *constants* that do not vary with  $n$ . For example, the theorem does not apply to a divide-and-conquer algorithm that recursively solves  $k = \sqrt{n}$  subinstances, or one whose subinstances are of size  $n/\log n$ . In such a case, a different tool is needed to solve the recurrence. Fortunately, the Master Theorem does apply to the vast majority of divide-and-conquer algorithms of interest.

### 3.1.1 Master Theorem with Log Factors

A recurrence such as

$$T(n) = 2T(n/2) + O(n \log n)$$

does not exactly fit the form of the master theorem above, since the additive term  $O(n \log n)$  does not look like  $O(n^d)$  for some constant  $d$ .<sup>7</sup> Such a recurrence can be handled by a more general form of the theorem, as follows.

**Theorem 16** Let  $T(n)$  be the following recurrence, where  $k \geq 1, b > 1, d \geq 0, w \geq 0$  are constants that do not vary with  $n$ :

$$T(n) = kT(n/b) + O(n^d \log^w n).$$

Then:

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } k < b^d \\ O(n^d \log^{w+1} n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d. \end{cases}$$

Applying the generalized master theorem to the recurrence

<sup>7</sup> Actually, it can be made to fit this form, since any  $O(n \log n)$  function is also, say,  $O(n^{1.001})$ —because  $O$  represents an *upper bound*, and  $\log n = O(n^\varepsilon)$  for an arbitrary positive constant  $\varepsilon > 0$ . However, this substitution is not *tight*—it does not hold with  $\Theta$  in place of  $O$ —and adding a tiny amount to the exponent is clunky.

$$T(n) = 2T(n/2) + O(n \log n),$$

we have  $k = 2, b = 2, d = 1, w = 1$ , so  $k = b^d$ . Therefore,

$$\begin{aligned} T(n) &= O(n^d \log^{w+1} n) \\ &= O(n \log^2 n). \end{aligned}$$

## 3.2 Integer Multiplication

We now turn our attention to algorithms for integer multiplication. For fixed-size data types, such as 32-bit integers, multiplication can be done in a constant amount of time, and is typically implemented as a hardware instruction for common sizes. However, if we are working with arbitrary  $n$ -bit numbers, we will have to implement multiplication ourselves in software. (As we will see later, multiplication of such “big” integers is essential to many cryptography algorithms.)

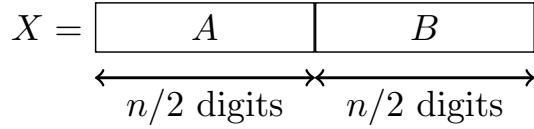
Let’s first take a look the standard grade-school long-multiplication algorithm.

$$\begin{array}{r}
 &&1&1&1&0&1&1\leftarrow 59 \\
 &\times&1&0&1&0&1&0\leftarrow 42 \\
 \hline
 &=&&1&1&1&0&1&1&59<<1 \\
 &+&&1&1&1&0&1&1&59<<3 \\
 &+&&1&1&1&0&1&1&59<<5 \\
 \hline
 2478\longrightarrow &=&1&0&0&1&1&0&1&0&1&1&0
 \end{array}$$

Here, the algorithm is illustrated for binary numbers, but it works the same as for decimal numbers, just in base two. We first multiply the top number by the last (least-significant) digit in the bottom number. We then multiply the top number by the second-to-last digit in the bottom number, but we shift the result leftward by one digit. We repeat this for each digit in the bottom number, adding one more leftward shift with each digit. Once we have done all the multiplications for each digit of the bottom number, we add up the results to compute the final product.

How efficient is this algorithm? If the input numbers are each  $n$  bits long, then each individual multiplication takes linear  $O(n)$  time: we have to multiply each digit in the top number by the single digit in the bottom number (plus a carry if we are working in decimal). Since we have to do  $n$  multiplications, computing the partial results takes  $O(n^2)$  total time. We then need to add the  $n$  partial results. The longest partial result is the last one, which is about  $2n$  digits long. Thus, we add  $n$  numbers, each of which has  $O(n)$  digits. Adding two  $O(n)$ -digit numbers takes  $O(n)$  time, so adding  $n$  of them takes a total of  $O(n^2)$  time. Adding the time for the multiplications and additions leaves us with a total of  $O(n^2)$  time for the entire multiplication. (All of the above bounds are tight, so the running time is in fact  $\Theta(n^2)$ .)

Can we do better? Let’s try to make use of the divide-and-conquer paradigm. We first need a way of breaking up an  $n$ -digit number into smaller pieces. We can do that by splitting it into the first  $n/2$  digits and the last  $n/2$  digits. For the rest of our discussion, we will work with decimal numbers, though the same reasoning applies to numbers in any other base. Assume that  $n$  is even for simplicity (we can ensure this by appending a zero in the most-significant digit, if needed).



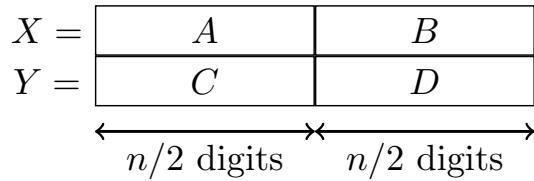
As an example, consider the number 376280. Here  $n = 6$ , and splitting the number into two pieces gives us 376 and 280. How are these pieces related to the original number? We have:

$$\begin{aligned} 376280 &= 376 \cdot 1000 + 280 \\ &= 376 \cdot 10^3 + 280 \\ &= 376 \cdot 10^{n/2} + 280. \end{aligned}$$

In general, when we split an  $n$ -digit number  $X$  into two  $n/2$ -digit pieces  $A$  and  $B$ , we have that  $X = A \cdot 10^{n/2} + B$ .

Let's now apply this splitting process to multiply two  $n$ -digit numbers  $X$  and  $Y$ . Split  $X$  into  $A$  and  $B$ , and  $Y$  into  $C$  and  $D$ , so that:

$$\begin{aligned} X &= A \cdot 10^{n/2} + B, \\ Y &= C \cdot 10^{n/2} + D. \end{aligned}$$



We can then expand  $X \cdot Y$  as:

$$\begin{aligned} X \cdot Y &= (A \cdot 10^{n/2} + B) \cdot (C \cdot 10^{n/2} + D) \\ &= A \cdot C \cdot 10^n + A \cdot D \cdot 10^{n/2} + B \cdot C \cdot 10^{n/2} + B \cdot D \\ &= A \cdot C \cdot 10^n + (A \cdot D + B \cdot C) \cdot 10^{n/2} + B \cdot D. \end{aligned}$$

This suggests a natural divide-and-conquer algorithm for multiplying  $X$  and  $Y$ :

- split them as above,
- *recursively* multiply  $A \cdot C$ ,  $A \cdot D$ , etc.,
- multiply each of these by the appropriate power of 10,
- sum everything up.

How efficient is this computation? First, observe that multiplying a number by  $10^k$  is the same as shifting it to the left by appending  $k$  zeros to the (least-significant) end of the number, so it can be done in  $O(k)$  time. So, the algorithm has the following subcomputations:

- 4 recursive multiplications of  $n/2$ -digit numbers ( $A \cdot C$ ,  $A \cdot D$ ,  $B \cdot C$ ,  $B \cdot D$ ),
- 2 left shifts, each of which takes  $O(n)$  time,
- 3 additions of  $O(n)$ -digit numbers, which take  $O(n)$  time.

Let  $T(n)$  be the time it takes to multiply two  $n$ -digit numbers using this algorithm. By the above analysis, it satisfies the recurrence

$$T(n) = 4T(n/2) + O(n).$$

Applying the Master Theorem with  $k = 4$ ,  $b = 2$ ,  $d = 1$ , we have that  $k > b^d$ . Therefore, the solution is

$$T(n) = O(n^{\log_2 4}) = O(n^2).$$

Unfortunately, this is the same as for the long-multiplication algorithm! We did a lot of work to come up with a divide-and-conquer algorithm, and it doesn't do any better than a naïve algorithm. Our method of splitting and recombining wasn't sufficiently "clever" to yield an improvement.

### 3.2.1 The Karatsuba Algorithm

Observe that the  $O(n^2)$  bound above has an exponent of  $\log_2 4 = 2$  because we recursed on *four* separate subinstances of size  $n/2$ . Let's try again, but this time, let's see if we can rearrange the computation so that we have *fewer than four* such subinstances. We previously wrote

$$X \cdot Y = A \cdot C \cdot 10^n + (A \cdot D + B \cdot C) \cdot 10^{n/2} + B \cdot D.$$

This time, we will write  $X \cdot Y$  in a different, more clever way using fewer multiplications of (roughly) "half-size" numbers. Consider the values

$$\begin{aligned} M_1 &= (A + B) \cdot (C + D) \\ M_2 &= A \cdot C \\ M_3 &= B \cdot D. \end{aligned}$$

Observe that  $M_1 = A \cdot C + A \cdot D + B \cdot C + B \cdot D$ , and we can subtract  $M_2 = A \cdot C$  and  $M_3 = B \cdot D$  to obtain

$$M_1 - M_2 - M_3 = A \cdot D + B \cdot C.$$

This is exactly the "middle" term in the above expansion of  $X \cdot Y$ . Thus:

$$X \cdot Y = M_2 \cdot 10^n + (M_1 - M_2 - M_3) \cdot 10^{n/2} + M_3.$$

This suggests a different divide-and-conquer algorithm for multiplying  $X$  and  $Y$ :

- split them as above,
- compute  $A + B, C + D$  and *recursively* multiply them to get  $M_1$ ,
- *recursively* compute  $M_2 = A \cdot C$  and  $M_3 = B \cdot D$ ,
- compute  $M_1 - M_2 - M_3$ ,
- multiply by appropriate powers of 10,
- sum up the terms.

This is known as the *Karatsuba algorithm*. How efficient is the computation? We have the following subcomputations:

- Computing  $M_1$  does two additions of  $n/2$ -digit numbers, resulting in two numbers that are up to  $n/2 + 1$  digits each. This takes  $O(n)$  time.
- Then these two numbers are multiplied, which takes essentially  $T(n/2)$  time. (The Master Theorem lets us ignore the one extra digit of input length, just like we can ignore floors and ceilings.)
- Computing  $M_2$  and  $M_3$  each take  $T(n/2)$  time.
- Computing  $M_1 - M_2 - M_3$ , multiplying by powers of 10, and adding up terms all take  $O(n)$  time.

So, the running time  $T(n)$  satisfies the recurrence

$$T(n) = 3T(n/2) + O(n).$$

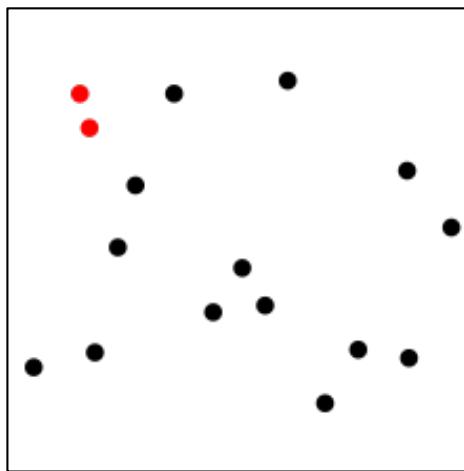
Applying the master theorem with  $k = 3$ ,  $b = 2$ ,  $d = 1$ , we have that  $k > b^d$ . This yields the solution

$$T(n) = O(n^{\log_2 3}) = O(n^{1.585}).$$

(Note that  $\log_2 3$  is slightly smaller than 1.585, so the second equality is valid because big-O represents an upper bound.) Thus, the Karatsuba algorithm gives us a runtime that is asymptotically much faster than the naïve algorithm! Indeed, it was the first algorithm discovered for integer multiplication that takes “subquadratic” time.

### 3.3 The Closest-Pair Problem

In the *closest-pair problem*, we are given  $n \geq 2$  points in  $d$ -dimensional space, and our task is to find a pair  $p, p'$  of the points whose distance apart  $\|p - p'\|$  is *smallest* among all pairs of the points; such points are called a “closest pair”. Notice that there may be ties among distances between points, so there may be more than one closest pair. Therefore, we typically say *a* closest pair, rather than *the* closest pair, unless we know that the closest pair is unique in some specific situation. This problem has several applications in computational geometry and data mining (e.g. clustering). The following is an example of this problem in two dimensions, where the (unique) closest pair is at the top left in red.



A naïve algorithm compares the distance between every pair of points and returns a pair that is the smallest distance apart; since there are  $\Theta(n^2)$  pairs, the algorithm takes  $\Theta(n^2)$  time. Can we do better?

Let's start with the problem in the simple setting of one dimension. That is, given a list of  $n$  real numbers  $x_1, x_2, \dots, x_n$ , we wish to find a pair of the numbers that are closest together. In other words, find some  $x_i, x_j$  that minimize  $|x_i - x_j|$ , where  $i \neq j$ .

Rather than comparing every pair of numbers, we can first sort the numbers. Then it must be the case that there is some closest pair of numbers that is adjacent in the sorted list. (Exercise: prove this formally. However, notice that not *every* closest pair must be adjacent in the sorted list, because there can be duplicate numbers.) So, we need only compare each pair of adjacent points to find some closest pair. The following is a complete algorithm:

#### Algorithm 17 (Closest Numbers)

```

Input: an array of  $n \geq 2$  numbers
Output: a closest pair of numbers in the array
function CLOSESTNUMBERS( $A[1, \dots, n]$ )
   $S = \text{MERGESORT}(A)$ 
```

```

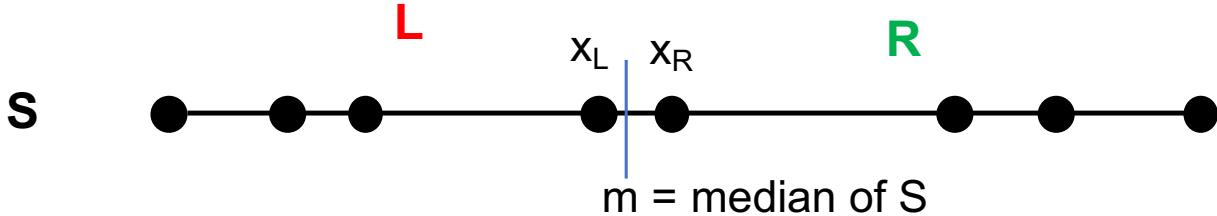
 $i = 1$ 
for  $k = 2$  to  $n - 1$  do
    if  $|S[k] - S[k + 1]| < |S[i] - S[i + 1]|$  then
         $i = k$ 
return  $S[i], S[i + 1]$ 

```

As we saw previously, merge sort takes  $\Theta(n \log n)$  time (assuming fixed-size numbers). The algorithm above also iterates over the sorted list, doing a constant amount of work in each iteration. This takes  $\Theta(n)$  time. Putting the two steps together results in a total running time of  $\Theta(n \log n)$ , which is better than the naïve  $\Theta(n^2)$ .

This algorithm works for one-dimensional points, i.e., real numbers. Unfortunately, it is not clear how to generalize this algorithm to two-dimensional points. While there are various ways we can sort such points, there is no obvious ordering that provides the guarantee that some closest pair of points is adjacent in the resulting ordering. For example, if we sort by  $x$ -coordinate, then a closest pair will be relatively close in their  $x$ -coordinates, but there may be another point with an  $x$ -coordinate between theirs that is very far away in its  $y$ -coordinate.

Let's take another look at the one-dimensional problem, instead taking a divide-and-conquer approach. Consider the median of all the points, and suppose we partition the points into two halves of (almost) equal size, according to which side of the median they lie on. For simplicity, here and below we assume without loss of generality that the median splits the points into halves that are as balanced as possible, by breaking ties between points as needed to ensure balance.



Now consider a closest pair of points in the full set. It must satisfy exactly one of the following three cases:

- both points are from the left half,
- both points are from the right half,
- it “crosses” the halves, with one point in the left half and the other point in the right half.

In the “crossing” case, we can draw a strong conclusion about the two points: they must consist of a *largest* point in the left half, and a *smallest* point in the right half. (For if not, there would be an even closer crossing pair, which would contradict the hypothesis about the original pair.) So, such a pair is the only crossing pair we need to consider when searching for a closest pair.

This reasoning leads naturally to a divide-and-conquer algorithm. We find the median and *recursively* find a closest pair within just the left-half points, and also within just the right-half points. We also consider a largest point on the left with a smallest point on the right. Finally, we return a closest pair among all three of these options. By the three cases above, the output must be a closest pair among all the points. Specifically, in the first case, the recursive call on the left half returns a closest pair for the full set, and similarly for the second case and the recursive call on the right half. And in the third case, by the above reasoning, the specific crossing pair constructed by the algorithm is a closest pair for the full set.

The full algorithm is as follows. For the convenience of the recursive calls, in the case  $n = 1$  we define the algorithm to return a “dummy” output  $(\perp, \perp, \infty)$  representing non-existent points that are infinitely far apart. Therefore, we do not have to check whether each recursive call involves more than one point, and some other pair under consideration will be closer than this dummy result.

**Algorithm 18 (1D Closest Pairs)**

**Input:** an array of  $n \geq 1$  real numbers  
**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

```

function CLOSESTPAIR1D( $A[1, \dots, n]$ )
    if  $n = 1$  then return ( $\perp, \perp, \infty$ )
    if  $n = 2$  then return ( $A[1], A[2], |A[1] - A[2]|$ )
    partition  $A$  by its median  $m$  into arrays  $L, R$ 
     $(\ell, \ell', \delta_L) = \text{CLOSESTPAIR1D}(L)$ 
     $(r, r', \delta_R) = \text{CLOSESTPAIR1D}(R)$ 
     $p =$  a largest element in  $L$ 
     $p' =$  a smallest element in  $R$ 
    return one of the triples  $(\ell, \ell', \delta_L), (r, r', \delta_R), (p, p', |p - p'|)$  that has smallest distance
  
```

Analyzing this algorithm for its running time, we can find the median of a set of points by sorting them and then taking the point in the middle. We can also obtain a largest element in the left side and a largest element in the right right from the sorted list. Partitioning the points by the median also takes  $\Theta(n)$  time; we just compare each point to the median. The non-recursive work is dominated by the  $\Theta(n \log n)$ -time sorting, so we end up with the recurrence:

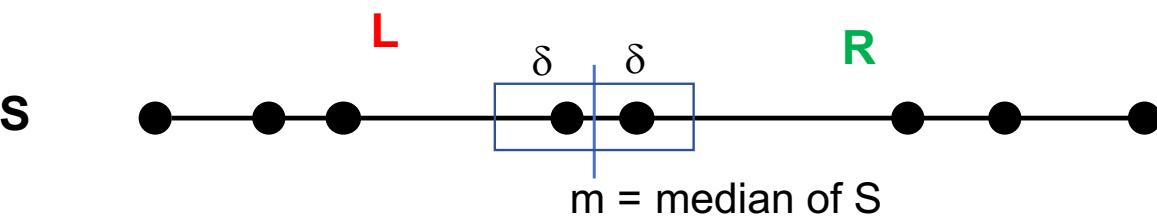
$$T(n) = 2T(n/2) + \Theta(n \log n).$$

This isn't quite covered by the basic form of the Master Theorem, since the additive term is not of the form  $\Theta(n^d)$  for a constant  $d$ . However, it is covered by the *Master Theorem with Log Factors* (page 15), which yields the solution  $T(n) = \Theta(n \log^2 n)$ . (See also a solution using substitution in Example 298.) This means that this algorithm is asymptotically less efficient than our previous one! However, there are two possible modifications we can make:

- Use a  $\Theta(n)$  median-finding algorithm<sup>8</sup> rather than sorting to find the median.
- Sort the points just once at the beginning, so that we don't need to re-sort in each recursive call.

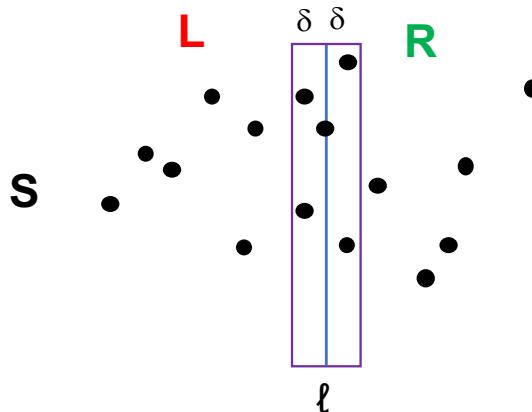
Either modification brings the running time of the non-recursive work down to  $\Theta(n)$ , resulting in a full running time of  $T(n) = \Theta(n \log n)$ . (The second option involves an additional  $\Theta(n \log n)$  on top of this for the presorting, but that still results in a total time of  $\Theta(n \log n)$ .)

**Exercise 19** In the one-dimensional closest-pair algorithm, we computed the median  $m$ , the closest-pair distance  $\delta_L$  on the left, and the closest-pair distance  $\delta_R$  on the right. Let  $\delta = \min\{\delta_L, \delta_R\}$ . How many points can lie in the interval  $[m, m + \delta]$ ? What about the interval  $(m - \delta, m + \delta)$ ?



Now let's try to generalize this algorithm to two dimensions. It's not clear how to split the points according to a median point, or even what a meaningful "median point" would be. So rather than doing that, we instead use a median *line* as defined by the median *x*-coordinate.

<sup>8</sup> [https://en.wikipedia.org/wiki/Median\\_ofmedians](https://en.wikipedia.org/wiki/Median_ofmedians)



As before, any closest pair for the full set must satisfy one of the following: both of its points are in the left half, both are in the right half, or it is a “crossing” pair with exactly one point in each half. Similarly to above, we will prove that in the “crossing” case, the pair must satisfy some specific conditions. This means that it will suffice for our algorithm to check *only* those crossing pairs that meet the conditions—since this will find a closest pair, it can ignore all the rest.

In two dimensions we cannot draw as strong of a conclusion about the “crossing” case as we could in one dimension. In particular, the  $x$ -coordinates of the pair may *not* be closest to the median line: there could be another crossing pair whose  $x$ -coordinates are even closer to the median line, but whose  $y$ -coordinates are very far apart, making that pair farther apart overall. Nevertheless, the  $x$ -coordinates are “relatively close” to the median line, as shown in the following lemma.

**Lemma 20** *Let  $\delta_L$  and  $\delta_R$  respectively be the closest-pair distances for just the left and right halves. If a closest pair for the entire set of points is “crossing,” then both of its points must be within distance  $\delta = \min\{\delta_L, \delta_R\}$  of the median line.*

**Proof 21** We prove the contrapositive. If a crossing pair of points has at least one point at distance greater than  $\delta$  from the median line, then the pair of points are more than  $\delta$  apart. Therefore, they cannot be a closest pair for the entire set, because there is another pair that is only  $\delta$  apart.  $\square$

Thus, the only crossing pairs that our algorithm needs to consider are those whose points lie in the “ $\delta$ -strip”, i.e., the space within distance  $\delta$  of the median line (in the  $x$ -coordinate); no other crossing pair can be a closest pair for the entire set. This leads to the following algorithm:

**Algorithm 22 (2D Closest Pair – First Attempt)**

**Input:** an array of  $n \geq 1$  points in the plane  
**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

```

function CLOSESTPAIR2DATTEMPT( $A[1, \dots, n]$ )
  if  $n = 1$  then return ( $\perp, \perp, \infty$ )
  if  $n = 2$  then return ( $A[1], A[2], \|A[1] - A[2]\|$ )
  partition  $A$  by its median  $x$ -coordinate  $m$  into arrays  $L, R$ 
   $(\ell, \ell', \delta_\ell) = \text{CLOSESTPAIR2DATTEMPT}(L)$ 
   $(r, r', \delta_r) = \text{CLOSESTPAIR2DATTEMPT}(R)$ 
   $\delta = \min\{\delta_\ell, \delta_r\}$ 
  find a closest pair  $(p, p') \in L \times R$  among the points whose  $x$ -coordinates are within  $\delta$  of  $m$ 
  return one of the triples  $(\ell, \ell', \delta_\ell), (r, r', \delta_r), (p, p', \|p - p'\|)$  that has smallest distance

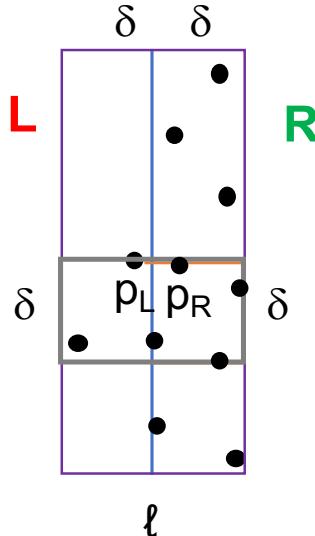
```

Apart from its checks of crossing pairs in the  $\delta$ -strip, the non-recursive work is same as in the one-dimensional algorithm, and it takes  $\Theta(n)$  time. (We can presort the points by  $x$ -coordinate or use a  $\Theta(n)$ -time median-finding algorithm, as before.) How long does it take to check the crossing pairs in the  $\delta$ -strip? A naïve examination would consider every pair of points where one is in the left side of the  $\delta$ -strip and the other is in its right side. But notice that in the worst case, *all* of the points can be in the  $\delta$ -strip! For example, this can happen if the points are close together in the  $x$ -dimension—in the extreme, they all lie on the median line—but far apart in the  $y$ -dimension. So in the worst case, we have  $n/2$  points in each of the left and right parts of the  $\delta$ -strip, leaving us with  $n^2/4 = \Theta(n^2)$  crossing pairs to consider. So we end up with the recurrence and solution

$$T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2).$$

This is no better than the naïve algorithm that just compares all pairs of points! We have not found an efficient enough non-recursive “combine” step.

Let’s again consider the case where a closest pair for the whole set is a “crossing” pair, and try to establish some additional stronger properties for it, so that our algorithm will not need to examine as many crossing pairs in its combine step. Let  $p_L = (x_L, y_L)$  and  $p_R = (x_R, y_R)$  respectively be the points from the pair that are on the left and right of the median line, and assume without loss of generality that  $y_L \geq y_R$ ; otherwise, replace  $p_L$  with  $p_R$  in the following analysis. Because this is a closest pair for the entire set of points,  $p_L$  and  $p_R$  are at most  $\delta = \min\{\delta_L, \delta_R\}$  apart, where as in Lemma 20 above,  $\delta_L$  and  $\delta_R$  are respectively the closest-pair distances for just the left and right sides. Therefore,  $0 \leq y_L - y_R \leq \delta$ .



We ask: how many of the given points  $p' = (x', y')$  in the  $\delta$ -strip can satisfy  $0 \leq y_L - y' \leq \delta$ ? Equivalently, any such point is in the  $\delta$ -by- $2\delta$  rectangle of the  $\delta$ -strip whose top edge has  $p_L$  on it. We claim that there can be at most *eight* such points, including  $p_L$  itself. (The exact value of eight is not too important; what matters is that it is a *constant*.) The key to the proof is that every pair of points must be at least  $\delta$  apart, so we cannot fit too many points into the rectangle. We leave the formal proof to [Exercise 25](#) below.

In conclusion, we have proved the following key structural lemma.

**Lemma 23** *If a closest pair for the whole set is a crossing pair, then its two points are in the  $\delta$ -strip, and they are within 7 positions of each other when all the points in the  $\delta$ -strip are sorted by  $y$ -coordinate.*

So, if a closest pair in the whole set is a crossing pair, then it suffices for the algorithm to compare each point in the  $\delta$ -strip with the (up to) seven points in the  $\delta$ -strip that precede it in sorted order by  $y$ -coordinate. By [Lemma 23](#), this will find a closest pair for the entire set, so the algorithm does not need to check any other pairs. The formal algorithm is as follows.

#### Algorithm 24 (2D Closest Pairs)

**Input:** an array of  $n \geq 1$  points in the plane  
**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

```

function CLOSESTPAIR2D( $A[1, \dots, n]$ )
    if  $n = 1$  then return ( $\perp, \perp, \infty$ )
    if  $n = 2$  then return ( $A[1], A[2], \|A[1] - A[2]\|$ )
    partition  $A$  by its median  $x$ -coordinate  $m$  into arrays  $L, R$ 
     $(\ell, \ell', \delta_\ell) = \text{CLOSESTPAIR2D}(L)$ 
     $(r, r', \delta_r) = \text{CLOSESTPAIR2D}(R)$ 
     $\delta = \min\{\delta_\ell, \delta_r\}$ 
     $D =$  the set of points whose  $x$ -coordinates are within  $\delta$  of  $m$ , sorted by  $y$ -coordinate
    for all  $p$  in  $D$  do
        consider the triple  $(p, p', \|p - p'\|)$  for the (up to) 7 points  $p'$  preceding  $p$  in  $D$ 
    return one of the triples among  $(\ell, \ell', \delta_\ell), (r, r', \delta_r)$ , and those above that has smallest distance
```

We have already seen that we can presort by  $x$ -coordinate, so that finding the median and constructing  $L, R$  in each run of the algorithm takes just  $O(n)$  time. We can also separately presort by  $y$ -coordinate (into a different array) so that we do not have to sort the  $\delta$ -strip in each run. Instead, we merely filter the points from the presorted array according to whether they lie within the  $\delta$ -strip, which also takes  $O(n)$  time. Finally, we consider at most  $7n = \Theta(n)$  pairs in the  $\delta$ -strip. So, the non-recursive work takes  $\Theta(n)$  time, resulting in the runtime recurrence and solution

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n).$$

This matches the asymptotic efficiency of the one-dimensional algorithm. The algorithm can be further generalized to higher dimensions, retaining the  $\Theta(n \log n)$  runtime for any fixed dimension.

**Exercise 25** Prove that any  $\delta$ -by- $2\delta$  rectangle of the  $\delta$ -strip can have at most 8 of the given points, where  $\delta$  is as defined in [Lemma 20](#).

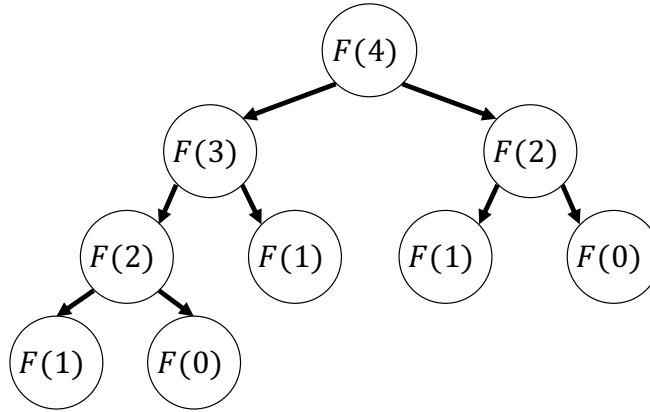
Specifically, prove that the left  $\delta$ -by- $\delta$  square of the rectangle can have at most four of the points (all from left subset), and similarly for the right square.

**Hint:** Partition each square into four congruent sub-squares, and show that each sub-square can have at most one point (from the relevant subset).

## DYNAMIC PROGRAMMING

The idea of subdividing a large problem instance into smaller instances of the same problem lies at the core of the divide-and-conquer paradigm. However, for some problems, this recursive subdivision may result in many recurrences of the exact same subinstance. Such situations are amenable to the paradigm of *dynamic programming*, which is applicable to problems that have the following features:

1. The *principle of optimality*, also known as an *optimal substructure*. This means that an optimal solution to a larger instance is made up of optimal solutions to smaller subinstances. For example, shortest-path problems on graphs generally obey the principle of optimality. If a shortest path between vertices  $a$  and  $b$  in a graph goes through some other vertex  $c$ , so that the path has the form  $a, u_1, \dots, u_j, c, v_1, \dots, v_k, b$ , then the subpath  $a, u_1, \dots, u_j, c$  must be a shortest path from  $a$  to  $c$ , and similarly for the subpath from  $c$  to  $b$ .
2. *Overlapping subinputs/subproblems*. This means that the same input occurs many times when recursively decomposing the original instance down to the base cases. A classic example of this is a recursive computation of the Fibonacci sequence, which follows the recurrence  $F(n) = F(n - 1) + F(n - 2)$ . The same subinstances appear over and over again, making a naïve computation takes time exponential in  $n$ . The characteristic of overlapping subinputs is what distinguishes dynamic programming from divide and conquer.



### 4.1 Implementation Strategies

The first, and typically most challenging and creative, step in formulating a dynamic-programming solution to a problem is to determine a recurrence relation that solutions adhere to. In the case of the Fibonacci sequence, for example, such a recurrence (and base cases) are already given, as:

$$F(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ F(n - 1) + F(n - 2) & \text{otherwise.} \end{cases}$$

Once we have established a recurrence relation and base cases, we can turn to an implementation strategy for *computing* the desired value(s) of the recurrence. There are three typical patterns:

1. *Top-down recursive.* The naïve implementation directly translates the recurrence relation into a recursive algorithm, as in the following:

**Algorithm 26 (Top-down Fibonacci)**

```

Input: an integer  $n \geq 0$ 
Output: the  $n$ th Fibonacci number
function FIBRECURSIVE( $n$ )
    if  $n \leq 1$  then return 1
    return FIBRECURSIVE( $n - 1$ ) + FIBRECURSIVE( $n - 2$ )

```

As mentioned previously, the problem with this strategy is that it repeats the same computations many times, to the extent that the overall number of recursive calls is exponential in  $n$ . More generally, naïve top-down implementations are wasteful when there are overlapping subinputs. (They do not use any auxiliary storage, so they are space efficient, but this is usually outweighed by their poor running times.<sup>9</sup>)

2. *Top-down memoized*, or simply *memoization*. This approach also translates the recurrence relation into a recursive algorithm, but it *saves every computed result in a lookup table*, and queries that table before doing any new computation. The following is an example:

**Algorithm 27 (Memoized Fibonacci)**

```

Input: an integer  $n \geq 0$ 
Output: the  $n$ th Fibonacci number
memo = an empty table (e.g., an array or dictionary)
function FIBMEMOIZED( $n$ )
    if  $n \leq 1$  then return 1
    if memo( $n$ ) is not defined then
        memo( $n$ ) = FIBMEMOIZED( $n - 1$ ) + FIBMEMOIZED( $n - 2$ )
    return memo( $n$ )

```

This memoized algorithm avoids recomputing the answer for any previously encountered input. Any call to FIBMEMOIZED( $n$ ), where  $n$  is not a base case, first checks the memo table to see if FIBMEMOIZED( $n$ ) has been computed before. If not, it computes it recursively as above, saving the result in memo. If the subinput was previously encountered, the algorithm just returns the previously computed result.

Memoization trades space for time. The computation of FIBMEMOIZED( $n$ ) requires  $O(n)$  auxiliary space to store the results for each subinput. On the other hand, since the answer to each subinput is computed only once, the overall number of operations required is  $O(n)$ , a significant improvement over the exponential naïve algorithm. However, for more complicated algorithms, it can be harder to analyze the running time of the associated algorithm.

3. *Bottom up.*<sup>10</sup> Rather than starting with the desired input and recursively working our way down to the base cases, we can invert the computation to start with the base case(s), and then work our way up to the desired input. As in recursion with memoization, we need a table to store the results for the subinputs we have handled so far, since those results will be needed to compute answers for larger inputs.

The following is a bottom-up implementation for computing the Fibonacci sequence:

<sup>9</sup> Space is required to represent the recursion stack; for the Fibonacci computation, the recursion depth can be as large as nearly  $n$ . For other algorithms, however, the naïve implementation can incur smaller space costs than the table-based approaches.

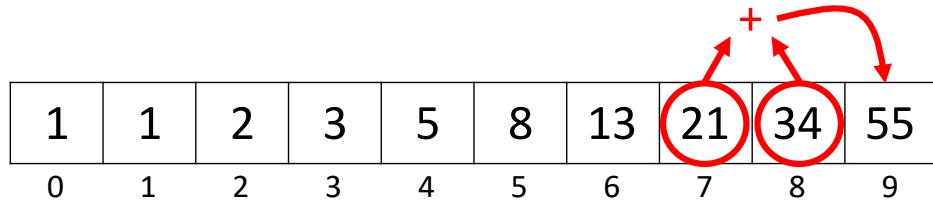
<sup>10</sup> In many contexts, the term “dynamic programming” is restricted specifically to the bottom-up implementation strategy, but the term can also encompass recursion with memoization. In this text, we mainly adopt the bottom-up strategy.

**Algorithm 28 (Bottom-up Fibonacci)**

**Input:** an integer  $n \geq 0$   
**Output:** the  $n$ th Fibonacci number

```
function FIBBOTTOMUP( $n$ )
    allocate table[0, ...,  $n$ ]
    table[0] = table[1] = 1
    for  $i = 2$  to  $n$  do
        table[ $i$ ] = table[ $i - 1$ ] + table[ $i - 2$ ]
    return table[ $n$ ]
```

We start with an empty table and populate it with the results for the base cases. Then we work our way forward, computing the result for each larger input from the previously computed results for the smaller inputs. We stop when we reach the desired input, and return the result. The following is an illustration of the table that is constructed during the computation of  $\text{FIBBOTTOMUP}(9)$ .



In the loop for a particular value of  $i$ , earlier iterations have already computed and stored the  $(i - 1)$ st and  $(i - 2)$ nd Fibonacci numbers—stored in  $\text{table}[i - 1]$  and  $\text{table}[i - 2]$ , respectively—so the algorithm just looks up those results from the table and adds them to get the  $i$ th Fibonacci number, which it stores in  $\text{table}[i]$ .

Like memoization, the bottom-up approach trades space for time. In the case of  $\text{FIBBOTTOMUP}(n)$ , it too uses a total of  $n$  array entries to store the results of the subinstances, and the overall number of additions required is also less than  $n$ .

In the specific case of computing the Fibonacci sequence, we don't actually need to keep the entire table for the entire computation: once we are computing the  $i$ th Fibonacci number, we no longer need the  $(i - 3)$ rd table entry or lower. So, at any moment we need only keep the two previously computed results. This lowers the storage overhead to just  $O(1)$  entries. However, this kind of space savings doesn't work in general for dynamic programming; other problems require maintaining most or all of the table throughout the computation.

The three implementation strategies have different tradeoffs. The naïve top-down strategy often takes the least implementation effort, as it is a direct translation of the recurrence relation to code. It can be the most space efficient (though including the space used by the recursive call stack complicates the comparison), but more importantly, it often is *very inefficient* in time, due to the many redundant computations.

Top-down recursion with memoization adds some implementation effort in working with a lookup table, and can require special care to implement correctly and safely in practice.<sup>11</sup>

Its main advantages over the bottom-up approach are:

- It maintains the same structure as the recurrence relation, so it typically is simpler to reason about and implement.
- It computes answers for only the subinputs that are actually needed. If the recurrence induces a “sparse” computation, meaning that it requires answers for relatively few of the smaller subinputs, then the top-down memoization

<sup>11</sup> In particular, it might require a global variable for the “memo table,” which is not considered a good programming technique due to its risks. For example, completely unrelated code might have access to the same global variable, resulting in incorrect results. Fortunately, some programming languages provide built-in facilities for converting a naïve recursive function into a memoized one, such as `@functools.lru_cache` in Python.

approach can be more time and space efficient than the bottom-up strategy.

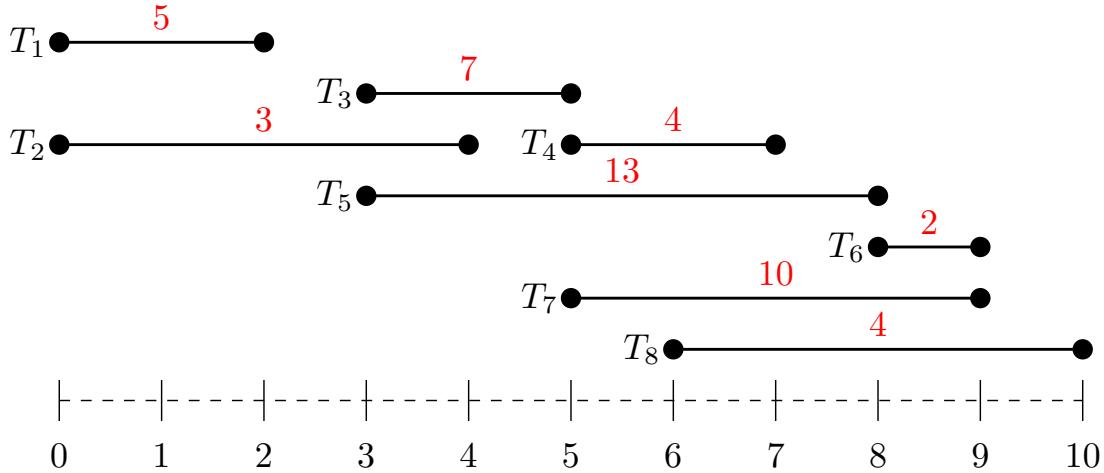
However, top-down memoization often suffers from higher constant-factor overheads than the bottom-up approach (e.g., function-call overheads and working with sparse lookup structures that are less time efficient than dense data structures). Thus, the bottom-up approach is preferable when answers to a large fraction of the subinstances are needed for computing the desired result, which is usually the case for dynamic programming problems.

## 4.2 Weighted Task Selection

As a first nontrivial example of dynamic programming, let us consider a problem called *weighted task selection* (WTS). In this problem, we are given a list of  $n$  tasks  $T_1, \dots, T_n$ . Each task  $T_i$  is a triple of numbers  $(s_i, f_i, v_i)$  with  $s_i < f_i$ , where  $s_i$  denotes the task's starting time,  $f_i$  denotes its finishing time, and  $v_i$  denotes its value. A pair of tasks  $T_i, T_j$  overlap if  $s_i \leq s_j < f_i$  or  $s_j \leq s_i < f_j$ , i.e., if the intersection of their time intervals  $[s_i, f_i] \cap [s_j, f_j]$  is nonempty. The goal in WTS is to select an *optimal* set of (pairwise) non-overlapping tasks, which is one that *maximizes the total value* of the selected tasks.

Note that there may be multiple different sets that have the same optimal value, so we say “*an*” optimal set, rather than “*the*” optimal set. Throughout this treatment it is convenient to assume without loss of generality that the tasks are sorted by their finish times  $f_i$  (which are not necessarily distinct); in particular, we can sort them in  $O(n \log n)$  time.

The figure below shows an example instance of the WTS problem with eight tasks. The set  $\{T_5, T_7\}$  has a large total value of  $13 + 10 = 23$ , but it is overlapping, so it cannot be selected. The set  $\{T_1, T_5, T_6\}$  is not overlapping, and has a total value of  $5 + 13 + 2 = 20$ , but it turns out that this is not optimal. An optimal set of tasks is  $\{T_1, T_3, T_7\}$ , and has a total value of  $5 + 7 + 10 = 22$ . In fact, in this case it is the *unique* optimal set (but again, in general an optimal set will not be unique).



This problem models various situations including scheduling of jobs on a single machine, choosing courses to take, etc. So it is indeed useful, but can we solve it efficiently?

The design and analysis of a dynamic programming algorithm usually follows the following template, or “recipe”, of steps:

1. Define and focus on the “**value version**” of the problem. For an optimization problem like WTS, temporarily put aside the goal of finding an optimal solution itself, and simply aim to find the *value* of an optimal solution (e.g., the lowest cost, the largest total value, etc.). In some cases, the original problem is already stated in a value version—e.g., count the *number* of objects meeting certain constraints—so there is nothing to do in this step.
2. Devise a **recurrence for the value in question**, including base case(s), by understanding how a solution is made up of solutions to appropriate subinputs. This step usually requires some creativity and insight, both to discover a good set of relevant subinputs, and to see how optimal solutions are related across subinputs.

3. By understanding the dependencies between subinputs as given by the recurrence, **implement the recurrence in (pseudo)code** that fills a table in a bottom-up fashion. Given the recurrence, this step is usually fairly “mechanical” and does not require a great deal of creativity.
4. If applicable, extend the pseudocode to **solve the original problem using “backtracking”**. Given the recurrence and an understanding of why it is correct, this is also usually quite mechanical.
5. Perform a **runtime analysis** of the algorithm. This is also usually fairly mechanical, and follows from analyzing the number of table entries that are filled, and the amount of time it takes to fill each entry (or in some cases, all the entries in total).

Notice that all steps but the second one are fairly mechanical, but they rely crucially on the recurrence derived in that step.

For example, for the above example instance of our task-selection problem, we first want to focus on designing an algorithm that simply computes the optimal *value* 22, instead of directly computing an optimal *set* of non-overlapping tasks  $\{T_1, T_3, T_7\}$ . As we will see, a small enhancement of the value-optimizing algorithm will also give us a selection of tasks that achieves the optimal value, so we actually lose nothing by initially focusing on the value alone. To do this, we first need to come up with a recurrence that yields the optimal value for a given set of tasks.

For  $0 \leq i \leq n$ , let  $\text{OPT}(i)$  denote the optimal value that can be obtained by selecting from among just the tasks  $T_1, \dots, T_i$ . The base case is  $i = 0$ , for which the optimal value is trivially  $\text{OPT}(0) = 0$ , because there are no tasks available to select. Now suppose that  $i \geq 1$ , and consider task  $T_i$  (which has the latest finish time among those we are considering). There are two possibilities: either  $T_i$  is in some optimal selection of tasks (from just  $T_1, \dots, T_i$ ), or it is not.

- If  $T_i$  is not in an optimal selection, then  $\text{OPT}(i) = \text{OPT}(i - 1)$ . This is because there is an optimal selection for all  $i$  tasks that selects from just the first  $i - 1$  tasks, so the availability of  $T_i$  does not affect the optimal value.
- If  $T_i$  is in an optimal solution, then  $\text{OPT}(i) = v_i + \text{OPT}(j)$ , where  $j < i$  is the maximum index such that  $T_j$  and  $T_i$  do not overlap, i.e.,  $f_j \leq s_i$  (taking  $j = 0$  if no such  $T_j$  exists). This is because in any optimal selection  $S$  that has  $T_i$ , the other selected tasks must come from  $T_1, \dots, T_j$  (because these are the tasks that don't overlap with  $T_i$ ), and those selected tasks must have optimal value (among the first  $j$  tasks). For if they did not, we could replace them with a higher-value selection from  $T_1, \dots, T_j$ , then include  $T_i$  to get a valid selection of tasks with a higher value than that of  $S$ , contradicting the assumed optimality of  $S$ .

Overall, the optimal value is the maximum of what can be obtained from these two possibilities. Therefore, we obtain the final recurrence as follows, for all  $i \geq 1$ :

$$\text{OPT}(i) = \max\{\text{OPT}(i - 1), v_i + \text{OPT}(j)\} \text{ for the maximum } j < i \text{ s.t. } f_j \leq s_i .$$

To ensure that such a  $j$  is always defined, we adopt the convention that  $f_0 = -\infty$ , so that  $j = 0$  is an option (in which case the second value in the above max expression is just  $v_i + \text{OPT}(0) = v_i$ ).

Now that we have a recurrence, we can write pseudocode that implements it by filling a table in a bottom-up manner.

#### Algorithm 29 (Weighted Task Selection, Value Version)

```

Input: array of tasks  $T[i] = (s_i, f_i, v_i)$ 
Output: maximum achievable value for (pairwise) non-overlapping tasks
function OPTIMALTASKSVALUE( $T[1, \dots, n]$ )
    allocate table[0, \dots, n]
    table[0] = 0
    for  $i = 1$  to  $n$  do
        find the maximum  $j < i$  such that  $f_j \leq s_i$ 
        table[i] = max{table[i - 1],  $v_i + \text{table}[j]$ }
    
```

▷ convention:  $f_0 = -\infty$

```
return table[n]
```

A naïve implementation of this algorithm, which just does a linear scan inside the loop to determine  $j$ , takes  $O(n^2)$  time because there are two nested loops that each take at most  $n$  iterations. Since the tasks are sorted by finish time, a more sophisticated implementation would use a binary search, which would take just  $O(\log n)$  time for the inner (search) loop, and hence  $O(n \log n)$  time overall. (Recall that the initial time to sort the tasks by finish time is also  $O(n \log n)$ .)

For a better understanding of this algorithm, let us see the filled table for the example instance given above. We see that indeed  $\text{OPT}(n) = \text{OPT}(8) = 22$ , which is the correct answer. (We can also observe that entry 7 of the table is also 22, indicating that there is a globally optimal selection from among just the first 7 tasks.)

Table 4.1: Example Table of Weighted Task Selection DP Algorithm

Index $i$	0	1	2	3	4	5	6	7	8
$\text{OPT}(i)$	0	5	5	12	16	18	20	22	22

Now that we have an efficient algorithm for the *value* version of the problem, let us see how to solve the problem we actually care about, which is to obtain an optimal *selection of tasks*. The idea is to keep some additional information showing “why” each entry of the table has the value it does, and to construct an optimal selection by “backtracking” through the table. That is, we enhance the algorithm to add “pointers” (sometimes called “breadcrumbs”) that store how each cell in the table was filled, based on the recurrence and the values of the previous cells.

To carry out this approach for our problem, we will add pointers, denoted by  $\text{backtrack}[i]$ , into our algorithm. When we fill in  $\text{table}[i]$ , we also set  $\text{backtrack}[i] = i - 1$  if we set  $\text{table}[i] = \text{table}[i - 1]$ , otherwise we set  $\text{backtrack}[i] = j$  where  $j$  is defined as in the recurrence and algorithm. Recalling the reasoning for why the recurrence is valid, these two possibilities respectively correspond to task  $T_i$  not being in, or being in, an optimal subset of tasks from  $T_1, \dots, T_i$ . The value of  $\text{backtrack}[i]$  indicates the prefix of tasks from which the rest of that optimal subset is drawn.

Given the two arrays ( $\text{table}$ ,  $\text{backtrack}$ ), we can now construct an optimal set of tasks, not just the optimal value. Start with index  $i = n$ . Recall that  $\text{table}[i] > \text{table}[i - 1]$  if and only if  $T_i$  is in some optimal selection of tasks. So, if  $\text{table}[i] > \text{table}[i - 1]$ , then we include  $T_i$  in the output selection, otherwise we skip it. We then “backtrack” by setting  $i = \text{backtrack}[i]$  to consider the appropriate prefix of tasks, and repeat this process until we have backtracked to the beginning.

The modified full pseudocode, including the backtracking, can be seen below.

#### Algorithm 30 (Weighted Task Selection, with Backtracking)

**Input:** array of tasks  $T[i] = (s_i, f_i, v_i)$ , sorted by  $f_i$   
**Output:** values and backtracking information

```

function OPTIMALTASKSINFO( $T[1, \dots, n]$ )
    allocate  $\text{table}[0, \dots, n]$ ,  $\text{backtrack}[1, \dots, n]$ 
     $\text{table}[0] = 0$ 
    for  $i = 1$  to  $n$  do
        find the maximum  $j < i$  such that  $f_j \leq s_i$  (where  $f_0 = -\infty$ )
         $\text{table}[i] = \max\{\text{table}[i - 1], v_i + \text{table}[j]\}$ 
        if  $\text{table}[i] = \text{table}[i - 1]$  then
             $\text{backtrack}[i] = i - 1$ 
        else
             $\text{backtrack}[i] = j$ 
    return ( $\text{table}$ ,  $\text{backtrack}$ )
```

**Input:** array of tasks  $T[i] = (s_i, f_i, v_i)$ , sorted by  $f_i$   
**Output:** a set of non-overlapping tasks having maximum total value

```

function OPTIMALTASKS( $T[1, \dots, n]$ )
  ( $\text{table}$ ,  $\text{backtrack}$ ) = OPTIMALTASKSINFO( $T$ )
   $i = n$ 
   $S = \emptyset$ 
  while  $i > 0$  do
    if  $\text{table}[i] > \text{table}[i - 1]$  then
       $S = S \cup \{T_i\}$ 
     $i = \text{backtrack}[i]$ 
  return  $S$ 

```

## 4.3 Longest Increasing Subsequence

Given a sequence  $S$  of numbers, an *increasing subsequence* of  $S$  is a subsequence whose elements are in strictly increasing order. As with a subsequence of a string, the elements need not be contiguous in the original sequence, but they must be taken in their original order.

Now suppose we want to find a *longest increasing subsequence (LIS)* of a given input sequence  $S$ , i.e., an increasing subsequence with the maximum number of values in it. As in the task-selection problem, we say “*an*” LIS, rather than “*the*” LIS, because an LIS may not be unique. For example, the sequence  $S = (0, 8, 7, 12, 5, 10, 4, 14, 3, 6)$  has several longest increasing subsequences:

- $(0, 8, 12, 14)$ ,
- $(0, 8, 10, 14)$ ,
- $(0, 7, 12, 14)$ ,
- $(0, 7, 10, 14)$ ,
- $(0, 5, 10, 14)$ .

As in the previous problem of task selection, before concerning ourselves with finding an LIS itself, we first devise a suitable “value version” of the problem and a recurrence for it. As a first attempt, let the input sequence be  $S[1, \dots, N]$ , and consider the subproblems of computing the LIS length for each prefix sequence  $S[1, \dots, i]$ , for  $i = 1, \dots, N$ . For the example of  $S = (0, 8, 7, 12, 5, 10, 4, 14, 3, 6)$ , we can determine these LIS lengths by hand.

1	2	2	3	3	3	3	4	4	4
1	2	3	4	5	6	7	8	9	10

Unfortunately, it is not clear how to relate the LIS length for a sequence to the LIS lengths for its prefixes. In the example above,  $S[1, \dots, 9]$  and  $S[1, \dots, 10]$  have the same LIS length, but that is not the case for  $S[1, \dots, 7]$  and  $S[1, \dots, 8]$ . Yet in both cases, the one additional element is larger than the previous one in the sequence (i.e.,  $S[10] > S[9]$  and  $S[8] > S[7]$ ). It seems that, without knowing something about the *contents* of an LIS itself (and not just the LIS length), it is unclear how the LIS length for a sequence is related to those for its prefixes.

In order to devise a dynamic-programming solution for the LIS problem, we need to formulate a “value version” of the problem that satisfies the optimal-substructure property. A clever idea that turns out to work is to restrict our view to subsequences of  $S[1, \dots, i]$  that include the last element  $S[i]$ . More specifically, for any  $i \geq 1$ , define END-LIS( $i$ ) to be the length of any longest increasing subsequence of  $S[1, \dots, i]$  that includes (and therefore ends with)  $S[i]$ . As we will see next, this restriction is just enough information about the contents of an LIS to satisfy the optimal-substructure property, and thereby derive a useful recurrence.

For both the base case and recursive cases, it is convenient to define a “sentinel” value of  $S[0] = -\infty$ . This element can be seen as the initial “placeholder” element in *any* LIS of *any* prefix  $S[1, \dots, i]$ , but it *does not contribute to the*

*length*. With this convention, in the base case  $i = 0$  we trivially have  $\text{END-LIS}(0) = 0$ , because the only possible subsequence consists merely of  $S[0]$ , which has length zero.

We next derive a recurrence for  $\text{END-LIS}(i)$  for any  $i \geq 1$ , by establishing an “optimal substructure” property for longest increasing subsequences.

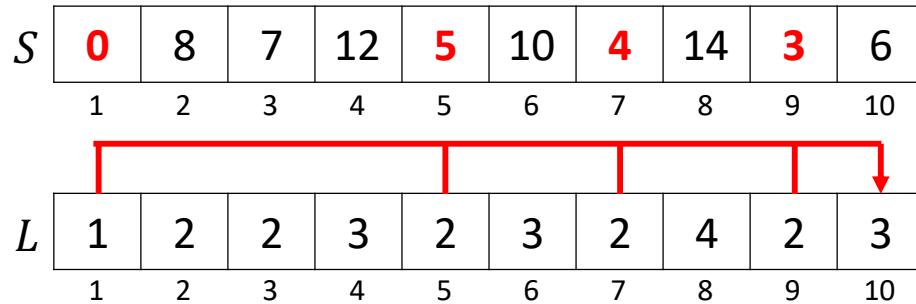
Let  $L$  be any LIS of  $S[1, \dots, i]$  that ends with  $S[i]$ , and let  $L'$  be  $L$  with this last element removed. Then the last element of  $L'$  (which might be the sentinel value  $S[0]$ ) must be  $S[j]$  for some  $0 \leq j < i$  where  $S[j] < S[i]$ , because  $L$  is an increasing subsequence of  $S$ .

We claim that  $L'$  must be an LIS of  $S[1, \dots, j]$  that ends with  $S[j]$ . For if it is not, then there would exist some increasing subsequence  $L^*$  of  $S[1, \dots, j]$  that ends with  $S[j] < S[i]$ , and is *longer* than  $L'$ . Then,  $L^*$  followed by  $S[i]$  would be an increasing subsequence of  $S[1, \dots, i]$  that ends with  $S[i]$ , and is *longer than*  $L$  (because  $L^*$  is longer than  $L'$ ). But this would contradict our initial hypothesis, that  $L$  is a *longest* such subsequence! So, such a  $L^*$  cannot exist, and the claim is proved.

In what we have just shown, there is no restriction on the value of  $0 \leq j < i$ , other than the requirement that  $S[j] < S[i]$ . Indeed, for any such  $j$ , any LIS of  $S[1, \dots, j]$  that ends with  $S[j]$  can be extended, by appending  $S[i]$ , into an LIS of  $S[1, \dots, i]$  that ends with  $S[i]$ . Therefore,  $\text{END-LIS}(i)$  is one larger than the largest of all these options, taken over all valid  $j$ . In summary, we have proved the following base case and recurrence for  $\text{END-LIS}$ :

$$\text{END-LIS}(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 + \max\{\text{END-LIS}(j) : 0 \leq j < i \text{ and } S[j] < S[i]\} & \text{if } i \geq 1. \end{cases}$$

The following gives the values of this recurrence for the example sequence  $S$  above, and also shows which values are referenced when evaluating  $\text{END-LIS}(10)$  according to the recurrence.



Using the above recurrence, we can straightforwardly write a bottom-up algorithm that computes  $\text{END-LIS}(i)$  for each  $i = 0, 1, \dots, N$ . Once we have these values, the actual (unconstrained) LIS length for  $S$  is simply the maximum value of  $\text{END-LIS}(i)$ , taken over all  $i$ . (This is because an LIS of  $S$  must end with *some*  $S[i]$  value, so its length is given by  $\text{END-LIS}(i)$ .) As with weighted task selection, we can also store “backpointers” while filling in the  $\text{END-LIS}$  table, and backtrack through the table to find the actual elements of a longest increasing subsequence.

How efficient is this algorithm? We must compute the  $N$  (non-base-case) values of  $\text{END-LIS}(i)$  (for  $i = 1, \dots, N$ ), and for each value we scan over all the elements of  $S[0, \dots, i - 1]$  (to compare them with  $S[i]$ ), as well as all the previous values of  $\text{END-LIS}$  in the worst case. Thus, it takes  $O(N)$  time to compute  $\text{END-LIS}(i)$  for a single  $i$ , and hence  $O(N^2)$  time to compute them all. Then, finding the maximum  $\text{END-LIS}(i)$  value takes  $O(n)$  time, as does the backtracking. So the algorithm as a whole takes  $O(N^2)$  time, and it uses  $O(N)$  space.

## 4.4 Longest Common Subsequence

As a richer example of dynamic programming, consider the problem of finding a *longest common subsequence* of two given strings. A *subsequence* of a string is a selection of zero or more of its characters, preserving their original order. (Alternatively, a subsequence is obtained by deleting zero or more of the string's characters.) The characters of the subsequence need not be adjacent in the original string.<sup>12</sup> For example, the following are subsequences of the string Fibonacci sequence:

- Fun
- seen
- cse

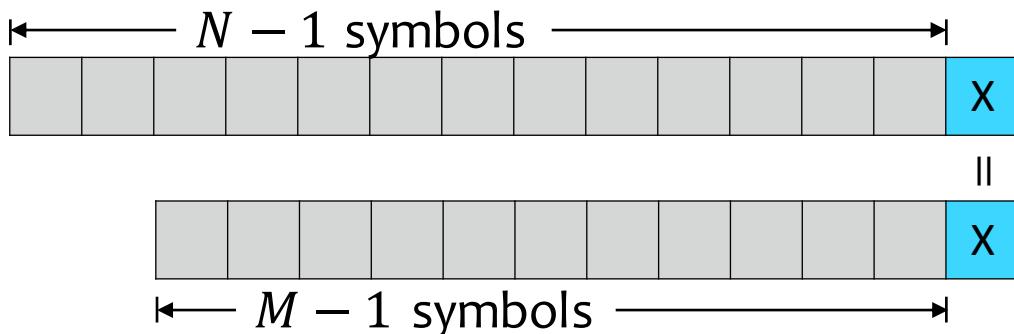
A *common subsequence* (CS) of two strings is a string that can be obtained as a subsequence of both strings, and a *longest common subsequence* (LCS) is one of maximum length. For instance, for the two strings Go blue! and Wolverines, some common subsequences are l, le, and ole. There is no common subsequence of length 4 or more, so ole is an LCS. As in the other problems considered above, in general an LCS is not unique (there can be multiple common subsequences of maximum length), though in this specific example it is unique.

Finding a longest common subsequence is useful in many applications, including DNA sequencing and computing the similarity between two texts. So, we wish to devise an efficient algorithm for determining an LCS of two input strings.

As we did above, let's temporarily set aside the problem of finding an LCS string itself, and first focus on just computing its *length*. For any two strings  $S_1, S_2$ , define  $\text{LCS}(S_1, S_2)$  to be the length of any LCS of the strings. To get a dynamic-programming algorithm, we first need to discover a recurrence relation and base case(s) that relate the LCS length for  $S_1$  and  $S_2$  to the LCS lengths for appropriate smaller subinputs.

Let  $N, M$  respectively be the lengths of  $S_1, S_2$ . First, we give the trivial base cases: if either string is the empty string—i.e., if  $N = 0$  or  $M = 0$  (or both)—then clearly  $\text{LCS}(S_1, S_2) = 0$ , because the only common subsequence is the empty sequence.

Now suppose that both  $N, M \geq 1$ , and consider just the last character in each of the strings.<sup>13</sup> There are two possibilities: either these characters are the same, or they are different. We first consider the consequences of the former case, which is depicted in the following figure.



**Lemma 31** If  $S_1[N] = S_2[M]$ , then there exists some LCS  $C$  of  $S_1$  and  $S_2$  that is obtained by selecting both  $S_1[N]$  and  $S_2[M]$  (and therefore ends with that character).

We emphasize that Lemma 31 says not only that  $C$  ends with the *character* that appears at the end of both strings, but also that it *specifically selects* both  $S_1[N]$  and  $S_2[M]$ . For example, if  $S_1 = \text{xyx}$  and  $S_2 = \text{x}$ , then  $\text{x}$  is an LCS, but

<sup>12</sup> Contrast this with a *substring*, in which all the selected characters must be adjacent in the original string.

<sup>13</sup> It is equally valid to compare the first characters of each string, but it turns out to be more convenient to work with the final characters, due to the indexing in the recurrences.

selecting the *first*  $x$  from  $S_1$  would not satisfy the claim, while taking the final  $x$  from  $S_1$  would. (The distinction is important for what we will show below.)

**Proof 32** Let  $x$  denote the character  $S_1[N] = S_2[M]$ . First consider any common subsequence  $C'$  of  $S_1$  and  $S_2$  that selects *neither*  $S_1[N]$  nor  $S_2[M]$ . Then we can get a common subsequence that is *even longer* than  $C'$  by also selecting  $S_1[N] = S_2[M]$  and appending it to  $C'$ , so  $C'$  is not an LCS. Therefore, *any* LCS must select *at least one of*  $S_1[N]$  or  $S_2[M]$ .

Now let  $C^*$  be an arbitrary LCS of  $S_1$  of  $S_2$ . If  $C^*$  happens to select both  $S_1[N]$  and  $S_2[M]$ , then the claim holds with  $C = C^*$ , and we are done. So, suppose that  $C^*$  selects just one of those two characters, say  $S_1[N]$  without loss of generality (the other case proceeds symmetrically). Since  $C^*$  ends with  $x = S_1[N]$ , the final selected character of  $S_2$  is also  $x$ , which appears somewhere before  $S_2[M]$ . So, we can modify the selected characters of  $S_2$  to “unselect” that final  $x$  and select  $S_2[M] = x$  instead. This results in the same common subsequence string  $C = C^*$ , but it is obtained by selecting both  $S_1[N]$  and  $S_2[M]$ , and it is an LCS of  $S_1, S_2$  because  $C^*$  is an LCS of  $S_1, S_2$  by hypothesis. This proves the claim.  $\square$

We now get the following consequence of [Lemma 31](#). It says that when  $S_1[N] = S_2[M]$ , an LCS of  $S_1$  and  $S_2$  can be obtained by taking any LCS of the prefix strings  $S_1[1, \dots, N-1]$  and  $S_2[1, \dots, M-1]$ , then appending the final shared character.

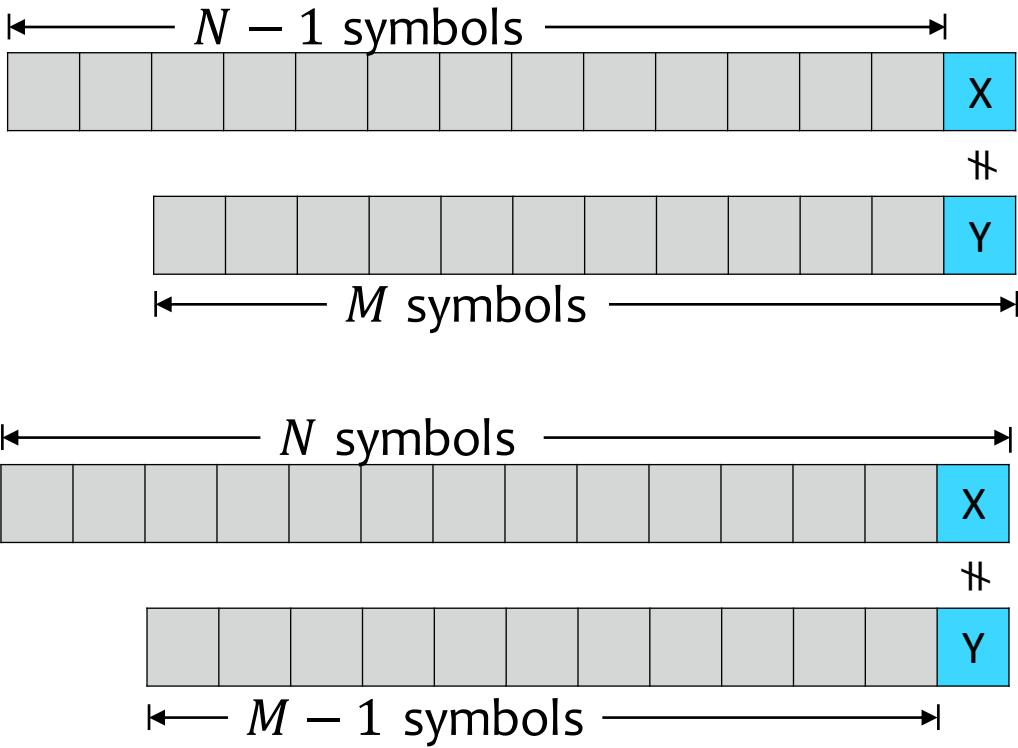
**Corollary 33** *If  $S_1[N] = S_2[M]$ , the common subsequences of  $S_1$  and  $S_2$  that select both  $S_1[N], S_2[M]$  are exactly the common subsequences of  $S_1[1, \dots, N-1]$  and  $S_2[1, \dots, M-1]$ , with  $S_1[N] = S_2[M]$  also selected and appended. It follows that*

$$LCS(S_1, S_2) = 1 + LCS(S_1[1, \dots, N-1], S_2[1, \dots, M-1]).$$

**Proof 34** In one direction, we can take any common subsequence of the prefix strings  $S_1[1, \dots, N-1], S_2[1, \dots, M-1]$ , then append  $S_1[N] = S_2[M]$ , to get a common subsequence of  $S_1, S_2$  that selects both  $S_1[N], S_2[M]$ . In the other direction, let  $C$  be any common subsequence of  $S_1, S_2$  that selects both  $S_1[N], S_2[M]$ ; these selections must correspond to the *final* character of  $C$ . So, removing the final character of  $C$  corresponds to “unselecting”  $S_1[N]$  and  $S_2[M]$ , which yields a common subsequence of the prefix strings. This proves the first claim.

For the second claim, the above correspondence implies that any *longest* common subsequence of  $S_1, S_2$  that selects both  $S_1[N], S_2[M]$  is in fact a *longest* common subsequence of the prefix strings, plus one character. Moreover, [Lemma 31](#) says that the “selects both  $S_1[N], S_2[M]$ ” restriction does not affect the optimal length, because there exists an LCS of  $S_1, S_2$  that meets this requirement. So, the LCS length for  $S_1, S_2$  (without any requirement on what characters are selected) is indeed one larger than the LCS length for the prefix strings, as claimed.  $\square$

Now we consider the case where the final characters of the two strings are different. The following lemma says that the LCS length is the maximum of the two LCS lengths where one of the strings remains unmodified, and the other is truncated by removing its final character. See the depiction in the following figure.



**Lemma 35** If  $S_1[N] \neq S_2[M]$ , then

$$LCS(S_1, S_2) = \max\{LCS(S_1[1, \dots, N-1], S_2), LCS(S_1, S_2[1, \dots, M-1])\}.$$

**Proof 36** In any common subsequence of  $S_1, S_2$ , the final character of at least one of the strings is not selected (because the final characters would have to appear at the end of the subsequence, and they do not match). Note that this could apply to either, or both, of the strings.

Next, observe that any common subsequence of  $S_1, S_2$  that does not select  $S_1[N]$  is also a common subsequence of  $S_1[1, \dots, N-1]$  and  $S_2$ , and vice versa. In other words, these two collections of subsequences are identical, and hence have the same maximum length.

Symmetrically, the same correspondence holds between the common subsequences of  $S_1, S_2$  that do not select  $S_2[M]$ , and the common subsequences of  $S_1$  and  $S_2[1, \dots, M-1]$ .

Since any common subsequence of  $S_1, S_2$  does not select  $S_1[N]$  or  $S_2[M]$  (or both), the length of a longest common subsequence of  $S_1, S_2$  is the maximum of the longest in each of the above two cases, as claimed.  $\square$

Putting all of the above together, we have arrived at our final overall recurrence for the LCS length:

**Theorem 37** For  $S_1 = S_1[1, \dots, N]$  and  $S_2 = S_2[1, \dots, M]$ ,

$$LCS(S_1, S_2) = \begin{cases} 0 & \text{if } N = 0 \text{ or } M = 0, \\ 1 + LCS(S_1[1, \dots, N-1], S_2[1, \dots, M-1]) & \text{if } S_1[N] = S_2[M], \\ \max(LCS(S_1[1, \dots, N-1], S_2), LCS(S_1, S_2[1, \dots, M-1])) & \text{if } S_1[N] \neq S_2[M]. \end{cases}$$

Now that we have a complete recurrence relation, we can proceed to give an algorithm, using the bottom-up approach.

We first observe that the recurrence refers only to subinputs consisting of a prefix  $S_1$  and a prefix of  $S_2$ . So, our algorithm will compute and store the value of the LCS function for every pair of such prefixes. That is, it will fill

an  $(N + 1)$ -by- $(M + 1)$  table in which the  $(i, j)$ th entry is  $\text{LCS}(S_1[1, \dots, i], S_2[1, \dots, j])$ , for all  $i \in [0, N]$  and  $j \in [0, M]$ . (By convention,  $S_1[1, \dots, 0]$  denotes the empty string, and similarly for  $S_2[1, \dots, 0]$ .)

For example, below is the complete table for the strings  $S_1 = \text{Go blue!}$  and  $S_2 = \text{Wolverines}$ , which has been filled using the recurrence from Theorem 37.

	W	o	l	v	e	r	i	n	e	s
0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0
o	0	0	1	1	1	1	1	1	1	1
b	0	0	1	1	1	1	1	1	1	1
l	0	0	1	2	2	2	2	2	2	2
u	0	0	1	2	2	2	2	2	2	2
e	0	0	1	2	2	3	3	3	3	3
!	0	0	1	2	2	3	3	3	3	3

As mentioned above, the  $(i, j)$ th entry of the table holds the LCS length for  $S_1[1, \dots, i]$  and  $S_2[1, \dots, j]$ . Using the recurrence relation, we can compute the value of the  $(i, j)$ th entry from the entries at the three locations  $(i - 1, j - 1)$ ,  $(i - 1, j)$ , and  $(i, j - 1)$ ; the first one is used when  $S_1[i] = S_2[j]$ , and the latter two are used when  $S_1[i] \neq S_2[j]$ . Entry  $(N, M)$  holds the LCS length for the full strings  $S_1[1, \dots, N]$  and  $S_2[1, \dots, M]$ .

The last thing we need before writing the algorithm is to determine a valid order in which to compute and fill the table entries. The only requirement is that before computing entry  $(i, j)$  for  $i, j > 0$ , the entries  $(i - 1, j - 1)$ ,  $(i - 1, j)$ ,  $(i, j - 1)$  should already have been computed and filled in. There are multiple orders that meet this requirement; we will compute the entries row by row from top to bottom (i.e., with increasing  $i$ ), moving left to right within each row (i.e., with increasing  $j$ ).

We now give the pseudocode for computing all the entries of the table.

**Algorithm 38 (LCS Table)**

**Input:** strings  $S_1, S_2$   
**Output:** table of LCS lengths for all prefixes  $S_1[1, \dots, i], S_2[1, \dots, j]$

```

function LCSTABLE( $S_1[1, \dots, N], S_2[1, \dots, M]$ )
    allocate table[0, ..., N][0, ..., M]
    for  $i = 0$  to  $N$  do ▷ base cases
        table[i][0] = 0
    for  $j = 0$  to  $M$  do
        table[0][j] = 0
    for  $i = 1$  to  $N$  do ▷ recursive cases
        for  $j = 1$  to  $M$  do
            if  $S_1[i] = S_2[j]$  then
                table[i][j] = 1 + table[i - 1][j - 1]
            else
                table[i][j] = max{table[i - 1][j], table[i][j - 1]}
    return table

```

Now that we have shown how to compute the *length* of a longest common subsequence, let's return to the original problem of computing such a subsequence itself. As in our previous examples, we can *backtrack* through the table to recover the characters of an LCS, from back to front. We start with the bottom-right entry ( $N, M$ ), and backtrack through a path until we reach some base-case entry. For each (non-base-case) entry  $(i, j)$  on the path, we check to see if the characters corresponding to that entry match, i.e., if  $S_1[i] = S_2[j]$ . If so, we prepend the matching character to our partial LCS, and we backtrack to entry  $(i - 1, j - 1)$ . If the characters do not match, we look at the entries above and to the left—namely,  $(i - 1, j)$  and  $(i, j - 1)$ —and backtrack to whichever one is larger (breaking a tie arbitrarily). This is because the larger of the two entries corresponds to the max value in the recurrence, i.e., it yields a longer completion of our partial LCS.

The following demonstrates a valid backtracking path for our example strings and LCS table. The solid path uses some arbitrary choices to break ties, while the dashed path always goes left in case of a tie. Both paths result in a valid LCS (and in this case, the same set of characters, though that isn't necessarily the case for all pairs of strings).

	W	o	l	v	e	r	i	n	e	s
W	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0
o	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
b	0	0	1	1	1	1	1	1	1	1
l	0	0	1	2	2	2	2	2	2	2
l	0	0	1	2	2	2	2	2	2	2
u	0	0	1	2	2	2	2	2	2	2
e	0	0	1	2	2	3	3	3	3	3
!	0	0	1	2	2	3	3	3	3	3

The algorithm for backtracking is as follows:

**Algorithm 39 (Longest Common Subsequence, via Backtracking)**

```

Input: strings  $S_1, S_2$ 
Output: a longest common subsequence of the strings
function LCS( $S_1[1, \dots, N], S_2[1, \dots, M]$ )
    table = LCSTABLE( $S_1, S_2$ )
     $s = \epsilon$  ▷ the empty string
     $i = N, j = M$ 
    while  $i > 0$  and  $j > 0$  do
        if  $S_1[i] = S_2[j]$  then
             $s = S_1[i] \parallel s$ 
             $i = i - 1, j = j - 1$ 
        else if table[ $i][j - 1] >$  table[ $i - 1][j]$  then
             $j = j - 1$ 
        else
             $i = i - 1$ 
    return  $s$ 

```

How efficient is this algorithm? Computing a single table entry requires a constant number of operations, because it simply compares two characters of the strings, looks at one or two neighboring entries, and either adds 1 or takes a maximum. Since there are  $(N + 1) \cdot (M + 1)$  entries overall, constructing the table takes  $O(NM)$  time and requires  $O(NM)$  space. Backtracking also does a constant number of operations per entry on the path, and the path takes at most  $N + M$  steps, so backtracking takes  $O(N + M)$  time. Thus, this algorithm uses a total of  $O(NM)$  time and space.

## 4.5 All-Pairs Shortest Paths

Suppose you are building a flight-aggregator website. Each day, you receive a list of flights from several airlines with their associated costs, and some flight segments may actually have negative cost if the airline wants to incentivize a particular route (see [hidden-city ticketing](#)<sup>14</sup> for an example of how this can be exploited in practice, and what the perils of doing so are). You'd like your users to be able to find the cheapest itinerary from point A to point B. To provide this service efficiently, you determine in advance the cheapest itineraries between all possible origin and destination locations, so that you need only look up an already computed result when the user puts in a query.

This situation is an example of the *all-pairs shortest path* problem. The set of cities and flights can be represented as a graph  $G = (V, E)$ , with the cities represented as vertices and the flight segments as edges in the graph. There is also a weight function  $weight : E \rightarrow \mathbb{R}$  that maps each edge to a cost. While an individual edge may have a negative cost, no negative cycles are allowed. (Otherwise a traveler could just fly around that cycle to make as much money as they want, which would be very bad for the airlines!) Our task is to find the lowest-cost path between all pairs of vertices in the graph.

How can we apply dynamic programming to this problem? We need to formulate it such that there are self-similar subproblems. To gain some insight, we observe that many aggregators allow the selection of *layover* airports, which are intermediate stops between the origin and destination, when searching for flights. The following is an example from [kayak.com](#)<sup>15</sup>.

<sup>14</sup> [https://en.wikipedia.org/wiki/Airline\\_booking\\_ploys#Hidden-city\\_ticketing](https://en.wikipedia.org/wiki/Airline_booking_ploys#Hidden-city_ticketing)

<sup>15</sup> <https://kayak.com>

Layover airports	Layover airports
<input checked="" type="checkbox"/> Atlanta (ATL)	<input checked="" type="checkbox"/> Atlanta (ATL)
<input type="checkbox"/> Baltimore (DD8)	<input type="checkbox"/> Baltimore (DD8)
<input checked="" type="checkbox"/> Boston (BOS)	<input type="checkbox"/> Boston (BOS)
<input checked="" type="checkbox"/> Buffalo (BUF)	<input type="checkbox"/> Buffalo (BUF)
<input checked="" type="checkbox"/> Charlotte (CLT)	<input type="checkbox"/> Charlotte (CLT)
<input checked="" type="checkbox"/> Chicago (ORD)	<input type="checkbox"/> Chicago (ORD)
<input checked="" type="checkbox"/> Cincinnati (CVG)	<input type="checkbox"/> Cincinnati (CVG)
<input checked="" type="checkbox"/> Cleveland (CLE)	<input type="checkbox"/> Cleveland (CLE)
<input checked="" type="checkbox"/> Columbus (CMH)	<input type="checkbox"/> Columbus (CMH)
<input type="checkbox"/> Denver (DEN)	<input type="checkbox"/> Denver (DEN)
<input checked="" type="checkbox"/> Detroit (DTW)	<input type="checkbox"/> Detroit (DTW)
<input checked="" type="checkbox"/> Fort Lauderdale (FLL)	<input type="checkbox"/> Fort Lauderdale (FLL)
<input checked="" type="checkbox"/> Miami (MIA)	<input type="checkbox"/> Miami (MIA)
<input checked="" type="checkbox"/> Myrtle Beach (MYR)	<input type="checkbox"/> Myrtle Beach (MYR)
<input checked="" type="checkbox"/> Norfolk (ORF)	<input type="checkbox"/> Norfolk (ORF)
<input checked="" type="checkbox"/> Orlando (MCO)	<input type="checkbox"/> Orlando (MCO)

We take the set of allowed layover airports as one of the key characteristics of a subproblem – computing shortest paths with a smaller set of allowed layover airports is a subproblem of computing shortest paths with a larger set of allowed layover airports. Then the base case is allowing only direct flights, with no layover airports.

Coming back to the graph representation of this problem, we formalize the notion of a layover airport as an *intermediate vertex* of a simple path, which is a path without cycles. Let  $p = \{v_1, v_2, \dots, v_m\}$  be a path from origin  $v_1$  to destination  $v_m$ . Then  $v_2, \dots, v_{m-1}$  are intermediate vertices.

Assume that the vertices are labeled as numbers in the set  $\{1, 2, \dots, |V|\}$ . We parameterize a subproblem by  $k$ , which signifies that the allowed set of intermediate vertices (layover airports) is restricted to  $\{1, 2, \dots, k\}$ . Then we define  $d^k(i, j)$  to be the length of the shortest path between vertices  $i$  and  $j$ , where the path is only allowed to go through intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

We have already determined that when no intermediate vertices are allowed, which is when  $k = 0$ , the shortest path between  $i$  and  $j$  is just the direct edge (flight) between them. Thus, our base case is

$$d^0(i, j) = \text{weight}(i, j)$$

where  $\text{weight}(i, j)$  is the weight of the edge between  $i$  and  $j$ .

We proceed to the recursive case. We have at our disposal the value of  $d^{k-1}(i', j')$  for all  $i', j' \in V$ , and we want to somehow relate  $d^k(i, j)$  to those values. The latter represents adding vertex  $k$  to our set of permitted intermediate vertices. There are two possible cases for the shortest path between  $i$  and  $j$  that is allowed to use any of the intermediate vertices  $1, 2, \dots, k$ :

- Case 1: The path does not go through  $k$ . Then the length of the shortest path that is allowed to go through  $1, 2, \dots, k$  is the same as that of the shortest path that is only allowed to go through  $1, 2, \dots, k-1$ , so  $d^k(i, j) = d^{k-1}(i, j)$ .
- Case 2: The path does go through  $k$ . Then this path is composed of two segments, one that goes from  $i$  to  $k$

and another that goes from  $k$  to  $j$ . We minimize the cost of the total path by minimizing the cost of each of the segments – the costs respect the principle of optimality.

Neither of the two segments may have  $k$  as an intermediate vertex – otherwise we would have a cycle. The only way for a path with a cycle to have lower cost than one without is for the cycle as a whole to have negative weight, which was explicitly prohibited in our problem statement. Since  $k$  is not an intermediate vertex in the segment between  $i$  and  $k$ , the shortest path between them that is allowed to go through intermediate vertices  $1, 2, \dots, k$  is the same as the shortest path that is only permitted to go through  $1, 2, \dots, k - 1$ . In other words, the length of this segment is  $d^{k-1}(i, k)$ . By the same reasoning, the length of the segment between  $k$  and  $j$  is  $d^{k-1}(k, j)$ .

Thus, we have that in this case,  $d^k(i, j) = d^{k-1}(i, k) + d^{k-1}(k, j)$ .

We don't know a priori which of these two cases holds, but we can just compute them both and take the minimum. This gives us the recursive case:

$$d^k(i, j) = \min(d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j))$$

Combining this with the base case, we have our complete recurrence relation:

$$d^k(i, j) = \begin{cases} \text{weight}(i, j) & \text{if } k = 0 \\ \min(d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j)) & \text{if } k \neq 0 \end{cases}$$

We can now construct a bottom-up algorithm to compute the shortest paths:

#### Algorithm 40 (Floyd-Warshall)

```

Input: a weighted directed graph
Output: all-pairs (shortest-path) distances in the graph
function FLOYDWARSHALL( $G = (V, E)$ )
    for all  $u, v \in V$  do
         $d_0(u, v) = \text{weight}(u, v)$ 
    for  $k = 1$  to  $|V|$  do
        for all  $u, v \in V$  do
             $d_k(u, v) = \min\{d_{k-1}(u, v), d_{k-1}(u, k) + d_{k-1}(k, v)\}$ 
    return  $d_{|V|}$ 
```

This is known as the *Floyd-Warshall* algorithm, and it runs in time  $O(|V|^3)$ . The space usage is  $O(|V|^2)$  if we only keep around the computed values of  $d^m(i, j)$  for iterations  $m$  and  $m + 1$ . Once we have computed these shortest paths, we need only look up the already computed result to find the shortest path between a particular origin and destination.

---

CHAPTER  
FIVE

---

## GREEDY ALGORITHMS

A *greedy algorithm* computes a solution to an optimization problem by making (and committing to) a sequence of locally optimal choices. In general, there is no guarantee that such a sequence of locally optimal choices produces a global optimum. However, for some specific problems and greedy algorithms, we can prove that the result is indeed a global optimum.

As an example, consider the problem of finding a *minimum spanning tree (MST)* of a weighted, connected, undirected graph. Given such a graph, we would like to find a subset of the edges so that the subgraph induced by those edges touches every vertex, is connected, and has minimum total edge cost. This is an important problem in designing networks, including transportation and communication networks, where we want to ensure that there is a path between any two vertices while minimizing the overall cost of the network.

Before we proceed, let's review the definition of a *tree*. There are three equivalent definitions:

**Definition 41 (Tree #1)** An undirected graph  $G$  is a *tree* if it is connected and acyclic (i.e., has no cycle).

A graph is *connected* if for any two vertices there is a path between them. A *cycle* is a nonempty sequence of adjacent edges that starts and ends at the same vertex.

**Definition 42 (Tree #2)** An undirected graph  $G$  is a tree if it is *minimally connected*, i.e., if it is connected, and removing any single edge causes it to become disconnected.

**Definition 43 (Tree #3)** An undirected graph  $G$  is a tree if it is *maximally acyclic*, i.e., if it has no cycle, and adding any single edge causes it to have a cycle.

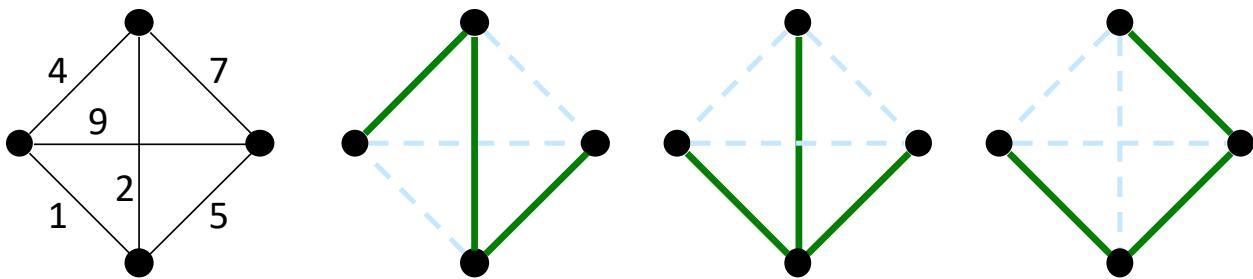
**Exercise 44** Show that the three definitions of a tree are equivalent.

**Definition 45 (Minimum spanning tree)** A *minimum spanning tree (MST)* of a connected graph is a subset of its edges that:

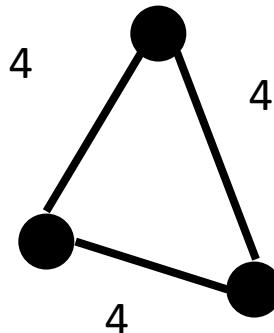
- connects all the vertices,
- is acyclic,
- and has minimum total edge weight over all subsets that meet the first two requirements.

The first two requirements imply that an MST (together with all the graph's vertices) is indeed a tree, by [Definition 41](#). Since the tree spans all the vertices of the graph, we call it a *spanning tree*. A *minimum spanning tree* is then a spanning tree that has the minimum weight over all spanning trees of the original graph.

The following illustrates three spanning trees of an example graph. The middle one is an MST, since its total weight of 8 is no larger than that of any other spanning tree.



A graph may have multiple minimum spanning trees (so we say “*an* MST,” not “*the* MST,” unless we have some specific MST in mind, or have a guarantee that there is a unique MST in the graph). In the following graph, any two edges form an MST.



Now that we understand what a minimum spanning tree is, let’s consider *Kruskal’s algorithm*, a greedy algorithm for computing an MST in a given graph. The algorithm simply examines the edges in sorted order by weight (from smallest to largest), selecting any edge that does not induce a cycle when added to the set of already-selected edges. It is “greedy” because it repeatedly selects an edge of minimum weight that does not induce a cycle (a locally optimal choice), and once it selects an edge, it never “un-selects” it (each choice is committed).

#### Algorithm 46 (Kruskal)

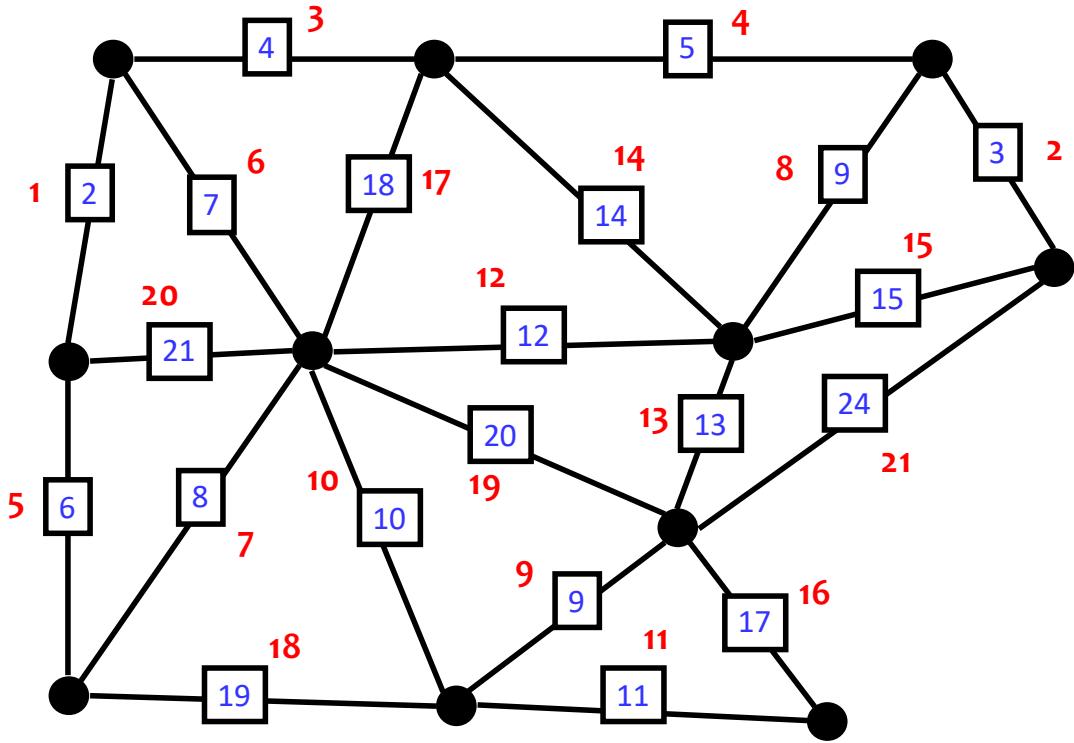
**Input:** a weighted, connected, undirected graph

**Output:** a minimum spanning tree of the graph

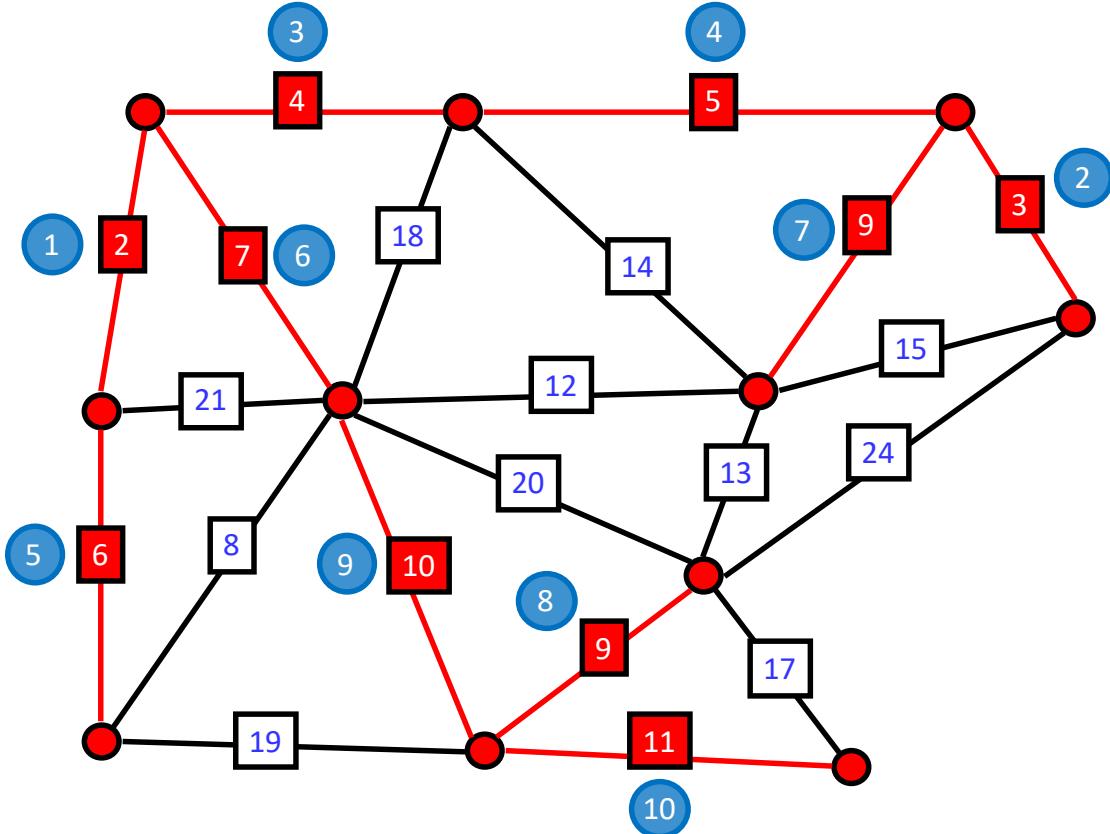
```

function KRUSKALMST( $G = (V, E)$ )
     $S = \emptyset$  ▷ empty set of edges
    for all edges  $e \in E$ , in increasing order by weight do
        if  $S \cup \{e\}$  does not have a cycle then
             $S = S \cup \{e\}$ 
    return  $S$ 
```

As an example, let’s see how Kruskal’s algorithm computes an MST of the following graph. We start by sorting the edges by their weights.



Then we consider the edges in order, including each edge in our partial result as long as adding it does not introduce a cycle among our selected edges.



Observe that when the algorithm considers the edge with weight 8, the two incident vertices are already connected by the already-selected edges, so adding that edge would introduce a cycle. Thus, the algorithm skips it and continues on to the next edge. In this graph, there are two edges of weight 9, so the algorithm arbitrarily picks one of them to consider first. The algorithm terminates when all edges have been examined. For this graph, the resulting spanning tree has a total weight of 66.

As stated previously, for many problems it is not the case that a sequence of locally optimal choices yields a global optimum. But as we will now show, for the MST problem, it turns out that Kruskal's algorithm does indeed produce a minimum spanning tree.

**Claim 47** *The output of Kruskal's algorithm is a tree.*

To prove this, we will assume for convenience that the input graph  $G$  is *complete*, meaning that there is an edge between every pair of vertices. We can make any graph complete by adding the missing edges with infinite weight, so that the new edges do not change the minimum spanning trees.

**Proof 48** Let  $T$  be the output of Kruskal's algorithm, which is the final value of  $S$ , on a complete input graph  $G$ . Recall from [Definition 43](#) that a tree is a maximally acyclic graph. Clearly  $T$  is acyclic, since Kruskal's algorithm initializes  $S$  to be the empty set, which is acyclic, and it adds an edge to  $S$  only if it does not induce a cycle.

So, it just remains to show that  $T$  is maximal. By definition, we need to show that for any potential edge  $e \notin T$  between two vertices of  $T$ , adding  $e$  to  $T$  would introduce a cycle. The input graph is complete, and the algorithm examined each of its edges, so it must have considered  $e$  at some point. Because  $T$  does not contain  $e$ , and no edge was ever removed from  $S$ , the algorithm did not add  $e$  to  $S$  when it considered  $e$ . Recall that when the algorithm considers an edge, it leaves it out of  $S$  only if it would create a cycle in  $S$  (at that time). So, since it did not add  $e$  to  $S$ , doing so would have induced a cycle at the time. And since no edges are ever removed from  $S$ , adding  $e$  to the final output  $T$  also would induce a cycle, which is what we set out to show. Thus,  $T$  is maximal, as desired.  $\square$

We can similarly demonstrate that the output of Kruskal's algorithm is a *spanning tree*, but we leave that as an exercise.

**Exercise 49** Show that if the input graph  $G$  is connected, then the output of Kruskal's algorithm spans all the vertices of  $G$ .

Next, we show that the result of Kruskal's algorithm is a minimum spanning tree. To do so, we actually prove a stronger claim:

**Claim 50** *At any point in Kruskal's algorithm on an input graph  $G$ , let  $S$  be the set of edges that have been selected so far. Then there is some minimum spanning tree of  $G$  that contains  $S$ .*

Since this claim holds at any point in Kruskal's algorithm, in particular it holds for the final output set of edges  $T$ , i.e., there is an MST that contains  $T$ . Since we have already seen above that  $T$  is a spanning tree, no edge of the graph can be added to  $T$  without introducing a cycle, so we can conclude that the MST containing  $T$  as guaranteed by [Claim 50](#) must be  $T$  *itself*, and hence  $T$  is an MST.

**Proof 51** We prove [Claim 50](#) by induction over the size of  $S$ , i.e., the sequence of edges added to it by the algorithm.

- **Base case:**  $S = \emptyset$  is the empty set. Every MST trivially contains  $\emptyset$  as a subset, so the claim holds.
- **Inductive step:** Let  $T$  be an MST that contains  $S$ . Suppose the algorithm would next add edge  $e$  to  $S$ . We need to show that there is some MST  $T'$  that contains  $S \cup \{e\}$ .

There are two possibilities: either  $e \in T$ , or not.

- Case 1:  $e \in T$ . The claim follows immediately: since  $T$  is an MST that contains  $S$ , and also  $e \in T$ , then  $T$  is an MST that contains  $S \cup \{e\}$ . (In other words, we can take  $T' = T$  in this case.)
- Case 2:  $e \notin T$ . Then by [Definition 43](#),  $T \cup \{e\}$  contains some cycle  $C$ , and  $e \in C$  because  $T$  alone

is acyclic.

By the code of Kruskal's algorithm, we know that  $S \cup \{e\}$  does not contain a cycle. Thus, there must be some edge  $f \in C$  for which  $f \notin S \cup \{e\}$ , and in particular,  $f \neq e$ . Since  $f \in C \subseteq T \cup \{e\}$ , we have that  $f \in T$ .

Observe that  $S \cup \{f\} \subseteq T$ , since  $S \subseteq T$  by the inductive hypothesis. Since  $T$  does not contain a cycle, neither does  $S \cup \{f\}$ .

Since adding  $f$  to  $S$  would not induce a cycle, the algorithm *must not have considered  $f$  yet* (at the time it considers  $e$  and adds it to  $S$ ), or else it would have added  $f$  to  $S$ . Because the algorithm considers edges in sorted order by weight, and it has considered  $e$  but has not considered  $f$  yet, it must be that  $w(e) \leq w(f)$ .

Now define  $T' = T \cup \{e\} \setminus \{f\}$ , which has weight  $w(T') = w(T) + w(e) - w(f) \leq w(T)$ . Moreover,  $T'$  is a spanning tree of  $G$ : for any two vertices, there is a path between them that follows such a path in  $T$ , but instead of using edge  $f$  it instead goes the “other way around” the cycle  $C$  using the edges of  $C \setminus \{f\} \subseteq T'$ .

Since  $T'$  is a spanning tree whose weight is no larger than that of  $T$ , and  $T$  is an MST,  $T'$  is also an MST.<sup>16</sup> And since  $S \subseteq T$  and  $f \notin S$ , we have that  $S \cup \{e\} \subseteq T \cup \{e\} \setminus \{f\} = T'$ . Thus,  $T'$  is an MST that contains  $S \cup \{e\}$ , as needed.

<sup>16</sup> This actually shows that  $w(T') = w(T)$  and hence  $w(e) = w(f)$ , but these facts are not needed for this proof.

We have proved that Kruskal's algorithm does indeed output an MST of its input graph. We also state without proof that its running time on an input graph  $G = (E, V)$  is  $O(|E| \log |E|)$ , by using an appropriate choice of supporting data structure to detect for cycles.

Observe that the analysis of Kruskal's algorithm is nontrivial. This is often the case for greedy algorithms. Again, it is usually not the case that locally optimal choices lead to a global optimum, so we typically need to do significant work to demonstrate that this is actually the case for a particular problem and algorithm.

A standard strategy for proving that a greedy algorithm correctly solves a particular optimization problem proceeds by establishing a “greedy choice” property, often by means of an exchange-style argument like the one we gave above for MSTs. A “greedy choice” property consists of two parts:

- Base case: the algorithm's initial state is contained in *some* optimal solution. Since a typical greedy algorithm's initial state is the empty set, this property is usually easy to show.
- Inductive step: for any (greedy) choice the algorithm makes, there *remains* an optimal solution that contains the algorithm's set of choices so far. (Observe that [Claim 50](#) has this form.)

With a greedy-choice property established, by induction, the algorithm's set of choices is at all times contained in *some* optimal solution. So, it just remains to show that the final output consists of a full solution (not just a partial one); it therefore must be an optimal solution.

To show the inductive step of the greedy-choice property, we have the inductive hypothesis that the previous choices  $S$  are contained in some optimal solution  $\text{OPT}$ , and we need to show that the updated choices  $S \cup \{s\}$  are contained in some optimal solution  $\text{OPT}'$ . If  $s \in \text{OPT}$ , then the claim holds trivially. But if  $s \notin \text{OPT}$ , then we aim to invoke some exchange argument, which modifies  $\text{OPT}$  to some *other* optimal solution  $\text{OPT}'$  that contains  $S \cup \{s\}$ . Indeed, this is exactly what we did in the correctness proof for Kruskal's argument, by changing  $\text{OPT} = T$  to  $\text{OPT}' = T' = T \cup \{e\} \setminus \{f\}$ , for some carefully identified edge  $f \in T$  whose weight is no smaller than that of  $e$ . This means that  $\text{OPT}'$  is also an optimal solution, as needed.

## **Part II**

# **Computability**

## INTRODUCTION TO COMPUTABILITY

We now turn our attention to fundamental questions about computation. While we have seen several algorithmic approaches to solving problems, when faced with a new problem, the first question we should consider is whether it is solvable at all. Otherwise, we might waste a lot of time on a fruitless effort to find an algorithm where none exists!

Before we can reason about what problems are solvable on a computer, we first need to define, in a general and abstract way, what a *problem* is, and what a *computer* (or *algorithm*) is. Rather than basing our answers on specific hardware and software technologies, programming languages, and implementation details (which go in and out of fashion over time), we want to develop *simple, fundamental abstractions* that are capable of modeling all kinds of computations. These abstractions will be easier to reason about than real computers and programs. Moreover, our abstractions will ideally be strong enough to capture all possible implementations of “computation”, so that results in our model will hold for all real computers, now and into the future.

### 6.1 Formal Languages

We start by defining a unifying abstraction to represent computational problems. The simplest kind of output a problem can have is a yes-or-no answer, like in this question:

- Is there a flight from Detroit to New York for less than \$100?

We can generalize the question above into a *predicate* that takes an input value:

- Is there a flight from Detroit to  $x$  for less than \$100?

For any particular destination  $x$ , the answer to this question is still either yes or no. We can further generalize this predicate to work with multiple input values:

- Is there a flight from  $x$  to  $y$  for less than  $z$ ?

While this predicate is expressed as taking three inputs  $x$ ,  $y$ , and  $z$ , we can equivalently consider it to be a predicate that takes a single input tuple  $w = (x, y, z)$  of those three components. Thus, we lose no generality in restricting to predicates that have a single input.

Another name for a predicate (which we will use more often) is a *decision problem*: given an input, the answer is simply a yes-or-no decision.

A predicate is applicable to some universe of inputs, such as all airports or cities, for instance. (In programming languages, a *type* is used to represent a universe of values.) The predicate is true (has a “yes” answer) for some subset of the inputs, and is false (a “no” answer) for all the other inputs. We call the set of “yes” instances the *language* of the predicate. The following is the language defined by our second predicate above:

$$L_{\text{getaways}} = \{x : \text{there is a flight from Detroit to } x \text{ for less than \$100}\}.$$

Therefore, we have recast the question “is there a flight from Detroit to  $x$  for less than \$100?” as the decision problem “is  $x \in L_{\text{getaways}}$ ?” Any predicate can be recast in such a way, as the decision problem of determining whether the input is in the corresponding language, and vice-versa.

The *complement* language is the subset of inputs that are not in the language, i.e., those for which the answer is “no”. We represent the complement using an overbar:

$$\overline{L_{\text{getaways}}} = \{x : \text{there is no flight from Detroit to } x \text{ for less than \$100}\} .$$

So far, we have unified computational problems under the abstraction of languages and their decision problems, of determining whether the input is in the language. Now we also unify the notion of “input” under a common abstraction: as a *string* over an *alphabet*.

To represent inputs, we first fix an *alphabet*, which we often denote by the Greek letter capital-Sigma, written  $\Sigma$ . (Warning: this is the same Greek letter used to denote summation, but in this context it has a completely different meaning.<sup>17</sup>)

**Definition 52 (Alphabet)** An alphabet is some *nonempty, finite* set of symbols.

As a few examples:

- $\Sigma_{\text{binary}} = \{0, 1\}$  is the alphabet consisting of just the symbols 0 and 1.
- $\Sigma_{\text{lowercase}} = \{a, b, \dots, z\}$  is the alphabet consisting of lowercase English letters.
- $\Sigma_{\text{ASCII}}$  can be defined as the alphabet consisting of all the symbols in the ASCII character set.
- $\Sigma_{\text{Greek}} = \{\alpha, \beta, \gamma, \dots, \omega\}$  is the alphabet consisting of the lower-case Greek letters.
- $\Sigma_{\text{BigTen}}$  can be defined as the alphabet consisting of all the schools in the Big Ten conference: Michigan, Indiana, Wisconsin, Ohio State, etc. (To depict them visually, we might use their logos.)

We emphasize that an alphabet can be *any* nonempty set of elements of any kind, as long as it is *finite*. So, for example, the set of natural numbers  $\mathbb{N} = \{0, 1, 2, \dots\}$  is *not* a valid alphabet.

The binary alphabet  $\Sigma = \{0, 1\}$  is often taken as the “default” choice, both because it is the “simplest” nontrivial alphabet, and modern computers operate on 0s and 1s at the lowest level. But fundamentally, we can use whatever alphabet we find convenient for our specific purpose.

**Definition 53 (String)** A *string* over an alphabet  $\Sigma$  is a *finite, ordered* sequence of symbols from  $\Sigma$ .

We stress that a string must have *finite* length. (Below we sometimes preface the term “string” by “(finite-length)” for emphasis, but this is redundant and does not change the meaning.)

For example, a binary string consists of any finite sequence of 0s and 1s (the symbols of  $\Sigma_{\text{binary}} = \{0, 1\}$ ), while every (lower-case) English word is a sequence of letters from the alphabet  $\Sigma_{\text{lowercase}} = \{a, b, \dots, z\}$ . (Not every string over this alphabet is a real English word, however.)

The concept of a string appears in most programming languages, which usually specify a “string” data type for sequences of elements over a certain character set (e.g., ASCII or UTF); the character set is the string’s alphabet. Programming languages often specify notation for *string literals*, such as enclosing a sequence of characters within matching double quotes. This notation itself is not part of the string data; for instance, the string literal "abcd" in C++ and Python represents the four-character<sup>18</sup> sequence *abcd*, and the quotation marks are not part of this sequence.

Note that the *empty sequence* of symbols is a valid string, over any alphabet. For instance, "" represents an empty string in C++ and Python. In computability, the standard notation for the empty string is the symbol  $\varepsilon$  (called a *varepsilon* in some formatting systems like LaTeX).

We typically display a nonempty string simply by writing out its symbols in order. For instance, 1011 is a string over the alphabet  $\Sigma_{\text{binary}}$ , consisting of the symbol 1, followed by the symbol 0, etc. The *length* of a string  $w$  is the number of symbols in  $w$ , and is denoted  $|w|$ ; for example,  $|1011| = 4$ .

<sup>17</sup> In LaTeX, the symbol is obtained using the command \Sigma, and it is also typeset differently than a summation (which is obtained using the command \sum).

<sup>18</sup> In C++, the character-array string representation also includes a null terminator to indicate the end of the string, but that is an implementation detail. Other implementations do not use this representation.

To indicate the set of all strings of a certain length over an alphabet, we use “product” and “power” notation for sets. Recall that the set product  $A \times B$  is the set of all *pairs* where the first component is an element of  $A$  and the second component is an element of  $B$ . So, for  $\Sigma = \{0, 1\}$ , its product with itself is  $\Sigma \times \Sigma = \{00, 01, 10, 11\}$ , and similarly for products of more copies of  $\Sigma$ . For a set  $A$  and a non-negative integer  $k$ , the notation  $A^k$  means the set product of  $k$  copies of  $A$ . For example:

$$\begin{aligned}\{0, 1\}^0 &= \{\varepsilon\} \quad (\text{because } \varepsilon \text{ is the only length-0 string}) \\ \{0, 1\}^1 &= \{0, 1\} \\ \{0, 1\}^2 &= \{0, 1\} \times \{0, 1\} = \{00, 01, 10, 11\} \\ \{0, 1\}^3 &= \{000, 001, 010, 011, 100, 101, 110, 111\}\end{aligned}$$

In general, then,  $\Sigma^k$  is the set of length- $k$  strings over  $\Sigma$ .

The notation  $\Sigma^*$  denotes the set of *all* (finite-length) strings over the alphabet  $\Sigma$ . Here, the “star” superscript \* is known as the *Kleene star* operator, whose formal definition is

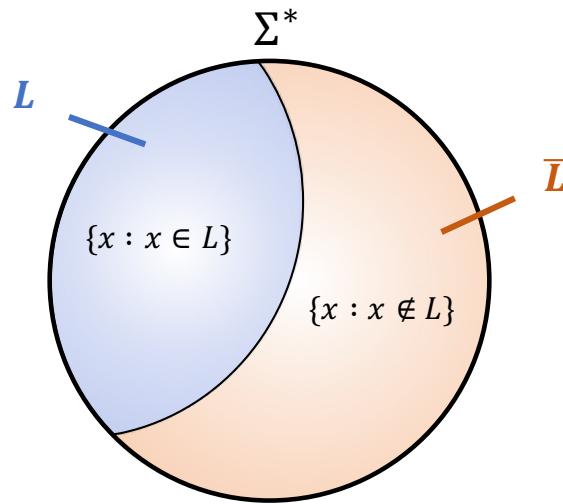
$$\Sigma^* = \bigcup_{k \geq 0} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

The above definition says that  $\Sigma^*$  is made up of all strings of length 0 over  $\Sigma$ , all strings of length 1 over  $\Sigma$ , all strings of length 2, and so on, over all finite lengths  $k$ . For example,

$$\begin{aligned}\Sigma_{\text{binary}}^* &= \{0, 1\}^* \\ &= \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \dots\}\end{aligned}$$

is the set of all (finite-length) binary strings. It is very important to understand that while  $\Sigma^*$  is an *infinite set* (for any alphabet  $\Sigma$ )—i.e., it has infinitely many elements (strings)—*each individual string* in  $\Sigma^*$  has some *finite length*. This is analogous to the natural numbers: there are infinitely many of them, but each one has some finite value (there is no natural number “infinity”).

**Definition 54 (Language)** A *language*  $L$  over an alphabet  $\Sigma$  is a subset of  $\Sigma^*$ , i.e.,  $L \subseteq \Sigma^*$ . In other words, it is a set of (finite-length) strings over  $\Sigma$ .



The following are examples of languages:

- $L_1 = \{11y : y \in \{0, 1\}^*\}$  is the language over  $\Sigma_{\text{binary}}$  consisting of all binary strings that begin with two ones.
- $L_2 = \{\text{hello}, \text{world}\}$  is the language over  $\Sigma_{\text{lowercase}}$  consisting of just the two strings *hello* and *world*.

- The *empty language* (empty set)  $\emptyset$  is a language over any alphabet  $\Sigma$ , which consists of no strings.

It is important to understand that the empty *language*  $\emptyset$  is different from the empty *string*  $\varepsilon$ : the former is a set that has no elements, while the latter is an individual string (ordered sequence) that has no characters. (The difference is akin to that between the values produced by `set()` and `str()` with no arguments in Python, or `std::set<std::string>{}` and `std::string{}` in C++.) Moreover, the language  $\{\varepsilon\}$  is different still: it is the set that consists of just one string, and that element is the empty string.

As illustrated above, some languages are finite, while others have infinitely many strings.

According to the definition, we can view the words of the English language itself as comprising a formal language over the English alphabet: it a set of strings made up of English characters. (However, note that this view does not capture any of the grammar or meaning of the English language, just its set of words.)

In computing, we often use the binary alphabet  $\{0, 1\}$ . Any data value can be represented as a binary string—in fact, real computers store all data in binary, from simple text files to images, audio files, and this very document itself. Thus, the languages we work with generally consist of binary strings. When we express a language such as

$$L_{\text{getaways}} = \{x : \text{there is a flight from Detroit to } x \text{ for less than \$100}\},$$

what we actually mean can be more formally written as:

$$L_{\text{getaways}} = \{y : y \text{ is the binary representation of an airport } x \text{ for which} \\ \text{there is a flight from Detroit to } x \text{ for less than \$100}\}.$$

We often use angle-brackets notation  $\langle x \rangle$  to represent the binary encoding of the value  $x$ . We can then express the above more concisely as:

$$L_{\text{getaways}} = \{\langle x \rangle : \text{there is a flight from Detroit to } x \text{ for less than \$100}\}.$$

However, for notational simplicity we often elide the angle brackets, making the binary encoding of the input implicit.

Now that we have a formal concept of a language (i.e., the “yes” instances of a decision problem), *solving* a decision problem entails determining whether a particular input is a member of that language. An algorithm  $M$  that solves a problem  $L$  is one that:

1. takes some arbitrary input  $x$ ,
2. performs some computations,
3. outputs “yes” — in other words, *accepts* — if  $x \in L$ ;
4. outputs “no” — in other words, *rejects* — if  $x \notin L$ .

If this is the case, we say that  $M$  *decides*  $L$ .

Can every language  $L$  be decided? In other words, is it the case that for any language  $L$ , there exists some algorithm  $M$  that decides  $L$ ? To answer this question, we need a formal definition of what an algorithm really is.

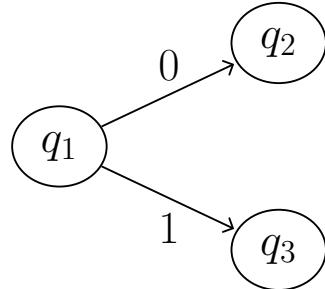
## 6.2 Overview of Automata

In computability theory, an abstract computing device is known as an *automaton* (plural: *automata*). There are numerous different abstract models of computation, such as state machines, recursive functions, lambda calculus, von Neumann machines, cellular automata, and so on. We will primarily concern ourselves with the Turing-machine model, though we will first briefly examine a more restricted variant called (deterministic) finite automata.

Both the finite-automata and Turing-machine models are forms of *state machines*. A machine includes a *finite* set of *states*, each representing a discrete status of a computation, and the *state transitions* the machine follows as it computes. An analogous concept in a C++ or Python program is a line of code (or program counter in a compiled executable): a

program has a *finite* number of lines, execution of the program is at a single line at any point in time, and the program transitions from one line to another as it runs (possibly revisiting lines over time, as in a loop).

In a state machine, state transitions are based on what is read from the input, what is in the machine's memory (if it has a separate memory), or both. We represent states and transitions graphically as follows:

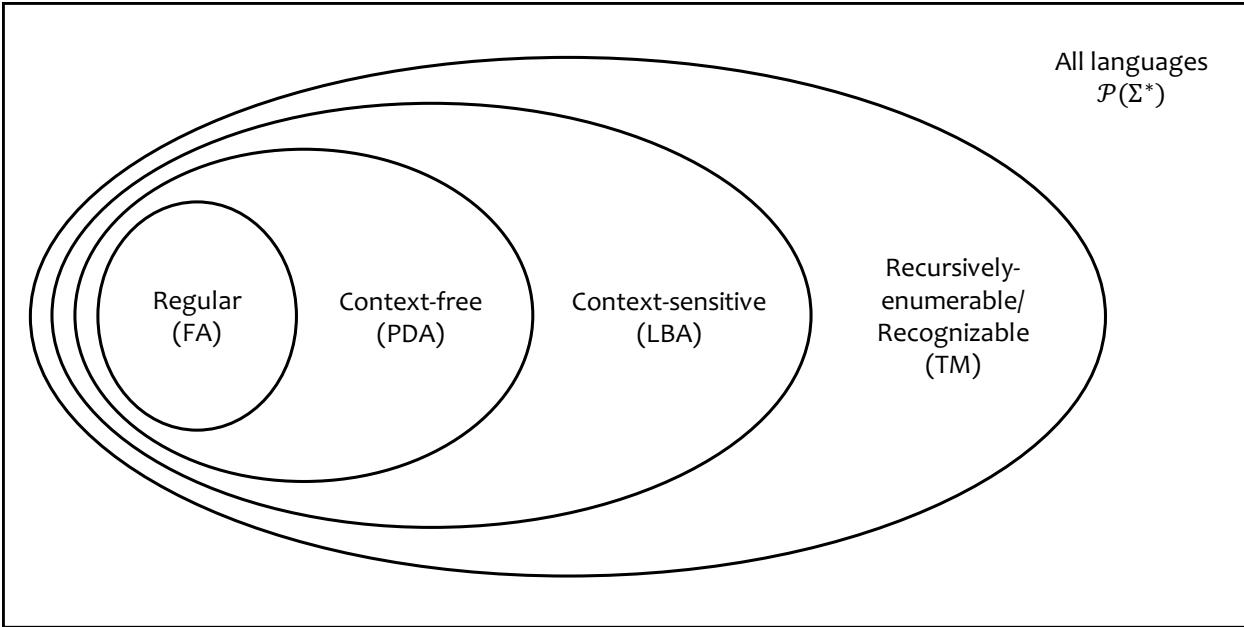


States are drawn as vertices in the graph, often labeled by names. A directed edge represents a transition, and the label denotes the conditions under which the transition is taken, as well as any side effects (e.g., writing a symbol to memory) of the transition. Mathematically, we represent the states as a finite set, and state transitions as a *transition function*, with states and other conditions as components of the domain, and states and side effects as components of the codomain.

The primary difference between the two models we'll discuss (finite automata and Turing machines) is what kind of "memory" the model has access to. This has a significant effect on the computational power of a machine. More generally, the *Chomsky hierarchy* is a collection of various classes of languages, where each class corresponds to what can be computed on a particular kind of state machine:

- *Regular languages* correspond to the computational power of *finite automata* (*FA* in the diagram below), which have no memory beyond their own states.
- *Context-free languages* correspond to *pushdown automata* (*PDA* in the diagram below), which have a single "stack" as memory, and can interact only with the top of the stack (pushing to and popping from it).
- *Context-sensitive languages* are computable by *linear-bounded automata* (*LBA* in the diagram below), which have access to memory that is *linear* in size with respect to the input, and which can be accessed in any position.
- *Recursively-enumerable languages*, also called *recognizable languages*, correspond to the power of *Turing machines* (*TM* in the diagram below), which have access to *unbounded* memory that can be accessed in any position.

Each of these classes of languages is strictly contained within the next, demonstrating the effect that the memory system has on computational power.

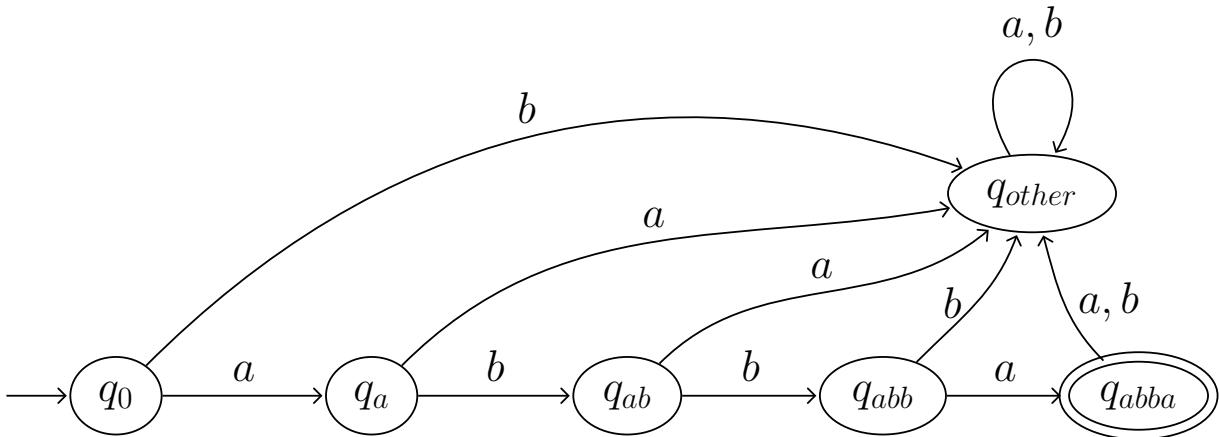


We next examine the finite-automata model in more detail, before turning our attention to Turing machines.

## FINITE AUTOMATA

The simplest state-machine model is that of a *finite automaton*, which consists of a finite number of states and no additional memory. The machine is in exactly one state at a time, and a computational step involves reading a symbol from the input string and transitioning to the next state (which may be the same state as before) based on what symbol is read. This model and slight variations of it are known by many different names, including *finite automata*, *deterministic finite automata (DFA)*, *finite-state automata*, *finite-state machine*, *finite acceptor*, and others. We will use the term *finite automata* to refer to the model.

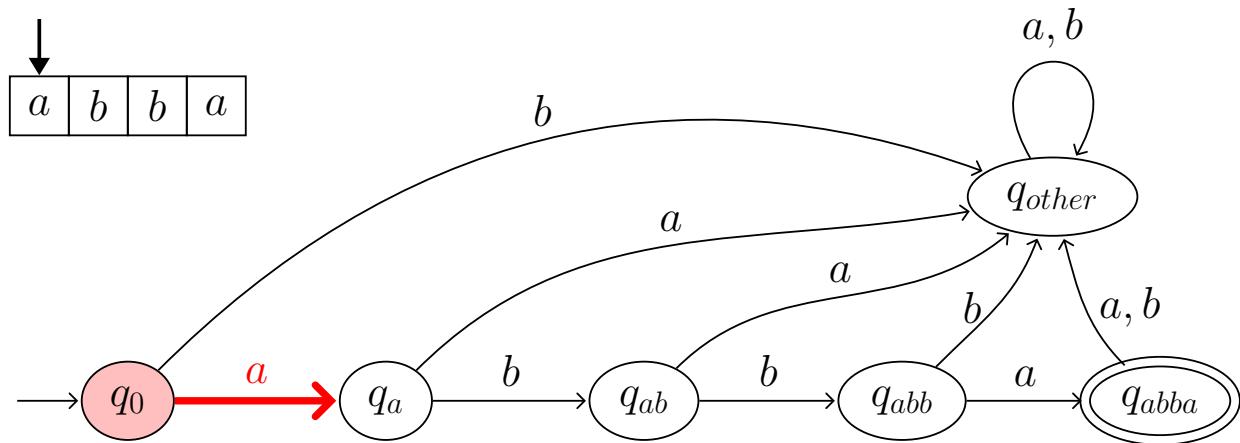
The following is a graphical representation of a finite automaton over the input alphabet  $\Sigma = \{a, b\}$ :



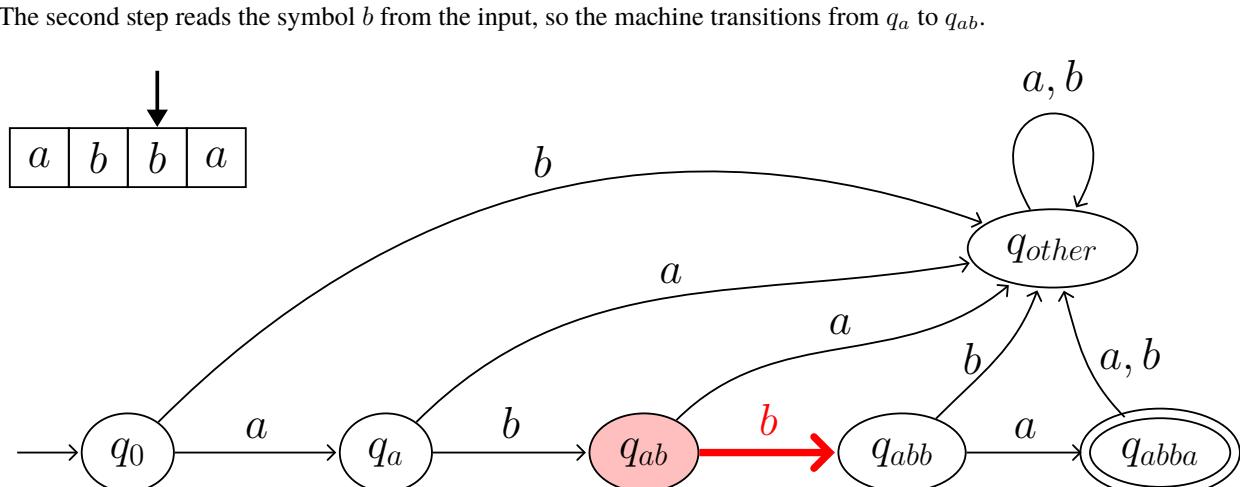
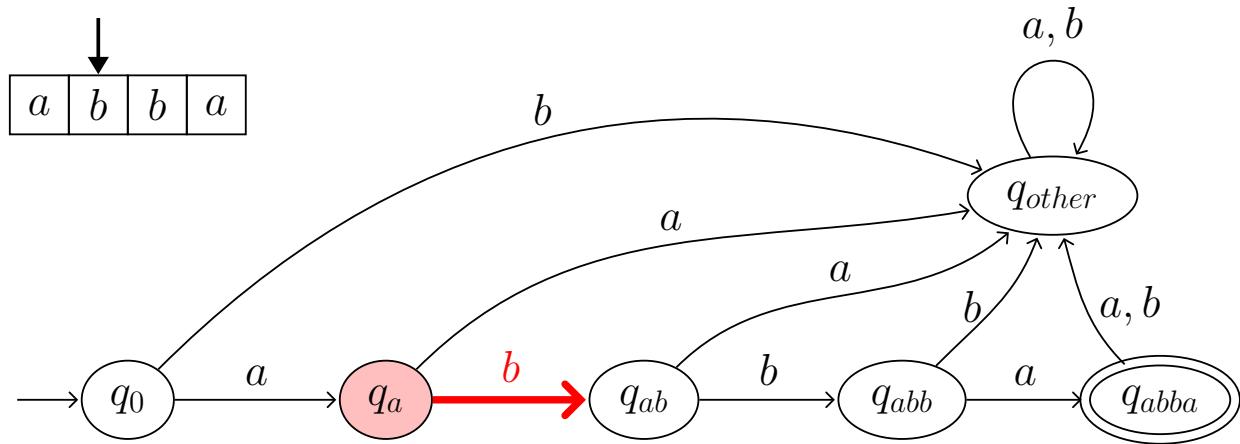
This automaton has six states:  $q_0, q_a, q_{ab}, q_{abb}, q_{abba}, q_{\text{other}}$ . The state  $q_0$  is the special *initial state*, which is the state in which computation begins. Graphically, this is denoted by an incoming edge with no label and no originating vertex. This particular automaton also has a single *accept state*,  $q_{abba}$ , depicted with a double circle. If the computation terminates in this state, the machine accepts the input, otherwise the machine rejects it. Finally, *transitions* between states are depicted as directed edges, with labels corresponding to the input symbol(s) that trigger the transition.

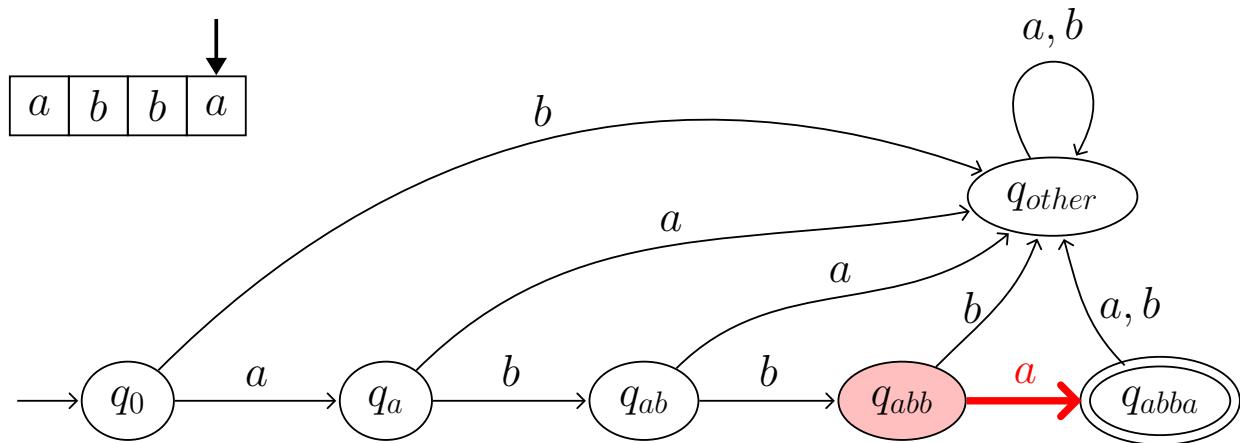
A finite automaton runs on an input string, performing a state transition for each symbol of the string, in sequence. The machine does a single pass over the input, and the computation terminates when (and only when) the entire input has been read. The result is acceptance if the final state is an accept state, and rejection otherwise.

As an example, we trace the execution of the above finite automaton on the input string  $abba$ . In the diagrams below, the top-left shows how much of the input has been read, and the current state is represented by a shaded vertex. Initially, the machine is in the start state  $q_0$ , and none of the input string has been read yet:

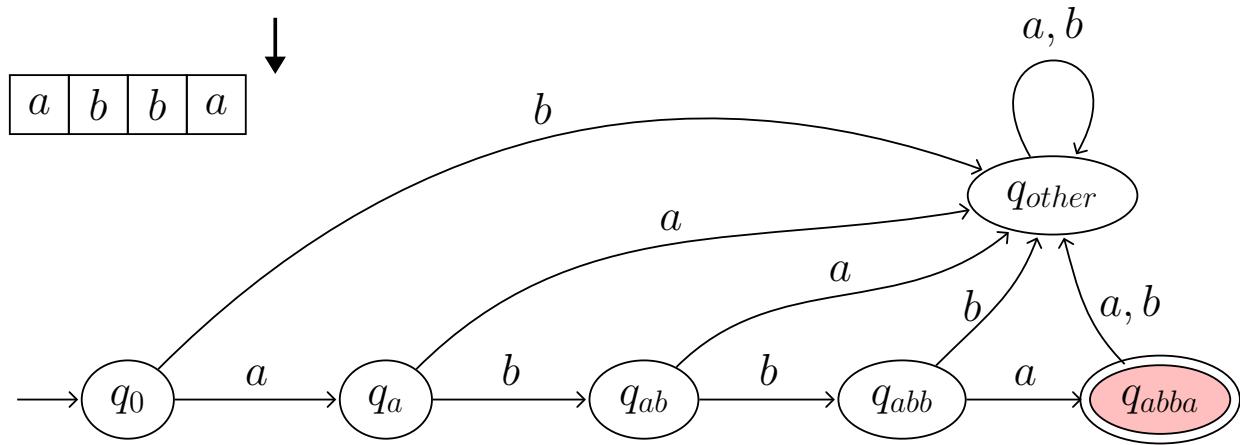


The first input symbol is an  $a$ , so the machine makes the transition labeled by an  $a$  from the state  $q_0$ , which goes to the state  $q_a$ .



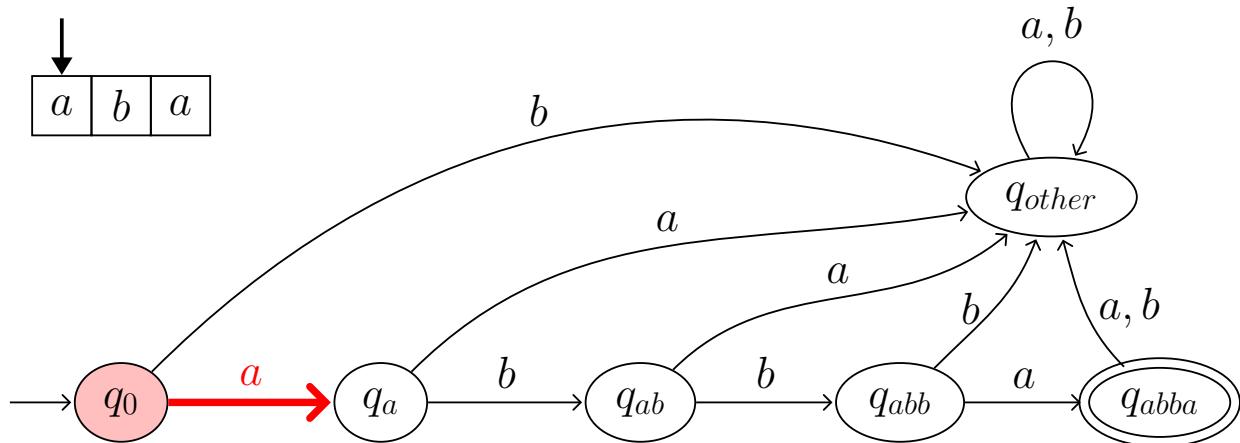


The fourth step reads an  $a$ , causing a transition from  $q_{abb}$  to  $q_{abba}$ .

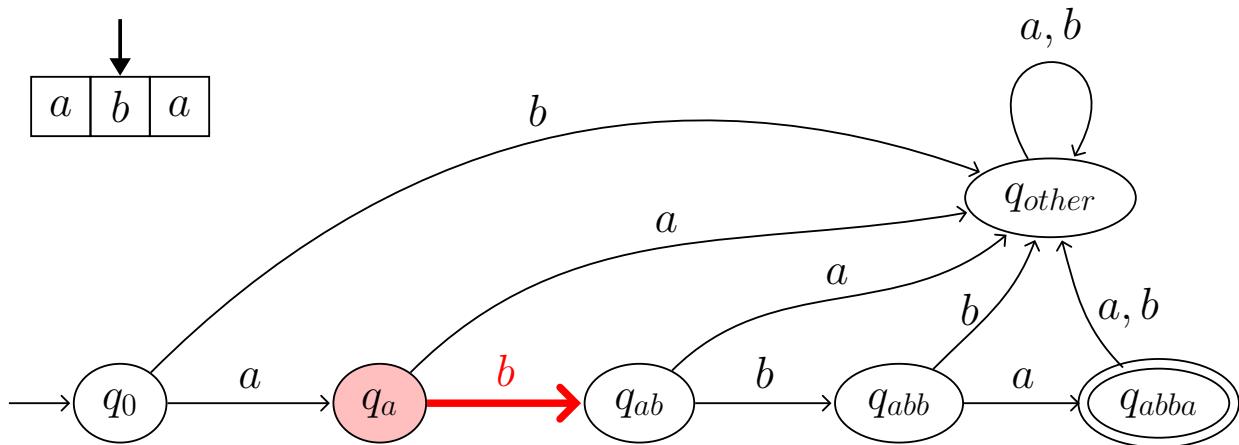


The entire input has now been read, so the computation is complete. The machine terminates in the state  $q_{abba}$ ; since this is an accepting state, the machine has accepted the input  $abba$ .

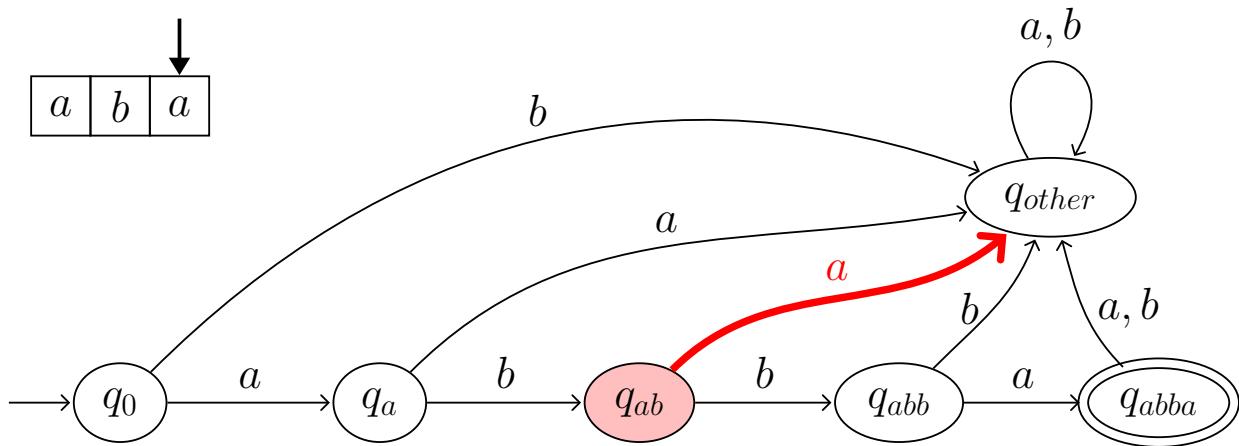
The following is an example of running the machine on input  $aba$ :



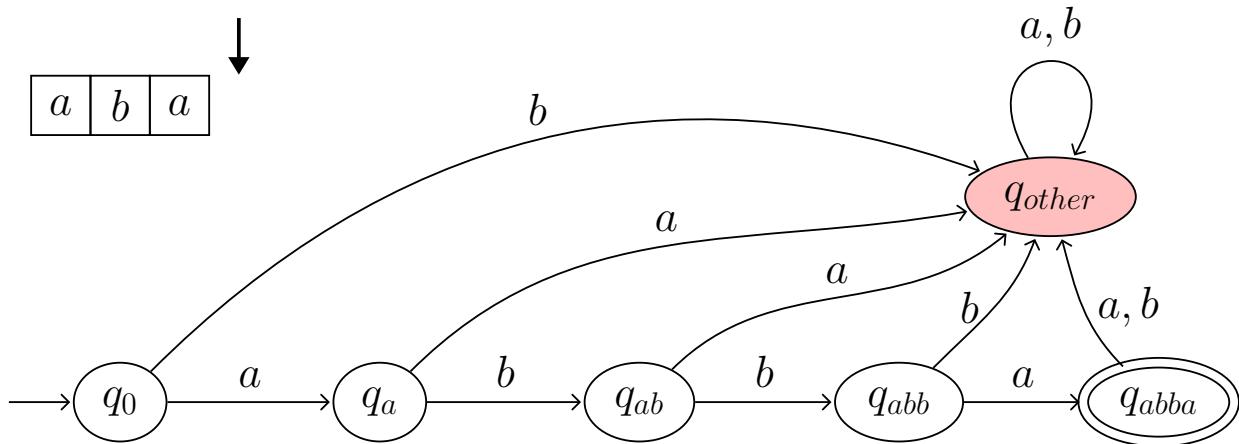
As before, the first symbol is an  $a$ , so the machine transitions from the start state  $q_0$  to the state  $q_a$ .



The second symbol is again a  $b$ , so the machine goes from state  $q_a$  to  $q_{ab}$ .



The third step reads the symbol  $a$ , so the machine transitions from  $q_{ab}$  to  $q_{other}$ .



The machine has processed the entire input, so the computation is complete and the machine terminates. Since the final state  $q_{other}$  is not an accepting state, the machine has rejected the input  $aba$ .

For this automaton, the only string of input symbols that leads to termination in an accepting state is  $abba$ , so this finite

automaton decides the language

$$L = \{abba\}$$

consisting of only the string *abba*. Note that an input like *abbab* causes the automaton to pass through the accepting state  $q_{abba}$  during the computation, but it ultimately terminates in the non-accept state  $q_{\text{other}}$ , so the machine *rejects* the input *abbab*.

## 7.1 Formal Definition

**Definition 55 (Finite automaton)** A finite automaton is a five-tuple

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

where:

- $\Sigma$  is the (finite) input alphabet; an input to the automaton is any (finite) string over this alphabet (i.e., an element of  $\Sigma^*$ ).
- $Q$  is the *finite* set of the automaton's states.
- $q_0 \in Q$  is the initial state.
- $F \subseteq Q$  is the subset of accepting states. If, after reading the entire input string, the machine terminates in one of these states, it is said to *accept* the string; otherwise, it is said to *reject* the string.
- $\delta$  is the *transition function*, which maps a state and input symbol to a next state:

$$\delta: Q \times \Sigma \rightarrow Q .$$

(Recall that  $Q \times \Sigma$  is the set of all pairs whose first component is a state in  $Q$  and whose second component is a symbol in  $\Sigma$ .) This function defines the automaton's transitions as follows: if it is in state  $q \in Q$  and reads the next input symbol  $\sigma \in \Sigma$ , then it transitions to the next state  $\delta(q, \sigma) \in Q$ .

We emphasize that  $\delta$  must be a *function*, i.e., for every state-symbol pair, there must be exactly one next state.

We often depict the states visually as vertices of a graph, and we depict the state transitions as directed edges from states to new states, labeled by the corresponding symbols. Specifically, if  $\delta(q, \sigma) = q'$ , then we have an edge from  $q$  to  $q'$ , labeled by  $\sigma$ . (If there are multiple transitions from  $q$  to  $q'$  for different symbols, we usually use the same edge for all of them, and label it by a comma-separated list of all the corresponding symbols.) Because  $\delta$  is a function, this means that for each state, there must be exactly one outgoing edge for each alphabet symbol.

Let us now see the formal definition of the example automaton given above.

- The input alphabet is  $\Sigma = \{a, b\}$ .
- The set of states is

$$Q = \{q_0, q_a, q_{ab}, q_{abb}, q_{abba}, q_{\text{other}}\} .$$

- The initial state is the one we named  $q_0$ .
- The subset  $F$  of accepting states consists of the single state  $q_{abba}$ , i.e.,  $F = \{q_{abba}\}$ .

- The transition function can be specified in list form, as:

$$\begin{aligned}
 \delta(q_0, a) &= q_a \\
 \delta(q_0, b) &= q_{\text{other}} \\
 \delta(q_a, a) &= q_{\text{other}} \\
 \delta(q_a, b) &= q_{ab} \\
 \delta(q_{ab}, a) &= q_{\text{other}} \\
 \delta(q_{ab}, b) &= q_{abb} \\
 \delta(q_{abb}, a) &= q_{abba} \\
 \delta(q_{abb}, b) &= q_{\text{other}} \\
 \delta(q_{abba}, a) &= q_{\text{other}} \\
 \delta(q_{abba}, b) &= q_{\text{other}} \\
 \delta(q_{\text{other}}, a) &= q_{\text{other}} \\
 \delta(q_{\text{other}}, b) &= q_{\text{other}} .
 \end{aligned}$$

Alternatively, it can be given in tabular form:

old state $q$	$\delta(q, a)$	$\delta(q, b)$
$q_0$	$q_a$	$q_{\text{other}}$
$q_a$	$q_{\text{other}}$	$q_{ab}$
$q_{ab}$	$q_{\text{other}}$	$q_{abb}$
$q_{abb}$	$q_{abba}$	$q_{\text{other}}$
$q_{abba}$	$q_{\text{other}}$	$q_{\text{other}}$
$q_{\text{other}}$	$q_{\text{other}}$	$q_{\text{other}}$

In either case, notice that for each state-symbol pair, there is exactly one corresponding next state.

**Definition 56 (Language of a finite automaton)** The *language of a finite automaton  $M$*  is the set of strings that the automaton accepts:  $L(M) = \{x \in \Sigma^* : M \text{ accepts } x\}$ . We also say that  $M$  *decides* this language: it accepts every string in  $L(M)$  and rejects every string not in  $L(M)$ .

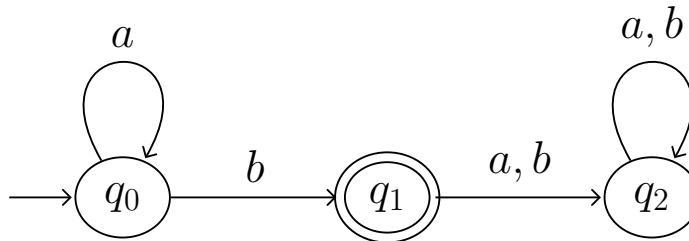
For the example automaton  $M$  above, we have

$$L(M) = \{abba\} .$$

**Example 57** Consider the finite automaton  $M = (Q, \Sigma, \delta, q_0, F)$  where:

$$\begin{aligned}
 Q &= \{q_0, q_1, q_2\} \\
 \Sigma &= \{a, b\} \\
 \delta(q_0, a) &= q_0 \\
 \delta(q_0, b) &= q_1 \\
 \delta(q_1, a) &= q_2 \\
 \delta(q_1, b) &= q_2 \\
 \delta(q_2, a) &= q_2 \\
 \delta(q_2, b) &= q_2 \\
 F &= \{q_1\}
 \end{aligned}$$

This machine has three states, of which  $q_1$  is the lone accept state. The following is a graphical representation:



The machine starts in state  $q_0$  and stays there as long as it reads just  $a$  symbols. If it then reads a  $b$ , the machine transitions to  $q_1$ . Any subsequent symbol moves the machine to  $q_2$ , where it stays until it reads the rest of the input. Since  $q_1$  is the only accepting state, the machine accepts only strings that end with a single  $b$ . Thus, the language of the machine is

$$L(M) = \{b, ab, aab, aaab, \dots\},$$

i.e., the set of strings that start with any number of  $a$ s, followed by a single  $b$ .

Finite automata have many applications, including software for designing and checking digital circuits, lexical analyzers of most compilers (usually the first phase of compilation, which splits a source file into individual words, or “tokens”), scanning large bodies of text (pattern matching), and reasoning about many kinds of systems that have a finite number of states (e.g., financial transactions, network protocols, and so on).

However, finite automata are not strong enough to model all possible computations. For instance, no finite automaton decides the language

$$L = \{0^n 1^n : n \in \mathbb{N}\} = \{\varepsilon, 01, 0011, 000111, \dots\}$$

consisting of strings composed of an arbitrary number of 0s followed by the same number of 1s. The basic intuition for why this is the case is that, since a finite automaton does not have any memory beyond its own states, it cannot reliably count how many 0s or 1s it has seen (beyond a certain fixed number), so it will output the incorrect answer on some input strings. See below for a rigorous proof that formalizes this intuition.

Yet we can easily write a program in most any programming language that decides  $L$ . The following is an example in C++. (To be precise, this program does not quite work, because if the input starts with enough 0s, the `count` variable, which has type `int`, will overflow and lose track of the number of 0s that have been read. This is exactly the issue with trying to use a finite automaton to decide  $L$ ; to overcome it, we would need to use a data type that can store arbitrarily large integers.)

```

#include <iostream>

int main() {
    int count = 0;
    char ch;
    while ((std::cin >> ch) && ch == '0') {
        ++count;
    }
    if (!std::cin || ch != '1') {
        return !std::cin && count == 0;
    }
    --count; // already read one 1
    while ((std::cin >> ch) && ch == '1') {
        --count;
    }
    return !std::cin && count == 0;
}
  
```

Thus, we need a more powerful model of computation to characterize the capabilities of real-world programs.

### Proof that no DFA decides $L = \{0^n 1^n : n \in \mathbb{N}\}$

We prove that no DFA decides the language  $L = \{0^n 1^n : n \in \mathbb{N}\}$ . Let  $M$  be an arbitrary DFA  $M$ ; we will show that  $M$  does not decide  $L$  by exhibiting an input on which  $M$  returns the wrong output, i.e.,  $M$  rejects a string in  $L$ , or accepts a string not in  $L$ .

Let  $s$  be the number of states of  $M$ , which is finite by definition of a DFA. Consider the execution of  $M$  on an input that begins with  $0^{s+1}$ . Since  $M$  has only  $s$  states, by the pigeonhole principle, there must be some state  $q \in Q$  that is *repeated* while  $M$  processes the string. Specifically, there exist some *distinct*  $0 \leq i < j \leq s + 1$  such that  $M$  is in state  $q$  after having read  $0^i$ , and also after having read  $0^j$ .

Now, consider running  $M$  on the strings  $0^i 1^i \in L$  and  $0^j 1^i \notin L$  (where the latter holds because  $i \neq j$ ). Notice that  $M$  must output the *same* decision on both of these strings, because it is in the same state  $q$  after reading the initial  $i$  or  $j$  zeros, and because both strings have  $1^i$  after those zeros. However, one of these strings is in  $L$  and the other is not, so  $M$  outputs the wrong decision on one of them, and therefore does not decide  $L$ , i.e.  $L(M) \neq L$ , as claimed.

The reasoning above is an particular example of applying the [pumping lemma<sup>19</sup>](#). If a language is regular (i.e., can be decided by a DFA), then every sufficiently long string in the language has some section  $w$  that can be repeated arbitrarily, or “*pumped*”, to produce another string that is also a member of the language. The pumping lemma can be used to prove that a language is non-regular, as we did above for  $L = \{0^n 1^n : n \in \mathbb{N}\}$ .

---

<sup>19</sup> [https://en.wikipedia.org/wiki/Pumping\\_lemma\\_for\\_regular\\_languages](https://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages)

---

CHAPTER  
EIGHT

---

## TURING MACHINES

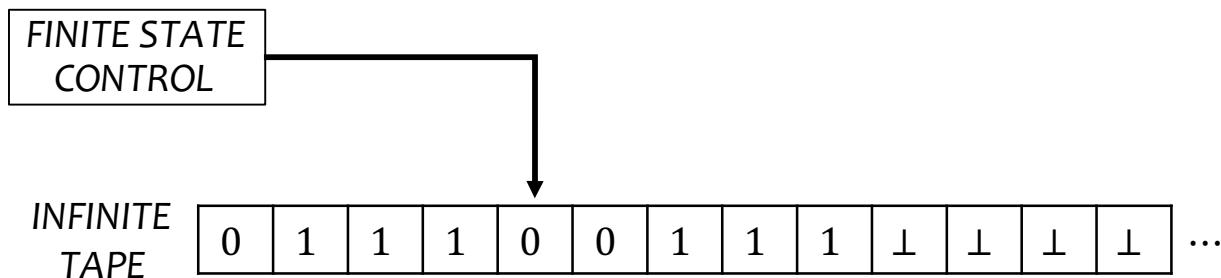
A *Turing machine* is a powerful abstract model for a computer, proposed by Alan Turing in 1936. The abstraction was designed to model pencil-and-paper computations by a human or other (possibly mechanical) “brain” that is following some prescribed set of rules. The essential features of the model are as follows:

- Each sheet of paper can hold a finite number of symbols from some finite alphabet.
- The amount of paper is unlimited – if we run out of space, we can just buy more paper.
- The “brain” can look at and modify only one symbol on the paper at a time.
- Though there may be an arbitrary amount of data stored on the paper, the “brain” can retain only a fixed amount at any time.

We model these features more specifically as follows, though we emphasize that many of these choices are fairly arbitrary, and can be adjusted without affecting the power of the model. (See *Equivalent Models* (page 89) for an example.)

- The paper is represented by an infinite *tape* that is divided into individual *cells*, each of which holds one symbol at a time (we view “blank” as a symbol of its own). For concreteness, in this text we use a “uni-directionally infinite” tape that has a leftmost cell and continues indefinitely to the right. (One can also consider a “bi-directionally infinite” tape without affecting the strength of the model.)
- The tape has a read/write *head* that is positioned over a single cell at a time, and it can read the contents of the cell, write a new symbol to that cell, and move one position to the left or right.
- Finally, the brain is modeled using a kind of finite-state machine: there are a finite set of *states*, with each state corresponding to what the brain “holds” at a specific time. Except for the states that terminate the machine, each state uses the symbol read by the head to determine what symbol to write (at the same cell), which direction to move the head, and what the next state will be.

The following is a pictorial representation of this model.



We formalize the notion of a Turing machine as follows.

**Definition 58 (Turing machine)**

A Turing machine is a seven-tuple

$$M = (\Sigma, \Gamma, Q, q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}, \delta)$$

whose components are as follows:

- $\Sigma$  is the (finite) *input alphabet*. An input to the machine is any (finite) string over this alphabet (i.e., an element of  $\Sigma^*$ ).
- $\Gamma$  is the (finite) *tape alphabet*, which must contain  $\Sigma$  as a subset (i.e.,  $\Sigma \subseteq \Gamma$ ), in addition to a special *blank symbol*  $\perp \in \Gamma$  that is not in the input alphabet (i.e.,  $\perp \notin \Sigma$ ). It may contain other elements as well. At all times, every symbol on the tape is an element of  $\Gamma$ .
- $Q$  is the *finite set of states*.
- $q_{\text{start}} \in Q$  is the *initial state*.
- $q_{\text{acc}}, q_{\text{rej}} \in Q$  are the (distinct) *accept state* and *reject state*, respectively. Together, they comprise the set of *final states*  $F = \{q_{\text{acc}}, q_{\text{rej}}\}$ .
- $\delta$  is the *transition function*. Its input is a non-final state and a tape symbol (as read by the head), and its output is a new state, a new tape symbol (to be written by the head), and a direction for the head to move (left or right). Formally, we have

$$\delta: (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

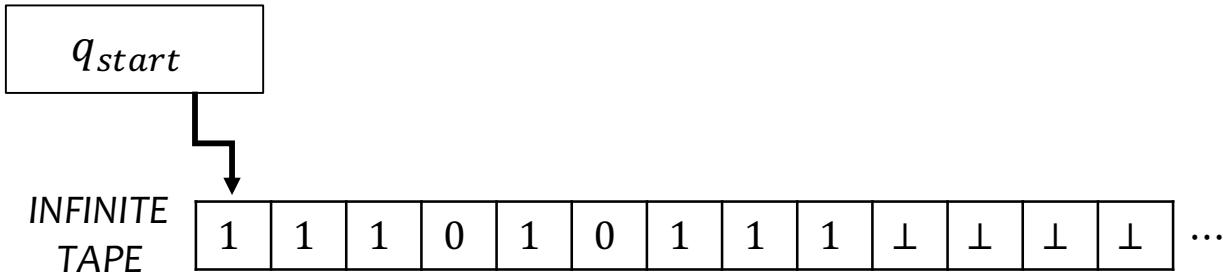
where  $L$  and  $R$  represent “left” and “right,” respectively. The meanings of the components of the transition function’s domain and codomain are as follows:

$$\delta: \underbrace{(Q \setminus F)}_{\text{Non-terminal state}} \times \underbrace{\Gamma}_{\text{Cell contents}} \rightarrow \underbrace{Q}_{\text{Next state}} \times \underbrace{\Gamma}_{\text{Symbol to write}} \times \underbrace{\{L, R\}}_{\text{Where to move the head}}$$

We now describe the process by which a Turing machine computes when given an *input string*, which can be any (finite) string over the input alphabet  $\Sigma$ . First, the machine is initialized as follows.

- Tape contents: the input string appears written on the tape from left to right, starting at the leftmost cell, with one symbol per cell. Every other cell contains the blank symbol  $\perp$ .
- The head is positioned at the leftmost cell of the tape.
- The initial “active” state is  $q_{\text{start}}$ .

The following illustrates the initial configuration of a machine when the input is the binary string 111010111:



A *computational step* consists of an application of the transition function to the current configuration of a machine—i.e., its active state, tape contents, and head position—to produce a new configuration. The machine’s active state must be a non-final state – otherwise, the machine has *halted* and no further computation is done. To perform a computational step from some non-final active state  $q \in Q \setminus F$ , the machine:

1. reads the symbol  $s \in \Gamma$  at the tape cell at which the head is positioned;

2. evaluates the transition function to get  $(q' \in Q, s' \in \Gamma, D \in \{L, R\}) = \delta(q, s)$ ;
3. makes  $q' \in Q$  the new active state;
4. writes  $s'$  to the cell at which the head is positioned;
5. moves the head left or right according to the value of  $D$ .

We stress that the transition function completely determines what happens in steps 3-5, based entirely on the active state and the symbol that is read by the head. That is, given the current state  $q$  and tape symbol  $s \in \Gamma$ , the output  $(q', s', d) = \delta(q, s)$  of the transition function gives the new active state  $q'$  (which may be the same as the original state  $q$ ), the symbol  $s'$  to write (which may be the same as the original symbol  $s$ ), and the direction in which to move the head (left or right). By convention, if the head is at the leftmost cell of the tape and the direction is left, then the head stays in the same (leftmost) position.

A Turing machine computes by repeatedly executing the above computational step, halting only when it transitions to one of the two final states  $q_{\text{acc}}$  or  $q_{\text{rej}}$ . If the machine ever reaches the state  $q_{\text{acc}}$ , the machine is said to *accept* the input. If the machine ever reaches  $q_{\text{rej}}$ , it is said to *reject* the input. As we will discuss more later, there is a third possibility: that the machine never reaches a final state at all, and continues applying the computational step indefinitely. In this case, we say that the machine *loops forever*, or just *loops*, on the input.

The main differences between finite automata and Turing machines are summarized as follows:

Property	Finite Automata	Turing Machines
Has an (unbounded) read/write memory (tape)	No	Yes
Can have zero, or more than one, accept state(s)	Yes	No
Has a reject state	No	Yes
Terminates when accept/reject state is reached	No	Yes
Direction input is read	Right	Right or Left
Must terminate when entire input is read	Yes	No

As an example of a Turing machine, here we define a simple machine that accepts any binary string composed entirely of ones (including the empty string), and rejects any string that contains a zero. The components of the seven-tuple are as follows:

- $\Sigma = \{0, 1\}$ ,
- $\Gamma = \{0, 1, \perp\}$ ,
- $Q = \{q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}\}$ ,
- the transition function  $\delta: \{q_{\text{start}}\} \times \{0, 1, \perp\} \rightarrow \{q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}\} \times \{0, 1, \perp\} \times \{L, R\}$  is specified by listing its output on each input:

$$\begin{aligned}\delta(q_{\text{start}}, 0) &= (q_{\text{rej}}, 0, R) \\ \delta(q_{\text{start}}, 1) &= (q_{\text{start}}, 1, R) \\ \delta(q_{\text{start}}, \perp) &= (q_{\text{acc}}, \perp, R).\end{aligned}$$

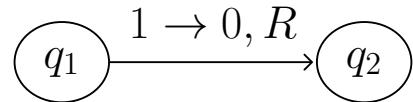
We can also specify the transition function in tabular format:

old state	read symbol	new state	written symbol	direction
$q_{\text{start}}$	0	$q_{\text{rej}}$	0	$R$
$q_{\text{start}}$	1	$q_{\text{start}}$	1	$R$
$q_{\text{start}}$	$\perp$	$q_{\text{acc}}$	$\perp$	$R$

The machine has just three states: the initial state  $q_{\text{start}}$  and the two final states  $q_{\text{acc}}, q_{\text{rej}}$ . In each computational step, it reads the symbol at the tape head, going to the reject state if it is a zero, staying in  $q_{\text{start}}$  if it is a one, and transitioning

to the accept state if it is a  $\perp$ . In all three cases, the machine leaves the contents of the cell unchanged (it writes the same symbol as it reads) and moves the head to the right, to the next input symbol (or to the first blank that appears after the input string).

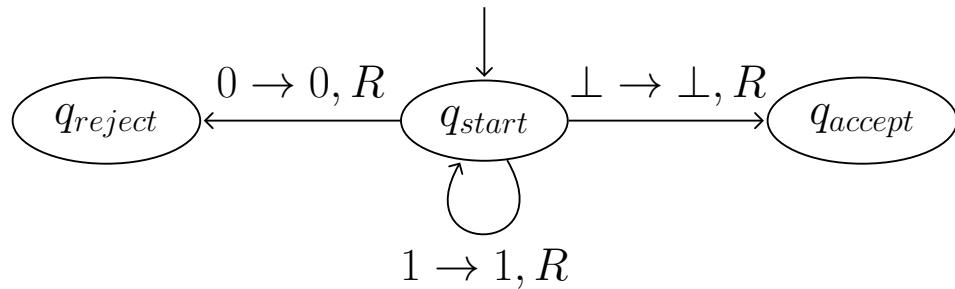
We have described the machine above by explicitly writing out its seven-tuple, including its entire transition function. More commonly, we describe a machine using a *state diagram* instead. Such a diagram is a labeled graph, with a vertex for each state and edges representing transitions between states. The following is an example of a transition:



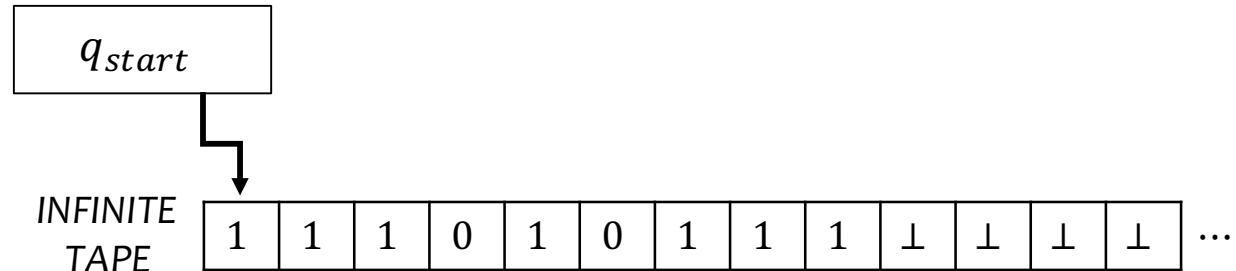
The edge goes from state  $q_1$  to state  $q_2$ , which are the state components of the transition function's input and output. The remaining components are noted on the edge label. The example diagram above means that  $\delta(q_1, 1) = (q_2, 0, R)$ : when the machine is in state  $q_1$  and it reads a 1 at the tape head, it transitions to state  $q_2$ , writes a 0 at the tape head, and moves the head to the right.

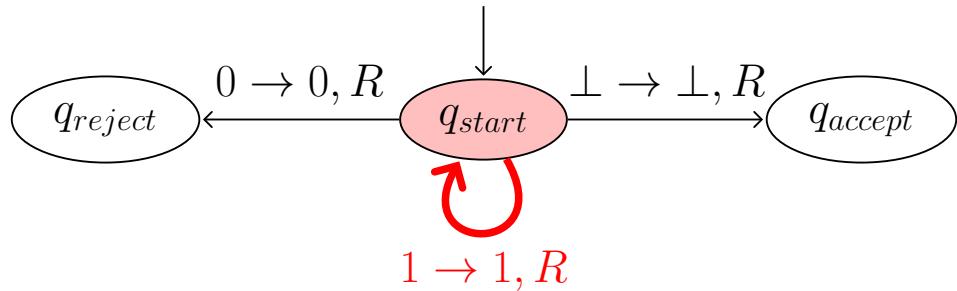
We emphasize that, because the state-transition function must be a *function* on the domain  $(Q \setminus F) \times \Gamma$ , for each non-final state in the diagram there must be exactly one outgoing edge (a transition) for each symbol in the tape alphabet  $\Gamma$ . Otherwise, the state-transition function would either be incomplete (missing an output for some input) or inconsistently defined (having multiple different outputs for the same input). Similarly, there cannot be an outgoing arrow from either of the final states.

The state diagram for the machine we defined above with a seven-tuple is as follows:

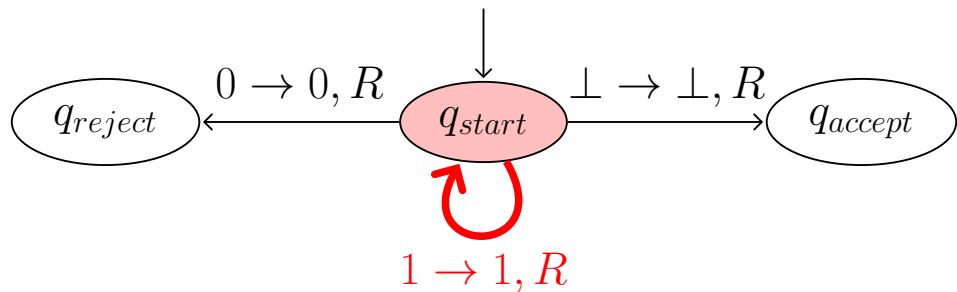
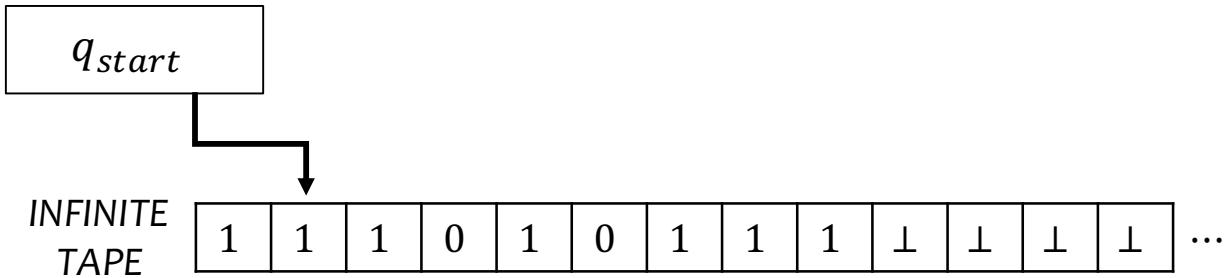


Let's run this machine on the input 111010111. The machine starts in the initial state, with the input written on the tape (followed by blanks) and the head in the leftmost position:

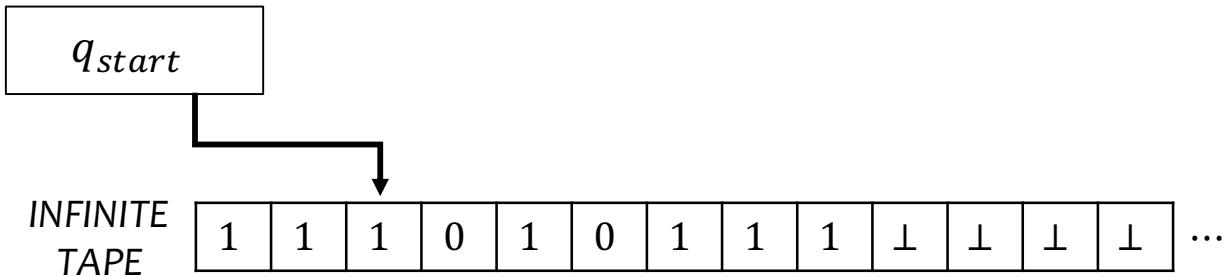


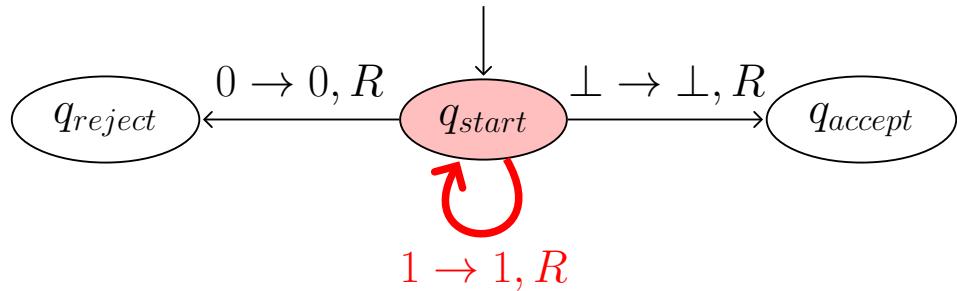


The cell at the head holds a 1. The transition function (looking at either the seven-tuple or the state diagram) tells us that  $\delta(q_{start}, 1) = (q_{start}, 1, R)$ , so the machine stays in the same state, writes the symbol 1 at the head, and moves the head to the right.

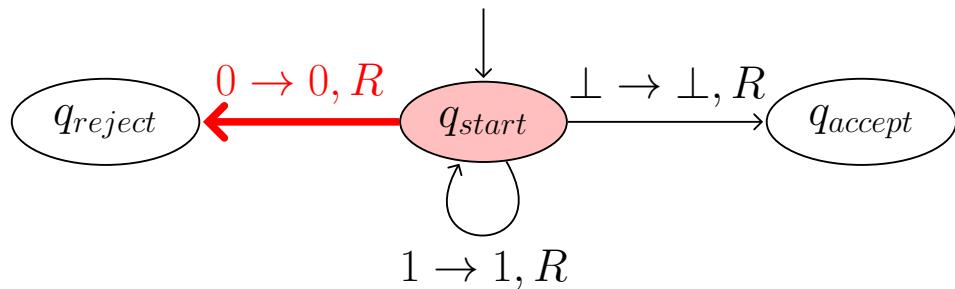
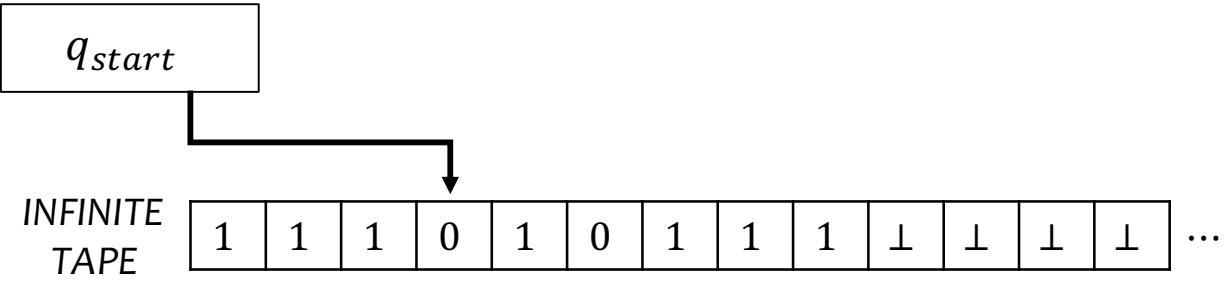


Again, we have a 1 at the head location. Again, the machine stays in the same state, leaves the symbol unchanged, and moves the head to the right.

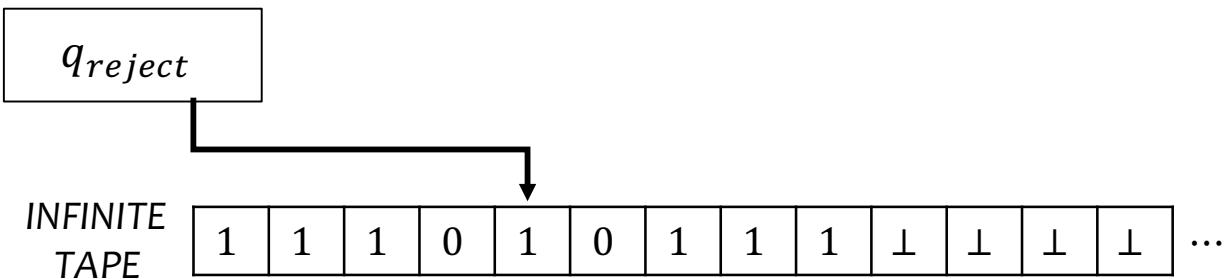


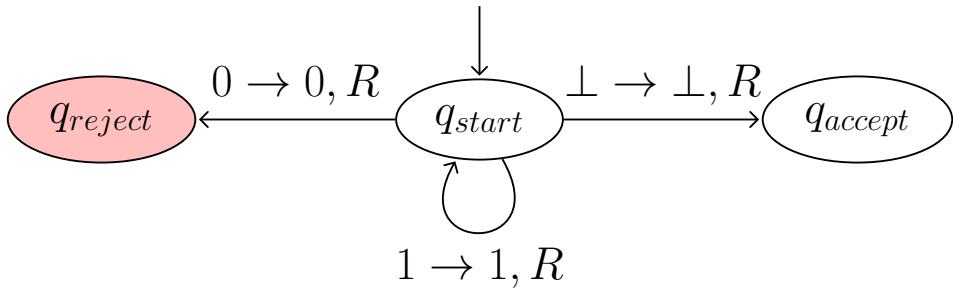


Once again, there is a 1 at the head, resulting in the same actions as in the previous two steps.



The machine now has a 0 at the head. The transition function has that  $\delta(q_{\text{start}}, 0) = (q_{\text{rej}}, 0, R)$ , so the machine transitions to the state  $q_{\text{rej}}$ , leaves the 0 unchanged, and moves the head to the right.



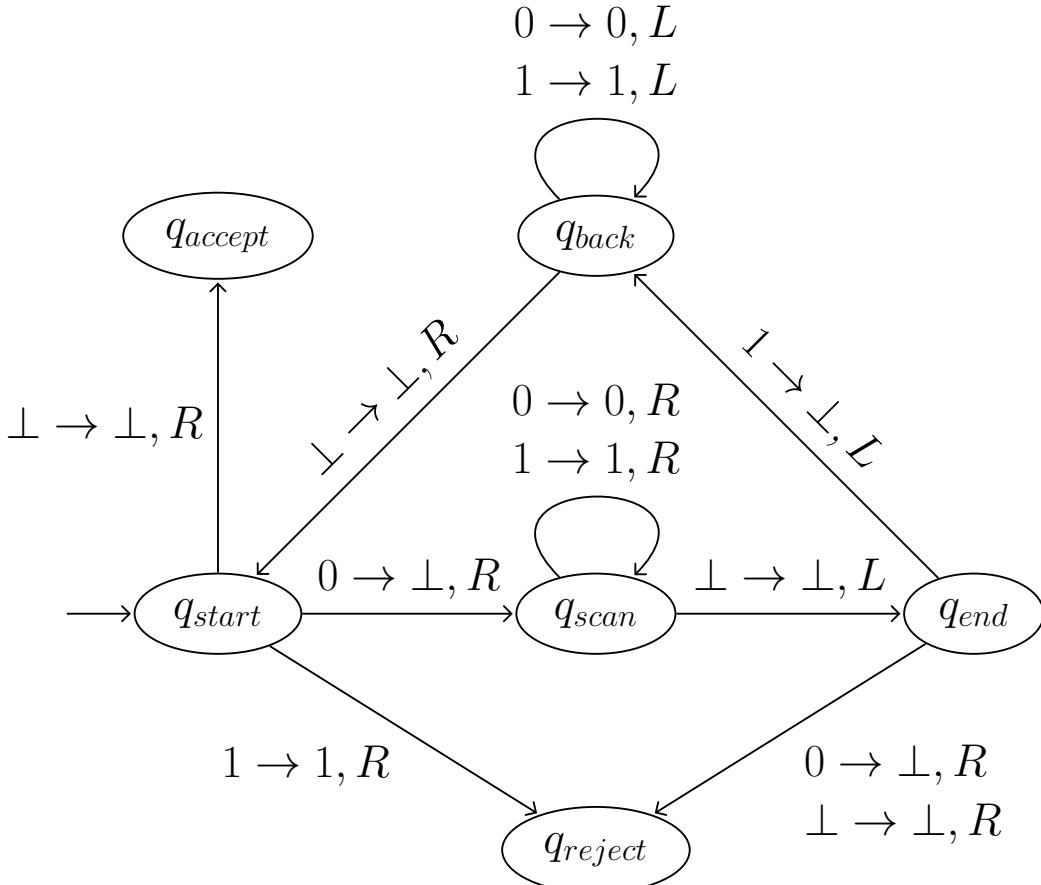


The machine is now in a final state, namely,  $q_{\text{rej}}$ , so the computation halts. Since the machine reached the reject state, it rejects the input 111010111.

Had the input been composed entirely of ones, the machine would have stayed in  $q_{\text{start}}$ , moving the head one cell to the right in each step. After examining all the input symbols, it would reach a cell that contains the blank symbol  $\perp$ . The transition function tells us that the machine would then move to the accept state  $q_{\text{acc}}$  (and leave the blank symbol unchanged, and move the head to the right). Thus, the machine would accept any input consisting entirely of ones. (This includes the empty string, because at startup the tape would consist entirely of blanks, so in the very first computational step the machine would read the blank symbol and transition to the accept state  $q_{\text{acc}}$ .)

On any input string, this machine will eventually reach a final state. (For this conclusion we are relying on the fact that by definition, every string has *finite* length.) In the worst case, it examines each input symbol once, so it runs in linear time with respect to the size of the input. While this machine halts on any input, we will soon see that this is not the case for all machines.

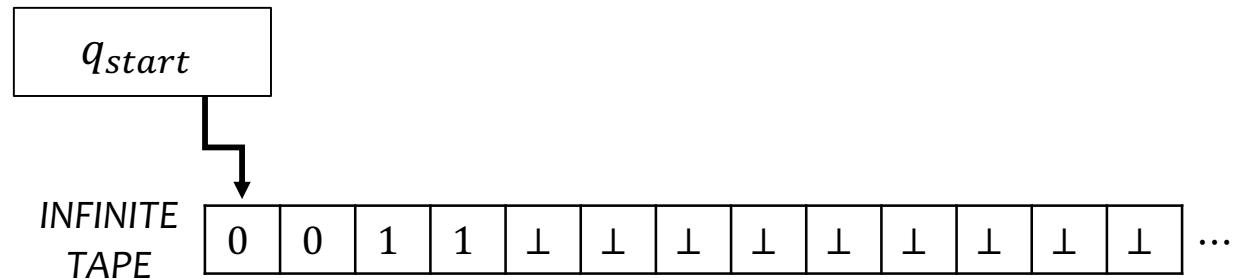
As a second example, consider the following machine:

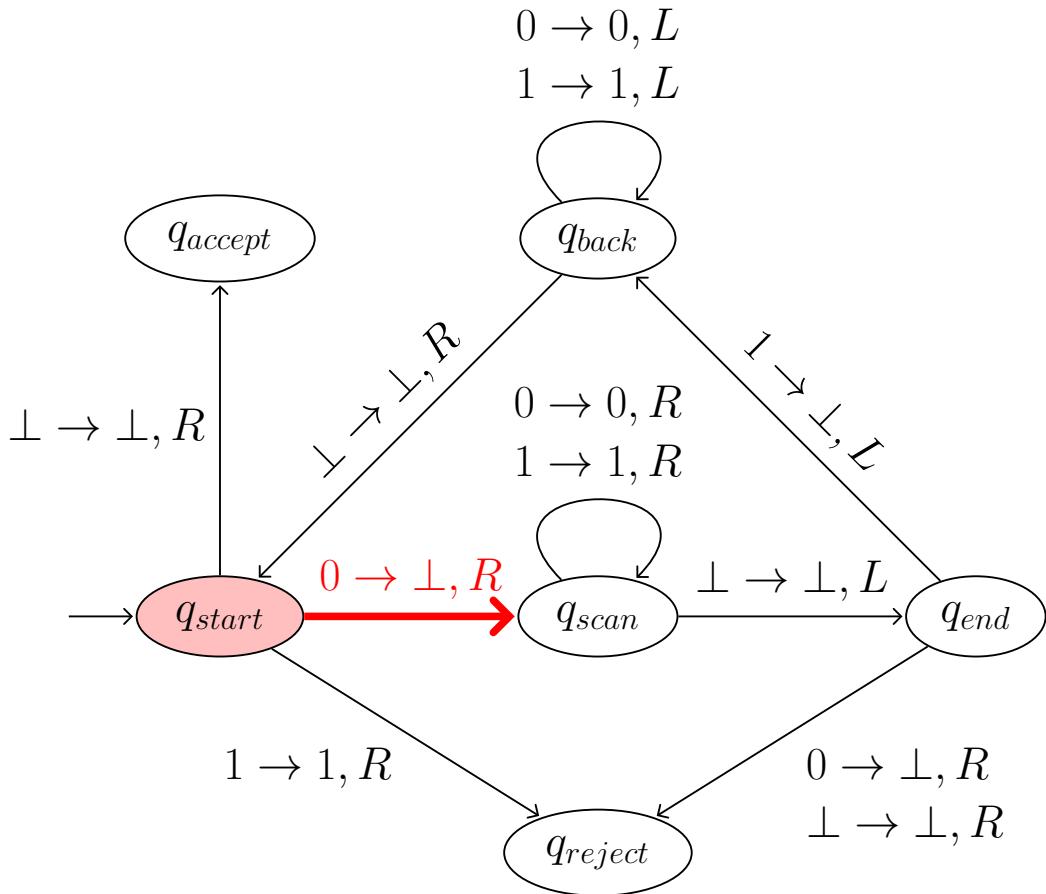


This machine has the input alphabet  $\Sigma = \{0, 1\}$ , the tape alphabet  $\Gamma = \{0, 1, \perp\}$ , six states  $Q = \{q_{\text{start}}, q_{\text{scan}}, q_{\text{end}}, q_{\text{back}}, q_{\text{acc}}, q_{\text{rej}}\}$ , and the following transition function:

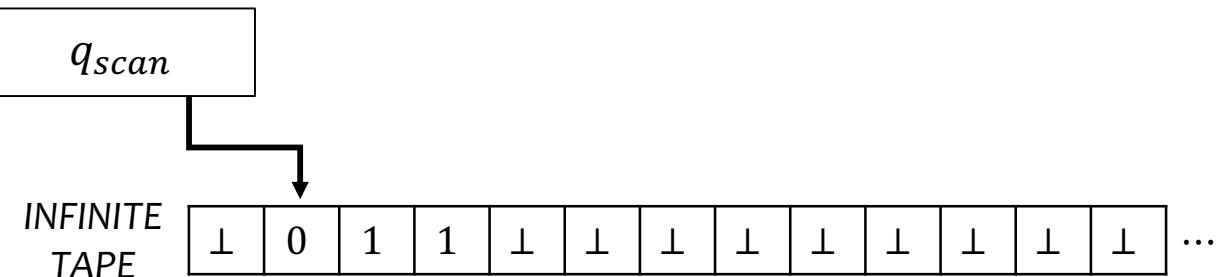
old state	read symbol	new state	written symbol	direction
$q_{\text{start}}$	0	$q_{\text{scan}}$	$\perp$	R
$q_{\text{start}}$	1	$q_{\text{rej}}$	1	R
$q_{\text{start}}$	$\perp$	$q_{\text{acc}}$	$\perp$	R
$q_{\text{scan}}$	0	$q_{\text{scan}}$	0	R
$q_{\text{scan}}$	1	$q_{\text{scan}}$	1	R
$q_{\text{scan}}$	$\perp$	$q_{\text{end}}$	$\perp$	L
$q_{\text{end}}$	0	$q_{\text{rej}}$	0	R
$q_{\text{end}}$	1	$q_{\text{back}}$	$\perp$	L
$q_{\text{end}}$	$\perp$	$q_{\text{rej}}$	$\perp$	R
$q_{\text{back}}$	0	$q_{\text{back}}$	0	L
$q_{\text{back}}$	1	$q_{\text{back}}$	1	L
$q_{\text{back}}$	$\perp$	$q_{\text{start}}$	$\perp$	R

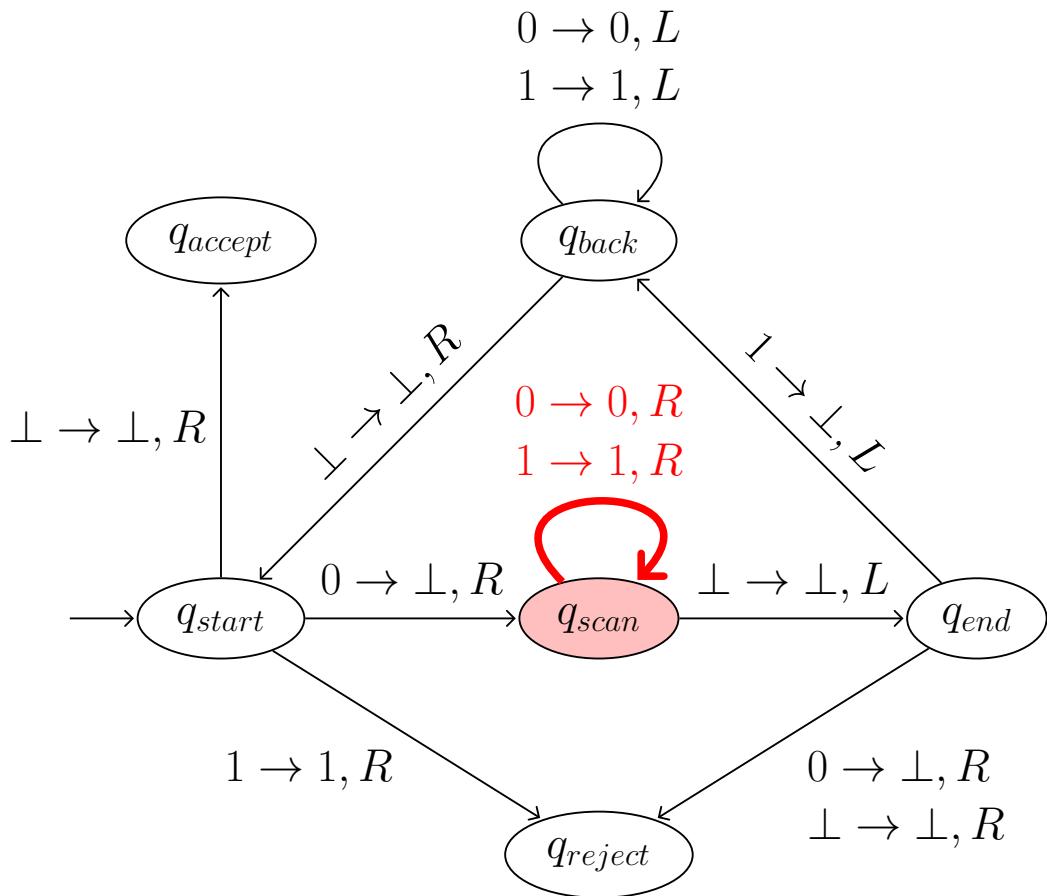
Let us trace the execution of this machine on the input 0011. The machine starts in the initial state, with the input written on the tape (followed by blanks) and the head in the leftmost position:



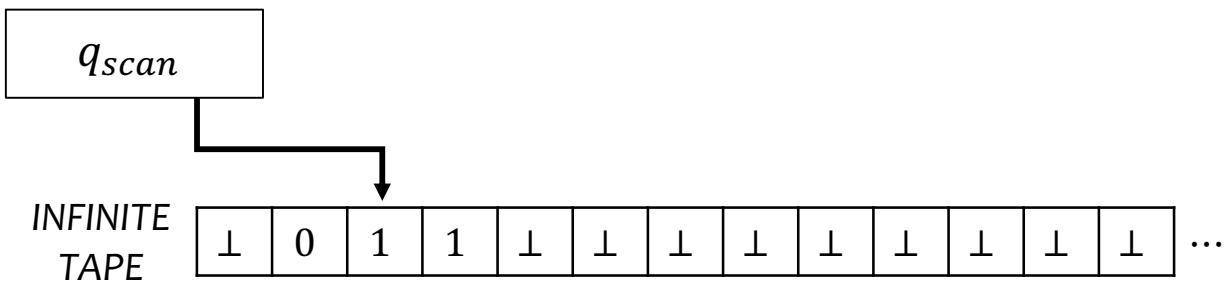


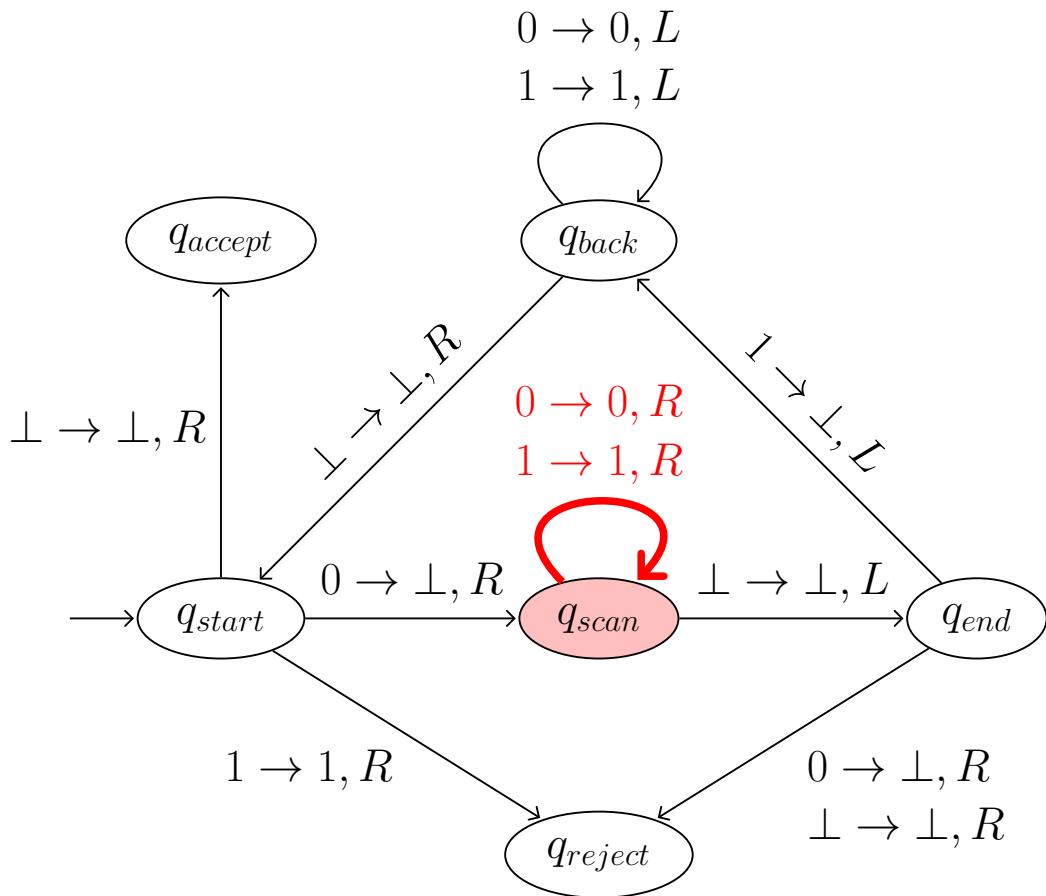
The cell at the head has a zero. The transition function tells us that  $\delta(q_{\text{start}}, 0) = (q_{\text{scan}}, \perp, R)$ , so the machine transitions to state  $q_{\text{scan}}$ , writes a blank symbol at the head, and moves the head to the right.



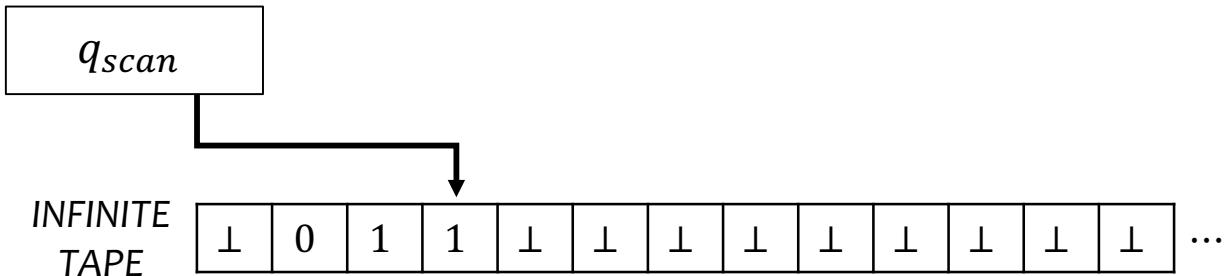


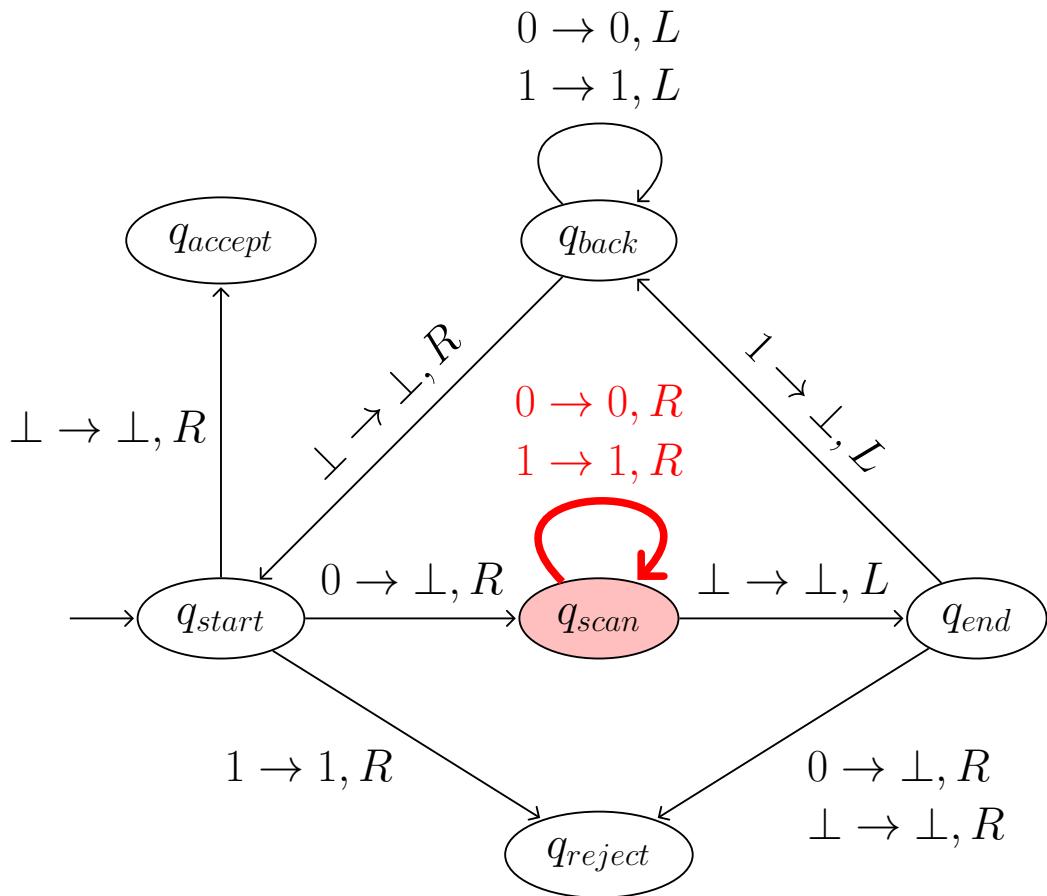
The cell at the head now has a zero. The transition function tells us that  $\delta(q_{scan}, 0) = (q_{scan}, 0, R)$ , so the machine stays in  $q_{scan}$ , leaves the symbol 0 under the head, and moves the head to the right.



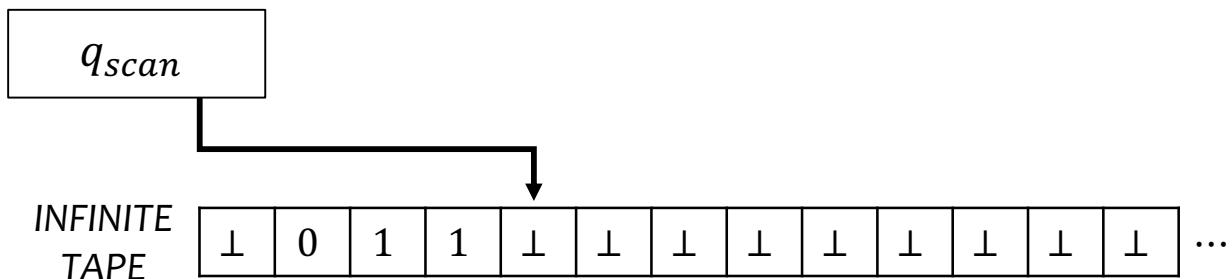


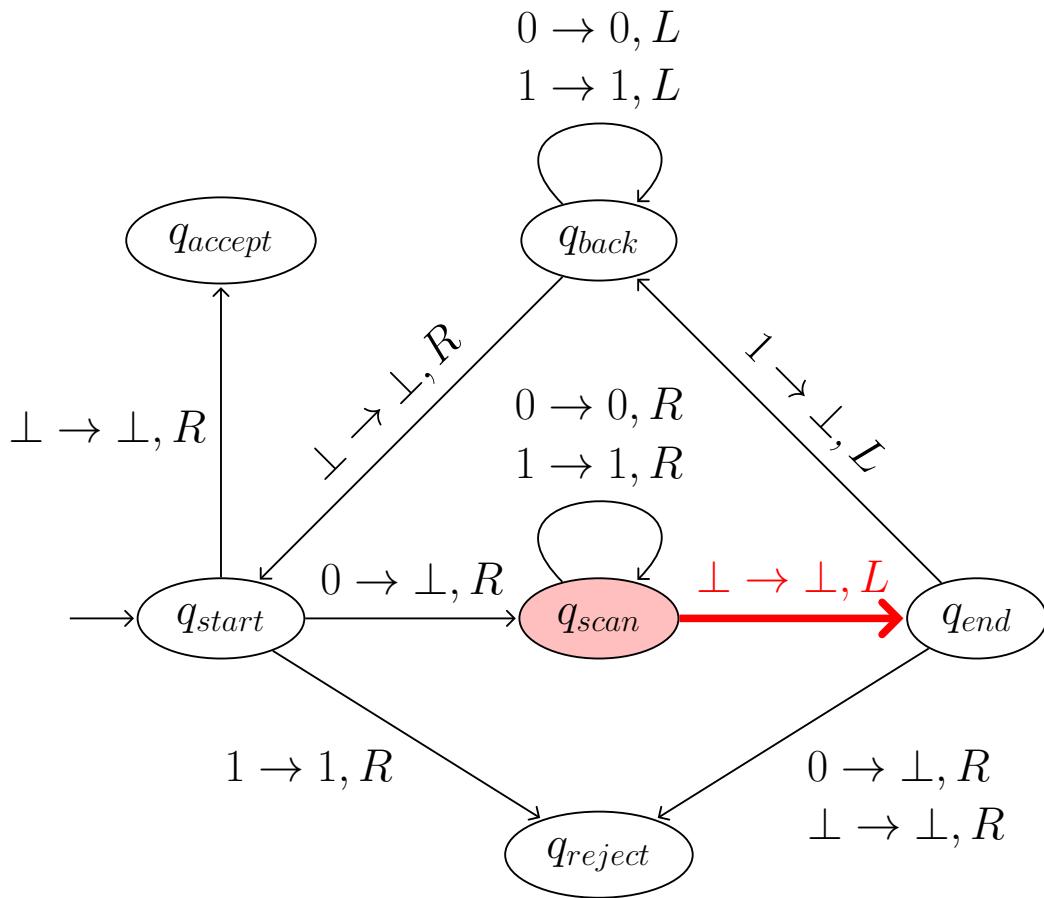
The cell at the head now has a one. The transition function tells us that  $\delta(q_{scan}, 1) = (q_{scan}, 1, R)$ , so the machine stays in  $q_{scan}$ , leaves the symbol 1 under the head, and moves the head to the right.



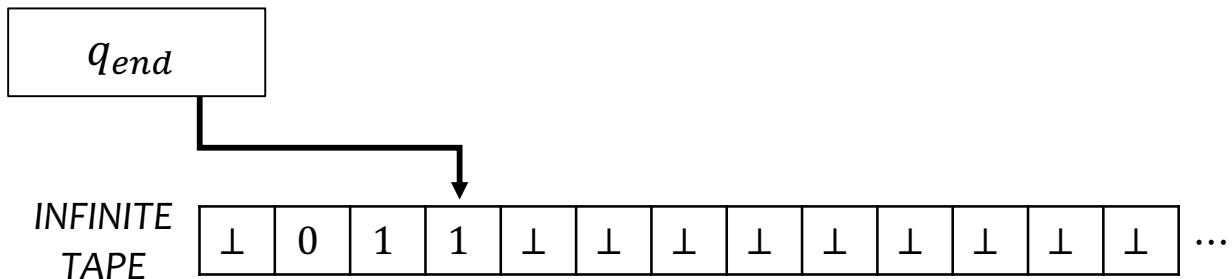


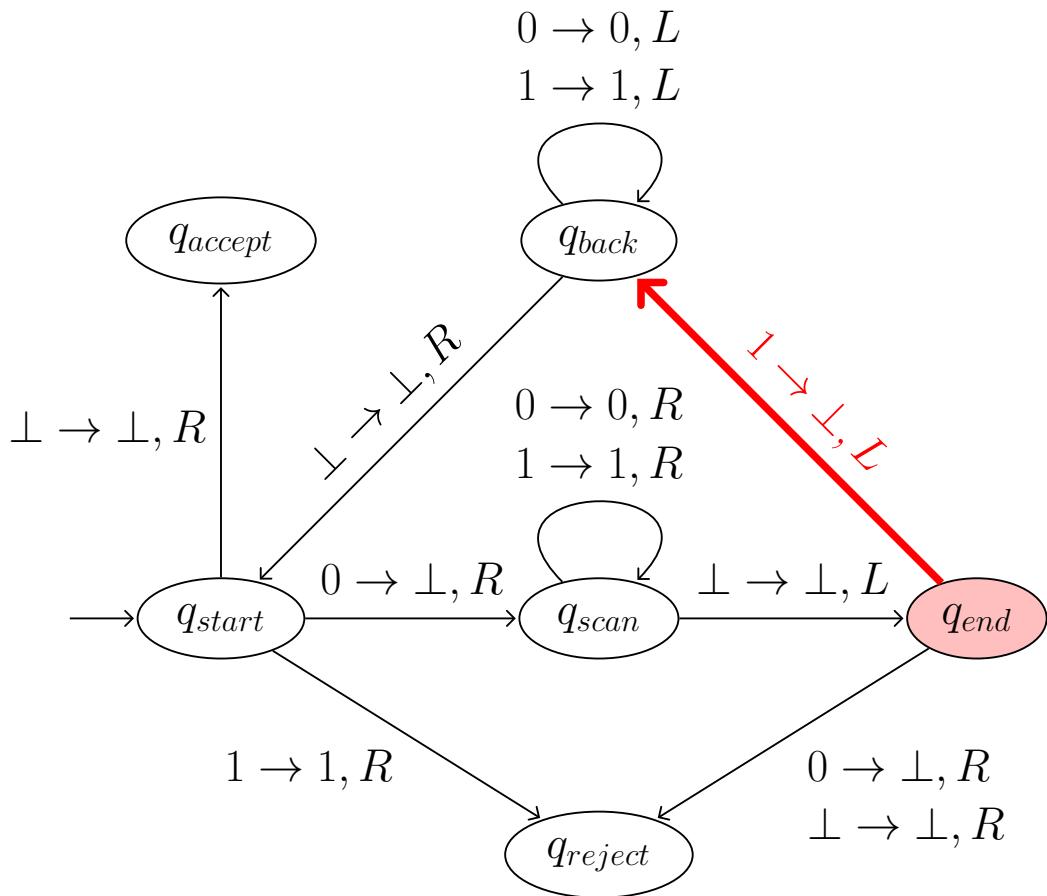
Again, the cell at the head now has a one, so the machine stays in the same state, leaves the cell unchanged, and moves the head to the right.



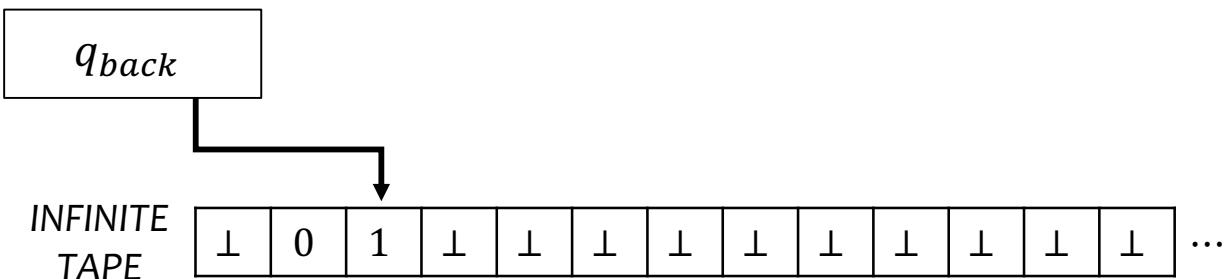


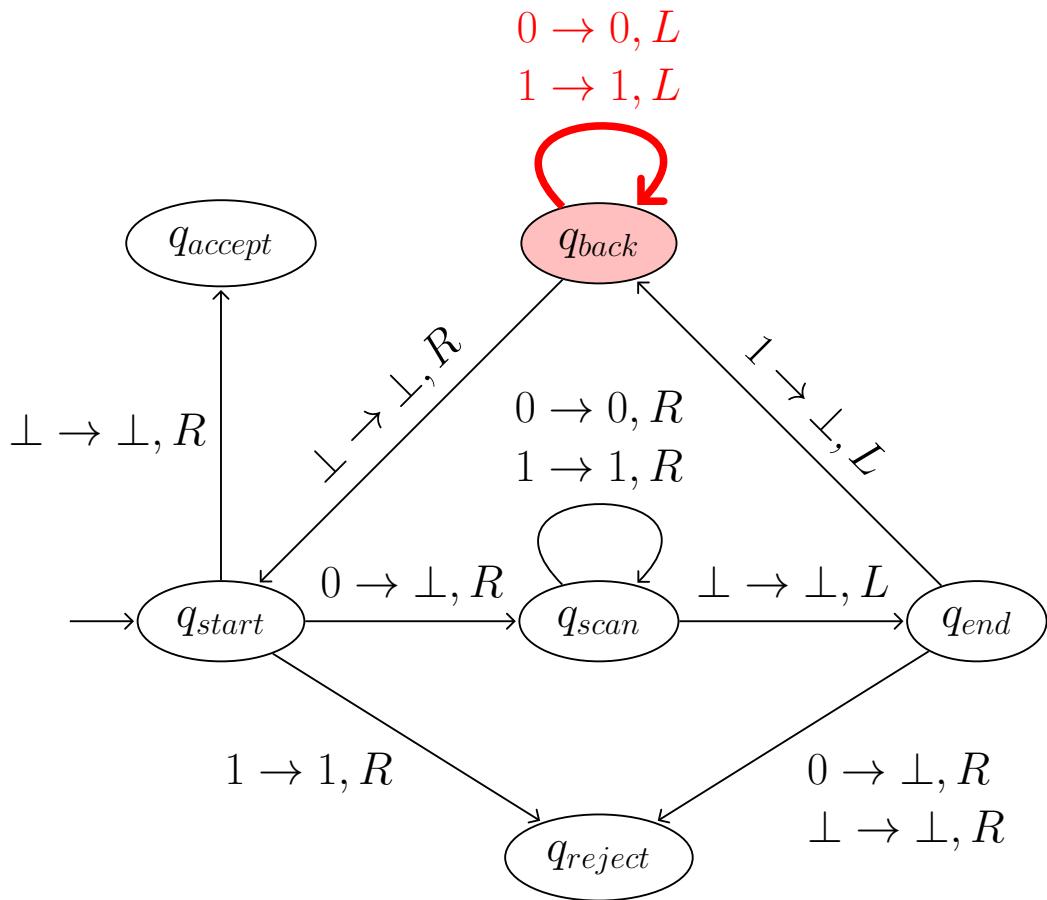
Now the cell at the head has a blank symbol, so the transition function tells us that the machine transitions to  $q_{end}$ , leaves the blank in the cell, and moves the head to the left.



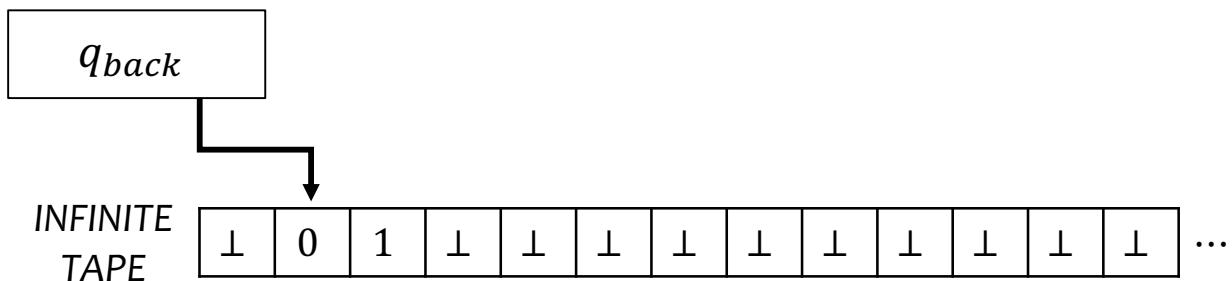


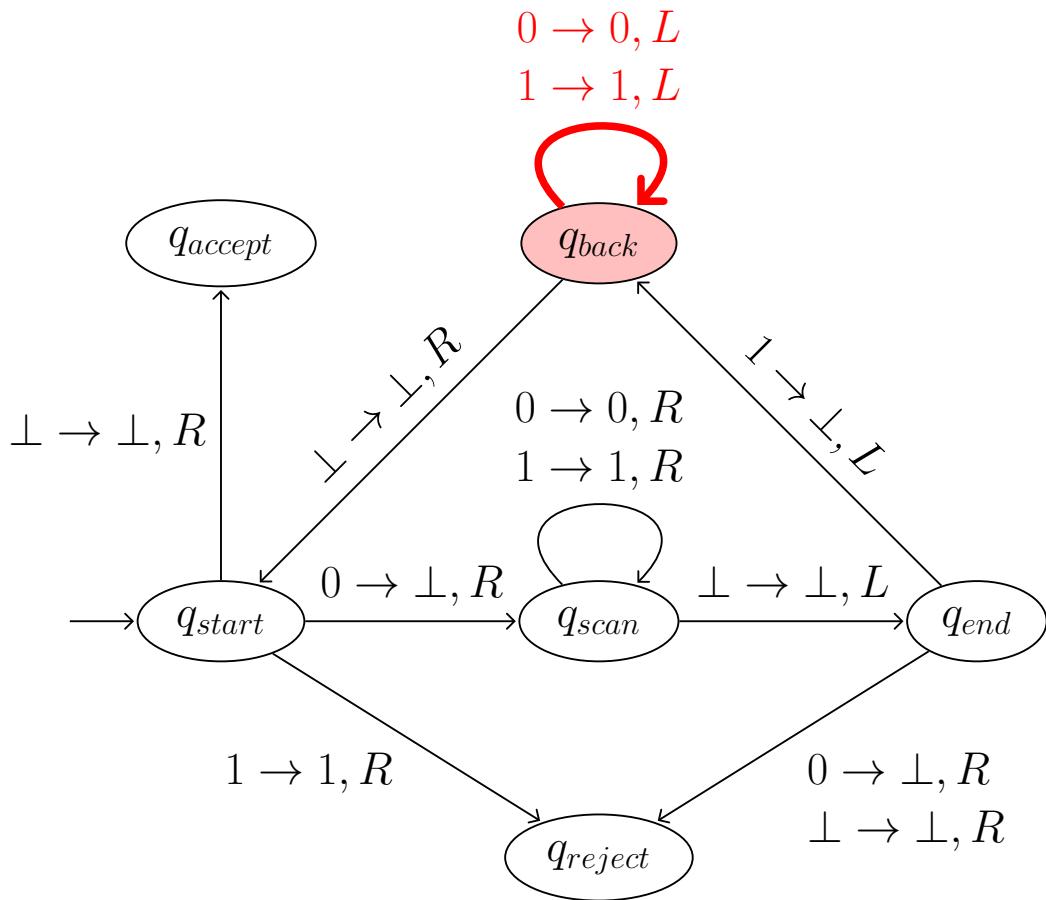
The cell at the head has a one, so the transition function tells us that the machine transitions to  $q_{back}$ , writes a blank to the cell, and moves the head to the left.



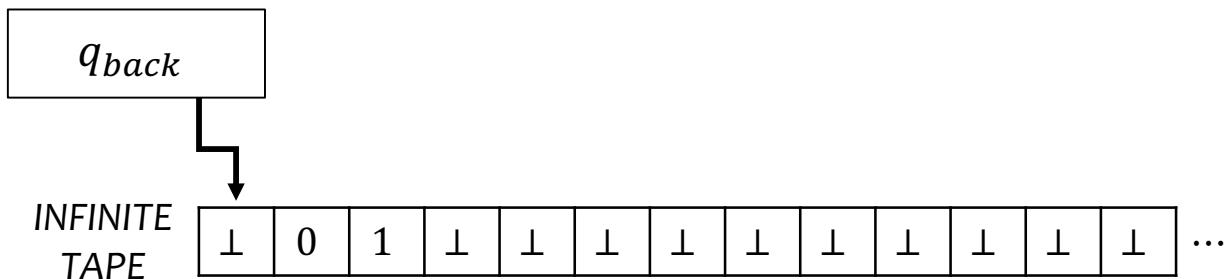


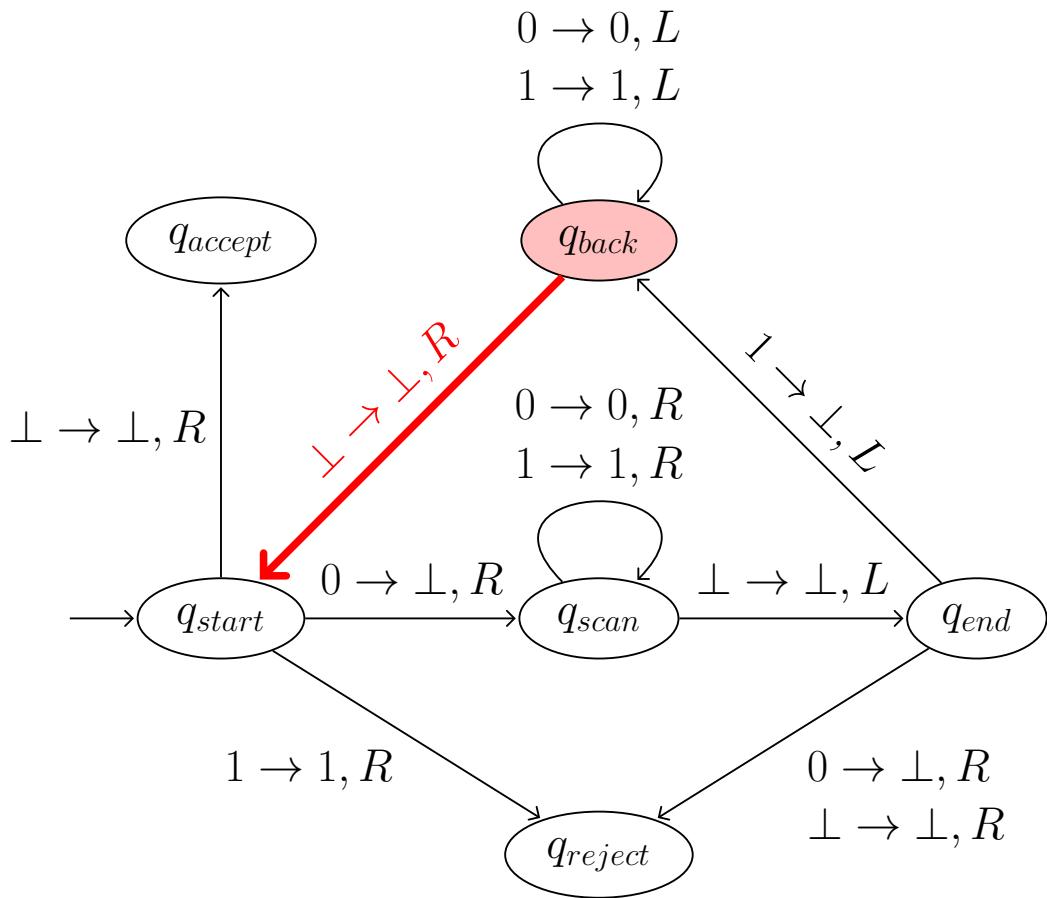
The cell at the head has a one, so the transition function tells us that the machine stays in  $q_{back}$ , leaves the one in the cell, and moves the head to the left.



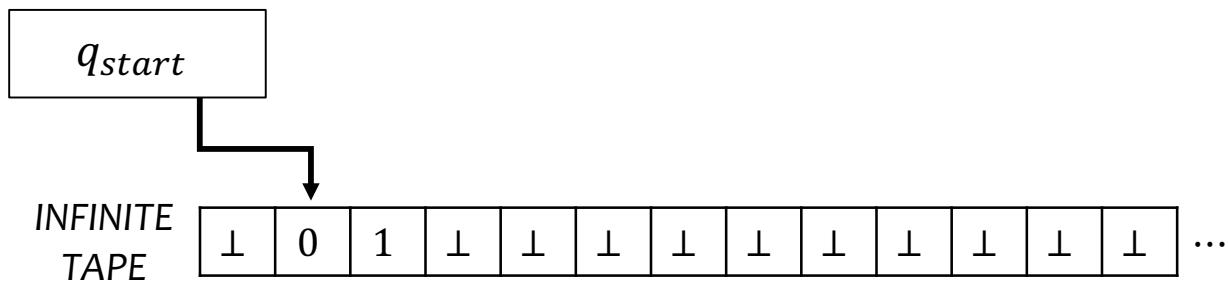


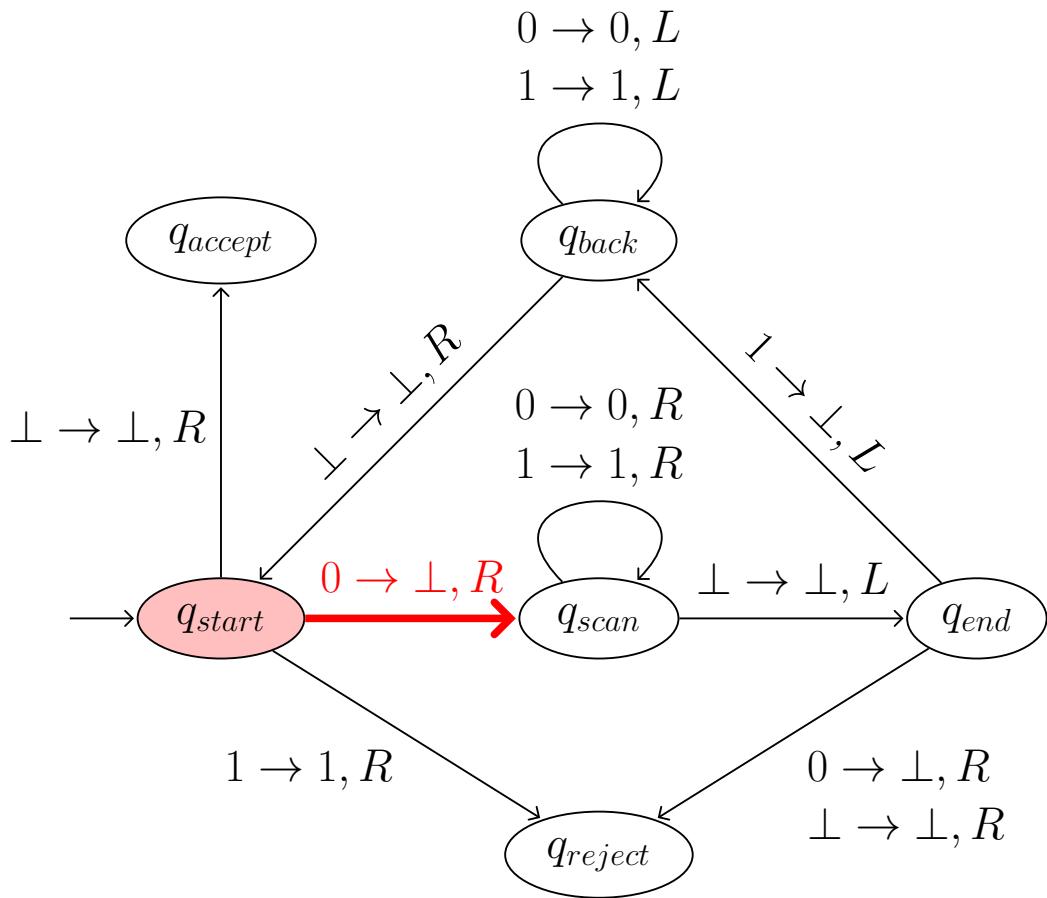
The cell at the head now has a zero, so the transition function tells us that the machine stays in  $q_{back}$ , leaves the zero in the cell, and moves the head to the left.



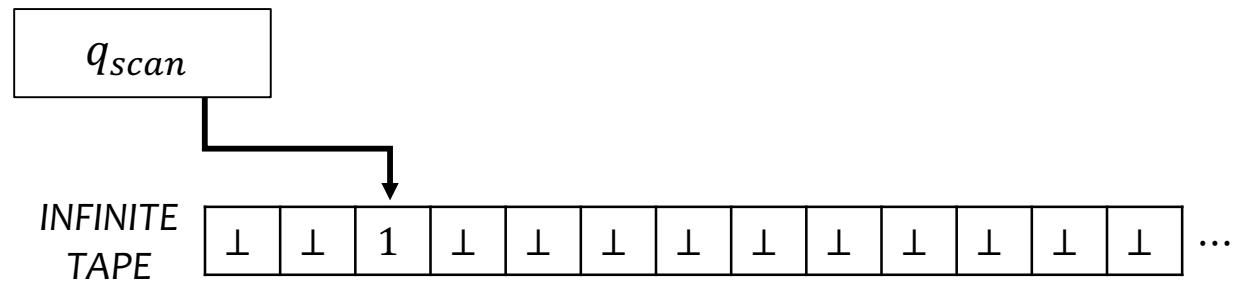


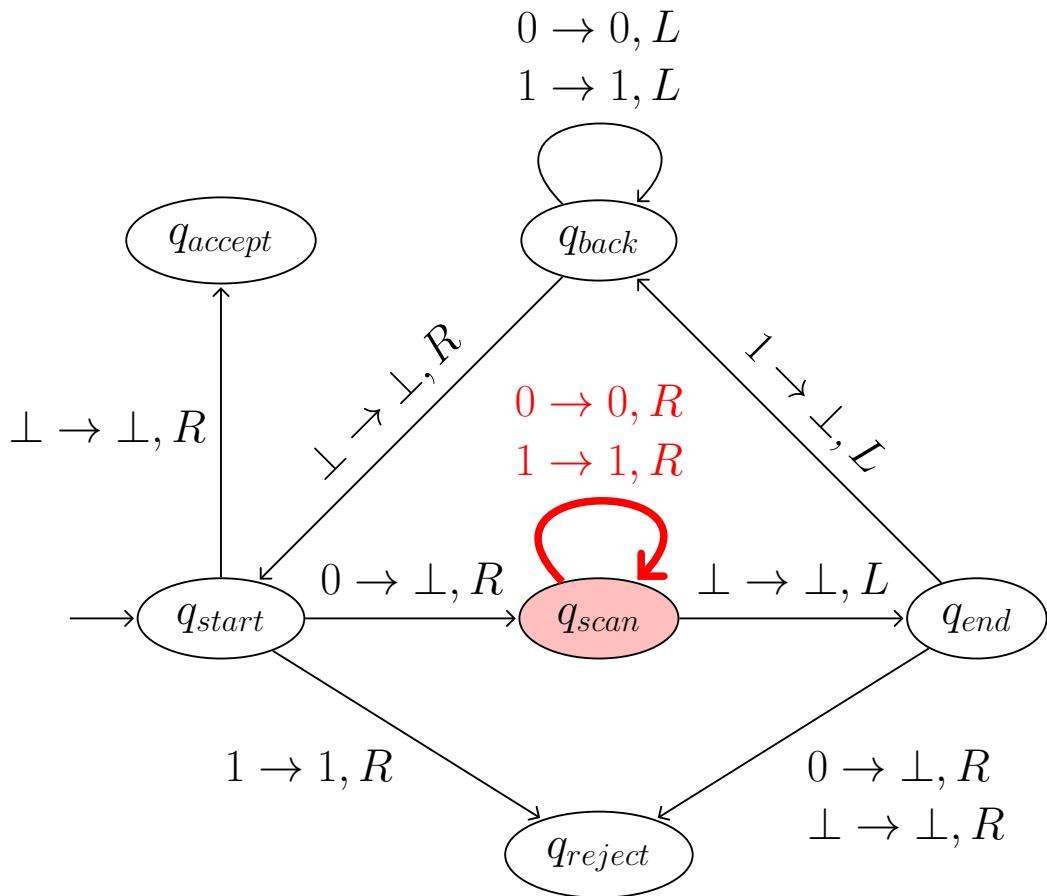
The cell at the head now has a blank, so the transition function tells us that the machine transitions to  $q_{start}$ , leaves the blank in the cell, and moves the head to the right.



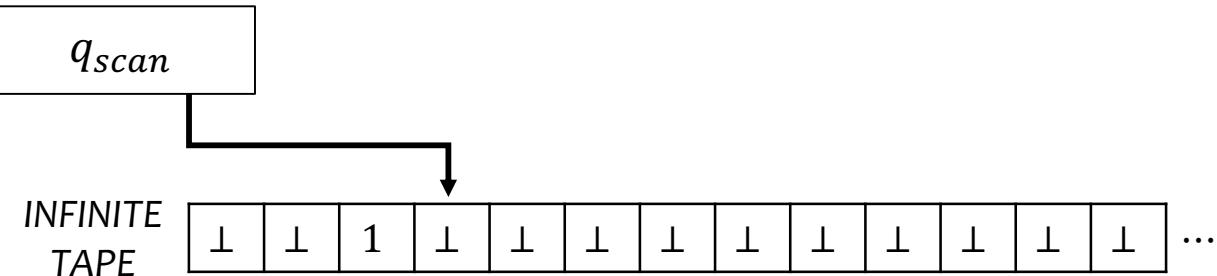


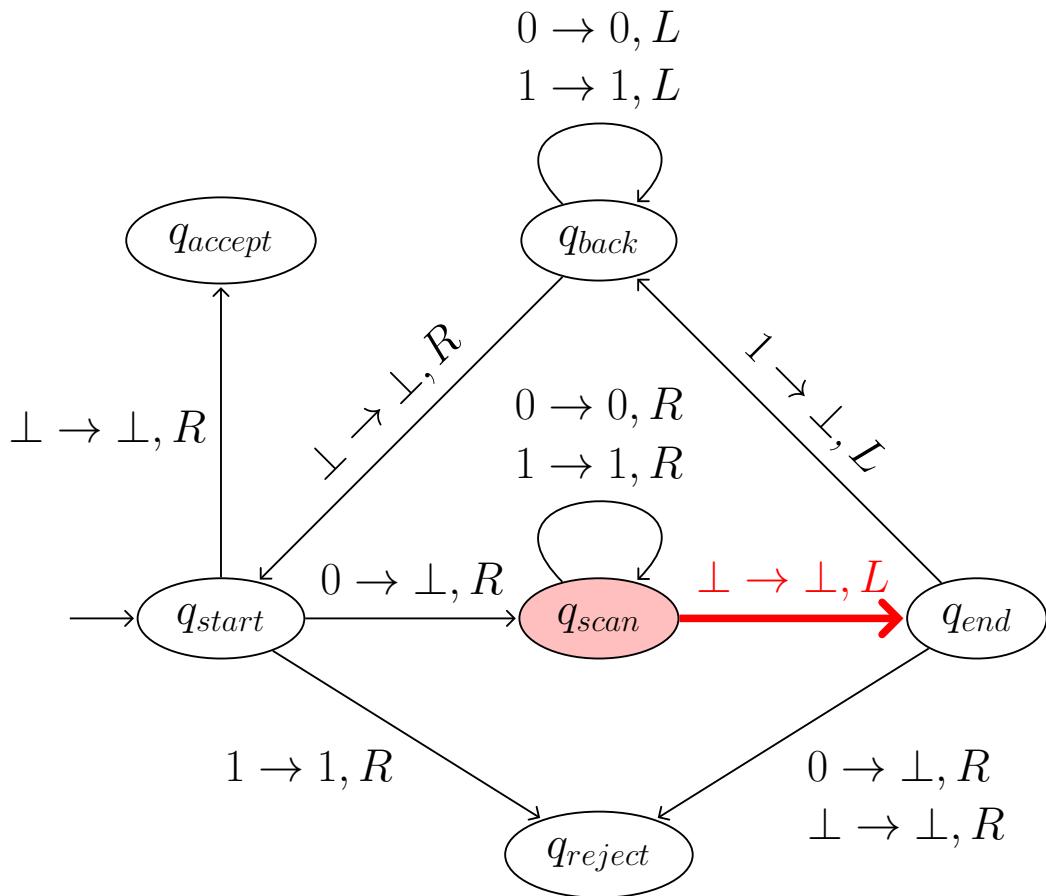
The cell at the head now has a zero, so the machine transitions to  $q_{\text{scan}}$ , writes a blank to the cell, and moves the head to the right.



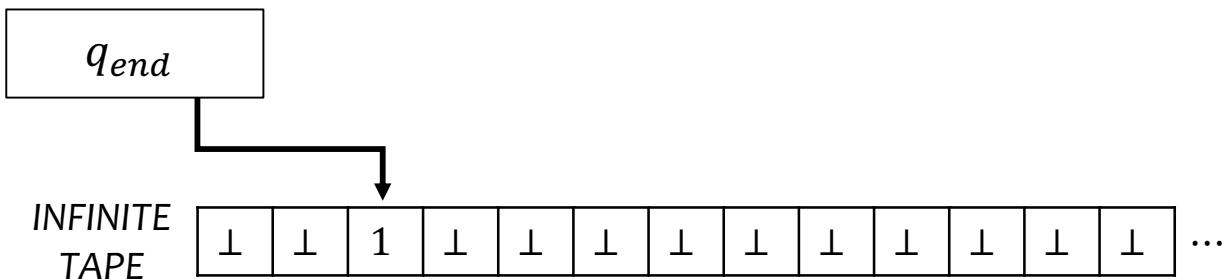


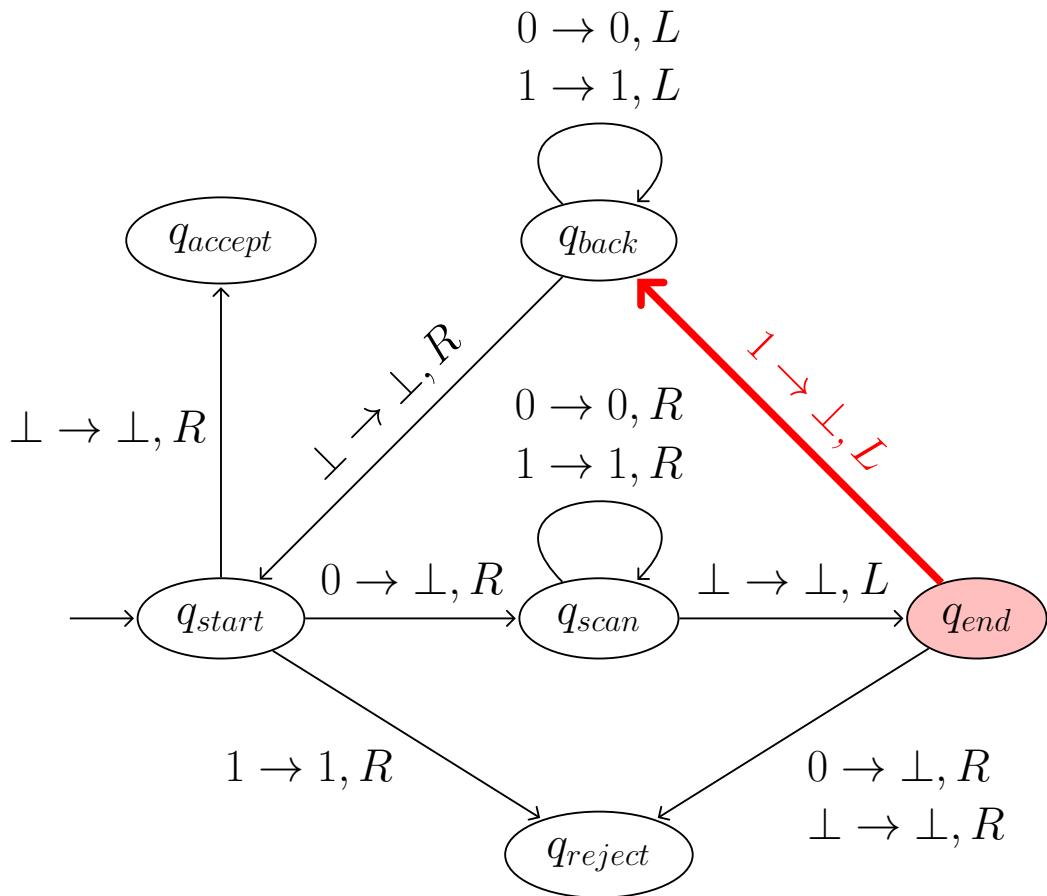
The cell at the head now has a one, so the machine stays in  $q_{scan}$ , leaves the one in the cell, and moves the head to the right.



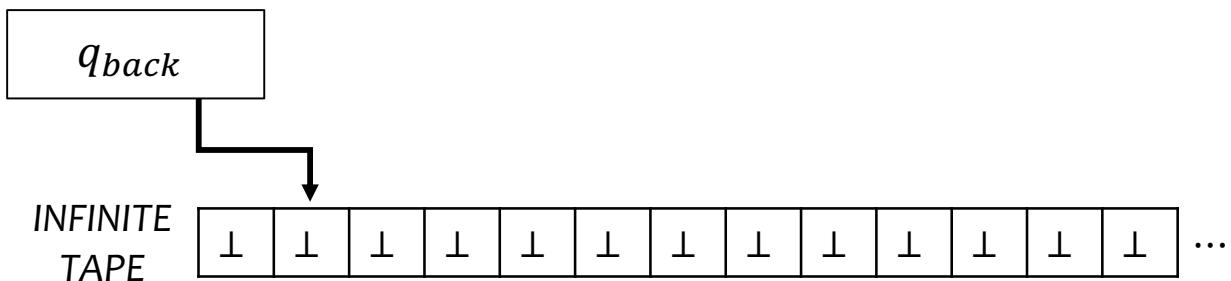


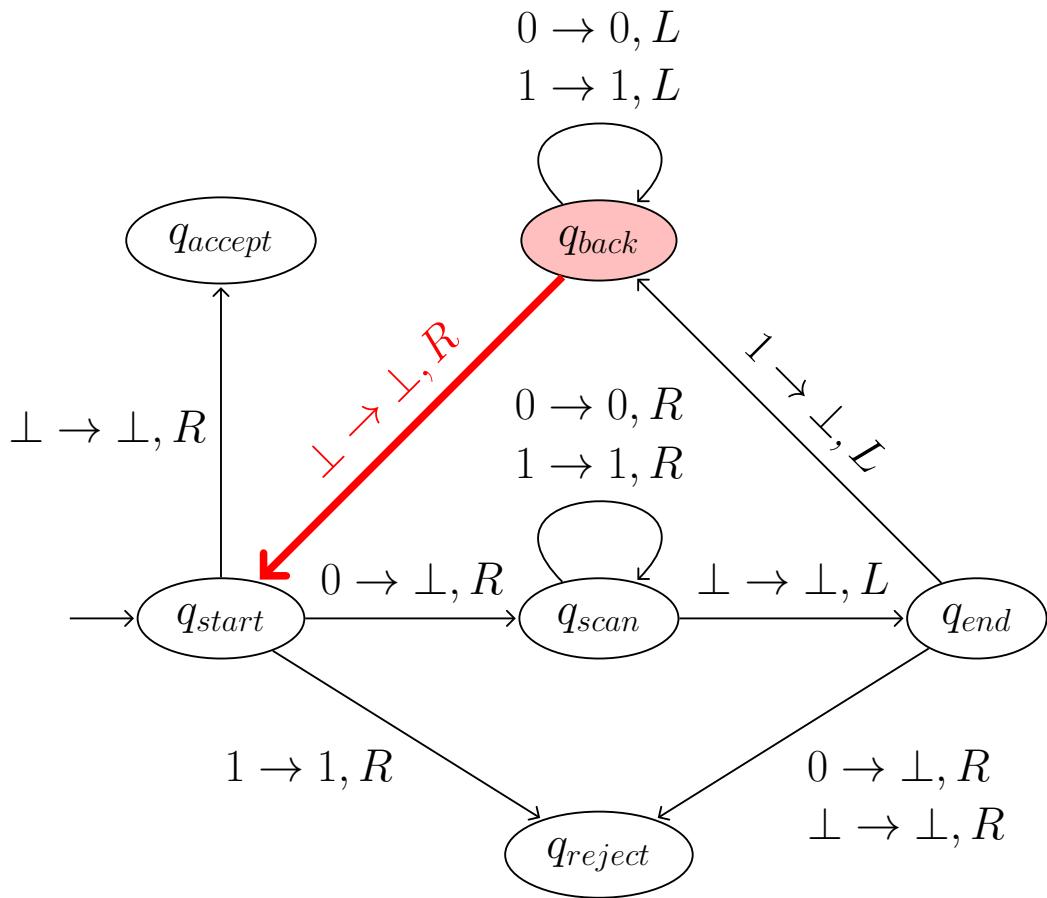
The cell at the head has a blank symbol, so the machine transitions to  $q_{end}$ , leaves the blank in the cell, and moves the head to the left.



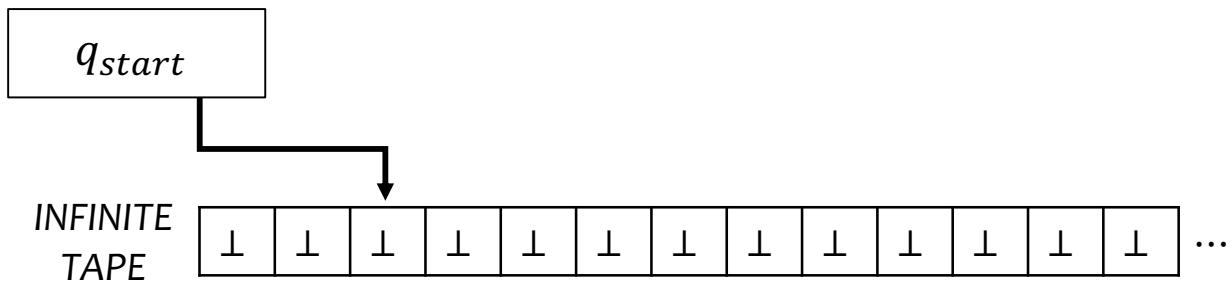


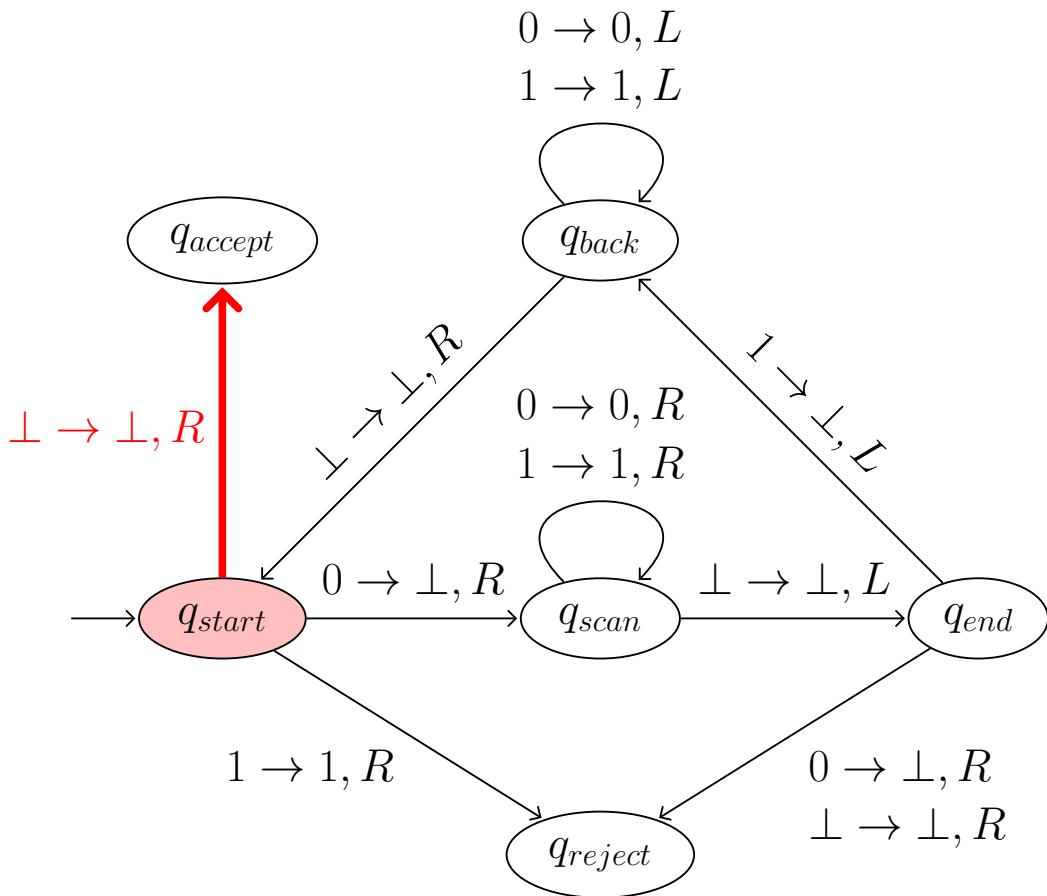
The cell at the head has a one, so the machine transitions to  $q_{back}$ , writes a blank to the cell, and moves the head to the left.



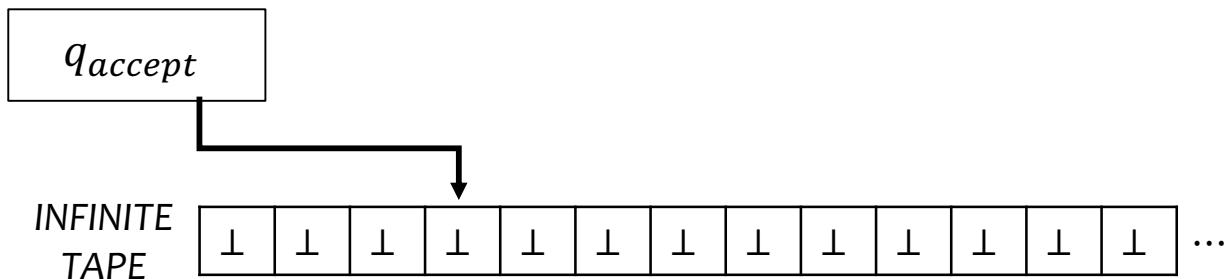


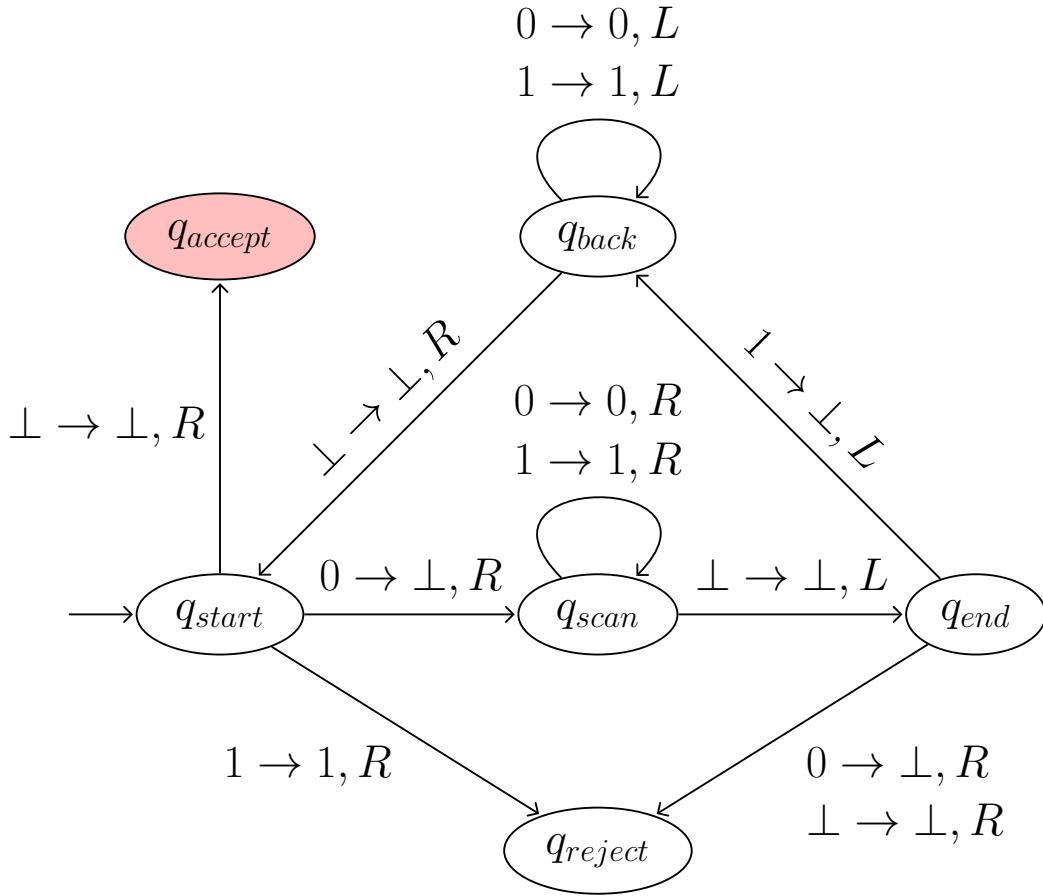
The cell at the head has a blank, so the machine transitions to  $q_{start}$ , leaves a blank in the cell, and moves the head to the right.





The cell at the head has a blank, so the machine transitions to  $q_{acc}$ , leaves a blank in the cell, and moves the head to the right.





The machine has reached the accept state  $q_{\text{acc}}$ , so it halts and accepts.

By generalizing the above example, we can see that this machine accepts any input string that consists of some non-negative number of zeros followed by the *same* number of ones, and rejects otherwise. In other words, it decides the language

$$L = \{0^n 1^n : n \in \mathbb{N}\} = \{\varepsilon, 01, 0011, 000111, \dots\}.$$

Recall that we proved above that this language cannot be decided by a *finite automaton*. However, we have just seen that it can be decided by a Turing machine.

How powerful is the Turing-machine model? The *Church-Turing thesis* asserts the following:

**Theorem 59 (Church-Turing Thesis)** *A function can be computed by an “effective” or “mechanical” procedure—in other words, an algorithm—if and only if it can be computed by a Turing machine.*

The terms “effective” and “mechanical” are not precisely and formally defined, so the Church-Turing thesis is not a statement that can be proved or disproved. Instead, it is an assertion that any well-defined procedure consisting of a *finite* number of precise instructions, which can be executed without any ingenuity or creativity, and which halts after a finite number of steps and produces a correct answer, can be represented by a Turing machine. Alternatively, it can be seen as the *definition* of what “algorithm” formally means: anything that can be represented by a Turing machine.

Support for the Church-Turing thesis can be found in the fact that many quite different-looking proposed models of computation have been shown to be *equivalent* to the Turing-machine model: whatever can be computed in such a model can also be computed by a Turing machine, and vice-versa. These include the lambda calculus and its variants, general recursive functions, most widely used computer programming languages, and more. So, despite the simplicity of the Turing-machine model, we have good reason to believe that it captures the essential nature of computation.

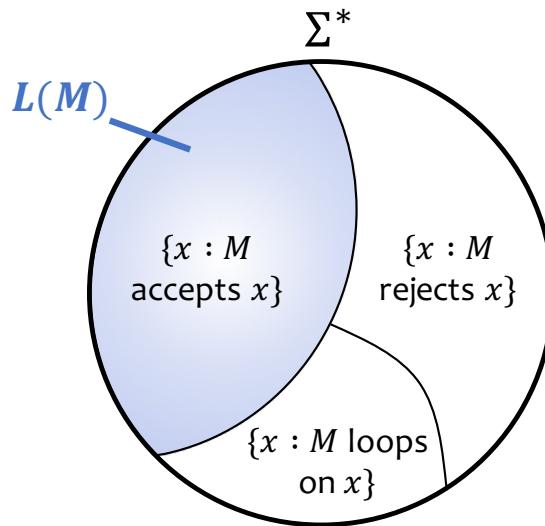
## 8.1 The Language of a Turing Machine

Now that we have formal definitions of both languages and Turing machines, which respectively formalize the notions of *problems* and *algorithms* (or *programs*), we connect them together. Recall that a Turing machine  $M$  has an input alphabet  $\Sigma$ , and  $\Sigma^*$  is the set of all possible inputs to the machine. When  $M$  is run on an input  $x \in \Sigma^*$ , which we often denote with the notation  $M(x)$ , there are three possible, mutually exclusive behaviors:

- $M$  accepts  $x$ , often written as “ $M(x)$  accepts”:  $M$  eventually halts due to reaching the accept state  $q_{\text{acc}}$ ;
- $M$  rejects  $x$ , often written as “ $M(x)$  rejects”:  $M$  eventually halts due to reaching the reject state  $q_{\text{rej}}$ ;
- $M$  loops on  $x$ , often written as “ $M(x)$  loops”:  $M$  never reaches a final state, and its execution continues forever.

**Definition 60 (Language of a Turing machine)** The *language*  $L(M)$  of a Turing machine  $M$  is the set of strings that the machine accepts:

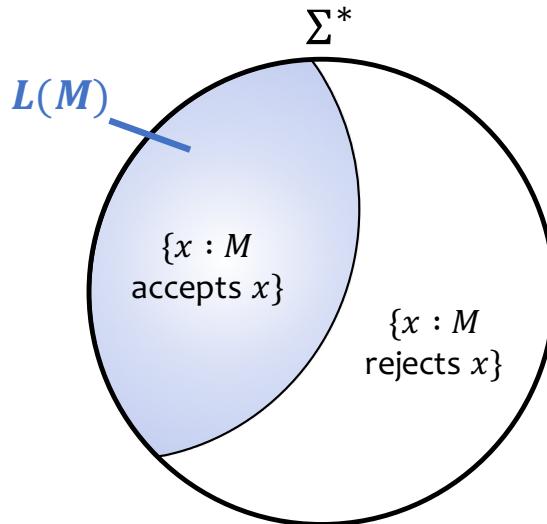
$$L(M) = \{x : M \text{ accepts } x\} .$$



By definition, every Turing machine  $M$  has a corresponding language  $L(M)$ , and  $x \in L(M)$  if and only if  $M(x)$  accepts; thus,  $x \notin L(M)$  if and only if  $M(x)$  rejects or loops.

For example, for the machine above that accepts all strings (and only those strings) that consist entirely of ones, we have that  $L(M) = \{1\}^* = \{\varepsilon, 1, 11, 111, \dots\}$ .

Some Turing machines halt (i.e., accept or reject) on every input; they do not loop on any input. Such a machine is called a *decider*, and it is said to *decide* its language.



**Definition 61 ('Decides' for Turing machines)** A Turing machine  $M$  decides a language  $L \subseteq \Sigma^*$  if:

1.  $M$  accepts every  $x \in L$ , and
2.  $M$  rejects every  $x \notin L$ .

Equivalently:

1.  $M$  halts on every  $x \in \Sigma^*$ , and
2.  $x \in L$  if and only if  $M$  accepts  $x$ .

A language is *decidable* if some Turing machine decides it.

The equivalence between the two pairs of conditions can be seen as follows: the first pair of properties immediately implies the latter pair. For the latter pair, the first property implies that on every input,  $M$  either accepts or rejects, and then the second property implies that  $M$  accepts every  $x \in L$  and rejects every  $x \notin L$ , as needed.

In general, a Turing machine might not decide any language, because it might loop on one or more inputs. But if  $M$  is a decider—i.e., it does not loop on any input—then  $M$  decides its language  $L(M)$ , and does not decide any other language. In other words, a particular Turing machine decides at most one language.

We also briefly mention a relaxation of the notion of deciding, called *recognizing*.

**Definition 62 ('Recognizes' for Turing machines)** A Turing machine  $M$  recognizes a language  $L \subseteq \Sigma^*$  if:

1.  $M$  accepts every  $x \in L$ , and
2.  $M$  rejects or loops on every  $x \notin L$ .

Equivalently:  $x \in L$  if and only if  $M$  accepts  $x$ .

A language is *recognizable* if some Turing machine recognizes it.

In comparison to deciding (Definition 61), here  $M$  must still accept every string in the language, but it is *not* required to halt on every input: it may loop on any string not in the language (indeed, it may loop on all such strings!). So, if a machine decides a language, it also recognizes that language, but not necessarily vice-versa. Observe that, by definition, every Turing machine  $M$  recognizes *exactly one* language, namely, the machine's language  $L(M)$ . See *Recognizability* (page 123) for further details and results.

Altogether, we now have the following formalizations:

- A *language* is the formalization of a decision problem.

- A *Turing machine* is the formalization of a program or algorithm.
- A machine *deciding* a language is the formalization of a program solving a decision problem.

Thus, our original question about whether a given problem is solvable by a computer is the same as asking whether its corresponding language is decidable.

## 8.2 Decidable Languages

By definition, a language is decidable if some Turing machine decides it, i.e., the machine accepts every string in the language and rejects every string not in the language. Does this mean that, to demonstrate that a language is decidable, we must formally define the seven-tuple of such a Turing machine? Fortunately, no. Because all known computer programming languages, including valid pseudocode, can be simulated by a Turing machine (see the *Church-Turing thesis* (page 84)), we can write an algorithm in such a language and be assured that there is a corresponding Turing machine. As an example, consider the following language:

$$L_{\text{GCD}} = \{(a \in \mathbb{N}, b \in \mathbb{N}) : \gcd(a, b) = 1\} .$$

An algorithm to decide this language is as follows:

```
function DECIDEGCD( $a, b$ )
  if EUCLID( $\max(a, b), \min(a, b)$ ) = 1 then accept
  reject
```

We now analyze this algorithm to show that it decides  $L_{\text{GCD}}$ , according to [Definition 61](#):

- If  $(a, b) \in L_{\text{GCD}}$ , then  $\gcd(a, b) = 1$  by definition, so the call to EUCLID returns 1, hence DECIDEGCD accepts  $(a, b)$ .
- If  $(a, b) \notin L_{\text{GCD}}$ , then  $\gcd(a, b) \neq 1$ , so the call to EUCLID returns a value not equal to 1, hence DECIDEGCD rejects  $(a, b)$ .

Alternatively, we can analyze the algorithm as follows: first, it halts on any input, because EUCLID does (recall that EUCLID runs in  $O(\log(a + b))$  iterations, in the worst case). Second,  $(a, b) \in L_{\text{GCD}}$  if and only if  $\gcd(a, b) = 1$ , which holds if and only if DECIDEGCD( $a, b$ ) accepts, by the correctness of EUCLID and direct inspection of the pseudocode of DECIDEGCD.

Thus, DECIDEGCD decides  $L_{\text{GCD}}$ , so  $L_{\text{GCD}}$  is a decidable language.

In general, to demonstrate that a particular language is decidable, we first write an algorithm, and then analyze the algorithm to show that it decides the language, according to [Definition 61](#). Depending on how the algorithm is written, it may be more convenient to establish one pair of properties or the other from [Definition 61](#). As in the above example, we will often show both ‘styles’ of analysis, although only one is needed for a particular proof.

Here we show some examples of how performing certain operations on decidable languages preserves decidability.

**Lemma 63** Let  $L$  be any decidable language. Then  $L' = L \cup \{\varepsilon\}$  is also decidable.

**Proof 64** By definition, since  $L$  is decidable, there exists some Turing machine  $D$  that decides  $L$ . We use it to construct another machine  $D'$  that decides  $L'$  (more precisely, because  $D$  exists, the following machine also exists):

```
function  $D'(x)$ 
  if  $x = \varepsilon$  then accept
```

```
return  $D(x)$ 
```

We analyze the behavior of  $D'$  on an arbitrary input string  $x$ , as follows:

- If  $x \in L' = L \cup \{\varepsilon\}$ , then either  $x = \varepsilon$  or  $x \in L$  (or both). In the first case,  $D'(x)$  accepts by the first line of pseudocode. In the second case,  $D(x)$  accepts because  $D$  decides  $L$ . So, in either case,  $D'(x)$  accepts, as needed.
- If  $x \notin L' = L \cup \{\varepsilon\}$ , then  $x \neq \varepsilon$  and  $x \notin L$ . Therefore, the first line of pseudocode does not accept, and then on the second line  $D(x)$  rejects, so  $D'(x)$  rejects, as needed.

So, by [Definition 61](#),  $D'$  does indeed decide  $L'$ , hence  $L'$  is decidable.

Alternatively, we can use the other pair of conditions in [Definition 61](#) to analyze  $D'$  as follows. First,  $D'$  halts on every input, because its first line does not loop, and on the second line  $D$  does not loop because it is a decider. Next,

$$\begin{aligned} x \in L' &\iff x \in L \cup \{\varepsilon\} \\ &\iff x = \varepsilon \text{ or } x \in L \\ &\iff D'(x) \text{ accepts on the first line, or } D(x) \text{ accepts} \\ &\iff D'(x) \text{ accepts.} \end{aligned}$$

We emphasize that for this second style of analysis, it is very important that all the implications hold in both “directions” (i.e., be “if and only if” statements), which needs to be carefully checked.  $\square$

**Lemma 65** *For any decidable languages  $L_1$  and  $L_2$ , their union  $L = L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$  is also decidable.*

**Proof 66** Since  $L_1$  and  $L_2$  are decidable, there exist machines  $M_1$  and  $M_2$ , respectively, that decide them. That is,  $M_1$  accepts every string in  $L_1$  and rejects every string not in  $L_1$ , and similarly for  $M_2$  and  $L_2$ . We use them to construct a new machine  $M$  that decides  $L$  (again, because  $M_1, M_2$  exist, so does the following machine  $M$ ):

```
function  $M(x)$ 
    if  $M_1(x)$  accepts then accept
    if  $M_2(x)$  accepts then accept
    reject
```

We analyze the behavior of  $M$  on an arbitrary input string  $x$ , as follows:

- If  $x \in L = L_1 \cup L_2$ , then  $x \in L_1$  or  $x \in L_2$  (or both). If the former case,  $M_1(x)$  accepts and hence  $M(x)$  accepts on its first line, and in the latter case,  $M_2(x)$  accepts and hence  $M(x)$  accepts on its second line. Either way,  $M(x)$  accepts, as needed.
- If  $x \notin L$ , then  $x \notin L_1$  and  $x \notin L_2$ . So, both  $M_1(x)$  and  $M_2(x)$  reject, and hence  $M(x)$  reaches its final line and rejects, as needed.

So, by [Definition 61](#),  $M$  does indeed decide  $L$ , hence  $L$  is decidable.

Alternatively, we can use the other pair of conditions in [Definition 61](#) to analyze  $M$  as follows. First,  $M$  halts on every input, because its calls to  $M_1$  and  $M_2$  halt (since they are deciders). Next, by the properties of  $M_1, M_2$  and the definition of  $M$ ,

$$\begin{aligned} x \in L &\iff x \in L_1 \text{ or } x \in L_2 \\ &\iff M_1(x) \text{ accepts or } M_2(x) \text{ accepts} \\ &\iff M(x) \text{ accepts,} \end{aligned}$$

where the last line holds by inspection of the code of  $M$ . □

**Example 65** demonstrates that the class of decidable languages is *closed* under union: if we take the union of any two members of the class, the result is also a member of the class, i.e., it is a decidable language.

**Exercise 67** Show that the class of decidable languages is closed under:

- a) intersection (i.e., if  $L_1$  and  $L_2$  are decidable, then so is  $L_1 \cap L_2$ );
- b) complement (i.e., if  $L$  is decidable, then so is  $\overline{L}$ ).

Is *every* language decidable (by a Turing machine)? In other words, can every decision problem be solved by some algorithm? We will consider this question below starting with the section on *Diagonalization* (page 92), following a digression on alternative but equivalent models of Turing machines.

## 8.3 Equivalent Models

As an illustration of the power of the simple “one-tape” Turing-machine model defined above, we will demonstrate that an extended model, that of the *two-tape* Turing machine, is actually no more powerful than the original one-tape model. Similar equivalences can be demonstrated for other natural variations of the Turing-machine model, such as one with a *two-way infinite* tape, one with a *two-dimensional* (or any finite-dimensional) tape, one with (a finite number of) *multiple heads*, one with *nondeterministic* operations, and combinations thereof.

Similar to the one-tape version, the two-tape model consists of a seven-tuple:

$$M = (\Sigma, \Gamma, Q, q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}, \delta).$$

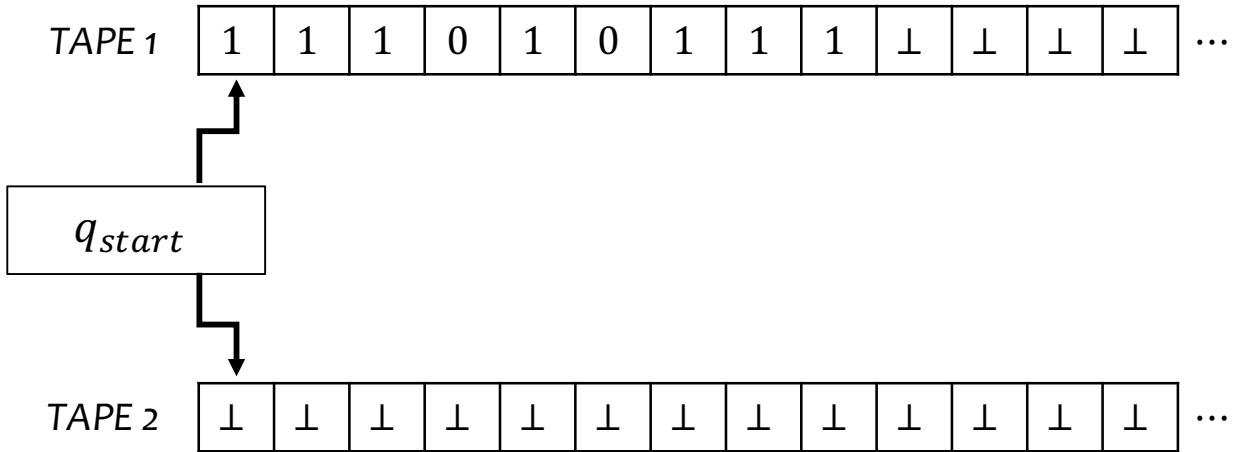
The components  $\Sigma$ ,  $\Gamma$ ,  $Q$ ,  $q_{\text{start}}$ ,  $q_{\text{acc}}$ , and  $q_{\text{rej}}$  are exactly as in the one-tape model. The transition function  $\delta$ , however, now takes two tape symbols as input, and outputs two tape symbols and two directions—in all cases, one for each tape head. Formally, we have

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

(Recall that the notation  $\Gamma^2$  is shorthand for  $\Gamma \times \Gamma$ .) In more detail, the components of the transition function are as follows:

$$\begin{array}{ccccccc} \delta : & \underbrace{(Q \setminus F)}_{\text{non-terminal state}} & \times & \underbrace{\Gamma}_{\text{Cell contents on tape 1}} & \times & \underbrace{\Gamma}_{\text{Cell contents on tape 2}} & \rightarrow \\ & & & & & & \\ & \underbrace{Q}_{\text{New state}} & \times & \underbrace{\Gamma}_{\text{Symbol to write on tape 1}} & \times & \underbrace{\Gamma}_{\text{Symbol to write on tape 2}} & \times \underbrace{\{L, R\}}_{\text{Where to move head 1}} \times \underbrace{\{L, R\}}_{\text{Where to move head 2}} \end{array}$$

The initial state of the machine has the input written on the first tape, followed by blank symbols, and only blank symbols on the second tape. Both heads begin at the leftmost cells of their respective tapes.



In each step, the machine is in some state  $q$  and reads one symbol from each tape, say,  $\gamma_1$  and  $\gamma_2$ . The transition function maps  $(q, \gamma_1, \gamma_2)$  to the next state, symbols to write on each tape, and directions to move each head:

$$\delta(q, \gamma_1, \gamma_2) = (q', \gamma'_1, \gamma'_2, d_1, d_2).$$

The machine transitions to state  $q'$ , writes  $\gamma'_1$  to the first tape and moves its head in the direction  $d_1$ , and writes  $\gamma'_2$  to the second tape and moves its head in the direction  $d_2$ .

We now show that the one-tape and two-tape models are *equivalent*, by proving that anything that can be computed in one model can also be computed by the other. In other words, for every one-tape Turing machine, there is a corresponding two-tape machine that, for each input, has exactly the same output behavior (accept, reject, or loop); and likewise, for every two-tape machine, there is a corresponding one-tape machine.

The first direction is conceptually trivial: for any one-tape Turing machine, we can construct an equivalent two-tape machine that just ignores its second tape. Formally, the two-tape machine has the same tape and input alphabets  $\Sigma$  and  $\Gamma$ , the same set of states  $Q$ , and the same start and final states  $q_{\text{start}}$ ,  $q_{\text{acc}}$ , and  $q_{\text{rej}}$ . The transition function  $\delta_2$  is slightly more subtle. For each entry

$$\delta(q, \gamma) = (q', \gamma', d)$$

of the one-tape machine, the two-tape machine's transition function has the corresponding entries

$$\delta_2(q, \gamma, \gamma_2) = (q', \gamma', \gamma_2, d, R)$$

for *every*  $\gamma_2 \in \Gamma$ . In other words, in each computational step, the two-tape machine ignores the contents of its second tape and moves its second head to the right, and does whatever the one-tape machine does with the first tape. (Note that even though the second tape consists of all blanks, to have a well-defined transition function  $\delta_2$  we must define the function for *all* pairs of tape symbols, which is why we need to consider *every*  $\gamma_2 \in \Gamma$ .) Since the machine's behavior is solely dependent on the contents of the first tape, the two-tape machine makes exactly the same transitions as the one-tape machine on a given input, and it ends up accepting, rejecting, or looping on each input exactly as the one-tape machine does.

The other direction of the equivalent is significantly more complicated: for an arbitrary two-tape machine, we need to construct a one-tape machine that produces the same output behavior on any given input. There are many ways to do this, and we describe one such construction at a high level.

First, we consider how a one-tape machine can store the contents of two tapes, as well as two head positions. A key observation is that any point during the two-tape machine's computation, the amount of actual data (i.e., excluding trailing blank cells) on the two tapes is *finite*. Thus, we can store that data on a single tape just as well. We use the following format:

$$\# \underbrace{\dots \bullet \gamma_1 \dots}_{\text{Contents of Tape 1}} \# \underbrace{\dots \bullet \gamma_2 \dots}_{\text{Contents of Tape 2}} \#$$

We use a special  $\#$  symbol (which we include in the tape alphabet of the one-tape machine) to separate the (finite) contents of each tape, and to indicate the endpoints of those contents. Between these markers, the contents of each tape are represented using the same tape symbols as in the two-tape machine, except that we denote the position of each head by placing a special ‘dot’ symbol  $\bullet$  (which we also include in the tape alphabet) to the left of the cell at which the head is currently located. Thus, if  $\Gamma$  is the tape alphabet of the two-tape machine, then the tape alphabet of the one-tape machine is

$$\Gamma_1 = \Gamma \cup \{\#, \bullet\}.$$

We won’t give a formal definition of the states or transition function, but we will describe how the machine works at a high level. Given an input  $w_1 w_2 \dots w_n$ , the one-tape machine does the following:

1. Rewrite the tape to be in the correct format:

$$\# \bullet w_1 \cdots w_n \# \bullet \sqcup \#$$

This entails shifting each input symbol by two spaces, placing a  $\#$  marker in the first cell and a dot in the second, placing a  $\#$  marker after the last input symbol (at the first blank cell), writing a dot symbol in the next cell, followed by a blank symbol, followed by one more  $\#$  marker.

2. Simulate each step of the two-tape machine:

- a) Scan the tape to find the two dot symbols, representing the positions of each head, and ‘remember’ the symbols stored to the right of the dots. (Because there are only a finite number of possibilities, the machine can ‘remember’ these symbols via a finite number of states, without writing anything to the tape.)
- b) Transition to a new state according to the transition function of the two-tape machine (depending on the current state and the symbols immediately to the right of the dots).
- c) Replace the symbols to the right of each dot with the symbols to be written to each tape, as specified in the transition function of the two-tape machine.
- d) Move the dots in the appropriate directions. If a dot is to move to the left but there is a  $\#$  marker there, then it stays in the same position (corresponding to what happens when a tape head tries to move left when at the leftmost tape cell). On the other hand, if the dot is to move to the right but there is a  $\#$  marker one more cell to the right, shift the marker and all the symbols following it (up to the final  $\#$  marker) one cell to the right, placing a blank symbol in the space created by the shift. This ensures that there is a valid symbol to the right of the dot (i.e., at the corresponding head) that can be read in the next simulated step.

Constructing this one-tape machine from the definition of the two-tape machine is a tedious, but completely mechanical, process. The important point is that the one-tape machine perfectly simulates the operation of the original two-tape machine, even though it typically takes many steps to carry out what the original machine does in a single step. If the two-tape machine accepts an input, then the one-tape simulation will do so as well, and similarly if the original machine rejects or loops.

We have demonstrated that a two-tape Turing-machine model is *equivalent* to the simpler, one-tape model. In general, a computational model or programming language that is powerful enough to simulate any one-tape Turing machine is said to be *Turing-complete*. In the other direction, a Turing-complete model is said to be *Turing-equivalent* if it can be simulated by a Turing machine; recall that the Church-Turing thesis asserts that any effective procedure can be simulated by a Turing machine.

All real-world programming languages, including C++, Python, and even LaTeX, are Turing-complete, and all of them can be simulated by a Turing machine. Thus, the results we prove about computation using the Turing-machine model also apply equivalently to real-world models. The simplicity of Turing machines makes such results easier to prove than if we were to reason about C++ or other complicated languages.

## DIAGONALIZATION

Is every language decidable (by a Turing machine)? We have seen that every machine has an associated language (of strings the machine accepts), but does every language have a corresponding machine that decides it? We consider this question next.

One way of answering the question of whether there exists an undecidable language is to compare the “number” of languages with the “number” of machines. Let  $\mathcal{L}$  be the set of all languages, and let  $\mathcal{M}$  be the set of all machines. If we could demonstrate that  $\mathcal{L}$  is a “larger” set than  $\mathcal{M}$ , i.e., that  $|\mathcal{M}| < |\mathcal{L}|$ , then there are “more” languages than machines, and it follows that there must be some language that does not have a corresponding machine.

However, both  $\mathcal{M}$  and  $\mathcal{L}$  are *infinite* sets, which is why we have put the terms “number”, “more”, etc. in “scare quotes”—these terms typically apply only to *finite* quantities. In order to make the above approach work, we need appropriate tools to reason about the sizes of *infinite* sets.

## 9.1 Countable Sets

We refer to the size of a set as its *cardinality*, which measures the number of elements in the set. For a finite set, its cardinality is an element of the natural numbers  $\mathbb{N}$ . For an infinite set, on the other hand, the number of elements is not a natural number.<sup>20</sup> For our purposes, however, we need only to reason about the *relative* sizes of infinite sets to answer questions like whether  $|\mathcal{M}| < |\mathcal{L}|$ .

To start with, we recall some terminology related to functions. A (*total*) function  $f: A \rightarrow B$  associates, or maps, each element  $a \in A$  to some element  $f(a) \in B$ . (This is in contrast to a *partial function*, which may leave some elements of  $A$  unmapped.)

**Definition 68 (Injective/one-to-one, “no larger than”)** A (total) function  $f: A \rightarrow B$  is *injective*, or *one-to-one*, if it maps each element of  $A$  to a *different* element of  $B$ . In other words,  $f$  is injective if

$$f(a_1) \neq f(a_2) \text{ for all } a_1 \neq a_2 \text{ (where } a_1, a_2 \in A\text{).}$$

If an injective function  $f: A \rightarrow B$  exists, then we write  $|A| \leq |B|$ , and say that “ $A$  is no larger than  $B$ .”

Intuitively, the existence of an injective function from  $A$  to  $B$  shows that  $B$  has “at least as many elements as”  $A$ , which is why we write  $|A| \leq |B|$ . The power of comparing cardinalities via injective functions, as opposed to just by counting elements, is that it is also meaningful and works in (some) expected ways even for *infinite* sets. For example, it is possible to show the following.

**Exercise 69** Prove that if  $A \subseteq B$ , then  $|A| \leq |B|$ . (Hint: there is a trivial injective function from  $A$  to  $B$ .)

<sup>20</sup> There are other kinds of numbers<sup>21</sup> that can represent the sizes of infinite sets, but they are beyond the scope of this text.

<sup>21</sup> [https://en.wikipedia.org/wiki/Cardinal\\_number](https://en.wikipedia.org/wiki/Cardinal_number)

**Exercise 70** Prove that if  $|A| \leq |B|$  and  $|B| \leq |C|$ , then  $|A| \leq |C|$ . (Hint: compose injective functions.)

However, we *strongly caution* that in the context of infinite sets, the notation  $|A| \leq |B|$  has just the *specific meaning* given by Definition 68; it may not satisfy other properties that you are accustomed to for comparing *finite* quantities. For example, we cannot necessarily add or subtract on both sides of the  $\leq$  symbol and preserve the inequality. When in doubt, refer to the definition.

We are most interested in reasoning about how the sizes of various sets compare to “standard” infinite sets like the natural numbers  $\mathbb{N}$ . We first define the notion of a “countable” set, which is one that is “no larger than” the set  $\mathbb{N}$  of natural numbers.

**Definition 71 (Countable)** A set  $S$  is *countable* if  $|S| \leq |\mathbb{N}|$ , i.e., if there exists an injective function  $f: S \rightarrow \mathbb{N}$  from  $S$  to the natural numbers  $\mathbb{N}$ .

For example, the set  $S = \{a, b, c\}$  is countable, since the following function is injective:

$$\begin{aligned} f(a) &= 0 \\ f(b) &= 1 \\ f(c) &= 2 . \end{aligned}$$

In fact, any *finite* set  $S$  is countable: we can just list all the elements of  $S$ , assigning them to natural numbers in the order that we listed them. Moreover, this idea doesn’t just apply to finite sets: the same strategy can apply to an *infinite* set, as long as we have some way of listing its elements in some order, as the following lemma shows.

**Lemma 72** A set  $S$  is countable if and only if there is some enumeration (or list) of its elements as  $S = \{s_0, s_1, s_2, \dots\}$ , in which each  $s \in S$  appears (at least once) as  $s = s_i$  for some  $i \in \mathbb{N}$ .

**Proof 73** First we show that if there is some enumeration  $s_0, s_1, s_2, \dots$  of the elements in  $S$ , then there is an injective function  $f$  from  $S$  to  $\mathbb{N}$ . For each  $s \in S$ , simply define  $f(s) = i$  to be the smallest (i.e., first) index  $i$  for which  $s = s_i$ . By assumption, every  $s \in S$  has such an index, so  $f$  is a (total) function. And  $f$  is injective: if  $s \neq s'$ , then  $f(s) \neq f(s')$  because two different elements of  $S$  cannot occupy the same position in the list.

Now we show the opposite direction, that if there is an injective function  $f: S \rightarrow \mathbb{N}$ , then there is an enumeration of  $S$  (in which each element appears *exactly once*, in fact). Essentially, we list the elements of  $S$  in “sorted order” by their associated natural numbers (under  $f$ ). Formally, we define the sequence as:  $f^{-1}(0)$  if it exists, then  $f^{-1}(1)$  if it exists, then  $f^{-1}(2)$  if it exists, etc. Here  $f^{-1}(i) \in S$  denotes the element of  $S$  that  $f$  maps to  $i \in \mathbb{N}$ , if it exists; note that such an element is *unique* because  $f$  is injective. Since  $f$  is a (total) function, every  $s \in S$  maps to exactly one  $i \in \mathbb{N}$ , so  $s$  will appear exactly once in the enumeration, as claimed.  $\square$

As a concrete example of the second direction in the above proof, consider the injective function  $f: \{x, y, z\} \rightarrow \mathbb{N}$  defined by  $f(x) = 376$ ,  $f(y) = 203$ , and  $f(z) = 475$ . Exactly three values of  $f^{-1}(i)$  for  $i \in \mathbb{N}$  exist—namely, for  $i = 203, 376, 475$ —and they yield the enumeration  $f^{-1}(203) = y$ ,  $f^{-1}(376) = x$ ,  $f^{-1}(475) = z$  according to the above proof.

Lemma 72 allows us to prove that a set is countable by describing an enumeration and showing that every element of the set appears somewhere in it, rather than explicitly defining an injective function from the set to the naturals. Finding an enumeration can be more convenient in many cases, as we will now see with some examples.

**Example 74** We show that the set of integers  $\mathbb{Z}$  is countable. Observe that a first attempt of listing the non-negative integers, then the negative integers, does not work:

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

There are infinitely many non-negative integers, so we never actually reach the negative ones. In other words, the above is not a valid enumeration of the integers, because there is no finite (natural number) position at which  $-1$  (or any other negative number) appears.

An attempt that does work is to order the integers by their *absolute values*, which interleaves the positive and negative elements:

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

An arbitrary positive integer  $i$  appears at (zero-indexed) position  $2i - 1$ , and an arbitrary negative integer  $-i$  appears at position  $2i$ . Thus, each integer appears at some finite position in the list.

The above corresponds to the following injective function  $f: \mathbb{Z} \rightarrow \mathbb{N}$ :

$$f(i) = \begin{cases} 2i - 1 & \text{if } i > 0 \\ -2i & \text{otherwise.} \end{cases}$$

Therefore,  $\mathbb{Z}$  is countable, and thus countably infinite.

**Example 75** We now show that the set of positive rational numbers  $\mathbb{Q}^+$  is countable. A positive rational number  $q \in \mathbb{Q}^+$  can be written as the ratio  $x/y$  of a pair of positive integers  $x, y \in \mathbb{N}^+$ . So we start by showing that the set of pairs of natural numbers  $\mathbb{N} \times \mathbb{N}$  is countable.

We use a similar idea to what we did to show that  $\mathbb{Z}$  is countable, i.e., demonstrating an interleaving of the elements that results in a valid enumeration. To get some inspiration, we can write down the pairs in two dimensions:

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>1</b>	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
<b>2</b>	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
<b>3</b>	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
<b>4</b>	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
<b>5</b>	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

To demonstrate an injective mapping to  $\mathbb{N}$ , however, we need to describe a one-dimensional list. Observe that we can do so by listing pairs  $(x, y)$  in order by the “anti-diagonals” satisfying  $x + y = k$  for  $k = 2, 3, 4, \dots$ . Each anti-diagonal has a finite number of elements (specifically,  $k - 1$ ), so for any particular pair, we eventually reach it after listing a finite number of diagonals.

	1	2	3	4	5
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

We can proceed in the same way for  $\mathbb{Q}^+$ . Observe that there are now duplicate elements in the enumeration; for example,  $2/2$  is equal to  $1/1$ . If we wish to, we can avoid duplicates by simply skipping over them when we encounter them. However, Lemma 72 allows an enumeration to contain duplicates, so skipping them is not necessary.

	1	2	3	4	5
1	1/1	1/2	1/3	1/4	1/5
2	2/1	<del>2/2</del>	2/3	<del>2/4</del>	2/5
3	3/1	3/2	<del>3/3</del>	3/4	3/5
4	4/1	<del>4/2</del>	4/3	<del>4/4</del>	4/5
5	5/1	5/2	5/3	5/4	<del>5/5</del>

	1	2	3	4	5
1	1/1	1/2	1/3	1/4	1/5
2	2/1	<del>2/2</del>	2/3	<del>2/4</del>	2/5
3	3/1	3/2	<del>3/3</del>	3/4	3/5
4	4/1	<del>4/2</del>	4/3	<del>4/4</del>	4/5
5	5/1	5/2	5/3	5/4	<del>5/5</del>

Since we can construct an enumeration of the positive rationals  $\mathbb{Q}^+$ , this set is countable.

**Exercise 76** Show that the set  $\mathbb{Q}$  of all rationals (both positive, negative, and zero) is countable. (Hint: use the fact that  $\mathbb{Q}^+$  is countable along with the idea from Example 74.)

## 9.2 Uncountable Sets

So far, we have demonstrated that several infinite sets are countable. But it turns out that not every set is countable; some are *uncountable*. How can we show this? We need to show that there *does not exist* any injective function from the set to the naturals. Equivalently (by Lemma 72), we need to show that there is no list that enumerates every element of the set.

In general, it is challenging to prove that there *does not exist* an object having certain properties. It is not enough to show that several attempts to construct such an object fail, because maybe some other clever attempt we have not thought of yet could work! Instead, we can sometimes prove the non-existence of an object via an indirect route: proof by contradiction. That is, we assume that an object having the properties *does exist*, and then use it to derive a logical contradiction. It follows that no such object exists.

The above strategy is often successful for showing the uncountability of certain sets. As a first example, we prove the following theorem using a technique called *diagonalization*. We assume, for the purpose of contradiction, that an enumeration of the set exists, and then we “use this enumeration against itself” to construct an element of the set that is *not* in the enumeration—a contradiction. The technique is called “diagonalization” because we construct the special element by going “down the diagonal” of the assumed enumeration and making different choices, so that the element differs from each element in the list in at least one position.

**Theorem 77** *The set of real numbers in the interval  $(0, 1)$  is uncountable.*

**Proof 78** Suppose for the sake of establishing a contradiction that this set is countable, which means that there is an enumeration  $r_0, r_1, r_2, \dots$  of the reals in  $(0, 1)$ . If we imagine writing the decimal expansions of these elements as the rows of an infinite table, the result looks something like the following:

$r_0: 0 .$	1	2	1	5	6	6
$r_1: 0 .$	2	3	3	9	9	7
$r_2: 0 .$	4	5	6	7	1	1
$r_3: 0 .$	3	2	8	9	4	5
$r_4: 0 .$	3	4	1	7	7	5
$r_5: 0 .$	4	2	4	3	2	3
$\vdots$					$\ddots$	

The decimal expansion in each row continues indefinitely to the right. If a number has a finite decimal representation, we pad it to the right with infinitely many zeros.

We now use this list to construct a real number  $r^* \in (0, 1)$  in a certain way as follows: we choose the  $i$ th digit of  $r^*$  (zero-indexed, after the decimal point) so that it *differs* from the  $i$ th digit of element  $r_i$  in the list. We also choose these digits to not have an infinite sequence of 0s or 9s, so that  $r^*$  has a *unique* decimal expansion. (This avoids the possibility that  $r^*$  has two different decimal expansions, like  $0.1999\cdots = 0.2000\cdots$ ) For example:

$r_0: 0.$	1	2	1	5	6	6	
$r_1: 0.$	2	3	3	9	9	7	
$r_2: 0.$	4	5	6	7	1	1	...
$r_3: 0.$	3	2	8	9	4	5	
$r_4: 0.$	3	4	1	7	7	5	
$r_5: 0.$	4	2	4	3	2	3	
⋮					⋮		
$r^*: 0.$	2	4	7	1	9	4	?

Here,  $r_0$  has 1 as its 0th digit, so we arbitrarily choose 2 as the 0th digit of  $r^*$ . Similarly,  $r_1$  has 3 as its 1st digit, so we arbitrarily choose 4 as the 1st digit of  $r^*$ ;  $r_2$  has 6 as its 2nd digit, so we arbitrarily choose 7 for the 2nd digit of  $r^*$ , and so on. Thus,  $r^*$  differs from  $r_i$  in the  $i$ th digit.

We now make a critical claim:  $r^*$  does not appear in the assumed enumeration of the elements of  $(0, 1)$ . To see this, consider some arbitrary position  $i$  and the value  $r_i$  that appears there in the enumeration. By construction, the  $i$ th digits of  $r^*$  and  $r_i$  are different, and  $r^*$  has a unique decimal expansion, so  $r^* \neq r_i$ . Since the position  $i$  was arbitrary, this means that  $r^*$  does not appear in the list.

We have therefore arrived at a contradiction, because we assumed at the start that every real number in  $(0, 1)$  appears somewhere in the enumeration, but  $r^* \in (0, 1)$  is a real number that does not. So, our original assumption must be false, and therefore the set of real numbers in  $(0, 1)$  is uncountable, as claimed.  $\square$

**Exercise 79** If we try to adapt the proof of Theorem 77 to prove that the set  $\mathbb{Z}$  of integers is uncountable, where does the attempted proof fail (as it must, because  $\mathbb{Z}$  is countable by Example 74)? Hint: what is different about the decimal expansions of real numbers versus integers?

As we will see next, diagonalization is a very powerful and general technique for proving not just the uncountability of certain sets, but also the undecidability of certain languages.

## 9.3 The Existence of an Undecidable Language

Returning to our original question of how the set of machines  $\mathcal{M}$  is related to the set of languages  $\mathcal{L}$ , we can show using the above techniques that  $\mathcal{M}$  is countable, whereas  $\mathcal{L}$  is uncountable. We can use this to show that there are languages that are not decided by any Turing machine, i.e., they are undecidable. In fact, we can even show that there are **uncountably more** undecidable languages than decidable languages. Thus, in a sense that can be made formal, “almost all” languages are undecidable!

**Exercise 80** Use diagonalization to show that the set of all languages  $\mathcal{L}$  is uncountable, and formalize and rigorously prove the above claims.

We leave the formalization of these countability arguments as an exercise, and instead give a direct diagonalization-

based proof that there is an undecidable language. For concreteness, we work with machines and languages having the input alphabet  $\Sigma = \{0, 1\}$ , but all the arguments straightforwardly generalize to any finite alphabet  $\Sigma$ .

To set up the proof, we first show that the set of input strings, and the set of Turing machines, are both countable.

**Lemma 81** *The (infinite) set  $\{0, 1\}^*$  of binary strings is countable.*

**Proof 82** We can enumerate the binary strings by length, as follows:

$$\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}.$$

This works because there are only finitely many strings of any fixed finite length (one string of length zero, two strings of length one, four strings of length two, etc.), and each string in  $\{0, 1\}^*$  has some finite length, so it will eventually appear in the enumeration.  $\square$

**Lemma 83** *The (infinite) set  $\mathcal{M}$  of Turing machines is countable.*

**Proof 84** The key observation is that any Turing machine  $M$  has a *finite* description, and hence can be encoded as a (finite) binary string  $\langle M \rangle \in \Sigma^*$  in some unambiguous way. To see this, notice that all the components of the seven-tuple are finite: the alphabets  $\Sigma, \Gamma$ , the set of states  $Q$  and the special states  $q_{\text{start}}, q_{\text{acc}}, q_{\text{rej}}$ , and the transition function  $\delta$ . In particular,  $\delta$  has a finite domain and codomain, so we can encode its list of input/output pairs as a (finite) binary string.

Since there is an injective encoding function from  $\mathcal{M}$  to  $\Sigma^*$ , and  $\Sigma^*$  is countable by Lemma 81,  $\mathcal{M}$  is countable as well. (See Exercise 70 to justify this rigorously.)  $\square$

We can now state and prove our main theorem.

**Theorem 85** *There is an undecidable language  $A \subseteq \Sigma^* = \{0, 1\}^*$ .*

**Proof 86** We proceed by diagonalization (but directly, not by contradiction). As shown above, both  $\Sigma^* = \{x_0, x_1, x_2, \dots\}$  and the set of Turing machines  $\mathcal{M} = \{M_0, M_1, M_2, \dots\}$  are countable. So, we can imagine an infinite, two-dimensional table with machines enumerated along the rows, and input strings along the columns:

	$\varepsilon$	0	1	00	01	10	11	000	...
$M_0$	yes	no	yes	yes	no	no	no	no	
$M_1$	yes	yes	no	no	no	no	no	yes	
$M_2$	yes	no	no	no	yes	no	no	yes	
$M_3$	yes	no	yes	no	yes	yes	no	no	
$M_4$	yes	no	yes	no	yes	no	no	yes	
$M_5$	yes	yes	no	no	no	no	yes	yes	
$M_6$	no	yes	no	no	yes	no	no	no	
$M_7$	yes	no	no	yes	no	yes	no	yes	
$\vdots$									$\ddots$

The  $(i, j)$ th entry of this table indicates whether machine  $M_i$  accepts input string  $x_j$ . For example, we have here that  $M_0$  accepts the string  $x_0 = \varepsilon$  but does not accept the string  $x_1 = 0$ , whereas  $M_1$  accepts both of those strings but does not accept the string  $x_2 = 1$ .

Now consider the *diagonal* of this table, which indicates whether machine  $M_i$  accepts input string  $x_i$ , for each  $i \in \mathbb{N}$ . By definition,  $x_i \in L(M_i)$  if and only if the  $i$ th diagonal entry in the table is “yes”.

We now construct the language  $A \subseteq \Sigma^*$  to correspond to the “negated diagonal.” Specifically, for each string  $x_i$ , we include it in  $A$  if and only if  $M_i$  does *not* accept  $x_i$ . Formally:

$$A = \{x_i : x_i \notin L(M_i), \text{ for } i \in \mathbb{N}\}.$$

$A$			1	00		10	11		...
-----	--	--	---	----	--	----	----	--	-----

By the above definition, for every  $i \in \mathbb{N}$ , we have that  $x_i \in A$  if and only if  $x_i \notin L(M_i)$ , so  $A \neq L(M_i)$ . Therefore, no machine  $M_i$  in the enumeration decides  $A$ . Since  $M_0, M_1, M_2, \dots$  enumerates *every* Turing machine, we conclude that *no machine* decides  $A$ —it is undecidable.

(In fact, we have shown even more: since  $A \neq L(M_i)$  for all  $i \in \mathbb{N}$ , there is no Turing machine that even *recognizes*  $A$ , i.e.,  $A$  is unrecognizable. See [Recognizability](#) (page 123) for details.)  $\square$

The above proof establishes the existence of an undecidable language, but the language is rather contrived: it is constructed based on the enumerations of machines and inputs, and the behaviors of these machines on these inputs. The associated decision problem does not seem like a “natural” problem we would care to solve in the first place, so perhaps it isn’t so important that this problem is unsolvable.

In the upcoming sections, we will prove that many quite natural and practically important languages of interest are also undecidable.

## “NATURAL” UNDECIDABLE PROBLEMS

I don't care to belong to any club that will have me as a member. — Groucho Marx

Suppose that while visiting a new town, you come across a barber shop with the following sign in its window:

Barber  $B$  is the best barber in town!  $B$  cuts the hair of all those people in town, and only those, who do not cut their own hair.

In other words, for any person  $X$  in the town, there are two possibilities:

1. If  $X$  cuts their own hair, then  $B$  does not cut  $X$ 's hair.
2. If  $X$  does not cut their own hair, then  $B$  cuts  $X$ 's hair.

Assuming that the sign is true, we now ask the question: does  $B$  cut their own hair? Since the barber is a person in the town, we can substitute  $X = B$  into the two cases above, and get:

1. If  $B$  cuts their own hair, then  $B$  does not cut  $B$ 's hair.
2. If  $B$  does not cut their own hair, then  $B$  cuts  $B$ 's hair.

Both cases result in a contradiction! Thus, our assumption that the sign is true must be incorrect. (Or perhaps the barber is not a person...)

This is known as the *barber paradox*. While its current form may seem like an idle amusement, in what follows we will see that we can devise an analogous paradox for Turing machines, which will yield a “natural” undecidable language. Then, we will use this undecidable language to show that there are many other natural and practically important undecidable languages.

### 10.1 Code as Input

Recall from the proof of Lemma 83 that any Turing machine itself can be unambiguously represented as a binary string, by encoding the (finite) components of its seven-tuple. We refer to this string encoding of a machine as its *code*, and denote it as usual with angle brackets:  $\langle M \rangle$  is the code of machine  $M$ .

Next, because the code of a machine is just a binary string, we can contemplate the following fascinating idea: we can give the **code of one machine as input to another machine!** We can even give a program *its own code* as input.

There are many useful examples of this kind of pattern in everyday computing:

- An *interpreter* is a program that takes the code of some arbitrary program as input, and runs it.
- A *compiler* is a program that takes the code of some arbitrary program as input, and converts it to some other form (e.g., machine-level instructions).
- A *debugger* is a program that takes the code of some arbitrary program as input, and runs it interactively under the control of the user (e.g., step by step, or until certain conditions on the state of the program are met, etc.).

- An *emulator* is a program that takes the code of some arbitrary program written for a certain kind of hardware device, and runs it on some other hardware device.

Here are some examples of how in practice, a C++ program could take its own code as input:

```
$ g++ prog.cpp -o prog.exe      # compile the program into an executable
$ ./prog.exe prog.cpp          # pass the code filename as a command-line argument
$ ./prog.exe < prog.cpp        # pass the code via standard input
$ ./prog.exe ``cat prog.cpp``   # pass the code as a single command-line argument
```

In the above example, we first passed the code `prog.cpp` as the input to the `g++` compiler. The `g++` compiler itself is an example of a [self-hosting<sup>22</sup>](#) program: it is compiled by passing its own code to itself! A [quine<sup>23</sup>](#) is an analogous concept of a program that *outputs* its own code.

## 10.2 The Barber Language

With the concept of code as input in hand, we now devise a computational analog of the barber paradox. The analogy arises from the following correspondences:

- instead of *people*, we consider *Turing machines*;
- instead of the *hair* of a person, we consider the **code** of a TM;
- instead of *cutting* the hair of a person, we consider *accepting* the code of a TM.

The advertisement in the barber shop then becomes:

*Turing machine B* is the best TM! It accepts the **code** of all *Turing machines*, and only those, that do not accept their own **code**.

More formally, the above says that the language  $L(B)$  of machine  $B$  is the following “barber” language:

$$L_{\text{BARBER}} = \{\langle M \rangle : M \text{ is a TM and } M(\langle M \rangle) \text{ does not accept}\}.$$

This is because  $B$  accepts the code  $\langle M \rangle$  of a Turing machine  $M$  if and only if  $M$  does not accept its own code  $\langle M \rangle$ .

Does such a Turing machine  $B$  exist? The following theorem proves that is does not, because its very existence would be a contradiction.

**Theorem 87** *The language  $L_{\text{BARBER}}$  is undecidable (and even unrecognizable), i.e., no Turing machine decides (or even recognizes)  $L_{\text{BARBER}}$ .*

**Proof 88** Assume for the sake of contradiction that there is a Turing machine  $B$  for which  $L(B) = L_{\text{BARBER}}$ . Just like we asked whether the barber cuts their own hair, let us now ask the analogous question: does  $B$  accept its own code, i.e., does  $B(\langle B \rangle)$  accept? (Since  $B$  is a Turing machine, it has some code  $\langle B \rangle$ , and the behavior of  $B(\langle B \rangle)$  is well defined: it either accepts or it does not.)

1. If  $B(\langle B \rangle)$  accepts, then  $\langle B \rangle \notin L_{\text{BARBER}}$  by definition, so by our assumption,  $B(\langle B \rangle)$  does not accept.
2. If  $B(\langle B \rangle)$  does not accept, then  $\langle B \rangle \in L_{\text{BARBER}}$  by definition, so by our assumption,  $B(\langle B \rangle)$  accepts.

Again, both cases result in a contradiction. Thus, our initial assumption was incorrect: no Turing machine  $B$  has the property that  $L(B) = L_{\text{BARBER}}$ , i.e., no Turing machine decides  $L_{\text{BARBER}}$ —it is undecidable. In fact, we have proved even more: that no Turing machine even *recognizes*  $L_{\text{BARBER}}$ .  $\square$

<sup>22</sup> [https://en.wikipedia.org/wiki/Self-hosting\\_\(compilers\)](https://en.wikipedia.org/wiki/Self-hosting_(compilers))

<sup>23</sup> [https://en.wikipedia.org/wiki/Quine\\_\(computing\)](https://en.wikipedia.org/wiki/Quine_(computing))

It is interesting to observe that the definition of  $L_{\text{BARBER}}$  and proof of [Theorem 87](#) can also be seen as a kind of diagonalization. Since the set of TMs is countable by [Lemma 83](#), there is an enumeration  $M_0, M_1, M_2, \dots$  of all Turing machines. Consider a two-dimensional infinite “table” with this enumeration along the rows, and the *codes*  $\langle M_0 \rangle, \langle M_1 \rangle, \langle M_2 \rangle, \dots$  along the columns. Entry  $(i, j)$  of the table corresponds to whether  $M_i(\langle M_j \rangle)$  accepts. The language corresponding to the “negated” diagonal, which is therefore not decided (or even recognized) by any TM, consists of exactly those strings  $\langle M_i \rangle$  for which  $M_i(\langle M_i \rangle)$  does not accept—and this is exactly the definition of  $L_{\text{BARBER}}$ !

The undecidable language  $L_{\text{BARBER}}$  is fairly “natural”, because it corresponds to the decision problem of determining whether a given program accepts its own code. (As we have seen, running a program on its own code is a natural and potentially useful operation.) Next, we will build upon the undecidability of  $L_{\text{BARBER}}$  to prove the undecidability of other, arguably even more “natural” languages that are concerned with the behavior of given programs on inputs that are typically not their own code.

## 10.3 The Acceptance Language and Simulation

We now define the language that corresponds to the following decision problem: given a Turing machine and a string, does the machine accept the string?

**Definition 89 (Acceptance Language for TMs)** The “acceptance” language for Turing machines is defined as

$$L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ accepts}\}.$$

Unlike the undecidable languages considered above, this language is *recognizable* by a certain Turing machine  $U$ , as we now explain. Given input  $(\langle M \rangle, x)$ , machine  $U$  examines the code of the input machine  $M$  and “simulates” running whatever steps that machine would take when run on the input string  $x$ , finally outputting the same decision (if it ever halts).

An *interpreter* is a real-world incarnation of this concept. For example, we can provide the source code of a Python program to the Python interpreter, which will read and execute the code:

```
$ python prog.py
```

Similarly, a general-purpose CPU can be seen as an interpreter: it is a fixed “program” (in this case, hardware) that, when given the source code (in this case, machine-level instructions) of some program, runs that program. An emulator can also be seen as a kind of interpreter.

A Turing machine  $U$  that is an interpreter of this kind is known as a *universal Turing machine*. Observe that  $U(\langle M \rangle, x)$  has the following behavior:

- If  $M(x)$  accepts, then  $U(\langle M \rangle, x)$  accepts.
- If  $M(x)$  rejects, then  $U(\langle M \rangle, x)$  rejects.
- If  $M(x)$  loops, then  $U(\langle M \rangle, x)$  loops.

Therefore,  $U$  recognizes  $L_{\text{ACC}}$  (i.e.,  $L(U) = L_{\text{ACC}}$ ), because it accepts every  $(\langle M \rangle, x) \in L_{\text{ACC}}$  and rejects or loops on every  $(\langle M \rangle, x) \notin L_{\text{ACC}}$ . (See [Definition 62](#).)

There are many examples of universal Turing machines in the literature, and their descriptions can be found elsewhere. For our purposes, what is important is the existence of universal machines, not the details of their implementation.

### Simulation Versus Subroutines

The idea of a machine  $U$  *simulating* the execution of a machine  $M$  whose code  $\langle M \rangle$  is provided as *input* to  $U$  is **quite different** from what we did in the proof of [Lemma 65](#). In that proof,  $M_1, M_2$  are *fixed machines* that exist by

hypothesis, and the constructed machine  $M$  has  $M_1, M_2$  “built in” to it as “*subroutines*”. For Turing machines, this can be done by including the states and transitions of the subroutine machine as part of the calling machine, with appropriate transitions between certain states of the two machines to represent the subroutine call and return value.

In practice, the analogous concept is using a library of pre-existing code. For example, in C++, we can use the `#include` directive, like so:

```
#include "M1.hpp" // include code of M1
#include "M2.hpp" // include code of M2

int M(string x) {
    M1(x);           // invoke M1
    ...
}
```

Now that we have shown that the language  $L_{\text{ACC}}$  is *recognizable*, the natural next question is whether it is *decidable*. For it to be decidable, there must exist some Turing machine  $C$  that has the following behavior on input  $(\langle M \rangle, x)$ :

- If  $M$  accepts  $x$ , then  $C(\langle M \rangle, x)$  accepts.
- If  $M$  does not accept  $x$  (i.e., it either rejects or loops), then  $C(\langle M \rangle, x)$  rejects.

Observe that a universal Turing machine  $U$  does not meet these requirements, because if  $M$  loops on  $x$ , then  $U(\langle M \rangle, x)$  loops; it does not reject. Thus, even though  $L(U)$  recognizes  $L_{\text{ACC}}$ , it loops on some inputs and therefore does not decide  $L_{\text{ACC}}$  (or any other language). In fact, it turns out that *no* Turing machine decides  $L_{\text{ACC}}$ .

**Theorem 90** *The language  $L_{\text{ACC}}$  is undecidable.*

**Proof 91** Assume for the sake of contradiction that there exists a Turing machine  $C$  that decides the language  $L_{\text{ACC}}$ . Below we use  $C$  to define another Turing machine  $B$  that decides the “barber” language  $L_{\text{BARBER}}$ . Since we previously showed that  $L_{\text{BARBER}}$  is undecidable (see [Theorem 87](#)), this is a contradiction, and our original assumption was false. Hence no Turing machine decides  $L_{\text{ACC}}$ , i.e.,  $L_{\text{ACC}}$  is undecidable.

We wish to define a Turing machine  $B$  that, when given the code  $\langle M \rangle$  of some arbitrary machine  $M$  as input, determines whether  $M(\langle M \rangle)$  accepts. The key idea is that  $B$  can use the machine  $C$  to determine this, because  $C$  decides  $L_{\text{ACC}}$  by assumption: given  $(\langle M' \rangle, x)$  for *any* Turing machine  $M'$  and *any* string  $x$  as input,  $C$  will correctly determine whether  $M'(x)$  accepts. So, for  $B$  to achieve its goal, it will invoke  $C$  on the machine  $M' = M$  and input string  $x = \langle M \rangle$ . The precise definition of  $B$  is as follows:

```
function B(⟨M⟩)
    if C(⟨M⟩, ⟨M⟩) accepts then reject
    accept
```

We analyze the behavior of  $B$  on an arbitrary input string  $\langle M \rangle$  as follows:

- If  $\langle M \rangle \in L_{\text{BARBER}}$ , then  $M(\langle M \rangle)$  does not accept by definition of  $L_{\text{BARBER}}$ , so  $(\langle M \rangle, \langle M \rangle) \notin L_{\text{ACC}}$  by definition of  $L_{\text{ACC}}$ , so  $C(\langle M \rangle, \langle M \rangle)$  rejects because  $C$  decides  $L_{\text{ACC}}$ , so  $B(\langle M \rangle)$  accepts by construction of  $B$ , as needed.
- Conversely, if  $\langle M \rangle \notin L_{\text{BARBER}}$ , then  $M(\langle M \rangle)$  accepts by definition of  $L_{\text{BARBER}}$ , so  $(\langle M \rangle, \langle M \rangle) \in L_{\text{ACC}}$  by definition of  $L_{\text{ACC}}$ , so  $C(\langle M \rangle, \langle M \rangle)$  accepts because  $C$  decides  $L_{\text{ACC}}$ , so  $B(\langle M \rangle)$  rejects by construction of  $B$ , as needed.

Therefore, by [Definition 61](#),  $B$  decides  $L_{\text{BARBER}}$ , as claimed.

Alternatively, we can analyze  $B$  using the equivalent form of [Definition 61](#), as follows. First, because  $B$  simply calls  $C$  as a subroutine and outputs the opposite answer, and  $C$  halts on any input (because  $C$  is a decider by

hypothesis),  $B$  also halts on any input. Next, by definitions of the languages and assumption on  $C$ ,

$$\begin{aligned} \langle M \rangle \in L_{\text{BARBER}} &\iff M(\langle M \rangle) \text{ does not accept} \\ &\iff (\langle M \rangle, \langle M \rangle) \notin L_{\text{ACC}} \\ &\iff C(\langle M \rangle, \langle M \rangle) \text{ rejects} \\ &\iff B(\langle M \rangle) \text{ accepts} , \end{aligned}$$

as required.  $\square$

Observe that in the proof of [Theorem 90](#), we showed that the (previously established) undecidability of  $L_{\text{BARBER}}$  implies the undecidability of the new language  $L_{\text{ACC}}$ . We did this by contradiction, or from another perspective, by establishing the *contrapositive* statement: that if  $L_{\text{ACC}}$  is decidable, then  $L_{\text{BARBER}}$  is decidable.

We proved this by using any hypothetical Turing machine  $C$  that decides  $L_{\text{ACC}}$  as a *subroutine* to construct a Turing machine  $B$  that decides  $L_{\text{BARBER}}$ . Importantly,  $B$  did *not* simulate its input machine  $M$  on the code  $\langle M \rangle$ , because doing this might cause  $B$  to loop. Instead,  $B$  “asked”  $C$  whether  $M(\langle M \rangle)$  accepts, and negated the answer. We do not know *how*  $C$  determines its answer, but this does not matter:  $C$  decides  $L_{\text{ACC}}$  by hypothesis, so it halts and returns the correct answer on any input it is given.

A transformation that uses a hypothetical decider for one language as a subroutine to construct a decider for another language is called a *Turing reduction*. This is a very powerful and general approach for relating the (un)decidability of one language to another. More generally, reductions of various kinds are one of the most important and central ideas in theoretical computer science. See the next section and [Turing Reductions](#) (page 106) for further examples and details.

## 10.4 The Halting Problem

Recall that the only difference between a universal Turing machine  $U$  (which we know exists) and a decider for  $L_{\text{ACC}}$  (which we proved does not exist) is how they behave on inputs  $(\langle M \rangle, x)$  where  $M$  loops on  $x$ : while  $U(\langle M \rangle, x)$  also loops, a decider for  $L_{\text{ACC}}$  must reject such  $(\langle M \rangle, x)$ . On all other inputs,  $U$  halts and returns the correct answer. So, if we could just first determine whether  $M(x)$  halts, we could decide  $L_{\text{ACC}}$ . The decision problem of determining whether a given machine  $M$  halts on a given input  $x$  is called the *halting problem*. Since a decider for the halting problem would yield a decider for  $L_{\text{ACC}}$ , this implies that the halting problem must be undecidable as well!

We now formalize this reasoning. We define  $L_{\text{HALT}}$ , the language corresponding to the halting problem, as follows.

**Definition 92 (Halting Language for TMs)** The “halting” language for Turing machines is defined as

$$L_{\text{HALT}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ halts}\} .$$

So:

- If  $M(x)$  accepts, then  $(\langle M \rangle, x) \in L_{\text{HALT}}$ .
- If  $M(x)$  rejects, then  $(\langle M \rangle, x) \in L_{\text{HALT}}$ .
- If  $M(x)$  loops, then  $(\langle M \rangle, x) \notin L_{\text{HALT}}$ .

**Theorem 93** *The language  $L_{\text{HALT}}$  is undecidable.*

**Proof 94** Assume for the sake of contradiction that there exists a Turing machine  $H$  that decides the language  $L_{\text{HALT}}$ . Below we use  $H$  to define another Turing machine  $C$  that decides  $L_{\text{ACC}}$ . Since we previously showed that  $L_{\text{ACC}}$  is undecidable (see [Theorem 90](#)), this is a contradiction, and our original assumption was false. Hence no Turing machine decides  $L_{\text{HALT}}$ , i.e.,  $L_{\text{HALT}}$  is undecidable.

We wish to define a Turing machine  $C$  that, when given input  $(\langle M \rangle, x)$  for some arbitrary machine  $M$  and arbitrary

string  $x$ , determines whether  $M(x)$  accepts. As discussed above,  $C$  could simulate  $M(x)$ , as in a universal Turing machine. However, this might loop, which is the only case that  $C$  must avoid. The key idea is that  $C$  can first use  $H$  to determine whether  $M(x)$  loops, and then act appropriately. The precise definition of  $C$  is as follows:

```
function  $C(\langle M \rangle, x)$ 
  if  $H(\langle M \rangle, x)$  rejects then reject
  simulate  $M(x)$  and output the same result
```

We analyze the behavior of  $C$  on an arbitrary input string  $(\langle M \rangle, x)$  as follows:

- If  $(\langle M \rangle, x) \in L_{\text{ACC}}$ , then  $M(x)$  accepts and thus halts, so  $(\langle M \rangle, x) \in L_{\text{HALT}}$  and thus the call to  $H(\langle M \rangle, x)$  accepts, so  $C$  reaches its second line, where the simulation of  $M(x)$  accepts, so  $C$  accepts, as needed.
- If  $(\langle M \rangle, x) \notin L_{\text{ACC}}$ , then  $M(x)$  either rejects or loops.
  - If it loops, then  $(\langle M \rangle, x) \notin L_{\text{HALT}}$  so the call to  $H(\langle M \rangle, x)$  rejects, and  $C$  rejects, as needed.
  - If it rejects, then  $(\langle M \rangle, x) \in L_{\text{HALT}}$  so the call to  $H(\langle M \rangle, x)$  accepts, so  $C$  reaches its second line, where the simulation of  $M(x)$  rejects, so  $C$  rejects, again as needed.

Therefore, by Definition 61,  $C$  decides  $L_{\text{ACC}}$ , as claimed. □

We have proved that  $L_{\text{HALT}}$  is undecidable. This is quite unfortunate, since the halting problem is a fundamental problem in software and hardware design. One of the most basic questions we can ask about a program's correctness is whether it halts on all inputs. Since  $L_{\text{HALT}}$  is undecidable, there is no general method to answer this question even for a *single* input, much less all inputs.

**Exercise 95** Construct an elementary proof that  $L_{\text{HALT}}$  is undecidable by following the steps below. This was the first “existence” proof for an undecidable language, and it is due to Alan Turing.

- a) Suppose there exists a Turing Machine  $H$  that decides  $L_{\text{HALT}}$ . Using  $H$ , design a Turing Machine  $M$  for which  $M(\langle T \rangle)$  halts if and only if  $H(\langle T \rangle, \langle T \rangle)$  rejects, for any Turing machine  $T$ .
- b) Devise an input  $x$  for  $M$  that yields a contradiction, and conclude that  $H$  does not exist, i.e.,  $L_{\text{HALT}}$  is undecidable.

## TURING REDUCTIONS

Observe that the above proofs of the undecidability of the acceptance language  $L_{\text{ACC}}$  (Theorem 90) and the halting language  $L_{\text{HALT}}$  (Theorem 93) follow a common pattern for showing that some language  $B$  is undecidable:

1. First assume for the sake of contradiction that  $B$  is decidable, i.e., there is some Turing machine  $M_B$  that decides it.
2. Use  $M_B$  as a subroutine to construct a decider  $M_A$  for some *known undecidable* language  $A$ .
3. Since  $M_A$  decides the undecidable language  $A$ , we have reached a contradiction. So the original assumption that  $B$  is decidable must be false, hence  $B$  is undecidable.

This kind of pattern is known as a *reduction* from language  $A$  to language  $B$ . That is, we *reduce* the task of solving  $A$  to that of solving  $B$ , so that if  $B$  is solvable, then  $A$  is solvable as well. We next formalize and abstract out this pattern, and apply it to show the undecidability of more languages.

### Libraries of Code

In practice, almost all programs make use of pre-existing components, often called *libraries*, to accomplish their intended tasks. As an example, consider a “hello world” program in C:

```
#include <stdio.h>

int main() {
    printf("Hello world!");
    return 0;
}
```

This program uses the standard `stdio.h` library, invoking the `printf()` function from that library. Observe that we do not need to know *how* `printf()` works to write this program—we need only know *what* it does (i.e., its input-output behavior). As long as the function does what it is claimed to do, the above program will work correctly—even if the implementation of `printf()` were to change to a completely different (but still correct) one. In other words, the program merely treats `printf()` as a “*black box*” that prints its argument.

From a certain perspective, the above program *reduces* the task of printing the *specific* string `Hello world!` to that of printing an *arbitrary* string. Since the latter task is accomplished by the `printf()` function, we can use it rather trivially to accomplish the former task.

**Definition 96 (Turing Reduction)** A *Turing reduction* from a language  $A$  to a language  $B$  is a Turing machine that decides  $A$  given access to an *oracle* (or “black box”) that decides  $B$ . If such a reduction exists, we say that  $A$  *Turing-reduces* to  $B$ , and write  $A \leq_T B$ .<sup>24</sup>

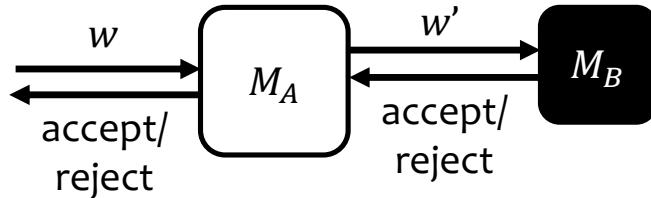
<sup>24</sup> As discussed more below, the “direction” of the reduction is *very* important. Upon first exposure (or even after many years of exposure) to this concept, the terminology “ $A$  reduces to  $B$ ” may appear to be in conflict with the notation  $A \leq_T B$ . The phrase “ $A$  reduces to  $B$ ”

refers to the problems’ “solvability”, and essentially means that solving  $A$  “comes down to” solving  $B$ . For example, addition of multi-digit numbers “reduces to” addition of single digits, because we can accomplish the former via the latter. By contrast, the notation  $A \leq_T B$  refers to the problems’ intrinsic “hardness”, and essentially says that  $A$  is “no harder to solve” than  $B$  is (ignoring efficiency).

An *oracle* (or “black box”) that decides a language is some abstract entity that, whenever it is invoked (or *queried*) on any string, it correctly identifies whether that string is in the language—it accepts if so, and rejects if not. Importantly, it does not loop on any input—a useful property that reductions often exploit.

We can think of an oracle as a subroutine or “library function” (see *the sidebar* (page 106)) that correctly performs its stated task on any input. However, it is *not necessarily*—and in many advanced settings, *is not*—a Turing machine. Oracles can potentially perform tasks that no Turing machine can do, like decide undecidable problems.

A reduction may query its oracle any (finite) number of times, on any string(s) of its choice—including none at all. However, the reduction may not “look inside” the oracle or rely in any way on *how* it works internally—it can use the oracle only as a “black box” subroutine.



The notation  $A \leq_T B$  reflects the intuition that, ignoring efficiency, problem  $A$  is “no harder to solve” than problem  $B$  is: if we can somehow solve  $B$ , then we can also solve  $A$ . However, we emphasize that the statement  $A \leq_T B$  does not require that  $B$  is *actually solvable* by an algorithm. It is merely a *conditional* statement that, if we have access to some *hypothetical* solver for  $B$  (as an oracle), then we can use it to solve  $A$  by an algorithm. Consistent with this intuition, one can show that  $\leq_T$  is *transitive*:

**Exercise 97** Formally prove that if  $A \leq_T B$  and  $B \leq_T C$ , then  $A \leq_T C$ .

We now prove the first formal implication of having a Turing reduction, which conforms to the above intuition.

**Lemma 98** Suppose that  $A \leq_T B$ . If  $B$  is decidable, then  $A$  is also decidable.

**Proof 99** Since  $A \leq_T B$ , by Definition 96 there is a Turing machine  $M_A$  that decides  $A$  when given access to an oracle that decides  $B$ . Since  $B$  is decidable, there is a Turing machine  $M_B$  that decides  $B$ . Thus, in the reduction  $M_A$ , we can *implement* its oracle using (the code of)  $M_B$ . This results in an ordinary Turing machine that decides  $A$  (without relying on any oracle), so  $A$  is indeed decidable.  $\square$

The following important corollary is merely the contrapositive of Lemma 98:<sup>25</sup>

**Lemma 100** Suppose that  $A \leq_T B$ . If  $A$  is undecidable, then  $B$  is also undecidable.

Lemma 100 is a powerful tool for proving that a language  $B$  is undecidable: it suffices to identify some other language  $A$  that is already known to be undecidable, and then prove that  $A \leq_T B$ . In particular, we do not need to set up and repeat all the surrounding “boilerplate” structure of a proof by contradiction. However, it is critical that we establish the reduction in the proper “*direction*”! That is, we must show how to decide  $A$  using an oracle for  $B$ , not the other way around.

Observe that our proofs of Theorem 90 and Theorem 93 above actually showed that  $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$  and  $L_{\text{ACC}} \leq_T L_{\text{HALT}}$ , respectively. This is because we constructed a Turing machine  $B$  that decides  $L_{\text{BARBER}}$  using *any* hypothetical

<sup>25</sup> We can also prove this by contradiction, since contraposition and contradiction are closely related. We are given that  $A \leq_T B$  and that  $A$  is undecidable. Suppose for the purposes of contradiction that  $B$  is decidable. Then by Lemma 98,  $A$  is also decidable, which contradicts what was given. So, our assumption about  $B$  is false, i.e.,  $B$  is undecidable.

decider  $C$  for  $L_{\text{ACC}}$  as an oracle (i.e., a black-box subroutine); similarly, we constructed a Turing machine  $C$  that decides  $L_{\text{ACC}}$  using any hypothetical decider  $H$  for  $L_{\text{HALT}}$  as an oracle.

Given that  $A \leq_T B$ , there are four possibilities for the (un)decidability of  $A$  or  $B$ , but only two of them imply anything about the (un)decidability of the other language:

Hypothesis	Implies
$A$ is decidable	nothing
$A$ is undecidable	$B$ is undecidable
$B$ is decidable	$A$ is decidable
$B$ is undecidable	nothing

In general, the other two possibilities don't imply anything about the decidability of the other language. In fact, we can show the following:

**Exercise 101** Prove that, if  $A$  is decidable, then  $A \leq_T B$  for any language  $B$ , regardless of whether  $B$  is decidable or not. (Hint: a reduction is not required to use its oracle.)

The fourth case, where  $B$  is undecidable, is more subtle. It is **not** the case that  $A \leq_T B$  for any language  $A$ . In fact, there is a hierarchy of “degrees” of undecidable languages<sup>26</sup>, and there are even pairs of “incomparable” undecidable languages<sup>27</sup>  $A, B$  for which neither  $A \leq_T B$  nor  $B \leq_T A$  holds. These topics are beyond the scope of this text.

## 11.1 The Halts-on-Empty Problem

We saw previously that the halting-problem language  $L_{\text{HALT}}$  is undecidable. In this section we show that a more restricted form of this language is also undecidable, via a Turing reduction that employs a very useful “hard-coding” technique. The restricted language corresponds to the decision problem of determining whether a given machine halts *on the empty-string input*.

**Definition 102 (Halts-on- $\varepsilon$  Language for TMs)** The “halts on the empty string” language for TMs is defined as

$$L_{\varepsilon\text{-HALT}} = \{\langle M \rangle : M \text{ is a Turing machine and } M(\varepsilon) \text{ halts}\}.$$

It is not too hard to see that  $L_{\varepsilon\text{-HALT}} \leq_T L_{\text{HALT}}$ , i.e., given access to an oracle  $H$  that decides  $L_{\text{HALT}}$ , there is a Turing machine  $E$  that decides  $L_{\varepsilon\text{-HALT}}$ :

```
function E(<math>\langle M \rangle</math>)
    return <math>H(\langle M \rangle, \varepsilon)</math>
```

(We leave the routine analysis as an exercise.) However, as we can see from the table above, this implies nothing about the decidability of  $L_{\varepsilon\text{-HALT}}$ , because  $L_{\text{HALT}}$  is undecidable. Instead, to show that  $L_{\varepsilon\text{-HALT}}$  is undecidable, we should Turing-reduce *from* some undecidable language  $A$  to  $L_{\varepsilon\text{-HALT}}$ , i.e., we should prove that  $A \leq_T L_{\varepsilon\text{-HALT}}$ .

Below we formally prove that  $L_{\text{HALT}} \leq_T L_{\varepsilon\text{-HALT}}$ , after discussing the key issues here first. Let  $E$  be an oracle that decides  $L_{\varepsilon\text{-HALT}}$ , i.e.:

- $E$  halts on any input, and
- $M'(\varepsilon)$  halts if and only if  $E(\langle M' \rangle)$  accepts, for any Turing machine  $M'$ .

(See the equivalent form of [Definition 61](#), which is more convenient for the present treatment.)

<sup>26</sup> [https://en.wikipedia.org/wiki/Turing\\_degree](https://en.wikipedia.org/wiki/Turing_degree)

<sup>27</sup> [https://en.wikipedia.org/wiki/Turing\\_degree#Order\\_properties](https://en.wikipedia.org/wiki/Turing_degree#Order_properties)

To show that  $L_{\text{HALT}} \leq_T L_{\varepsilon\text{-HALT}}$ , we need to construct a reduction (a Turing machine)  $H$  that decides  $L_{\text{HALT}}$  given access to the oracle  $E$ . Specifically, we need it to be the case that:

- $H$  halts on any input, and
- $M(x)$  halts if and only if  $H(\langle M \rangle, x)$  accepts, for any Turing machine  $M$  and string  $x$ .

Hence, we need to design the reduction  $H$  to work for an input *pair*—a machine  $M$  and a string  $x$ —even though its oracle  $E$  takes only one input—a machine  $M'$ . So, we wish to somehow transform the original  $M$  and  $x$  into just a machine  $M'$  on which to query  $E$ , so that  $E$ 's (correct) answer on  $M'$  will reveal the correct answer for the original inputs  $M$  and  $x$ .

A very useful strategy in this kind of circumstance is to have the reduction “**hard-code** the original inputs into the code of a related new machine that the oracle can properly handle. In the present case, the reduction will construct the code for a new program  $M'$  that ignores its own input and simply runs  $M$  on  $x$ . Importantly, the code for this  $M'$  can be constructed from the code for  $M$  and  $x$  simply by “syntactic” processing, not *running* any of this code (see *the sidebar* (page 109)). By construction,  $M'$  halts on  $\varepsilon$  (or any other string) if and only if  $M$  halts on  $x$ . Because  $E$  correctly determines whether the former is the case by hypothesis, it also implicitly reveals whether the latter is the case—which is exactly what the reduction needs to determine. We now proceed more formally.

**Theorem 103** We have that  $L_{\text{HALT}} \leq_T L_{\varepsilon\text{-HALT}}$ , so  $L_{\varepsilon\text{-HALT}}$  is undecidable.

**Proof 104** Let  $E$  be an oracle that decides  $L_{\varepsilon\text{-HALT}}$ , which has the properties stated above. We claim that the following Turing machine  $H$  decides  $L_{\text{HALT}}$  given access to  $E$ :

```
function  $H(\langle M \rangle, x)$ 
    construct Turing machine “ $M'(w)$ : ignore  $w$  and return  $M(x)$ ”
    return  $E(\langle M' \rangle)$ 
```

The constructed machine  $M'$  takes an input  $w$ , which it ignores, and then it just runs  $M$  on the original input  $x$ . So, we have the following key property:  $M(x)$  halts if and only if  $M'(w)$  halts on all  $w$ , and in particular, if and only if  $M'(\varepsilon)$  halts.

We now analyze the behavior of  $H$  on an arbitrary input  $(\langle M \rangle, x)$ . First,  $H$  halts on any input, because it just constructs (but does not run!) the code for  $M'$  from  $M$  and  $x$ , and then invokes  $E$ , which halts by hypothesis. Second, by the definition of  $L_{\text{HALT}}$ , the key property above, the hypothesis on  $E$ , and the definition of  $H$ ,

$$\begin{aligned} (\langle M \rangle, x) \in L_{\text{HALT}} &\iff M(x) \text{ halts} \\ &\iff M'(\varepsilon) \text{ halts} \\ &\iff E(\langle M' \rangle) \text{ accepts} \\ &\iff H(\langle M \rangle, x) \text{ accepts} , \end{aligned}$$

as needed. So, by [Definition 61](#),  $H$  decides  $L_{\text{HALT}}$ , as claimed. □

We stress that a very important point about the above proof of [Theorem 103](#) is that the reduction  $H$  **merely constructs**  $M'$  and queries  $E$  on it;  $H$  **does not run/simulate**  $M'$  or  $M$ . This is important because  $M'$  itself runs some arbitrary input code  $M$  that may loop on  $x$ , so we cannot “risk” running it in a machine that we want to decide some language. Instead,  $H$  queries  $E$  on  $\langle M' \rangle$ , which is “safe” because  $E$  halts on any input, by hypothesis. Indeed,  $E$  correctly determines whether  $M'(\varepsilon)$  halts, which holds if and only if  $M(x)$  halts, by construction of  $M'$ .

### Constructing a Program in C++

Suppose that we have a C++ library for an interpreter  $U$ . Then given the C++ code for a program  $M$  and an input string  $x$ , we can easily construct C++ code for a program that ignores its input and just runs  $M$  on  $x$ . The full

definition of the reduction  $H$  (which decides  $L_{\text{HALT}}$ ) given subroutine  $E$  (an oracle that decides  $L_{\varepsilon\text{-HALT}}$ ) is as follows:

```
int H(string M_code, string x) {
    string Mp =
        "#include \"U.hpp\"\n"
        + "int Mprime(string w) {\n"
        + "    return U(\"" + M_code + "\", \"\" + x + "\");\n"
        + "}";
    return E(Mp);
}
```

We hardcode the inputs  $M_{\text{code}}$  and  $x$  as part of the code  $Mp$  for  $Mprime()$ .<sup>28</sup> Since  $M_{\text{code}}$  is code in string format, we cannot run it directly, but we can pass it to the interpreter  $U$  to simulate its execution on  $x$ .

## 11.2 More Undecidable Languages and Turing Reductions

Here we define a variety of other languages and prove them undecidable, using the reduction techniques introduced above.

**Example 105** Define

$$L_{\text{FOO}} = \{\langle M \rangle : M \text{ is a Turing machine and } foo \in L(M)\}.$$

We show that  $L_{\text{ACC}} \leq_T L_{\text{FOO}}$ , hence that  $L_{\text{FOO}}$  is undecidable. In fact, the proof easily generalizes to work for any fixed string in place of  $foo$ , or even any fixed subset of strings.

Let  $F$  be an oracle that decides  $L_{\text{FOO}}$ , i.e.,

- $F$  halts on any input, and
- $M(foo)$  accepts if and only if  $F(\langle M \rangle)$  accepts.

We claim that the following Turing machine  $C$  decides  $L_{\text{ACC}}$  given access to oracle  $F$ . (Interestingly, observe that the code of  $C$  is *identical* to that of the reduction  $H$  from the proof of [Theorem 103](#), though they are meant to decide different languages, and their oracles are different.)

```
function C( $\langle M \rangle, x$ )
    construct Turing machine “ $M'(w)$ : ignore  $w$  and return  $M(x)$ ”
    return  $F(\langle M' \rangle)$ 
```

The key property here is that  $M(x)$  accepts if and only if  $M'(foo)$  accepts, i.e.,  $foo \in L(M')$ . Indeed, this is true more generally for any fixed string or fixed subset of strings, not just  $foo$ .

We analyze the behavior of  $C$  on an arbitrary input  $(\langle M \rangle, x)$ . First,  $C$  halts on any input, because it just constructs (the code for) a machine  $M'$  from that of  $M$  and  $x$ , and then invokes  $F$ , which halts by hypothesis. Second, by

<sup>28</sup> Technically, Since  $M_{\text{code}}$  and  $x$  may contain newlines and other characters that are not allowed in a C++ string literal, we would need to either process the strings to replace such characters with their respective escape sequences, or use raw string literals<sup>Page 110, 29</sup>.

<sup>29</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2442.htm>

the definition of  $L_{\text{ACC}}$ , the key property above, the hypothesis on  $F$ , and the definition of  $C$ ,

$$\begin{aligned} (\langle M \rangle, x) \in L_{\text{ACC}} &\iff M(x) \text{ accepts} \\ &\iff M'(foo) \text{ accepts} \\ &\iff F(\langle M' \rangle) \text{ accepts} \\ &\iff C(\langle M \rangle, x) \text{ accepts} , \end{aligned}$$

as needed. So, by Definition 61,  $C$  decides  $L_{\text{ACC}}$ , as claimed.

**Example 106** Define

$$L_\emptyset = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \emptyset\} .$$

In other words,  $L_\emptyset$  is the set of (the codes of) *machines* that do not accept *any* input. This language corresponds to the decision problem of determining whether a given program accepts *some* (unspecified) string.

We stress that  $L_\emptyset$  is **not the empty language**  $\emptyset = \{\}$ . The latter is the language that contains *no* strings, and it is decidable because it is decided by the trivial Turing machine  $M_{\text{reject}}$  that ignores its input and immediately rejects. On the other hand,  $L_\emptyset$  consists of (the codes of) *infinitely many* different machines. For example, (the codes of) the following machines are all elements of  $L_\emptyset$ : the just-described machine  $M_{\text{reject}}$ , a modified  $M_{\text{reject}}$  that does one operation before rejecting, the machine  $M_{\text{loop}}$  that ignores its input and loops, the machine that rejects  $\varepsilon$  but loops on every other string, etc.

We show that  $L_{\text{ACC}} \leq_T L_\emptyset$ , hence  $L_\emptyset$  is undecidable. The reduction is almost identical to the ones above, with just a small tweak: the reduction *negates* the output of its oracle. Let  $E$  be an oracle that decides  $L_\emptyset$ . We claim that the following Turing machine  $C$  decides  $L_{\text{ACC}}$  given access to oracle  $E$ :

```
function  $C(\langle M \rangle, x)$ 
construct Turing machine “ $M'(w)$ : ignore  $w$  and return  $M(x)$ ”
return the opposite of  $E(\langle M' \rangle)$ 
```

Observe that if  $M(x)$  accepts, then  $L(M') = \Sigma^* \neq \emptyset$ , whereas if  $M(x)$  does not accept, then  $L(M') = \emptyset$ . So, we have the key property that  $(\langle M \rangle, x) \in L_{\text{ACC}}$  if and only if  $\langle M' \rangle \notin L_\emptyset$ .

We analyze the behavior of  $C$  on an arbitrary input  $(\langle M \rangle, x)$ . First,  $C$  halts by similar reasoning as in the prior examples. Second, by the key property above, the hypothesis on  $E$ , and the definition of  $C$ ,

$$\begin{aligned} (\langle M \rangle, x) \in L_{\text{ACC}} &\iff L(M') \neq \emptyset \\ &\iff E(\langle M' \rangle) \text{ rejects} \\ &\iff C(\langle M \rangle, x) \text{ accepts} . \end{aligned}$$

So, by Definition 61,  $C$  decides  $L_{\text{ACC}}$ , as claimed.

**Example 107** Define

$$L_{\text{EQ}} = \{(\langle M_1 \rangle, \langle M_2 \rangle) : M_1, M_2 \text{ are Turing machines and } L(M_1) = L(M_2)\} .$$

This language corresponds to the decision problem of determining whether two given programs accept *exactly the same strings*, i.e., whether they are “functionally identical” in terms of which strings they accept.

Since this language is concerned with the languages of two given machines, it is convenient to reduce from another language that also involves the language of a given machine, namely,  $L_\emptyset$ . We show that  $L_\emptyset \leq_T L_{\text{EQ}}$ , hence  $L_{\text{EQ}}$  is undecidable. Let  $Q$  be an oracle that decides  $L_{\text{EQ}}$ , i.e.,

- $Q$  halts on any input, and
- $(\langle M_1 \rangle, \langle M_2 \rangle) \in L_{EQ}$  if and only if  $Q(\langle M_1 \rangle, \langle M_2 \rangle)$  accepts.

The challenge here is that, given (the code of) a machine  $M$  (an instance of  $L_\emptyset$ ), the reduction should transform it into (the codes of) *two* machines  $M_1$  and  $M_2$ , so that  $Q$ 's (correct) answer about whether  $L(M_1) = L(M_2)$  will reveal whether  $L(M) = \emptyset$ . The key idea is for the reduction to let  $M_1 = M$ , and to construct machine  $M_2$  so that  $L(M_2) = \emptyset$ ; then  $L(M) = \emptyset$  if and only if  $L(M_1) = L(M_2)$ . There are many such machines that would work as  $M_2$ , but the one that just immediately rejects is simplest.

We claim that the following Turing machine  $E$  decides  $L_\emptyset$  given access to oracle  $Q$ :

```
function  $E(\langle M \rangle)$ 
    construct Turing machine “ $M_2(w)$ : reject”
    return  $Q(\langle M \rangle, \langle M_2 \rangle)$ 
```

Observe that  $L(M_2) = \emptyset$ , trivially. So, we have the key property that  $\langle M \rangle \in L_\emptyset$  if and only if  $(\langle M \rangle, \langle M_2 \rangle) \in L_{EQ}$ .

We analyze the behavior of  $E$  on an arbitrary input  $\langle M \rangle$ . First,  $E$  halts because it just constructs the (fixed) machine  $M_2$  and queries  $Q$ , which halts by hypothesis. Second, by the key property above, the hypothesis on  $Q$ , and the definition of  $E$ ,

$$\begin{aligned} \langle M \rangle \in L_\emptyset &\iff (\langle M \rangle, \langle M_2 \rangle) \in L_{EQ} \\ &\iff Q(\langle M \rangle, \langle M_2 \rangle) \text{ accepts} \\ &\iff E(\langle M \rangle, x) \text{ accepts}, \end{aligned}$$

as needed. So, by Definition 61,  $E$  decides  $L_\emptyset$ , as claimed.

**Example 108** We have seen that  $L_{ACC} \leq_T L_{HALT}$ . We now show the other direction, that  $L_{HALT} \leq_T L_{ACC}$ .

Let  $C$  be an oracle that decides  $L_{ACC}$ , i.e.,

- $C$  halts on every input, and
- $M(x)$  accepts if and only if  $C(\langle M \rangle, x)$  accepts.

We wish to construct a Turing machine  $H$  that decides  $L_{HALT}$ , i.e.,

- $H$  halts on every input, and
- $M(x)$  halts (accepts or rejects) if and only if  $H(\langle M \rangle, x)$  accepts.

The only case where the behavior of  $H$  needs to differ from that of  $C$  is where  $M(x)$  rejects; we need  $H$  to accept, whereas  $C$  rejects. So, to use  $C$  to get the correct answer in this case, we construct a new program  $M'$  that just negates the accept/reject decision (if any) of  $M$ .

We claim that the following Turing machine  $H$  decides  $L_{HALT}$  given access to oracle  $C$ :

```
function  $H(\langle M \rangle, x)$  construct Turing machine “ $M'(w)$ : return the opposite of  $M(w)$ ”
    if  $C(\langle M \rangle, x)$  accepts or  $C(\langle M' \rangle, x)$  accepts then accept
    reject
```

We analyze the behavior of  $H$  on an arbitrary input  $(\langle M \rangle, x)$ . First,  $H$  halts on any input, because it just constructs the code of  $M'$  from that of  $M$  and  $x$ , and twice queries  $C$ , which halts by hypothesis. Second, by the definition

of  $L_{\text{HALT}}$ , the construction of  $M'$ , the hypothesis on  $C$ , and the definition of  $H$ ,

$$\begin{aligned} (\langle M \rangle, x) \in L_{\text{HALT}} &\iff M(x) \text{ accepts or } M(x) \text{ rejects} \\ &\iff (\langle M \rangle, x) \in L_{\text{ACC}} \text{ or } (\langle M' \rangle, x) \in L_{\text{ACC}} \\ &\iff C(\langle M \rangle, x) \text{ accepts or } C(\langle M' \rangle, x) \text{ accepts} \\ &\iff H(\langle M \rangle, x) \text{ accepts}, \end{aligned}$$

as needed. So, by Definition 61,  $H$  decides  $L_{\text{HALT}}$ , as claimed.

Observe that the reduction  $H$  invoked the oracle  $C$  *twice* on different inputs. In general, to prove that  $A \leq_T B$ , the Turing machine that decides  $A$  may call the oracle that decides  $B$  any (finite) number of times, including more than once, or even not at all.

**Exercise 109** Prove that  $L_{\text{ACC}} \leq_T L_{\varepsilon\text{-HALT}}$ . (Hint: there is a fairly complex direct proof by reduction, and a very short proof using results that have already been stated.)

**Exercise 110** Prove that the language

$$L_{\varepsilon\text{-ACC}} = \{\langle M \rangle : M \text{ accepts } \varepsilon\}$$

is undecidable.

**Exercise 111** Determine, with proof, whether the following language is decidable:

$$L_3 = \{\langle M \rangle : M \text{ is a Turing machine and } |L(M)| = 3\}.$$

**Exercise 112** Determine, with proof, whether the following language is decidable:

$$L_{\text{EVEN}} = \{\langle M \rangle : M \text{ is a Turing machine and } |L(M)| \text{ is finite and even}\}.$$

## 11.3 Wang Tiling

All the examples of undecidable languages we have seen thus far have been concerned with the behavior of Turing machines. However, there are many other languages whose definitions *appear* to have nothing to do with Turing machines or computation, and yet are also undecidable! It turns out that solving these problems would require solving unsolvable problems about computation, because we can “embed” the halting problem into these problems.

Here we give an overview of the undecidability of a certain *geometric* problem, that of tiling a plane<sup>30</sup> using some set of shapes.

The computational question is: given a finite set  $S$  of shapes, called *tiles*, is it possible to tile the plane using only the tiles from  $S$ ? We may use as many copies of each tile as we need, but they may not overlap or leave any uncovered space. For the purposes of this discussion, we also disallow rotation of tiles. (If we want to allow certain rotations, we can just include them in the tile set.)

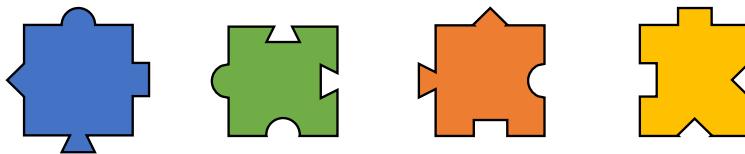
An example of a valid tiling is the following from *Study of Regular Division of the Plane with Reptiles* by M. C. Escher<sup>31</sup>: Here, the plane is tiled by a regular pattern using three reptile-shaped tiles.

<sup>30</sup> <https://en.wikipedia.org/wiki/Tessellation>

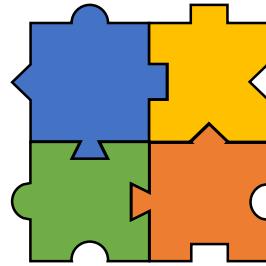
<sup>31</sup> [https://en.wikipedia.org/wiki/M.\\_C.\\_Escher](https://en.wikipedia.org/wiki/M._C._Escher)



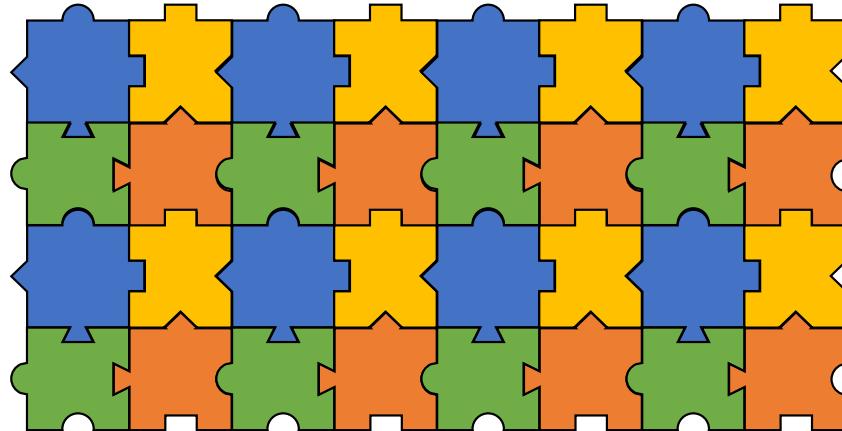
Another example more akin to a jigsaw puzzle uses the following set of tiles:



First, we can fit the tiles together like so:



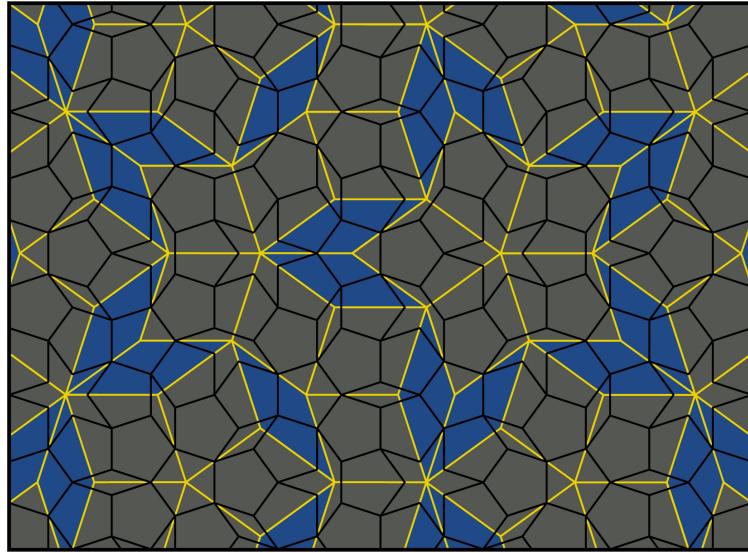
Then we can repeat this pattern to tile the whole plane:



While the two examples above admit *periodic* tilings, it is also possible to tile the plane in a *non-periodic* manner<sup>32</sup>. The following is an example of a [Penrose tiling](#)<sup>33</sup> that does so:

<sup>32</sup> [https://en.wikipedia.org/wiki/Aperiodic\\_tiling](https://en.wikipedia.org/wiki/Aperiodic_tiling)

<sup>33</sup> [https://en.wikipedia.org/wiki/Penrose\\_tiling](https://en.wikipedia.org/wiki/Penrose_tiling)



Not all tile sets admit tilings of the entire plane; for example, the set consisting of just a regular pentagon cannot do so. A natural question is whether there is an *algorithm* that, given an arbitrary (finite) set  $S$  of tile shapes, determines whether it is possible to tile the plane with that set. We define the corresponding language  $L_{\text{TILE}}$  as follows:

$$L_{\text{TILE}} = \{\langle S \rangle : S \text{ is a tile set that admits a tiling of the entire plane}\}.$$

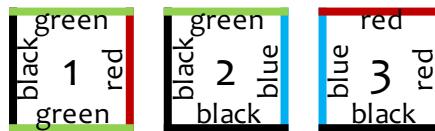
Is  $L_{\text{TILE}}$  decidable? The mathematician Hao Wang posed this and related questions in the early 1960s—conjecturing that the answer is “yes”—but in 1966, his student Robert Berger proved that amazingly, the answer is no! That is, there is *no* algorithm that, given an arbitrary (but finite) set of tile shapes, correctly determines whether those shapes can be used to tile the plane.

The key insight behind the proof is that tiling with certain shapes can be made to correspond exactly with the *execution of a given Turing machine*. A bit more precisely, for any Turing machine  $M$  there is a corresponding (computable) tile set  $S_M$  for which determining whether  $S_M$  tiles the plane is equivalent to determining whether  $M(\varepsilon)$  halts—i.e., whether  $\langle M \rangle \in L_{\varepsilon\text{-HALT}}$ . This insight is the heart of the proof that  $L_{\varepsilon\text{-HALT}} \leq_T L_{\text{TILE}}$ , so  $L_{\text{TILE}}$  is indeed undecidable.

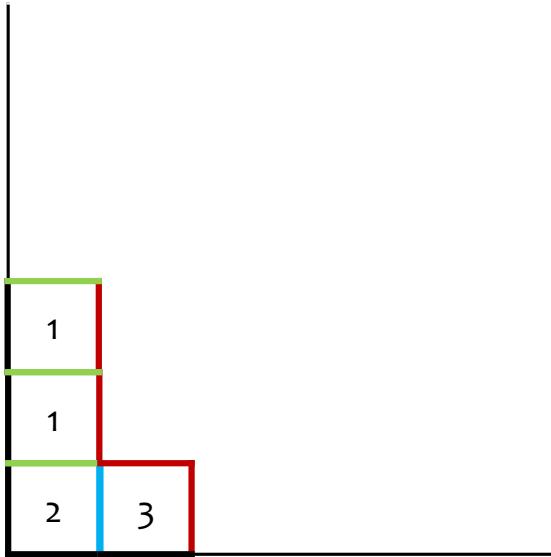
For the rest of the discussion, we restrict to the slightly simpler problem of determining whether a tile set  $S$  can tile the *upper-right quadrant* of the plane, rather than a whole plane. That is, the language we consider is:

$$L_{\text{QTILE}} = \{\langle S \rangle : S \text{ is a tile set that admits a tiling of the upper-right quadrant}\}.$$

To simplify the picture and avoid complicated-looking shapes, we restrict our attention to tiles that are unit squares with a “color” on each edge. Two tiles may be adjacent only if their shared edges overlap completely, and the colors of these edges match. We can think of each color as representing some distinct “cut out” and “pasted on” shape, so that only tiles with matching colors “fit together” in the required way. (As mentioned before, tiles may not be rotated or flipped.) These are known as *Wang tiles* or *Wang dominoes*, after the mathematician Hao Wang, who studied the computability of domino questions like this one.



We also color the boundary of the quadrant black.



The question then is whether a given set of tiles can tile the whole quadrant under these constraints. For the tile set above, we can see that it cannot. Only tile 2 may be placed in the bottom-left corner, then only tile 3 and tile 1 may appear to its right and above it, respectively. While we can continue placing more copies of tile 1 in the upward direction, no tile has a left boundary that is colored red, so we cannot fill out the rest of the quadrant to the right.

While we were able to reason that in this specific case, the quadrant cannot be tiled with the given set, in general, no computational process can reach a correct conclusion. To demonstrate this, we will construct a tile set for which any tiling corresponds exactly to the execution of an arbitrary given Turing machine. For concreteness we will illustrate this for a single, simple machine, but the same process can be done for any given machine.

Before proceeding to our construction, we define a notation for representing the execution state, or *configuration*, of a Turing machine. We need to capture several details that uniquely describe the configuration of the machine, which consists of:

- the machine's active state,
- the contents of the tape, and
- the position of the head.

We represent these details by an infinite sequence over the (finite) alphabet  $\Gamma \cup Q$ . Specifically, the sequence contains the full contents of the tape, plus the active state  $q \in Q$  immediately to the left of the symbol for the cell at which the head is located. Thus, if the input to a machine is  $s_1 s_2 \dots s_n$  and the initial state is  $q_0$ , the following encodes the starting configuration of the machine:

$$q_0 s_1 s_2 \dots s_n \perp \perp \dots$$

Since the head is at the leftmost cell and the state is  $q_0$ , the string has  $q_0$  to the left of the leftmost tape symbol. Then if the transition function gives  $\delta(q_0, s_1) = (q', s', R)$ , the new configuration is:

$$s' q' s_2 \dots s_n \perp \perp \dots$$

The first cell has been modified to  $s'$ , the machine is in state  $q'$ , and the head is over the second cell, represented here by writing  $q'$  to the left of that cell's symbol.

As a more concrete example, the configuration of *the machine we saw previously that accepts strings containing only*

[ones](#) (page 64), when run on the input 111010111, is as follows at each step:

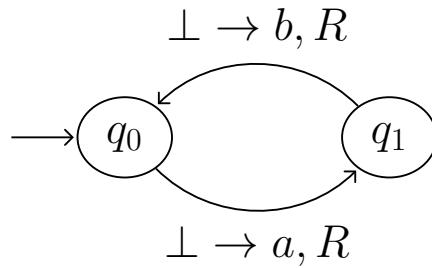
```

 $q_{\text{start}}111010111\perp\perp\perp\perp\perp\cdots$ 
 $1q_{\text{start}}111010111\perp\perp\perp\perp\perp\cdots$ 
 $11q_{\text{start}}1010111\perp\perp\perp\perp\perp\cdots$ 
 $111q_{\text{start}}010111\perp\perp\perp\perp\perp\cdots$ 
 $1110q_{\text{rej}}10111\perp\perp\perp\perp\perp\cdots$ 

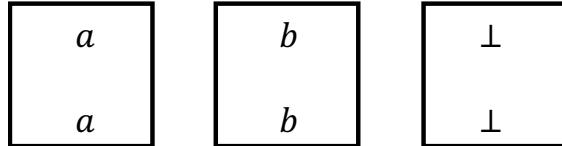
```

We will encode such a configuration string using an infinite sequence of tiles, where the “colors” along the top edges of the tiles indicate the configuration.

For a concrete running example, consider the simple machine that just alternately writes two different symbols to the tape. Given an initial blank tape (i.e.,  $\varepsilon$  as the input), the machine will write symbols indefinitely, never terminating.



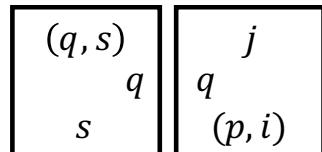
We start constructing our tile set by introducing a tile for each symbol in the tape alphabet. Each tile’s top and bottom edge is “colored” by the symbols appearing there (respectively), and the left and right edges have the “blank” color.



We next examine the transition function. For each left-moving transition

$$\delta(p, i) = (q, j, L)$$

and each element of the tape alphabet  $s \in \Gamma$ , we introduce the following pair of tiles:

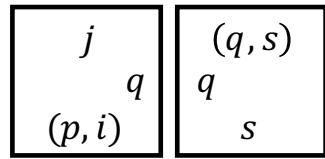


This represents part of the configuration when applying the transition. The bottom of the right tile indicates that we are originally in state  $p$  with the head on the symbol  $i$ , and the bottom of the left tile indicates there is a symbol  $s$  in the cell to the left. The tops of the tiles represent the result of the transition. The new state is  $q$  and the head moves to the left, which we indicate by the top of the left tile. The symbol in the right tile is overwritten with  $j$ , which we indicate at the top of the right tile. Finally, we label the adjacent edges with  $q$  to enable the two tiles to be adjacent to each other, but not to the “symbol” tiles constructed above (which have “blank”-colored left and right edges).

Similarly, for each right-moving transition

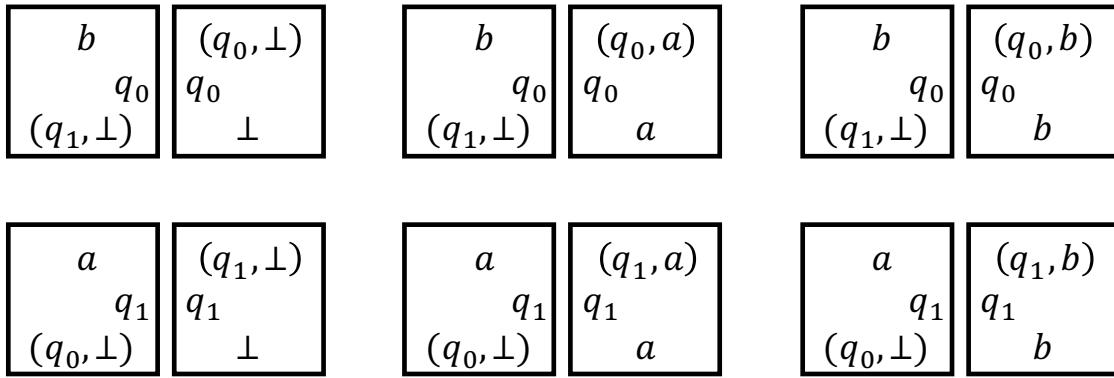
$$\delta(p, i) = (q, j, R)$$

and tape symbol  $s$ , we introduce the following pair of tiles:



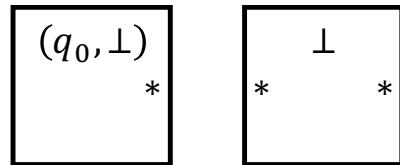
Here, the head moves to the right, so the original state  $p$  appears at the bottom in the left tile and the new state  $q$  appears at the top in the right tile.

For our simple Turing machine above, we have just two rightward transitions and three tape symbols, so the resulting transition tiles are those below:

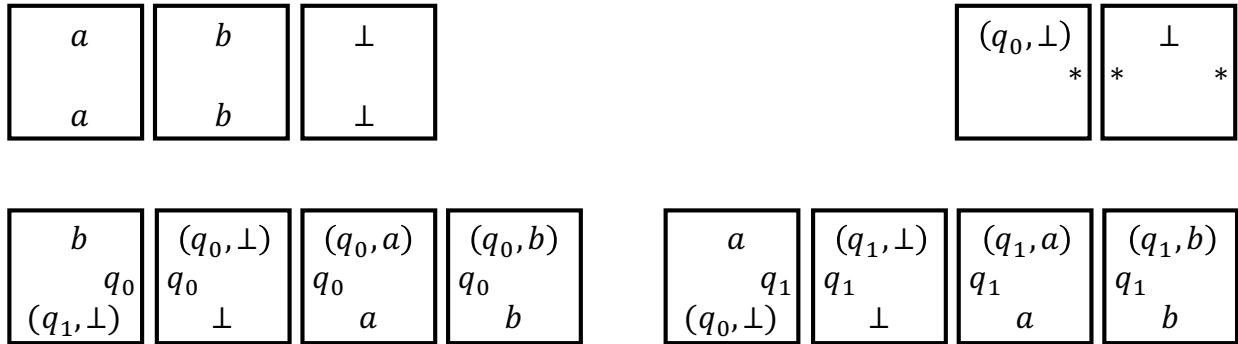


(We have shown duplicate tiles here, but since we are ultimately constructing a set, we would merge any duplicates into one.)

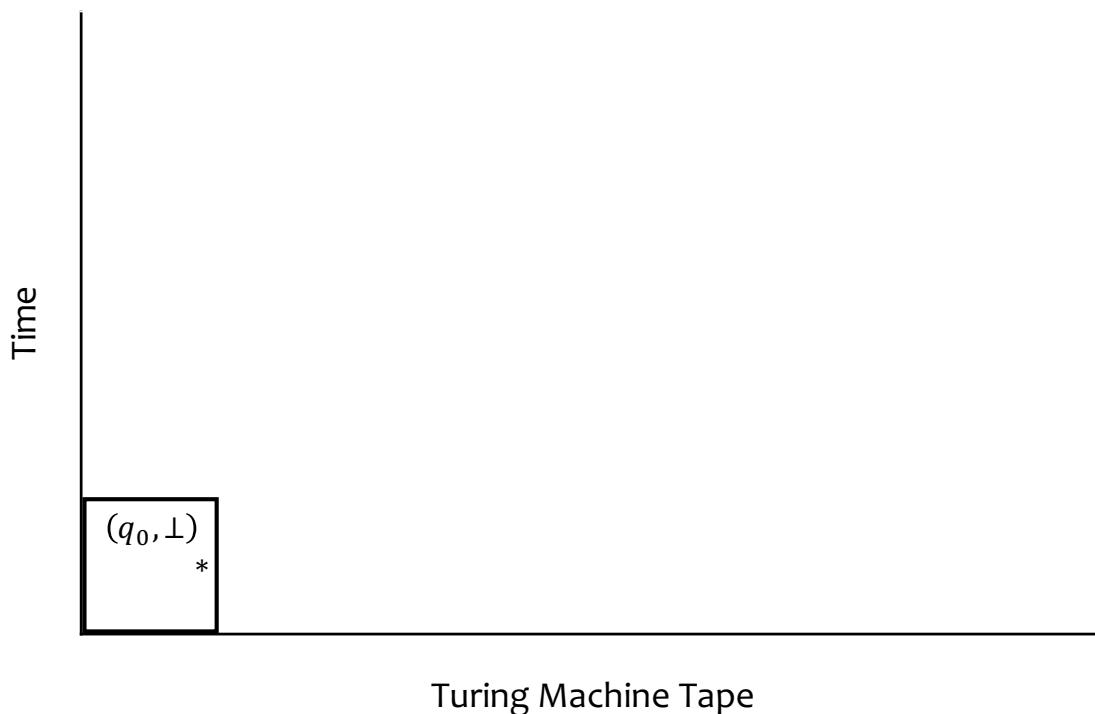
Finally, we introduce “start” tiles to encode the initial state of the machine. Since we are interested in the empty string as input, only blank symbols appear in our start tiles. We have a corner tile that encodes the start state and the initial head position all the way to the left, and we have a bottom tile that encodes the remaining blank cells:



The full set of tiles for the simple Turing machine, with duplicates removed, is as follows:

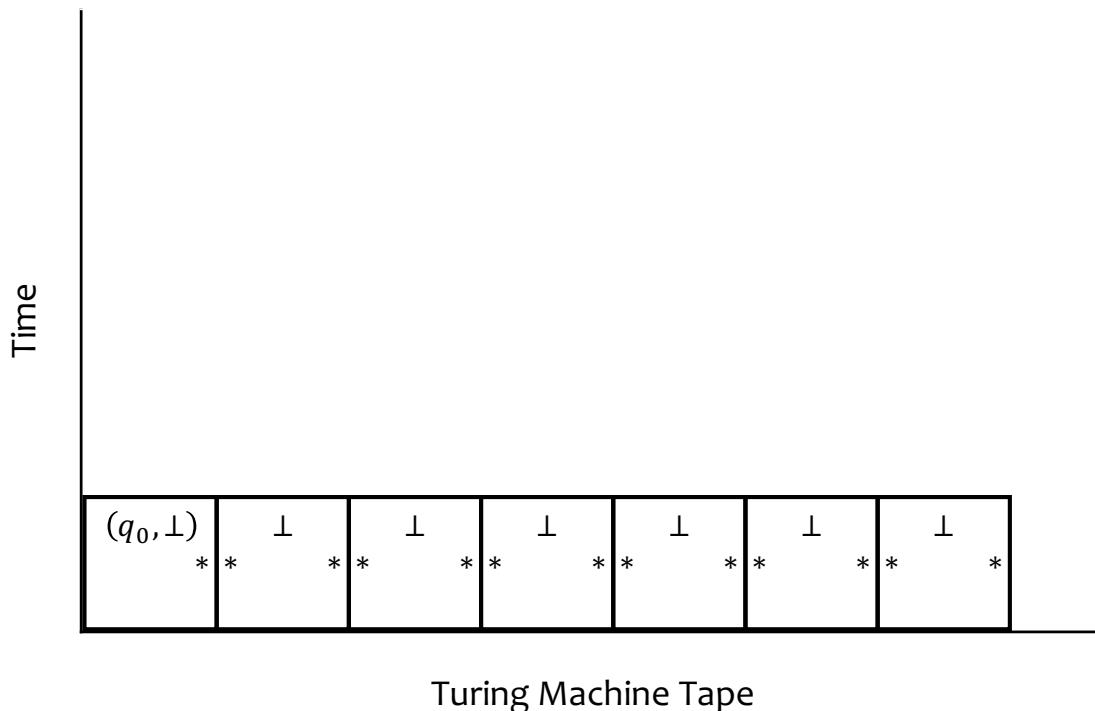


Now that we have constructed our tile set, we can consider how a tiling of the quadrant would look, if one exists. First, the only tile that can go in the bottom-left corner of the quadrant is our corner start tile.



Turing Machine Tape

Once we place the corner tile, the only tile that can go to its right is a bottom start tile, and likewise to its right, indefinitely.

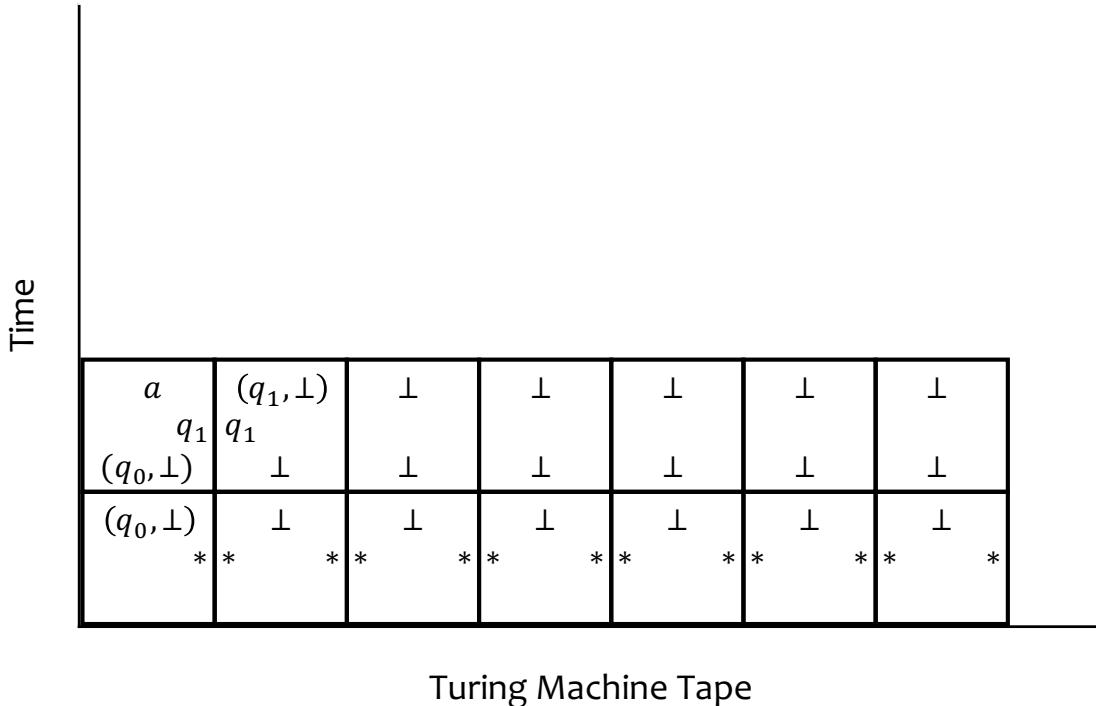


Turing Machine Tape

The resulting bottom row encodes the initial state of the machine: every cell contains the blank symbol (since the input is  $\epsilon$ ), the initial state is  $q_0$ , and the head is at the leftmost cell. The position of the head is indicated by which tile has both a state and a symbol in its top edge.

Now that the bottom row is determined, we consider what tiles may go above that row. Only one of the tiles has  $(q_0, \perp)$

at its bottom edge, and it is one of the tiles corresponding to the transition  $\delta(q_0, \perp) = (q_1, a, R)$ . This forces us to use one of the other tiles for that transition to its right, and since that tile's bottom edge must also match the top edge of the tile below it, we use the corresponding transition tile that has  $\perp$  on its bottom edge. By similar reasoning, the remaining tiles in the row must be symbol tiles for  $\perp$ .



Turing Machine Tape

The configuration represented in the second row above corresponds to the machine having written  $a$  in the first cell, being in the state  $q_1$ , with its head on the second cell. This is exactly the result of the transition.

To fill the next row, we must use a symbol tile for  $a$  at the left. Then, we must use tiles for the transition  $\delta(q_1, \perp) = (q_0, b, R)$ . Finally, we must fill out the rest of the row with symbol tiles for  $\perp$ .

Time	$a$	$b$	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$	$b$ $q_0$	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$	$(q_1, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$ $q_1$	$(q_1, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$

Turing Machine Tape

Again, the configuration represented here corresponds exactly with the state of the machine after two steps. The next row is similarly determined.

Time	$a$	$b$	$a$ $q_1$	$(q_1, \perp)$	$\perp$	$\perp$	$\perp$
	$a$	$b$	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$	$b$ $q_0$	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$	$(q_1, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$a$ $q_1$	$(q_1, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$(q_0, \perp)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$	$*$ $*$

Turing Machine Tape

If the machine halts after some step, then there is no way to fill in the next row of the tiling, because there are no transition tiles for the final states  $q_{\text{acc}}, q_{\text{rej}}$ —so the quadrant cannot be tiled. Conversely, if the machine runs forever, then every row can be filled with corresponding tiles, so the quadrant can be tiled. In other words, the question of whether the quadrant can be tiled is *equivalent* to the question of whether the machine loops on input  $\varepsilon$ . Stated more

precisely, the key property is as follows: the code of the machine is in  $L_{\varepsilon\text{-HALT}}$  if and only if the corresponding tile set is *not* in  $L_{\text{QTILE}}$ .

The following formalizes our proof of undecidability of  $L_{\text{QTILE}}$ , by showing that  $L_{\varepsilon\text{-HALT}} \leq_T L_{\text{QTILE}}$  via a Turing reduction. Suppose that  $T$  is an oracle that decides  $L_{\text{QTILE}}$ . We claim that the following Turing machine  $E$  decides  $L_{\varepsilon\text{-HALT}}$  given access to  $T$ :

```
function  $E(\langle M \rangle)$  construct tile set  $S_M$  from  $\langle M \rangle$  as outlined above
return the opposite of  $T(\langle S_M \rangle)$ 
```

We analyze the behavior of  $E$  on an arbitrary input  $\langle M \rangle$ . First,  $E$  halts because constructing the tile set is a finite computation based on the finite *code*—alphabets, states, transition function—of  $M$ , and  $T$  halts by hypothesis. (We stress that he reduction **does not simulate/run**  $M$ , or attempt to tile the quadrant; it just “syntactically” converts  $M$ ’s *code* into a finite set of tiles.) Second, by definition of  $L_{\varepsilon\text{-HALT}}$ , the key property of the tile set, the hypothesis on  $T$ , and the definition of  $E$ ,

$$\begin{aligned} \langle M \rangle \in L_{\varepsilon\text{-HALT}} &\iff M(\varepsilon) \text{ halts} \\ &\iff \langle S_M \rangle \notin L_{\text{QTILE}} \\ &\iff T(\langle S_M \rangle) \text{ rejects} \\ &\iff E(\langle M \rangle) \text{ accepts} , \end{aligned}$$

as needed. Thus,  $E$  decides  $L_{\varepsilon\text{-HALT}}$ , and we have demonstrated that  $L_{\varepsilon\text{-HALT}} \leq_T L_{\text{QTILE}}$ . Since  $L_{\varepsilon\text{-HALT}}$  is undecidable, so is  $L_{\text{QTILE}}$ .

**Exercise 113** Modify the construction we used above to show instead that  $L_{\text{HALT}} \leq_T L_{\text{QTILE}}$ .

(Hint: only the set of start tiles needs to be changed, and the set will be different for different machine inputs.)

---

CHAPTER  
TWELVE

---

## RECOGNIZABILITY

Recall that there is a relaxation of the notion of deciding, called recognizing. For convenience we repeat Definition 62 here.

**Definition 114 ('Recognizes' for Turing machines)** A Turing machine  $M$  *recognizes* a language  $L \subseteq \Sigma^*$  if:

1.  $M$  accepts every  $x \in L$ , and
2.  $M$  rejects or loops on every  $x \notin L$ .

Equivalently:  $x \in L$  if and only if  $M$  accepts  $x$ .

A language is *recognizable* if some Turing machine recognizes it.

Notice that, as in the definition of “decides”, the machine still must accept every string in the language. However, here there is no requirement that the machine halts on all inputs; it may loop on some (or even all) inputs that are not in the language. In other words, any output the machine produces is correct, but for inputs where the correct answer is “no” (and only for such inputs), the machine need not produce an output at all.

Tautologically, any Turing machine  $M$  recognizes exactly one language, namely, the language  $L(M) = \{x \in \Sigma^* : M(x) \text{ accepts}\}$  of those strings the machine accepts. Also trivially, a machine that decides a language also recognizes that same language, but not necessarily vice-versa.

A language may be undecidable but recognizable. We have seen several examples of undecidable languages, including  $L_{\text{ACC}}$ ,  $L_{\text{HALT}}$ , and  $L_{\varepsilon\text{-HALT}}$ , and in our coverage of *simulation* (page 102) we saw that the universal Turing machine  $U$  recognizes the language  $L(U) = L_{\text{ACC}}$ , so  $L_{\text{ACC}}$  is recognizable. And we will see below,  $L_{\text{HALT}}$  and  $L_{\varepsilon\text{-HALT}}$  are recognizable as well.

Is *every* language recognizable? We have already seen that this is not the case: the diagonalization proof of Theorem 85 constructed a (contrived) unrecognizable language, and Theorem 87 showed that *the “barber” language* (page 101) is also unrecognizable. In this section we will see a technique for showing that other languages are unrecognizable as well.

But before dealing with *unrecognizability*, how can we show that a language  $L$  is *recognizable*? Similar to how we show that a language is decidable, we need only give a Turing machine that recognizes  $L$ . For example, there is a straightforward recognizer for the halting language  $L_{\text{HALT}}$ .

**Theorem 115** *The language  $L_{\text{HALT}}$  is recognizable.*

**Proof 116** We claim that the following Turing machine recognizes  $L_{\text{HALT}}$ .

```
function  $H_r(\langle M \rangle, x)$  simulate  $M(x)$  and ignore its output (if any)
    accept
```

We analyze the behavior of  $H_r$  on an arbitrary input  $(\langle M \rangle, x)$ :

- If  $(\langle M \rangle, x) \in L_{\text{HALT}}$ , then  $M(x)$  halts. In this case, the simulation of  $M(x)$  in  $H_r$  terminates, and  $H_r$  proceeds to accept  $(\langle M \rangle, x)$ , as needed.
- If  $(\langle M \rangle, x) \notin L_{\text{HALT}}$ , then  $M(x)$  loops. In this case, the simulation of  $M(x)$  in  $H_r$  does not terminate (and the second line is never reached), so  $H_r$  loops on  $(\langle M \rangle, x)$ .

Therefore, by Definition 62,  $H_r$  recognizes  $L_{\text{HALT}}$ . □

### Exercise 117

Adapt the proof of Theorem 115 to prove that  $L_{\varepsilon\text{-HALT}}$  is recognizable.

As another example, in Lemma 65 we proved that the class of decidable languages is closed under the union operation, i.e., if  $L_1, L_2$  are any decidable languages, then their union  $L = L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$  is also decidable. Does the same hold with “decidable” replaced by “recognizable”? It does indeed, but adapting the proof to work with recognizability is more subtle, as we now discuss.

Because  $L_1$  and  $L_2$  are recognizable, there exist Turing machines  $M_1$  and  $M_2$  that recognize  $L_1$  and  $L_2$ , respectively. Let’s consider whether the machine  $M$  we constructed in the proof of Lemma 65 recognizes  $L = L_1 \cup L_2$ :

```
function  $M(x)$ 
  if  $M_1(x)$  accepts then accept
  if  $M_2(x)$  accepts then accept
  reject
```

Unfortunately, this machine does **not necessarily recognize**  $L$ , because  $M_1$  may not be a decider. Specifically, suppose that  $x \notin L_1$  but  $x \in L_2$ , so  $x \in L$ , and therefore we need  $M(x)$  to accept in order for  $M$  to recognize  $L$ . Now suppose that  $M_1(x)$  happens to loop—which it may do, because it is merely a recognizer for  $L_1$ . Then  $M(x)$  loops as well—but this is incorrect behavior! Even though  $M_2(x)$  is guaranteed to accept,  $M(x)$  never reaches the point of simulating it. As a potential fix, we could try swapping the first two lines of  $M$ , but that attempt would fail due to a symmetric issue.

The issue here is similar to what we encountered in the proof that the set of integers is countable (Example 74). Our attempt to enumerate the integers by first listing the nonnegative integers failed, because we would never exhaust them to reach the negative integers. The solution was to *interleave* the nonnegative and negative integers, so that every specific integer would eventually be reached.

Here we can follow an analogous strategy called *alternation*, interleaving the simulated executions of  $M_1$  and  $M_2$ :

```
function  $M(x)$ 
  alternate between simulations of  $M_1(x)$  and  $M_2(x)$ 
  if either one accepts then accept
  if both reject then reject
```

How can we alternate between the executions of  $M_1(x)$  and  $M_2(x)$ ? When simulating their executions in a Turing machine, we run one step of  $M_1$  (i.e., apply its transition function once), then one step of  $M_2$ , then another step of  $M_1$ , then another step of  $M_2$ , and so on. (If a simulated machine halts, then we do not run any further steps of it, because such steps are not defined.) Similarly, real computers simulate parallel execution of multiple programs on a single processor by rapidly switching between programs. So, we can alternate executions both in theory and in practice.

We now analyze the behavior of the above  $M$  on an arbitrary input  $x$ :

- If  $x \in L = L_1 \cup L_2$ , then  $x \in L_1$  or  $x \in L_2$  (or both). So, at least one of  $M_1(x), M_2(x)$  accepts, because  $M_i$  is a recognizer for  $L_i$ . Since  $M$  alternates between the simulated executions of  $M_1(x)$  and  $M_2(x)$ , it will eventually reach a point at which one of them accepts, so  $M(x)$  accepts, as needed.
- If  $x \notin L$ , then  $x \notin L_1$  and  $x \notin L_2$ . So, neither  $M_1(x)$  nor  $M_2(x)$  accepts; each one either rejects or loops. There are two cases: if both  $M_1(x)$  and  $M_2(x)$  reject, then  $M(x)$  will reach its final line and reject. Otherwise,

at least one of  $M_1(x), M_2(x)$  loops, so the alternating simulation runs forever, i.e.,  $M(x)$  loops. Either way,  $M(x)$  rejects or loops, as needed.

Alternatively and more succinctly, we can see by definition of  $L$ , hypothesis on  $M_1$  and  $M_2$ , and definition of  $M$ ,

$$\begin{aligned} x \in L &\iff x \in L_1 \text{ or } x \in L_2 \\ &\iff M_1(x) \text{ accepts or } M_2(x) \text{ accepts} \\ &\iff M(x) \text{ accepts} . \end{aligned}$$

Therefore, by Definition 62,  $M$  recognizes  $L$ , hence  $L$  is recognizable. We have just proved the following theorem.

**Theorem 118** *For any recognizable languages  $L_1$  and  $L_2$ , their union  $L = L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$  is also recognizable. In other words, the class of recognizable languages is closed under union.*

**Exercise 119** Adapt the proof of Theorem 118 to prove that the class of recognizable languages is closed under intersection.

Is the class of recognizable languages closed under *complement*, like the class of decidable languages is? In other words, if  $L$  is a recognizable language, is its complement  $\overline{L}$  necessarily recognizable? It turns out that the answer is *no!* We will prove this somewhat indirectly, by showing that if a language and its complement are both recognizable, then the language is decidable.

**Theorem 120** *If a language  $L$  and its complement  $\overline{L}$  are both recognizable, then  $L$  is decidable (and by symmetry, so is  $\overline{L}$ ).*

**Proof 121** By hypothesis, there exist Turing machines  $M$  and  $\overline{M}$  that recognize  $L$  and  $\overline{L}$ , respectively. Since  $M$  recognizes  $L$ ,

$$x \in L \iff M(x) \text{ accepts} .$$

And since  $\overline{M}$  recognizes  $\overline{L}$ ,

$$x \notin L \iff x \in \overline{L} \iff \overline{M}(x) \text{ accepts} .$$

So, for every input  $x$ , exactly one of  $M(x)$  and  $\overline{M}(x)$  accepts, and  $x \in L$  if and only if  $M(x)$  accepts and  $\overline{M}(x)$  does not accept.

We now claim that the following Turing machine  $M'$  decides  $L$ . The key idea is to alternate between the executions of  $M$  and  $\overline{M}$  on the input, and see which one accepts to determine whether the input is in  $L$ .

```
function  $M(x)$ 
    alternate between simulations of  $M(x)$  and  $\overline{M}(x)$ 
    if  $M(x)$  ever accepts then accept
    if  $\overline{M}(x)$  ever accepts then reject
```

We analyze the behavior of  $M'$  on an arbitrary input  $x$ . First,  $M'(x)$  halts, because as noted above, exactly one of  $M(x)$  and  $\overline{M}(x)$  accepts, and either case causes  $M'(x)$  to halt. Second, by the properties of  $M$  and  $\overline{M}$  stated above and the definition of  $M'$ ,

$$\begin{aligned} x \in L &\iff M(x) \text{ accepts and } \overline{M}(x) \text{ does not accept} \\ &\iff M'(x) \text{ accepts} . \end{aligned}$$

So, by Definition 61,  $M'$  decides  $L$ , hence  $L$  is decidable. □

## 12.1 Unrecognizable Languages

The contrapositive of [Theorem 120](#) is that if a language is *undecidable*, then either it or its complement (or both) is *unrecognizable*. This gives us a tool to show that a language is unrecognizable.

**Corollary 122** *If a language  $L$  is undecidable, then at least one of  $L$  and  $\overline{L}$  is unrecognizable. In particular, if  $L$  is undecidable and its complement  $\overline{L}$  is recognizable, then  $L$  is unrecognizable.*

From [Corollary 122](#) we can immediately conclude that the complement languages<sup>34</sup>

$$\overline{L_{\text{ACC}}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ does not accept}\}$$

and

$$\overline{L_{\text{HALT}}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ loops}\}$$

are *unrecognizable*.

A language whose complement is recognizable (without any other restriction on the language itself) is called *co-recognizable*.

**Definition 123** A language  $L$  is *co-recognizable* if  $\overline{L}$  is recognizable.

Since the class of decidable languages is closed under complement, and every decidable language is recognizable, any decidable language is both recognizable and co-recognizable. In fact, [Claim 120](#) implies that the class of decidable languages is the intersection of the class of recognizable languages and the class of co-recognizable languages.<sup>35</sup> (Be careful about interpreting the preceding sentence: it is not saying anything about the intersection of *languages themselves*; it is referring to the intersection of *classes* of languages, i.e., the set of languages that are members of both classes.)

Since every program  $M$  recognizes exactly one language  $L(M)$ , it “co-recognizes” exactly one language  $\overline{L(M)}$ . Since the set of Turing machines is countable (see [Lemma 83](#)), and since each Turing machine (co-)recognizes exactly one language, the set of (co-)recognizable languages is also countable. Because the set of all languages is uncountable, “almost all” languages are neither recognizable nor co-recognizable.

**Example 124** We show that the language

$$L_{\text{NO-FOO}} = \{\langle M \rangle : M \text{ is a Turing machine and } \text{foo} \notin L(M)\}$$

is unrecognizable.

We first observe that  $L_{\text{NO-FOO}} = \overline{L_{\text{FOO}}}$ , where  $L_{\text{FOO}}$  is the undecidable language from [Example 105](#). Since the complement of any undecidable language is undecidable,  $L_{\text{NO-FOO}}$  is undecidable as well.

We now show that  $L_{\text{FOO}}$  is recognizable. The following program recognizes it:

```
function R( $\langle M \rangle$ ) return  $M(\text{foo})$ 
```

<sup>34</sup> To be precise, the complement language  $\overline{L_{\text{ACC}}}$  consists of *all strings* that are *not* of the form  $(\langle M \rangle, x)$  where  $M$  is a Turing machine that accepts  $x$  (and similarly for  $\overline{L_{\text{HALT}}}$ ). Depending on the input encoding, this might include “malformed” strings that do not encode any Turing machine-string pair  $(\langle M \rangle, x)$  at all. In this case, the above description of  $\overline{L_{\text{ACC}}}$  would not be correct, because it does not include such strings. However, we can assume without loss of generality that *every* string encodes some input object of the proper form, simply by redefining any “malformed” strings to represent some fixed “default” object. Throughout the text we implicitly assume this, to simplify the descriptions of complement languages.

<sup>35</sup> The class of recognizable languages is denoted RE (short for *recursively enumerable*, an equivalent definition of recognizable), and the class of co-recognizable languages is denoted coRE. The class of decidable languages is denoted R (short for *recursive*). We have  $\text{RE} \cap \text{coRE} = \text{R}$ . Some languages are neither in RE nor in coRE, but how to prove this is beyond the scope of this text.

This is because

$$\begin{aligned} \langle M \rangle \in L_{\text{FOO}} &\iff M(\text{foo}) \text{ accepts} \\ &\iff R(\langle M \rangle) \text{ accepts} . \end{aligned}$$

Since  $L_{\text{NO-FOO}}$  is undecidable and its complement language  $L_{\text{FOO}}$  is recognizable, by Corollary 122 we conclude that  $L_{\text{NO-FOO}}$  is unrecognizable.

**Example 125** We show that the language

$$L_{\text{NO-FOO-BAR}} = \{\langle M \rangle : M \text{ is a TM and } \text{foo} \notin L(M) \text{ and } \text{bar} \notin L(M)\}$$

is unrecognizable.

Its complement language is  $L_{\text{FOO-OR-BAR}} = \{\langle M \rangle : \text{foo} \in L(M) \text{ or } \text{bar} \in L(M)\}$ . As noted in Example 105, this language is undecidable, so  $L_{\text{NO-FOO-BAR}}$  is undecidable as well.

We now show that  $L_{\text{FOO-OR-BAR}}$  is recognizable. The following program recognizes it:

```
function  $R(\langle M \rangle)$ 
    alternate between simulations of  $M(\text{foo})$  and  $M(\text{bar})$ 
    if at least one accepts then accept
    if both reject then reject
```

This is because

$$\begin{aligned} \langle M \rangle \in L_{\text{FOO-OR-BAR}} &\iff M(\text{foo}) \text{ accepts or } M(\text{bar}) \text{ accepts} \\ &\iff R(\langle M \rangle) \text{ accepts} . \end{aligned}$$

Since  $L_{\text{NO-FOO-BAR}}$  is undecidable and its complement language  $L_{\text{FOO-OR-BAR}}$  is recognizable, by Corollary 122 we conclude that  $L_{\text{NO-FOO-BAR}}$  is unrecognizable.

## 12.2 Dovetailing

Here we introduce a powerful simulation technique called “dovetailing”, which we use to prove that the language

$$L_\emptyset = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \emptyset\}$$

from Example 106 is unrecognizable. Because we already showed that it is undecidable, by Corollary 122, it suffices to show that its complement language

$$\overline{L_\emptyset} = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) \neq \emptyset\}$$

is recognizable. To do so, we define a recognizer  $R$  for  $\overline{L_\emptyset}$ . We require the following behavior for an arbitrary input machine  $M$ :

- If  $M$  accepts at least one input string, then  $R(\langle M \rangle)$  must accept;
- If  $M$  does not accept any input string, then  $R(\langle M \rangle)$  must reject or loop.

In other words, we require that  $\langle M \rangle \in \overline{L_\emptyset} \text{ if and only if } R(\langle M \rangle) \text{ accepts}$ .

Given (the code of) an arbitrary machine  $M$ , we don't know which input(s) it accepts, if any. If we wanted to determine whether  $M$  accepts at least one input from some *finite* set  $S = \{x_1, \dots, x_n\}$ , then similar to how we showed that the union of two recognizable languages is recognizable, we could alternate among simulated executions of  $M$  on each

element  $x_i \in S$ , until one of those executions accepts (if ever). Unfortunately, in the present context, the set of possible inputs to  $M$  is the (countably) *infinite* set  $\Sigma^* = \{x_1, x_2, x_3, \dots\}$ , so this basic alternation technique does not work—it would run one step of  $M(x_1)$ , then one step of  $M(x_2)$ , then one step of  $M(x_3)$ , and so on, without ever returning to  $M(x_1)$ .

More explicitly, we want to simulate running  $M$  on infinitely many inputs  $x \in \Sigma^*$  for an unbounded number of steps  $s \in \mathbb{N}^+$  “in parallel,” so that simulation will ultimately accept if (and only if)  $M$  accepts at least one of the inputs  $x_i$  within some (finite) number of steps  $s_j$ . Essentially, we have the product  $\Sigma^* \times \mathbb{N}^+$  of two countably infinite sets, and we need to come up with an ordering so that we will eventually reach each element  $(x_i, s_j) \in \Sigma^* \times \mathbb{N}^+$  of this product.

Similarly to how our enumeration of the integers (Example 74) inspired the idea of alternating simulations of *two* executions, our enumeration of the rationals (Example 75) inspires a similar scheduling of (countably) *infinitely many* executions, as illustrated below. This process is called *dovetailing*.

		step							
		1	2	3	4	5	...		
input		(ε, 1)	(ε, 2)	(ε, 3)	(ε, 4)	(ε, 5)			
ε		(ε, 1)	(ε, 2)	(ε, 3)	(ε, 4)	(ε, 5)			
0		(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)			
1		(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)			
00		(00, 1)	(00, 2)	(00, 3)	(00, 4)	(00, 5)			
01		(01, 1)	(01, 2)	(01, 3)	(01, 4)	(01, 5)			
10		(10, 1)	(10, 2)	(10, 3)	(10, 4)	(10, 5)			
11		(11, 1)	(11, 2)	(11, 3)	(11, 4)	(11, 5)			
000		(000, 1)	(000, 2)	(000, 3)	(000, 4)	(000, 5)			
⋮									

Our simulation proceeds in rounds. In round  $i$ , we perform a single additional step of each execution whose input index is at most  $i$ . More precisely, we define  $R$  as follows, where  $\Sigma^* = \{x_1, x_2, x_3, \dots\}$  is our standard enumeration of all input strings.

```

function  $R(\langle M \rangle)$ 
    for  $i = 1, 2, \dots$  do
        for  $j = 1$  to  $i$  do simulate one more step of  $M(x_j)$ 
        if that step accepts then accept
    
```

In the first round,  $R$  simulates the first step of the execution of  $M$  on the first input  $x_1$ . In the second round,  $R$  simulates another step on  $x_1$ , and the first one on  $x_2$ . In the third round,  $R$  simulates another step on  $x_1$  and  $x_2$ , and the first step on  $x_3$ , and so on. Observe that any particular round  $i$  executes a finite number  $i$  of steps in total, so the next round will run. And, any particular input  $x_i$  begins its execution in round  $i$ , i.e., after a finite number of steps, and its execution continues in every subsequent round until it halts (if ever).

More formally, we analyze the behavior of  $R$  as follows. By inspection of its code,  $R(\langle M \rangle)$  accepts only if some

(simulated)  $M(x_i)$  accepts, i.e.,  $R(\langle M \rangle)$  accepts  $\implies \langle M \rangle \in \overline{L_\emptyset}$ . For the other direction, if  $\langle M \rangle \in \overline{L_\emptyset}$ , then there exists some  $x_i \in \Sigma^*$  and  $s_j \in \mathbb{N}^+$  for which  $M(x_i)$  accepts on step  $s_j$ . Then  $R(\langle M \rangle)$  will accept in round  $i + j - 1$ , because it runs the first step of the simulation on  $x_i$  in round  $i$ , and it runs one more step of that simulation in each subsequent round. So, we have shown that  $\langle M \rangle \in \overline{L_\emptyset} \iff R(\langle M \rangle)$  accepts, as needed.

An interesting point is that  $R$  *does not reject* any input, i.e., for every  $\langle M \rangle \in L_\emptyset$ , we have that  $R(\langle M \rangle)$  loops. This is because  $R$  keeps simulating  $M$  on an endless supply of additional inputs, searching for one that  $M$  accepts. But because there is no such input, the search never ends.

To recap, we have seen that  $L_\emptyset$  is undecidable, and that its complement  $\overline{L_\emptyset}$  is recognizable, so by Corollary 122,  $L_\emptyset$  is unrecognizable.

**Exercise 126** Determine, with proof, whether the following language is recognizable, co-recognizable, or both:

$$L_{\text{REJ}} = \{(\langle M \rangle, x) : M \text{ rejects } x\}.$$

**Exercise 127** Determine, with proof, whether the following language is recognizable, co-recognizable, or both:

$$L_{\text{HATES-EVENS}} = \{\langle M \rangle : M \text{ does not accept any even number}\}.$$

---

CHAPTER  
THIRTEEN

---

## RICE'S THEOREM

We previously proved (see Example 106) that

$$L_\emptyset = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \emptyset\}$$

is undecidable.

Let's define a similar language and determine whether it too is undecidable. Define

$$L_{\{\varepsilon\}} = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = \{\varepsilon\}\}.$$

As we saw with  $L_\emptyset$  and  $\emptyset = \{\}$ , the languages  $L_{\{\varepsilon\}}$  and  $\{\varepsilon\}$  are **not the same**. The latter is the language containing only the single element  $\varepsilon$ , and it is decidable because it is decided by the following Turing machine:

```
function  $D_{\{\varepsilon\}}(x)$ 
  if  $x = \varepsilon$  then accept
  reject
```

On the other hand,  $L_{\{\varepsilon\}}$  is an *infinite* set consisting of (the codes of) all Turing machines that accept only the empty string  $\varepsilon$ .

Here we prove that  $L_{\text{ACC}} \leq_T L_{\{\varepsilon\}}$ , hence  $L_{\{\varepsilon\}}$  is undecidable. Suppose that  $E$  is an oracle that decides  $L_{\{\varepsilon\}}$ . We claim that the following Turing machine  $C$  decides  $L_{\text{ACC}}$  given access to  $E$ :

```
function  $C(\langle M \rangle, x)$  construct the following Turing machine:
  function  $M'(w)$ 
    if  $w = \varepsilon$  then return  $M(x)$ 
    else reject
  return  $E(\langle M' \rangle)$ 
```

We analyze  $C$  as follows. Observe that if  $M(x)$  accepts, then  $L(M') = \{\varepsilon\}$ , because  $M'$  accepts on  $w = \varepsilon$  and rejects all other  $w$ . Whereas if  $M(x)$  does not accept, then  $L(M') = \emptyset \neq \{\varepsilon\}$  because  $M'$  does not accept any string. So overall,  $M(x)$  accepts if and only if  $L(M') = \{\varepsilon\}$ , which implies that  $(\langle M \rangle, x) \in L_{\text{ACC}} \iff C(\langle M \rangle, x)$  accepts. And since  $C$  halts on any input by inspection,  $C$  decides  $L_{\text{ACC}}$ .

We can generalize the reasoning from the previous two proofs as follows.

**Theorem 128** *Let  $A$  be a recognizable language, and define the language*

$$L_A = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) = A\}$$

*consisting of (the codes of) all Turing machines  $M$  for which  $L(M) = A$ . Then  $L_A$  is undecidable.*

Before proving the theorem, we remark that since  $A$  is recognizable, there exists a Turing machine  $R$  for which  $L(R) = A$ . In fact, there are infinitely many such machines. Given any Turing machine  $R$  that recognizes  $A$ , we can construct a distinct Turing machine  $R'$  that also recognizes  $A$ : letting  $Q$  be the set of states of  $R$ , we construct  $R'$  by adding to  $Q$  a new state  $q'$  that has no incoming transitions; this does not affect the behavior of the machine. Since  $L_A$  has infinitely many elements, it cannot be decided by just hardcoding its elements into a Turing machine. Indeed,  $L_A$  is undecidable, as we now prove.

**Proof 129** Assume that  $A \neq \emptyset$ , since we have already shown that  $L_\emptyset$  is undecidable (see Example 106).

We show that  $L_{\text{ACC}} \leq L_A$ . Letting  $D$  be an oracle that decides  $L_A$ , we construct a Turing machine  $C$  that decides  $L_{\text{ACC}}$  given access to  $D$ . Since  $A$  is recognizable, there exists a Turing machine  $R$  that recognizes  $A$ , i.e.,  $L(R) = A$ , which we also use in our construction of  $C$ .

```

function  $C(\langle M \rangle, x)$  construct the following Turing machine:
  function  $M'(w)$ 
    if  $M(x)$  accepts then return  $R(w)$ 
    else reject
  return  $D(\langle M' \rangle)$ 

```

(Note that it is important here that  $R$  is a *Turing machine*, not a “black-box” oracle, because we need to build its *code* into the code of the Turing machine  $M'$ .)

We first analyze the behavior of  $M'$ :

- If  $M(x)$  accepts, then  $L(M') = A$ , because  $M'$  accepts exactly those strings that  $R$  accepts, and  $L(R) = A$  by hypothesis.
- Conversely, if  $M(x)$  does not accept, then  $L(M') = \emptyset \neq A$ , because  $M'(w)$  either loops (if  $M(x)$  loops) or rejects (if  $M(x)$  rejects) for *any* input  $w$ .

So altogether, we have the key property that  $(\langle M \rangle, x) \in L_{\text{ACC}}$  if and only if  $L(M') = A$ .

The behavior of  $C$  on an arbitrary input  $(\langle M \rangle, x)$  is as follows. First,  $C$  halts, because it just constructs  $M'$  and queries  $D$  on it, which halts. Second, by the key property above

$$\begin{aligned}
 (\langle M \rangle, x) \in L_{\text{ACC}} &\iff L(M') = A \\
 &\iff D(\langle M' \rangle) \text{ accepts} \\
 &\iff C(\langle M \rangle, x) \text{ accepts} .
 \end{aligned}$$

So by Definition 61,  $C$  decides  $L_{\text{ACC}}$ , as claimed. □

It is worth mentioning that Theorem 128 holds only for *recognizable* languages  $A$ . If  $A$  is *unrecognizable*, then  $L_A$  is actually decidable! This is because by definition of unrecognizable, there is *no* Turing machine  $M$  for which  $L(M) = A$ , so  $L_A = \emptyset$ , which is trivially decided by the machine that rejects all inputs.

We can further generalize Theorem 128 by considering *sets* of languages, rather than just a single language in isolation.

**Definition 130 (Semantic Property)** A *semantic property* is a set of languages.

We often use the symbol  $\mathbb{P}$  to represent a semantic property. The following is an example of a semantic property:

$$\mathbb{P}_\infty = \{L \subseteq \Sigma^* : |L| \text{ is infinite}\} .$$

$\mathbb{P}_\infty$  is the set of languages that have infinitely many elements. Examples of languages in  $\mathbb{P}_\infty$  include  $\Sigma^*$ ,  $\{x \in \Sigma^* : x \text{ has no } 1\text{s}\}$  (assuming  $\Sigma \neq \{1\}$ ), and every undecidable language (but not every decidable language, because some of these are finite). Examples of languages that are not in  $\mathbb{P}_\infty$  include  $\emptyset$  and  $\{x \in \Sigma^* : |x| < 100\}$ .

**Definition 131 (Trivial Semantic Property)** A semantic property  $\mathbb{P}$  is *trivial* if either *every* recognizable language is in  $\mathbb{P}$ , or *no* recognizable language is in  $\mathbb{P}$ .

Examples of trivial properties include:

- $\mathbb{P} = \emptyset$ , which has no recognizable language in it;
- the set of all recognizable languages  $\mathbb{P} = \text{RE}$ , which has every recognizable language in it;
- the set of all unrecognizable languages  $\mathbb{P} = \overline{\text{RE}}$ , which has no recognizable language in it; and
- the set of all languages  $\mathbb{P} = \mathcal{P}(\Sigma^*)$ , which has every recognizable language in it.

Example of nontrivial properties include:

- $\mathbb{P} = \{\emptyset, \Sigma^*\}$ , because it has the recognizable language  $\emptyset$  but not the recognizable language  $\{\varepsilon\}$ ;
- $\mathbb{P} = \{L_{\text{BARBER}}, L_{\text{ACC}}, L_{\text{HALT}}, \text{every finite language } L\}$ , because it has the recognizable language  $L_{\text{ACC}}$  but not the recognizable language  $\overline{L_\emptyset}$ ; and
- the set of *decidable* languages  $\mathbb{P} = \text{R}$ , because some recognizable languages (like  $\Sigma^*$ ) are decidable, and other recognizable languages (like  $L_{\text{ACC}}$  and  $L_{\text{HALT}}$ ) are not decidable.

We now generalize our notion of a language of (codes of) Turing machines, from machines that recognize a *specific* language  $A$ , to machines that recognize *some* language in a semantic property  $\mathbb{P}$ . Define  $L_{\mathbb{P}}$  as follows:

$$L_{\mathbb{P}} = \{\langle M \rangle : M \text{ is a Turing machine and } L(M) \in \mathbb{P}\}.$$

The next two results show that whether  $L_{\mathbb{P}}$  is decidable is determined entirely by whether  $\mathbb{P}$  is trivial.

**Claim 132** If  $\mathbb{P}$  is a trivial semantic property, then  $L_{\mathbb{P}}$  is decidable.

**Proof 133** If  $\mathbb{P}$  is trivial, then it either has all recognizable languages, or no recognizable language.

- **Case 1:**  $\mathbb{P}$  has all recognizable languages. Then the code of *every* Turing machine  $M$  is in  $L_{\mathbb{P}}$ : since  $L(M)$  is recognizable by definition,  $L(M) \in \mathbb{P}$ , so  $\langle M \rangle \in L_{\mathbb{P}}$ . So,  $L_{\mathbb{P}}$  is decided by the machine that accepts any valid Turing-machine code.
- **Case 2:**  $\mathbb{P}$  has no recognizable language. Then the code of *no* program  $M$  is in  $L_{\mathbb{P}}$ : since  $L(M)$  is recognizable by definition,  $L(M) \notin \mathbb{P}$ , so  $\langle M \rangle \notin L_{\mathbb{P}}$ . So,  $L_{\mathbb{P}}$  is decided by the machine that just rejects its input.

We now state and prove a very general and powerful theorem known as *Rice's theorem*.

**Theorem 134 (Rice's Theorem)** If  $\mathbb{P}$  is a nontrivial semantic property, then  $L_{\mathbb{P}}$  is undecidable.

**Proof 135** We show that  $L_{\text{ACC}} \leq_T L_{\mathbb{P}}$ , hence  $L_{\mathbb{P}}$  is undecidable. Letting  $D$  be an oracle that decides  $L_{\mathbb{P}}$ , we construct a Turing machine  $C$  that decides  $L_{\text{ACC}}$  given access to  $D$ .

First, we consider the case where  $\emptyset \notin \mathbb{P}$ . Since  $\mathbb{P}$  is nontrivial, there is some recognizable language  $A \in \mathbb{P}$ , and  $A \neq \emptyset$  because  $\emptyset \notin \mathbb{P}$ . Since  $A$  is recognizable, there is a Turing machine  $R_A$  that recognizes it. We construct  $C$  exactly as in the proof of [Theorem 128](#):

```
function  $C(\langle M \rangle, x)$  construct the following Turing machine:  

function  $M'(w)$   

  if  $M(x)$  accepts then return  $R_A(w)$   

  elsereject  

return  $D(\langle M' \rangle)$ 
```

Very similar to our analysis in the proof of [Theorem 128](#), because  $A \in \mathbb{P}$  and  $\emptyset \notin \mathbb{P}$ , we have the key property that  $(\langle M \rangle, x) \in L_{\text{ACC}}$  if and only if  $L(M') \in \mathbb{P}$ . The analysis of the behavior of  $C$  is also very similar:  $C$  halts on every input, and  $(\langle M \rangle, x) \in L_{\text{ACC}} \iff C(\langle M \rangle, x)$  accepts, so  $C$  decides  $L_{\text{ACC}}$ , as needed.

The case where  $\emptyset \in \mathbb{P}$  proceeds symmetrically: by nontriviality, there is some recognizable language  $B \notin \mathbb{P}$  with  $B \neq \emptyset$ , and some Turing machine  $R_B$  that recognizes  $B$ . Then we construct  $C$  as follows (the only difference with the  $C$  constructed above is that it uses the recognizer  $R_B$  instead of  $R_A$ , and it negates the output of the oracle  $D$ ):

```
function  $C(\langle M \rangle, x)$  construct the following Turing machine:  

function  $M'(w)$   

  if  $M(x)$  accepts then return  $R_B(w)$   

  elsereject  

return the opposite of  $D(\langle M' \rangle)$ 
```

The analysis is symmetric to the above, based on the key property that  $(\langle M \rangle, x) \in L_{\text{ACC}}$  if and only if  $L(M') \notin \mathbb{P}$ . (This is why  $C$  negates the answer of  $D$ .) So,  $C$  decides  $L_{\text{ACC}}$ , as needed.  $\square$

## 13.1 Rice's Theorem and Program Analysis

Rice's theorem has profound implications for compilers and program analysis. While the formulation given above is with respect to the *language* of a Turing machine, the typical expression of Rice's theorem in the field of programming languages is more expansive:

Any “nontrivial” question about the runtime behavior of a given program is undecidable.

Here, “nontrivial” essentially means there are some programs that have the behavior in question, and others that do not.<sup>36</sup>

As an example, let's consider the question of whether a given Turing machine, when run on a given input  $x$ , ever writes the symbol 1 to its tape. The language is defined as follows:

$$L_{\text{WritesOne}} = \{(\langle M \rangle, x) : M \text{ is a Turing machine and } M(x) \text{ writes a 1 to its tape at some point}\}.$$

This language is not in the form required by our formulation of Rice's theorem, both because the membership condition is not determined by  $L(M)$ , and because the input has an extra argument  $x$ . However, we can show that it is undecidable via reduction, by proving that  $L_{\text{HALT}} \leq_T L_{\text{WritesOne}}$ . That is, given an oracle  $W$  that decides  $L_{\text{WritesOne}}$ , we can construct a Turing machine  $H$  that decides  $L_{\text{HALT}}$ .

<sup>36</sup> The behavior in question must be concerned with program meaning, or *semantics*, and it must be robust with respect to semantics-preserving transformations. For instance, a question like, “Does the program halt in less than a thousand steps?” is not about the program's meaning, and the answer is different for different programs that have identical semantics, so the question is not robust.

First, given any Turing machine  $M$ , we can construct a new machine  $M'$  having the property that  $M'$  writes a 1 to its tape if and only if  $M$  halts (when  $M$  and  $M'$  are run on the same input). Start by introducing a new symbol  $\hat{1}$  to the tape alphabet of  $M'$ , and modifying the transition function so that it interprets both 1 and  $\hat{1}$  as 1 when *reading* from the tape, but it always *writes*  $\hat{1}$  instead of 1. (This distinction is needed because the input string may have 1s in it.) Then, replace  $q_{\text{acc}}$  and  $q_{\text{rej}}$  with new states  $q'_a$  and  $q'_r$ , add new accept/reject states  $q'_{\text{acc}}/q'_{\text{rej}}$ , and include the transitions  $\delta(q'_a, \gamma) = (q'_{\text{acc}}, 1, R)$  and  $\delta(q'_r, \gamma) = (q'_{\text{rej}}, 1, R)$  for all  $\gamma$  in the modified tape alphabet. For any input  $x$ , by construction,  $M'(x)$  reaches  $q'_a$  or  $q'_r$  if and only if  $M(x)$  halts, then  $M'(x)$  writes a 1 in the next step, and this is the *only* way for  $M'(x)$  to write a 1. Thus,  $M'(x)$  writes a 1 if and only if  $M(x)$  halts.

Using the above transformation, we define the reduction  $H$  as follows:

```
function  $H(\langle M \rangle, x)$  construct  $M'$  from  $M$  as described above
return  $W(\langle M' \rangle, x)$ 
```

Clearly,  $H$  halts on any input. And since  $M'(x)$  writes a 1 if and only if  $M(x)$  halts,  $W$  accepts  $(\langle M' \rangle, x)$  if and only if  $(\langle M \rangle, x) \in L_{\text{HALT}}$ . So,  $H$  decides  $L_{\text{HALT}}$ , as needed.

A similar reduction exists for any nontrivial question about program behavior, for any Turing-complete language. (We need the language to be Turing-complete because the reduction embeds an arbitrary Turing machine into a program in that language.) The implication is that there is no algorithm that does perfect program analysis; any such analysis produces the wrong result (or no result at all) for some nonempty set of inputs.

### Soundness and Completeness

Suppose we wish to analyze a given program for some nontrivial program behavior  $B$ . Define the language

$$L_B = \{\langle M \rangle : M \text{ has behavior } B\}.$$

We might hope for an analysis algorithm  $A$  having the following properties:

- if  $\langle M \rangle \in L_B$ , then  $A$  accepts  $\langle M \rangle$ ;
- if  $\langle M \rangle \notin L_B$ , then  $A$  rejects  $\langle M \rangle$ .

Unfortunately, we have seen that such an algorithm does not exist, because  $L_B$  is undecidable. We must settle for an imperfect analysis. We have several choices:

- A *complete* analysis accepts all inputs that have the behavior  $B$  (i.e., those in  $L_B$ ), but also may accept or loop on some that do not have behavior  $B$ . That is, a complete analysis  $A_C$  satisfies:
  - if  $\langle M \rangle \in L_B$ , then  $A_C(\langle M \rangle)$  accepts;
  - for  $\langle M \rangle \notin L_B$ , there is no general guarantee about what  $A_C(\langle M \rangle)$  does—it may accept, reject, or loop.

So, if a complete analysis *rejects* a program, then the program is guaranteed *not* to have the behavior  $B$ . But if the analysis *accepts* the program, it may or may not have behavior  $B$ . (And in general, we cannot detect if the analysis loops on a program.)

Observe that the following analysis is complete, but useless: ignore the program and just accept. In order to be useful, a complete analysis will usually have some guarantee that it rejects any program from some important *subclass* of those that do not have behavior  $B$  (but not all such programs).

- A *sound* analysis rejects all inputs that do not have behavior  $B$  (i.e., those not in  $L_B$ ), but also may reject or loop on some that do have behavior  $B$ . That is, a sound analysis  $A_S$  satisfies:
  - for  $\langle M \rangle \in L_B$ , there is no general guarantee on what  $A_S(\langle M \rangle)$  does—it may accept, reject, or loop;
  - if  $\langle M \rangle \notin L_B$ , then  $A_S(\langle M \rangle)$  rejects.

So, if a sound analysis *accepts* a program, then the program is guaranteed to have the behavior  $B$ . But if the analysis *rejects* the program, it may or may not have behavior  $B$ .

Observe that the following analysis is sound, but useless: ignore the program and just reject. In order to be useful, a sound analysis will usually have some guarantee that it accepts any program from some important *subclass* of those that have behavior  $B$  (but not all such programs).

- An analysis that is both *incomplete* and *unsound* might answer incorrectly on any input program, regardless of whether the program has behavior  $B$ . So, the output of the analysis does not guarantee anything about the program's behavior.

Because an incomplete and unsound analysis comes with no guarantees about its output in any case, we typically design analyses that are either complete or sound (but not both).

A *type system* is a concrete example of a program analysis, and the usual choice for static type systems is to use a *sound* analysis, rather than a complete one. This means that if the compiler accepts a program, we are guaranteed that the program will not have a type error at runtime—it is “safe”. However, the compiler does reject some programs that are free of runtime type errors. The burden is on the programmer to write their program in such a way that the compiler can guarantee it is “safe”. Often, the compiler can provide some useful guarantee that it will accept any “safe” program from some broad class.

# **Part III**

# **Complexity**

---

CHAPTER  
FOURTEEN

---

## INTRODUCTION TO COMPLEXITY

In the previous unit, we considered which computational problems are solvable by algorithms, without any constraints on the time and memory used by the algorithm. We saw that even with unlimited resources, many problems—indeed, “most” problems—are not solvable by any algorithm.

In this unit on computational complexity, we consider problems that *are* solvable by algorithms, and focus on *how efficiently* they can be solved. Primarily, we will be concerned how much *time* it takes to solve a problem.<sup>37</sup> We mainly focus on *decision* problems—i.e., languages—then later broaden our treatment to *functional*—i.e., search—problems.

A Turing machine’s running time, also known as *time complexity*, is the number of steps it takes until it halts; similarly, its *space complexity* is the number of tape cells it uses. We focus primarily on time complexity, and as previously discussed (page 3), we are interested in the *asymptotic* complexity with respect to the input size, in the worst case. For any particular asymptotic bound  $O(t(n))$ , we define the set of languages that are decidable by Turing machines having time complexity  $O(t(n))$ , where  $n$  is the input size:

**Definition 136 (DTIME)** Define

$$\text{DTIME}(t(n)) = \{L \subseteq \Sigma^* : L \text{ is decided by some Turing machine with time complexity } O(t(n))\}.$$

The set  $\text{DTIME}(t(n))$  is an example of a *complexity class*: a set of languages whose complexity in some metric of interest is bounded in some specified way. Concrete examples include  $\text{DTIME}(n)$ , the class of languages that are decidable by Turing machines that run in (at most) linear  $O(n)$  time; and  $\text{DTIME}(n^2)$ , the class of languages decidable by quadratic-time Turing machines.

When we discussed computability, we defined two classes of languages: the decidable languages (also called *recursive*), denoted by  $\text{R}$ , and the recognizable languages (also called *recursively enumerable*), denoted by  $\text{RE}$ . The definitions of these two classes are actually *model-independent*: they contain the same languages, regardless of whether our computational model is Turing machines, lambda calculus, or some other sensible model, as long as it is Turing-equivalent.

Unfortunately, this kind of model independence does not extend to  $\text{DTIME}(t(n))$ . Consider the concrete language

$$\text{PALINDROME} = \{x : x = x^R, \text{i.e., } x \text{ equals its reversal}\}.$$

In a computational model with random-access memory, or even in the two-tape Turing-machine model,  $\text{PALINDROME}$  can be decided in linear  $O(n)$  time, where  $n = |x|$ : just walk pointers inward from both ends of the input, comparing character by character.

However, in the standard one-tape Turing-machine model, it can be proved that there is *no*  $O(n)$ -time algorithm for  $\text{PALINDROME}$ ; in fact, it requires  $\Omega(n^2)$  time to decide. Essentially, the issue is that an algorithm to decide this language would need to compare the first and last symbols of  $x$ , which requires moving the head sequentially over the entire string, and do similarly for second and second-to-last symbols of  $x$ , and so on. Because  $n/4$  of the pairs require

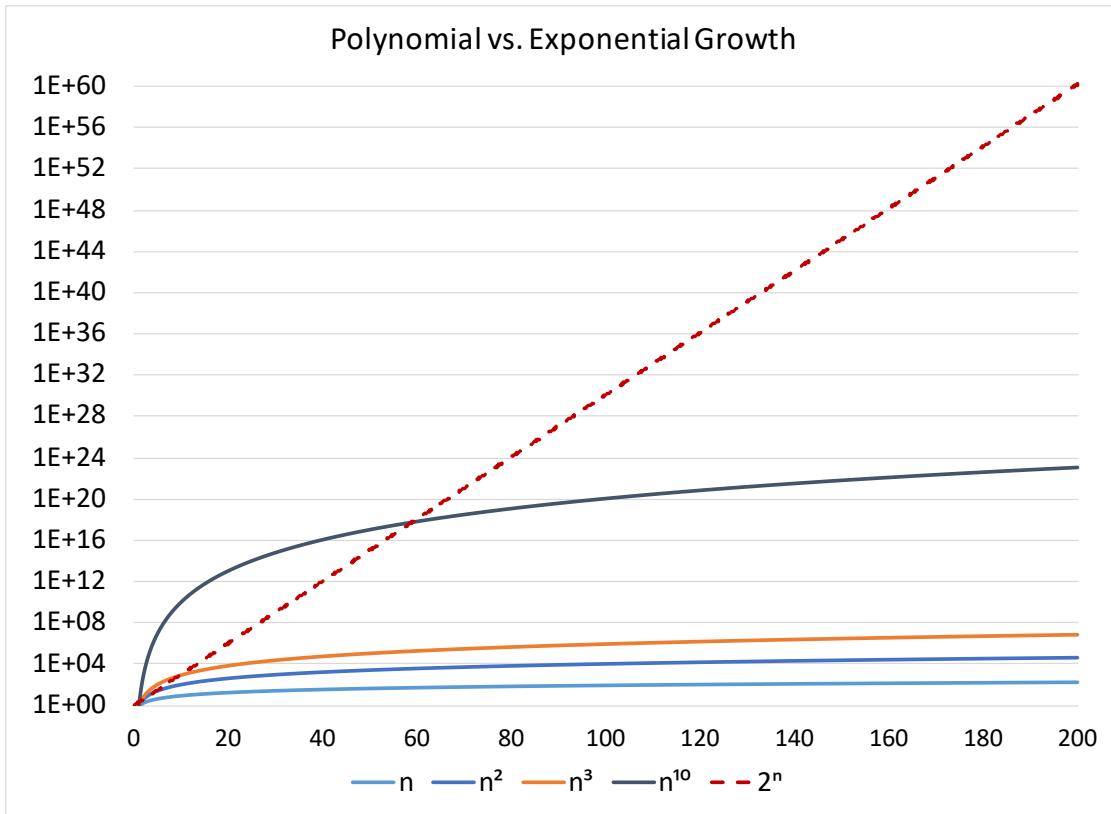
<sup>37</sup> More generally, the field of computational complexity is concerned with quantifying all kinds of different resources, like time, memory, randomness, etc.; and with understanding various kinds of computational models, like Turing machines, branching programs, formulas, nondeterminism, interaction, etc.

moving the head by at least  $n/2$  cells each, this results in a total running time of  $\Omega(n^2)$ .<sup>38</sup> So, PALINDROME  $\notin$  DTIME( $n$ ), but in other computational models, PALINDROME can be decided in  $O(n)$  time.

A model-dependent complexity class is very inconvenient, because we wish to analyze algorithms at a higher level of abstraction, without worrying about the details of the underlying computational model, or how the algorithm would be implemented on it, like the particulars of data structures (which can affect asymptotic running times).

## 14.1 Polynomial Time and the Class P

We wish to define a complexity class that captures all problems that can be solved “efficiently”, and is also model-independent. As a step toward this goal, we note that there is an enormous qualitative difference between the growth of *polynomial* functions versus *exponential* functions. The following illustrates how several polynomials in  $n$  compare to the exponential function  $2^n$ :



The vertical axis in this plot is logarithmic, so that the growth of  $2^n$  appears as a straight line. Observe that even a polynomial with a fairly large exponent, like  $n^{10}$ , grows much slower than  $2^n$  (except for small  $n$ ), with the latter exceeding the former for all  $n \geq 60$ .

In addition to the dramatic difference in growth between polynomials and exponentials, we also observe that polynomials have nice *closure* properties: if  $f(n), g(n)$  are polynomially bounded, i.e.,  $f(n) = O(n^c)$  and  $g(n) = O(n^{c'})$

<sup>38</sup> We caution that the above is *not a proof* that deciding PALINDROME requires  $\Omega(n^2)$  time on a standard Turing machine. A correct proof would need to consider *all* Turing machines, even ones that use very clever strategies we might not be able to imagine. Such a proof is known, though it is quite subtle.

for some constants  $c, c'$ , then

$$\begin{aligned} f(n) + g(n) &= O(n^{\max\{c, c'\}}), \\ f(n) \cdot g(n) &= O(n^{c+c'}), \\ f(g(n)) &= O(n^{c \cdot c'}) \end{aligned}$$

are polynomially bounded as well, because  $\max\{c, c'\}$ ,  $c + c'$ , and  $c \cdot c'$  are constants. These facts can be used to prove that if we compose a polynomial-time algorithm that uses some subroutine with a polynomial-time implementation of that subroutine, then the resulting full algorithm is also polynomial time. This composition property allows us to obtain polynomial-time algorithms in a *modular* way, by designing and analyzing individual components in isolation.

Thanks to the above properties of polynomials, it has been shown that the notion of polynomial-time computation is “robust” across many popular computational models, like the many variants of Turing machines, lambda calculi, etc. That is, any of these models can simulate any other one with only polynomial “slowdown”. So, any particular problem is solvable in polynomial time either in *all* such models, or in *none* of them.

The above considerations lead us to define “efficient” to mean “polynomial time (in the input size)”. From this we get the following model-independent complexity class of languages that are decidable in polynomial time (across many models of computation).

**Definition 137 (Complexity Class P)** The complexity class  $P$  is defined as

$$\begin{aligned} P &= \bigcup_{k \geq 1} \text{DTIME}(n^k) \\ &= \{L : L \text{ is decided by some polynomial-time TM}\}. \end{aligned}$$

In other words, a language  $L \in P$  if it is decided by some Turing machine that runs in time  $O(n^k)$  for some constant  $k$ .

In brief,  $P$  is the class of “efficiently decidable” languages. Based on what we saw in the algorithms unit, this class includes many fundamental problems of interest, like the *decision* versions of the greatest common divisor problem, sorting, the longest increasing subsequence problem, etc. Note that since  $P$  is a class of *languages*, or *decision* problems, it technically does not include the *search* versions of these problems—but see below for the close relationship between search and decision.

As a final remark, the *extended Church-Turing thesis* posits that the notion of polynomial-time computation is “robust” across *all* realistic models of computation:

**Theorem 138 (Extended Church-Turing Thesis)** *A problem is solvable in polynomial time on a Turing machine if and only if it is solvable in polynomial time in any “realistic” model of computation.*

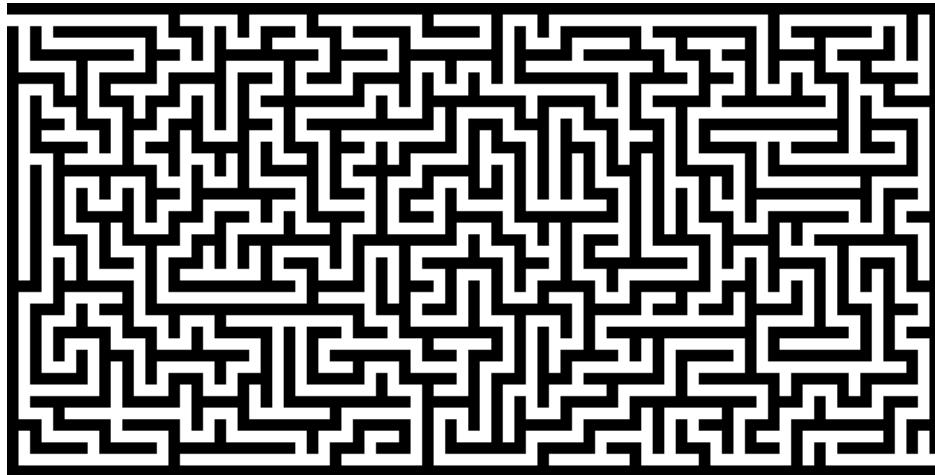
Because “realistic” is not precisely defined, this is just a thesis, not a statement than can be proved. Indeed, this is a major strengthening of the standard Church-Turing thesis, and there is no strong agreement about whether it is even true! In particular, the model of *quantum* computation may pose a serious challenge to this thesis: it is known that polynomial-time quantum algorithms exist for certain problems, like factoring integers into their prime divisors, that we *do not know how to solve in polynomial time on Turing machines!* So, if quantum computation is “realistic”, and if there really is no polynomial-time factoring algorithm in the TM model, then the extended Church-Turing thesis is false. While both of these hypotheses seem plausible, they are still uncertain at this time: real devices that can implement the full model of quantum computation have not yet been built, and we do not have any *proof* that factoring integers (or any other problem that quantum computers can solve in polynomial time) *requires* more than polynomial time on a Turing machine.

## 14.2 Examples of Efficient Verification

For some computational problems, it is possible to efficiently *verify* whether a *claimed solution* is actually correct—regardless of whether *computing* a correct solution “from scratch” can be done efficiently. Examples of this phenomenon are abundant in everyday life, and include:

- **Logic and word puzzles** like mazes, Sudoku, or crosswords: these come in various degrees of difficulty to solve, some quite challenging. But given a proposed solution, it is straightforward to check whether it satisfies the “rules” of the puzzle. (See below for a detailed example with mazes.)
- **Homework problems** that ask for correct software code (with justification) or mathematical proofs: producing a good solution might require a lot of effort and creativity to discover the right insights. But given a candidate solution, one can check relatively easily whether it is clear and correct, simply by applying the rules of logic. For example, to verify a claimed mathematical proof, we just need to check whether each step follows logically from the hypotheses and the previous steps, and that the proof reaches the desired conclusion.
- **Music, writing, video, and other media:** creating a high-quality song, book, movie, etc. might require a lot of creativity, effort, and expertise. But given such a piece of media, it is relatively easy for even a non-expert to decide whether it is engaging and worthwhile to them (even though this is a subjective judgment that may vary from person to person). For example, even though the authors of this text could not come close to writing a beautiful symphony or a hit pop song, we easily know one when we hear one.

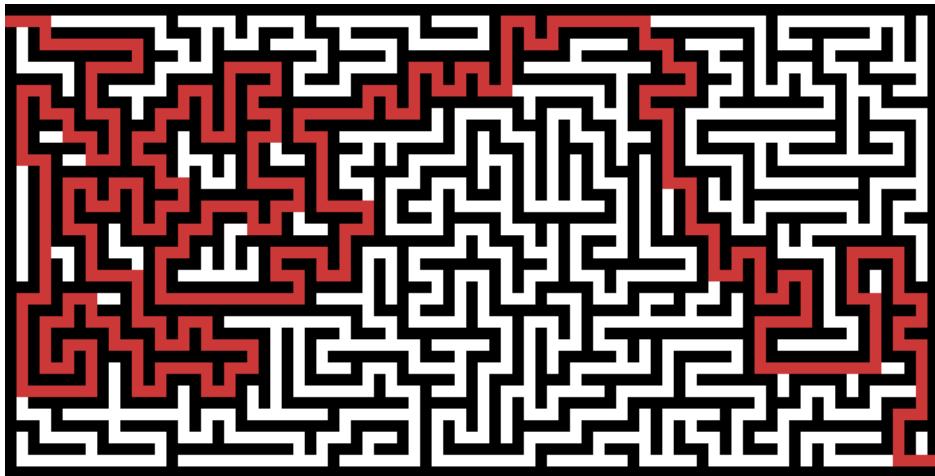
As a detailed example, consider the following maze, courtesy of Kees Meijer’s maze generator<sup>39</sup>:



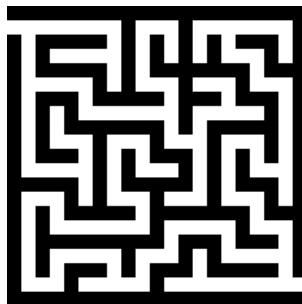
At first glance, it is not clear whether this maze has a solution. However, suppose that someone—perhaps a very good solver of mazes, or the person who constructed the maze—were to claim that this maze is indeed solvable. Is there some way that they could convince us of this fact?

A natural idea is simply to provide us with a path through the maze as “proof” of the claim. It is easy for us to check that this proof is valid, by verifying that the path goes from start to finish without crossing any “wall” of the maze. By definition, any solvable maze has such a path, so it is possible to convince us that a solvable maze really is solvable.

<sup>39</sup> <https://keesiemeijer.github.io/maze-generator/>



On the other hand, suppose that a given maze *does not* have a solution. Then no matter what *claimed* “proof” someone might give us, the checks we perform will cause us to reject it: because the maze is not solvable, any path must either fail to go from start to finish, or cross a wall somewhere (or both).



In summary: for any given maze, checking that a claimed solution goes from start to finish without crossing any wall is an *efficient verification* procedure:

- The checks can be performed efficiently, i.e., in polynomial time in the size of the maze.
- If the maze is solvable, then *there exists* a “proof”—namely, a path through the maze from start to finish—that the procedure will accept. *How to find* such a proof is not the verifier’s concern; all that matters is that it exists.<sup>40</sup>
- If the maze is not solvable, then *no* claimed “proof” will satisfy the procedure.

Observe that we need *both* of the last two conditions in order for the procedure to be considered a correct verifier:

- An “overly skeptical” verifier that cannot be “convinced” by anything, even though the maze is actually solvable, would not be correct.
- Similarly, an “overly credulous” verifier that can be “fooled” into accepting, even when the maze is *not* solvable, would also be incorrect.

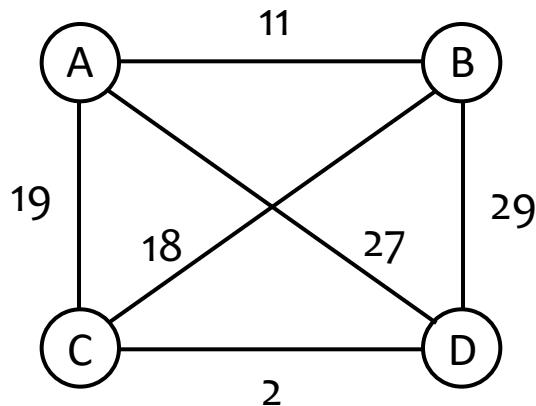
As already argued, our verification procedure has the right balance of “skepticism” and “credulity”.

As another example, the *traveling salesperson problem (TSP)* is the task of finding a minimum-weight *tour* of a given weighted graph. In this text, a “tour” of a graph is a *cycle that visits every vertex exactly once*, i.e., it is a path that visits every vertex and then immediately returns to its starting vertex. (Beware that some texts define “tour” slightly differently.) Without loss of generality, for TSP we can assume that the graph is *complete*—i.e., there is an edge between every pair of vertices—by filling in any missing edges with edges of enormous (or infinite) weight. This does not affect

<sup>40</sup> The reader might have noticed that for any given maze, it is actually possible to use a graph-search algorithm like BFS or DFS to *find* a solution efficiently, when one exists. So, MAZE is even efficiently *decidable*, i.e.,  $\text{MAZE} \in \text{P}$ . This does not contradict anything written above about MAZE also being efficiently *verifiable*. However, it does give us an alternative efficient verifier for MAZE, which just ignores the provided “proof” and determines on its own whether the given maze is solvable. This verifier trivially meets all the efficiency and correctness conditions we have laid out.

the solution(s), because any tour that uses any of these new edges will have larger weight than any tour that does not use any of them.

Suppose we are interested in a minimum-weight tour of the following graph:



If someone were to claim that the path  $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$  is a minimum-weight tour, could we efficiently verify that this is true? There doesn't seem to be an obvious way to do so, apart from considering all other tours and checking whether any of them have smaller weight. While this particular graph only has three distinct tours (ignoring reversals and different starting points on the same cycle), in general, the number of tours in a graph is exponential in the number of vertices, so this approach would not be efficient. So, it is not clear whether we can efficiently verify that a claimed minimum-weight tour really is one.

However, let's modify the problem to be a *decision* problem, which simply asks whether there is a tour whose total weight is *within some specified “budget”*:

Given a weighted graph  $G$  and a budget  $k$ , does  $G$  have a tour of weight at most  $k$ ?

Can the existence of such a tour be proved to an efficient, suitably skeptical verifier? Yes: simply provide such a tour as the proof. The verifier, given a path in  $G$ —i.e., a list of vertices—that is *claimed* to be such a tour, would check that all of the following hold:

1. the path starts and ends at the same vertex,
2. the path visits every vertex in the graph exactly once (except for the repeated start vertex at the end), and
3. the sum of the weights of the edges on the path is at most  $k$ .

(All of these tests can be performed efficiently; see the formalization in [Algorithm 144](#) below for details.)

For example, consider the following *claimed* “proofs” that the above graph has a tour that is within budget  $k = 60$ :

- The path  $A \rightarrow B \rightarrow D \rightarrow C$  does not start and end at the same vertex, so the verifier's first check would reject it as invalid.
- The path  $A \rightarrow B \rightarrow D \rightarrow A$  starts and ends at the same vertex, but it does not visit vertex  $C$ , so the verifier's second check would reject it.
- The path  $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$  satisfies the first two checks, but it has total weight 61, which is not within the budget, so the verifier's third check would reject it.
- The path  $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$  satisfies the first two checks, and its total weight of 58 is within the budget, so the verifier would *accept* it.

These examples illustrate that when the graph has a tour that is within the budget, there are still “proofs” that are “unconvincing” to the verifier—but there will also be a “convincing” proof, which is what matters to us.

On the other hand, it turns out that the above graph does *not* have a tour that is within a budget of  $k = 57$ . And for this graph and budget, the verifier will reject no matter what claimed “proof” it is given. This is because every tour in the graph—i.e., every path that would pass the verifier’s first two checks—has total weight at least 58.

In general, for any given graph and budget, the above-described verification procedure is correct:

- If there is a tour of the graph that is within the budget, then *there exists* a “proof”—namely, such a tour itself—that the procedure will accept. (As before, *how to find* such a proof is not the verifier’s concern.)
- If there is no such tour, then *no* claimed “proof” will satisfy the procedure, i.e., it cannot be “fooled” into accepting.

## 14.3 Efficient Verifiers and the Class NP

We now generalize the above examples to formally define the notion of “efficient verification” for an arbitrary decision problem, i.e., a language. This definition captures the notion of an efficient procedure that can be “convinced”—by a suitable “proof”—to accept any string in the language, but cannot be “fooled” into accepting a string that is not in the language.

**Definition 139 (Efficient Verifier)** An *efficient verifier* for a language  $L$  is a Turing machine  $V(x, c)$  that takes two inputs, an *instance*  $x$  and a *certificate* (or “claimed proof”)  $c$  whose size  $|c|$  is polynomial in  $|x|$ , and satisfies the following properties:

- **Efficiency:**  $V(x, c)$  runs in time polynomial in its input size.<sup>41</sup>
- **Completeness:** if  $x \in L$ , then **there exists** some  $c$  for which  $V(x, c)$  accepts.
- **Soundness:** if  $x \notin L$ , then **for all**  $c$ ,  $V(x, c)$  rejects.

Alternatively, completeness and soundness together are equivalent to the following property (which is often more natural to prove for specific verifiers):

- **Correctness:**  $x \in L$  if and only if there exists some  $c$  (of size polynomial in  $|x|$ ) for which  $V(x, c)$  accepts.

We say that a language is *efficiently verifiable* if there is an efficient verifier for it.

---

<sup>41</sup> Notice that polynomial in the size of  $(x, c)$  is equivalent to polynomial in the size of  $x$  alone, because  $|c|$  is polynomial in  $|x|$ .

The claimed equivalence can be seen by taking the contrapositive of the soundness condition, which is: if there exists some  $c$  for which  $V(x, c)$  accepts, then  $x \in L$ . (Recall that when negating a predicate, “for all” becomes “there exists”, and vice-versa.) This contrapositive statement and completeness are, respectively, the “if” and “only if” parts of correctness.

We sometimes say that a certificate  $c$  is “valid” or “invalid” for a given instance  $x$  if  $V(x, c)$  accepts or rejects, respectively. Note that the *decision of the verifier* is what determines whether a certificate is “(in)valid”—not the other way around—and that the validity of a certificate depends on both the instance *and the verifier*. There can be many different verifiers for the same language, which in general can make different decisions on the same  $x, c$  (though they all must reject when  $x \notin L$ ).

In general, the instance  $x$  and certificate  $c$  are *arbitrary strings* over the verifier’s input alphabet  $\Sigma$  (and they are separated on the tape by some special character that is not in  $\Sigma$ , but is in the tape alphabet  $\Gamma$ ). However, for specific languages,  $x$  and  $c$  will typically represent various mathematical objects like integers, arrays, graphs, vertices, etc. As in the computability unit, this is done via appropriate *encodings* of the objects, denoted by  $\langle \cdot \rangle$ , where without loss of generality *every* string decodes as some object of the desired “type” (e.g., integer, list of vertices, etc.). Therefore, when we write the pseudocode for a verifier, we can treat the instance and certificate as already having the desired types.

### 14.3.1 Discussion of Completeness and Soundness

We point out some of the important aspects of completeness and soundness (or together, correctness) in [Definition 139](#).

1. Notice some similarities and differences between the notions of *verifier* and *decider* ([Definition 61](#)):
  - When the input  $x \in L$ , both kinds of machine must accept, but a verifier need only accept for *some* value of the certificate  $c$ , and may reject for others. By contrast, a decider does not get any other input besides  $x$ , and simply must accept it.
  - When the input  $x \notin L$ , both kinds of machine must reject, and moreover, a verifier must reject for *all* values of  $c$ .
2. There is an *asymmetry* between the definitions of completeness and soundness:
  - If a (sound) verifier for language  $L$  *accepts* some input  $(x, c)$ , then we can correctly conclude that  $x \in L$ , by the contrapositive of soundness. (A sound verifier cannot be “fooled” into accepting an instance that is not in the language.)
  - However, if a (complete) verifier for  $L$  *rejects* some input  $(x, c)$ , then in general we *cannot reach any conclusion* about whether  $x \in L$ : it might be that  $x \notin L$ , or it might be that  $x \in L$  but  $c$  is just not a valid certificate for  $x$ . (In the latter case, by completeness, some other certificate  $c'$  is valid for  $x$ .)

In summary, while every string  $x \in L$  has a “convincing proof” of its membership in  $L$ , a string  $x \notin L$  does not necessarily have a proof of its non-membership in  $L$ .

### 14.3.2 Discussion of Efficiency

We also highlight some important but subtle aspects of the notion of efficiency from [Definition 139](#).

1. First, we restrict certificates  $c$  to have size that is some polynomial in the size of the instance  $x$ , i.e.,  $|c| = O(|x|^k)$  for some constant  $k$ . The specific polynomial can depend on the verifier, but the same polynomial bounds the certificate size for *all* instances of the language. We make this restriction because we want verification to be efficient in every respect. It would not make sense to allow certificates to be (say) exponentially long in the instance size, since even *reading* such a long certificate should not be considered “efficient.”

We emphasize that merely requiring the verifier to have polynomial running time (in the size of the input) would not prohibit such pathological behavior. This is because the verifier’s input is both the instance  $x$  and certificate  $c$ . For example, consider a certificate that consists of all the exponentially many tours in a graph. The verifier could check all of them in time *linear* in the certificate size, and hence linear in the size of its input. But this is not efficient in the sense we want, because it takes exponential time in the size of the graph.

2. With the above in mind, an equivalent definition of verifier efficiency, without any explicit restriction on the certificate size, is “running time polynomial in the size of *the instance  $x$  alone*”—not the entire input  $x, c$ . Such a verifier can read only polynomially many of the initial symbols of  $c$ , because reading each symbol and moving the head takes a computational step. So, the rest of the certificate (if any) is irrelevant, and can be truncated without affecting the verifier’s behavior. Therefore, we can assume without loss of generality that the certificate size is polynomial in the size of the instance  $x$ , as we do in [Definition 139](#) above.

### 14.3.3 The Class NP

With the notion of an efficient verifier in hand, analogously to how we defined P as the class of efficiently decidable languages, we define NP as the class of efficiently verifiable languages.

**Definition 140 (Complexity Class NP)** The complexity class NP is defined as the set of *efficiently verifiable languages*:

$$\text{NP} = \{L : L \text{ is efficiently verifiable}\}.$$

In other words, a language  $L \in \text{NP}$  if there is an efficient verifier for it, according to Definition 139.<sup>42</sup>

<sup>42</sup> We caution that NP **does not** stand for “not polynomial”. It is an abbreviation of the name “*nondeterministic* polynomial,” which comes from an equivalent definition of the class based on the computational model of nondeterministic Turing machines. (This model is beyond the scope of this text.) A possible alternative name would be VP, for “verifiable in polynomial time”.

Let us now formalize our first example of efficient verification from above: the decision problem of determining whether a given maze has a solution. A maze can be represented as an undirected graph, with a vertex for each “intersection” in the maze (along with the start and end points), and edges between adjacent positions. So, the decision problem is to determine whether, given such a graph, there exists a path from the start vertex  $s$  to the end vertex  $t$ . This can be represented as the language<sup>43</sup>

$$\text{MAZE} = \{(G, s, t) : G \text{ is an undirected graph that has a path from vertices } s \text{ to } t\}.$$

We define an efficient verifier for MAZE as follows; the precise pseudocode is given in Algorithm 141. An instance is a graph  $G = (V, E)$ , a start vertex  $s$ , and a target vertex  $t$ . A certificate is a sequence of up to  $|V|$  vertices in the graph. (This limit on the number of vertices in the certificate can be enforced by the decoding, and it ensures that the certificate size is polynomial in the instance size; see the *Discussion of Efficiency* (page 144).) The verifier checks that the sequence describes a valid path in the graph from  $s$  to  $t$ , i.e., that the first and last vertices in the sequence are  $s$  and  $t$  (respectively), and that there is an edge between every pair of consecutive vertices in the sequence.

**Algorithm 141 (Efficient Verifier for MAZE)**

**Input:** instance: a graph and start/end vertices; certificate: a list of up to  $|V|$  vertices  
**Output:** whether the certificate represents a path in the graph from the start to the end

```

function VERIFYMAZE( $(G = (V, E), s, t), c = (v_1, \dots, v_m)$ )
    if  $v_1 \neq s$  or  $v_m \neq t$  then reject
    for  $i = 1$  to  $m - 1$  do
        if  $(v_i, v_{i+1}) \notin E$  then reject
    accept
```

**Lemma 142** VERIFYMAZE is an efficient verifier for MAZE, so MAZE  $\in \text{NP}$ .

**Proof 143** We show that VERIFYMAZE satisfies Definition 139 by showing that it is efficient and correct.

First, VERIFYMAZE runs in time polynomial in the size  $|G| \geq |V|$  of the graph: it compares two vertices from the certificate against the start and end vertices in the instance, and checks whether there is an edge between up to  $|V| - 1$  pairs of consecutive vertices in the certificate. Each pair of vertices can be checked in polynomial time. The exact polynomial depends on the representation of  $G$  and the underlying computational model, but it is polynomial in any reasonable representation (e.g., adjacency lists, adjacency matrix), which is all that matters for our purposes here.

As for correctness:

<sup>43</sup> Henceforth, for notational simplicity we will omit the encoding of inputs (e.g.,  $\langle G, s, t \rangle$ ) when defining languages, taking it to be implicit.

- If  $(G, s, t) \in \text{MAZE}$ , then by definition there is some path  $s = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{m-1} \rightarrow v_m = t$  in  $G$  that visits  $m \leq |V|$  vertices in total, because any cycle in the path can be removed. Then by inspection of the pseudocode,  $\text{VERIFYMAZE}((G, s, t), c = (v_1, v_2, \dots, v_m))$  accepts.
- Conversely, if  $\text{VERIFYMAZE}((G, s, t), c = (v_1, \dots, v_m))$  accepts for some certificate  $c$ , then by inspection of the pseudocode,  $c$  represents a path from  $v_1 = s$  to  $v_m = t$  in  $G$ , so  $(G, s, t) \in \text{MAZE}$  by definition.

Alternatively, instead of showing the “conversely” part of correctness as above, we could have argued that  $\text{VERIFYMAZE}$  is *sound*, as follows: if  $(G, s, t) \notin \text{MAZE}$ , then by definition there is no path between  $s$  and  $t$  in  $G$ , so any certificate  $c$  will either not start at  $s$ , not end at  $t$ , or it will have some pair of consecutive vertices with no edge between them. Thus, by inspection of the pseudocode,  $\text{VERIFYMAZE}((G, s, t), c)$  will reject.

This kind of reasoning is correct, but it is more cumbersome and error prone, since it involves more cases and argues about the *non-existence* of certain objects. Usually, and especially for more complex verifiers, it is easier and more natural to directly prove the correctness condition ( $x \in L$  if and only if there exists  $c$  such that  $V(x, c)$  accepts) instead of soundness.

Next, returning to the “limited-budget” TSP example, we define the corresponding language

$$\text{TSP} = \{(G, k) : G \text{ is a weighted graph that has a tour of total weight at most } k\}.$$

A certificate for a given instance  $(G = (V, E), k)$  is a sequence of up to  $|V|$  vertices in the graph. The verifier simply checks that the vertices form a tour whose cost is at most  $k$ . The precise pseudocode is given in [Algorithm 144](#).

**Algorithm 144 (Efficient Verifier for TSP)**

**Input:** instance: weighted graph and weight budget; certificate: a list of up to  $|V|$  vertices in the graph

**Output:** whether the certificate represents a tour within the budget

```

function  $\text{VERIFYTSP}((G = (V, E, w), k), c = (v_0, v_1, \dots, v_m))$ 
    if  $m \neq |V|$  or  $v_0 \neq v_m$  then reject
    if  $v_i = v_j$  for some distinct  $1 \leq i, j \leq m$  then reject
     $t = 0$ 
    for  $i = 0$  to  $m - 1$  do
         $t = t + w(v_i, v_{i+1})$  ▷ sum the edge weights on the tour
    if  $t > k$  then reject
    accept

```

**Lemma 145**  $\text{VERIFYTSP}$  is an efficient verifier for TSP, so  $\text{TSP} \in \text{NP}$ .

**Proof 146** We show that  $\text{VERIFYTSP}$  satisfies [Definition 139](#) by showing that it is efficient and correct.

First,  $\text{VERIFYTSP}$  runs in time polynomial in the size  $|G, k|$  of the instance: checking for duplicate vertices  $v_i, v_j$  can be done in polynomial time, e.g., by checking all  $O(m^2) = O(|V|^2)$  pairs of distinct  $1 \leq i, j \leq m$ . Then the algorithm loops over  $|V|$  edges, summing their weights. These weights are included in the input instance, so they can be summed in polynomial time in the instance size. Finally, the algorithm compares the sum against  $k$ , which can be done in polynomial time.

We now argue correctness:

- If  $(G, k) \in \text{TSP}$ , then by definition,  $G$  has a tour of weight at most  $k$ . Let  $c$  be the sequence of  $|V| + 1$  vertices in such a tour, starting and ending at the same arbitrary vertex, which has size polynomial in  $|G|$ . By inspection, we see that  $\text{VERIFYTSP}((G, k), c)$  accepts, because all of  $V$ ’s checks are satisfied by this  $c$ .
- Conversely, if  $\text{VERIFYTSP}((G, k), c)$  accepts for some  $c = (v_0, \dots, v_m)$ , then because all of  $V$ ’s checks are satisfied, this  $c$  starts and ends at the same vertex  $v_0 = v_m$ , it visits all  $|V|$  vertices exactly once (because there are no duplicate vertices among  $v_1, \dots, v_m$ ), and the total weight of all  $m$  edges between consecutive

vertices in  $c$  is at most  $k$ . Therefore,  $c$  is a tour of  $G$  having total weight at most  $k$ , hence  $(G, k) \in \text{TSP}$ , as needed.

## 14.4 P Versus NP

We have defined two complexity classes:

- P is the class of languages that can be decided efficiently.
- NP is the class of languages that can be verified efficiently.

More precisely, a language  $L \in P$  if there exists a polynomial-time algorithm  $D$  such that:

- if  $x \in L$ , then  $D(x)$  accepts;
- if  $x \notin L$ , then  $D(x)$  rejects.

Similarly,  $L \in NP$  if there exists a polynomial-time algorithm  $V$  such that:

- if  $x \in L$ , then  $V(x, c)$  accepts for at least one (poly-sized) certificate  $c$ ;
- if  $x \notin L$ , then  $V(x, c)$  rejects for all certificates  $c$ .

How are these two classes related? First, if a language is efficiently decidable, then it is also efficiently verifiable, trivially: the verifier can just ignore the certificate, and determine on its own whether the input is in the language, using the given efficient decider. (As an exercise, formalize this argument according to the definitions.) This gives us the following result.

**Lemma 147**  $P \subseteq NP$ .

The above relationship allows for two possibilities:

- $P \subsetneq NP$ , i.e., P is a *proper* subset of (hence not equal to) NP; or
- $P = NP$ .

The latter possibility would mean that every efficiently *verifiable* problem is also efficiently *decidable*. Is this the case? What is the answer to the question

Does  $P = NP$  ?

We do not know the answer to this question! Indeed, the “P versus NP” question is perhaps the greatest open problem in Computer Science—and even one of the most important problems in all of Mathematics, as judged by the Clay Mathematics Institute<sup>44</sup>, which has offered a \$1 million prize for its resolution.

Consider the two example languages from above, MAZE and TSP. We saw that both are in NP, and indeed, they have similar definitions, and their verifiers also have much in common. However, we know that MAZE  $\in P$ : it can be decided efficiently simply by checking whether a breadth-first search from vertex  $s$  reaches vertex  $t$ . On the other hand, we do not know whether TSP is in P: we do not know of any efficient algorithm that decides TSP, and we do not know how to prove that no such algorithm exists. Most experts *believe* that there is no efficient algorithm for TSP, which would imply that  $P \neq NP$ , but the community has no idea how to *prove* this.

How could we hope to resolve the P-versus-NP question? To show that the two classes are not equal, as most experts believe, it would suffice to demonstrate that some *single* language is in NP, but is not in P. However, this seems exceedingly difficult to do: we would need to somehow prove that, of all the infinitely many efficient algorithms—including many we have not yet discovered and cannot even imagine—none of them decides the language in question.<sup>45</sup> On

<sup>44</sup> <https://www.claymath.org/millennium-problems/>

<sup>45</sup> Recall that in the computability unit, we proved the existence of *undecidable* languages via techniques like diagonalization. Clearly, an undecidable language cannot be decided efficiently, so it is not in P. However, the challenge here is that we seek a language that is not in P, but also *is* in NP. It can be shown that every language in NP is decidable, so an undecidable language will not serve our purposes. More generally, known techniques like diagonalization and several others have been shown to face fundamental obstacles for resolving P versus NP.

the other hand, showing that  $P = NP$  also seems very difficult: we would need to demonstrate that *every* one of the infinitely many languages in  $NP$  can be decided efficiently, i.e., by some polynomial-time algorithm.

Fortunately, a rich theory has been developed that will make the resolution of  $P$  versus  $NP$  (somewhat) simpler. As we will see below, it is possible to prove that some languages in  $NP$  are the “*hardest*” ones in that class, in the following sense:

an efficient algorithm for *any one* of these “*hardest*” languages would imply an efficient algorithm for *every* language in  $NP$ !

So, to prove that  $P = NP$ , it would suffice to prove that *just one* of these “*hardest*” languages is in  $P$ . And in the other direction, the most promising route to prove  $P \neq NP$  is to show that one of these “*hardest*” languages is not in  $P$ —because if *some*  $NP$  language is not in  $P$ , then the same goes for all these “*hardest*” languages in  $NP$ .

In summary, the resolution of the  $P$ -versus- $NP$  question lies entirely with the common fate of these “*hardest*” languages:

- Any *one* of them has an efficient algorithm, if and only if *all* of them do, if and only if  $P = NP$ .
- Conversely, any one of them does *not* have an efficient algorithm, if and only if *none* of them do, if and only if  $P \neq NP$ .

It turns out that there are thousands of known “*hardest*” languages in  $NP$ . In fact, as we will see, TSP is one of them! We will prove this via a series of results, starting with the historically first language that was shown to be one of the “*hardest*” in  $NP$ , next.

## SATISFIABILITY AND THE COOK-LEVIN THEOREM

Our first example of a “hardest” problem in NP is the *satisfiability* problem for *Boolean formulas*. Given as input a Boolean formula like

$$\phi = (y \vee (\neg x \wedge z) \vee \neg z) \wedge (\neg x \vee z),$$

we wish to determine whether there is a true/false *assignment* to its *variables* that makes the formula *evaluate* to true. Let us define the relevant terms:

- a (*Boolean*) *variable* like  $x$  or  $y$  or  $x_{42}$  can be *assigned* the value “true” (often represented as 1) or “false” (represented as 0);
- a *literal* is either a variable or its negation, e.g.,  $x$  or  $\neg y$ ;
- an *operator* is either conjunction (AND), represented as  $\wedge$ ; disjunction (OR), represented as  $\vee$ ; or negation (NOT), represented either as  $\neg$  or with an overline, like  $\neg(x \wedge y)$  or, equivalently,  $\overline{x \wedge y}$ .

A Boolean formula is a well-formed mathematical expression involving literals combined with operators, and following the usual rules of parentheses for grouping.

An initial observation is that using the rules of logic like De Morgan’s laws<sup>46</sup>, we can eliminate any double-negations, and we can iteratively move all negations “inward” to the literals, e.g.,  $\neg(x \wedge \neg y) = (\neg x \vee y)$ . So, from now on we assume without loss of generality that the negation operator appears only in literals.

We define the *size* of a formula to be the number of literals it has, counting duplicate appearances of the same literal.<sup>47</sup> For example, the formula  $\phi$  above has size 6. Note that the size of a formula is at least the number of distinct variables that appear in the formula.

An *assignment* is a mapping of the variables in a formula to truth values. We can represent an assignment over  $n$  variables (under some fixed order) as an  $n$ -tuple, such as  $(a_1, \dots, a_n)$ . For the above formula, the assignment  $(0, 1, 0)$  maps  $x$  to the value 0 (false),  $y$  to the value 1 (true), and  $z$  to the value 0 (false). The notation  $\phi(a_1, \dots, a_n)$  denotes the value of  $\phi$  when *evaluated* on the assignment  $(a_1, \dots, a_n)$ , i.e., with each variable substituted by its assigned value.

(More generally, we can also consider a *partial assignment*, which maps a subset of the variables to truth values. Evaluating a formula on a partial assignment is denoted like  $\phi(x = 0, y = 1)$ , which assigns  $x$  to false and  $y$  to true; this yields another formula in the remaining variables.)

A *satisfying assignment* for a formula is an assignment that makes the formula evaluate to true. In the above example,  $(0, 1, 0)$  is a satisfying assignment:

$$\begin{aligned} \phi(0, 1, 0) &= (1 \vee (\neg 0 \wedge 0) \vee \neg 0) \wedge (\neg 0 \vee 0) \\ &= (1 \vee (1 \wedge 0) \vee 1) \wedge (1 \vee 0) \\ &= 1 \wedge 1 \\ &= 1. \end{aligned}$$

<sup>46</sup> [https://en.wikipedia.org/wiki/De\\_Morgan%27s\\_laws](https://en.wikipedia.org/wiki/De_Morgan%27s_laws)

<sup>47</sup> Defining size as the number of literals is more robust and convenient than other definitions we might consider, like the total number of symbols in the formula (including operators and parentheses), while still being *proportional* to other reasonable notions. For example, “moving negations inward” does not change the number of literals, but it can change the number of symbols.

On the other hand,  $(1, 0, 1)$  is not a satisfying assignment:

$$\begin{aligned}\phi(1, 0, 1) &= (0 \vee (\neg 1 \wedge 1) \vee \neg 1) \wedge (\neg 1 \vee 1) \\ &= (0 \vee (0 \wedge 1) \vee 0) \wedge (0 \vee 1) \\ &= 0 \wedge 1 \\ &= 0.\end{aligned}$$

A formula  $\phi$  is *satisfiable* if it has at least one satisfying assignment. The decision problem of determining whether a given formula is satisfiable corresponds to the following language.

**Definition 148 (Satisfiability Language)** The (Boolean) satisfiability language is defined as

$$\text{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\}.$$

A first observation is that SAT is decidable. A formula  $\phi$  of size  $n$  has at most  $n$  variables, so there are at most  $2^n$  possible assignments. Therefore, we can decide SAT using a brute-force algorithm that simply iterates over all of the assignments of its input formula, accepting if at least one of them satisfies the formula, and rejecting otherwise. Although this is *not efficient*, it is apparent that it does decide SAT.

We also observe that SAT has an efficient *verifier*, so it is in NP.

**Lemma 149**  $SAT \in \text{NP}$ .

**Proof 150** A certificate, or “claimed proof”, for a formula  $\phi$  is just an assignment for its variables. The following efficient verifier simply evaluates the given formula on the given assignment and accepts if the value is true, otherwise it rejects.

**Input:** instance: a Boolean formula; certificate: an assignment for its variables

**Output:** whether the assignment satisfies the formula

```
function  $V_{\text{SAT}}(\phi, \alpha)$ 
    if  $\phi(\alpha) = 1$  then accept
    reject
```

For efficiency, first observe that the size of the certificate is polynomial (indeed, linear) in the size of the instance  $\phi$ , because it consists of a true/false value for each variable that appears in the formula. This verifier runs in linear time in the size of its input formula  $\phi$ , because evaluating each AND/OR operator in the formula reduces the number of terms in the expression by one.

For correctness, by the definitions of SAT and  $V_{\text{SAT}}$ ,

$$\begin{aligned}\phi \in \text{SAT} &\iff \text{exists } \alpha \text{ s.t. } \phi(\alpha) = 1 \\ &\iff \text{exists } \alpha \text{ s.t. } V_{\text{SAT}}(\phi, \alpha) \text{ accepts ,}\end{aligned}$$

so by Definition 139,  $V_{\text{SAT}}$  is indeed an efficient verifier for SAT, as claimed.  $\square$

Is SAT *efficiently* decidable?—i.e., is  $SAT \in \text{P}$ ? The decider we described above is not efficient: it takes time *exponential* in the number of variables in the formula, and the number of variables may be as large as the *size* of the formula (i.e., the number of literals in it), so in the worst case the algorithm runs in exponential time in its input size.

However, the above does not prove that  $SAT \notin \text{P}$ ; it just shows that *one specific* (and naïve) algorithm for deciding SAT is inefficient. Conceivably, there could be a more sophisticated *efficient* algorithm that cleverly analyzes an arbitrary input formula in some way to determine whether it is satisfiable. Indeed, there are regular conferences<sup>48</sup> and competitions<sup>49</sup> to which researchers submit their best ideas, algorithms, and software for solving SAT. Although many

<sup>48</sup> <http://satisfiability.org/>

<sup>49</sup> <http://www.satcompetition.org/>

algorithms have been developed that perform very impressively on large SAT instances of interest, none of them is believed to run in polynomial time and to be correct on all instances.

To date, we actually *do not know* whether SAT is efficiently decidable—we do not know an efficient algorithm for it, and we do not know how to prove that no such algorithm exists. Yet although the question of whether  $SAT \in P$  is unresolved, the *Cook-Levin Theorem* says that SAT is a “hardest” problem in NP.<sup>50</sup>

**Theorem 151 (Cook-Levin)** *If  $SAT \in P$ , then every NP language is in P, i.e.,  $P = NP$ .*

The full proof of the Cook-Levin Theorem is ingenious and rather intricate, but its high-level idea is fairly easy to describe. Let  $L$  be an arbitrary language in NP; this means there is an efficient *verifier*  $V$  for  $L$ . Using the hypothesis that  $SAT \in P$ , we will construct an efficient *decider* for  $L$ , which implies that  $L \in P$ , as claimed.

The key idea behind the efficient decider for  $L$  is that, using just the fact that  $V$  is an efficient verifier for  $L$ , we can efficiently *transform* any instance of  $L$  into an instance of SAT that has the same “yes/no answer”: either both instances are in their respective languages, or neither is. More precisely, there is an efficient procedure that maps any instance  $x$  of  $L$  to a corresponding Boolean formula  $\phi_{V,x}$ , such that<sup>51</sup>

$$x \in L \iff \phi_{V,x} \in SAT .$$

By the hypothesis that  $SAT \in P$ , there is an efficient decider  $D_{SAT}$  for SAT, so from all this we get the following efficient decider for  $L$ :

```
function  $D_L(x)$ 
construct  $\phi_{V,x}$  as described below
return  $D_{SAT}(\phi_{V,x})$ 
```

Since  $\phi_{V,x}$  can be constructed in time polynomial in the size of  $x$ , and  $D_{SAT}$  runs in time polynomial in the size of  $\phi_{V,x}$ , by composition,  $D_L$  as a whole runs in time polynomial in the size of  $x$ . And the fact that  $D_L$  is correct (i.e., it decides  $L$ ) follows directly from the correctness of  $D_{SAT}$  and the above-stated relationship between  $x$  and  $\phi_{V,x}$ .

This completes the high-level description of the proof strategy. In the following section we show how to efficiently construct  $\phi_{V,x}$  from  $x$ , so that they satisfy the above-stated relationship.

---

<sup>50</sup> The Cook-Levin Theorem is named after its discoverers, Stephen Cook and Leonid Levin, who found it in the early 1970s. Interestingly, they discovered this theorem *independently* of each other, with Cook working in the USA (and later Canada), and Levin working in Russia. Due to the “Cold War” of the time, the two countries’ research communities did not have much interaction.

<sup>51</sup> In fact, the above equivalence follows from a stronger property: by the careful design of  $\phi_{V,x}$ , there is a two-way correspondence between the valid certificate(s)  $c$  for  $x$  (if any), and the satisfying assignment(s) for  $\phi_{V,x}$  (if any). That is, any certificate  $c$  that makes  $V(x, c)$  accept directly yields a corresponding satisfying assignment for  $\phi_{V,x}$ , and any satisfying assignment for  $\phi_{V,x}$  yields a corresponding certificate  $c$  that makes  $V(x, c)$  accept.

## PROOF OF THE COOK-LEVIN THEOREM

### 16.1 Configurations and Tableaus

In order to describe the construction of  $\phi_{V,x}$ , we first need to recall the notion of a *configuration* of a Turing machine and its representation as a sequence, which we previously discussed in *Wang Tiling* (page 113). A configuration encodes a “snapshot” of a Turing machine’s execution: the contents of the machine’s tape, the active state  $q \in Q$ , and the position of the tape head. We represent these as an infinite sequence over the alphabet  $\Gamma \cup Q$  (the union of the finite tape alphabet and the TM’s finite set of states), which is simply:

- the contents of the tape, in order from the leftmost cell,
- with the active state  $q \in Q$  inserted directly to the left of the symbol corresponding to the cell on which the head is positioned.

For example, if the input is  $x_1 x_2 \cdots x_n$  and the initial state is  $q_0$ , then the following sequence represents the machine’s starting configuration:

$$q_0 x_1 x_2 \cdots x_n \perp \perp \dots$$

Since the head is at the leftmost cell and the state is  $q_0$ , the string has  $q_0$  inserted to the left of the leftmost tape symbol. If the transition function gives  $\delta(q_0, x_1) = (q', x', R)$ , then the next configuration is represented by

$$x' q' x_2 \cdots x_n \perp \perp \dots$$

The first cell’s symbol has been changed to  $x'$ , the machine is in state  $q'$ , and the head is at the second cell, represented here by writing  $q'$  to the left of that cell’s symbol.

As stated above in the proof overview, for any language  $L \in \text{NP}$  there exists an efficient verifier  $V$  that takes as input an instance  $x$ , a certificate  $c$  whose size is some polynomial in  $|x|$ , and which runs in time polynomial in the input size, and hence in  $|x|$  as well. So altogether, there is some constant  $k$  such that  $V(x, c)$  runs for at most  $|x|^k$  steps before halting, for any  $x, c$ .<sup>52</sup> Since the head starts at the leftmost cell and can move only one position in each step, this implies that  $V(x, c)$  can read or write only the first  $|x|^k$  cells of the tape. Thus, instead of using an infinite sequence, we can represent any configuration during the execution of  $V(x, c)$  using just a *finite* string of length about  $|x|^k$ , ignoring the rest since it cannot affect the execution.

For an instance  $x$  of size  $n = |x|$ , we can represent the sequence of *all* the configurations of the machine, over its (at most)  $n^k$  computational steps, as an  $n^k$ -by- $n^k$  *tableau*, with one configuration per row; for convenience later on, we also place a special  $\#$  symbol at the start and end of each row.<sup>53</sup> So, each cell of the tableau contains a symbol from

<sup>52</sup> To be completely accurate, this holds for all *sufficiently large*  $|x|$  (and all  $c$ ). There are only finitely many  $x$  that are not “sufficiently large,” so the yes/no answer of whether  $x \in L$  for each of these  $x$  can be “hard-coded” into the decider for  $L$ . Hence we restrict our attention to deciding  $L$  for all sufficiently large  $x$ .

<sup>53</sup> To be completely accurate, we need  $n^k + 1$  rows and  $n^k + 3$  columns to represent the tableau:  $t$  computational steps require  $t + 1$  rows, and we need  $n^k$  columns for the tape cells, plus one for the active state and two for the  $\#$  symbols. These constant terms are not important to the proof, so for notational simplicity we ignore them. In addition, if the machine halts in fewer than  $n^k$  steps, we “pad” the tableau to have exactly  $n^k + 1$  rows by appending copies of the final row (corresponding to the halting configuration).

the set  $S = \Gamma \cup Q \cup \{\#\}$ , where  $\Gamma$  is the finite tape alphabet of  $V$ ;  $Q$  is the finite set of states in  $V$ ; and  $\# \notin \Gamma \cup Q$  is a special extra symbol.

$n^k$										
$n^k$	#	$q_{st}$	$w_1$	$w_2$	...	$w_n$	$\perp$	...	$\perp$	#
	#								#	
	#								#	
	#								#	

Initial configuration  
After 1 step  
 $V$  halts after at most  $n^k$  steps

Observe that, since  $V$  is deterministic, the contents of the *first* row of the tableau *completely determine* the contents of *all* the rows, via the “code” of  $V$ . Moreover, adjacent rows represent a single computational step of the machine, and hence are identical to each other, except in the vicinity of the symbols representing the active states before and after the transition. Finally,  $V(x, c)$  accepts if and only if the accept-state symbol  $q_{\text{acc}} \in Q$  appears somewhere in its tableau. These are important feature of tableaus that are exploited in the construction of the formula  $\phi_{V,x}$ , which we describe next.

## 16.2 Constructing the Formula

With the notion of a computational tableau in hand, we now describe the structure of the formula  $\phi_{V,x}$  that is constructed from the instance  $x$  (also using the fixed code of  $V$ ).

- The *variables* of the formula represent the *contents* of all the cells in a potential tableau for  $V$ . That is, assigning Boolean values to all the variables fully specifies the contents of a *claimed* tableau, including the value of a certificate  $c$  in the first row.

Conceptually, we can think of an assignment to the variables, and the potential tableau that they represent, as a *claimed execution transcript* for  $V$ .

- The formula  $\phi_{V,x}$  is carefully designed to evaluate whether the claimed tableau (as specified by the variables’ values) meets two conditions:
  1. it is the *actual execution tableau* of  $V(x, c)$ , for the *specific given* instance  $x$ , and whatever certificate  $c$  appears in the first row, and
  2. it is an *accepting* tableau, i.e.,  $V(x, c)$  accepts.

Conceptually, we can think of  $\phi_{V,x}$  as *checking* whether the claimed execution transcript for  $V$  is genuine, and results in  $V$  accepting the specific instance  $x$ .

In summary, the formula  $\phi_{V,x}$  evaluates to true *if and only if* its variables are set so that they specify the full and

accepting execution tableau of  $V(x, c)$ , for the certificate  $c$  specified by the variables. Therefore, as needed,

$$\begin{aligned}\phi_{V,x} \in \text{SAT} &\iff \phi_{V,x} \text{ is satisfiable} \\ &\iff \text{there exists } c \text{ s.t. } V(x, c) \text{ accepts} \\ &\iff x \in L,\end{aligned}$$

where the last equivalence holds by [Definition 139](#).

We now give the details. The variables of the formula are as follows:

For each cell position  $i, j$  of the tableau, and each symbol  $s \in S$ , there is a Boolean variable  $t_{i,j,s}$ .

So, there are  $|S| \cdot n^{2k} = O(n^{2k})$  variables in total, which is polynomial in  $n = |x|$  because  $|S|$  and  $2k$  are constants. (Recall that  $S$  is a fixed finite alphabet that does not vary with  $n$ .)

Assigning Boolean values to these variables specifies the contents of a *claimed* tableau, as follows:

Assigning  $t_{i,j,s} = \text{true}$  specifies that cell  $i, j$  of the claimed tableau has symbol  $s$  in it.

Observe that the variables can be assigned *arbitrary* Boolean values, so for the same cell location  $i, j$ , we could potentially assign  $t_{i,j,s} = \text{true}$  for *multiple* different values of  $s$ , or none at all. This would make the contents of cell  $i, j$  undefined. The formula  $\phi_{V,x}$  is designed to “check” for this and evaluate to false in such a case, and also to check for all the other needed properties on the claimed tableau, as we now describe.

The formula  $\phi_{V,x}$  is defined as the conjunction (AND) of four subformulas:

$$\phi_{V,x} = \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{start},x} \wedge \phi_{\text{move},V}.$$

Each subformula “checks” (or “enforces”) a certain condition on the variables and the claimed tableau they specify, evaluating to true if the condition holds, and false if it does not. The subformulas check the following conditions:

- $\phi_{\text{cell}}$  checks that each cell of the claimed tableau is well defined, i.e., it contains exactly one symbol;
- $\phi_{\text{accept}}$  checks that the claimed tableau is an accepting tableau, i.e., that  $q_{\text{acc}}$  appears somewhere in it;
- $\phi_{\text{start},x}$  checks that the first row of the claimed tableau is valid, i.e., it represents the initial configuration of the verifier running on the *given instance*  $x$  and some certificate  $c$  (as specified by the variables);
- $\phi_{\text{move},V}$  checks that each non-starting row of the claimed tableau follows from the previous one, according to the transition function of  $V$ .

Observe that, as needed above, all these conditions hold *if and only if* the claimed tableau is indeed the *actual accepting* execution tableau of  $V(x, c)$ , for the certificate  $c$  that appears in the first row. So, it just remains to show how to design the subformulas to correctly check their respective conditions, which we do next.

### 16.2.1 Cell Consistency

To check that a given cell  $i, j$  has a well-defined symbol, we need *exactly one* of the variables  $t_{i,j,s}$  to be true, over all  $s \in S$ . The formula

$$\bigvee_{s \in S} t_{i,j,s}$$

checks that *at least one* of the variables is true, and the formula

$$\neg \bigvee_{\text{distinct } s, s' \in S} (t_{i,j,s} \wedge t_{i,j,s'}) = \bigwedge_{\text{distinct } s, s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'})$$

checks that *no more than one* of the variables is true (where the equality holds by [De Morgan’s laws](#)<sup>54</sup>).

<sup>54</sup> [https://en.wikipedia.org/wiki/De\\_Morgan%27s\\_laws](https://en.wikipedia.org/wiki/De_Morgan%27s_laws)

Putting these together over all cells  $i, j$ , the subformula

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i, j \leq n^k} \left[ \bigvee_{s \in S} t_{i,j,s} \wedge \bigwedge_{\text{distinct } s, s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'}) \right].$$

checks that for all  $i, j$ , exactly one of  $t_{i,j,s}$  is true, as needed.

The formula has  $O(|S|^2) = O(1)$  literals for each cell (because  $S$  is a fixed alphabet that does not vary with  $n = |x|$ ), so  $\phi_{\text{cell}}$  has size  $O(n^{2k})$ , which is polynomial in  $n = |x|$  because  $2k$  is a constant.

### 16.2.2 Accepting Tableau

To check that the claimed tableau is an accepting one, we just need to check that at least one cell has  $q_{\text{acc}}$  as its symbol, which is done by the following subformula:

$$\phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} t_{i,j,q_{\text{acc}}}.$$

This has one literal per cell, for a total size of  $O(n^{2k})$ , which again is polynomial in  $n = |x|$ .

### 16.2.3 Starting Configuration

For the top row of the tableau, which represents the starting configuration, we suppose that the encoding of an input pair  $(x, c)$  separates  $x$  from  $c$  on the tape using a special symbol  $\$ \in \Gamma \setminus \Sigma$  that is in the tape alphabet  $\Gamma$  but not in the input alphabet  $\Sigma$ . (Recall that  $x$  and  $c$  are strings over  $\Sigma$ , so  $\$$  unambiguously separates them.) Letting  $m = |c|$ , the top row of the tableau is therefore as follows:

#	$q_{\text{st}}$	$x_1$	$\cdots$	$x_n$	\$	$c_1$	$\cdots$	$c_m$	$\perp$	$\cdots$	#
---	-----------------	-------	----------	-------	----	-------	----------	-------	---------	----------	---

The row starts with the # symbol. Since the machine is in active state  $q_{\text{start}}$ , and the head is at the leftmost cell of the tape,  $q_{\text{start}}$  is the second symbol. The contents of the tape follow, which are the symbols of the input  $x$ , then the \$ separator, then the symbols of the certificate  $c$ , then blanks until we have  $n^k$  symbols, except for the last symbol, which is #.

We now describe the subformula  $\phi_{\text{start},x}$  that checks that the first row of the claimed tableau has the above form. We require the row to have the *specific, given* instance  $x$  in its proper position of the starting configuration. This is checked by the formula

$$\phi_{\text{input}} = t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \cdots \wedge t_{1,n+2,x_n}.$$

For the tableau cells corresponding to the certificate, **our construction does not know what certificate(s)  $c$ , if any, would cause  $V(x, c)$  to accept**, or even what their sizes are (other than that they are less than  $n^k$ ). Indeed, a main idea of this construction is that **any satisfying assignment for  $\phi_{V,x}$ , if one exists, specifies a valid certificate  $c$  for  $x$  (and vice versa)**.

So, our formula allows *any* symbol  $s \in \Sigma \cup \{\perp\}$  from the input alphabet, as well as blank, to appear in the positions after the separator \$. We just need to ensure that any blanks appear only *after* the certificate string, i.e., not before any symbol from  $\Sigma$ . Overall, the formula that checks that the portion of the first row dedicated to the certificate is well-formed is

$$\phi_{\text{cert}} = \bigwedge_{j=n+4}^{n^k-1} \left( t_{1,j,\perp} \vee \left( \bigvee_{s \in \Sigma} t_{1,j,s} \wedge \neg t_{1,j-1,\perp} \right) \right).$$

In words, this says that each cell in the zone dedicated to the certificate either has a blank, or has an input-alphabet symbol *and* the preceding symbol is not blank.

Putting these pieces together with the few other fixed cell values, the subformula that checks the first row of the claimed tableau is:

$$\phi_{\text{start},x} = t_{1,1,\#} \wedge t_{1,2,q_{\text{start}}} \wedge \phi_{\text{input}} \wedge t_{1,n+3,\$} \wedge \phi_{\text{cert}} \wedge t_{1,n^k,\#} .$$

This subformula has one literal for each symbol in the instance  $x$ , the start state, and the special  $\#$  and  $\$$  symbols. It has  $O(|\Sigma|) = O(1)$  literals for each cell in the zone dedicated to the certificate (recall that the tape alphabet  $\Gamma$  is fixed and does not vary with  $n = |x|$ ). Thus, this subformula has size  $O(n^k)$ , which again is polynomial in  $n = |x|$ .

#### 16.2.4 Transitions

Finally, we describe the subformula that checks whether every non-starting row in the claimed tableau follows from the previous one. This is the most intricate part of the proof, and the only one that relies on the transition function, or “code”, of  $V$ .

As a warmup first attempt, recall that any syntactically legal configuration string  $r$  (regardless of whether  $V$  can actually reach that configuration) determines the next configuration string  $r'$ , according to the code of  $V$ . So, letting  $i, i + 1$  denote a pair of adjacent row indices, and  $r$  denote a legal configuration string, we could write:

1. the formula that checks whether row  $i$  equals  $r$  *and* row  $i + 1$  equals the corresponding  $r'$ , namely,

$$\phi_{i,r} = \bigwedge_{j=1}^{n^k} (t_{i,j,r_j} \wedge t_{i+1,j,r'_j}) ;$$

2. the formula that checks whether row  $i + 1$  follows from row  $i$ , namely,

$$\phi_i = \bigvee_{\text{legal } r} \phi_{i,r} ;$$

3. the formula that checks whether *every* non-starting row follows from the previous one, namely,

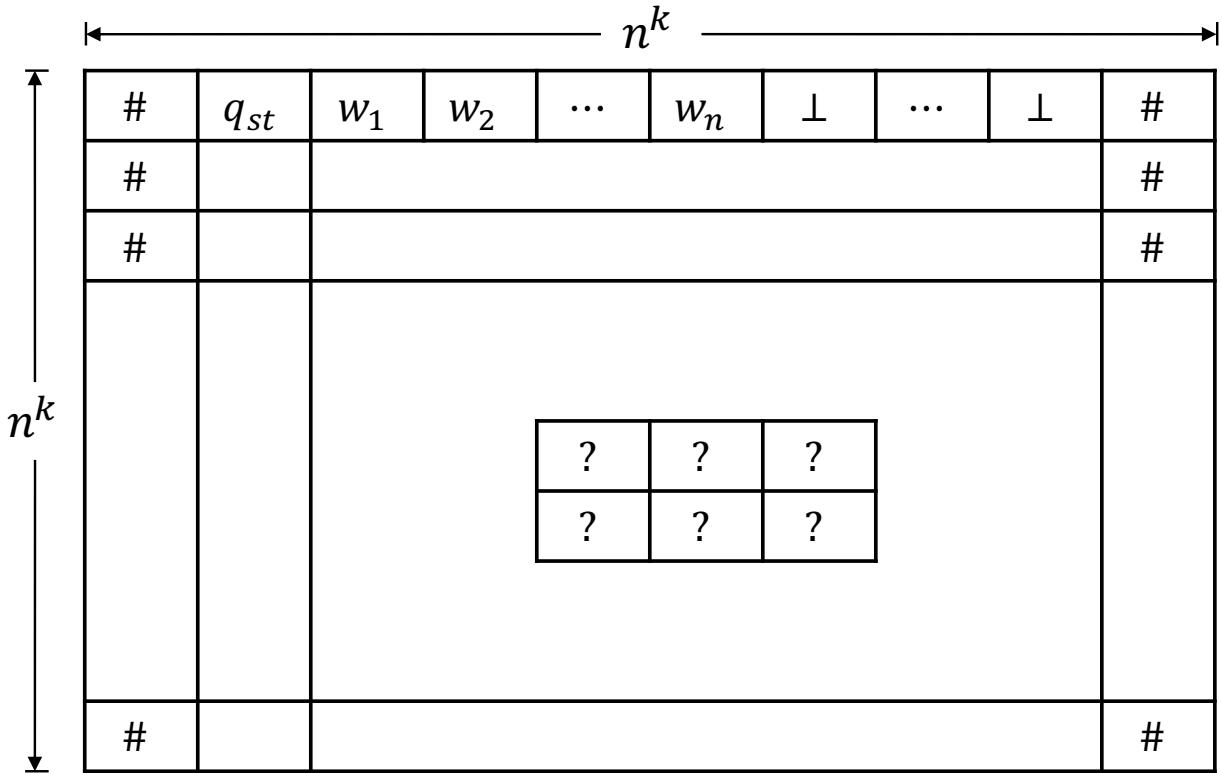
$$\phi_{\text{move},V} = \bigwedge_{i=1}^{n^k - 1} \phi_i .$$

Because  $\phi_{\text{start},x}$  checks that the first row of the claimed tableau is a valid starting configuration,  $\phi_{\text{start},x} \wedge \phi_{\text{move},V}$  does indeed check that the entire claimed tableau is valid.

This approach gives a *correct* formula, but unfortunately, it is not *efficient*—the formula does not have size polynomial in  $n = |x|$ . The only problem is in the second step: because there are multiple valid choices for each symbol of a legal  $r$ , and  $r$  has length  $n^k$ , there are *exponentially many* such  $r$ . So, taking the OR over all of them results in a formula of exponential size.

To overcome this efficiency problem, we proceed similarly as above, but using a finer-grained breakdown of adjacent rows than in  $\phi_{i,r}$  above. The main idea is to consider the 2-by-3 “windows” (subrectangles) of adjacent rows, and check that every window is “valid” according to the code of  $V$ . It can be proved that if all the overlapping 2-by-3 windows of a claimed pair of rows are valid, then the pair *as a whole* is valid. This is essentially because valid adjacent rows are almost identical (except around the cells with the active states), and the windows overlap sufficiently to guarantee that any invalidity in the claimed rows will be exposed in some 2-by-3 window.<sup>55</sup>

<sup>55</sup> Interestingly, the same does not hold for 2-by-2 windows: it is possible to construct a pair of rows in which every 2-by-2 window is valid, but the pair of rows as a whole does *not* represent a correct transition according to the code of  $V$ . So, we must use windows of width at least 3, and 3 suffices.



There are  $|S|^6 = O(1)$  distinct possibilities for a 2-by-3 window, but not all of them are valid. Below we describe all the valid windows, and based on this we construct the subformula  $\phi_{\text{move},V}$  as follows.

For any particular valid window  $w$ , let  $s_1, \dots, s_6$  be its six symbols, going left-to-right across its first then second row. Similarly to  $\phi_{i,r}$  above, the following simple formula checks whether the window of the claimed tableau with top-left corner at  $i, j$  matches  $w$ :

$$\phi_{i,j,w} = (t_{i,j,s_1} \wedge t_{i,j+1,s_2} \wedge t_{i,j+2,s_3} \wedge t_{i+1,j,s_4} \wedge t_{i+1,j+1,s_5} \wedge t_{i+1,j+2,s_6}) .$$

Now, letting  $W$  be the set of all valid windows, similarly to  $\phi_i$  above, the following formula checks whether the window at  $i, j$  of the claimed tableau is valid:

$$\phi_{i,j} = \bigvee_{w \in W} \phi_{i,j,w} .$$

Finally, we get the subformula that checks whether *every* window in the tableau is valid:

$$\phi_{\text{move},V} = \bigwedge_{\substack{1 \leq i \leq n^k - 1 \\ 1 \leq j \leq n^k - 2}} \phi_{i,j} .$$

The set  $W$  of valid windows has size  $|W| \leq |S|^6 = O(1)$ , so each  $\phi_{i,j}$  has  $O(1)$  literals. Because there are  $O(n^{2k})$  windows to check, the subformula  $\phi_{\text{move},V}$  has size  $O(n^{2k})$ , which again is polynomial in  $n = |x|$ .

We now describe all the valid windows. The precise details of this are not so essential, because all we used above were the facts that the valid windows are well defined, and that there are  $O(1)$  of them. We use the following notation in the descriptions and diagrams:

- $\gamma$  for an arbitrary element of  $\Gamma$ ,
- $q$  for an arbitrary element of  $Q$ ,

- $\rho$  for an arbitrary element of  $\Gamma \cup Q$ ,
- $\sigma$  for an arbitrary element of  $\Gamma \cup \{\#\}$ .

First, a window having the  $\#$  symbol somewhere in it is valid only if  $\#$  is in the first position of both rows, or is in the third position of both rows. So, every valid window has one of the following forms, where the symbols  $\rho_1, \dots, \rho_6$  must additionally be valid according to the rules below.

#	$\rho_1$	$\rho_2$
#	$\rho_3$	$\rho_4$

$\rho_1$	$\rho_2$	$\rho_3$
$\rho_4$	$\rho_5$	$\rho_6$

$\rho_1$	$\rho_2$	#
$\rho_3$	$\rho_4$	#

If a window is not in the vicinity of a state symbol  $q \in Q$ , then the machine's transition does not affect the portion of the configuration corresponding to the window, so the top and bottom rows match:

$\sigma_1$	$\gamma$	$\sigma_2$
$\sigma_1$	$\gamma$	$\sigma_2$

To reason about what happens in a window that is in the vicinity of a state symbol, we consider how the configuration changes as a result of a right-moving transition  $\delta(q, \gamma) = (q', \gamma', R)$ :

$\gamma_2$	$\gamma_3$	$\gamma_4$	$\gamma_5$	$q$	$\gamma$	$\gamma_6$	$\gamma_7$	$\gamma_8$
$\gamma_2$	$\gamma_3$	$\gamma_4$	$\gamma_5$	$q'$	$q'$	$\gamma_6$	$\gamma_7$	$\gamma_8$
1		2		3		4		

The head moves to the right, so  $q'$  appears in the bottom row at the column to the right of where  $q$  appears in the top row. The symbol  $\gamma$  is replaced by  $\gamma'$ , though its position moves to the left to compensate for the rightward movement of the state symbol. There are four windows affected by the transition, labeled above by their leftmost columns. We thus have the following four kinds of valid windows corresponding to a rightward transition  $\delta(q, \gamma) = (q', \gamma', R)$ :

$\sigma$	$\gamma_2$	$q$
$\sigma$	$\gamma_2$	$\gamma'$

$\sigma$	$q$	$\gamma$
$\sigma$	$\gamma'$	$q'$

$q$	$\gamma$	$\sigma$
$\gamma'$	$q'$	$\sigma$

$\gamma$	$\gamma_2$	$\sigma$
$q'$	$\gamma_2$	$\sigma$

Note that we have used  $\sigma$  for the edges of some windows, as there can be either a tape symbol or the tableau-edge marker  $\#$  in these positions.

We now consider how the configuration changes as a result of a left-moving transition  $\delta(q, \gamma) = (q', \gamma', L)$ :

$\gamma_2$	$\gamma_3$	$\gamma_4$	$\gamma_5$	$q$	$\gamma$	$\gamma_6$	$\gamma_7$	$\gamma_8$
$\gamma_2$	$\gamma_3$	$\gamma_4$	$q'$	$\gamma_5$	$\gamma'$	$\gamma_6$	$\gamma_7$	$\gamma_8$
1	2	3	4	5				

The head moves to the left, so  $q'$  appears in the bottom row at the column to the left of where  $q$  appears in the top row. As with a rightward transition, we replace the old symbol  $\gamma$  with the new symbol  $\gamma'$ . This time, however, the symbol to the left of the original position of the head moves right to compensate for the leftward movement of the head. So here we have the following five kinds of valid windows corresponding to the leftward transition  $\delta(q, \gamma) = (q', \gamma', R)$ :

$\sigma$	$\gamma_3$	$\gamma_2$
$\sigma$	$\gamma_3$	$q'$
$\sigma$	$\gamma_2$	$q$

$\sigma$	$\gamma_2$	$q$
$\sigma$	$q'$	$\gamma_2$
$\gamma_2$	$q'$	$\gamma'$

$q$	$\gamma$	$\sigma$
$\gamma_2$	$\gamma'$	$\sigma$

$\gamma$	$\gamma_2$	$\sigma$
$\gamma'$	$\gamma_2$	$\sigma$

We also need to account for the case where the head is at the leftmost cell on the tape, in which case the head does not move:

#	$q$	$\gamma$	$\gamma_2$	$\gamma_3$	$\gamma_4$
#	$q'$	$\gamma'$	$\gamma_2$	$\gamma_3$	$\gamma_4$
1	2	3			

Here we have three kinds of valid windows, but the last one actually has the same structure as the last window in the normal case for a leftward transition. Thus, we have only two more kinds of valid windows:

#	$q$	$\gamma$
#	$q'$	$\gamma'$

$q$	$\gamma$	$\gamma_2$
$q'$	$\gamma'$	$\gamma_2$

Finally, we need one last set of valid windows to account for the machine reaching the accept state before the last row of the tableau. As stated above, the valid tableau is “padded” with copies of the final configuration, so the valid windows for this case look like:

$q_{acc}$	$\gamma$	$\sigma$
$q_{acc}$	$\gamma$	$\sigma$

$\sigma_1$	$q_{acc}$	$\sigma_2$
$\sigma_1$	$q_{acc}$	$\sigma_2$

$\sigma$	$\gamma$	$q_{acc}$
$\sigma$	$\gamma$	$q_{acc}$

## 16.3 Conclusion

We summarize the main claims about what we have just done.

Using an efficient verifier  $V$  for an arbitrary language  $L \in \text{NP}$ , we have demonstrated how to efficiently construct a corresponding formula  $\phi_{V,x}$  from a given instance  $x$  of  $L$ . The formula can be constructed in time polynomial in  $|x|$ , and in particular the total size of the formula is  $O(|x|^{2k})$  literals.

Then, we (almost entirely) proved that  $x \in L \iff \phi_{V,x} \in \text{SAT}$ . In the  $\implies$  direction: by correctness of  $V$ , any  $x \in L$  has at least one certificate  $c$  that makes  $V(x, c)$  accept, so the computation tableau for  $V(x, c)$  is accepting. In turn, by design of the formula  $\phi_{V,x}$ , this tableau defines a satisfying assignment for the formula (i.e., an assignment to its variables that makes  $\phi_{V,x}$  evaluate to true). So,  $x \in L$  implies that  $\phi_{V,x} \in \text{SAT}$ .

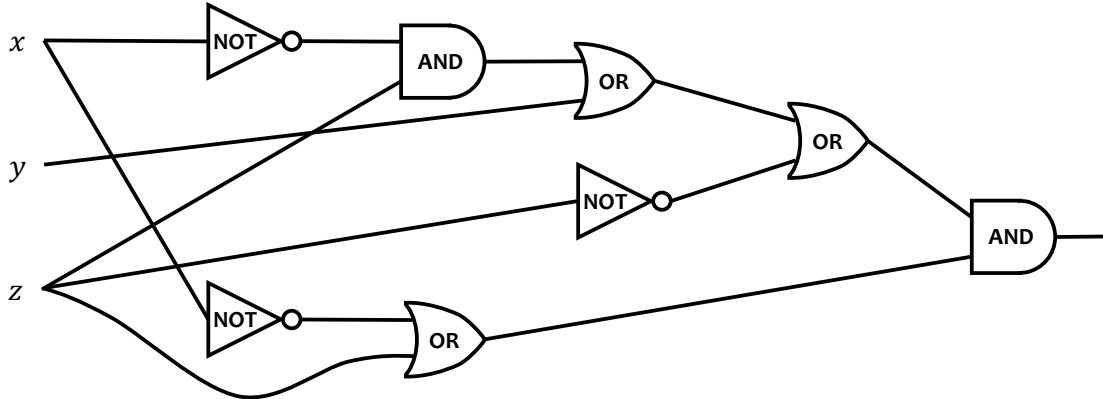
In the  $\impliedby$  direction: if the constructed formula  $\phi_{V,x} \in \text{SAT}$ , then by definition it has a satisfying assignment. Again by the special design of the formula, any satisfying assignment defines the contents of an actual accepting computation tableau for  $V(x, c)$ , for the  $c$  appearing in the top row of the tableau. Since  $V(x, c)$  accepts for this  $c$ , we conclude that  $x \in L$  by the correctness of the verifier. So,  $\phi_{V,x} \in \text{SAT}$  implies that  $x \in L$ .

Finally, if  $\text{SAT} \in \text{P}$ , then an efficient decider  $D_{\text{SAT}}$  for SAT exists, and we can use it to decide  $L$  efficiently: given an instance  $x$ , simply construct  $\phi_{V,x}$  and output  $D_{\text{SAT}}(\phi_{V,x})$ ; by the above property, this correctly determines whether  $x \in L$ . As a result,  $\text{SAT} \in \text{P}$  implies that  $\text{NP} \subseteq \text{P}$  and hence  $\text{P} = \text{NP}$ , as claimed.

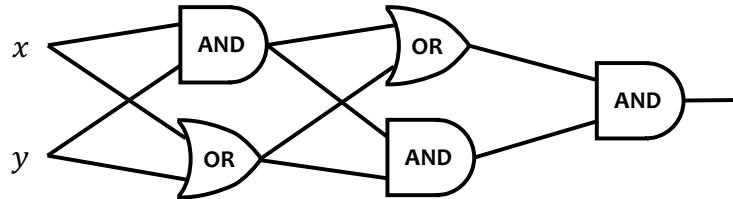
### Circuit Satisfiability

A problem that is equivalent to formula satisfiability is that of *circuit satisfiability*. Given a circuit  $C$  composed of  $m$  logic gates over  $n$  binary inputs and with a single binary output, is there a set of input values such that the output is 1? In other words, is there an assignment to the inputs that satisfies the circuit?

Without loss of generality, assume that each logic gate is either a unary NOT gate, a binary AND gate, or a binary OR gate. (This is a *universal gate set*, i.e., any Boolean function can be expressed in terms of these gates.) Since these gates correspond to negation, conjunction, and disjunction, we can mechanically convert a Boolean formula into an equivalent circuit whose size is linear in the size of the formula. The following is a circuit corresponding to the formula  $(y \vee (\neg x \wedge z) \vee \neg z) \wedge (\neg x \vee z)$ :



What about the other direction, of converting a circuit into a formula? In general, the gates in a circuit may have multiple *fan-out*, meaning that the output of a gate may be used as the inputs to multiple other gates. The following is an example:

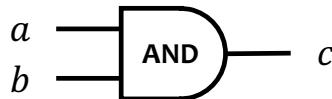


A naïve translation can result in an exponential increase in size: wherever there is multiple fan-out, the subformula corresponding to the intermediate result would be repeated. For the circuit above, such a translation would be

$$((x \wedge y) \vee (x \vee y)) \wedge ((x \wedge y) \wedge (x \vee y)) .$$

There are two copies each of the subformulas  $x \wedge y$  and  $x \vee y$ . For larger circuits, a subformula may be repeated many times.

Rather than naïvely repeating, for the output of each gate we can introduce a new variable. For instance, the following is an AND gate with existing variables  $a$  and  $b$  as inputs, and a new variable  $c$  as output:



The relationship between the inputs  $a$  and  $b$  and the output  $c$  is  $c = a \wedge b$ . A formula cannot directly express equality, but observe that variables  $x$  and  $y$  are equal exactly when either both of them are true, or both are false. Thus,  $x = y$  is logically equivalent to  $(x \wedge y) \vee (\neg x \wedge \neg y)$ . For  $c = a \wedge b$ , this idea translates to

$$\begin{aligned} c = a \wedge b &\equiv (c \wedge (a \wedge b)) \vee (\neg c \wedge \neg(a \wedge b)) \\ &\equiv (c \wedge (a \wedge b)) \vee (\neg c \wedge (\neg a \vee \neg b)) . \end{aligned}$$

In the second step, we applied De Morgan's laws<sup>56</sup> to move negation inwards, so that only variables (and not subformulas) are negated.

We can similarly express an OR gate with inputs  $a$  and  $b$  and output  $c$ :

$$\begin{aligned} c = a \vee b &\equiv (c \wedge (a \vee b)) \vee (\neg c \wedge \neg(a \vee b)) \\ &\equiv (c \wedge (a \vee b)) \vee (\neg c \wedge (\neg a \wedge \neg b)) . \end{aligned}$$

For a NOT gate with input  $a$ , we can simply express the output as  $\neg a$  without introducing a new variable.

Now that we have subformulas for each gate, the formula for the full circuit is just the conjunction of the subformulas for each gate. Since each gate introduces at most one variable and a subformula of at most six literals, the size of the resulting formula is linear in the size of the circuit.

We have demonstrated that the formula-satisfiability problem SAT is equivalent to the circuit-satisfiability problem CSAT. Since the former is a “hardest” problem in NP, so is the latter. Next, we generalize and formalize the notion of efficient transformations between problems by introducing the notion of *polynomial-time mapping reductions*.

<sup>56</sup> [https://en.wikipedia.org/wiki/De\\_Morgan%27s\\_laws](https://en.wikipedia.org/wiki/De_Morgan%27s_laws)

---

CHAPTER  
SEVENTEEN

---

## NP-COMPLETENESS

With the example of SAT and the Cook-Levin theorem in hand, we now formally define what it means to be a “hardest” problem in NP, and demonstrate several examples of such problems.

### 17.1 Polynomial-Time Mapping Reductions

The heart of the Cook-Levin theorem is an efficient algorithm for converting an instance of an (arbitrary) language  $L \in \text{NP}$  to an instance  $\phi_{V,x}$  of SAT, such that

$$x \in L \iff \phi_{V,x} \in \text{SAT} .$$

Thus, any (hypothetical) efficient decider for SAT yields an efficient decider for  $L$ , which simply converts its input  $x$  to  $\phi_{V,x}$  and invokes the decider for SAT on it.

This kind of efficient transformation, from an arbitrary instance of one problem to a corresponding instance of another problem having the same “yes/no answer”, is known as a *polynomial-time mapping reduction*, which we now formally define.

**Definition 152 (Polynomial-Time Mapping Reduction)** A *polynomial-time mapping reduction*—also known as a *Karp reduction* for short—from a language  $A$  to a language  $B$  is a function  $f: \Sigma^* \rightarrow \Sigma^*$  having the following properties:<sup>57</sup>

1. **Efficiency:**  $f$  is *polynomial-time computable*: there is an algorithm that, given any input  $x \in \Sigma^*$ , outputs  $f(x)$  in time polynomial in  $|x|$ .
2. **Correctness:** for all  $x \in \Sigma^*$ ,

$$x \in A \iff f(x) \in B .$$

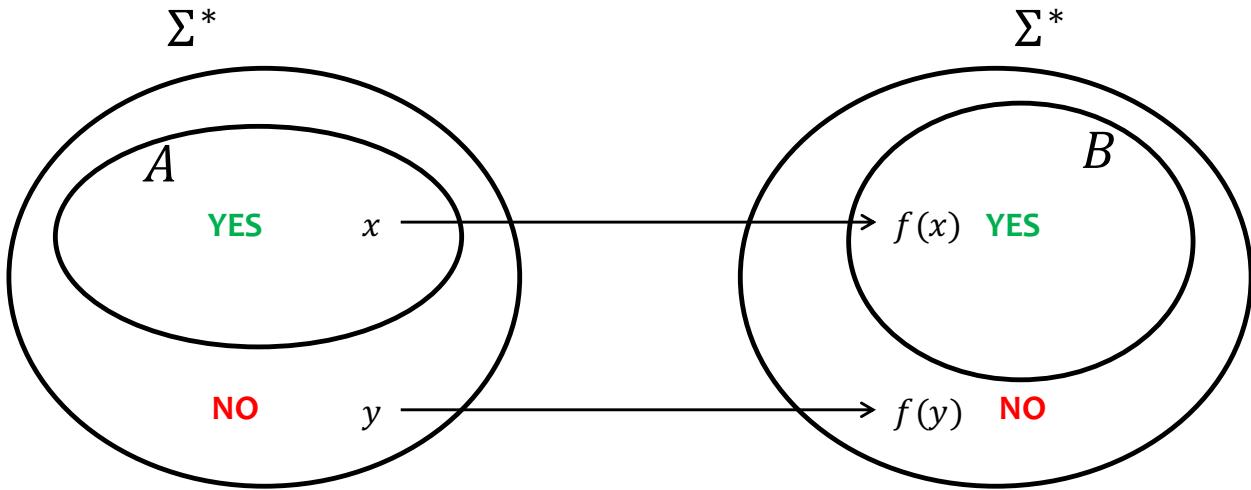
Equivalently, if  $x \in A$  then  $f(x) \in B$ , and if  $x \notin A$  then  $f(x) \notin B$ .<sup>58</sup>

If such a reduction exists, we say that  $A$  *polynomial-time mapping-reduces* (or *Karp-reduces*) to  $B$ , and write  $A \leq_p B$ .

---

<sup>57</sup> It is meaningful to drop the efficiency property and consider just (correct) mapping reductions, but these are not useful in the context of P and NP, so in this text we always require mapping reductions to be efficient.

<sup>58</sup> The claimed equivalence follows just by taking the contrapositive of the direction  $x \in A \iff f(x) \in B$ .



We first observe that a polynomial-time mapping reduction is essentially a special case of (or more precisely, implies the existence of) a *Turing reduction* (page 106): we can decide  $A$  by converting the input instance of  $A$  to an instance of  $B$  using the efficient transformation function, then querying an oracle that decides  $B$ . The following makes this precise.

**Lemma 153** *If  $A \leq_p B$ , then  $A \leq_T B$ .*

**Proof 154** Because  $A \leq_p B$ , there is an efficiently computable function  $f$  such that  $x \in A \iff f(x) \in B$ . To show that  $A \leq_T B$ , we give a Turing machine  $M_A$  that decides  $A$  using an oracle  $M_B$  that decides  $B$ :

```
function  $M_A(x)$ 
    compute  $y = f(x)$ 
    return  $M_B(y)$ 
```

Clearly,  $M_A$  halts on any input, because  $f$  is polynomial-time computable and  $M_B$  halts on any input. And by the correctness of  $M_B$  and the code of  $M_A$ ,

$$\begin{aligned} x \in A &\iff y = f(x) \in B \\ &\iff M_B(y) \text{ accepts} \\ &\iff M_A(x) \text{ accepts}. \end{aligned}$$

Therefore,  $M_A$  decides  $A$ , by Definition 61. □

Observe that the Turing reduction given in Proof 154 simply:

1. applies an efficient transformation to its input,
2. invokes its oracle *once*, on the resulting value, and
3. immediately outputs the oracle's answer.

In fact, several (but not all!) of the Turing reductions we have seen for proving undecidability (e.g., for the *halts-on-empty language* (page 108) and *other undecidable languages* (page 110)) are essentially polynomial-time mapping reductions, because they have this exact form. (To be precise, the *efficient transformation* in such a Turing reduction is the mapping reduction, and its use of its oracle is just fixed “boilerplate”.)

Although a polynomial-time mapping reduction is essentially a Turing reduction, the reverse does not hold in general, due to some important differences:

- A Turing reduction has no efficiency constraints, apart from the requirement that it halts: the reduction may run

for arbitrary finite time, both before and after its oracle call(s).

By contrast, in a polynomial-time mapping reduction, the output of the conversion function must be computable in time polynomial in the input size.

- A Turing reduction is a Turing machine that decides one language given an oracle (“black box”) that decides another language, which it may use arbitrarily. In particular, it may invoke the oracle multiple times (or not at all), and perform arbitrary “post-processing” on the results, e.g., negate the oracle’s answer.

By contrast, a polynomial-time mapping reduction does not involve any explicit oracle; it is merely a conversion function that must “preserve the yes/no answer”. Therefore, there is no way for it to “make multiple oracle calls,” nor for it to “post-process” (e.g., negate) the yes/no answer for its constructed instance. Implicitly, a polynomial-time mapping reduction corresponds to making just one oracle call and then immediately outputting the oracle’s answer.<sup>59</sup>

In Lemma 98 we saw that if  $A \leq_T B$  and  $B$  decidable, then  $A$  is also decidable. Denoting the class of decidable languages by  $R$ , this result can be restated as:

If  $A \leq_T B$  and  $B \in R$ , then  $A \in R$ .

Polynomial-time mapping reductions give us an analogous result with respect to membership in the class  $P$ .

**Lemma 155** *If  $A \leq_p B$  and  $B \in P$ , then  $A \in P$ .*

**Proof 156** Because  $B \in P$ , there is a polynomial-time Turing machine  $M_B$  that decides  $B$ . And because  $A \leq_p B$ , the Turing machine  $M_A$  defined in Proof 154, with the machine  $M_B$  as its “oracle”, decides  $A$ .

In addition,  $M_A(x)$  runs in time polynomial in its input size  $|x|$ , by composition of polynomial-time algorithms. Specifically, by the hypothesis  $A \leq_p B$ , computing  $f(x)$  runs in time polynomial in  $|x|$ , and as already noted,  $M_B$  runs in time polynomial in its input length, so the composed algorithm  $M_A$  runs in polynomial time.

Since  $M_A$  is a polynomial-time Turing machine that decides  $A$ , we conclude that  $A \in P$ , as claimed.  $\square$

Analogously to how Lemma 100 immediately follows from Lemma 98, the following corollary is merely the contrapositive of Lemma 155.

**Lemma 157** *If  $A \leq_p B$  and  $A \notin P$ , then  $B \notin P$ .*

## 17.2 NP-Hardness and NP-Completeness

With the notion of a polynomial-time mapping (or Karp) reduction in hand, the heart of the Cook-Levin theorem can be restated as saying that

$A \leq_p SAT$  for every language  $A \in NP$ .

Combining this with Lemma 155, as an immediate corollary we get the statement of Theorem 151: if  $SAT \in P$ , then every NP language is in P, i.e.,  $P = NP$ .

Informally, the above says that SAT is “at least as hard as” every language in NP (under Karp reductions). This is a very important property that turns out to be shared by many languages, so we define a special name for it.

**Definition 158 (NP-Hard)** A language  $L$  is **NP-hard** if  $A \leq_p L$  for every language  $A \in NP$ .

We define  $NPH = \{L : L \text{ is NP-hard}\}$  to be the class of all such languages.

<sup>59</sup> Alternatively, we could consider a relaxed definition that allows for multiple oracle calls and arbitrary polynomial-time pre- and post-processing. In other words, this would be a *polynomial-time Turing reduction*, also known as a *Cook reduction*. Such reductions are also very useful, but they do not allow us to make distinctions between complexity classes like NP and coNP, hence the focus on efficient mapping reductions.

We stress that a language need not be in NP, or even be decidable, to be NP-hard. For example, it can be shown that the undecidable language  $L_{\text{ACC}}$  is NP-hard (see [Exercise 167](#)).

With the notion of NP-hardness in hand, the core of the Cook-Levin theorem is as follows.

**Theorem 159 (Cook-Levin core)** *SAT is NP-hard.*

Previously, we mentioned the existence of languages that are, informally, the “hardest” ones in NP, and that SAT was one of them. We now formally define this notion.

**Definition 160 (NP-Complete)** A language  $L$  is *NP-complete* if:

1.  $L \in \text{NP}$ , and
2.  $L$  is NP-hard.

We define  $\text{NPC} = \{L : L \text{ is NP-complete}\}$  to be the class of all such languages.

Since  $\text{SAT} \in \text{NP}$  (by [Lemma 149](#)) and  $\text{SAT}$  is NP-hard (by Cook-Levin),  $\text{SAT}$  is indeed NP-complete.

To show that some language  $L$  of interest is NP-hard, do we need to repeat and adapt all the work of the Cook-Levin theorem, with  $L$  in place of SAT? Thankfully, we do not! Analogously to [Lemma 100](#)—which lets us prove *undecidability* by giving a *Turing* reduction from a known-undecidable language—the following lemma shows that we can establish *NP-hardness* by giving a *Karp* reduction from a known-NP-hard language. (We do this below for several concrete problems of interest, in [More NP-complete Problems](#) (page 169).)

**Lemma 161** *If  $A \leq_p B$  and  $A$  is NP-hard, then  $B$  is NP-hard.*

**Proof 162** This follows from the fact that  $\leq_p$  is a *transitive* relation; see [Exercise 165](#) below. By the two hypotheses and [Definition 158](#),  $L \leq_p A$  for every  $L \in \text{NP}$ , and  $A \leq_p B$ , so  $L \leq_p B$  by transitivity. Since this holds for every  $L \in \text{NP}$ , by definition we have that  $B$  is NP-hard, as claimed.  $\square$

Combining [Lemma 155](#), [Lemma 157](#), and [Lemma 161](#), the fact that  $A \leq_p B$  has the following implications in various scenarios.

Hypothesis	Implies
$A \in \text{P}$	nothing
$A \notin \text{P}$	$B \notin \text{P}$
$A$ is NP-hard	$B$ is NP-hard
$B \in \text{P}$	$A \in \text{P}$
$B \notin \text{P}$	nothing
$B$ is NP-hard	nothing

## 17.3 Resolving P versus NP

The concepts of NP-hardness and NP-completeness are powerful tools for making sense of, and potentially resolving, the P-versus-NP question. As the following theorem shows, the two possibilities  $\text{P} = \text{NP}$  and  $\text{P} \neq \text{NP}$  each come with a variety of equivalent “syntactically weaker” conditions. So, resolving P versus NP comes down to establishing *any one* of these conditions—which is still a very challenging task! In addition, the theorem establishes strict relationships between the various classes of problems we have defined, under each of the two possibilities  $\text{P} = \text{NP}$  and  $\text{P} \neq \text{NP}$ .

**Theorem 163 (P versus NP)** *The following statements are equivalent, i.e., if any one of them holds, then all of them hold.*

1. Some NP-hard language is in P.  
(In set notation:  $\text{NPH} \cap \text{P} \neq \emptyset$ .)
2. Every NP-complete language is in P.  
(In set notation:  $\text{NPC} \subseteq \text{P}$ .)
3.  $\text{P} = \text{NP}$ .  
(In words: every language in NP is also in P, and vice-versa.)
4. Every language, except the “trivial” languages  $\Sigma^*$  and  $\emptyset$ , is NP-hard.<sup>60</sup>  
(In set notation:  $\mathcal{P}(\Sigma^*) \setminus \{\Sigma^*, \emptyset\} \subseteq \text{NPH}$ .)

*It follows that  $\text{P} \neq \text{NP}$  if and only if no NP-hard language is in P (i.e.,  $\text{NPH} \cap \text{P} = \emptyset$ ), which holds if and only if some nontrivial language is not NP-hard.*

<sup>60</sup> We remark that Statement 4 must exclude the two trivial languages  $\Sigma^*$  and  $\emptyset$ , because they are not NP-hard, regardless of whether  $\text{P} = \text{NP}$ ; see Exercise 166.

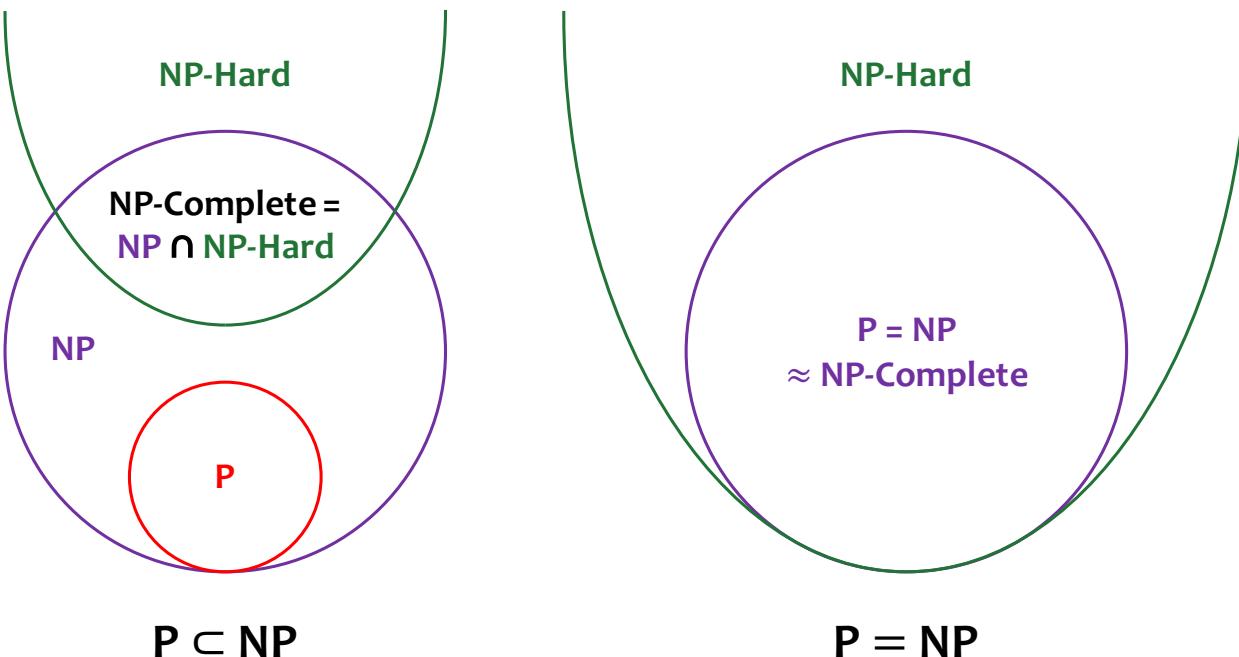
Before giving the proof of [Theorem 163](#), we discuss its main consequences for attacking the P-versus-NP question. First, by the equivalence of Statements 1 and 2, **all NP-complete problems have the same “status”**: either *all* of them are efficiently solvable—in which case  $\text{P} = \text{NP}$ , by the equivalence with Statement 3—or *none* of them is, in which case  $\text{P} \neq \text{NP}$ .

So, to prove that  $\text{P} = \text{NP}$ , it would be sufficient (and necessary) to have an efficient algorithm for *any single* NP-complete (or even just NP-hard) problem. This is by the equivalence of Statements 1 and 3.

To prove that  $\text{P} \neq \text{NP}$ , it would trivially suffice to prove that *any single* problem in NP is *not* in P. For this we might as well focus our efforts on showing this for some NP-complete problem, because by the equivalence of Statements 3 and 1, *some* NP problem lacks an efficient algorithm if and only if *every* NP-complete problem does.

Alternatively, to prove that  $\text{P} \neq \text{NP}$ , by the equivalence of Statements 3 and 4 it would suffice to show that some nontrivial language in NP is *not* NP-hard. For this we might as well focus our efforts on showing this for some language in P, because by the equivalence of Statements 4 and 1, if *any* nontrivial language in NP is not NP-hard, then *every* nontrivial language in P is not NP-hard.

Based on [Theorem 163](#), the following Venn diagram shows the necessary and sufficient relationships between the classes P, NP, NP-hard (NPH), and NP-complete (NPC), for each of the two possibilities  $\text{P} \neq \text{NP}$  and  $\text{P} = \text{NP}$ .



**Proof 164** We refer to Statement 1 as “S1”, and similarly for the others. The following implications hold immediately by the definitions (and set operations):

- S2  $\implies$  S1 because an NP-complete, and hence NP-hard, language exists (e.g., SAT).

(In set notation: if  $NPH \cap NP = NPC \subseteq P$ , then by intersecting with  $P$ , we get that  $NPH \cap P = NPC \neq \emptyset$ .)

- S3  $\implies$  S2 because by definition, every NP-complete language is in NP.

(In set notation: if  $P = NP$ , then  $NPC \subseteq NP \subseteq P$ .)

- S4  $\implies$  S1 because there is a nontrivial language  $L \in P \subseteq NP$  (e.g.,  $L = MAZE$ ), and by S4,  $L$  is NP-hard.

(In set notation:  $\emptyset \neq P \setminus \{\Sigma^*, \emptyset\} \subseteq NPH$ , where the last inclusion is by S4, so by intersecting everything with  $P$ , we get that  $\emptyset \neq P \setminus \{\Sigma^*, \emptyset\} \subseteq NPH \cap P$  and hence  $NPH \cap P \neq \emptyset$ .)

So, to prove that each statement implies every other one, it suffices to show that  $S1 \implies S3 \implies S4$ .

For  $S1 \implies S3$ , suppose that some NP-hard language  $L$  is in  $P$ . By definition,  $A \leq_p L$  for all  $A \in NP$ , so by Lemma 155,  $A \in P$ . Therefore,  $NP \subseteq P$ , and this is an equality because  $P \subseteq NP$ , as we have already seen.

For  $S3 \implies S4$ , suppose that  $P = NP$  and let  $L$  be an arbitrary nontrivial language; we show that  $L$  is NP-hard. By nontriviality, there exist some *fixed, distinct* strings  $y \in L$  and  $z \notin L$ . Let  $A \in NP = P$  be arbitrary, so there is an efficient algorithm  $M_A$  that decides  $A$ . We show that  $A \leq_p L$  via the function  $f$  computed by the following pseudocode:

```
function F(x)
  if  $M_A(x)$  accepts then return  $y$ 
  return  $z$ 
```

It is apparent that  $F$  is efficient, because  $M_A$  is. And for correctness, by the correctness of  $M_A$ , the properties of

$y \neq z$ , and the code of  $F$ ,

$$\begin{aligned} x \in A &\iff M_A(x) \text{ accepts} \\ &\iff f(x) = y \\ &\iff f(x) \in L . \end{aligned}$$

Since  $A \leq_p L$  for all  $A \in \text{NP}$ , we have shown that  $L$  is NP-hard, as needed.  $\square$

**Exercise 165** Show that polynomial-time mapping reductions are *transitive*. That is, if  $A \leq_p B$  and  $B \leq_p C$ , then  $A \leq_p C$ .

**Exercise 166** Show that *no* language Karp-reduces to  $\Sigma^*$  or to  $\emptyset$ , except for  $\Sigma^*$  and  $\emptyset$  themselves (respectively). In particular, these languages are *not* NP-hard, regardless of whether  $P = NP$ .

**Exercise 167** Show that  $L_{\text{ACC}}$  is NP-hard.

## MORE NP-COMPLETE PROBLEMS

The Cook-Levin Theorem (Theorem 159, showing that SAT is NP-hard), together with the concept of polynomial-time mapping reductions and Lemma 161, are powerful tools for establishing the NP-hardness of a wide variety of other natural problems. We do so for several such problems next.

### 18.1 3SAT

As a second example of an NP-complete problem, we consider satisfiability of Boolean formulas having a special structure. First, we introduce some more relevant terminology.

- A *clause* is a disjunction (OR) of literals, like  $(a \vee \neg b \vee \neg c \vee d)$ .
- A Boolean formula is in *conjunctive normal form (CNF)* if it is a conjunction (AND) of clauses. An example of a CNF formula is

$$\phi = (a \vee b \vee \neg c \vee d) \wedge (\neg a) \wedge (a \vee \neg b) \wedge (c \vee d).$$

- A *3CNF formula* is a Boolean formula in conjunctive normal form where **each clause has exactly three literals**, like the following:

$$\phi = (a \vee b \vee \neg c) \wedge (\neg a \vee \neg a \vee d) \wedge (a \vee \neg b \vee d) \wedge (c \vee d \vee d).$$

Notice that clauses may have repeated literals, even though this has no effect on the satisfiability of the formula.

**Definition 168 (3SAT, 3CNF Satisfiability)** The 3CNF-satisfiability language is defined as the set of satisfiable 3CNF formulas:

$$3SAT = \{\phi : \phi \text{ is a satisfiable 3CNF formula}\}.$$

Observe that every satisfiable 3CNF formula is a satisfiable Boolean formula, but not vice versa. Conceivably, the extra “structure” of 3CNF formulas might make it easier to decide whether they are satisfiable, as compared with general Boolean formulas. However, it turns out that this is not the case: the problem of deciding 3CNF satisfiability is NP-complete.<sup>61</sup>

**Theorem 169** *3SAT is NP-complete.*

First, we must show that  $3SAT \in NP$ . An efficient verifier for 3SAT is actually identical to the one for SAT, except that its input is a 3CNF formula. (As usual, the verifier interprets any input string as being an instance of the relevant language.)

---

<sup>61</sup> Interestingly, it turns out that 2SAT, the language of satisfiable CNF formulas having *two* literals per clause, is in  $P$ ! That is, given a 2CNF formula we can efficiently decide whether it is satisfiable. This is one of many cases where a slight change in the structure of problem can make an enormous difference in its complexity.

**Input:** instance: a 3CNF formula; certificate: assignment to its variables

**Output:** whether the assignment satisfies the formula

```

function  $V(\phi, \alpha)$ 
  if  $\phi(\alpha) = 1$  then accept
  reject

```

By the same reasoning that applied to the verifier for SAT (see Lemma 149), this is an efficient and correct verifier for 3SAT.

We now show that 3SAT is NP-hard, by proving that  $SAT \leq_p 3SAT$ . We do so by defining an efficient mapping reduction  $f$  that, given an arbitrary Boolean formula  $\phi$ , outputs a 3CNF formula  $f(\phi)$  such that  $\phi \in SAT \iff f(\phi) \in 3SAT$ . The reduction  $f$  and its analysis are as follows.

1. First, convert  $\phi$  to a formula  $\phi'$  in *conjunctive normal form* that is satisfiable if and only if  $\phi$  is; we say that  $\phi$  and  $\phi'$  are *equisatisfiable*. (We stress that the formulas are not necessarily *equivalent*, because they may have different variables.) We can perform this transformation efficiently, *as described in detail below* (page 170).
2. Then, convert the CNF formula  $\phi'$  to the output 3CNF formula  $f(\phi)$ , as follows.
  - a) For each clause having more than three literals, split it into a pair of clauses by introducing a new “dummy” variable. Specifically, convert a clause

$$(l_1 \vee l_2 \vee \dots \vee l_k)$$

with  $k > 3$  literals into the two clauses

$$(z \vee l_1 \vee l_2) \wedge (\neg z \vee l_3 \vee \dots \vee l_k),$$

where  $z$  is some new variable that does not appear anywhere else in the formula. Observe that these two clauses have 3 and  $k - 1$  literals, respectively, so the two new clauses both have strictly fewer literals than the original clause.<sup>62</sup> We thus repeat the splitting process until every clause has at most three literals.

We claim that each splitting preserves (un)satisfiability, i.e., the original formula before splitting and new CNF formula are equisatisfiable (i.e., either both are satisfiable, or neither is). Indeed, more generally, we claim that the formulas  $(\rho \vee \rho')$  and  $\sigma = (z \vee \rho) \wedge (\neg z \vee \rho')$  are equisatisfiable for *any* formulas  $\rho, \rho'$  that do not involve variable  $z$ .

- In one direction, suppose that  $\sigma$  is satisfiable, i.e., it has a satisfying assignment  $\alpha$ . Then the same assignment (ignoring the value of  $z$ ) satisfies  $(\rho \vee \rho')$ , because  $\alpha$  makes exactly one of  $z, \neg z$  false, so it must make the corresponding one of  $\rho, \rho'$  true, and therefore satisfies  $(\rho \vee \rho')$ .
  - In the other direction, suppose that  $(\rho \vee \rho')$  is satisfiable, i.e., it has a satisfying assignment  $\alpha$ . Then  $\alpha$  leaves  $z$  unassigned (because  $z$  does not appear in  $\rho$  or  $\rho'$ ), and it makes at most one of  $\rho, \rho'$  false; we therefore assign  $z$  to make the corresponding literal  $z$  or  $\neg z$  true. (If  $\alpha$  satisfies both  $\rho, \rho'$ , we assign  $z$  arbitrarily.) This satisfies  $\sigma$  because it makes both of  $(z \vee \rho)$  and  $(\neg z \vee \rho')$  true.
- b) Finally, for each clause that has fewer than three literals, expand it to have three literals simply by repeating its first literal. For example, the clause  $(a)$  becomes  $(a \vee a \vee a)$ , and  $(\neg b \vee c)$  becomes  $(\neg b \vee \neg b \vee c)$ . Trivially, this produces an equivalent formula in 3CNF form.

Each transformation on a clause can be done in constant time, and the total number of transformations is linear in the number of literals in (i.e., size of)  $\phi'$ . This is because the splitting transformation transforms a clause having  $k > 3$  literals into a pair having 3 and  $k - 1$  literals. So, the running time of this phase is linear in the size of  $\phi'$  (and in particular, the size of  $f(\phi)$  is linear in the size of  $\phi'$ ).

Because both phases run in polynomial time in the length of their respective inputs, their composition runs in time polynomial in the length of the input formula  $\phi$ , as needed. This completes the proof that  $SAT \leq_p 3SAT$ .

<sup>62</sup> Observe that this splitting technique cannot produce smaller clauses having only *two* literals each, because splitting a clause with three or more literals yields some new clause having at least three literals. This is why this approach does not yield a polynomial-time mapping reduction from SAT to 2SAT. Nor should we expect it to: as mentioned above,  $2SAT \in P$ , so such a reduction would imply that  $P = NP$ !

### Efficiently converting a formula to CNF

Here we show how to efficiently convert any formula  $\phi$  into another formula  $\phi'$  in *conjunctive normal form* (CNF), so that the two formulas are equisatisfiable:  $\phi$  is satisfiable if and only if  $\phi'$  is. Recall that a formula is in CNF if it is a conjunction (AND) of disjunctions (ORs) of literals.

As a warm-up, observe that we can convert a formula to an *equivalent* CNF formula, via the distributive rule for OR over AND. Specifically, we can convert  $\rho \vee (\sigma \wedge \tau)$  to the equivalent formula  $(\rho \vee \sigma) \wedge (\rho \vee \tau)$ , for any subformulas  $\rho, \sigma, \tau$ . By repeatedly applying this rule, we can “push ORs inward” until we get a CNF formula. Unfortunately, this conversion is *not efficient* in general: because the distributive rule makes two copies of  $\rho$ , each application of the rule can nearly double the size of the formula, so the formula size can blow up exponentially overall. An example formula for which this happens is  $(x_1 \wedge y_1) \vee \dots \vee (x_k \wedge y_k)$ , which expands to the AND of  $2^k$  clauses of the form  $(v_1 \vee \dots \vee v_k)$ , where each  $v_i$  is one of  $x_i$  or  $y_i$ .

Instead, we seek an *efficient* transformation algorithm, i.e., one that runs in time polynomial in the size of the input formula (so in particular, the size of the output formula must also be polynomial). Here we describe an algorithm known as the *Tseitin transformation* (sometimes spelled Tseytin), which runs in *linear* time.

First, recall that by De Morgan’s laws, we assume without loss of generality that any negations in the formula apply directly to variables, i.e., that we have “pushed negations inward” to the variables themselves. So, any formula is one of:

- the base case: a literal, like  $x$  or  $\neg y$ ;
- the OR of at least two subformulas, like  $x \vee y$  or  $(\neg x) \vee y \vee (x \wedge \neg z)$ ;
- the AND of at least two subformulas, like  $x \wedge y$  or  $(x \vee y) \wedge (y \vee z) \wedge (\neg z)$ .

The first idea of the Tseitin transformation is that the new formula  $\phi'$  has all the variables of the input formula  $\phi$ , plus one *new variable* for each *non-base-case subformula* within  $\phi$  (including  $\phi$  itself, if applicable). Thus, each subformula in  $\phi$ , whether base case or not, has an associated literal: each (base-case) literal is associated with itself, and each (non-base-case) OR/AND subformula is associated with its corresponding new variable.

For example, the following shows a small formula  $\phi$  and the new variables  $s_i$  associated with its subformulas:

$$\phi = \underbrace{(\neg x \wedge y \wedge \neg y)}_{s_1} \vee x .$$

The second idea is that for each non-base-case subformula  $\rho$  in  $\phi$ , the new formula enforces the constraint that  $\rho$ ’s *associated variable* must equal  $\rho$ ’s *value*. This is done via a conjunction of clauses (ANDs of ORs of literals), as follows:

- For an OR of  $k$  subformulas, let  $s$  denote the (new) variable associated with the OR expression, and let  $\ell_1, \dots, \ell_k$  be the literals associated with the  $k$  subformulas. Then it can be seen that the conjunction of clauses

$$(\neg s \vee \ell_1 \vee \dots \vee \ell_k) \wedge (s \vee \neg \ell_1) \wedge \dots \wedge (s \vee \neg \ell_k) .$$

is logically equivalent to the constraint  $s = \ell_1 \vee \dots \vee \ell_k$ . That is, an assignment satisfies the conjunction if and only if the equality holds.

For example, the above example formula  $\phi$  is associated with  $s_1$ , and is the OR of two subformulas whose associated literals are  $s_2, x$  (note that one of the subformulas is a base case, and the other is not). So, we get the conjunction of clauses

$$(\neg s_1 \vee s_2 \vee x) \wedge (s_1 \vee \neg s_2) \wedge (s_1 \vee \neg x) .$$

- For an AND of  $k$  subformulas, let  $s$  denote the (new) variable associated with the AND expression, and let  $\ell_1, \dots, \ell_k$  be the literals associated with the  $k$  subformulas. Then it can be seen that the conjunction of clauses

$$(s \vee \neg\ell_1 \vee \dots \vee \neg\ell_k) \wedge (\neg s \vee \ell_1) \wedge \dots \wedge (\neg s \vee \ell_k)$$

is logically equivalent to the constraint  $s = \ell_1 \wedge \dots \wedge \ell_k$ .

For example, the above example has the subformula  $(\neg x \wedge y \wedge \neg y)$ , which is associated with  $s_2$ , and is the AND of three base-case subformulas whose associated literals are  $\neg x, y, \neg y$ . So, we get the conjunction of clauses

$$(s_2 \vee x \vee \neg y \vee y) \wedge (\neg s_2 \vee \neg x) \wedge (\neg s_2 \vee y) \wedge (\neg s_2 \vee \neg y).$$

Finally, the output formula  $\phi'$  is the AND of all the clauses obtained from *all* the non-base-case subformulas of  $\phi$  (including  $\phi$  itself, if applicable), *and* the literal associated with  $\phi$  itself. By construction,  $\phi'$  is a CNF formula.

For the above example formula  $\phi$ , the output formula is

$$\begin{aligned} \phi' = & s_1 \wedge \\ & (\neg s_1 \vee s_2 \vee x) \wedge (s_1 \vee \neg s_2) \wedge (s_1 \vee \neg x) \wedge \\ & (s_2 \vee x \vee \neg y \vee y) \wedge (\neg s_2 \vee \neg x) \wedge (\neg s_2 \vee y) \wedge (\neg s_2 \vee \neg y). \end{aligned}$$

In general, we can see that  $\phi$  and  $\phi'$  are equisatisfiable:

- If  $\phi$  has a satisfying assignment  $\alpha$ , then we can get a satisfying assignment for  $\phi'$  by extending  $\alpha$ , setting each new variable  $s_i$  to be the truth value of its associated subformula under  $\alpha$ . As argued above, this satisfies all the clauses obtained from all the subformulas of  $\phi$ , and since  $\alpha$  satisfies  $\phi$ , this also makes the literal associated with  $\phi$  itself true, so it satisfies  $\phi'$ .
- If  $\phi'$  has a satisfying assignment  $\alpha$ , then we claim that the same assignment (ignoring the new variables) also satisfies  $\phi$ . Since  $\alpha$  satisfies all the clauses of  $\phi'$ , as argued above, each subformula of  $\phi$  evaluates (under  $\alpha$ ) to the value of the subformula's associated variable. Moreover, the literal associated with  $\phi$  itself (which is also a clause of  $\phi'$ ) is also true under  $\alpha$ , so  $\alpha$  satisfies  $\phi$ .

Finally, we claim that the transformation runs in linear time, and in particular, the size of  $\phi'$  is linear in the size of  $\phi$ . (Recall that the size of a formula is the number of literals that appears in it.) Each non-base-case (AND/OR) subformula of  $\phi$  has some  $k \geq 2$  component subformulas, and is converted to clauses having a total of  $3k + 1 < 4k$  literals, in  $O(k)$  time. Across all such subformulas in  $\phi$ , the total number of components (i.e., the sum of all  $k$ ) is less than twice the number of literals in  $\phi$ , because each such subformula has at least two components. (In other words, the total number of nodes in a rooted tree, where every internal node has at least two children, is less than twice the number of leaves.) So,  $\phi'$  is at most 8 times as large as  $\phi$ .

### Modifying the proof of Cook-Levin to show that 3SAT is NP-hard

We can modify our proof of the Cook-Levin theorem to directly show that 3SAT is NP-hard. In particular, we demonstrate how to construct  $\phi_{V,x}$  so that it is in conjunctive normal form. We can then apply the transformation discussed above to turn a CNF formula into a 3CNF one.

We first observe that a CNF formula can be constructed recursively from smaller subformulas:

- A formula  $\phi = x$  with a single variable is in CNF – it has the single clause  $(x)$ .
- A formula  $\phi = \phi_1 \wedge \phi_2$  is in CNF if  $\phi_1$  and  $\phi_2$  are in CNF – it consists of the combination of the clauses in  $\phi_1$  and  $\phi_2$ .

- A formula  $\phi = \phi_1 \vee \phi_2$  is only in CNF if  $\phi_1$  and  $\phi_2$  each have a single clause –  $\phi$  is then just a single clause that combines the literals in  $\phi_1$  and  $\phi_2$ .

Examining each piece of  $\phi_{V,x}$ , we see:

- The cell-consistency subformula

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} \left[ \bigvee_{s \in S} t_{i,j,s} \wedge \bigwedge_{\text{distinct } s,s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'}) \right]$$

is in CNF. The inner conjunction is a conjunction of disjunctions, so it is clearly in CNF. The inner disjunction is a single clause, so it is in CNF. We combine the two with a conjunction, and the result is also in CNF since the two subformulas are in CNF. Finally, the outer conjunction also produces a CNF formula since the individual pieces are in CNF.

- The acceptance subformula

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} t_{i,j,q_{\text{acc}}}$$

is a single clause in CNF.

- The starting-configuration subformula

$$\phi_{\text{start},x} = t_{1,1,\#} \wedge t_{1,2,q_{\text{start}}} \wedge \phi_{\text{input}} \wedge t_{1,n+3,\$} \wedge \phi_{\text{cert}} \wedge t_{1,n^k,\#}$$

is in CNF – it consists of  $n^k$  clauses, each with either a single literal (e.g.,  $(t_{1,1,\#})$ ) or  $|\Gamma|$  literals (for each clause in  $\phi_{\text{cert}}$ ).

- The transition formula for a window with upper-left corner at  $i,j$

$$\phi_{i,j} = \bigvee_{w \in W} \phi_{i,j,w}$$

is **not** in CNF. The individual  $\phi_{i,j,w}$  consist of a sequence of conjunctions, so  $\phi_{i,j}$  is a disjunction of conjunctions, rather than a conjunction of disjunctions required for CNF. However, we can convert  $\phi_{i,j}$  to CNF by applying the distributive law for  $\vee$  and  $\wedge$ . In particular, we have

$$(\bigwedge_i a_i) \vee (\bigwedge_j b_j) = \bigwedge_{i,j} (a_i \vee b_j) .$$

We can see that this law holds if we consider two cases:

- If all  $a_i$  are true, then the left-hand side is true. The right-hand side is also true – each clause  $(a_i \vee b_j)$  contains an  $a_i$ , so each clause is satisfied, which makes the formula as a whole true.  
This same reasoning applies for the case when all  $b_j$  are true.
- If at least one  $a_i$  is false and at least one  $b_j$  is false, then the left-hand side is false. The right-hand side is also false, since there is a clause  $(a_i \vee b_j)$  where both  $a_i$  and  $b_j$  are false.

How does the size of the right-hand formula compare to the left-hand one? If the two conjunctions on the left contain  $m$  and  $n$  literals, respectively, then the left-hand side has  $m + n$  total literals. The right-hand side has  $m \cdot n$  clauses, each with two literals, for a total size of  $2mn$ . When  $m = n$ , we go from  $2m$  literals to a size of  $2m^2$ .

Suppose we have  $c$  conjunctions on the left-hand side, each with  $m$  literals. Then repeated application of the distributive law takes us from a formula with  $cm$  literals to one with  $cm^c$ . For  $\phi_{i,j}$ , we go from a size of  $6|W|$  in its original form to a size of  $|W| \cdot 6^{|W|}$  in CNF. This is exponential in the number of distinct valid windows  $|W|$ , but this number is a characteristic of the verifier  $V$ , not the input  $x$ . So even though it is a larger constant than before, it is still a constant.

Once we have  $\phi_{i,j}$  in CNF, then

$$\phi_{\text{move},V} = \bigwedge_{\substack{1 \leq i \leq n^k - 1 \\ 1 \leq j \leq n^k - 2}} \phi_{i,j}$$

is also in CNF. And while its size is larger than previously, it is only by a constant factor, and its total size is still  $O(n^{2k})$  literals.

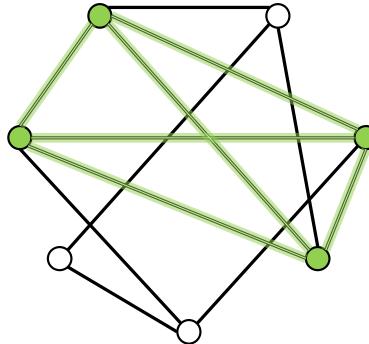
Since  $\phi_{V,x}$  is just a conjunction of the individual pieces above, it is in CNF if those pieces are each in CNF. We can then apply the CNF-to-3CNF transformation to obtain a formula in 3CNF. Thus, if we can decide 3SAT, we can decide an arbitrary language in NP, and 3SAT is NP-hard.

## 18.2 Clique

NP-completeness arises in many problems beyond Boolean satisfiability. As an example, we consider the *clique* problem.

For an undirected graph  $G = (V, E)$ , a *clique* is a subset  $C \subseteq V$  of the vertices for which there is an edge between every pair of (distinct) vertices in  $C$ . Equivalently, the subgraph *induced* by  $C$ , meaning the vertices in  $C$  and all the edges between them, is a complete graph.

The following is an example of a clique of size four, with the vertices of the clique and the edges between them highlighted:



We are often interested in finding a clique of maximum size in a graph, called a *maximum clique* for short. For instance, if a graph represents a group of people and their individual friend relationships, we might want to find a largest set of mutual friends—hence the name “clique”. However, this is not a decision problem; to obtain one, much like we did for the *traveling salesperson problem* (page 140), here we introduce a non-negative “threshold” parameter  $k$ , which does not exceed the number of vertices in the graph (because no clique can exceed this size):

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph that has a clique of size (at least) } k\}.$$

Since the search version of the clique problem is a *maximization* problem, the corresponding decision problem involves a *lower bound* on the clique size. Observe that if a graph has a clique of size *at least*  $k$ , then it also has one of size *exactly*  $k$ , because removing arbitrary vertices from a clique still results in a clique. In other words, any subset of a clique is itself a clique.

**Theorem 170** CLIQUE is NP-complete.

To prove the theorem, we show in the following two lemmas that CLIQUE is in NP, and is NP-hard.

**Lemma 171**  $\text{CLIQUE} \in \text{NP}$ .

**Proof 172** To prove the lemma we give an efficient verifier for CLIQUE, according to Definition 139. Our verifier takes an instance—i.e., a graph  $G$  and threshold  $k$ —and a certificate, which is a set of vertices that is claimed to be a clique of size  $k$  in the graph. The verifier simply checks whether this is indeed the case, i.e., whether the certificate has  $k$  vertices and there is an edge between each pair of these vertices.

**Input:** instance: an undirected graph and non-negative integer  $k$ ; certificate: a subset of vertices in the graph

**Output:** whether the vertices form a clique of size  $k$  in the graph

```
function VERIFYCLIQUE(( $G = (V, E)$ ,  $k$ ),  $C \subseteq V$ )
  if  $|C| \neq k$  then reject
  for all distinct  $u, v \in C$  do
    if  $(u, v) \notin E$  then reject
  accept
```

We first analyze the running time of the verifier. The first check counts the number of vertices in the vertex subset  $C \subseteq V$ , which runs in linear time in the size of the graph. Then the number of loop iterations is quadratic in the number of vertices, and each iteration can be done efficiently by looking up edges of the graph. So, the total running time is polynomial in the size of the instance, as needed.

We now prove the verifier's correctness. We need to show that  $(G, k) \in \text{CLIQUE}$  if and only if there exists some (polynomial-size)  $C \subseteq V$  for which  $\text{VERIFYCLIQUE}((G, k), C)$  accepts.

In one direction, if  $(G, k) \in \text{CLIQUE}$ , then by definition there is some clique  $C \subseteq V$  of size  $k$ , which is of size polynomial (indeed, linear) in the size of the instance  $(G, k)$ . In  $\text{VERIFYCLIQUE}((G, k), C)$ , all the checks pass because  $|C| = k$  and there is an edge between every pair of vertices in  $C$ . Thus, the verifier accepts  $((G, k), C)$ , as needed.

Conversely, if  $\text{VERIFYCLIQUE}((G, k), C)$  accepts for some certificate  $C \subseteq V$ , then by the code of the verifier,  $|C| = k$  and there is an edge between every pair of vertices in  $C$ . So, by definition,  $C$  is a clique of size  $k$  in  $G$ , hence  $(G, k) \in \text{CLIQUE}$ , as needed.  $\square$

**Lemma 173**  $\text{CLIQUE}$  is NP-hard.

To prove the lemma we will give a polynomial-time mapping reduction from the NP-hard language 3SAT. Before doing so, let's review the definition of 3SAT. An instance is a 3CNF formula, which is the AND of clauses that are each the OR of three literals:

$$\phi = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_4 \vee \ell_5 \vee \ell_6) \wedge \cdots \wedge (\ell_{3m-2} \vee \ell_{3m-1} \vee \ell_{3m}).$$

Each literal  $\ell_i$  is either a variable itself (e.g.,  $x$ ) or its negation (e.g.,  $\neg x$ ). By definition, 3SAT is the language of all *satisfiable* 3CNF formulas; a formula is satisfiable if its variables can be assigned true/false values that make the formula evaluate to true.

Observe that a satisfying assignment simultaneously satisfies *every* clause, meaning that each clause has *at least one literal* that is true. So, a formula with  $m$  clauses is satisfiable if and only if there is some selection of  $m$  literals, one from each clause, that can simultaneously be made true under some assignment. This is a key fact we use in the reduction to CLIQUE.

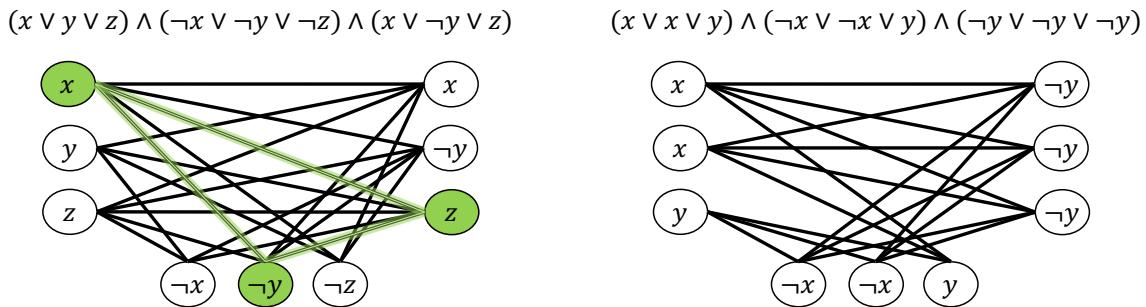
**Proof 174** We give a polynomial-time mapping reduction  $f$  from 3SAT to CLIQUE, showing that  $\text{3SAT} \leq_p \text{CLIQUE}$ . Since 3SAT is NP-hard, it follows that CLIQUE is NP-hard.

We need to define a polynomial-time computable function  $f$  that transforms a given 3CNF formula  $\phi$  into a

corresponding instance  $f(\phi) = (G, k)$  of the clique problem, so that  $\phi \in 3\text{SAT} \iff f(\phi) \in \text{CLIQUE}$ . Given a 3CNF formula  $\phi$ , the reduction constructs a CLIQUE instance  $(G, k)$  with graph  $G$  and threshold  $k$  defined as follows:

- For each literal in  $\phi$ , including duplicates, there is a corresponding vertex in  $G$ . So, a formula with  $m$  clauses yields a graph  $G$  with exactly  $3m$  vertices.
- For each pair of vertices in  $G$ , there is an edge between them if their corresponding literals are in *different* clauses of  $\phi$ , and are “*consistent*”—i.e., they can simultaneously be true. Specifically, two literals are consistent if they are not negations of each other (i.e.,  $x$  and  $\neg x$  for some variable  $x$ ). In other words, two literals are consistent if they are the same literal, or they involve different variables.
- The threshold is set to  $k = m$ , the number of clauses in  $\phi$ .

For example, the following illustrates the graphs corresponding to two different formulas having three clauses each, where one formula is satisfiable and the other is unsatisfiable:



Observe that for the satisfiable formula on the left, the corresponding graph has a clique of size three (one such clique is highlighted), while the graph for the unsatisfiable formula on the right does not have a clique of size three.

We first show that the reduction is efficient. Creating one vertex per literal in the input formula takes linear time in the size of the formula, and creating the edges takes quadratic time, because for each pair of literals in different clauses, it takes constant time to determine whether they are consistent. So, the reduction is polynomial time in the size of the input formula.

We now prove that the reduction is correct. We need to show that  $\phi \in 3\text{SAT} \iff f(\phi) = (G, k) \in \text{CLIQUE}$ .

We first show that  $\phi \in 3\text{SAT} \implies (G, k) \in \text{CLIQUE}$ . Since  $\phi \in 3\text{SAT}$ , it is satisfiable, which means there is some assignment  $\alpha$  under which each clause in  $\phi$  has at least one literal that is true. We show that there is a corresponding clique in  $G$  of size  $k = m$ , the number of clauses in  $\phi$ . Let  $s_1, s_2, \dots, s_m$  be a selection of one literal from each clause that is true under the assignment  $\alpha$ . Then we claim that their corresponding vertices form a clique in  $G$ :

- Since all the literals  $s_i$  are true under assignment  $\alpha$ , each pair of them is consistent, i.e., no two of them are a variable and its negation.
- So, since all the literals  $s_i$  come from different clauses, each pair of their corresponding vertices has an edge between them, by construction of the graph.

Therefore,  $G$  has a clique of size  $k = m$ , so  $(G, k) \in \text{CLIQUE}$ , as needed.

We now show that converse, that  $f(\phi) = (G, k) \in \text{CLIQUE} \implies \phi \in 3\text{SAT}$ . It is important to understand that we are **not** “inverting” the function  $f$  here. Rather, we are showing that **if the graph-threshold pair produced as the output of  $f(\phi)$  is in CLIQUE**, then the input formula  $\phi$  must be in 3SAT.

Since  $f(\phi) = (G, k = m) \in \text{CLIQUE}$ , the output graph  $G$  has some clique  $C = \{c_1, \dots, c_m\}$  of size  $m$ . We show that  $\phi$  is satisfiable, by using the vertices in  $C$  to construct a corresponding assignment  $\alpha$  that satisfies  $\phi$ .

- Since  $C$  is a clique, there is an edge between every pair of vertices in  $C$ . So, by the reduction's construction of the graph, the vertices in  $C$  must correspond to *consistent* literals from *different* clauses.
- Since there are  $m$  vertices in  $C$ , they correspond to exactly one literal from each clause.
- Since these literals are consistent, it is possible for all of them to be true simultaneously. That is, no two of these literals are a variable and its negation.
- Therefore, we can define an assignment  $\alpha$  to the variables of  $\phi$  so that each literal corresponding to a vertex in  $C$  is true. That is, if a variable  $x$  is one such literal, we set  $x$  to be true; and if a negated variable  $\neg x$  is one such literal, we set  $x$  to be false. (Any variables that remain unset after this process can be set arbitrarily.) Since all the literals in question are consistent, the assignment is well defined; there is no conflict in setting them.

We can now see that  $\alpha$  is a satisfying assignment: the literals corresponding to the vertices in  $C$  are all true under  $\alpha$ , and they collectively come from all  $m$  clauses, so  $\alpha$  satisfies  $\phi$ . Thus  $\phi \in \text{3SAT}$ , as needed. This completes the proof.  $\square$

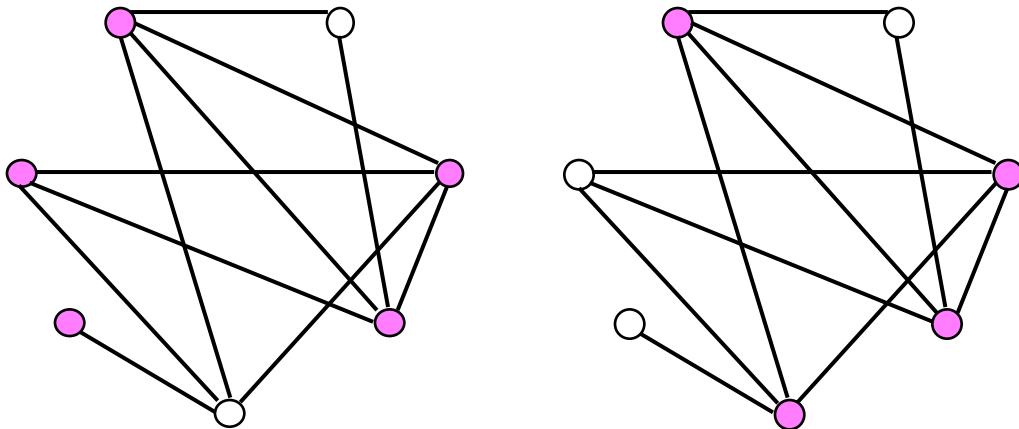
It is worth remarking that the above reduction from 3SAT to CLIQUE produces “special-looking” instances  $(G, k)$  of CLIQUE: the graph  $G$  has exactly  $3k$  vertices, which are in groups of three where there is no edge among any such group. There is no issue with this: an (efficient) algorithm that solves CLIQUE must be correct on special instances of this form (along with all others), so any such algorithm could also be used to (efficiently) solve 3SAT by applying the reduction to the input formula and invoking the CLIQUE algorithm on the result. Indeed, the reduction implies that solving CLIQUE even when it is restricted to such special instances is “at least as hard as” solving any problem in NP.

## 18.3 Vertex Cover

For an undirected graph  $G = (V, E)$ , a *vertex cover* is a subset  $C \subseteq V$  of the vertices for which every edge in the graph is “covered” by a vertex in  $C$ . An edge  $e = (u, v)$  is *covered* by  $C$  if at least one of its endpoints  $u, v$  is in  $C$ . So, formally,  $C \subseteq V$  is a vertex cover for  $G$  if

$$\forall e = (u, v) \in E, u \in C \vee v \in C.$$

The following are two vertex covers for the same graph:



The cover on the left has a size of five (vertices), while the cover on the right has a size of four.

Given a graph, we are interested in finding a vertex cover of *minimum size*, called a *minimum vertex cover* for short. Observe that  $V$  itself is a vertex cover in all cases, though it may be far from optimal.

As a motivating example for the vertex-cover problem, consider a museum that consists of a collection of interconnected hallways, with exhibits on the walls. The museum needs to hire guards to monitor and protect the exhibits, but since

guards are expensive, it seeks to minimize the number of guards while still ensuring that each hallway is protected by at least one guard. The museum layout can be represented as a graph, with the hallways as edges and the vertices as hallway intersections or endpoints. A guard placed at a vertex protects the exhibits in any of the hallways adjacent to that vertex. So, the museum wants to find a minimum vertex cover for this graph, to determine how many guards to hire and where to place them.

As we did for the *traveling salesperson problem* (page 140), we first define a decision version of the vertex cover problem, with a non-negative “budget” parameter  $k$  that does not exceed the number of vertices in the graph:

$$\text{VERTEX-COVER} = \{(G, k) : G \text{ is an undirected graph that has a vertex cover of size (at most) } k\}.$$

Since the search version of vertex cover is a *minimization* problem, the corresponding decision problem involves an *upper bound* (“budget”) on the vertex cover size. Observe that if a graph has a vertex cover of size *less than*  $k$ , then it also has one of size *exactly*  $k$ , simply by adding some arbitrary vertices. In other words, any superset of a vertex cover is also a vertex cover.

We now show that VERTEX-COVER is NP-hard, leaving the proof that VERTEX-COVER  $\in$  NP as an exercise. Combining these two results, we conclude that VERTEX-COVER is NP-complete.

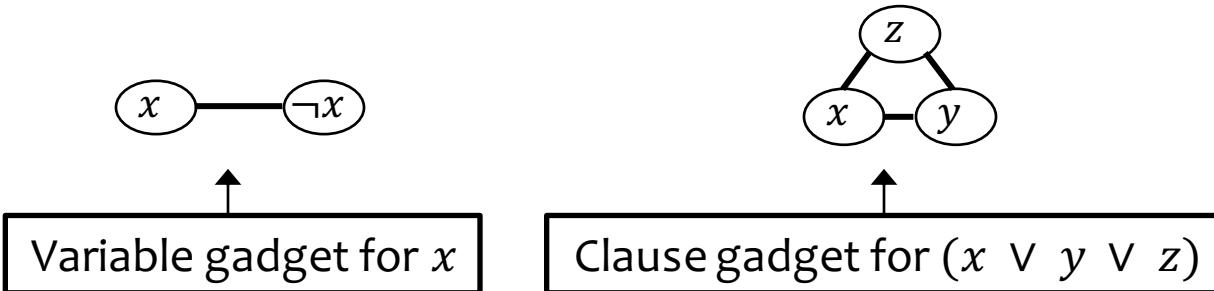
**Exercise 175** Show that VERTEX-COVER  $\in$  NP.

**Lemma 176** VERTEX-COVER is NP-hard.

Before giving the formal proof, we first give the key ideas and an illustrative example. We will demonstrate a polynomial-time mapping reduction from the NP-hard language 3SAT: given a 3CNF formula, the reduction constructs a graph and a budget that corresponds to the formula in a certain way, so that the formula is satisfiable if and only if the graph has a vertex cover within the size budget.

The graph is made up of “gadget” subgraphs, one for each variable and clause in the formula. These are connected together with appropriate edges according to the contents of the formula.

A gadget for a variable  $x$  is a “barbell,” consisting of two vertices respectively labeled  $x$  and  $\neg x$ , and connected by an edge. A gadget for a clause is a triangle whose three vertices are each labeled by a corresponding one of the literals. Example variable and clause gadgets are depicted here:

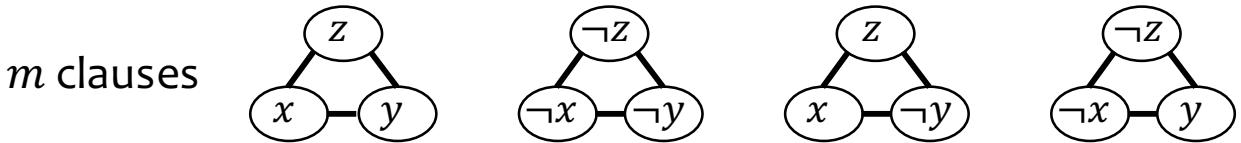


The reduction first creates one variable gadget for each variable, and one clause gadget for each clause. Then, it connects these gadgets using edges as follows: it adds an edge between each clause-gadget vertex and the (unique) variable-gadget vertex having the *same* literal label.

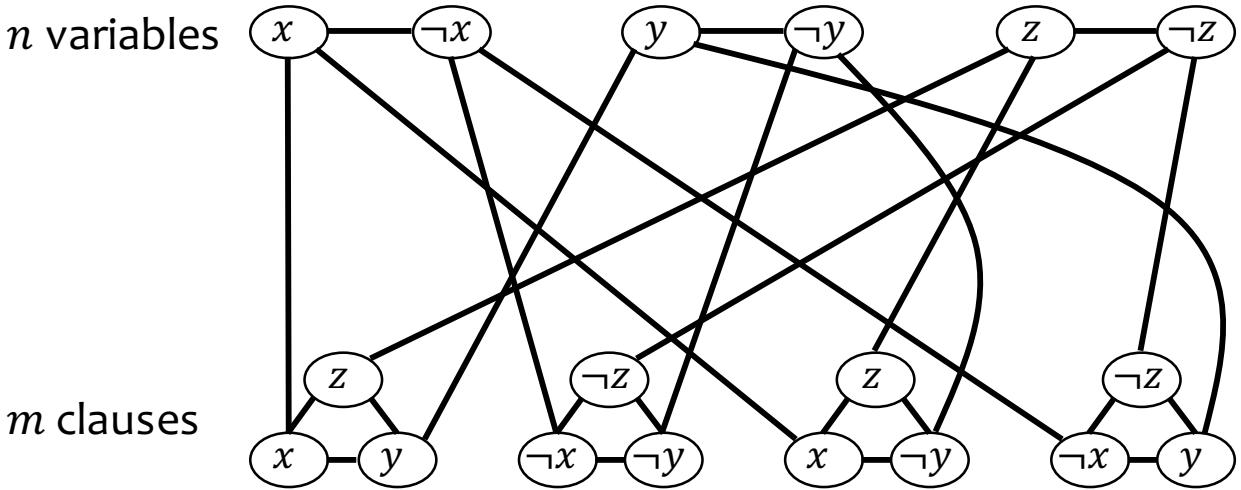
As a concrete example, consider the input formula

$$\phi = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z).$$

To construct the corresponding graph, we start with a gadget for each variable and clause:



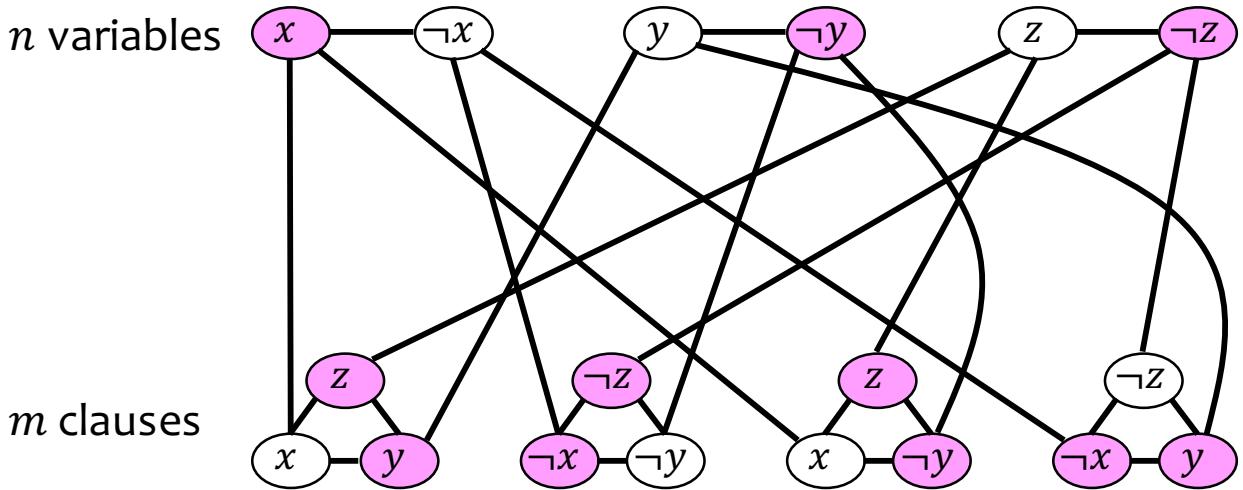
Then, we connect each clause-gadget vertex to the variable-gadget vertex having the same literal label:



Observe that *any* vertex cover of the graph must cover each variable-gadget edge, so it must have at least one vertex from each variable gadget. Similarly, it must cover all three edges in each clause gadget, so it must have at least *two* vertices from each clause gadget. All this holds even ignoring the “crossing” edges that go between the variable and clause gadgets.

With this in mind, the reduction finally sets the budget to be  $k = n + 2m$ , where  $n$  is the number of variables and  $m$  is the number of clauses in the input formula. This corresponds to asking whether there is a vertex cover having *no “additional” vertices* beyond what is required by just the gadgets themselves. As the following proof shows, it turns out that there is a vertex cover of this size if and only if the input formula is satisfiable: any satisfying assignment of the formula yields a corresponding vertex cover of size  $k$ , and vice versa.

For example, for the above example formula and corresponding graph, setting  $x = \text{true}$  and  $y = z = \text{false}$  satisfies the formula, and the graph has a corresponding vertex cover as depicted here:



Conversely, we can show that for *any* vertex cover of size  $k = n + 2m$  in the graph, there is a corresponding satisfying assignment for  $\phi$ , obtained by setting the variables so that the literals of all the variable-gadget vertices in the cover are true.

**Proof 177** We give a polynomial-time mapping reduction from 3SAT to VERTEX-COVER, showing that  $\text{3SAT} \leq_p \text{VERTEX-COVER}$ . Since 3SAT is NP-hard, it follows that VERTEX-COVER is NP-hard as well.

We need to define a polynomial-time computable function  $f$  that transforms a given 3CNF formula  $\phi$  into a corresponding instance  $f(\phi) = (G, k)$  of the vertex-cover problem, so that  $\phi \in \text{3SAT} \iff (G, k) \in \text{VERTEX-COVER}$ . The function  $f$  constructs  $G$  and  $k$  as follows:

- For each variable  $x$  in  $\phi$ , it creates a “variable gadget” consisting of two vertices, respectively labeled by the literals  $x$  and  $\neg x$ , with an edge between them.
- For each clause  $(\ell_i \vee \ell_j \vee \ell_k)$  in  $\phi$ , where  $\ell_i, \ell_j, \ell_k$  are literals, it creates a “clause gadget” consisting of three vertices, respectively labeled by these literals, with an edge between each pair of these vertices (i.e., a triangle).
- For each vertex of each clause gadget, it creates an edge between that vertex and the (unique) variable-gadget vertex having the same label. That is, a clause-gadget vertex labeled by literal  $x$  (respectively,  $\neg x$ ) has an edge to the variable-gadget vertex labeled  $x$  (respectively,  $\neg x$ ).
- Finally, set  $k = n + 2m$  where  $n, m$  are respectively the number of variables and clauses in  $\phi$ .

To see that the reduction is efficient, observe that there are  $2n + 3m$  vertices in  $G$ , with  $n$  edges within the variable gadgets,  $3m$  edges within the clause gadgets, and  $3m$  edges that cross between the clause and variable gadgets, for a total of  $O(m + n)$  edges. Thus,  $G$  has size  $O(n + m)$ , and it (along with  $k$ ) can be constructed efficiently by iterating over the input formula  $\phi$ .

We now show that the reduction is correct. We need to show that  $\phi \in \text{3SAT} \iff f(\phi) = (G, n + 2m) \in \text{VERTEX-COVER}$ .

We first show that  $\phi \in \text{3SAT} \implies f(\phi) = (G, n + 2m) \in \text{VERTEX-COVER}$ . In other words, if  $\phi$  is satisfiable, then  $G$  has a vertex cover of size  $n + 2m$ . Since  $\phi$  is satisfiable, there is some satisfying assignment  $\alpha$  under which each clause in  $\phi$  has at least one literal that is true. We show by construction that there is a corresponding vertex cover  $C$  of size  $n + 2m$  in  $G$ . We define  $C$  as follows:

- For each variable gadget, include in  $C$  the vertex labeled by the literal that is *true* under  $\alpha$ . That is, for each variable  $x$ ’s gadget, place the vertex labeled by  $x$  in  $C$  if  $\alpha$  assigns  $x$  to be true, otherwise place the vertex labeled  $\neg x$  in  $C$ .

Observe that exactly one vertex from each of the variable gadgets is in  $C$ , and these  $n$  vertices cover all  $n$  of the variable-gadget edges.

- In each clause gadget, observe that at least one of its three vertices is labeled by a literal that is *true* under  $\alpha$ , because  $\alpha$  satisfies all the clauses in  $\phi$ . Identifying an arbitrary one of these vertices per clause gadget, include in  $C$  the *other two* vertices of each gadget.

Observe that  $C$  has  $2m$  clause-gadget vertices, and these cover all  $3m$  of the clause-gadget edges.

As we have just seen,  $C$  has  $n + 2m$  vertices in total, and covers all the edges that are “internal” to the gadgets. So, it just remains to show that  $C$  also covers all the “crossing” edges that go between the variable gadgets and clause gadgets.

To see this, recall that the crossing edges are those that go from each clause-gadget vertex  $v$  to the variable-gadget vertex having the same literal label  $\ell$ . If  $v \in C$ , then its crossing edge is covered by  $C$ , so now suppose that  $v \notin C$ . Then by construction of  $C$ ,  $v$ ’s literal  $\ell$  is *true* under the assignment  $\alpha$ . Therefore, the *variable-gadget* vertex labeled  $\ell$  is in  $C$ , and hence  $v$ ’s crossing edge is covered by  $C$ . Because this reasoning holds for every clause-gadget vertex, *all* the crossing edges in  $G$  are covered by  $C$ . We conclude that  $C$  is a vertex cover of  $G$ , hence  $(G, k) \in \text{VERTEX-COVER}$ , as needed.

We now show the converse direction, that  $(G, n + 2m) \in \text{VERTEX-COVER} \implies \phi \in \text{3SAT}$ . In other words, if  $G$  has a vertex cover  $C$  of size at most  $n + 2m$ , then  $\phi$  has a satisfying assignment. Indeed, we will show by construction that for any such  $C$ , there is a corresponding satisfying assignment for  $\phi$ .

As observed above, *any* vertex cover of  $G$  must have at least one vertex from each variable gadget (to cover that gadget’s edge), and at least two vertices from each clause gadget (to cover that gadget’s three edges). So, since  $C$  is a vertex cover of size at most  $n + 2m$ , it must have size *exactly*  $n + 2m$ , and therefore  $C$  has:

- exactly one vertex from each variable gadget, and
- exactly two vertices from each clause gadget.

Based on this, we define an assignment for  $\phi$  as follows: for each variable  $x$ ’s gadget, identify which one of its vertices (labeled either  $x$  or  $\neg x$ ) is in  $C$ , and set the value of  $x$  so that the label of this vertex is true. That is, if the variable-gadget vertex labeled  $x$  is in  $C$ , set  $x$  to be true; otherwise, set  $x$  to be false. (Note that this assignment is well defined because  $C$  has exactly one vertex from each variable gadget.)

We show that this assignment satisfies  $\phi$ , using the fact that  $C$  is a vertex cover. Consider an arbitrary clause in  $\phi$ ; we claim that the assignment satisfies it (and hence the entire formula) because the clause has at least one true literal.

In the gadget corresponding to the clause, there is exactly one vertex  $v$  that is not in  $C$ , which is labeled by some literal  $\ell$  that appears in the clause. Recall that the other endpoint of  $v$ ’s “crossing edge” is the variable-gadget vertex having label  $\ell$ . Since  $C$  covers this crossing edge, and  $v \notin C$ , this other endpoint must be in  $C$ . So by definition,  $\ell$  is set to true under the assignment. Finally, since  $\ell$  is one of the literals in the clause in question, the assignment satisfies the clause, as claimed. This completes the proof.  $\square$

**Exercise 178** a) An *independent set* of an undirected graph  $G = (V, E)$  is a subset of the vertices  $I \subseteq V$  such that no two vertices in  $I$  have an edge between them. Define the language (with non-negative threshold  $k \leq |V|$ )

$$\text{INDEPENDENT-SET} = \{(G, k) : G \text{ is an undirected graph with an independent set of size at least } k\}.$$

Show that  $\text{VERTEX-COVER} \leq_p \text{INDEPENDENT-SET}$ .

**Hint:** If  $G$  has a vertex cover of size  $m$ , what can be said about the vertices that are not in the cover?

b) Recall the language

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph that has a clique of size at least } k\}.$$

Show that INDEPENDENT-SET  $\leq_p$  CLIQUE.

## 18.4 Set Cover

We now consider another problem, that of hiring workers for a project. To complete the project, we need workers that collectively have some set of skills. For instance, to build a house, we might need an architect, a general contractor, an engineer, an electrician, a plumber, a painter, and so on. There is a pool of candidate workers, where each worker has a certain set of skills. For example, there might be a worker who is proficient in plumbing and electrical work, another who can put up drywall and paint, etc. Our goal is to hire a team of workers of *minimum size* that “covers” all the required skills, i.e., for each skill, at least one hired worker has that skill.

We formalize this problem as follows. We are given an instance that consists of a set  $S$ , representing the required skills, along with some subsets  $S_i \subseteq S$ , representing the set of skills that each worker has. We wish to select as few of the  $S_i$  as possible to cover the set  $S$ . In other words, we want to find a smallest set of indices  $C$  such that

$$\bigcup_{i \in C} S_i = S.$$

As a concrete example, consider the following small instance:

$$\begin{aligned} S &= \{1, 2, 3, 4, 5, 6, 7\} \\ S_1 &= \{1, 2, 3\} \\ S_2 &= \{3, 4, 6, 7\} \\ S_3 &= \{1, 4, 7\} \\ S_4 &= \{1, 2, 6\} \\ S_5 &= \{3, 5, 7\} \\ S_6 &= \{4, 5\}. \end{aligned}$$

There is no cover of size two: in order to cover skill 5, we must select subset  $S_5$  or  $S_6$ , and no single subset covers all the remaining skills. However, there are several covers of size three. One example is  $C = \{1, 2, 6\}$ , which gives us

$$\begin{aligned} \bigcup_{i \in C} S_i &= S_1 \cup S_2 \cup S_6 \\ &= \{1, 2, 3\} \cup \{3, 4, 6, 7\} \cup \{4, 5\} \\ &= S. \end{aligned}$$

We now define a language that corresponds to the decision version of the set-cover problem. As with vertex cover, we include a budget  $k$  for the size of the cover, which does not exceed the number of given subsets.

$$\text{SET-COVER} = \{(S, S_1, S_2, \dots, S_n, k) : \exists C \subseteq \{1, \dots, n\} \text{ of size (at most) } k \text{ s.t. } \bigcup_{i \in C} S_i = S\}.$$

It is straightforward to show that  $\text{SET-COVER} \in \text{NP}$ : a certificate is a set of indices  $C \subseteq \{1, \dots, n\}$ , and the verifier simply checks that  $|C| \leq k$  and that  $\bigcup_{i \in C} S_i = S$ , which can be done efficiently.

The following lemma shows that, like 3SAT and VERTEX-COVER, SET-COVER is a “hardest” problem in NP.

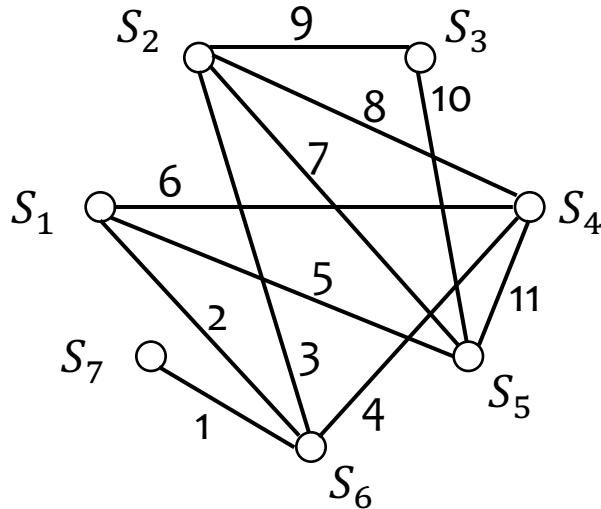
**Lemma 179** *SET-COVER is NP-hard.*

Before providing the formal proof, we give the key ideas and an illustrative example. First, we choose a suitable NP-hard language to reduce from. While there are many choices that could work, it is preferable to choose a language that is “defined similarly” to the target language, because this typically makes the reduction and analysis simpler and less error prone.

For this reason, we will demonstrate a polynomial-time mapping reduction from the NP-hard language VERTEX-COVER. Given a VERTEX-COVER instance  $(G = (V, E), k)$ , the reduction needs to construct an SET-COVER instance  $(S, S_1, \dots, S_n, k')$ —i.e., a set  $S$ , subsets  $S_i \subseteq S$ , and budget  $k'$ —so that  $G$  has a vertex cover of size (at most)  $k$  if and only if  $S$  can be covered by (at most)  $k'$  of the subsets.

To help conceive of such a reduction, it is first useful to identify the specific similarities between the VERTEX-COVER and SET-COVER problems. The VERTEX-COVER problem asks to cover all the edges of a given graph by some limited number of its vertices budget, where each vertex covers its incident edges. The SET-COVER problem asks to cover all the elements of a set (e.g., of skills) by a limited number of some given subsets (e.g., workers). So, a natural idea is for the reduction to generate a set that corresponds to the edges of the graph (these are the objects to be covered), and subsets that correspond to the vertices—specifically, each subset is the set of edges incident to its corresponding vertex. Because the vertices are in correspondence with the subsets, the reduction leaves the budget unchanged.

As an example, consider the following graph, where for convenience we have numbered the vertices and edges.



This graph has a vertex cover of size four, consisting of the vertices labeled with indices 2, 4, 5, 6.

On this graph, the reduction outputs the following sets:

$$\begin{aligned}
 S &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} \\
 S_1 &= \{2, 5, 6\} \\
 S_2 &= \{3, 7, 8, 9\} \\
 S_3 &= \{9, 10\} \\
 S_4 &= \{4, 6, 8, 11\} \\
 S_5 &= \{5, 7, 10, 11\} \\
 S_6 &= \{1, 2, 3, 4\} \\
 S_7 &= \{1\} .
 \end{aligned}$$

Observe that  $C = \{2, 4, 5, 6\}$  is a set cover of size four, and corresponds to the vertex cover mentioned above. This holds more generally: any vertex cover of the graph corresponds to a set cover, by taking the subsets that correspond to the selected vertices. And in the opposite direction, for any selection of subsets that covers  $S$ , the corresponding vertices cover all the edges of the graph.

**Proof 180** We prove that VERTEX-COVER  $\leq_p$  SET-COVER, and hence SET-COVER is NP-hard.

We need to define a polynomial-time computable function  $f$  that transforms a given VERTEX-COVER instance  $(G, k)$  into a corresponding instance  $f(G, k) = (S, S_1, \dots, S_n, k)$  of SET-COVER, so that  $(G, k) \in \text{VERTEX-COVER} \iff f(G, k) \in \text{SET-COVER}$ .

Following the above ideas, the function outputs the set of edges, and for each vertex, a subset consisting of the edges incident to that vertex. Formally, we define  $f(G = (V, E), k) = (S, S_v \text{ for every } v \in V, k)$ , where  $S = E$  and

$$S_v = \{e \in E : e \text{ is incident to } v, \text{i.e., } e = (u, v) \text{ for some } u \in V\}.$$

Observe that this function can be computed efficiently, since for each vertex it just looks up all the edges incident to that vertex, which can be done by a naïve algorithm in  $O(|V| \cdot |E|)$  time.

We now show that the reduction is correct. We need to show that  $(G = (V, E), k) \in \text{VERTEX-COVER} \iff f(G, k) = (S, S_v \text{ for all } v \in V, k) \in \text{SET-COVER}$ .

For this purpose, the key property of the reduction is that a subset of vertices  $C \subseteq V$  is a vertex cover of  $G$  if and only if the subsets  $S_v$  for  $v \in C$  cover  $S$ . This is because the set of edges covered by the vertices in  $C$  is

$$\bigcup_{v \in C} \{e \in E : e \text{ is incident to } v\} = \bigcup_{v \in C} S_v.$$

Since  $E = S$ , the left-hand side equals  $E$  (i.e.,  $C$  is a vertex cover of  $G$ ) if and only if the right-hand side equals  $S$  (i.e., the subsets  $S_v$  for  $v \in C$  cover  $S$ ).

Correctness then follows immediately from this key property:  $G$  has a vertex cover of size (at most)  $k$  if and only if some (at most)  $k$  of the subsets  $S_v$  cover  $S$ .  $\square$

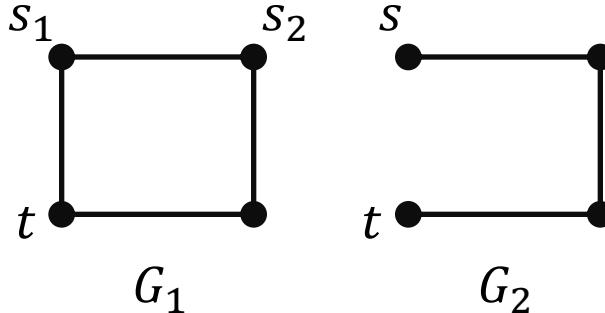
We conclude our discussion by observing that by the above reduction, we can see VERTEX-COVER as a natural special case of SET-COVER. Specifically, using the subsets of incident edges means that every element of  $S = E$  belongs to *exactly two* of the subsets  $S_v$  (namely, the ones corresponding to the two endpoints of the edge). This is a special case of set cover; in general, any element is allowed to be in any number (including zero) of the given subsets. An algorithm for SET-COVER must be correct on all instances, so in particular it must be correct on these special cases; combined with the reduction, this would also yield a correct algorithm for VERTEX-COVER.<sup>63</sup> (This phenomenon is similar to what we observed about the special kinds of graphs constructed in the above reduction from 3SAT to VERTEX-COVER.)

## 18.5 Hamiltonian Cycle

In a graph  $G$ , a *Hamiltonian path* from vertex  $s$  to vertex  $t$  is a path that starts at  $s$ , ends at  $t$ , and visits each vertex exactly once. A *Hamiltonian cycle* is a cycle that visits every vertex exactly once. The graph may be directed or undirected; we focus primarily on the undirected case, but everything we do here easily extends to the directed case as well.

For example,  $G_1$  below has a Hamiltonian cycle, as well as a Hamiltonian path from  $s_1$  to  $t$ , but it does not have a Hamiltonian path from  $s_2$  to  $t$ . On the right,  $G_2$  has a Hamiltonian path from  $s$  to  $t$ , but it does not have a Hamiltonian cycle.

<sup>63</sup> Because both VERTEX-COVER and SET-COVER are NP-complete, they can each be Karp-reduced to each other. However, the reduction from SET-COVER to VERTEX-COVER is not as straightforward as the reverse direction, because not every SET-COVER instance corresponds to a VERTEX-COVER instance in a “natural, simple” way. In particular, the reduction  $f$  given above is not surjective, so it is not invertible on an arbitrary SET-COVER instance.



Two natural decision problems are whether a given graph has a Hamiltonian cycle, and whether it has a Hamiltonian path from one given vertex to another. The associated languages are defined as follows:

$$\begin{aligned}\text{HAMPATH} &= \{(G, s, t) : G \text{ is a graph with a Hamiltonian path from } s \text{ to } t\} \\ \text{HAMCYCLE} &= \{G : G \text{ is a graph with a Hamiltonian cycle}\}.\end{aligned}$$

Both languages are in NP: a certificate is a claimed Hamiltonian path or cycle (as appropriate) in the graph, and the verifier merely needs to check whether it meets the required conditions (in particular, that it visits every vertex exactly once), which can be done in polynomial time in the size of the instance.

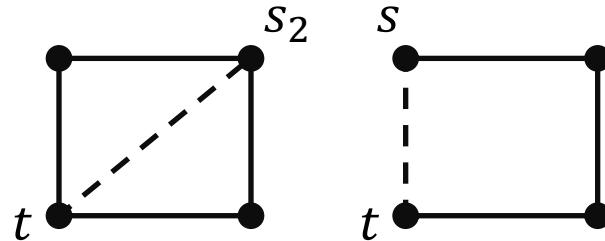
We claim without proof that HAMPATH is NP-hard; it is possible to reduce 3SAT to HAMPATH, but the reduction is quite complicated, so we will not consider it here. Instead, we prove the following.

**Lemma 181**  $\text{HAMPATH} \leq_p \text{HAMCYCLE}$ , and hence HAMCYCLE is NP-hard.

Before giving the formal proof, we explore an “almost correct” attempt, which motivates a fix to yield a correct reduction.

We need to demonstrate an efficient transformation that maps an arbitrary HAMPATH instance  $(G, s, t)$  to a HAMCYCLE instance  $G'$ , such that  $G$  has a Hamiltonian path from  $s$  to  $t$  if and only if  $G'$  has a Hamiltonian cycle.

As a first natural attempt, we consider a reduction that constructs a new graph  $G'$  that is equal to  $G$ , but with an additional edge between  $s$  and  $t$ , if there isn’t already one in  $G$ . It is easy to see that if  $G$  has a Hamiltonian path from  $s$  to  $t$ , then  $G'$  has a Hamiltonian cycle, which simply follows a Hamiltonian path from  $s$  to  $t$ , then returns to  $s$  via the  $(s, t)$  edge. So, this reduction maps any “yes” instance of HAMPATH to a “yes” instance of HAMCYCLE. Unfortunately, it does not map every “no” instance of HAMPATH to a “no” instance of HAMCYCLE, so the reduction is not correct. The following illustrates the reduction on example “no” and “yes” instances of HAMPATH, with the dotted line representing the added edge:

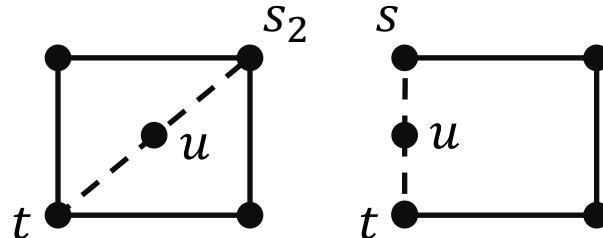


In the original graph on the left, there is no Hamiltonian path from  $s_2$  to  $t$ , yet the new graph does have a Hamiltonian cycle, which just uses the original edges.

The key problem with this reduction attempt is that it does not “force” a Hamiltonian cycle to use the  $(s, t)$  edge. (Indeed, this is the case for the above counterexample.) However, if a Hamiltonian cycle *does* use the  $(s, t)$  edge, then we could drop that edge from the cycle to obtain a Hamiltonian path in the original graph. (This works regardless of

whether that edge is in the original graph.) So, if the reduction can somehow ensure that any Hamiltonian cycle in the constructed graph must use specific added edge(s) between  $s$  and  $t$ , the basic approach behind the first attempt can be made to work.

To do this, we define a reduction that adds a new *vertex*, with edges to both  $s$  and  $t$ , as in the following:



Here the new graph on the left does not have a Hamiltonian cycle, and the new graph on the right does, which are correct outcomes in these cases. More generally, if there is a Hamiltonian cycle in the constructed graph, then it must visit the new vertex exactly once, so it must use the two new edges consecutively. Remove these two edges from the cycle yields a Hamiltonian path from  $s$  to  $t$ , as needed. We now proceed formally.

**Proof 182** We need to define a polynomial-time computable function  $f$  that transforms a given HAMPATH instance  $(G, s, t)$  into a corresponding instance  $G' = f(G, s, t)$  of HAMCYCLE, so that  $(G, s, t) \in \text{HAMPATH} \iff G' = f(G, s, t) \in \text{HC}$ .

On input  $(G = (V, E), s, t)$ , the reduction creates a new vertex  $u \notin V$  and outputs the graph

$$G' = f(G, s, t) = (V' = V \cup \{u\}, E' = E \cup \{(s, u), (t, u)\}).$$

The function is clearly efficient, since it merely adds one new vertex and two edges to the input graph.

We now show that the reduction is correct. We first show that  $(G, s, t) \in \text{HAMPATH} \implies G' = f(G, s, t) \in \text{HAMCYCLE}$ . By hypothesis, there exists a Hamiltonian path  $P$  from  $s$  to  $t$  in  $G$ . Then  $G' \in \text{HAMCYCLE}$  because there is a corresponding Hamiltonian cycle in  $G'$ : it starts at  $s$ , goes to  $t$  via path  $P$  (which is a path in  $G'$ ), then takes the new edges from  $t$  to  $u$ , and  $u$  back to  $s$ . This is a clearly a cycle in  $G'$ , and it is Hamiltonian because  $P$  visits every vertex in  $V$  exactly once, and the two new edges merely visit  $u$  and return to the start, so every vertex in  $V' = V \cup \{u\}$  is visited exactly once.

Now we show that  $G' = f(G, s, t) \in \text{HAMCYCLE} \implies (G, s, t) \in \text{HAMPATH}$ . By hypothesis, there is a Hamiltonian cycle  $C$  in  $G'$ ; we show that there is a corresponding Hamiltonian path from  $s$  to  $t$  in  $G$ . Since  $C$  is a cycle that visits every vertex exactly once, we can view it as starting and ending at any vertex we like, in particular the new vertex  $u$ . Because by construction  $u$ 's only edges are the two new ones to  $s$  and  $t$ , we can view the cycle  $C$  as starting with edge  $(u, s)$  and ending with edge  $(t, u)$ . Removing these edges from the cycle, what remains is a path from  $s$  to  $t$ , and it is Hamiltonian in  $G$  because  $C$  visits every vertex in  $V' = V \cup \{u\}$  exactly once, hence removing the new edges visits every vertex in  $V$  exactly once.  $\square$

**Exercise 183** Modify the above reduction and proof to work for *directed* graphs.

**Exercise 184** Define the language

$$\text{LONG-PATH} = \{(G, k) : G \text{ is an undirected graph with a simple path of length } k\}$$

A *simple path* is a path without any cycles. Show that LONG-PATH is NP-complete.

**Exercise 185** Recall the traveling salesperson language:

$$\text{TSP} = \{(G, k) : G \text{ is a weighted graph that has a tour of total weight at most } k\}.$$

Show that TSP is NP-hard by proving that HAMCYCLE  $\leq_p$  TSP.

## 18.6 Concluding Remarks

We have explored only the tip of the iceberg when it comes to NP-complete problems. Such problems are everywhere, including constraint satisfaction (SAT, 3SAT), covering problems (vertex cover, set cover), resource allocation (knapsack, subset sum), scheduling, graph coloring, model checking, social networks (clique, maximum cut), routing (HAMPATH, TSP), games (Sudoku, Battleship, Super Mario Brothers, Pokémon), and so on. An efficient algorithm for any one of these problems yields efficient algorithms for all of them. So, either all of them are efficiently solvable, or none of them are.

The skills to reason about a problem and determine whether it is NP-hard are critical. Researchers have been working for decades to find an efficient algorithm for an NP-complete problem, to no avail. This means it would require an enormous breakthrough to do so, and may very well be impossible. Instead, when we encounter such a problem, a better path is to focus on *approximation* algorithms, which we turn to next.

## SEARCH PROBLEMS AND SEARCH-TO-DECISION REDUCTIONS

Thus far, we have focused our attention on *decision* problems, formalized as *languages*. We now turn to *search* (or *functional*) problems—those that may ask for more than just a yes/no answer. Most algorithmic problems we encounter in practice are typically phrased as search problems. Some examples of (computable) search problems include:<sup>64</sup>

- Given an array, sort it.
- Given two strings, find a *largest common subsequence* of the strings.
- Given a weighted graph and starting and ending vertices, find a *shortest path* from the start to the end.
- Given a Boolean formula, find a *satisfying assignment*, if one exists.
- Given a graph, find a *Hamiltonian cycle* in the graph, if one exists.
- Given a graph, find a *largest clique* in the graph.
- Given a graph, find a *smallest vertex cover* in the graph.

In general, a search problem may be:

- an *exact* problem, such as finding a satisfying assignment or a Hamiltonian path;
- a *minimization* problem, such as finding a shortest path or a minimum vertex cover;
- a *maximization* problem, such as finding a largest common subsequence or a maximum clique.

Minimization and maximization problems are also called *optimization problems*.

Recall that for each kind of search problem, we can define a corresponding decision problem.<sup>65</sup> For an exact problem, the corresponding decision problem is whether there *exists* a solution that meets the required criteria. The following is an example:

$$\text{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\} .$$

For minimization and maximization problems, we introduce a budget or threshold. Recall the following examples:

$$\begin{aligned}\text{CLIQUE} &= \{(G, k) : G \text{ is an undirected graph that has a clique of size (at least) } k\} \\ \text{VERTEX-COVER} &= \{(G, k) : G \text{ is an undirected graph that has a vertex cover of size (at most) } k\} .\end{aligned}$$

We have seen that the languages SAT, CLIQUE, and VERTEX-COVER are NP-complete. How do the search problems compare in difficulty to their respective decision problems?

First, given an oracle (or efficient algorithm) that solves a search problem, we can efficiently decide the corresponding language. In other words, there is an *efficient Turing reduction* from the decision problem to the search problem, so the decision problem is “no harder than” the search problem. For instance, given an oracle that finds a satisfying assignment

<sup>64</sup> *Kolmogorov complexity* (page 267) is an example of an *uncomputable* functional problem.

<sup>65</sup> We previously saw that a functional problem has an *equivalent formulation as a decision problem* (page 267). However, this correspondence was based on translating a functional problem to a sequence of decision problems on the binary representation of the answer. This is different than what we are considering now, which is between a search problem and a “natural” formulation of a corresponding decision problem.

(if one exists) for a given Boolean formula, we can decide the language SAT simply by calling the oracle on the input formula, accepting if it returns a satisfying assignment, and rejecting otherwise. As another example, given an oracle that finds a largest clique in a given graph, we can decide the language CLIQUE as follows: on input a graph  $G$  and a threshold  $k$ , simply call the oracle on  $G$  and check whether the size of the returned clique (which is a largest clique in  $G$ ) is at least  $k$ .

What about the other direction, of efficiently reducing a search problem to its decision version? For example, if we have an oracle (or efficient algorithm)  $D$  that decides the language CLIQUE, can we use it to efficiently *find a clique of maximum size* in a given graph? In other words, is there an efficient Turing reduction from the search version of the max-clique problem to the decision version? It turns out that the answer is yes, though proving this is more involved than in the other direction.

The first step is to find the size  $k^*$  of a largest clique in the input graph  $G = (V, E)$ . We do this by calling  $D(G, k)$  for each value of  $k$  from  $|V|$  down to 1, stopping at the first value  $k^*$  for which  $D$  accepts.

**Input:** an undirected graph

**Output:** the size of (number of vertices in) a largest clique in the graph

```
function SIZEMAXCLIQUE( $G = (V, E)$ )
    for  $k = |V|$  down to 1 do
        if  $D(G, k)$  accepts then return  $k^* = k$ 
```

Since a clique in  $G$  is a subset of the vertex set  $V$ , the size  $k^*$  of a largest clique is between 1 and  $|V|$ , inclusive. So,  $D(G, k)$  must accept for at least one value of  $k$  in that range. By design, SIZEMAXCLIQUE outputs the largest such  $k$ , so it is correct. The algorithm is also efficient, since it makes as most  $|V|$  calls to the oracle (or efficient algorithm)  $D$ .

Note that in general for other optimization problems, a linear search to find the optimal size or value *might not be efficient*; it depends on how the number of possible values relates to the input size. For the max-clique problem, the size of the search space is linear in the input size—there are  $|V|$  possible answers—so a linear search suffices. For other problems, the size of the search space might be exponential in the input size, in which case we should use a binary search.

Once we have determined the size of a largest clique, we can then use the oracle to find a clique of that size. A strategy that works for many problems is a “destructive” one, where we use the oracle to determine which pieces of the input to remove until all we have left is an object we seek. To find a clique of maximum size, we consider each vertex, temporarily discarding it and all its incident edges, and query the oracle to see if the reduced graph still has a clique of the required size. If so, the vertex is unnecessary—there is a largest clique that does not use the vertex—so we remove it permanently. Otherwise, the discarded vertex is part of every largest clique, so we undo its temporary removal. We perform this check for all the vertices in sequence. By a careful (and non-obvious) argument, it can be proved that at the end, we are left with just a largest clique in the original graph, and no extraneous vertices. The formal pseudocode and analysis is as follows.

**Input:** an undirected graph  $G$  and the size  $k^*$  of a largest clique in  $G$

**Output:** a largest clique in  $G$

```
function FINDMAXCLIQUE( $G = (V, E), k^*$ )
    for all  $v \in V$  do
        let  $G'$  be  $G$ , but with  $v$  and all its incident edges removed
        if  $D(G', k^*)$  accepts then
             $G = G'$ 
    return  $C = V(G)$ , the set of (remaining) vertices in  $G$ 
```

It is straightforward to see that the algorithm is efficient: it loops over each vertex exactly once, temporarily removing it and its incident edges from the graph, which can be done efficiently, and then invoking the oracle (or efficient algorithm).

**Lemma 186** *The above FINDMAXCLIQUE algorithm is correct, i.e., it outputs a clique of size  $k^*$ , the maximum clique size in the input graph (by hypothesis).*

**Proof 187** We first claim that the algorithm outputs a set of vertices  $C \subseteq V$  of the original graph  $G$  that contains a clique of size  $k^*$ . To see this, observe that the algorithm maintains the invariant that  $G$  has a clique of size  $k^*$  at all times. This holds for the original input by hypothesis, and the algorithm changes  $G$  only when it sets  $G = G'$  where  $D(G', k^*)$  accepts, which means that  $G'$  has a clique of size  $k^*$  (by hypothesis on  $D$ ). So, the claimed invariant is maintained, and therefore the output set of vertices  $C = V(G)$  contains a clique of size  $k^*$ .

Now we prove the lemma statement, that the output set  $C$  is a clique of size  $k^*$ , with no additional vertices. By the above argument,  $C$  contains a clique  $K \subseteq C$  of size  $k^*$ . We now show that  $C$  has no additional vertices beyond those in  $K$ , i.e.,  $C = K$ , which proves the lemma. Let  $v \in V \setminus K$  be an arbitrary vertex of the original graph that is not in  $K$ . At some point, the algorithm temporarily removed  $v$  from the graph to get some  $G'$ . Since  $K \subseteq C$  remained in the graph throughout the entire execution (except for temporary removals) and  $v \notin K$ , the call to  $D(G', k^*)$  accepted, and hence  $v$  was permanently removed from the graph. So, we have shown that every vertex not in  $K$  was permanently removed when it was considered, and therefore  $C = K$ , as claimed.  $\square$

There are also search-to-decision reductions for *minimization* problems; here we give one for the vertex-cover problem. Suppose we have an oracle (or efficient algorithm)  $D$  that decides the language VERTEX-COVER. We can use it to efficiently find a minimum vertex cover as follows. Given an input graph  $G = (V, E)$ , we do the following:

- First, find the size  $k^*$  of a smallest vertex cover by calling  $D(G, k)$  for each  $k$  from 0 to  $|V|$ , stopping at the first value  $k^*$  for which  $D$  accepts. So,  $k^*$  is then the size of a smallest vertex cover.
- For each vertex  $v \in V$ , temporarily remove it and all its incident edges from the graph, and ask  $D$  whether this “reduced” graph has a vertex cover of size at most  $k^* - 1$ . If so, recursively find such a vertex cover in the reduced graph, and add  $v$  to it to get a size- $k^*$  vertex cover of  $G$ , as desired. Otherwise, restore  $v$  and its edges to the graph, and continue to the next vertex.

The full search algorithm and analysis is as follows:

**Input:** an undirected graph  $G$

**Output:** the size of a smallest vertex cover of  $G$

```
function SIZEMINVERTEXCOVER( $G$ )
     $k = 0$ 
    while  $D(G, k)$  rejects do
         $k = k + 1$ 
    return  $k^* = k$ 
```

**Input:** an undirected graph  $G$  and the size  $k^*$  of a smallest vertex cover of  $G$

**Output:** a smallest vertex cover of  $G$

```
function FINDMINVERTEXCOVER( $G = (V, E), k^*$ )
    if  $k^* = 0$  then
        return  $\emptyset$                                 ▷ base case
    for all  $v \in V$  do
        let  $G'$  be  $G$ , but with  $v$  and all its incident edges removed
        if  $D(G', k^* - 1)$  accepts then
            return  $\{v\} \cup \text{FINDMINVERTEXCOVER}(G', k^* - 1)$ 
```

SIZEMINVERTEXCOVER is correct because the size of a smallest vertex cover in  $G = (V, E)$  is between 0 and  $|V|$  (inclusive), and it is the smallest  $k$  in this range for which  $D(G, k)$  accepts, by hypothesis on  $D$ . Also, this algorithm is efficient because it calls the oracle (or efficient algorithm)  $D$  at most  $|V| + 1$  times, which is linear in the input size.

**Lemma 188** *FINDMINVERTEXCOVER is correct and efficient.*

**Proof 189** First we show correctness. The base case  $k^* = 0$  is clearly correct, because by the input precondition,  $G$  has a vertex cover of size zero, namely,  $\emptyset$ .

Now we consider the recursive case on input  $(G = (V, E), k^* > 0)$ . By the input precondition,  $k^*$  is the size of a smallest vertex cover of  $G$  (of which there may be more than one). The algorithm considers each vertex  $v \in V$  in turn, defining  $G'$  to be  $G$  with  $v$  and all its incident edges removed.

First observe that for any vertex cover of  $G'$ , adding  $v$  to it yields a vertex cover of  $G$ , because  $v$  covers every edge of  $G$  that is not in  $G'$ . So, every vertex cover of  $G'$  has size *at least*  $k^* - 1$  (because every vertex cover of  $G$  has size at least  $k^*$ ). Therefore, by hypothesis on  $D$ , if  $D(G', k^* - 1)$  accepts, then the smallest vertex cover in  $G'$  has size *exactly*  $k^* - 1$ . So the precondition of the recursive call is satisfied, and by induction/recursion, the recursive call returns a size- $(k^* - 1)$  vertex cover  $C'$  of  $G'$ . Then  $\{v\} \cup C'$  is a size- $k^*$  vertex cover of  $G$ , so the final output is correct.

It just remains to show that  $D(G', k^* - 1)$  accepts in some iteration. By hypothesis,  $G$  has at least one size- $k^*$  vertex cover; let  $v$  be any vertex in any such cover  $C$ . When the algorithm considers  $v$ , because  $v$  does not cover any edge of  $G'$ , it must be that  $C' = C \setminus \{v\}$  covers all the edges in  $G'$ . So,  $C'$  is a vertex cover of  $G'$  of size  $k^* - 1$ , and therefore  $D(G', k^* - 1)$  accepts, as needed.

Finally we show efficiency. Each loop iteration removes a vertex and its incident edges from the graph and calls  $D$ , which can be done efficiently. The algorithm does at most  $|V|$  such iterations before  $D$  accepts in one of them, at which point the algorithm does one recursive call with argument  $k^* - 1$  and then immediately outputs a correct answer. Since each call to the algorithm does at most  $|V|$  loop iterations, and there are  $k^* \leq |V|$  calls in total (plus the base-case call), the total number of iterations is  $O(|V|^2)$ , so the algorithm is efficient.  $\square$

**Exercise 190** Using access to an oracle (or efficient algorithm)  $D$  that decides SAT, give an efficient algorithm that, on input a Boolean formula  $\phi$ , outputs a satisfying assignment for  $\phi$  if one exists, otherwise outputs “no satisfying assignment exists.”

**Hint:** first use the oracle to determine whether a satisfying assignment exists. Then, for each variable, set it to either true or false and use the oracle to determine whether there is a satisfying assignment of the remaining variables.

We have shown that the search problems of finding a maximum clique, or a minimum vertex cover, efficiently reduce to (i.e., are “no harder than”) the respective decision problems CLIQUE or VERTEX-COVER, and vice versa. That is, any algorithm for one form would imply a comparably efficient algorithm for the other form. In particular, even though the decision versions of these problems might appear syntactically “easier”—they ask only to *determine the existence* of a certain object in a graph, not to actually *find* one—they still capture the intrinsic difficulty of the search versions.

More generally, it is known (though we will not prove this) that *any* NP-complete decision problem has an efficient algorithm if and only if its corresponding search problem does. More formally, there is an efficient Turing reduction from the decision version to the search version, and vice versa.

## APPROXIMATION ALGORITHMS

Search problems, and especially optimization problems, allow us to consider *approximation* algorithms, which we allow to output answers that are “close” to correct or optimal ones. For example, an approximate sorting algorithm may output an array that is “almost” sorted, i.e., a small number of items may be out of order. An approximate vertex-cover algorithm may output a vertex cover that is somewhat larger than a smallest one.

While optimal solutions are of course preferable, for many important problems of interest (like clique and vertex cover) it is unlikely that optimal solutions can be computed efficiently—this would imply that  $P = NP$ , because the corresponding decision problems are NP-hard. So, to circumvent this likely intractability, we instead seek efficient algorithms that produce “close to optimal” solutions, which may be good enough for applications.<sup>66</sup>

An optimization problem seeks a solution (meeting some validity criteria) whose “value” is optimal for a given input, where the definition of “value” depends on the specific problem in question. For a minimization problem, an optimal solution is one whose value is as small as possible (for the given input), and similarly for a maximization problem. An *approximation* is a solution whose value is within some specified factor of optimal, as defined next.

**Definition 191 ( $\alpha$ -approximation)** For an input  $x$  of an optimization problem, let  $\text{OPT}(x)$  denote the value of an optimal solution for  $x$ . For a non-negative real number  $\alpha$ , called the *approximation ratio* or *approximation factor*, an  $\alpha$ -approximation for input  $x$  is a solution whose value  $V$  satisfies:

- For a minimization problem, with  $1 \leq \alpha$ ,

$$\text{OPT}(x) \leq V \leq \alpha \cdot \text{OPT}(x).$$

(Note that  $\text{OPT}(x) \leq V$  holds trivially, by definition of  $\text{OPT}$ .)

- For a maximization problem, with  $\alpha \leq 1$ ,

$$\alpha \cdot \text{OPT}(x) \leq V \leq \text{OPT}(x).$$

(Note that  $V \leq \text{OPT}(x)$  holds trivially, by definition of  $\text{OPT}$ .)

Notice that the closer  $\alpha$  is to one, the better the guarantee on the “quality” of the approximation, i.e., the tighter the above inequalities are.

In more detail, any optimal solution is equivalent to a 1-approximation, and for a minimization problem, it is also (say) a 2-approximation; likewise, any 2-approximation is also (say) a 10-approximation, etc. In general, for a minimization problem and any  $1 \leq \alpha \leq \alpha'$ , any  $\alpha$ -approximation is also a  $\alpha'$ -approximation, and symmetrically for maximization problems. This is simply because the tighter bound required of an  $\alpha$ -approximation trivially implies the looser bound required of an  $\alpha'$ -approximation; the value is not required to be *equal* to any particular multiple of the optimum.

---

<sup>66</sup> Even for efficiently solvable problems like longest common subsequence, the polynomial runtimes might be too large for huge instances that arise in practice, like DNA sequences. In such cases we might seek even faster approximation algorithms for these problems.

**Definition 192 ( $\alpha$ -approximation Algorithm)** An  $\alpha$ -approximation algorithm for an optimization problem is an algorithm  $A$  whose output  $A(x)$  is an  $\alpha$ -approximation for  $x$ , for all inputs  $x$ .

In other words, for any input  $x$ :

- $A(x)$  outputs a valid solution for  $x$ , and
- letting  $\text{ALG}(x)$  denote the value of the algorithm's output  $A(x)$ , we have that  $\text{ALG}(x) \leq \alpha \cdot \text{OPT}(x)$  for a minimization problem, and  $\alpha \cdot \text{OPT}(x) \leq \text{ALG}(x)$  for a maximization problem.

Similar comments as above apply to approximation algorithms; e.g., a 2-approximation algorithm is also a 10-approximation algorithm. An  $\alpha$ -approximation algorithm is not ever required to output a solution whose value is *equal* to any particular multiple of the optimum.

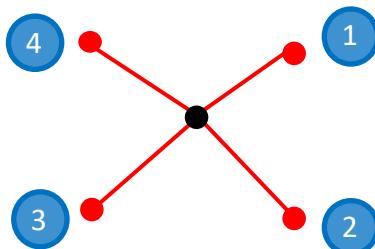
## 20.1 Minimum Vertex Cover

Let us return to the problem of finding a minimum vertex cover. Unless  $P = NP$ , there is no efficient algorithm for this problem. So, we instead relax our goal: is there an efficient algorithm to *approximate* a minimum vertex cover, with a good approximation ratio  $\alpha \geq 1$ ? Our first attempt is what we call the “*single-cover*” algorithm. It works by repeatedly taking an edge, covering it by placing one of its endpoints in the cover, and removing that endpoint and all its incident edges (which are now covered) from the graph. The precise pseudocode is as follows.

**Input:** an undirected graph  
**Output:** a vertex cover of the graph  
**function** SINGLECOVER( $G = (V, E)$ )  
   $C = \emptyset$   
  **while**  $G$  has at least one edge **do**  
    choose an arbitrary edge  $e \in E$ , and let  $v$  be an arbitrary endpoint of  $e$   
     $C = C \cup \{v\}$   
    remove  $v$  and all its incident edges from  $G$   
  **return**  $C$

It is clear that this algorithm outputs a vertex cover of  $G$ , because it removes only those edges covered by  $C$  from the graph, and does not halt until every edge has been removed (i.e., covered). It is also clear that the algorithm is efficient, because it does at most  $|E|$  iterations, and each iteration can be performed efficiently.

How good of an approximation does this algorithm obtain? Unfortunately, its approximation factor can be as large as  $|V| - 1$ , which is trivial: in *any* simple graph, *any*  $|V| - 1$  vertices form a vertex cover. (Indeed, for an edge to be uncovered, both of its endpoints must not be selected.) To see how the algorithm can obtain such a poor approximation, consider the following star-shaped graph:



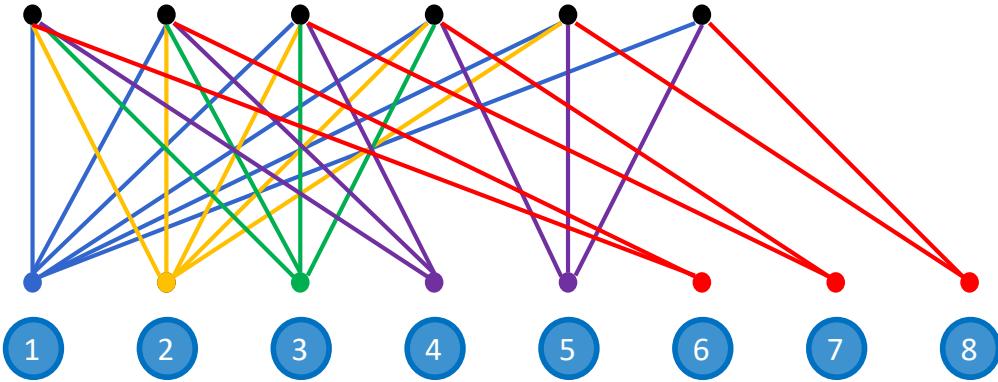
In this graph, a minimum vertex cover consists of just the vertex in the center, so it has size one. However, because the algorithm chooses to remove an arbitrary endpoint of each selected edge, it might choose all the vertices along the “outside” of the star. This would result in it outputting a vertex cover of size four, for an approximation ratio of four. Moreover, larger star-shaped graphs (with more vertices around the center vertex) can result in arbitrarily large

approximation ratios, of just one less than the number of vertices. In particular, there is no constant  $\alpha$  such that the single-cover algorithm obtains an approximation ratio of  $\alpha$  on all input graphs.

Now let us consider another, slightly more sophisticated algorithm. Observe that in the star-shaped graph, the center vertex has the largest *degree* (number of incident edges). If we choose that vertex first, then we cover all the edges, thereby obtaining the optimal cover size (of just a single vertex) for this type of graph. This observation motivates a *greedy* strategy that repeatedly selects and removes a vertex having the (currently) largest degree:

**Input:** an undirected graph  
**Output:** a vertex cover of the graph  
**function** GREEDYCOVER( $G = (V, E)$ )  
 $C = \emptyset$   
**while**  $G$  has at least one edge **do**  
    choose a vertex  $v \in V$  of largest (current) degree  
     $C = C \cup \{v\}$   
    remove  $v$  and all its incident edges from  $G$   
**return**  $C$

This algorithm outputs a vertex cover of  $G$  and is efficient, for the same reasons that apply to SINGLECOVER. While the algorithm works very well for star-shaped graphs, there are other graphs for which its approximation ratio can be arbitrarily large, i.e., not bounded by any constant. As an illustration, consider the following bipartite graph:



An optimal cover consists of the six vertices in the top layer. However, the algorithm instead chooses the eight vertices in the bottom layer, and thus obtains an approximation ratio of  $8/6 = 4/3$ . The vertex at the bottom left has degree six, versus the maximum degree of five among the rest of the vertices, so the bottom-left vertex is selected first, and removed from the graph. Then, the second vertex in the bottom layer has degree five, whereas all the other vertices have degree at most four, so it is selected and removed. Then, the unique vertex with largest degree is the third one in the bottom layer, so it is selected and removed, and so on, until all the bottom-layer vertices are selected and the algorithm terminates.

While the algorithm obtains an approximation ratio of  $4/3$  on this graph, larger graphs with a similar structure can be constructed with  $k$  vertices in the top layer, constituting a minimum vertex cover, and  $\approx k \log k$  vertices in the bottom layer, which the algorithm selects instead. So, on these graphs the algorithm obtains an approximation ratio of  $\approx \log k$ , which can be made arbitrarily large by increasing  $k$ .

To get a better approximation ratio, we revisit the single-cover algorithm, but make a seemingly counter-intuitive modification: whenever we choose an uncovered edge, instead of selecting *one* of its endpoints (arbitrarily), we select *both* of its endpoints, and remove them and their incident edges from the graph. For this reason, we call this the “*double-cover*” algorithm. The pseudocode is as follows.

**Input:** an undirected graph  
**Output:** a 2-approximate vertex cover of the graph

```

function DOUBLECOVER( $G = (V, E)$ )
   $C = \emptyset$ 
  while  $G$  has at least one edge do
    choose an arbitrary edge  $e = (u, v) \in E$ 
     $C = C \cup \{u, v\}$ 
    remove both  $u, v$  and all their incident edges from  $G$ 
  return  $C$ 

```

It is apparent that this algorithm outputs a vertex cover of  $G$  and is efficient, for the same reasons that SINGLECOVER is efficient.

At first glance, double-cover might seem like “overkill,” because to cover an edge it suffices to select only one of its endpoints. However, as we will see next, double-cover has the benefit of allowing us to compare both the algorithm’s output, and an optimal solution, to a common “benchmark” quantity. More specifically, we will *upper-bound* the size of the algorithm’s output in terms of the benchmark, and *lower-bound* the size of an optimal vertex cover in terms of the same benchmark. Combining these bounds then yields an upper bound on the algorithm’s output in terms of an optimal solution, which establishes the approximation ratio.

The benchmark we use to analyze double-cover is given by sets of *pairwise disjoint* edges. In a given graph  $G = (V, E)$ , a subset of edges  $S \subseteq E$  is pairwise disjoint if no two edges in  $S$  share a common vertex.

**Claim 193** *In any graph  $G = (V, E)$ , if  $S \subseteq E$  is pairwise disjoint, then any vertex cover of  $G$  has at least  $|S|$  vertices. In particular,  $OPT \geq |S|$ , where  $OPT$  is the size of a minimum vertex cover of  $G$ .*

**Proof 194** Let  $C$  be an arbitrary vertex cover of  $G$ . Since no two edges in  $S$  share a common endpoint, each vertex in  $C$  covers *at most one* edge in  $S$ . And since  $C$  covers every edge in  $E$ , in particular it covers every edge in  $S \subseteq E$ . So,  $C$  has at least  $|S|$  vertices.  $\square$

**Claim 195** *For any input graph  $G = (V, E)$ , the set  $S$  of edges chosen by DOUBLECOVER is pairwise disjoint.*

**Proof 196** We claim that DOUBLECOVER obeys the following loop invariant: whenever it reaches the top of its while loop, the edges it has been selected so far are pairwise disjoint, and moreover, no edge in (the current state of)  $G$  has a common endpoint with any of the selected edges. In particular, it follows that at the end of the algorithm, the selected edges are pairwise disjoint, as claimed.

We prove the claim by induction. Initially, the claim holds trivially, because no edge has been selected yet. Now assume as the inductive hypothesis that the claim holds at the start of some loop iteration; we show that it also holds at the end of the iteration. By the inductive hypothesis, the newly selected edge  $e = (u, v)$  from  $G$  is disjoint from the previously selected ones (which are themselves pairwise disjoint), and hence all these selected edges are pairwise disjoint. Moreover, the iteration removes the endpoints  $u, v$  and all their incident edges from the graph, so by this and the inductive hypothesis, no edge of the updated  $G$  has a common endpoint with any selected edge. The claimed loop invariant therefore follows by induction.  $\square$

Finally, we combine the previous two claims to obtain an approximation ratio for DOUBLECOVER.

**Theorem 197** *DOUBLECOVER is a 2-approximation algorithm for the minimum vertex cover problem.*

**Proof 198** On input graph  $G$ , let  $OPT$  be the size of a minimum vertex cover, let  $S$  be the set of all the edges selected by a run of DOUBLECOVER, and let  $ALG$  be the size of the output cover.

Because the output cover consists of the endpoints of each edge in  $S$  (and no others),  $ALG \leq 2|S|$ . In fact, this upper bound is actually an equality, because  $S$  is pairwise disjoint by Claim 195, but all we will need is the inequality. Next, because  $S$  is pairwise disjoint, by Claim 193 we have that  $|S| \leq OPT$ . Combining these two

bounds, we get that

$$\text{ALG} \leq 2 \cdot |S| \leq 2 \cdot \text{OPT},$$

hence the theorem follows by [Definition 192](#). □

It is worthwhile to reflect on the general structure of the argument, which is common to many analyses of approximation algorithms. We first identified a “benchmark” quantity, namely, the number of edges selected by a run of the **DOUBLECOVER** algorithm. We then proved both an upper bound on the algorithm’s output size ALG, and a lower bound on the optimal value OPT, in terms of this benchmark. Combining these, we obtained an upper bound on ALG in terms of OPT, as desired.

## 20.2 Maximum Cut

As another example, we consider the problem of finding a *maximum cut* in an undirected graph. We first define the relevant terminology.

- A *cut* in an undirected graph  $G = (V, E)$  is just a subset  $S \subseteq V$  of the vertices; this partitions  $V$  into two disjoint sets,  $S$  and its complement  $\bar{S} = V \setminus S$ , called the *sides* of the cut.
- An edge of the graph *crosses* the cut if its endpoints are in opposite sides of the cut, i.e., one of its endpoints is in  $S$  and the other is in  $\bar{S}$ . Observe that a “self-loop” edge (i.e., one whose endpoints are the same vertex) does not cross any cut, so without loss of generality we assume that the graph has no such edge. The set of crossing edges is denoted

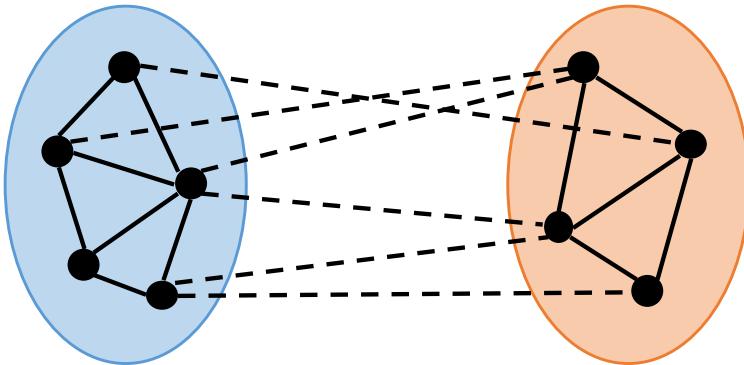
$$C(S) = \{(u, v) \in E : u \in S \text{ and } v \in \bar{S}\}.$$

(Recall that edges  $(u, v)$  are undirected, so for a crossing edge we can require that  $u \in S$  and  $v \in \bar{S}$  without loss of generality.)

- Finally, the *size* of the cut is  $|C(S)|$ , the number of crossing edges. (Note that in this context, the size of a cut  $S$  is *not* the number of vertices in  $S$ !)

Observe that by symmetry, any cut  $S$  is equivalent to its complement cut  $\bar{S}$ : it has the same crossing edges, and hence the same size. So, we can consider  $S$  and  $\bar{S}$  to be freely interchangeable.

The following is an example of a cut  $S$ , with its crossing edges  $C(S)$  represented as dashed line segments:



A *maximum cut* of a graph, or *max-cut* for short, is a cut of maximum size in the graph, i.e., one having the largest possible number of crossing edges. A max-cut represents a partition of the graph that is the most “robust,” in the sense that disconnecting the two sides from each other requires removing the maximum number of edges. Max-cuts have important applications to circuit design, combinatorial optimization, and statistical physics (e.g., magnetism).

The decision version of the maximum-cut problem is defined as:

$$\text{MAX-CUT} = \{(G, k) : G \text{ has a cut of size at least } k\}.$$

This is an NP-complete problem (we omit a proof), so there is no efficient algorithm for the optimization problem unless P = NP.<sup>67</sup> So, we instead focus on designing an efficient approximation algorithm.

To motivate the algorithm's approach, we start with a key observation. Fix a cut and an arbitrary vertex  $v$  of a graph, and consider the edges incident to that vertex. If a strict majority of those edges do *not* cross the cut—i.e., a majority of  $v$ 's neighbors are on  $v$ 's side of the cut—then we can get a larger cut by moving  $v$  to the other side of the cut. This is because doing so turns each of  $v$ 's non-crossing edges into a crossing edge, and vice versa, and it does not affect any other edge. (Here we have used the assumption that the graph has no self-loop edge.) We now formalize this observation more precisely.

For a cut  $S$  of a graph  $G = (V, E)$  and a vertex  $v \in V$ , let  $S(v)$  denote the side of the cut to which  $v$  belongs, i.e.,  $S(v) = S$  if  $v \in S$ , and  $S(v) = \bar{S}$  otherwise. Let  $E_v \subseteq E$  denote the set of edges incident to  $v$ , and partition it into the subsets of crossing edges  $C_v(S) = E_v \cap C(S)$  and non-crossing edges  $N_v(S) = E_v \setminus C_v(S) = E_v \cap \bar{C}(S)$ .

**Lemma 199** *Let  $S$  be a cut in a graph  $G = (V, E)$ ,  $v \in V$  be a vertex, and  $S' = S(v) \setminus \{v\}$  be the cut (or its equivalent complement) obtained by moving  $v$  to the other side. Then*

$$|C(S')| = |C(S)| + |N_v(S)| - |C_v(S)|.$$

*In particular, the size of  $S'$  is larger than the size of  $S$  if and only if a strict majority of  $v$ 's incident edges do not cross  $S$ .*

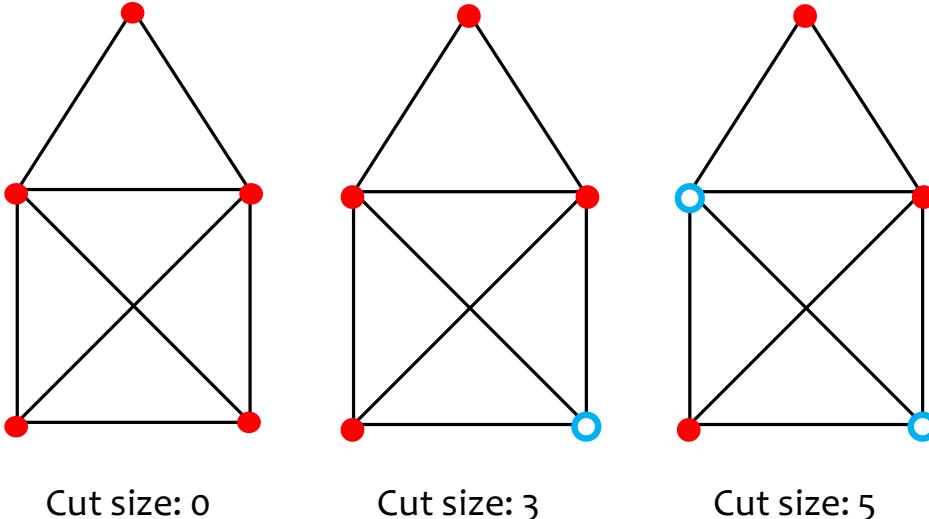
**Proof 200** By definition, the edges crossing  $S' = S(v) \setminus \{v\}$  are the same as those crossing  $S$ , except for the edges incident to  $v$ . For every edge  $e \in E_v$ , we have that  $e \in C(S) \iff e \notin C(S')$ , because  $v$  was moved to the other side of the cut. Therefore,  $C(S') = C(S) \cup N_v(S) \setminus C_v(S)$ . Since  $C(S) \cap N_v(S) = \emptyset$  and  $C_v(S) \subseteq C(S)$ , the claim follows.  $\square$

Lemma 199 suggests the following “local search” algorithm for approximating max-cut: starting from an arbitrary cut, repeatedly find a vertex for which a strict majority of its incident edges do not cross the cut, and move it to the other side of the cut. Once no such vertex exists, output the current cut. The pseudocode is as follows.

**Input:** an undirected graph  
**Output:** a 1/2-approximate maximum cut in the graph  
**function** LOCALSEARCHCUT( $G = (V, E)$ )  
      $S = V$   
     **while** there is a vertex  $v$  for which  $N_v(S) > C_v(S)$  **do**  
          $S = S(v) \setminus \{v\}$   
     **return**  $S$

The following illustrates a possible execution of the local-search algorithm on an example graph:

<sup>67</sup> By contrast, the *minimum* cut problem is efficiently solvable by network-flow algorithms.



Initially, all the vertices are in  $S$ , illustrated above as solid red circles. Since there is no crossing edge, the algorithm can arbitrarily choose any vertex to move to the other side of the cut, depicted as open circles; suppose that it chooses the bottom-right vertex. Next, every vertex still in  $S$  has a strict majority of its edges as non-crossing, so the algorithm can arbitrarily choose any one of them to move to the other side of the cut; suppose that it chooses the middle-left vertex. At this point, no single vertex can be switched to increase the size of the cut, so the algorithm outputs the current cut, which has size five. For comparison, the maximum cut for this graph has size six; try to find it.

We now analyze the `LOCALSEARCHCUT` algorithm, first addressing efficiency. It is clear that each iteration of the algorithm can be done efficiently, by inspecting each vertex and its incident edges. However, it is not immediately obvious how many iterations are performed, or if the algorithm is even guaranteed to halt at all. In particular, it is possible that some vertex switches back and forth, from one side of the cut to the other, several times during an execution of the algorithm. So, one might worry that the algorithm could potentially switch a sequence of vertices back and forth forever. Fortunately, this cannot happen, which we can prove using the *potential method* (page 7).

**Claim 201** *On any input graph  $G = (V, E)$ , `LOCALSEARCHCUT` runs for at most  $|E|$  iterations.*

**Proof 202** At all times during the algorithm, the number of crossing edges  $|C(S)|$  is between 0 and  $|E|$ , inclusive. By the choice of each selected vertex and [Claim 199](#), each iteration of the algorithm increases  $|C(S)|$  by at least 1. So, the algorithm must halt within  $|E|$  iterations.  $\square$

We now analyze the algorithm's quality of approximation on an arbitrary input graph  $G = (V, E)$ . Since max-cut is a *maximization* problem, we seek to prove a *lower bound* on the size  $\text{ALG} = |C(S)|$  of the algorithm's output cut  $C$ , and an *upper bound* on the size  $\text{OPT}$  of a maximum cut in the graph, in terms of some common benchmark quantity.

It turns out that a good choice of benchmark is  $|E|$ , the total number of edges in the graph. Trivially,  $\text{OPT} \leq |E|$ . A lower bound on  $\text{ALG}$  is given by the following.

**Theorem 203** *For any undirected input graph  $G = (V, E)$ , `LOCALSEARCHCUT`( $G$ ) outputs a cut  $S$  of size  $|C(S)| \geq |E|/2$ , and hence `LOCALSEARCHCUT` is a  $1/2$ -approximation algorithm for the max-cut problem.*

**Proof 204** By definition of `LOCALSEARCHCUT`( $G$ ), it outputs a cut  $S$  for which at least half of every vertex's incident edges cross  $S$ , i.e.,  $|C_v(S)| \geq \frac{1}{2}|E_v|$  for every  $v \in V$ . We claim that this implies that at least half of all the edges in the graph cross  $S$ . Indeed,

$$|C(S)| = \frac{\sum_{v \in V} |C_v(S)|}{2} \geq \frac{\sum_{v \in V} |E_v|}{4} = \frac{|E|}{2},$$

where the first equality holds because summing  $|C_v(S)|$  over all  $v \in V$  “double counts” every crossing edge (via both of its endpoints), the inequality uses the lower bound on  $|C_v(S)|$ , and the final equality holds because summing the degrees  $|E_v|$  over all  $v \in V$  double counts every edge in the graph (again via both endpoints).

Next, observe that any maximum cut  $S^*$  of  $G$  (and indeed, any cut at all) has size  $\text{OPT} := |C(S^*)| \leq |E|$ . Combining these two inequalities, we see that  $\frac{1}{2}\text{OPT} \leq |E|/2 \leq |C(S)|$ , and the theorem follows by Definition 192.  $\square$

## 20.3 Knapsack

The *knapsack problem* involves selecting a subset of items that maximizes their total value, subject to a weight constraint. More specifically: we are given  $n$  items, where item  $i$  has some non-negative value  $v_i \geq 0$  and weight  $w_i \geq 0$ . We also have a “knapsack” that has a given weight *capacity*  $C$ . We wish to select a subset  $S$  of the items that *maximizes* the total value

$$\text{value}(S) = \sum_{i \in S} v_i$$

subject to the weight limit:

$$\sum_{i \in S} w_i \leq C .$$

In this problem formulation, we cannot select any single item more than once; each item either is in  $S$ , or not. So, this formulation is also known as the *0-1 knapsack problem*.

Without loss of generality, we assume that  $0 < w_i \leq C$ , i.e., each item’s weight is positive and no larger than the knapsack capacity  $C$ . This is because a zero-weight item can be selected at no cost and without decreasing the obtained value, and an item that weighs more than the capacity cannot be selected at all.

A natural dynamic-programming algorithm computes an optimal set of items in  $\Theta(nC)$  operations, which at first glance may appear to be efficient. However, because the input capacity  $C$  is represented in binary (or some other non-unary base), the  $C$  factor in the running time is actually *exponential in its size* (in digits). Thus, the dynamic-programming algorithm actually is *not efficient*; its running time is not polynomial in the input size.<sup>68</sup> More concretely, the knapsack capacity  $C$ , and  $n$  items’ weights and values, could each be about  $n$  bits long, making the input size  $\Theta(n^2)$  and the running time  $\Omega(n2^n)$ .

In fact, the decision version of the knapsack problem is known to be NP-complete (via a reduction from SUBSET-SUM), so there is no efficient algorithm for this problem unless P = NP. Therefore, we instead attempt to approximate an optimal solution efficiently.

A natural approach is to use a greedy algorithm. Our first attempt is what we call the *relatively greedy* algorithm, in which we select items by their *relative* value  $v_i/w_i$  per unit of weight, in non-increasing order, until we cannot take any more.

**Input:** arrays  $V, W$  of item values and weights, sorted non-increasing by relative value  $V[i]/W[i]$ , and a knapsack capacity  $C$

**Output:** a valid selection of items

```
function RELATIVELYGREEDY( $V[1, \dots, n], W[1, \dots, n], C$ )
     $S = \emptyset$ , weight = 0
    for  $i = 1$  to  $n$  do
        if weight +  $W[i] \leq C$  then
```

<sup>68</sup> An algorithm is said to run in *pseudopolynomial time* if its running time is polynomial in the *value* of the integers in the input, as opposed to the *size* (in digits) of the integers. Equivalently, the algorithm runs in time polynomial in the input size if all numbers in the input are encoded in unary. The dynamic-programming algorithm for knapsack runs in pseudopolynomial time.

```

 $S = S \cup \{i\}$ 
weight = weight +  $W[i]$ 
return  $S$ 

```

The overall running time is dominated by ensuring that the input is appropriately sorted, which takes  $O(n \log n)$  comparisons (e.g., using *merge sort* (page 13)) of the relative values, which can be done efficiently (as always, in terms of the input size, in bits). Thus, this algorithm is efficient.

Unfortunately, the approximation ratio obtained by the RELATIVELYGREEDY algorithm can be arbitrarily bad. Consider just two items, the first having value  $v_1 = 1$  and weight  $w_1 = 1$ , and the second having value  $v_2 = 100$  and weight  $w_2 = 200$ . These items have relative values 1 and  $1/2$ , respectively, so the algorithm will consider them in this order. If the weight capacity is  $C = 200$ , then the algorithm selects the first item, but then cannot select the second item because it would exceed the capacity. (Remember that the algorithm is greedy, and hence does not “reconsider” any of its choices.) So, the algorithm outputs  $S = \{1\}$ , which has a total value of  $\text{value}(S) = 1$ . However, the (unique) optimal solution is  $S^* = \{2\}$ , which has a total value of  $\text{value}(S^*) = 100$ . So for this input, the approximation ratio is just  $1/100$ . Moreover, we can easily generalize this example so that the approximation ratio is as small of a positive number as we would like.

The main problem in the above example is that the relatively-greedy algorithm de-prioritizes the item with the largest *absolute* value, in favor of the one with a larger *relative* value. At the other extreme, we can consider the “single-greedy” algorithm, which selects *just one* item having the largest (absolute) value. (The algorithm takes just one item, even if there is capacity for more.) The pseudocode is as follows.

```

function SINGLEGREEDY( $V, W, C$ )
    let  $i$  be the index of an item having largest value  $V[i]$  return  $S = \{i\}$ 

```

The running time is dominated by the  $O(n)$  comparisons of the entries in the value array, which can be done efficiently.

By inspection, SINGLEGREEDY outputs the optimal solution for the above example input. Unfortunately, there are other examples for which this algorithm has an arbitrarily bad approximation ratio. Consider 201 items, where all items but the last have value 1 and weight 1, while the last item has value 2 and weight 200. Formally,

$$\begin{aligned}
 v_1 &= v_2 = \dots = v_{200} = 1 \\
 w_1 &= w_2 = \dots = w_{200} = 1 \\
 v_{201} &= 2 \\
 w_{201} &= 200 .
 \end{aligned}$$

With a capacity  $C = 200$ , the optimal solution consists of items 1 through 200, for a total value of 200. However, SINGLEGREEDY selects just item 201, for a value of 2. (Note that there is no capacity left, so dropping the restriction to a single item would not help here.) Thus, it obtains an approximation ratio of just  $1/100$  on this input. Moreover, this example can be generalized to make the ratio arbitrarily small.

Observe that for this second example input, the relatively-greedy algorithm would actually output the optimal solution. From these two examples, we might guess that the single-greedy algorithm does well on inputs where the relatively greedy algorithm does poorly, and vice versa. This motivates us to consider the *combined-greedy* algorithm, which runs both the relatively greedy and single-greedy algorithms on the same input, and chooses whichever output obtains larger value (breaking a tie arbitrarily).

```

function COMBINEDGREEDY( $V, W, C$ )
     $S_1 = \text{RELATIVELYGREEDY}(V, W, C)$ 
     $S_2 = \text{SINGLEGREEDY}(V, W, C)$ 
    if  $\text{value}(S_1) > \text{value}(S_2)$  then
        return  $S_1$ 
    else
        return  $S_2$ 

```

Perhaps surprisingly, even though each algorithm can output an arbitrarily bad approximation for certain inputs, it can be shown that COMBINEDGREEDY is a  $1/2$ -approximation algorithm for the knapsack problem! That is, on any input it outputs a solution whose value is at least half the optimal value for that input.

**Exercise 205** In this exercise, we will prove that the COMBINEDGREEDY algorithm is a  $1/2$ -approximation algorithm for the 0-1 knapsack problem.

- Define the *fractional* knapsack problem as a variant that allows selecting any partial amount of each item, between 0 and 1, inclusive. For example, we can take  $3/7$  of item  $i$ , for a value of  $(3/7)v_i$ , at a weight of  $(3/7)w_i$ . Show that for this fractional variant, the optimal value is no smaller than for the original 0-1 variant (for the same weights, values, and knapsack capacity).
- Prove that the *sum* of the values obtained by the relatively greedy and single-greedy algorithms (for 0-1 knapsack) is at least the optimal value for the *fractional* knapsack problem (on the same input).
- Using the previous parts, prove that COMBINEDGREEDY is a  $1/2$ -approximation algorithm for the 0-1 knapsack problem.

## 20.4 Other Approaches to NP-Hard Problems

Efficient approximation algorithms are some of the most useful ways of dealing with NP-hard optimization problems. This is because they come with provable guarantees: on any input, they run in polynomial time, and we have the assurance that their output is “close” to optimal.

There are other fruitful approaches for dealing with NP-hardness as well, which can be used on their own or combined together. These include:

- Heuristics.* These are algorithms that might not output a correct or (near-)optimal solution on all inputs, or might run for more than polynomial time on some inputs; however, they tend to do well in practice, for the kinds of inputs that arise in the real-world application. For example, many SAT solvers used in practice run in exponential time in the worst case, and/or might reject some satisfiable formulas, but they run fast enough and give good answers on the kind of formulas that arise in practice (e.g., in software verification).
- Restricting to special cases.* The motivating application of an algorithmic problem may have additional “structure” that is not present in its NP-hard formulation. For example, there are efficient algorithms for finding maximum cuts in *planar* graphs, which are graphs that can be drawn on the plane without any crossing edges. This may hold in some scenarios, like networks of road segments, but not in others, like fiber-optic networks.

As another example, even though the general traveling salesperson problem TSP cannot be approximated to within a constant factor unless  $P = NP$ , there is an efficient 2-approximation algorithm for “*metric*” TSP, which is the special case where the edge distances satisfy the triangle inequality. This may hold in some applications, like road networks, but not in others, like airfare prices.

- Restricting to small inputs.* In practice, the inputs we encounter might be small enough that we can afford to use certain “inefficient” algorithms, even though they scale poorly as the input size grows. For example, in the knapsack program, if the capacity is not too large, the non-polynomial-time dynamic-programming algorithm might be fast enough for our needs.

**Part IV**

**Randomness**

---

CHAPTER  
**TWENTYONE**

---

## RANDOMIZED ALGORITHMS

Thus far, every algorithm we have seen is *deterministic*. This means that, when run on a given input, the steps the algorithm performs and the output it produces are entirely determined. So, when run multiple times on the same input, the algorithm produces the same output. At heart, this is because any Turing machine’s transition rules are deterministic, i.e., it has a transition *function*.

In this unit, we consider how *randomness* can be applied in computation. We will exploit randomness to design simple, probabilistic algorithms, and we will discuss several tools that can be used to analyze randomized algorithms.

Since Turing machines are deterministic, they cannot make random choices—so how are randomized algorithms possible? In real life, when we want to make a random choice, we can flip a coin, roll a die, or the like. Similarly, to model randomized algorithms and Turing machines, we augment them with an *external source of randomness*.

For this purpose, the simplest source of randomness is a (virtual) “fair coin” that the algorithm can “flip” as needed. Formally, we can define a randomized Turing machine as having a special “coin flip” state that, whenever it is entered, writes a fresh unbiased random bit at the location of the tape head, and transitions back to the previous state. With this basic feature it is possible to simulate richer sources of randomness, like the roll of a die with any finite number of sides.

In this text, we write algorithms and pseudocode that make truly random, unbiased, and independent choices from whatever finite set is convenient. In the real world, when implementing and using randomized algorithms—especially for cryptography—it is vital to use a high-quality source of randomness. (Ensure this can be a very delicate matter, but it is outside the scope of this course.)

As a first example of the utility of randomized algorithms, consider the game of *rock-paper-scissors* between two players. In this game, both players simultaneously choose one of three options: rock, paper, or scissors. The game is a tie if both players choose the same option. If they make different choices, then rock beats (“smashes”) scissors, scissors beats (“cuts”) paper, and paper beats (“covers”) rock:

		Player A		
		Rock	Paper	Scissors
Player B	Rock	Tie	A wins	B wins
	Paper	B wins	Tie	A wins
	Scissors	A wins	B wins	Tie

Suppose we write an algorithm (or strategy) to play the game, e.g., in a best-out-of-five sequence of rounds. Similar to worst-case analysis of algorithms, we should consider our strategy’s performance against an opponent strategy that plays optimally against it. In other words, we should assume that our opponent “knows” our strategy, and plays the best it can with that knowledge.

If we “hard-code” a strategy—like playing rock in the first two rounds, then paper, then scissors, then paper—then how will this strategy perform? The strategy is fixed and deterministic, so an optimal opponent can win every round (namely, play paper in the first two rounds, then scissors, then rock, then scissors). More generally, for *any deterministic strategy*—even one that is “adaptive” based on the opponent’s moves—there *exists* an opponent strategy that entirely beats it. This illustrates the problem with deterministic strategies: they make the program’s actions predictable, and thus easily countered.

Now consider a strategy that chooses a *random* action in each move, with equal probability for rock, paper, and scissors. Here, even an opponent that knows this strategy does not know in advance which action the strategy will choose, and therefore cannot universally counter that move. In fact, we prove below that *no matter what strategy the opponent uses*, the probability that the opponent wins an individual round is just 1/3 (and the same goes for tying and losing the round). To see why this is the case, we first review some basic tools of probability.

## 21.1 Review of Probability

We first recall the basic definitions and axioms of probability. For simplicity, we focus on *discrete* probability, which will be sufficient for our purposes.

### 21.1.1 Probability Spaces and Events

**Definition 206 (Probability Space)** A (discrete) *probability space* is a countable set  $\Omega$ , called the *sample space* of possible *outcomes*, along with a *probability function*  $\Pr: \Omega \rightarrow [0, 1]$  that assigns a probability to each outcome, and satisfies

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1 .$$

Let’s elaborate on this definition and consider some examples. The sample space  $\Omega$  is the set of every possible outcome—i.e., “every thing that could potentially happen”—in a randomized process, which is also known as an *experiment*. Each outcome has a probability between 0 and 1 (inclusive) of occurring, as defined by the probability function  $\Pr$ , and all these probabilities must sum to 1.

**Example 207 (Three Fair Coins: Probability Space)** Consider the experiment of tossing three “fair” (unbiased) coins in sequence. We formalize this as a probability space—i.e., a sample space and a probability function on it.

The sample space  $\Omega$  is the set of all  $8 = 2^3$  possible sequences of three heads or tails that can occur, i.e.,

$$\Omega = \{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT\} ,$$

where  $H$  represents heads and  $T$  represents tails.

Next, we define an appropriate probability function. To represent the fairness of the coins, we require that each of the eight outcomes has the same probability. Since by definition the probabilities of all eight outcomes must sum to 1, we should define  $\Pr[\omega] = 1/8$  for every outcome  $\omega \in \Omega$ , e.g.,  $\Pr[HHH] = \Pr[HHT] = \dots = \Pr[TTT] = 1/8$ .<sup>69</sup>

These definitions generalize straightforwardly to the probability experiment of tossing  $n \geq 0$  fair coins, where the sample space is  $\Omega = \{H, T\}^n$ , and each outcome  $\omega \in \Omega$  has the same probability  $\Pr[\omega] = 1/2^n$ .

<sup>69</sup> If we wanted to, we could also define some impossible outcomes by including them in our sample space and assigning them probabilities of zero, like  $FFF$  for the outcome that all three coins turn into fish. This would still be a valid probability space because it satisfies all the requirements from the definition, but the impossible outcomes would just be needless clutter. In other situations, however, it might be useful or simpler to define some zero-probability outcomes.

**Definition 208 (Event)** For a probability space  $(\Omega, \Pr)$ , an *event* is any subset  $E \subseteq \Omega$  of the sample space. The probability function is extended to events as

$$\Pr[E] = \sum_{\omega \in E} \Pr[\omega].$$

In other words, an event is any subset of outcomes, and its probability is the sum of the probabilities of those outcomes. We now build on the previous example.

**Example 209 (Three Fair Coins: Events)** Continuing from the probability space for tossing three fair coins as defined in [Example 207](#), we now consider some example events.

- The event “the first toss comes up heads” is formalized as  $E_1 = \{HHH, HHT, HTH, HTT\} \subseteq \Omega$ , i.e., the subset of all outcomes in which the first character is  $H$ . By definition, the probability of this event is the sum of the probabilities of all the outcomes in the event. Since this event has four outcomes, and each has probability  $1/8$ , we get that  $\Pr[E_1] = 4/8 = 1/2$ . This matches our intuition that the probability of getting heads on the first toss (ignoring what happens on the remaining two tosses) is  $1/2$ .
- The event “the first and third tosses come up the same” is formalized as the subset  $E_2 = \{HHH, HTH, THT, TTT\} \subseteq \Omega$ . Similarly, the probability of this event is  $\Pr[E_2] = 4/8 = 1/2$ . This also matches our intuition: no matter how the first toss comes up, the third toss has a  $1/2$  probability of matching it.
- The event “the same number of heads and tails come up” is formalized as the *empty* subset  $E_3 = \emptyset \subseteq \Omega$ . This is because no outcome has the same number of heads and tails.<sup>70</sup> By definition, the probability of this event is  $\Pr[E_3] = 0$ .

<sup>70</sup> If the impossible outcome FFF (as mentioned in the previous footnote) was a member of the sample space, then because it has the same number of heads and tails, it would be an element of event  $E_3$ . However, we would still have  $\Pr[E_3] = 0$ , because  $\Pr[FFF] = 0$ .

Because events are just subsets of the sample space, we can use set relations to relate their probabilities. For example, for any events  $E_1 \subseteq E_2$ , we have that  $\Pr[E_1] \leq \Pr[E_2]$ , because  $\Pr[\omega] \geq 0$  for every outcome  $\omega$ . (However, we caution that  $E_1 \subsetneq E_2$  does not imply that  $\Pr[E_1] < \Pr[E_2]$ , because the events in  $E_2 \setminus E_1$  might have probability zero.)

We can also use set operations on events to yield other events. For example, every event  $E$  has a *complement* event  $\bar{E} = \Omega \setminus E$ , whose probability is  $\Pr[\bar{E}] = 1 - \Pr[E]$ , because

$$\begin{aligned}\Pr[E] + \Pr[\bar{E}] &= \sum_{\omega \in E} \Pr[\omega] + \sum_{\omega \in \bar{E}} \Pr[\omega] \\ &= \sum_{\omega \in \Omega} \Pr[\omega] \\ &= 1.\end{aligned}$$

We can also consider intersections and unions of events. Notationally, instead of  $\cap$  and  $\cup$ , we usually use the logical operators  $\wedge$  (AND) and  $\vee$  (OR), respectively. This captures the idea that the intersection of two events represents the occurrence of both events simultaneously, and their union represents the occurrence of one event or the other (or both).

For two events  $E_1, E_2$ , the probability  $\Pr[E_1 \wedge E_2]$  that both occur simultaneously is called their *joint probability* (and similarly for more events). It follows immediately that this probability is at most  $\min\{\Pr[E_1], \Pr[E_2]\}$ , because  $E_1 \wedge E_2 \subseteq E_1, E_2$ . When considering the intersection of events, an important case is when, informally, the (non-)occurrence of one event does not affect the probability of the (non-)occurrence of another event. This is known as *independence*; the formal definition is as follows.

**Definition 210 (Independent Events)** Two events  $E_1, E_2$  (of the same probability space) are *independent* if  $\Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2]$ .

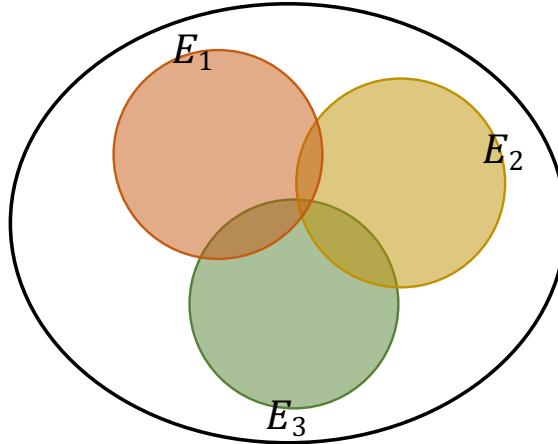
More generally, events  $E_1, \dots, E_n$  are *mutually independent* if  $\Pr[E_1 \wedge \dots \wedge E_n] = \Pr[E_1] \cdots \Pr[E_n]$ . They are *pairwise independent* if every pair of them is independent.

Mutual independence of several events is a stronger notion than pairwise independence: a collection of events can be pairwise independent but not mutually independent (see Example 218 below).

**Example 211 (Three Fair Coins: Independence)** Continuing from Example 209, we consider several events and whether they are independent:

- The events  $E_1$  (“the first toss comes up heads”) and  $E_2$  (“the first and third tosses come up the same”) are independent, because  $E_1 \wedge E_2 = \{HHH, HTH\}$  and hence  $\Pr[E_1 \wedge E_2] = 1/4 = 1/2 \cdot 1/2 = \Pr[E_1] \cdot \Pr[E_2]$ . This matches our intuition that whether or not the first toss comes up heads, the probability that the third toss matches the first one is not affected.
- The three events “the first/second/third toss comes up heads” are *mutually* independent (and hence also pairwise independent): their intersection is the event  $\{HHH\}$ , which has probability  $1/8 = (1/2)^3$ , which is the product of the  $1/2$  probabilities of the three individual events. The same holds if we replace “heads” with “tails” in any of these events. All this matches our intuition that the results of any of the tosses do not affect any of the other tosses.
- The two events “the first/last pair of tosses come up heads” are *not* independent: their intersection is the event  $\{HHH\}$ , which has probability  $1/8$ , but they each have probability  $1/4$ . This matches our intuition: the first pairs of tosses coming up heads makes it more likely that the last pairs of tosses also do, because they have the middle (second) toss in common.
- The three events “the [first two/last two/first and third] tosses come up the same” are *pairwise* independent, but not *mutually* independent. Each of them has probability  $1/2$ , and the intersection of any two of them is the event  $\{HHH, TTT\}$ , which has probability  $1/4 = 1/2 \cdot 1/2$ . However, the intersection of all three of them is also  $\{HHH, TTT\}$ , whereas the product of their individual  $1/2$  probabilities is  $1/8$ . This failure of mutual independence is because the occurrence of any two of these events implies the occurrence of the third one.

When considering the union of events, the following lemma is a basic but widely applicable tool.



**Lemma 212 (Union Bound)** Let  $E_1, E_2 \subseteq \Omega$  be events of the same probability space. Then

$$\Pr[E_1 \vee E_2] \leq \Pr[E_1] + \Pr[E_2],$$

with equality if the events are disjoint (i.e., if  $E_1 \wedge E_2 = \emptyset$ ).

More generally, the probability of the union of any countably many events is at most the sum of their individual probabilities, with equality if the events are pairwise disjoint.

**Proof 213** By definition,

$$\begin{aligned}\Pr[E_1 \vee E_2] &= \sum_{\omega \in E_1 \vee E_2} \Pr[\omega] \\ &\leq \sum_{\omega \in E_1} \Pr[\omega] + \sum_{\omega \in E_2} \Pr[\omega] \\ &= \Pr[E_1] + \Pr[E_2],\end{aligned}$$

where the inequality holds because the two sums on the right include every  $\omega \in E_1 \vee E_2$  at least once, and  $\Pr[\omega] \geq 0$  for those  $\omega$  that are double-counted, i.e., the  $\omega \in E_1 \wedge E_2$ . In particular, if  $E_1, E_2$  are disjoint, then no outcome is double-counted, hence the bound is an equality.

The more general claim (for more than two events) follows via induction, by applying the main claim (for two events) to the first event and the union of the others, and applying the inductive hypothesis to the latter union.  $\square$

### 21.1.2 Random Variables

A *random variable* is, informally, a numerical quantity whose value is determined by the outcome of a probability experiment. It is defined formally as follows.

**Definition 214 (Random Variable)** A *random variable* of a probability space  $(\Omega, \Pr)$  is a function  $X: \Omega \rightarrow \mathbb{R}$  from the sample space to the real numbers. So, the set of potential values of  $X$  is its image  $X(\Omega) = \{X(\omega) : \omega \in \Omega\}$ .

For any real number  $v \in \mathbb{R}$ , the notation  $X = v$  denotes the event  $X^{-1}(v) = \{\omega \in \Omega : X(\omega) = v\}$ , i.e., the set of all outcomes that map to value  $v$  under  $X$ . The events  $X \geq v$ ,  $X < v$ , etc. are defined similarly.

By the above definition, the probabilities of the events  $X = v$  and  $X \geq v$  are, respectively,

$$\begin{aligned}\Pr[X = v] &= \sum_{\omega: X(\omega)=v} \Pr[\omega], \\ \Pr[X \geq v] &= \sum_{\omega: X(\omega)\geq v} \Pr[\omega].\end{aligned}$$

A frequently useful kind of random variable is an *indicator* random variable, also called a *0-1 random variable*.

**Definition 215 (Indicator Random Variable)** An *indicator* random variable (of a probability space  $(\Omega, \Pr)$ ) is one that is limited to the values 0 and 1, i.e., a function  $X: \Omega \rightarrow \{0, 1\}$ .

An indicator random variable can be defined for any event  $E$ , as  $X(\omega) = 1$  for every  $\omega \in E$  and  $X(\omega) = 0$  otherwise. This means that the random variable has value 1 if the event occurs, and value 0 if the event does not occur. In the other direction, any indicator random variable  $X$  has a corresponding event  $X^{-1}(1) = \{\omega \in \Omega : X(\omega) = 1\}$ , the set of exactly those outcomes that make the random variable have value 1.

Similar to the case for events, we can define independence for random variables.

**Definition 216 (Independent Random Variables)** Random variables  $X_1, X_2$  are *independent* if for any  $v_1, v_2 \in \mathbb{R}$ , the events  $X_1 = v_1$  and  $X_2 = v_2$  are independent, i.e.,

$$\Pr[X_1 = v_1 \wedge X_2 = v_2] = \Pr[X_1 = v_1] \cdot \Pr[X_2 = v_2].$$

Mutual and pairwise independence of several random variables are defined similarly, as in [Definition 210](#).

**Example 217 (Three Fair Coins: Random Variables and Independence)** Building on Example 207 from above, we define several random variables and consider whether they are independent.

We first define a random variable  $X$  representing the *number of heads that come up*, as the function  $X : \Omega \rightarrow \mathbb{R}$  where

$$\begin{aligned} X(TTT) &= 0, \\ X(HTT) = X(THT) = X(TTH) &= 1, \\ X(HHT) = X(HTH) = X(THH) &= 2, \\ X(HHH) &= 3. \end{aligned}$$

The event  $X = 0$  is  $\{TTT\}$ , and hence has probability  $\Pr[X = 0] = 1/8$ . The event  $X \geq 2$  is  $\{HHT, HTH, THH, HHH\}$ , and hence has probability  $\Pr[X \geq 2] = 4/8 = 1/2$ . The events  $X = 1.5$  and  $X = 5$  are both  $\emptyset$ , and hence each have probability zero.

For each  $i = 1, 2, 3$ , we can define an *indicator* random variable  $X_i$  that indicates whether the  $i$ th toss comes up heads. Specifically,  $X_i(\omega) = 1$  if the  $i$ th character of  $\omega$  is  $H$ , and  $X_i(\omega) = 0$  otherwise. For example,  $X_2$  is the indicator random variable for the event  $\{HHH, HHT, THH, THT\}$ . By inspection,  $\Pr[X_i = 0] = \Pr[X_i = 1] = 1/2$  for every  $i$ .

Observe that  $X = X_1 + X_2 + X_3$ , because  $X$  is the total number of heads, and each  $X_i$  contributes 1 to the sum if and only if the  $i$ th toss comes up heads (otherwise it contributes 0).

The random variables  $X_1, X_2, X_3$  are *mutually* independent (and hence also pairwise independent). To see this, consider any  $v_1, v_2, v_3 \in \{0, 1\}$ . Then the event  $X_1 = v_1 \wedge X_2 = v_2 \wedge X_3 = v_3$  consists of exactly one outcome (because the  $v_i$  values collectively specify the result of every toss), so it has probability  $1/8 = (1/2)^3$ , which is the product of the  $1/2$  probabilities of the three individual events  $X_i = v_i$ .

All of the definitions and claims from this example generalize straightforwardly to the probability experiment of tossing any number of fair coins.

**Example 218 (Three Fair Coins: More Random Variables and (Non-)Independence)** We can also define a random variable  $Y$  representing the *number of pairs of tosses that come up the same*. That is,

$$\begin{aligned} Y(HHH) = Y(TTT) &= 3, \\ Y(\omega) &= 1 \text{ otherwise.} \end{aligned}$$

For each pair of coin tosses, we can define an *indicator* random variable that indicates whether that pair comes up the same. Specifically, for each  $i = 1, 2, 3$ , consider the pair of tosses that does *not include* the  $i$ th toss; then let  $Y_i = 1$  if this pair comes up the same, and  $Y_i = 0$  otherwise. For example,  $Y_1$  is the indicator random variable for the event  $\{HHH, HTT, THH, TTT\}$ , so  $\Pr[Y_1 = 1] = \Pr[Y_1 = 0] = 1/2$ , and similarly for  $Y_2, Y_3$ .

Observe that  $Y = Y_1 + Y_2 + Y_3$ , because  $Y$  is the total number of pairs that come up the same, and each  $Y_i$  corresponds to one of the three pairs, contributing 1 to the sum if that pair comes up the same, and 0 otherwise.

The random variables  $Y_1, Y_2, Y_3$  are *pairwise* independent, but not *mutually* independent. To see pairwise independence, take any values  $v_i, v_j \in \{0, 1\}$  for any  $i \neq j$ . Then the event  $Y_i = v_i \wedge Y_j = v_j$  consists of exactly two outcomes, so it has probability  $1/4$ , which is the product of the  $1/2$  probabilities of the two individual events  $Y_i = v_i$  and  $Y_j = v_j$ . This matches our intuition that whether one pair of coins comes up the same does not affect whether another pair of coins does (even if the two pairs have a coin in common).

To see that the  $Y_i$  are not *mutually* independent, there are several counterexamples to the definition. For instance, the event  $Y_1 = 1 \wedge Y_2 = 1 \wedge Y_3 = 1$  is  $\{HHH, TTT\}$ , so it has probability  $1/4$ , which is not the product of the  $1/2$  probabilities of the three individual events  $Y_i = 1$  for  $i = 1, 2, 3$ . This matches our intuition: if two pairs of coins come up the same, then this implies that the third pair comes up the same as well.

**Definition 219** The *probability distribution* (also known as *probability mass function*) of a random variable  $X$  is the function  $p_X : \mathbb{R} \rightarrow [0, 1]$  that maps each real value to the probability that  $X$  takes on this value, i.e.,  $p_X(v) = \Pr[X = v]$ .

**Example 220 (Three Fair Coins: Probability Distributions)** Continuing from Example 217, the probability distribution of the random variable  $X$  representing the number of heads that come up is

$$\begin{aligned} p_X(0) &= \Pr[X = 0] = 1/8, \\ p_X(1) &= \Pr[X = 1] = 3/8, \\ p_X(2) &= \Pr[X = 2] = 3/8, \\ p_X(3) &= \Pr[X = 3] = 1/8, \\ p_X(v) &= \Pr[X = v] = 0 \text{ otherwise.} \end{aligned}$$

The probability distribution of the random variable  $Y$  from Example 218 is  $p_Y(1) = 3/4$ ,  $p_Y(3) = 1/4$ , and  $p_Y(v) = 0$  otherwise.

### 21.1.3 Expectation

The probability distribution of a random variable gives the probability of each potential value of the variable. Often, we seek more succinct “summary” quantities about the random variable. One such quantity is its *expectation*, which is the “weighted average” of the random variable, where each value is weighted by its probability.

**Definition 221 (Expectation)** The *expected value*, also known as *expectation*, of a random variable  $X$  is defined as

$$\mathbb{E}[X] = \sum_{v \in X(\Omega)} v \cdot \Pr[X = v].$$

The name “expected value” is a bit of a misnomer, because it is not really the value we “expect” to get in the probability experiment. Indeed, a random variable might not even be able to take on its expected value at all! (Example 224 below gives a simple example of this.) Instead, it is good to think of the expectation as the *average* value of a random variable.

Recall that

$$\Pr[X = v] = \sum_{\omega: X(\omega)=v} \Pr[\omega].$$

Plugging this into the definition of expectation, we get that

$$\begin{aligned} \mathbb{E}[X] &= \sum_{v \in X(\Omega)} v \cdot \left( \sum_{\omega: X(\omega)=v} \Pr[\omega] \right) \\ &= \sum_{v \in X(\Omega)} \sum_{\omega: X(\omega)=v} X(\omega) \cdot \Pr[\omega] \\ &= \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega] \end{aligned}$$

as an alternative expression of  $\mathbb{E}[X]$ .

**Lemma 222** If  $X$  is an indicator random variable, then  $\mathbb{E}[X] = \Pr[X = 1]$ .

**Proof 223** This follows directly from the definition of expectation, and the fact that an indicator random variable

is limited to the values 0, 1:

$$\mathbb{E}[X] = 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] = \Pr[X = 1].$$

**Example 224 (Three Fair Coins: Expectations)** Continuing from [Example 217](#), the expectation of the random variable  $X$  (the number of heads that come up) is

$$\begin{aligned}\mathbb{E}[X] &= \sum_{v \in X(\Omega)} v \cdot \Pr[X = v] \\ &= \sum_{v=0}^3 v \cdot \Pr[X = v] \\ &= 0 \cdot 1/8 + 1 \cdot 3/8 + 2 \cdot 3/8 + 3 \cdot 1/8 \\ &= 12/8 = 3/2.\end{aligned}$$

This matches our intuition, that when we toss three fair coins, the average number of heads that come up is  $3/2$ . Notice that this expectation is *not* a value that  $X$  can actually take on; it is merely the “average” value.

Continuing from [Example 218](#), the expectation of  $Y$  (the number of *pairs* of tosses that come up the same) is

$$\begin{aligned}\mathbb{E}[Y] &= \sum_{v \in Y(\Omega)} v \cdot \Pr[Y = v] \\ &= 1 \cdot 3/4 + 3 \cdot 1/4 \\ &= 6/4 = 3/2.\end{aligned}$$

This also might (or might not!) match our intuition: there are three pairs of tosses, and each one has probability  $1/2$  of coming up the same. Even though these three events are *not* mutually independent (see [Example 218](#)), the expected number of equal pairs of tosses happens to be  $3/2$ . This is not a coincidence, as we will see next.

An important feature of averages is that it is impossible for all potential outcomes to be above average, nor can they all be below average. In other words, there must be some outcome that is at least the expectation, and one that is at most the expectation. This is formalized in the following lemma, which is known as the *averaging argument*.

**Lemma 225 (Averaging Argument)** *Let  $X$  be a random variable of a probability space  $(\Omega, \Pr)$ . Then there exists an outcome  $\omega \in \Omega$  for which  $X(\omega) \geq \mathbb{E}[X]$ , and an outcome  $\omega' \in \Omega$  for which  $X(\omega') \leq \mathbb{E}[X]$ .*

For example, building on [Example 224](#), there exists at least one outcome for which  $X$ , the number of heads that come up, is at least  $3/2$ . Indeed,  $HHH$  and  $HHT$  are two such outcomes.

**Proof 226** Let  $E = \mathbb{E}[X]$ , and suppose for the purposes of contradiction that  $X(\omega) < E$  for every  $\omega \in \Omega$ . Then by the alternative formula for the expectation, and because the  $\Pr[\omega]$  are non-negative and sum to 1 (so at least one of them is positive),

$$\begin{aligned}\mathbb{E}[X] &= \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega] \\ &< \sum_{\omega \in \Omega} E \cdot \Pr[\omega] \\ &= E \cdot \sum_{\omega \in \Omega} \Pr[\omega] = E.\end{aligned}$$

Thus  $\mathbb{E}[X] < E$ , which contradicts the definition of  $E$ , hence the first claim is proved. The second claim proceeds symmetrically.  $\square$

For a *non-negative* random variable, its expectation gives some useful information about its probability distribution. Informally, it “not so likely” that the random variable is “much larger than” its expectation. *Markov’s inequality* quantifies this.

**Theorem 227 (Markov’s Inequality)** *Let  $X$  be a non-negative random variable, i.e.,  $X$  never takes on a negative value, and let  $a > 0$ . Then*

$$\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}.$$

If  $\mathbb{E}[X] > 0$ , this implies that for any  $t > 0$ , the probability that  $X$  is at least  $t$  times its expectation is at most  $1/t$ . Notice that these statements are trivial (or “useless”) for  $a \leq \mathbb{E}[X]$  or  $t \leq 1$ , because in these cases Markov’s inequality gives probability upper bounds that are  $\geq 1$ , and the probability of *any* event is trivially  $\leq 1$ . So, Markov’s inequality is useful only when we take  $a > \mathbb{E}[X]$  or  $t > 1$ .

As an example, if the class average on an exam is 70 points (and negative points are not possible), then Markov’s inequality implies that the fraction of the class that earned at least  $a = 90$  points is at most  $70/90 = 7/9$ . (Stated probabilistically: if we choose a student uniformly at random, then the probability that the student earned  $\geq 90$  points is  $\leq 7/9$ .) Similarly, at most  $(7/6)$ ths of the class earned  $\geq 60$  points, but this is a trivial statement.

**Proof 228** By definition of expectation,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{v \in X(\Omega)} v \cdot \Pr[X = v] \\ &= \left( \sum_{v < a} v \cdot \Pr[X = v] \right) + \left( \sum_{v \geq a} v \cdot \Pr[X = v] \right) \\ &\geq \sum_{v \geq a} v \cdot \Pr[X = v] \\ &\geq \sum_{v \geq a} a \cdot \Pr[X = v] \\ &= a \cdot \Pr[X \geq a].\end{aligned}$$

The first inequality holds because  $X$  is non-negative, so every  $v$  in the first sum is non-negative, and therefore that sum is non-negative. The second inequality holds because  $v \geq a$ , and all the  $\Pr[X = v]$  are non-negative. The last equality holds because the event  $X \geq a$  is the union of all the disjoint events  $X \geq v$  for  $v \geq a$ .

Finally, dividing both sides of the inequality by  $a > 0$  (which preserves the direction of the inequality), we get that

$$\frac{\mathbb{E}[X]}{a} \geq \Pr[X \geq a].$$

The following ‘reverse’ version of Markov’s inequality says, informally, that if a random variable is upper bounded, then it has a certain positive probability of exceeding any value less than its expectation. (Note that by the averaging argument, there is some nonzero probability that the random variable is at least its expectation. The reverse Markov inequality gives a *specific* probability bound, which increases as the threshold of interest decreases.)

**Theorem 229 (Reverse Markov’s Inequality)** *Let  $X$  be a random variable that is never larger than some value  $b$ . Then for any  $a < b$ ,*

$$\Pr[X \geq a] \geq \Pr[X > a] \geq \frac{\mathbb{E}[X] - a}{b - a}.$$

Notice that the statement is trivial (or “useless”) for  $a \geq \mathbb{E}[X]$ , because in this case it gives a probability lower bound that is  $\leq 0$ , and the probability of *any* event is  $\geq 0$ .

For example, if the class average on an exam is 70 points out of a maximum of  $b = 100$ , then the ‘reverse’ Markov inequality implies that at least  $(70 - 60)/(100 - 60) = 1/4$  of the class earned more than  $a = 60$  points. However, it does not say anything about how much of the class earned 70 or more points.

**Proof 230** The first inequality holds because the event  $X > a$  is a subset of the event  $X \geq a$ . So, it remains to prove the second inequality.

Define the random variable  $Y = b - X$ . This is non-negative because  $X$  is never larger than  $b$ , so we can apply Markov’s inequality to it, and it has expectation  $\mathbb{E}[Y] = b - \mathbb{E}[X]$ . Hence,

$$\begin{aligned}\Pr[X \leq a] &= \Pr[b - Y \leq a] \\ &= \Pr[Y \geq b - a] \\ &\leq \frac{\mathbb{E}[Y]}{b - a} \\ &= \frac{b - \mathbb{E}[X]}{b - a}.\end{aligned}$$

So, considering the complementary event,

$$\Pr[X > a] = 1 - \Pr[X \leq a] \geq \frac{(b - a) - (b - \mathbb{E}[X])}{b - a} = \frac{\mathbb{E}[X] - a}{b - a}.$$

Often, we can express a random variable  $X$  as a linear combination of some other, typically “simpler,” random variables  $X_i$ . We can then apply *linearity of expectation* to compute the expectation of  $X$  from the expectations of the  $X_i$ . Importantly, the  $X_i$  do not need to be independent in order for linearity of expectation to apply.

**Theorem 231 (Linearity of Expectation)** Let  $X_1, X_2, \dots$  be any countably many random variables, and  $c_1, c_2, \dots \in \mathbb{R}$ . Then

$$\mathbb{E}\left[\sum_i c_i \cdot X_i\right] = \sum_i c_i \cdot \mathbb{E}[X_i].$$

In other words, constant-factor scalings and summations can be “moved in and out of” expectations.

**Proof 232** We prove the case for a linear combination of two variables; the general case then follows by induction. From the alternative formulation of expectation,

$$\begin{aligned}\mathbb{E}[c_1 X_1 + c_2 X_2] &= \sum_{\omega \in \Omega} (c_1 X_1(\omega) + c_2 X_2(\omega)) \cdot \Pr[\omega] \\ &= \sum_{\omega \in \Omega} (c_1 X_1(\omega) \cdot \Pr[\omega] + c_2 X_2(\omega) \cdot \Pr[\omega]) \\ &= \sum_{\omega \in \Omega} c_1 X_1(\omega) \cdot \Pr[\omega] + \sum_{\omega \in \Omega} c_2 X_2(\omega) \cdot \Pr[\omega] \\ &= c_1 \cdot \sum_{\omega \in \Omega} X_1(\omega) \cdot \Pr[\omega] + c_2 \cdot \sum_{\omega \in \Omega} X_2(\omega) \cdot \Pr[\omega] \\ &= c_1 \cdot \mathbb{E}[X_1] + c_2 \cdot \mathbb{E}[X_2].\end{aligned}$$

**Example 233 (Three Fair Coins: Linearity of Expectation)** Continuing from [Example 217](#), recall that the random variable  $X$  is the number of heads that come up. In [Example 224](#), we showed directly via the definition that  $\mathbb{E}[X] = 3/2$ . What follows is an alternative, simpler way—especially when generalizing to more coin tosses—to obtain the same result using linearity of expectation.

Recall that  $X$  is the sum of the three indicator random variables  $X_i$  that indicate whether the  $i$ th toss comes up heads:  $X = X_1 + X_2 + X_3$ . Recall from Example 217 that  $\mathbb{E}[X_i] = \Pr[X_i = 1] = 1/2$  for every  $i$ . So, by linearity of expectation (Lemma 231),

$$\mathbb{E}[X] = \mathbb{E}[X_1 + X_2 + X_3] = \mathbb{E}[X_1] + \mathbb{E}[X_2] + \mathbb{E}[X_3] = 3/2.$$

Similarly, in Example 218, we defined the random variable  $Y$  to be the number of *pairs* of tosses that come up the same, and showed directly from the definition that  $\mathbb{E}[Y] = 3/2$ .

We observed that  $Y$  is the sum of the three indicator random variables  $Y_i$  that indicate whether the corresponding pair of tosses come up the same:  $Y = Y_1 + Y_2 + Y_3$ . Even though these indicator random variables are not mutually independent, we can still apply linearity of expectation to their sum:

$$\mathbb{E}[Y] = \mathbb{E}[Y_1 + Y_2 + Y_3] = 3/2.$$

**Example 234 (Biased Coins)** Suppose we repeatedly flip a biased coin that has probability  $p \in [0, 1]$  of coming up heads. We can determine the expected number of heads in  $n$  flips, using indicator random variables and linearity of expectation.

For  $i = 1, \dots, n$ , define  $X_i$  to be the indicator random variable indicating whether the  $i$ th flip comes up heads. Then  $\mathbb{E}[X_i] = \Pr[X_i] = p$ . Define  $X$  to be the total number of heads that come up in  $n$  flips, and observe that  $X = \sum_{i=1}^n X_i$ . So, by linearity of expectation,

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n \mathbb{E}[X_i] = \sum_{i=1}^n p = p \cdot n.$$

By contrast, deriving this expectation directly via the definitions (of expectation, and of the random variable  $X$ ) would be rather complicated: we would need to determine  $\Pr[X = v]$  for each  $v = 0, 1, \dots, n$ , then analyze  $\sum_{i=1}^n v \cdot \Pr[X = v]$ . Linearity of expectation makes the derivation almost trivial.

### 21.1.4 Analyzing Rock-Paper-Scissors

Now let's return to the rock-paper-scissors example, modeling one round of the game as a probability space, and analyzing the appropriate events. The sample space—i.e., the set of outcomes that can potentially occur—is the set  $\Omega = \{R, P, S\}^2 = \{RR, RP, RS, PR, PP, PS, SR, SP, SS\}$  of possible move pairs made by the two players, where  $R, P, S$  respectively denote the moves rock, paper, and scissors, and the  $i$ th character represents player  $i$ 's move, for  $i = 1, 2$ .

Next, let  $R_i$  be the event that player  $i$  plays rock, i.e.,  $R_1 = \{RR, RP, RS\}$  and  $R_2 = \{RR, PR, SR\}$ , and similarly for  $P_i$  and  $S_i$ .

We need to assign probabilities to the outcomes. Player  $i$ 's strategy can be seen as choosing a move at random according to some specified probabilities of events  $R_i, P_i, S_i$ , which must sum to 1 because they partition the sample space. (This is true even for deterministic strategies, in which case the chosen move has probability 1.) Since the players move simultaneously, we model their choices as *independent*. So, letting  $M_i$  be any one of  $R_i, P_i, S_i$  for  $i = 1, 2$ , we see that the event  $M_1 \wedge M_2$  consists of a single outcome, and we have that

$$\Pr[M_1 \wedge M_2] = \Pr[M_1] \cdot \Pr[M_2].$$

Now, let  $W_1 = \{RS, PR, SP\}$  be the event that player 1 wins. Suppose that player 1 plays uniformly at random, i.e., plays R,P,S each with equal probability, i.e.,  $\Pr[R_1] = \Pr[P_1] = \Pr[S_1] = 1/3$ . Then *no matter how player 2 plays*,

the probability that player 1 wins is

$$\begin{aligned}\Pr[W_1] &= \Pr[R_1 \wedge S_2] + \Pr[P_1 \wedge R_2] + \Pr[S_1 \wedge P_2] \\ &= \Pr[R_1] \cdot \Pr[S_2] + \Pr[P_1] \cdot \Pr[R_2] + \Pr[S_1] \cdot \Pr[P_2] \\ &= \frac{1}{3}(\Pr[S_2] + \Pr[R_2] + \Pr[P_2]) \\ &= 1/3.\end{aligned}$$

The first equation uses independence, and the final equation uses the fact that the events  $S_2, R_2, P_2$  partition the sample space, so their probabilities must sum to 1. The conclusion is that playing uniformly at random yields a  $1/3$  probability of winning, no matter what strategy the opponent uses. This is a significant improvement over any deterministic strategy, against which an opponent can win with certainty.<sup>71</sup> A similar analysis shows that the probabilities of tying and of losing are both  $1/3$  as well.

The rock-paper-scissors example is illustrative of a more general notion of considering algorithms as games. In this perspective, we view one “player” as choosing an algorithm for a computational problem. A second player known as the *adversary* then chooses an input (of a certain size) to the algorithm. The adversary aims to maximize the number of steps taken by the algorithm on the given input; i.e., to induce the algorithm’s worst-case runtime behavior.

If the first player chooses a deterministic algorithm, then the adversary can find an input that maximizes the number of steps—even if the algorithm happens to be much faster on “most” other inputs. On the other hand, if the first player chooses a *randomized* algorithm—i.e., one that makes some random choices—then the adversary might not be able to induce worst-case behavior, because it cannot predict the algorithm’s random choices. In a good randomized algorithm, the random choices will tend to avoid the worst-case behavior no matter what the input is, although there is still some (ideally very small) probability that it will make “unlucky” choices that lead to poor performance.

## 21.2 Randomized Approximation Algorithms

We can use randomness to design simple algorithms that produce an  *$\alpha$ -approximation* (page 192) *in expectation*, meaning that the expected value of the output is within an  $\alpha$  factor of the value of an optimal solution.

As an example, recall the search version of the 3SAT problem: given a 3CNF (page 169) Boolean formula, find a satisfying assignment for it, if one exists. Since a 3CNF formula is the AND of several clauses, this problem is asking for an assignment that satisfies all of the clauses simultaneously.

Because 3SAT is NP-hard, there is no efficient algorithm for this problem unless  $P = NP$ . So, let us relax the goal, and seek an assignment that *satisfies as many clauses as we can manage*. In other words, we seek an approximation algorithm for the problem of maximizing the number of satisfied clauses.

In what follows, we actually restrict the input to be what we call an “*exact*”-3CNF formula. This is a 3CNF formula with the added restriction that each clause involves three *distinct variables*. (Different clauses can involve the same variable, however.) For example, the clause

$$(x \vee \neg y \vee \neg z)$$

is allowed, but the clause

$$(x \vee \neg y \vee \neg x)$$

is not, since it involves only two distinct variables (the literals  $x$  and  $\neg x$  involve the same variable). Similarly, the clause  $(y \vee y \vee \neg z)$  is not allowed.

The problem of finding an assignment that maximizes the number of satisfied clauses in a given exact-3CNF formula is known as Max-E3SAT, and its decision version is NP-complete. Despite this, there is a very simple randomized

<sup>71</sup> In fact, it can be shown that a winning probability of  $1/3$  is the best any strategy can obtain against an optimal opponent strategy. Moreover, the strategy that chooses a move uniformly at random is the only one that has this property.

algorithm that obtains a  $7/8$ -approximation, in expectation. The algorithm is as follows: assign *random* truth values (true or false) to the formula's variables, uniformly and independently. That is, for each variable, assign it to be true or false each with  $1/2$  probability, independently of all the others.

**Theorem 235** *Let  $\phi$  be an exact-3CNF formula with  $m$  clauses. Then assigning its variables with uniformly random and independent truth values satisfies  $(7/8)$ ths of  $\phi$ 's clauses in expectation, i.e.,*

$$\mathbb{E}[\text{number of satisfied clauses in } \phi] = 7m/8.$$

*In particular, this is a  $7/8$ -approximation algorithm for Max-E3SAT, in expectation.*

The proof of this theorem proceeds by a powerful and widely applicable strategy. First, we define a suitable probability space and a random variable  $X$  that captures the quantity of interest, namely, the number of satisfied clauses. This variable  $X$  typically has a complicated probability distribution that makes it hard to analyze its expectation directly. Instead, we express  $X$  as the *sum* of several simpler *indicator* random variables  $X_i$ . By linearity of expectation,  $\mathbb{E}[X]$  is the sum of the *individual* expectations  $\mathbb{E}[X_i] = \Pr[X_i = 1]$ , where the equality holds because the  $X_i$  are indicators. Finally, we analyze the probabilities of the individual events  $X_i = 1$ , which are simple to understand, and arrive at the conclusion.

**Proof 236** First, suppose that the expected number of satisfied clauses is indeed  $7m/8$  (which we prove below). The optimal number of clauses that can be simultaneously satisfied is  $\text{OPT} \leq m$ , so the expected number of satisfied clauses is  $7m/8 \geq (7/8) \cdot \text{OPT}$ , hence this is indeed a  $7/8$ -approximation algorithm for Max-E3SAT.

We now analyze the expected number of satisfied clauses. The probability space is the set of all assignments to the variables of  $\phi$ , where each assignment is equally likely—i.e., the assignment is chosen uniformly at random.

Now, define  $X$  be the random variable for the number of satisfied clauses in  $\phi$ . That is, for any particular assignment  $\alpha$ , we define  $X(\alpha)$  to be the number of clauses satisfied by  $\alpha$ . Because the clauses of  $\phi$  can share variables in various ways, the probability distribution and expectation of  $X$  can be quite complicated and difficult to analyze directly. Instead, we decompose  $X$  into a sum of much simpler *indicator variables*.

Specifically, for each  $i = 1, \dots, m$  we define an indicator random variable  $X_i$  that indicates whether the  $i$ th clause is satisfied, i.e.,  $X_i = 1$  if so and  $X_i = 0$  if not. Then because  $X$  is the *total* number of satisfied clauses, we have that

$$X = X_1 + \dots + X_m = \sum_{i=1}^m X_i.$$

So, by linearity of expectation (Lemma 231),

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^m X_i\right] = \sum_{i=1}^m \mathbb{E}[X_i].$$

It is important to realize that because the clauses of  $\phi$  can share variables in various ways, the indicators  $X_i$  are typically *not independent*, even pairwise. For example, because clauses  $(x \vee y \vee \neg z)$  and  $(u \vee \neg v \vee x)$  share the variable  $x$ , the former clause being satisfied makes it more likely that the latter clause is also satisfied (i.e., their indicators are positively correlated). Fortunately, linearity of expectation holds even for dependent variables, so the above equation is still valid.

All that remains to analyze  $\mathbb{E}[X_i] = \Pr[X_i = 1]$  (see Lemma 222), i.e., the probability that a uniformly random assignment satisfies the  $i$ th clause. Every clause in the exact-3CNF formula  $\phi$  is the OR of exactly three literals involving distinct variables, e.g.,  $(x \vee \neg y \vee z)$ . The clause fails to be satisfied exactly when *all three* of its literals are *false*. Because the variables are assigned uniformly at random, each literal has probability  $1/2$  of being *false* (regardless of whether the literal is a variable or its negation). And because the literals in clause  $i$  involve distinct

variables, their values are mutually independent. Thus,

$$\begin{aligned}\Pr[X_i = 0] &= \Pr[\text{clause } i \text{ is unsatisfied}] \\ &= \Pr[\text{the 1st and 2nd and 3rd literals of clause } i \text{ are all false}] \\ &= \prod_{j=1}^3 \Pr[\text{the } j\text{th literal of clause } i \text{ is false}] \\ &= (1/2)^3 = 1/8.\end{aligned}$$

Next, because  $X_i$  is an indicator random variable (it is limited to values 0 and 1),

$$\mathbb{E}[X_i] = \Pr[X_i = 1] = 1 - \Pr[X_i = 0] = 7/8.$$

Finally, returning to  $X$  itself,

$$\mathbb{E}[X] = \sum_{i=1}^m \mathbb{E}[X_i] = \sum_{i=1}^m (7/8) = 7m/8.$$

By the averaging argument ([Lemma 225](#)), from [Theorem 235](#) we can further conclude that for any exact-3CNF formula, there *exists* an assignment that satisfies at least  $7/8$ ths of the clauses. Notice that this is a “deterministic” statement—it makes no reference to probability—but we proved it using the tools of probability, and it is not obvious how we could hope to do so without them! (This is an example of the “probabilistic method,” which is a very powerful technique with many applications.)

[Theorem 235](#) says that if we assign variables at random, then *on average*,  $7/8$ ths of the clauses are satisfied. However, this does not say anything about *how likely* it is that a certain fraction of clauses are satisfied. For example, we might want a lower bound on the probability that at least half (or two-thirds, or three-quarters, or 90%) of the clauses are satisfied. Written in terms of the random variable  $X$  representing the number of satisfied clauses, we might hope to get a lower bound on  $\Pr[X \geq c \cdot m]$  for  $c = 1/2$ , or various other  $c \in [0, 1]$ .

Notice that because  $X$  is defined as the number of satisfied clauses, it is non-negative (i.e., it never takes on a negative value). Thus, we can apply Markov’s inequality ([Lemma 227](#)) to obtain that

$$\Pr[X \geq m/2] \leq \frac{7m/8}{m/2} = \frac{7}{4}.$$

Unfortunately, this is a trivial (or “useless”) statement, because the probability bound is  $7/4 > 1$ , and the probability of *any* event is  $\leq 1$ . Furthermore, the probability bound goes in the “wrong direction” from what we want: it says that the probability of satisfying at least half the clauses is *at most* some value, whereas we hope to show that it is *at least* some positive value. An upper bound is not useful to us, because it is consistent with the actual probability being zero. So, even if we replaced  $c = 1/2$  with, say,  $c = 9/10$  to get a nontrivial bound, the statement would not be of the kind we want. We need to use a different tool.

Observe that the number of satisfied clauses  $X$  cannot exceed  $m$ , the total number of clauses in the given formula. So, we can apply the ‘reverse’ Markov inequality ([Lemma 229](#)) to it, with  $b = m$ . Setting  $a = m/2 < b$ , we get that

$$\Pr[X > m/2] \geq \frac{7m/8 - m/2}{m - m/2} = \frac{3m/8}{m/2} = 3/4.$$

That is, a random assignment satisfies more than half the clauses with at least 75 percent probability. More generally, for any  $c < 7/8$ , setting  $a = c \cdot m < b$ ,

$$\Pr[X > c \cdot m] \geq \frac{7m/8 - c \cdot m}{m - c \cdot m} = \frac{7/8 - c}{1 - c} > 0.$$

So, for any  $c < 7/8$  there is a positive probability of satisfying more than a  $c$  fraction of the clauses. However, the ‘reverse’ Markov inequality does not tell us anything useful about the probability of satisfying  $7/8$ ths or more of the clauses.

### Derandomized Algorithm for Max-E3SAT

Remarkably, the randomized algorithm for Max-E3SAT can be *derandomized* into a deterministic algorithm that is guaranteed to satisfy at least  $7/8$ ths of the input formula's clauses. The algorithm considers each variable in turn, computes *conditional expectations* for each potential choice of its truth value, and sets the variable according to a best choice. This is called the [method of conditional probabilities](#)<sup>72</sup> (or the *method of conditional expectations*). It is notable that even though the algorithm is entirely deterministic, it is heavily inspired by the randomized algorithm, and it is not clear how it could have been discovered without taking a probabilistic perspective.

Let  $A, B$  be events with  $\Pr[B] > 0$ . The *conditional probability*  $\Pr[A|B]$  captures the probability that event  $A$  occurs, given that event  $B$  occurs. It is defined as

$$\begin{aligned}\Pr[A|B] &= \frac{\Pr[A \cap B]}{\Pr[B]} \\ &= \frac{\sum_{\omega \in A \cap B} \Pr[\omega]}{\sum_{\omega \in B} \Pr[\omega]}.\end{aligned}$$

Next, the *conditional expectation* of a random variable  $X$  conditioned on event  $B$ , which captures the “average” value of  $X$  given that event  $B$  occurs, is defined as

$$\mathbb{E}[X|B] = \sum_{v \in X(\Omega)} v \cdot \Pr[X = v|B].$$

It can be shown that

$$\mathbb{E}[X] = \Pr[B] \cdot \mathbb{E}[X|B] + \Pr[\bar{B}] \cdot \mathbb{E}[X|\bar{B}].$$

Since  $\mathbb{E}[X]$  is the weighted average of  $\mathbb{E}[X|B]$  and  $\mathbb{E}[X|\bar{B}]$ , by an averaging argument, at least one of  $\mathbb{E}[X|B]$  or  $\mathbb{E}[X|\bar{B}]$  is  $\geq \mathbb{E}[X]$ .

We now apply this to Max-E3SAT. Let  $\phi$  be an exact-3CNF formula with  $m$  clauses and  $n$  variables  $x_1, \dots, x_n$ . Let  $X$  be the number of satisfied clauses for a random assignment. [Theorem 235](#) established that  $\mathbb{E}[X] = 7m/8$ . Now consider the events  $x_1 = 0$  and  $x_1 = 1$ . These are complementary events, so by the reasoning above, at least one of  $\mathbb{E}[X|x_1 = 0]$  or  $\mathbb{E}[X|x_1 = 1]$  is  $\geq 7m/8$ . This generalizes to the case where variables  $x_1, \dots, x_{i-1}$  have been set so that the conditional expectation of  $X$  remains  $\geq 7m/8$ , and we consider the two cases  $x_i = 0$  and  $x_i = 1$ . Thus, we can deterministically build an assignment  $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$  one entry at a time, by doing the following for each variable  $x_i$  in sequence, i.e., for  $i = 1, \dots, n$ :

- Compute  $\mathbb{E}[X|x_1 = \alpha_1, \dots, x_{i-1} = \alpha_{i-1}, x_i = 0]$  and  $\mathbb{E}[X|x_1 = \alpha_1, \dots, x_{i-1} = \alpha_{i-1}, x_i = 1]$ . In other words, given the partial assignment so far, consider both  $x_i = 0$  and  $x_i = 1$  and compute the resulting conditional expectations of  $X$ .
- Compute the conditional expectations for partial assignments using linearity of expectation over the clauses, and by considering the values of any variables that have already been set. For example, consider the clause  $(x_1 \vee \neg x_3 \vee x_7)$ . If we have set  $x_1 = 1$ , then the expectation for this clause is 1, regardless of how the other variables have been set (or not). But if we have set  $x_1 = 0$  and the other two variables are not yet set, the expectation for this clause is  $3/4$ .
- If setting  $x_i = 0$  produces a larger expectation, then fix  $\alpha_i = 0$ , otherwise fix  $\alpha_i = 1$ .

At each step, at least one of the two choices must have expectation at least  $7m/8$ , by the averaging argument. Thus, in the end, the full assignment also has expectation at least  $7m/8$ . But at that point, there are no more random choices to make (i.e., variables to set), so the full assignment must satisfy at least  $7m/8$  clauses.

The full algorithm is as follows:

**Input:** an exact-3CNF formula

**Output:** an assignment that satisfies at least 7/8ths of the formula's clauses

```

function APPROXEXACT3SAT( $\phi$ )
   $A$  = an empty assignment for  $\phi$ 
  for all variables  $x_i$  in  $\phi$  do
    for all Boolean values  $v = 0, 1$  do
       $\mu_v$  = EXPECTEDSATISFIED( $\phi, A + (x_i = v)$ )
      if  $\mu_0 > \mu_1$  then
         $A = A + (x_i = 0)$ 
      else
         $A = A + (x_i = 1)$ 
  return  $A$ 
```

**Input:** an exact 3CNF formula and a partial assignment for it

**Output:** the expected number of satisfied clauses, over a uniformly random completion of the assignment

```

function EXPECTEDSATISFIED( $\phi, A$ )
   $\mu = 0$ 
  for all clauses  $C_j$  in  $\phi$  do
    if  $C_j$  is satisfied by  $A$  then
       $\mu = \mu + 1$ 
    else
       $k$  = the number of unassigned variables (by  $A$ ) in  $C_j$ 
       $\mu = \mu + (1 - 1/2^k)$ 
  return  $\mu$ 
```

The EXPECTEDSATISFIED subroutine uses linearity of expectation to compute the expected number of satisfied clauses, for a uniformly random completion of the given partial assignment. Let  $X$  be the total number of satisfied clauses, and let  $X_j$  be the indicator random variable for whether clause  $C_j$  is satisfied. Then if  $C_j$  is already satisfied by the given partial assignment  $A$ ,  $\mathbb{E}[X_j|A] = 1$ . If  $C_j$  is not yet satisfied, the probability of it being satisfied by a random assignment of the remaining variables depends on how many unset variables  $C_j$  has. By the same reasoning as for the randomized algorithm, this is  $1 - (1/2)^k$  if there are  $k$  unset variables. Thus,  $\mathbb{E}[X_j|A] = 1 - (1/2)^k$ .

**Exercise 237** Recall the *max-cut* (page 196) problem. The following is a randomized algorithm that outputs a random cut in a given undirected graph (which has no self-loop edge, without loss of generality):

```

function RANDOMCUT( $G = (V, E)$ )
   $S = \emptyset$ 
  for all  $v \in V$  do
    add  $v$  to  $S$  with probability  $1/2$ 
  return  $S$ 
```

Recall that  $C(S)$  is the set of crossing edges for the cut  $S$ , i.e., those that have exactly one endpoint in  $S$ . The size of the cut is  $|C(S)|$ .

- Prove that *in expectation*, this algorithm outputs a  $1/2$ -approximation of a maximum cut in the given undirected graph. That is, the expected size of the returned cut is at least half the size of a maximum cut.

**Hint:** Use linearity of expectation to show that in expectation, half of the edges of the graph are in  $C(S)$ . Similarly, give an upper bound on the size of a maximum cut (also in terms of the number of edges in the graph).

- Conclude that, for any undirected graph  $G = (V, E)$ , there exists a cut of size at least  $|E|/2$ . (By contrast,

<sup>72</sup> [https://en.wikipedia.org/wiki/Method\\_of\\_conditional\\_probabilities](https://en.wikipedia.org/wiki/Method_of_conditional_probabilities)

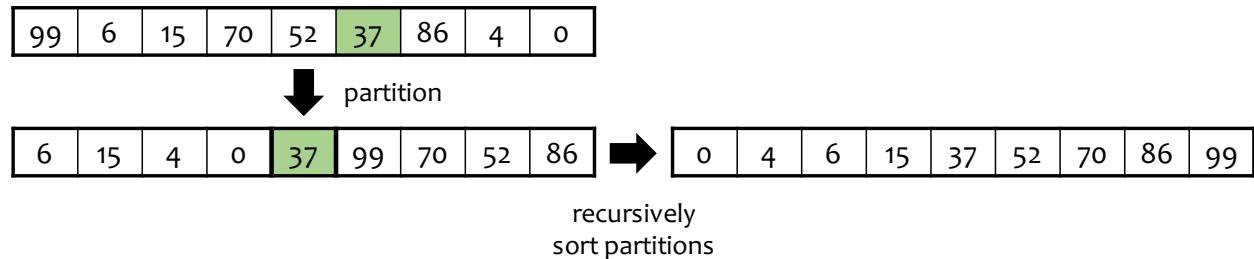
in Theorem 203 we saw an efficient deterministic algorithm that finds such a cut, with certainty.)

## 21.3 Quick Sort

Another example of an algorithm that takes advantage of randomness is *quick sort*. This is a recursive algorithm that sorts an array as follows:

1. First, it chooses some element of the array as the *pivot*.
2. Then, it *partitions* the array around the pivot, by comparing the pivot to every other element, placing all the smaller elements to the “left” of the pivot—not necessarily in sorted order—and all the larger ones to the “right” of the pivot. Without loss of generality, we assume that the array elements are distinct, by breaking ties according to the elements’ positions in the array.
3. Finally, it recursively sorts the two subarrays to the left and right of the pivot, which results in the entire array being sorted.

The following is an illustration of an execution of quick sort, with the recursive work elided:



The running time of quick sort is dominated by the number of comparisons between pairs of elements. Once a pivot is selected, partitioning an array of  $n$  elements compares every other element to the pivot. This is a total of  $n - 1 = \Theta(n)$  comparisons of non-recursive work, which is followed by the recursive calls on the two subarrays on each side of the pivot. If selecting the pivot takes  $O(n)$  time and can guarantee that the two subarrays are “balanced,” i.e., approximately equal in size, then the recurrence for the total number of comparisons (and hence the overall running time) is

$$T(n) = 2T(n/2) + \Theta(n).$$

This is the same recurrence as for *merge sort* (page 13), and it has the closed-form solution  $T(n) = \Theta(n \log n)$ . However, quick sort tends to be faster in practice than merge sort and other deterministic  $O(n \log n)$  sorting algorithms, in large part because it is fast and easy to implement *in place*—i.e., reordering the array within its own memory space, using only a small amount of auxiliary memory. (By comparison, the standard merge subroutine from merge sort makes a full copy of the array with each call.) Thus, variants of quick sort are widely used in practice.

One way to guarantee a balanced partition is to use the *median* element as the pivot, which can be found in linear time<sup>73</sup>. However, this subroutine is rather complicated to implement, and its linear runtime has a fairly large hidden constant factor.

A much simpler and more practical solution is to choose a *uniformly random* array element as the pivot. The pseudocode is as follows.

### Algorithm 238 (Randomized Quick Sort)

```
function RANDQUICKSORT( $A[1, \dots, n]$ )
  if  $n = 1$  then return  $A$ 
   $p$  = an index chosen uniformly at random from  $\{1, \dots, n\}$ 
```

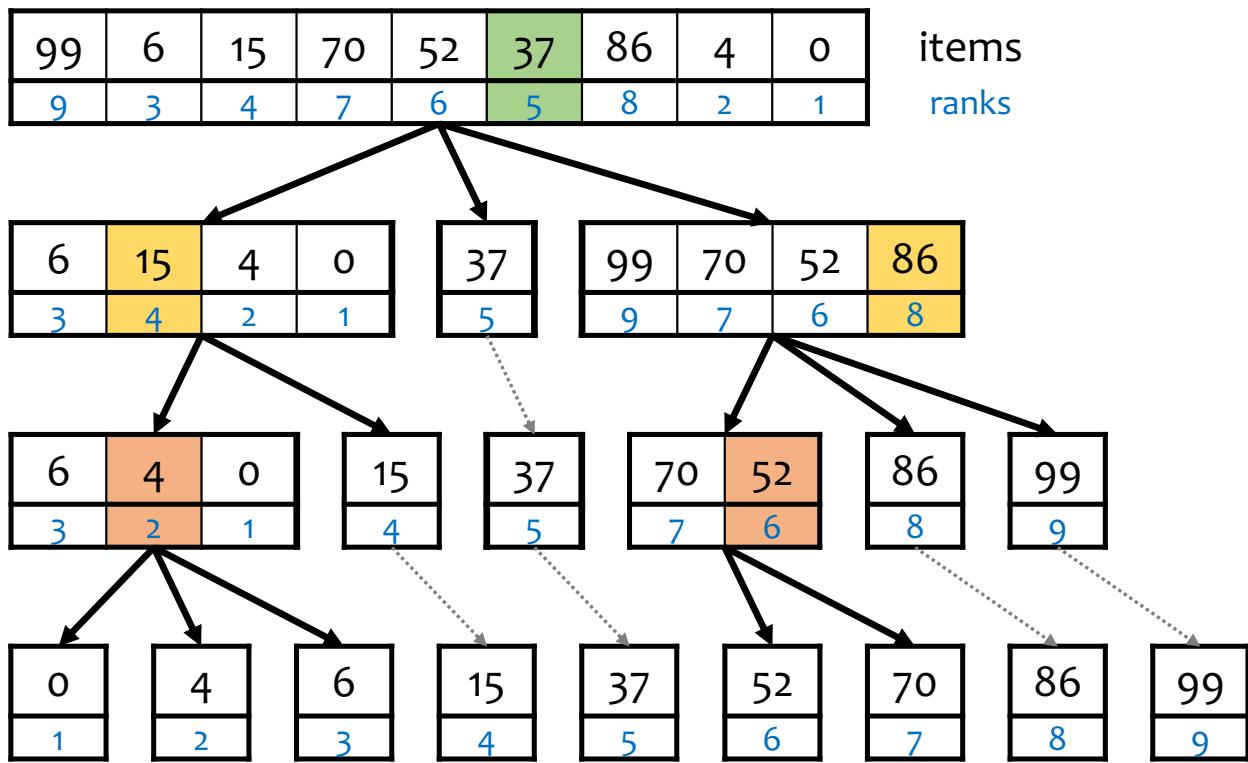
<sup>73</sup> [https://en.wikipedia.org/wiki/Median\\_of\\_medians](https://en.wikipedia.org/wiki/Median_of_medians)

```

 $(L, R) = \text{PARTITION}(A, p)$ 
 $\text{return RANDQUICKSORT}(L) + A[p] + \text{RANDQUICKSORT}(R)$ 
function PARTITION( $A[1, \dots, n], p$ )
    initialize empty arrays  $L, R$ 
    for  $i = 1$  to  $n$  do
        if  $i \neq p$  and  $A[i] < A[p]$  then
             $L = L + A[i]$ 
        else if  $i \neq p$  then
             $R = R + A[i]$ 
    return  $(L, R)$ 

```

The following figure illustrates a possible execution of RANDQUICKSORT on an example input array, with the random choices of pivot elements highlighted, and the *rank* of each element displayed below it. The rank of an element is the number of elements in the array (including itself) that are less than or equal to it. So, the smallest element has rank 1, the next-smallest has rank 2, etc.



The running time of RANDQUICKSORT is a random variable that depends in a complicated way on the particular random choices of pivots made throughout the algorithm. Roughly speaking, choices of pivots that yield “unbalanced” subarrays tend to yield larger running times than ones that yield more balanced subarrays. How can we hope to analyze this very complicated probability experiment? As we did above, we express the running time as a sum of much simpler *indicator variables*, apply linearity of expectation, and then analyze these indicators individually. The rest of this section is devoted to proving the following theorem.

**Theorem 239** *On any array of  $n$  elements, the expected running time of RANDQUICKSORT is  $O(n \log n)$ .*

Recall that the running time of RANDQUICKSORT is dominated by the number of comparisons of pairs of array elements. First observe that if a certain pair of elements is ever compared, then it is never compared again. This is because at the time of comparison, one of the elements is the pivot, which is never compared with anything after partitioning around it is complete. So, the total number of comparisons done by RANDQUICKSORT is the total number of distinct pairs of

elements that are ever compared.

Accordingly, define the random variable  $X$  to be the total number of comparisons during the entire execution of quick sort on an arbitrary array of  $n$  elements. In addition, for any ranks  $1 \leq i < j \leq n$ , define  $X_{ij}$  to be the indicator random variable that indicates whether the pair of elements having ranks  $i, j$  are ever compared during the execution, i.e.,  $X_{ij} = 1$  if this pair is ever compared, and  $X_{ij} = 0$  otherwise. By the observations from the previous paragraph,

$$X = \sum_{1 \leq i < j \leq n} X_{ij} = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}.$$

Therefore, by linearity of expectation (Lemma 231), the expected total number of comparisons is

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ij}].$$

So, for every  $i < j$  it just remains to analyze the individual expectations  $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1]$  *in isolation* (where the equality holds by Lemma 222), and sum them. (As above, the random variables  $X_{ij}$  are highly correlated in difficult-to-understand ways, but this does not matter for linearity of expectation.)

Under what conditions are the elements of ranks  $i < j$  ever compared, and what is the probability  $\Pr[X_{ij} = 1]$  that this occurs? By inspection of Algorithm 238, we see that elements are compared only in the PARTITION subroutine. In particular, the only comparisons made during a call to PARTITION are between the selected pivot element and all the other elements in the input subarray. This has two important implications:

- When a subarray is input to PARTITION, none of its elements have been compared with each other yet.
- In the subarray input to PARTITION, if one element is less than the pivot and another is greater than the pivot, then these two elements are placed on opposite sides of the pivot, and thus are never compared with each other (because the recursive calls operate completely separately on the two sides).

Therefore, the two elements  $a < b$  having ranks  $i < j$  are compared if and only if the *first* pivot chosen from the range  $[a, b]$  is either  $a$  or  $b$ . To elaborate: for as long as pivots are chosen from outside this range,  $a$  and  $b$  remain in the same subarray, and have not yet been compared. Then, once an element from this range is chosen as a pivot:

- it is either  $a$  or  $b$ , in which case  $a$  and  $b$  are compared with each other, or
- it is an element strictly between  $a$  and  $b$ , in which case  $a$  and  $b$  are placed in different subarrays and never compared with each other.

$a$		$b$
0	4	6
1	2	3
15	4	5
37	5	6
52	6	7
70	7	8
86	8	9
99	9	

$i$                                    $j$

Therefore,  $\mathbb{E}[X_{ij}]$  is the probability that the first pivot whose rank is between  $i$  and  $j$  (inclusive) has rank either  $i$  or  $j$ . There are  $j - i + 1$  elements in this range. Each pivot is chosen uniformly at random from the subarray input to PARTITION, and as argued above, when the first pivot from this range is selected, the entire range is part of the subarray. So, given that the pivot is in the range, it is *uniformly random* over the range, i.e., it is equally likely to be any of the  $j - i + 1$  elements in the range. (This statement can be formalized using *conditional probability*.) Because exactly two of these possible choices make  $X_{ij} = 1$ , we have that

$$\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j - i + 1}.$$

Resuming from what we showed above,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= 2 \sum_{i=1}^{n-1} \sum_{t=2}^{n-i+1} \frac{1}{t}.\end{aligned}$$

The quantity  $\sum_{t=2}^k 1/t$  is part of the harmonic series<sup>74</sup>, and is known to be less than  $\ln(k)$ . Thus,

$$\mathbb{E}[X] < 2 \sum_{i=1}^{n-1} \ln(n-i+1) \leq 2 \sum_{i=1}^{n-1} \ln(n) = 2(n-1) \ln(n) = O(n \log n),$$

which completes the proof.

**Exercise 240** Although `RANDQUICKSORT` runs in *expected* time  $O(n \log n)$  on an array of  $n$  elements, it can potentially run in time  $\Omega(n^2)$ , due to an unlucky choice of random pivots.

Prove that for any constant  $c > 0$ ,

$$\lim_{n \rightarrow \infty} \Pr[\text{RANDQUICKSORT on an } n\text{-element array takes } \geq cn \log^2 n \text{ steps}] = 0.$$

That is, as the array size grows large, randomized quick sort is very unlikely to run in even  $\Omega(n \log^2 n)$  time.

**Hint:** This follows immediately from [Theorem 239](#) and Markov's inequality ([Lemma 227](#)), without needing to understand any of the analysis of `RANDQUICKSORT`.

## 21.4 Skip Lists

Randomness can also be used to obtain very *simple* data structures that provide excellent performance in expectation. As an example, here we consider a structure called a “skip list”, which implements a *dictionary* abstract data type. A dictionary is a structure that stores data as *key-value* pairs, and supports inserting, deleting, and finding (looking up) data by its associated key. For example, in a database, students’ academic records might be keyed by their unique ID numbers.

For simplicity, we focus on storing just the keys themselves, since the associated data can be attached to them using pointers. We assume without loss of generality that keys can be *ordered* in some consistent way. (For example, in the absence of a “natural” ordering, one can use lexicographic order on the string representations of the keys.)

There are many ways to implement a dictionary abstraction using a variety of underlying data structures, such as arrays, linked lists, hash tables, and binary search trees. However, simple deterministic implementations have worst-case running times that are *linear* in the number of items in the dictionary, for some or all operations. For example, if we insert elements  $1, 2, \dots, n$  into a sorted linked list or non-balancing binary search tree, most of the insertions take  $\Theta(n)$  time each. Similarly, finding a key takes  $\Theta(n)$  time in the worst case, because we may need to traverse most or all of the list.

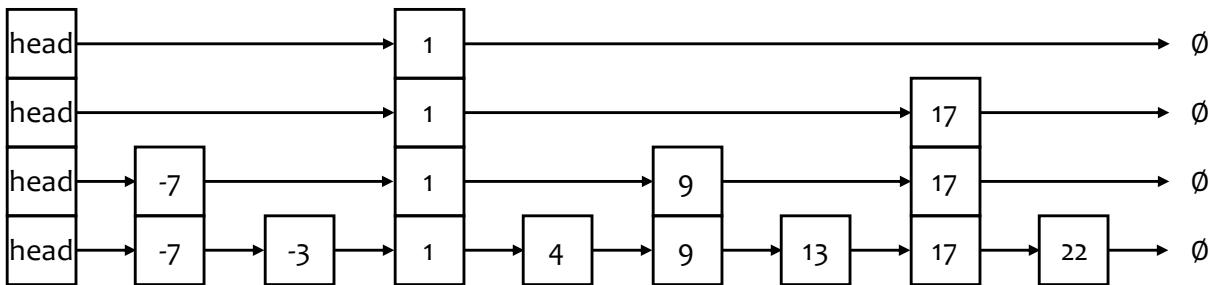
One common way to avoid such poor performance is to use a more advanced data structure that does some “rebalancing” operations to ensure  $O(\log n)$  worst-case runtimes for all operations, regardless of which elements are inserted or

<sup>74</sup> [https://en.wikipedia.org/wiki/Harmonic\\_series\\_\(mathematics\)](https://en.wikipedia.org/wiki/Harmonic_series_(mathematics))

deleted. There are a variety of such structures, like [AVL trees<sup>75</sup>](#), red-black trees<sup>76</sup>, scapegoat trees<sup>77</sup>, and so on. However, these structures can be complicated and difficult to implement correctly, and can have significant hidden constant factors in their runtimes and memory overheads.

Here we investigate an alternative, very *simple* data structure called a “*skip list*”, which uses randomness to obtain *expected*  $O(\log n)$  runtimes for all dictionary operations. A skip list is essentially a linked list with *multiple levels*, where each level has about half the elements of the level below it.<sup>78</sup>

Like in a subway or highway system, skip lists can be thought of as having “express lanes” that make it possible to reach a desired element more quickly by “skipping over” many undesired elements at a time—just as a subway’s express train can go faster by skipping past many regular stops. A skip list, however, typically has *several* levels of “express lanes,” each one able to skip over more elements than the one below it. Moreover, the choice of which elements appear on each level, and which are skipped, is made *at random*. This tends to keep the data structure sufficiently “balanced” to support fast operations, in expectation.<sup>79</sup>



The above diagram illustrates a possible state of a skip list. Every level has a sentinel “head” and a terminal null pointer; for convenience, their values are defined to be smaller and larger (respectively) than every element. The bottom level has *all* of the elements, in a sorted linked list. Every other level has some *subset* of the elements in the level below it, also in a sorted linked list. In addition, each element above the bottom level also points to its duplicate in the level below it; these downward pointers are illustrated by stacking the duplicate elements. Therefore, from any non-terminal position in the skip list, we may go “rightward” to the next-larger element in the same level, or (when not in the bottom level) “downward” to the same element at the next-lowest level.<sup>80</sup>

To *search* for an element  $e$ , we start at the sentinel head of the top level. We maintain the invariant that we are always located at some element  $a < e$  in some level, and repeat the following loop:

1. Consider the successor element  $a'$  of  $a$  in the current level.
2. If  $a' < e$ , move to  $a'$  (setting  $a = a'$ ) and repeat.
3. Otherwise (i.e.,  $a' \geq e$ ), if we are not in the bottom level, move downward (to the same element  $a$ ) and repeat; else terminate the loop.

At this point, we are in the bottom level, at the largest element smaller than  $e$ . If its successor is the desired element  $e$ , output it; otherwise, report that  $e$  is not in the structure.

Note that as an optimization, if we ever find that  $a' = e$  in some level, we can immediately return it, instead of always descending to the bottom level. However, for defining the insertion and deletion algorithms, and for the probabilistic analysis below, it is more convenient to define the search so that it ultimately ends in the bottom level.

For example, the following diagram illustrates the search for element 0 in the above skip list. The search terminates in failure, since in the bottom level we end up at -3, whose successor is 1.

<sup>75</sup> [https://en.wikipedia.org/wiki/AVL\\_tree](https://en.wikipedia.org/wiki/AVL_tree)

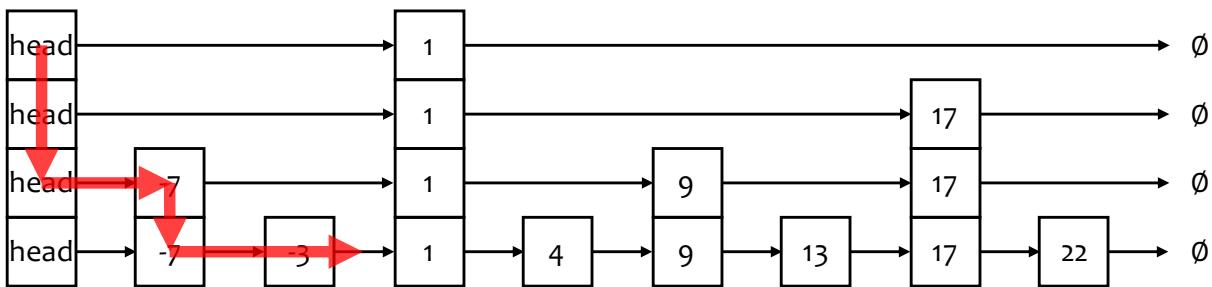
<sup>76</sup> [https://en.wikipedia.org/wiki/Red%E2%80%93black\\_tree](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)

<sup>77</sup> [https://en.wikipedia.org/wiki/Scapegoat\\_tree](https://en.wikipedia.org/wiki/Scapegoat_tree)

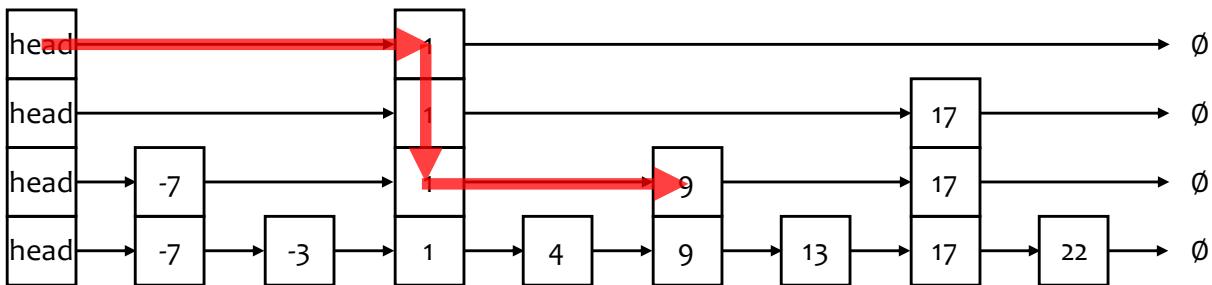
<sup>78</sup> Ratios other than 1/2 may be used instead, trading off the space and time overheads.

<sup>79</sup> In fact, with more work it can be shown that a skip list remains sufficiently balanced with *high probability*, not just in expectation.

<sup>80</sup> In a real implementation, we would keep just one copy of each element, with an array of pointers to its successors in each of the levels in which it appears.



The following diagram illustrates the successful search for element 9 (optimized to return as soon as it is found, above the bottom level):



To *delete* an element  $e$  from the structure, we first search for it. If  $e$  is not in the structure, there is nothing more to do. Otherwise, the search finds  $e$  in the bottom level, and along the way it also finds  $e$ 's predecessor in every level in which  $e$  appears. (These are the elements at which the search moves downward.) So, we just delete  $e$  from each of those levels, by making each predecessor point to the corresponding successor of  $e$  in that level. Notice that the running time of deletion is within a constant factor of the running time of the search.

To *insert* an element  $e$ , we first search for  $e$ . If it is already in the structure, then there is nothing more to do. Otherwise, the search terminates in the bottom level, at the largest element  $a < e$ ; along the way, it also finds the largest element smaller than  $e$  in each level.

We first insert  $e$  as the successor of  $a$  in the bottom level. Then, we determine *at random* which higher levels the element  $e$  will occupy, using the following loop:

1. Flip a fair coin.
2. If it comes up heads, insert  $e$  in the next level up (as the successor of the largest element smaller than  $e$ ), and repeat.
3. Otherwise, stop.

In other words, we insert  $e$  into one higher level for each head that comes up, until tails comes up. Note that there is no upper limit to how many levels we might insert  $e$  into, and we create new levels as needed.

Observe that the number of copies of the inserted element is a random variable, which equals the total number of coin flips until tails comes up (inclusive). So, this random variable follows the [geometric distribution](#)<sup>81</sup> with success probability  $1/2$ , which has expectation 2. Thus, the runtime of insertion is, in expectation, only a constant more than the runtime of the search.

Based on the above insertion process, we can also bound the expected size (memory usage) of a skip list. If there are  $n$  distinct elements, then by linearity of expectation on the number of copies of each element, the expected total number of copies is  $2n$ . So, the expected total memory usage is  $O(n)$ .

Finally, based on what we have already seen, to determine the expected runtime of any of the operations (insert, delete, search), it just remains to analyze the expected runtime of a search. We have the following theorem.

<sup>81</sup> [https://en.wikipedia.org/wiki/Geometric\\_distribution](https://en.wikipedia.org/wiki/Geometric_distribution)

**Theorem 241** Assume that all the operations (insert, delete, search) performed on a skip list are independent of the internal random choices made during insertions. Then in a skip list with  $n$  distinct elements, the expected running time of a search is  $O(\log n)$ .

Before proving the theorem, we discuss the meaning and importance of the assumption, that the sequence of operations is independent of the internal random choices. Without this assumption, a skip list may have very poor performance. For example, if an adversarial user is able to learn which elements appear above the bottom level, then the user could just delete all such elements from the structure. This makes the skip list just an ordinary linked list, with its associated  $\Theta(n)$  runtimes. But notice that in this scenario, the sequence of operations depends on the skip list's internal random choices, because the deleted elements are exactly those that are randomly promoted above the bottom level. By contrast, under the independence assumption, we can treat every element as if it has just been inserted and promoted at random according to the above process.

To prove the theorem, we first show the following useful lemma.

**Lemma 242** Under the independence assumption from Theorem 241, in a skip list with  $n$  distinct elements, the expected height (i.e., number of levels above the bottom level) is less than  $\log_2 n + 2$ .

**Proof 243** We first analyze the expected number of elements on each level, and use this to determine the expected height of the structure.

Define  $X_{ij}$  to be the indicator random variable that indicates whether the  $j$ th element appears on level  $i$ , where  $i = 0$  corresponds to the bottom level. Then by the insertion rule and the independence assumption,

$$\Pr[X_{ij} = 1] = 1/2^i .$$

This is because an element appears on level  $i$  if at least  $i$  heads come up when inserting it, and the coin flips are fair and independent. Thus,  $\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = 1/2^i$ .

Now define random variable  $n_i$  to be the number of (non-sentinel) elements on level  $i$ ; this is just the sum of the indicators  $X_{ij}$ . So, by linearity of expectation and the formula for geometric series, the expected number of elements on level  $i$  is

$$\mathbb{E}[n_i] = \mathbb{E}\left[\sum_{j=1}^n X_{ij}\right] = \sum_{j=1}^n \mathbb{E}[X_{ij}] = \sum_{j=1}^n \frac{1}{2^i} = \frac{n}{2^i} .$$

(For example, the expected number of elements on level 1 is  $n/2$ , on level 2 is  $n/4$ , etc.)

Now, we analyze the expected height of the structure, i.e., the number of levels above the bottom one that have at least one element. For intuition, the expected number of elements on level  $i$  is less than 1 when

$$\mathbb{E}[n_i] = \frac{n}{2^i} < 1 \iff 2^i > n \iff i > \log_2 n ,$$

so we should expect roughly  $\ell(n) = \log_2 n$  levels.

To show this properly, define  $H_i$  to be the indicator random variable that indicates whether level  $i$  has at least one element, i.e., whether  $n_i \geq 1$ . Because  $n_i$  is non-negative, by Markov's inequality and the fact that  $n = 2^{\ell(n)}$ ,

$$\mathbb{E}[H_i] = \Pr[n_i \geq 1] \leq \mathbb{E}[n_i]/1 = \frac{n}{2^i} = \frac{1}{2^{i-\ell(n)}} .$$

Notice that this upper bound on  $\mathbb{E}[H_i]$  is nontrivial only for  $i > \ell(n)$ , because the expectation of any indicator random variable is at most 1. For  $i \leq \ell(n)$  we have the trivial but tighter bound  $\mathbb{E}[H_i] \leq 1$ .

Now, the total height  $H$  of the skip list is

$$H = \sum_{i=1}^{\infty} H_i ,$$

so by linearity of expectation, the above bounds on  $\mathbb{E}[H_i]$ , and the formula for a geometric series, the expected height is

$$\begin{aligned} \mathbb{E}[H] &= \mathbb{E}\left[\sum_{i=1}^{\infty} H_i\right] \\ &= \sum_{i=1}^{\infty} \mathbb{E}[H_i] \\ &= \sum_{1 \leq i \leq \ell(n)} \mathbb{E}[H_i] + \sum_{i > \ell(n)} \mathbb{E}[H_i] \\ &\leq \sum_{1 \leq i \leq \ell(n)} 1 + \sum_{i > \ell(n)} \frac{1}{2^{i-\ell(n)}} \\ &< \ell(n) + \sum_{k=0}^{\infty} \frac{1}{2^k} \\ &= \log_2 n + 2 . \end{aligned}$$

Now we can prove [Theorem 241](#), which says that the expected running time of a search in a skip list of  $n$  distinct elements is  $O(\log n)$ .

**Proof 244** The running time of the search for an element  $e$  is proportional to the total number of downward and rightward moves during the search, starting from the topmost sentinel head. The number of downward moves is exactly the height of the structure, whose expectation we analyzed above in [Lemma 242](#). So, it remains to analyze the expected number of rightward moves. The simplest way to do this rigorously is to consider the search path in *reverse*.

The end of the search path is the largest value  $a < e$  in the bottom level, and hence in the entire structure as well. Suppose that this  $a$  is not the sentinel head, otherwise there is no rightward move in the entire search. The search moved rightward to this  $a$  only if  $a$  is *not* present in the next level up. Indeed, if  $a$  were in the next level up, then the prior step of the search must have been downward from that copy of  $a$ , because the search reaches the largest value less than  $e$  on each level.

By definition of the insertion process (and the independence assumption), with probability  $1/2$ , the first coin toss when  $a$  was inserted came up tails, in which case  $a$  is not present in the next level up, and so the prior step was rightward. In this case, the exact same reasoning applies to the step prior to that one, and so on, until we reach an element that either is in the next level up or is the sentinel head. So, in the bottom level, the number of rightward moves  $R_0$  is the number of independent fair coin tosses (out of  $\leq n$  tosses) that come up tails before the first heads comes up. This is bounded by one less than a geometric random variable with success probability  $1/2$ , so  $\mathbb{E}[R_0] < 1$ .

The exact same reasoning applies to the number of rightward moves on *every* level. That is, the last value visited on level  $i$  is the largest value  $a < e$  on that level. If  $a$  is not the sentinel head, then the prior step was rightward only if  $a$  is not in the next level up, which holds with probability  $1/2$  by definition of the insertion process, and so on. So, defining random variable  $R_i$  to be the number of rightward moves on level  $i$ , we have that  $\mathbb{E}[R_i] < 1$  for all  $i$ .

Separately, the number of rightward moves on level  $i$  cannot exceed the total number of (non-sentinel) elements  $n_i$  on level  $i$ . So,  $\mathbb{E}[R_i] \leq \mathbb{E}[n_i] = n/2^i$ , as shown in the proof of [Lemma 242](#) above. This bound on  $\mathbb{E}[R_i]$  is

tighter than the one obtained above when  $n/2^i < 1$ , or equivalently, when  $i > \ell(n) = \log_2(n)$ .

Finally, let  $H$  be the height of the structure, which is the number of downward moves in the search, and let  $R$  be the total number of rightward moves, which is the sum of all the  $R_i$  for  $i \geq 0$ . Then by linearity of expectation, the above bounds on  $\mathbb{E}[R_i]$ , and Lemma 242 (and proceeding similarly to its proof), the expected total number of moves in the search is

$$\begin{aligned}\mathbb{E}[H + R] &= \mathbb{E}[H] + \mathbb{E}\left[\sum_{i=0}^{\infty} R_i\right] \\ &= \mathbb{E}[H] + \sum_{i=0}^{\infty} \mathbb{E}[R_i] \\ &= \mathbb{E}[H] + \sum_{0 \leq i \leq \ell(n)} \mathbb{E}[R_i] + \sum_{i > \ell(n)} \mathbb{E}[R_i] \\ &< \mathbb{E}[H] + \sum_{0 \leq i \leq \ell(n)} 1 + \sum_{i > \ell(n)} \frac{n}{2^i} \\ &< \mathbb{E}[H] + (\ell(n) + 1) + 2 \\ &< 2 \log_2 n + 5.\end{aligned}$$

Therefore, the expected running time of the search is  $O(\log n)$ , as claimed. □

---

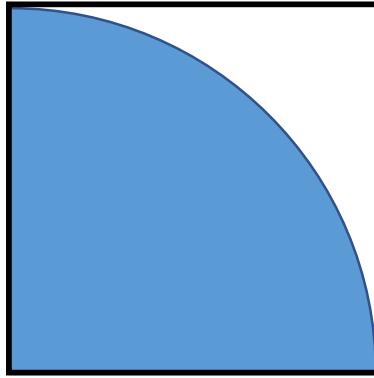
CHAPTER  
TWENTYTWO

---

## MONTE CARLO METHODS AND CONCENTRATION BOUNDS

Some algorithms rely on repeated trials to compute an approximation of a result. Such algorithms are called *Monte Carlo methods*, which are distinct from *Monte Carlo algorithms* (page 284) – a Monte Carlo *method* does repeated sampling of a probability distribution, while a Monte Carlo *algorithm* is an algorithm that may produce the wrong result within some bounded probability. The former are commonly approximation algorithms for estimating a quantity, while the latter are exact algorithms that sometimes produce the wrong result. As we will see (page 287), there is a connection: repeated sampling (the strategy of a Monte Carlo method) can be used to amplify the probability of getting the correct result from a Monte Carlo algorithm.

As an example of a Monte Carlo method, we consider an algorithm for estimating the value of  $\pi$ , the area of a unit circle (a circle with a radius of one). If such a circle is located in the plane, centered at the origin, its top-right quadrant is as follows:



This quadrant falls within the square interval between  $(0, 0)$  and  $(1, 1)$ . The area of the quadrant is  $\pi/4$ , while the area of the interval is 1. Thus, if we choose a random point in this interval, the probability that it lies within the quadrant of the circle is  $\pi/4$ <sup>82</sup>. This motivates the following algorithm for estimating the value of  $\pi$ :

```
function ESTIMATEPi(n)
    count = 0
    for i = 1 to n do
        x, y = values in [0, 1] chosen uniformly and independently at random
        if  $x^2 + y^2 \leq 1$  then
            count = count + 1
    return 4 · count/n
```

The algorithm randomly chooses points between  $(0, 0)$  and  $(1, 1)$ , counting how many of them fall within the unit circle, which is when the point's distance from the origin is at most one. We expect this number to be  $\pi/4$  of the samples, so the algorithm returns four times the ratio of the points that fell within the circle as an estimate of  $\pi$ .

---

<sup>82</sup> This follows from applying the tools of continuous probability, which we will not discuss in any detail in this text.

We formally show that the algorithm returns the value of  $\pi$  in expectation. Let  $X$  be a random variable corresponding to the value returned, and let  $Y_i$  be an indicator random variable that is 1 if the  $i$ th point falls within the unit circle. As argued previously, we have:

$$\Pr[Y_i = 1] = \pi/4.$$

We also have that

$$X = 4/n \cdot (Y_1 + \dots + Y_n).$$

This leads to the following expected value for  $X$ :

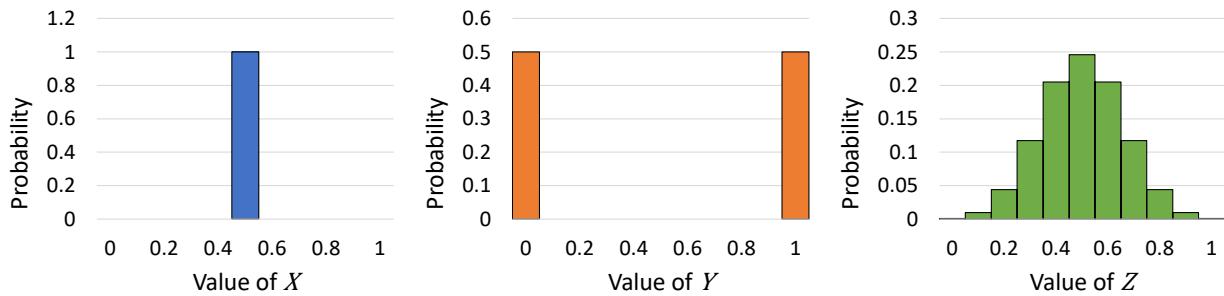
$$\begin{aligned}\mathbb{E}[X] &= 4/n \cdot (\mathbb{E}[Y_1] + \mathbb{E}[Y_2] + \dots + \mathbb{E}[Y_n]) \\ &= 4/n \cdot (\Pr[Y_1 = 1] + \Pr[Y_2 = 1] + \dots + \Pr[Y_n = 1]) \\ &= 4/n \cdot (\pi/4 + \pi/4 + \dots + \pi/4) \\ &= \pi.\end{aligned}$$

The expected value of the algorithm's output is indeed  $\pi$ .

When estimating  $\pi$ , how likely are we to actually get a result that is close to the expected value? While we can apply Markov's inequality, the bound we get is very loose, and it does not give us any information about how the number of samples affects the quality of the estimate. The *law of large numbers* states that the actual result converges to the expected value as the number of samples  $n$  increases. But how fast does it converge? There are many types of *concentration bounds* that allow us to reason about the deviation of a random variable from its expectation; Markov's inequality is just the simplest one. *Chebyshev's inequality* (page 229) is another simple bound that makes use of more information about a random variable, namely its *variance*. *Chernoff bounds* are yet another tool. There are multiple variants of Chernoff bounds, including the *multiplicative form* (page 274) and the additive *Hoeffding bounds* (page 234).

## 22.1 Variance and Chebyshev's Inequality

The expectation of a random variable gives us one piece of information about its probability distribution, but there are many aspects of the distribution that it does not capture. For instance, the following illustrates three random variables that have different distributions but the same expected value of 0.5:



Beyond the expectation, the next most important aspect of a probability distribution is its “spread”<sup>83</sup>. The *variance* of a random variable encapsulates this information.

<sup>83</sup> There are higher-order “moments”<sup>84</sup> of a distribution as well, such as skewness<sup>85</sup> and kurtosis<sup>86</sup>. However, expectation and variance (or standard deviation, its square root) are the most commonly used.

<sup>84</sup> [https://en.wikipedia.org/wiki/Standardized\\_moment](https://en.wikipedia.org/wiki/Standardized_moment)

<sup>85</sup> <https://en.wikipedia.org/wiki/Skewness>

<sup>86</sup> <https://en.wikipedia.org/wiki/Kurtosis>

**Definition 245 (Variance)** Suppose  $X$  is a random variable with expectation  $\mathbb{E}[X]$ . Then the *variance* of  $X$  is defined as

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2],$$

or equivalently,

$$\text{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

The second definition follows from the first due to linearity of expectation:

$$\begin{aligned}\text{Var}(X) &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\ &= \mathbb{E}[X^2 - 2\mathbb{E}[X] \cdot X + \mathbb{E}[X]^2] \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X] \cdot \mathbb{E}[X] + \mathbb{E}[\mathbb{E}[X]^2] \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X] \cdot \mathbb{E}[X] + \mathbb{E}[X]^2 \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2 \\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2.\end{aligned}$$

In the fourth step, we used the fact that  $\mathbb{E}[X]$  is a constant to pull it out of the outer expectation: by linearity of expectation,  $\mathbb{E}[cY] = c\mathbb{E}[Y]$  for any constant  $c$ .

The variance tells us the average square of the distance of a random variable from its expectation. Taking the square root of the variance gives us an approximate measure of the distance itself; this is called the *standard deviation*, and it is often denoted by the symbol  $\sigma$ :

$$\sigma(X) = \sqrt{\text{Var}(X)}.$$

**Example 246** Suppose that random variable  $X$  has the distribution

$$X = 0.5 \text{ with probability 1}$$

and  $Y$  has the distribution

$$Y = \begin{cases} 0 & \text{with probability 1/2} \\ 1 & \text{with probability 1/2.} \end{cases}$$

Both  $X$  and  $Y$  have expectations 0.5. We calculate their variances. The distributions of  $X^2$  and  $Y^2$  are as follows:

$$X^2 = 0.25 \text{ with probability 1}$$

$$Y^2 = \begin{cases} 0 & \text{with probability 1/2} \\ 1 & \text{with probability 1/2.} \end{cases}$$

Therefore,

$$\text{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = 0.25 - 0.5^2 = 0$$

$$\text{Var}(Y) = \mathbb{E}[Y^2] - \mathbb{E}[Y]^2 = 0.5 - 0.5^2 = 0.25.$$

We see that while  $X$  and  $Y$  have the same expectation, their variances differ. This quantifies the fact that  $X$  has a certain value with certainty, whereas  $Y$  can take on multiple values, so  $Y$  has larger variance.

**Example 247** Let  $X$  be an indicator random variable with probability  $p$  of being 1:

$$X = \begin{cases} 0 & \text{with probability } 1 - p \\ 1 & \text{with probability } p. \end{cases}$$

Then  $\mathbb{E}[X] = \Pr[X = 1] = p$ . Observe that  $X^2 = X$  in all cases, so  $\mathbb{E}[X^2] = \mathbb{E}[X] = p$ . Thus, the variance of  $X$  is

$$\begin{aligned} \text{Var}(X) &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 \\ &= p - p^2 \\ &= p(1 - p). \end{aligned}$$

As a more complex example, define  $X$  to be the number of heads over  $n$  tosses of a biased coin with probability  $p$  of coming up heads. In Example 234 we calculated that  $\mathbb{E}[X] = pn$ . Defining  $X_i$  to be the indicator random variable for whether the  $i$ th toss comes up heads, we can infer from Example 247 that  $\text{Var}(X_i) = p(1 - p)$ . What can we say about the variance  $\text{Var}(X) = \text{Var}(\sum_i X_i)$  of  $X$  itself?

For expectation, we know from Theorem 231 that linearity of expectation holds for any random variables, even arbitrarily correlated ones. For variance, this is not the case. As an example, for a random variable  $Y$ , define  $Z = Y + Y = 2Y$ . It is *not* the case that  $\text{Var}(Z) = 2 \text{Var}(Y)$ ; in fact,  $\text{Var}(Z) = 4 \text{Var}(Y)$ .

**Theorem 248** Let  $X$  be a random variable. Then  $\text{Var}(cX) = c^2 \cdot \text{Var}(X)$  for any constant  $c$ .

**Proof 249** By definition of variance,

$$\begin{aligned} \text{Var}(cX) &= \mathbb{E}[(cX)^2] - \mathbb{E}[cX]^2 \\ &= \mathbb{E}[c^2 X^2] - (c \mathbb{E}[X])^2 \\ &= c^2 \cdot \mathbb{E}[X^2] - c^2 \cdot \mathbb{E}[X]^2 \\ &= c^2 \cdot (\mathbb{E}[X^2] - \mathbb{E}[X]^2) \\ &= c^2 \cdot \text{Var}(X). \end{aligned}$$

In the second and third steps, we applied linearity of expectation to pull the constants  $c$  and  $c^2$  out of the expectations.  $\square$

For a sum of random variables, the variances sum if the random variables are *independent*; in fact, *pairwise* independence suffices for this. To establish this, we first demonstrate that the expectation of the product of independent random variables is the product of their expectations.

**Lemma 250** Let  $X$  and  $Y$  be independent random variables. Then

$$\mathbb{E}[XY] = \mathbb{E}[X] \cdot \mathbb{E}[Y].$$

**Proof 251** By definition of expectation,

$$\begin{aligned} \mathbb{E}[XY] &= \sum_{x,y} xy \cdot \Pr[X = x \wedge Y = y] \\ &= \sum_x \sum_y xy \cdot \Pr[X = x] \cdot \Pr[Y = y] \\ &= \left( \sum_x x \cdot \Pr[X = x] \right) \left( \sum_y y \cdot \Pr[Y = y] \right) \\ &= \mathbb{E}[X] \cdot \mathbb{E}[Y]. \end{aligned}$$

In the second step, we used the fact that  $X$  and  $Y$  are independent, so that  $\Pr[X = x \wedge Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$ .  $\square$

By induction, we conclude that

$$\mathbb{E}\left[\prod_i X_i\right] = \prod_i \mathbb{E}[X_i]$$

for mutually independent random variables  $X_i$ .

We now consider the variance of the sum of independent random variables.

**Theorem 252** *Let  $X$  and  $Y$  be independent random variables. Then  $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$ .*

**Proof 253** By definition of variance, linearity of expectation, and Lemma 250,

$$\begin{aligned} \text{Var}(X + Y) &= \mathbb{E}[(X + Y)^2] - \mathbb{E}[X + Y]^2 \\ &= \mathbb{E}[X^2 + 2XY + Y^2] - (\mathbb{E}[X] + \mathbb{E}[Y])^2 \\ &= \mathbb{E}[X^2] + 2\mathbb{E}[XY] + \mathbb{E}[Y^2] - (\mathbb{E}[X]^2 + 2\mathbb{E}[X] \cdot \mathbb{E}[Y] + \mathbb{E}[Y]^2) \\ &= \mathbb{E}[X^2] + 2\mathbb{E}[X] \cdot \mathbb{E}[Y] + \mathbb{E}[Y^2] - (\mathbb{E}[X]^2 + 2\mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[Y]^2) \\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2 \\ &= \text{Var}(X) + \text{Var}(Y). \end{aligned}$$

By induction, we conclude that

$$\text{Var}\left(\sum_i X_i\right) = \sum_i \text{Var}(X_i)$$

for mutually independent random variables  $X_i$ . (In fact, a close inspection of the proof reveals that *pairwise* independence suffices.)

Thus, letting  $X$  be the number of heads over  $n$  tosses of a biased coin with probability  $p$  of coming up heads, we have that

$$\begin{aligned} \text{Var}(X) &= \text{Var}\left(\sum_i X_i\right) \\ &= \sum_i \text{Var}(X_i) \\ &= \sum_i p(1-p) \\ &= p(1-p)n. \end{aligned}$$

*Chebyshev's inequality* bounds the probability that a random variable differs from its expectation by some threshold.

**Theorem 254 (Chebyshev's Inequality)** *Let  $X$  be a random variable and  $a > 0$ . Then*

$$\Pr[|X - \mathbb{E}[X]| \geq a] \leq \frac{\text{Var}(X)}{a^2}.$$

**Proof 255** We apply *Markov's inequality* (page 211) to the random variable  $Y = (X - \mathbb{E}[X])^2$ , which has expectation  $\mathbb{E}[Y] = \text{Var}(X)$  by definition. Observe that because  $Y$  is the square of a real number, it is non-negative, so Markov's inequality applies to it. Furthermore, because  $a$  is positive,  $Y = (X - \mathbb{E}[X])^2 \geq a^2$  holds

if and only if  $|X - \mathbb{E}[X]| \geq a$ . Thus,

$$\begin{aligned}\Pr[|X - \mathbb{E}[X]| \geq a] &= \Pr[Y \geq a^2] \\ &\leq \frac{\mathbb{E}[Y]}{a^2} \\ &= \frac{\text{Var}(X)}{a^2}.\end{aligned}$$

**Example 256** Suppose we toss a fair coin  $n$  times. We use Chebyshev's inequality to calculate an upper bound on the probability of getting  $\leq 49\%$  or  $\geq 51\%$  heads.

Let  $X$  be the number of heads. We previously showed that  $\mathbb{E}[X] = n/2$ , and  $\text{Var}(X) = n/4$ . Then

$$\begin{aligned}\Pr[|X - n/2| \geq n/100] &\leq \frac{\text{Var}(X)}{(n/100)^2} \\ &= 10000 \cdot \frac{n/4}{n^2} \\ &= \frac{2500}{n}.\end{aligned}$$

For example, for 10,000 tosses, the probability of deviating from the expectation by 1% is at most 1/4, while for 1,000,000 tosses, it is at most 1/400.

In the above example, the random variable  $X = X_1 + \dots + X_n$  is the sum of (mutually) *independent, identically distributed* (i.i.d.) random variables  $X_i$ . In such a scenario, it is often more convenient to reason about the “normalized” value  $X/n$  rather than  $X$  itself, since the expectation of  $X/n$  is independent of  $n$ : by linearity of expectation,  $\mathbb{E}[X] = n \cdot \mathbb{E}[X_i]$ , whereas  $\mathbb{E}[X/n] = \mathbb{E}[X]/n = \mathbb{E}[X_i]$ . The following is an alternative formulation of Chebyshev's inequality for this case.

**Corollary 257** Let  $X = X_1 + \dots + X_n$  be the sum of  $n \geq 1$  independent, identically distributed random variables  $X_i$ . Let  $\varepsilon > 0$  be a deviation from the expectation  $\mathbb{E}[X/n] = \mathbb{E}[X_i]$ . Then

$$\Pr[|X/n - \mathbb{E}[X_i]| \geq \varepsilon] \leq \frac{\text{Var}(X_i)}{\varepsilon^2 n}.$$

**Proof 258** The event  $|X/n - \mathbb{E}[X_i]| \geq \varepsilon$  is equivalent to the event  $|X - \mathbb{E}[X]| \geq \varepsilon n$ , by multiplying both sides by  $n$ . By the standard form of Chebyshev's inequality, we have

$$\begin{aligned}\Pr[|X/n - \mathbb{E}[X_i]| \geq \varepsilon] &= \Pr[|X - \mathbb{E}[X]| \geq \varepsilon n] \\ &\leq \frac{\text{Var}(X)}{\varepsilon^2 n^2} \\ &= \frac{n \cdot \text{Var}(X_i)}{\varepsilon^2 n^2} \\ &= \frac{\text{Var}(X_i)}{\varepsilon^2 n}.\end{aligned}$$

In the second-to-last step, we used [Theorem 252](#) to deduce that  $\text{Var}(X) = n \cdot \text{Var}(X_i)$ , since the  $X_i$  are independent.  $\square$

We can redo [Example 256](#) using [Corollary 257](#) to obtain the same bounds.

In conclusion, Chebyshev's inequality tells us that for  $n$  independent coin tosses, the probability of deviating by at least (say) 1% from the expected number of heads decreases at least linearly in  $n$ . In fact, the probability actually decreases *much faster* than this! Using [multiplicative Chernoff bounds](#) (page 274) or [Hoeffding's inequality](#) (page 234) from the following sections, we can show that the probability decreases *exponentially* in the number of tosses.

## 22.2 Hoeffding's Inequality

*Hoeffding's inequality*, also called *Chernoff-Hoeffding bounds*, is a set of concentration bounds that can give tighter results than Markov's inequality, Chebyshev's inequality, or other forms of Chernoff bounds. For simplicity, we restrict ourselves to the special case of independent indicator random variables having expectation

$$\Pr[X_i = 1] = p_i$$

for each indicator  $X_i$ .<sup>87</sup>

Let  $p$  denote the expectation of  $X/n$ . Then by linearity of expectation,

$$p = \mathbb{E}[X/n] = \sum_i \mathbb{E}[X_i]/n = \frac{1}{n} \sum_{i=1}^n p_i .$$

That is, the expectation of  $X/n$  is just the average of the  $p_i$ .

Hoeffding's inequality bounds the probability that  $X/n$  deviates from its expectation by a threshold of interest.

**Theorem 259 (Hoeffding's Inequality)** Let  $X = X_1 + \dots + X_n$  be the sum of independent indicator random variables  $X_i$  with  $\mathbb{E}[X_i] = p_i$ , and let

$$p = \mathbb{E}[X/n] = \frac{1}{n} \sum_{i=1}^n p_i .$$

Let  $\varepsilon > 0$  be a deviation from the expectation. Then we have the “upper tail” bound

$$\Pr[X/n \geq p + \varepsilon] \leq e^{-2\varepsilon^2 n} ,$$

and the same holds for the “lower tail”  $\Pr[X/n \leq p - \varepsilon]$ . Combining these via the union bound,

$$\Pr[|X/n - p| \geq \varepsilon] \leq 2e^{-2\varepsilon^2 n} .$$

We again consider the example of tossing a fair coin  $n$  times. Let  $H$  be the total number of heads that come up, and let  $H_i$  be the indicator random variable that indicates whether the  $i$ th toss comes up heads. We have that  $\mathbb{E}[H_i] = \Pr[H_i = 1] = 1/2$ , and  $\mathbb{E}[H/n] = \mathbb{E}[H_i] = 1/2$  for any number of tosses  $n$ .

What is a bound on the probability that in ten tosses, at least six heads come up? This is a deviation of  $\varepsilon = 0.1$  from the (normalized) expectation of  $p = 0.5$ , so applying the upper-tail Hoeffding's inequality gives us:

$$\Pr[H/n \geq p + \varepsilon] = \Pr[H/n \geq 0.5 + 0.1] \leq e^{-2 \cdot (0.1)^2 \cdot 10} \approx 0.9802^{10} \approx 0.82 .$$

Now, what is a bound on the probability that in 100 tosses, at least 60 heads come up? This is again the same deviation  $\varepsilon = 0.1$  from the expectation of  $p = 0.5$ ; only the number of trials is different. Applying the upper tail again, we get

$$\Pr[H/n \geq p + \varepsilon] \leq e^{-2 \cdot (0.1)^2 \cdot 100} \approx 0.9802^{100} \approx 0.14 .$$

This is a significantly tighter bound than what we would get from Chebyshev's inequality (Theorem 254); see Example 256 for a comparison. It is also tighter than what we would get from the *multiplicative Chernoff bound* (page 274).

**Example 260** Suppose we have a coin that is biased by probability  $\varepsilon$  towards either heads or tails, but we don't know which one. In other words, either:

- $\Pr[\text{heads}] = \frac{1}{2} + \varepsilon$  and  $\Pr[\text{tails}] = \frac{1}{2} - \varepsilon$ , or
- $\Pr[\text{heads}] = \frac{1}{2} - \varepsilon$  and  $\Pr[\text{tails}] = \frac{1}{2} + \varepsilon$

<sup>87</sup> See the [appendix](#) (page 306) for the general case of Hoeffding's inequality, as well as proofs of the *special* (page 302) and *general* (page 306) cases.

To determine in which direction the coin is biased, we toss the coin  $n$  times. If the results include at least  $n/2$  heads, we assume the coin is biased towards heads, otherwise we assume it is biased towards tails. How many tosses should we do to guarantee our answer is correct with probability at least  $1 - \delta$ ?

Let  $X$  be the number of heads, and let  $X_i$  be an indicator that is 1 if the  $i$ th toss is heads, 0 if it is tails. Then  $X = X_1 + \dots + X_n$  is the sum of independent indicator random variables with  $\mathbb{E}[X_i]$  equal to either  $\frac{1}{2} + \varepsilon$  or to  $\frac{1}{2} - \varepsilon$ . We analyze the two cases individually.

- **Case 1:** The coin is biased towards heads. Then  $\mathbb{E}[X_i] = p = \frac{1}{2} + \varepsilon$ . Our guess is erroneous when:

$$\begin{aligned} X &< \frac{n}{2} \\ \frac{1}{n}X &< \frac{1}{2} \\ &= \left(\frac{1}{2} + \varepsilon\right) - \varepsilon \\ &= p - \varepsilon \end{aligned}$$

By the lower-tail Hoeffding's inequality, we have

$$\begin{aligned} \Pr\left[\frac{1}{n}X < p - \varepsilon\right] &\leq \Pr\left[\frac{1}{n}X \leq p - \varepsilon\right] \\ &\leq e^{-2\varepsilon^2 n} \end{aligned}$$

In the first step, we used the fact that  $\Pr[Y \leq a] = \Pr[Y < a] + \Pr[Y = a] \geq \Pr[Y < a]$  for a random variable  $Y$  and any value  $a$ .

- **Case 2:** The coin is biased towards tails. Then  $\mathbb{E}[X_i] = p = \frac{1}{2} - \varepsilon$ . Our guess is erroneous when:

$$\begin{aligned} X &\geq \frac{n}{2} \\ \frac{1}{n}X &\geq \frac{1}{2} \\ &= \left(\frac{1}{2} - \varepsilon\right) + \varepsilon \\ &= p + \varepsilon \end{aligned}$$

By the upper-tail Hoeffding's inequality, we have

$$\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] \leq e^{-2\varepsilon^2 n}$$

In either case, the probability of error after  $n$  tosses is upper-bounded by  $e^{-2\varepsilon^2 n}$ . We want this probability to be no more than  $\delta$ :

$$\begin{aligned} e^{-2\varepsilon^2 n} &\leq \delta \\ 1/\delta &\leq e^{2\varepsilon^2 n} \\ \ln(1/\delta) &\leq 2\varepsilon^2 n \\ \frac{\ln(1/\delta)}{2\varepsilon^2} &\leq n \end{aligned}$$

If  $\varepsilon = 0.01$  (i.e. the coin is biased towards heads or tails by 1%) and  $\delta = 0.0001$  (we want to be correct at least 99.99% of the time), then

$$n \geq \frac{\ln(1/0.0001)}{2 \cdot 0.01^2} \approx 46502$$

tosses suffice.

**Exercise 261** Suppose we have a coin that is either fair, or is biased by probability  $\varepsilon$  towards heads, but we don't know which is the case. In other words, either:

- $\Pr[\text{heads}] = \frac{1}{2}$  and  $\Pr[\text{tails}] = \frac{1}{2}$ , or
- $\Pr[\text{heads}] = \frac{1}{2} + \varepsilon$  and  $\Pr[\text{tails}] = \frac{1}{2} - \varepsilon$

We toss the coin  $n$  times and count the number of heads  $X$ .

- a. For what values of  $X$  should we guess the coin is fair, and for what values that it is biased towards heads?
- b. How many tosses should we do to guarantee our answer is correct with probability at least  $1 - \delta$ ?
- c. How many tosses should we do for  $\varepsilon = 0.01$  and  $\delta = 0.0001$ ? How does this compare to the situation in Example 260 with  $\varepsilon = 0.005$  and  $\delta = 0.0001$ ?

## 22.3 Polling

Rather than applying concentration bounds to compute the probability of a deviation for a specific sample size, we often wish to determine how many samples we need to be within a particular deviation with high confidence. One application of this is *big data*, where we have vast datasets that are too large to examine in their entirety, so we sample the data instead to estimate the quantities of interest. Similar to this is *polling* – outside of elections themselves, we typically do not have the resources to ask the entire population for their opinions, so we need to sample the populace to estimate the support for a particular candidate or political position. In both applications, we need to determine how many samples are needed to obtain a good estimate.

In general, a poll estimates the fraction of the population that supports a particular candidate by asking  $n$  randomly chosen people whether they support that candidate. Let  $X$  be a random variable corresponding to the number of people who answer this question in the affirmative. Then  $X/n$  is an estimate of the level of support in the full population.

A typical poll has both a *confidence level* and a *margin of error* – the latter corresponds to the deviation from the true fraction  $p$  of people who support the candidate, and the former corresponds to a bound on the probability that the estimate is within that deviation. For example, a 95% confidence level and a margin of error of  $\pm 2\%$  requires that

$$\Pr\left[\left|\frac{X}{n} - p\right| \leq 0.02\right] \geq 0.95$$

More generally, for a confidence level  $1 - \gamma$  and margin of error  $\varepsilon$ , we require

$$\Pr\left[\left|\frac{X}{n} - p\right| \leq \varepsilon\right] \geq 1 - \gamma$$

or equivalently

$$\Pr\left[\left|\frac{X}{n} - p\right| > \varepsilon\right] < \gamma$$

Formally, we define indicator variables  $X_i$  as

$$X_i = \begin{cases} 1 & \text{if person } i \text{ supports the candidate} \\ 0 & \text{otherwise.} \end{cases}$$

for each person  $i$  in the set that we poll. Then  $X = X_1 + \dots + X_n$  is the sum of independent indicator variables, with

$$\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = np$$

### 22.3.1 Analysis with Hoeffding's Inequality

Applying Hoeffding's inequality to polling, the combined inequality gives us

$$\Pr\left[\left|\frac{1}{n}X - p\right| \geq \varepsilon\right] \leq 2e^{-2\varepsilon^2 n}$$

For a 95% confidence level and a margin of error of  $\pm 2\%$ , we require that

$$\Pr\left[\left|\frac{X}{n} - p\right| \leq 0.02\right] \geq 0.95$$

However, this isn't quite in the form where we can apply Hoeffding's inequality, so we need to do some manipulation first. We have:

$$\begin{aligned}\Pr\left[\left|\frac{X}{n} - p\right| \leq 0.02\right] &= 1 - \Pr\left[\left|\frac{X}{n} - p\right| > 0.02\right] \\ &\geq 1 - \Pr\left[\left|\frac{X}{n} - p\right| \geq 0.02\right]\end{aligned}$$

Hoeffding's inequality gives us:

$$\Pr\left[\left|\frac{X}{n} - p\right| \geq 0.02\right] \leq 2e^{-2 \cdot 0.02^2 \cdot n}$$

Substituting this into the above, we get:

$$\Pr\left[\left|\frac{X}{n} - p\right| \leq 0.02\right] \geq 1 - 2e^{-2 \cdot 0.02^2 \cdot n}$$

We want this to be at least 0.95:

$$\begin{aligned}1 - 2e^{-2 \cdot 0.02^2 \cdot n} &\geq 0.95 \\ 2e^{-2 \cdot 0.02^2 \cdot n} &\leq 0.05 \\ e^{2 \cdot 0.02^2 \cdot n} &\geq 40 \\ 2 \cdot 0.02^2 \cdot n &\geq \ln 40 \\ n &\geq \frac{\ln 40}{2 \cdot 0.02^2} \\ &\approx 4611.1\end{aligned}$$

Thus, we obtain the given confidence level and margin of error by polling at least 4612 people. Observe that this does not depend on the total population size!

For an arbitrary margin of error  $\pm \varepsilon$ , we obtain:

$$\begin{aligned}\Pr\left[\left|\frac{X}{n} - p\right| \leq \varepsilon\right] &= 1 - \Pr\left[\left|\frac{X}{n} - p\right| > \varepsilon\right] \\ &\geq 1 - \Pr\left[\left|\frac{X}{n} - p\right| \geq \varepsilon\right] \\ &\geq 1 - 2e^{-2\varepsilon^2 n}\end{aligned}$$

To achieve an arbitrary confidence level  $1 - \gamma$ , we need:

$$\begin{aligned} 1 - 2e^{-2\varepsilon^2 n} &\geq 1 - \gamma \\ 2e^{-2\varepsilon^2 n} &\leq \gamma \\ e^{2\varepsilon^2 n} &\geq \frac{2}{\gamma} \\ 2\varepsilon^2 n &\geq \ln\left(\frac{2}{\gamma}\right) \\ n &\geq \frac{1}{2\varepsilon^2} \ln\left(\frac{2}{\gamma}\right) \end{aligned}$$

More generally, if we wish to gauge the level of support for  $m$  different candidates, the *sampling theorem* tells us that the number of samples required is logarithmic in  $m$ .

**Theorem 262 (Sampling Theorem)** Suppose  $n$  people are polled to ask which candidate they support, out of  $m$  possible candidates. Let  $X^{(j)}$  be the number of people who state that they support candidate  $j$ , and let  $p_j$  be the true level of support for that candidate. We wish to obtain

$$\Pr\left[\bigcap_j \left(\left|\frac{X^{(j)}}{n} - p_j\right| \leq \varepsilon\right)\right] \geq 1 - \gamma$$

In other words, we desire a confidence level  $1 - \gamma$  that all estimates  $X^{(j)}/n$  are within margin of error  $\pm \varepsilon$  of the true values  $p_j$ . We obtain this when the number of samples  $n$  is

$$n \geq \frac{1}{2\varepsilon^2} \ln\left(\frac{2m}{\gamma}\right)$$

The sampling theorem can be derived from applying the *union bound* (page 206).

In conclusion, when sampling from a large dataset, the number of samples required does depend on the desired accuracy of the estimation and the range size (i.e. number of possible answers). But it does not depend on the population size. This makes sampling a powerful technique for dealing with big data, as long as we are willing to tolerate a small possibility of obtaining an inaccurate estimate.

## 22.4 Load Balancing

Job scheduling is another application where we can exploit randomness to construct a simple, highly effective algorithm. In this problem, we have  $k$  servers, and there are  $n$  jobs that need to be distributed to these servers. The goal is to *balance* the load among the servers, so that no one server is overloaded. This problem is very relevant to content-delivery networks – there may be on the order of millions or even billions of concurrent users, and each user needs to be routed to one of only hundreds or thousands of servers. Thus, we have  $n \gg k$  in such a case.

One possible algorithm is to always send a job to the most lightly loaded server. However, this requires significant coordination – the scheduler must keep track of the load on each server, which requires extra communication, space, and computational resources. Instead, we consider a simple, randomized algorithm that just sends each job to a random server. The expected number of jobs on each server is  $n/k$ . But how likely are we to be close to this ideal, balanced load?

We start our analysis by defining random variables  $X^{(j)}$  corresponding to the number of jobs assigned to server  $j$ . We would like to demonstrate that the joint probability of  $X^{(j)}$  being close to  $n/k$  for all  $j$  is high. We first reason about the load on an individual server. In particular, we wish to compute a bound on

$$\Pr\left[X^{(j)} \geq \frac{n}{k} + c\right]$$

for some value  $c > 0$ , i.e. the probability that server  $j$  is overloaded by at least  $c$  jobs. Let  $X_i^{(j)}$  be an indicator random variable that is 1 if job  $i$  is sent to server  $j$ . Since the target server for job  $i$  is chosen uniformly at random out of  $k$  possible choices, we have

$$\mathbb{E}[X_i^{(j)}] = \Pr[X_i^{(j)} = 1] = \frac{1}{k}$$

We also have  $X^{(j)} = X_1^{(j)} + \dots + X_n^{(j)}$ , giving us

$$\mathbb{E}[X^{(j)}] = \sum_i \mathbb{E}[X_i^{(j)}] = \frac{n}{k}$$

as we stated before. Then:

$$\begin{aligned} \Pr[X^{(j)} \geq \frac{n}{k} + c] &= \Pr\left[\frac{1}{n}X^{(j)} \geq \frac{1}{k} + \frac{c}{n}\right] \\ &\leq e^{-2(c/n)^2 n} \\ &= e^{-2c^2/n} \end{aligned}$$

by the upper-tail Hoeffding's inequality.

Now that we have a bound on an individual server being overloaded by  $c$  jobs, we wish to bound the probability that there is *some* server that is overloaded. The complement event is that no server is overloaded, so we can equivalently compute the joint probability that none of the servers is overloaded by  $c$  or more jobs:

$$\Pr[X^{(1)} < \frac{n}{k} + c, \dots, X^{(n)} < \frac{n}{k} + c]$$

However, we cannot do so by simply multiplying the individual probabilities together – that only works if the probabilities are independent, and in this case, they are not. In particular, if one server is overloaded, some other server must be underloaded, so the random variables  $X^{(j)}$  are not independent.

Rather than trying to work with the complement event, we attempt to directly compute the probability that there is at least one overloaded server:

$$\Pr\left[(X^{(1)} \geq \frac{n}{k} + c) \cup \dots \cup (X^{(n)} \geq \frac{n}{k} + c)\right]$$

More succinctly, we denote this as:

$$\Pr\left[\bigcup_j (X^{(j)} \geq \frac{n}{k} + c)\right]$$

We have a union of events, and we want to analyze the probability of that union. We use the *union bound* (Lemma 212):

$$\begin{aligned} \Pr\left[\bigcup_j (X^{(j)} \geq \frac{n}{k} + c)\right] &\leq \sum_j \Pr[X^{(j)} \geq \frac{n}{k} + c] \\ &\leq \sum_j e^{-2c^2/n} \\ &= k \cdot e^{-2c^2/n} \\ &= e^{\ln k} \cdot e^{-2c^2/n} \\ &= e^{\ln k - 2c^2/n} \end{aligned}$$

For  $c = \sqrt{n \ln k}$ , we get:

$$\begin{aligned} \Pr \left[ \bigcup_j (X^{(j)} \geq \frac{n}{k} + \sqrt{n \ln k}) \right] &\leq e^{\ln k - 2(\sqrt{n \ln k})^2/n} \\ &= e^{\ln k - 2(n \ln k)/n} \\ &= e^{-\ln k} \\ &= 1/e^{\ln k} \\ &= 1/k \end{aligned}$$

With concrete values  $n = 10^{10}$  and  $k = 1000$ , we compute the overload relative to the expected value as:

$$\begin{aligned} \frac{\sqrt{n \ln k}}{n/k} &= \frac{\sqrt{10^{10} \ln 1000}}{10^7} \\ &\approx 0.026 \end{aligned}$$

This is an overhead of about 2.6% above the expected load. The probability that there is a server overloaded by at least 2.6% is bounded from above by  $1/k = 0.001$ , or  $\leq 0.1\%$ . Thus, when there are  $n = 10^{10}$  jobs and  $k = 1000$  servers, the randomized algorithm has a high probability ( $\geq 99.9\%$ ) of producing a schedule where the servers all have a low overhead ( $\leq 2.6\%$ ).

**Exercise 263** Previously, we *demonstrated* (page 237) that to use polling to achieve a confidence level  $1 - \gamma$  and a margin of error  $\pm \varepsilon$ ,

$$n \geq \frac{1}{2\varepsilon^2} \ln\left(\frac{2}{\gamma}\right)$$

samples are sufficient when there is a single candidate. We also saw the *sampling theorem* (page 238), which states that for  $m$  candidates,

$$n \geq \frac{1}{2\varepsilon^2} \ln\left(\frac{2m}{\gamma}\right)$$

samples suffice to achieve a confidence level  $1 - \gamma$  that all estimates  $X^{(j)}/n$  are within margin of error  $\pm \varepsilon$ .

Use the union bound to demonstrate that this latter result follows from the result for a single candidate.

## **Part V**

# **Cryptography**

---

CHAPTER  
**TWENTYTHREE**

---

## INTRODUCTION TO CRYPTOGRAPHY

Security and privacy are core principles in computing, enabling a wide range of applications including online commerce, social networking, wireless communication, and so on. *Cryptography*, which is concerned with techniques and protocols for secure communication, is fundamental to building systems that provide security and privacy. In this unit, we will examine several cryptographic protocols, which address the following needs:

- *authentication*: proving one's identity
- *privacy/confidentiality*: ensuring that no one can read the message except the intended receiver
- *integrity*: guaranteeing that the received message has not been altered in any way

Our standard problem setup is that we have two parties who wish to communicate, traditionally named *Alice* and *Bob*. However, they are communicating over an insecure channel, and there is an eavesdropper *Eve* who can observe all their communication, and in some cases, can even modify the data in-flight. How can Alice and Bob communicate while achieving the goals of authentication, privacy, and integrity?

A central goal in designing a cryptosystem is *Kerckhoff's principle*:

A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.

Thus, we want to ensure that Alice and Bob can communicate securely even if Eve knows every detail about what protocol they are using, other than the *key*, a secret piece of knowledge that is never communicated over the insecure channel.

We refer to the message that Alice and Bob wish to communicate as the *plaintext*. We want to design a cryptosystem that involves encoding the message in such a way as to prevent Eve from recovering the plaintext, even with access to the *ciphertext*, the result of encoding the message. There are two levels of security around which we can design a cryptosystem:

- *Information-theoretic*, or *unconditional*, security: Eve cannot learn the secret message communicated between Alice and Bob, even with unbounded computational power.
- *Computational*, or *conditional*, security: to learn any information about the secret message, Eve will have to solve a computationally hard problem.

The cryptosystems we examine all rely to some extent on modular arithmetic. Before we proceed further, we review some basic details about modular arithmetic.

## 23.1 Review of Modular Arithmetic

Modular arithmetic is a mathematical system that maps the infinitely many integers to a finite set, those in  $\{0, 1, \dots, n-1\}$  for some positive *modulus*  $n \in \mathbb{Z}^+$ . The core concept in this system is that of *congruence*: two integers  $a$  and  $b$  are said to be congruent modulo  $n$ , written as

$$a \equiv b \pmod{n}$$

when  $a$  and  $b$  differ by an integer multiple of  $n$ :

$$\exists k \in \mathbb{Z}. a - b = kn$$

Note that  $a$  and  $b$  need not be in the range  $[0, n)$ ; in fact, if  $a \neq b$ , than at least one must be outside this range for  $a$  and  $b$  to be congruent modulo  $n$ . More importantly, for any integer  $i \in \mathbb{Z}$ , there is exactly one integer  $j \in \{0, 1, \dots, n-1\}$  such that  $i \equiv j \pmod{n}$ . This is because the elements in this set are at most  $n-1$  apart, so no two elements differ by a multiple of  $n$ . At the same time, by Euclid's division lemma<sup>88</sup>, we know that there exist unique integers  $q$  and  $r$  such that

$$i = nq + r$$

where  $0 \leq r < n$ . Thus, each integer  $i \in \mathbb{Z}$  is mapped to exactly one integer  $j \in \{0, 1, \dots, n-1\}$  by the congruence relation modulo  $n$ .

Formally, we can define a set of *equivalence classes*, denoted by  $\overline{\mathbb{Z}_n}$ , corresponding to each element in  $\{0, 1, \dots, n-1\}$ :

$$\overline{\mathbb{Z}_n} = \{\overline{0}, \overline{1}, \dots, \overline{n-1}\}$$

Each class  $\overline{j}$  consists of the integers that are congruent to  $j$  modulo  $n$ :

$$\overline{j} = \{j, j-n, j+n, j-2n, j+2n, \dots\}$$

However, the overline notation here is cumbersome, so we usually elide it, making the equivalence classes implicit instead:

$$\mathbb{Z}_n = \{0, 1, \dots, n-1\}$$

We refer to determining the equivalence class of an integer  $i$  modulo  $n$  as *reducing* it modulo  $n$ . If we know what  $i$  is, we need only compute its remainder when divided by  $n$  to reduce it. More commonly, we have a complicated expression for  $i$ , consisting of additions, subtractions, multiplications, exponentiations, and so on. Rather than evaluating the expression directly, we can take advantage of properties of modular arithmetic to simplify the task of reducing the expression.

**Property 264** Suppose  $a \equiv a' \pmod{n}$  and  $b \equiv b' \pmod{n}$  for a modulus  $n \geq 1$ . Then

$$a + b \equiv a' + b' \pmod{n}$$

and

$$a - b \equiv a' - b' \pmod{n}$$

**Proof 265** By definition of congruence, we have  $a - a' = kn$  and  $b - b' = mn$  for some integers  $k$  and  $m$ . Then

$$\begin{aligned} a + b &= (kn + a') + (mn + b') \\ &= (k + m)n + a' + b' \end{aligned}$$

Since  $a + b$  and  $a' + b'$  differ by an integer  $(k + m)$  multiple of  $n$ , we conclude that  $a + b \equiv a' + b' \pmod{n}$ .

<sup>88</sup> [https://en.wikipedia.org/wiki/Euclidean\\_division](https://en.wikipedia.org/wiki/Euclidean_division)

The proof for  $a - b \equiv a' - b' \pmod{n}$  is similar. □

**Property 266** Suppose  $a \equiv a' \pmod{n}$  and  $b \equiv b' \pmod{n}$  for a modulus  $n \geq 1$ . Then

$$ab \equiv a'b' \pmod{n}$$

**Proof 267** By definition of congruence, we have  $a - a' = kn$  and  $b - b' = mn$  for some integers  $k$  and  $m$ . Then

$$\begin{aligned} ab &= (kn + a') \cdot (mn + b') \\ &= kmn^2 + a'mn + b'kn + a'b' \\ &= (kmn + a'm + b'k)n + a'b' \end{aligned}$$

Since  $ab$  and  $a'b'$  differ by an integer  $(kmn + a'm + b'k)$  multiple of  $n$ , we conclude that  $ab \equiv a'b' \pmod{n}$ . □

**Corollary 268** Suppose  $a \equiv b \pmod{n}$ . Then for any integer  $k \geq 0$ ,

$$a^k \equiv b^k \pmod{n}$$

We can prove Corollary 268 by observing that  $a^k = a \cdot a \cdot \dots \cdot a$  is the product of  $k$  copies of  $a$  and applying Property 266 along with induction over  $k$ . We leave the details as an exercise.

The following is an example that applies the properties above to reduce a complicated expression.

**Example 269** Suppose we wish to find an element  $a \in \mathbb{Z}_7$  such that

$$(2^{203} \cdot 3^{281} + 4^{370})^{376} \equiv a \pmod{7}$$

Note that the properties above do not give us the ability to reduce any of the exponents modulo 7. (Later, we will see *Fermat's little theorem* (page 256), which does give us a means of simplifying exponents. We also will see *fast modular exponentiation* (page 245), but we will not use that here.) We can use Property 268 once we reduce the base

$$2^{203} \cdot 3^{281} + 4^{370}$$

which we can recursively reduce using the properties above.

Let's start with  $2^{203}$ . Observe that  $2^3 = 8 \equiv 1 \pmod{7}$ . Then

$$\begin{aligned} 2^{203} &= 2^{201} \cdot 2^2 \\ &= (2^3)^{67} \cdot 4 \\ &\equiv 1^{67} \cdot 4 \pmod{7} \\ &\equiv 4 \pmod{7} \end{aligned}$$

Similarly,  $3^3 = 27 \equiv -1 \pmod{7}$ , so  $3^6 \equiv 1 \pmod{7}$ . This gives us

$$\begin{aligned} 3^{281} &= 3^{276} \cdot 3^3 \cdot 3^2 \\ &= (3^6)^{46} \cdot 3^3 \cdot 3^2 \\ &\equiv 1^{46} \cdot -1 \cdot 9 \pmod{7} \\ &\equiv -9 \pmod{7} \\ &\equiv 5 \pmod{7} \end{aligned}$$

We also have  $4^3 = 2^6 = (2^3)^2 \equiv 1 \pmod{7}$ , so

$$\begin{aligned} 4^{370} &= 4^{369} \cdot 4 \\ &= (4^3)^{123} \cdot 4 \\ &\equiv 4 \pmod{7} \end{aligned}$$

Combining these using the addition and multiplication properties above, we get

$$\begin{aligned} 2^{203} \cdot 3^{281} + 4^{370} &\equiv 4 \cdot 5 + 4 \pmod{7} \\ &\equiv 24 \pmod{7} \\ &\equiv 3 \pmod{7} \end{aligned}$$

We can now reduce the full expression:

$$\begin{aligned} (2^{203} \cdot 3^{281} + 4^{370})^{376} &\equiv 3^{376} \pmod{7} \\ &\equiv 3^{372} \cdot 3^4 \pmod{7} \\ &\equiv (3^6)^{62} \cdot (3^2)^2 \pmod{7} \\ &\equiv 1^{62} \cdot 9^2 \pmod{7} \\ &\equiv 1 \cdot 2^2 \pmod{7} \\ &\equiv 4 \pmod{7} \end{aligned}$$

Thus,  $a \equiv 4 \pmod{7}$ .

### 23.1.1 Fast Modular Exponentiation

There are several ways to compute a modular exponentiation  $a^b \pmod{n}$  efficiently, and we take a look at two here.

The first is to apply a top-down, divide-and-conquer strategy. We have:

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ (a^{b/2})^2 & \text{if } b > 0 \text{ is even} \\ a \cdot (a^{(b-1)/2})^2 & \text{if } b > 0 \text{ is odd} \end{cases}$$

This leads to the following algorithm:

#### Algorithm 270 (Top-down Fast Modular Exponentiation)

```
function MODEXP( $a, b, n$ )
  if  $b = 0$  then return 1
   $m = \text{MODEXP}(a, \lfloor b/2 \rfloor, n)$ 
   $m = m \cdot m \pmod{n}$ 
  if  $b$  is odd then
     $m = a \cdot m \pmod{n}$ 
  return  $m$ 
```

This gives us the following recurrence for the number of multiplications and modulo operations:

$$T(b) = T(b/2) + O(1)$$

Applying the *Master theorem* (page 14), we get  $T(b) = O(\log b)$ .

Furthermore, the numbers in this algorithm are always computed modulo  $n$ , so they are at most as large as  $n$ . Thus, each multiplication and modulo operation can be done efficiently with respect to  $O(\log n)$ , and the algorithm as a whole is efficient.

An alternative method is to apply a bottom-up strategy. Here, we make use of the binary representation of  $b$ ,

$$b = b_r \cdot 2^r + b_{r-1} \cdot 2^{r-1} + \cdots + b_0 \cdot 2^0$$

where  $b_i$  is either 0 or 1. Then

$$\begin{aligned} a^b &= a^{b_r \cdot 2^r + b_{r-1} \cdot 2^{r-1} + \cdots + b_0 \cdot 2^0} \\ &= a^{b_r \cdot 2^r} \times a^{b_{r-1} \cdot 2^{r-1}} \times \cdots \times a^{b_0 \cdot 2^0} \end{aligned}$$

Thus, we can compute  $a^{2^i}$  for each  $0 \leq i \leq r$ , where  $r = \lfloor \log b \rfloor$ . We do so as follows:

**Algorithm 271 (Bottom-up Fast Modular Exponentiation)**

```

function ModExpBottomUp( $a, b, n$ )
    allocate powers[0, ...,  $\lfloor \log b \rfloor$ ]
    powers[0] =  $a$ 
    for  $i = 1$  to  $\lfloor \log b \rfloor$  do
        powers[i] = powers[i - 1] · powers[i - 1] mod  $n$ 
    prod = 1
    for  $i = 0$  to  $\lfloor \log b \rfloor$  do
        if  $\lfloor b/2^i \rfloor$  is odd then
            prod = prod · powers[i] mod  $n$ 
    return prod

```

The operation  $\lfloor b/2^i \rfloor$  is a right shift on the binary representation of  $b$ , and it can be done in linear time. As with the top-down algorithm, we perform  $O(\log b)$  multiplication and modulo operations, each on numbers that are  $O(\log n)$  in size. Thus, the runtime is efficient in the size of the input.

### 23.1.2 Division and Modular Inverses

We have seen how to do addition, subtraction, multiplication, and exponentiation in modular arithmetic, as well as several properties that help in reducing a complication expression using these operations. What about division? Division is not a closed operation over the set of integers, so it is perhaps not surprising that division is not always well-defined in modular arithmetic. However, for some combinations of  $n$  and  $a \in \mathbb{Z}_n$ , we can determine a *modular inverse* that allows us to divide by  $a$ . Recall that in standard arithmetic, dividing by a number  $x$  is equivalent to multiplying by the  $x^{-1}$ , the multiplicative inverse of  $x$ . For example:

$$21/4 = 21 \cdot 4^{-1} = 21 \cdot \frac{1}{4} = \frac{21}{4}$$

In the same way, division by  $a$  is defined modulo  $n$  exactly when an inverse  $a^{-1}$  exists for  $a$  modulo  $n$ .

**Theorem 272** Let  $n$  be a positive integer and  $a$  be an element of  $\mathbb{Z}_n^+$ . An inverse of  $a$  modulo  $n$  is an element  $b \in \mathbb{Z}_n^+$  such that

$$a \cdot b \equiv 1 \pmod{n}$$

An inverse of  $a$ , denoted as  $a^{-1}$ , exists modulo  $n$  if and only if  $a$  and  $n$  are coprime, i.e.  $\gcd(a, n) = 1$ .

A modular inverse can be efficiently found using the *extended Euclidean algorithm*, a modification of *Euclid's algorithm* (page 5).

**Algorithm 273 (Extended Euclidean Algorithm)**

**Input:** integers  $x \geq y \geq 0$ , not both zero

**Output:** their greatest common divisor  $g = \gcd(x, y)$ , and integers  $a, b$  such that  $ax + by = g$

```
function EXTENDED-EUCLID( $x, y$ )
    if  $y = 0$  then return  $(x, 1, 0)$ 
     $(g, a', b') = \text{EXTENDED-EUCLID}(y, x \bmod y)$ 
    return  $(g, b', a' - b' \cdot \lfloor x/y \rfloor)$ 
```

The complexity bound is *the same as for Euclid's algorithm* (page 10) –  $O(\log x)$  number of operations.

**Claim 274** Given  $x > y \geq 0$ , the extended Euclidean algorithm returns a triple  $(g, a, b)$  such that

$$g = ax + by$$

and  $g = \gcd(x, y)$ .

**Proof 275** We demonstrate that  $g = ax + by$  by (strong) induction over  $y$ :

- **Base case:**  $y = 0$ . The algorithm returns  $(g, a, b) = (x, 1, 0)$ , and we have

$$ax + by = 1 \cdot x + 0 \cdot 0 = x = g$$

- **Inductive step:**  $y > 0$ . Let  $x = qy + r$ , so that  $q = \lfloor x/y \rfloor$  and  $r = x \bmod y < y$ . Given arguments  $y$  and  $r$ , the recursion produces  $(g, a', b')$ . By the inductive hypothesis, we assume that

$$g = a'y + b'r$$

The algorithm computes the return values  $a$  and  $b$  as

$$\begin{aligned} a &= b' \\ b &= a' - b'q \end{aligned}$$

Then

$$\begin{aligned} ax + by &= b'x + (a' - b'q)y \\ &= b'(qy + r) + (a' - b'q)y \\ &= b'r + a'y \\ &= g \end{aligned}$$

Thus, we conclude that  $g = ax + by$  as required.

Given  $x$  and  $y$ , when  $g = \gcd(x, y) = 1$ , we have

$$\begin{aligned} ax &= 1 - by \\ by &= 1 - ax \end{aligned}$$

which imply that

$$\begin{aligned} ax &\equiv 1 \pmod{y} \\ by &\equiv 1 \pmod{x} \end{aligned}$$

by definition of modular arithmetic. Thus, the extended Euclidean algorithm computes  $a$  as the inverse of  $x$  modulo  $y$  and  $b$  as the inverse of  $y$  modulo  $x$ , when these inverses exist.

**Example 276** We compute the inverse of 13 modulo 21 using the extended Euclidean algorithm. Since 13 and 21 are coprime, such an inverse must exist.

We keep track of the values of  $x$ ,  $y$ ,  $g$ ,  $a$ , and  $b$  at each recursive step of the algorithm. First, we trace the algorithm from the initial  $x = 21$ ,  $y = 13$  down to the base case:

Step	$x$	$y$
0	21	13
1	13	8
2	8	5
3	5	3
4	3	2
5	2	1
6	1	0

We can then trace the algorithm back up, computing the return values  $g, a, b$ :

Step	$x$	$y$	$g$	$a$	$b$
6	1	0	1	1	0
5	2	1	1	0	$1 - 0 \cdot \lfloor \frac{2}{1} \rfloor = 1$
4	3	2	1	1	$0 - 1 \cdot \lfloor \frac{3}{2} \rfloor = -1$
3	5	3	1	-1	$1 - (-1) \cdot \lfloor \frac{5}{3} \rfloor = 2$
2	8	5	1	2	$-1 - 2 \cdot \lfloor \frac{8}{5} \rfloor = -3$
1	13	8	1	-3	$2 - (-3) \cdot \lfloor \frac{13}{8} \rfloor = 5$
0	21	13	1	5	$-3 - 5 \cdot \lfloor \frac{21}{13} \rfloor = -8$

Thus, we have  $by = -8 \cdot 13 \equiv 1 \pmod{21}$ . Translating  $b$  to an element of  $\mathbb{Z}_{21}$ , we get  $b = -8 \equiv 13 \pmod{21}$ , which means that 13 is its own inverse modulo 21. We can verify this:

$$13 \cdot 13 = 169 = 8 \cdot 21 + 1 \equiv 1 \pmod{21}$$

**Exercise 277** The extended Euclidean algorithm is a constructive proof that when  $\gcd(a, n) = 1$  for  $n \in \mathbb{Z}^+$  and  $a \in \mathbb{Z}_n^+$ , an inverse of  $a$  exists modulo  $n$ . Show that no such inverse exists when  $\gcd(a, n) > 1$ , completing the proof of Theorem 272.

## 23.2 One-time Pad

We now take a look at the first category of cryptosystems, that of information-theoretic security, which relies on one-time pads.

**Definition 278 (One-time Pad)** A *one-time pad* is an encryption technique that relies on a key with the following properties:

- The key is a random string at least as long as the plaintext.
- The key is preshared between the communicating parties over some secure channel.
- The key is only used once (hence the name *one-time pad*).

As an example of a scheme that uses a one-time pad, consider a plaintext message

$$m = m_1 m_2 \dots m_n$$

where each symbol  $m_i$  is a lowercase English letter. Let the secret key also be composed of lowercase letters:

$$k = k_1 k_2 \dots k_n$$

Here, the key is the same length as the message. We encrypt the message as

$$E_k(m) = c_1 c_2 \dots c_n$$

by taking the sum of each plaintext character and the corresponding key character modulo 26:

$$c_i \equiv m_i + k_i \pmod{26}$$

We map between lowercase characters and integers modulo 26, with 0 taken to be the letter  $a$ , 1 to be the letter  $b$ , and so on. Then decryption is as follows:

$$D_k(c) = d_1 d_2 \dots d_n, \text{ where } d_i \equiv c_i - k_i \pmod{26}$$

Applying both operations results in  $D_k(E_k(m)) = m$ , the original plaintext message.

As a concrete example, suppose  $m = \text{flower}$  and  $k = \text{lafswl}$ . Then the ciphertext is

$$E_k(m) = \text{qltoac}$$

and we have  $D_k(E_k(m)) = \text{flower}$ .

Observe that Eve has no means of learning any information about  $m$  from the ciphertext `qltoac`, other than that the message is six letters (and we can avoid even that by padding the message with random additional data). Even if Eve knows that the message is in English, she has no way of determining which six-letter word it is. For instance, the ciphertext `qltoac` could have been produced by the plaintext `futile` and key `1ragpy`. In fact, for **any** six-letter sequence  $s$ , there is a key  $k$  such that  $E_k(s) = c$  for any six-letter ciphertext  $c$ . Without having access to either the plaintext or key directly, Eve cannot tell whether the original message is `flower`, `futile`, or some other six-letter sequence.

Thus, a one-time-pad scheme provides information-theoretic security – an adversary cannot recover information about the message that they do not already know. In fact, one-time-pad schemes are the **only** cryptosystems that provide this level of security. However, there are significant tradeoffs, which are exactly the core requirements of a one-time pad:

- The key must be preshared between the communicating parties through some other, secure channel.
- The key has to be as long as the message, which limits the amount of information that can be communicated given a preshared key.
- The key can only be used once.

These limitations make one-time pads costly to use in practice. Perhaps by relaxing some of these restrictions, we can obtain “good-enough” security at a lower cost? We first take a look at what happens when we reduce the key size – in fact, we will take this to the extreme, reducing our key size to a single symbol. This results in a scheme known as a *Caesar cipher*.

**Definition 279 (Caesar Cipher)** Let  $m = m_1 m_2 \dots m_n$  be a plaintext message. Let  $s$  be a single symbol to be used as the key. Then in a Caesar cipher,  $m$  is encrypted as

$$E_s(m) = c_1 c_2 \dots c_n, \text{ where } c_i \equiv m_i + s \pmod{26}$$

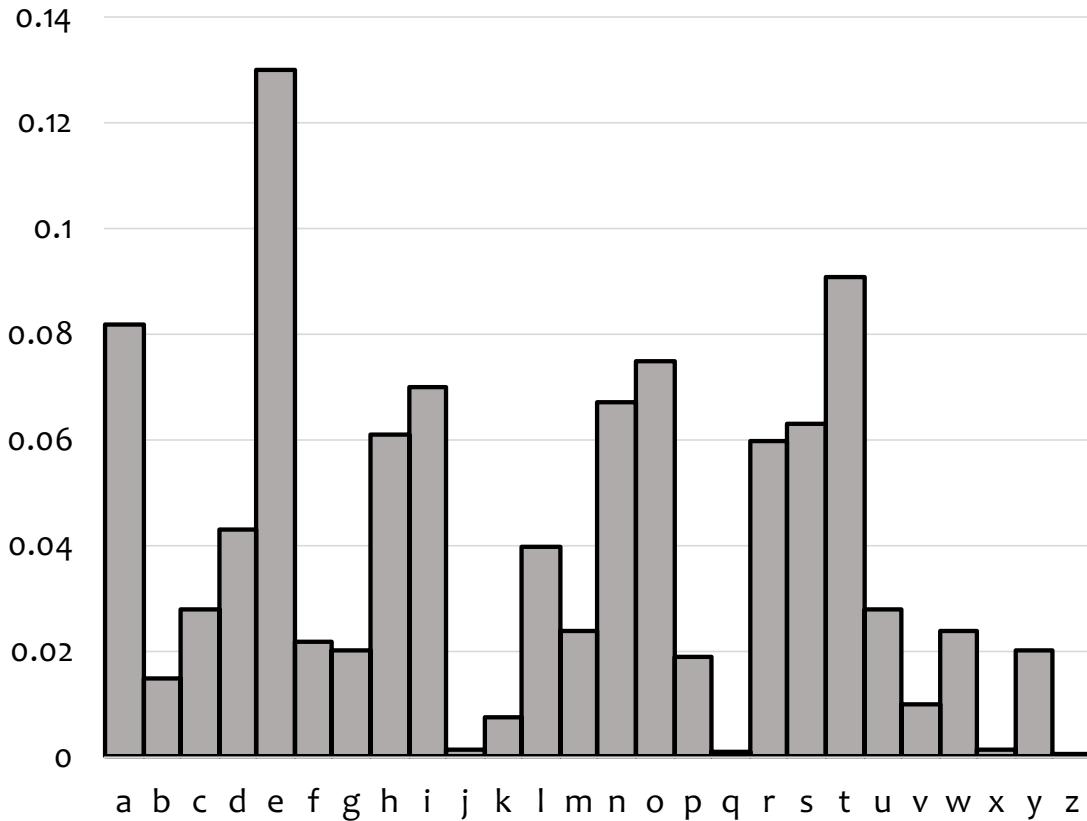
and a ciphertext  $c$  is decrypted as

$$D_s(c) = d_1 d_2 \dots d_n, \text{ where } d_i \equiv c_i - s \pmod{26}$$

The result is  $D_s(E_s(m)) = m$ .

Observe that a Caesar cipher is similar to a one-time pad, except that the key only has one character of randomness as opposed to  $n$ ; essentially, we have  $k_i = s$  for all  $i$ , i.e. a key where all the characters are the same.

Unfortunately, the Caesar cipher suffers from a fatal flaw in that it can be defeated by statistical analysis – in particular, the relative frequencies of letters in the ciphertext allow symbols to be mapped to the underlying message, using a frequency table of how common individual letters are in the language in which the message is written. In English, for instance, the letters *e* and *t* are most common. A full graph of frequencies of English letters in “average” English text is as follows:



A Caesar cipher merely does a circular shift of these frequencies, making it straightforward to recover the key as the magnitude of that shift. The longer the message, the more likely the underlying frequencies match the average for the language, and the more easily the scheme is broken.

**Exercise 280** Suppose the result of applying a Caesar cipher produces the ciphertext

$$E_s(m) = \text{cadcq}$$

Use frequency analysis to determine both the original message  $m$  and the key  $s$ .

Note that another weakness in the Caesar-cipher scheme as described above is that the key is restricted to one of twenty-six possibilities, making it trivial to brute force the mapping. However, the scheme can be tweaked to use a much larger character set, making it harder to brute force but still leaving it open to statistical attacks in the form of frequency analysis.

A scheme that compromises between a one-time pad and Caesar cipher, by making the key larger than a single character but smaller than the message size, still allows information to leak through statistical attacks. Such a scheme is essentially the same as reusing a one-time pad more than once. What can go wrong if we do so?

Suppose we have the following two plaintext messages

$$\begin{aligned} m &= m_1 m_2 \dots m_n \\ m' &= m'_1 m'_2 \dots m'_n \end{aligned}$$

where each character is a lowercase English letter. If we encode them both with the same key  $k = k_1 k_2 \dots k_n$ , we obtain the ciphertexts

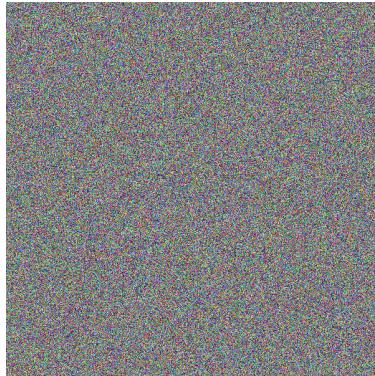
$$\begin{aligned} E_k(m) &= c_1 c_2 \dots c_n, \text{ where } c_i \equiv m_i + k_i \pmod{26} \\ E_k(m') &= c'_1 c'_2 \dots c'_n, \text{ where } c'_i \equiv m'_i + k_i \pmod{26} \end{aligned}$$

If both these ciphertexts go out over the insecure channel, Eve can observe them both and compute their difference:

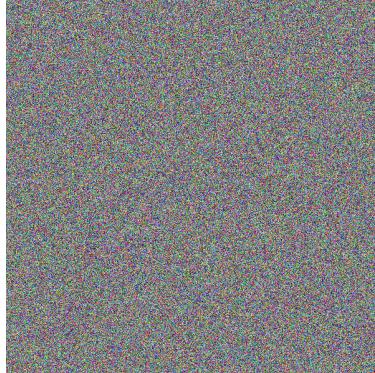
$$\begin{aligned} E_k(m) - E_k(m') &= (c_1 - c'_1)(c_2 - c'_2) \dots (c_n - c'_n) \\ &= (m_1 - m'_1)(m_2 - m'_2) \dots (m_n - m'_n) \\ &= m - m' \end{aligned}$$

Here, all subtractions are done modulo 26. The end result is the character-by-character difference between the two messages (modulo 26). Unless the plaintext messages are random strings, this difference is not random! Again, statistical attacks can be used to obtain information about the original messages.

As a pictorial illustration of how the difference of two messages reveals information, the following is an image encoded with a random key under a one-time pad:



The result appears as just random noise, as we would expect. The following is another image encoded with the same key:



This result too appears as random noise. However it is the **same** noise, and if we subtract the two ciphertexts, the noise all cancels out:



The end result clearly reveals information about the original images.

In summary, the only way to obtain information-theoretic security is by using a one-time-pad scheme, where the key is at least as long as the message and is truly used only once. A one-time-pad-like scheme that compromises on these two characteristics opens up the scheme to statistical attacks. The core issue is that **plaintext messages are not random** – they convey information between parties by virtue of not being random. In a one-time pad, the key is the actual source of the randomness in the ciphertext. And if we weaken the scheme by reducing the key size or reusing a key, we lose enough randomness to enable an adversary to learn information about the plaintext messages.

---

CHAPTER  
TWENTYFOUR

---

## DIFFIE-HELLMAN KEY EXCHANGE

Many encryption schemes, including a one-time pad, are *symmetric*, using the same key for both encryption and decryption. Such a scheme requires a preshared secret key that is known to both communicating parties. Ideally, we'd like the two parties to be able to establish a shared key even if all their communication is over an insecure channel. This is known as the *key exchange* problem.

One solution to the key-exchange problem is the *Diffie-Hellman* protocol. The central idea is that each party has its own secret key – this is a *private key*, since it is never shared with anyone. They each use their own private key to generate a *public key*, which they transmit to each other over the insecure channel. A public key is generated in such a way that recovering the private key would require solving a computationally hard problem, resulting in computational rather than information-theoretic security. Finally, each party uses its own private key and the other party's public key to obtain a shared secret.

Before we examine the details of the Diffie-Hellman protocol, we discuss some relevant concepts from modular arithmetic. Let  $\mathbb{Z}_q$  refer to the set of integers between 0 and  $q - 1$ , inclusive:

$$\mathbb{Z}_q = \{0, 1, 2, \dots, q - 1\}$$

We then define a *generator* of  $\mathbb{Z}_p$ , where  $p$  is prime, as follows:

**Definition 281 (Generator)** An element  $g \in \mathbb{Z}_p$ , where  $p$  is prime, is a *generator* of  $\mathbb{Z}_p$  if for every nonzero element  $x \in \mathbb{Z}_p$  (i.e. every element in  $\mathbb{Z}_p^+$ ), there exists a number  $i \in \mathbb{N}$  such that:

$$g^i \equiv x \pmod{p}$$

In other words,  $g$  is an  $i$ th root of  $x$  over  $\mathbb{Z}_p$  for some natural number  $i$ .

As an example,  $g = 2$  is a generator of  $\mathbb{Z}_5$ :

$$\begin{aligned} 2^0 &= 1 \equiv 1 \pmod{5} \\ 2^1 &= 2 \equiv 2 \pmod{5} \\ 2^2 &= 4 \equiv 4 \pmod{5} \\ 2^3 &= 8 \equiv 3 \pmod{5} \end{aligned}$$

Similarly,  $g = 3$  is a generator of  $\mathbb{Z}_7$ :

$$\begin{aligned} 3^0 &= 1 \equiv 1 \pmod{7} \\ 3^1 &= 3 \equiv 3 \pmod{7} \\ 3^2 &= 9 \equiv 2 \pmod{7} \\ 3^3 &= 27 \equiv 6 \pmod{7} \\ 3^4 &= 81 \equiv 4 \pmod{7} \\ 3^5 &= 243 \equiv 5 \pmod{7} \end{aligned}$$

On the other hand,  $g = 2$  is not a generator of  $\mathbb{Z}_7$ :

$$\begin{aligned} 2^0 &= 1 \equiv 1 \pmod{7} \\ 2^1 &= 2 \equiv 2 \pmod{7} \\ 2^2 &= 4 \equiv 4 \pmod{7} \\ 2^3 &= 8 \equiv 1 \pmod{7} \\ 2^4 &= 16 \equiv 2 \pmod{7} \\ 2^5 &= 32 \equiv 4 \pmod{7} \\ &\dots \end{aligned}$$

We see that the powers of  $g = 2$  are cyclic, without ever generating the elements  $\{3, 5, 6\} \subseteq \mathbb{Z}_7^+$ .

**Theorem 282** *If  $p$  is prime, then  $\mathbb{Z}_p$  has at least one generator.*

This theorem was first proved by Gauss<sup>89</sup>. Moreover,  $\mathbb{Z}_p$  has  $\phi(p-1)$  generators when  $p$  is prime, where  $\phi(n)$  is Euler's totient function<sup>90</sup>, making it relatively easy to find a generator over  $\mathbb{Z}_p$ .

We now return to the Diffie-Hellman protocol, which is as follows:

**Protocol 283 (Diffie-Hellman)** Suppose two parties, henceforth referred to as *Alice* and *Bob*, wish to establish a shared secret key  $k$  but can only communicate over an insecure channel. Alice and Bob first establish public parameters  $p$ , a very large prime number, and  $g$ , a generator of  $\mathbb{Z}_p$ . Then:

- Alice picks a random  $a \in \mathbb{Z}_p^+$  as her private key, computes  $A \equiv g^a \pmod{p}$  as her public key, and sends  $A$  to Bob over the insecure channel.
- Bob picks a random  $b \in \mathbb{Z}_p^+$  as his private key, computes  $B \equiv g^b \pmod{p}$  as his public key, and sends  $B$  to Alice over the insecure channel.
- Alice computes

$$k \equiv B^a \equiv (g^b)^a \equiv g^{ab} \pmod{p}$$

as the shared secret key.

- Bob computes

$$k \equiv A^b \equiv (g^a)^b \equiv g^{ab} \pmod{p}$$

as the shared secret key.

This protocol is efficient – suitable values of  $p$  and  $g$  can be obtained from public tables, and Alice and Bob each need to perform two modular exponentiations, which *can be done efficiently* (page 245). But is it secure? By Kerckhoff's principle, Eve knows how the protocol works, and she observes the values  $p$ ,  $g$ ,  $A \equiv g^a \pmod{p}$ , and  $B \equiv g^b \pmod{p}$ . However, she does not know  $a$  or  $b$ , since those values are never communicated and so are known only to Alice and Bob, respectively. Can Eve obtain  $k \equiv g^{ab} \pmod{p}$  from what she knows? The *Diffie-Hellman assumption* states that she cannot do so efficiently.

**Claim 284 (Diffie-Hellman Assumption)** *There is no efficient algorithm that computes  $g^{ab} \pmod{p}$  given:*

- a prime  $p$ ,
- a generator  $g$  of  $\mathbb{Z}_p$ ,

<sup>89</sup> [https://en.wikipedia.org/wiki/Carl\\_Friedrich\\_Gauss](https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss)

<sup>90</sup> [https://en.wikipedia.org/wiki/Euler%27s\\_totient\\_function](https://en.wikipedia.org/wiki/Euler%27s_totient_function)

- $g^a \pmod{p}$ , and
- $g^b \pmod{p}$ .

What if, rather than attempting to recover  $g^{ab} \pmod{p}$  directly, Eve attempts to recover one of the private keys  $a$  or  $b$ ? This entails solving the *discrete logarithm* problem.

**Definition 285 (Discrete Logarithm)** Let  $q$  be a modulus and let  $g$  be a generator of  $\mathbb{Z}_q$ . The *discrete logarithm* of  $x \in \mathbb{Z}_q^+$  with respect to the base  $g$  is an integer  $i$  such that

$$g^i \equiv x \pmod{q}$$

The discrete logarithm problem is the task of computing  $i$ , given  $q$ ,  $g$ , and  $x$ .

If  $q$  is prime and  $g$  is a generator of  $\mathbb{Z}_q$ , then every  $x \in \mathbb{Z}_q^+$  has a discrete log  $i$  with respect to base  $g$  such that  $0 \leq i < q - 1$ <sup>91</sup>. Thus, a brute-force algorithm to compute the discrete log is to try every possible value in this range. However, such an algorithm requires checking  $O(q)$  possible values, and this is exponential in the size of the inputs, which take  $O(\log q)$  bits to represent. The brute-force algorithm is thus inefficient.

Do better algorithms exist for computing a discrete log? Indeed they do, including the [baby-step giant-step algorithm](#)<sup>92</sup>, which requires  $O(\sqrt{q})$  multiplications on numbers of size  $O(\log q)$ . However, as we saw in [primality testing](#) (page 270), this is still not polynomial in the input size  $O(\log q)$ . In fact, the *discrete logarithm assumption* states that no efficient algorithm exists.

**Claim 286 (Discrete Logarithm Assumption)** *There is no efficient algorithm to compute  $i$  given:*

- $q$ ,
- a generator  $g$  of  $\mathbb{Z}_q$ , and
- $g^i \pmod{q}$ ,

where  $0 \leq i < q - 1$ .

Neither the Diffie-Hellman nor the discrete logarithm assumption have been proven. However, they have both held up in practice, as there is no known algorithm to defeat either assumption on classical computers. As we will briefly discuss later, the assumptions do not hold on [quantum computers](#) (page 261), but this is not yet a problem in practice as no scalable quantum computer exists.

<sup>91</sup> This is a consequence of [Fermat's little theorem](#) (page 256), which states that  $x^{q-1} \equiv 1 \pmod{q}$  for prime  $q$  and  $x \in \mathbb{Z}_q^+$ .

<sup>92</sup> [https://en.wikipedia.org/wiki/Baby-step\\_giant-step](https://en.wikipedia.org/wiki/Baby-step_giant-step)

---

CHAPTER  
TWENTYFIVE

---

RSA

The Diffie-Hellman protocol uses private and public keys for key exchange, resulting in a shared key known to both communicating parties, which can then be used in a symmetric encryption scheme. However, the concept of a *public-key cryptosystem* can also be used to formulate an *asymmetric* encryption protocol, which uses different keys for encryption and decryption. The *RSA (Rivest-Shamir-Adleman)* cryptosystem gives rise to one such protocol that is widely used in practice.

As with Diffie-Hellman, the RSA system relies on facts about modular arithmetic. Core to the working of RSA is *Fermat's little theorem* and its variants.

**Theorem 287 (Fermat's Little Theorem)** *Let  $p$  be a prime number. Let  $a$  be any element of  $\mathbb{Z}_p^+$ , where*

$$\mathbb{Z}_p^+ = \{1, 2, \dots, p-1\}$$

*Then  $a^{p-1} \equiv 1 \pmod{p}$ .*

**Corollary 288** *Let  $p$  be a prime number. Then  $a^p \equiv a \pmod{p}$  for any element  $a \in \mathbb{Z}_p$ , where*

$$\mathbb{Z}_p = \{0, 1, \dots, p-1\}$$

As an example, let  $p = 7$ . Then:

$$\begin{array}{lll} 0^7 &= 0 & \equiv 0 \pmod{7} \\ 1^7 &= 1 & \equiv 1 \pmod{7} \\ 2^7 &= 128 = 2 + 18 \cdot 7 & \equiv 2 \pmod{7} \\ 3^7 &= 2187 = 3 + 312 \cdot 7 & \equiv 3 \pmod{7} \\ 4^7 &= 16384 = 4 + 2340 \cdot 7 & \equiv 4 \pmod{7} \\ 5^7 &= 78125 = 5 + 11160 \cdot 7 & \equiv 5 \pmod{7} \\ 6^7 &= 279936 = 6 + 39990 \cdot 7 & \equiv 6 \pmod{7} \end{array}$$

Fermat's little theorem can be extended to the product of two distinct primes.

**Theorem 289** *Let  $n = pq$  be a product of two distinct primes, i.e.  $p \neq q$ . Let  $a$  be an element of  $\mathbb{Z}_n$ . Then*

$$a^{1+k\phi(n)} \equiv a \pmod{n}$$

*where  $k \in \mathbb{N}$  and  $\phi(n)$  is Euler's totient function, whose value is the number of elements in  $\mathbb{Z}_n^+$  that are coprime to  $n$ . For a product of two distinct primes  $n = pq$ ,*

$$\phi(pq) = (p-1)(q-1)$$

**Proof 290** We prove Theorem 289 using Fermat's little theorem. First, we show that for any  $a \in \mathbb{Z}_n$  where  $n = pq$  is the product of two distinct primes  $p$  and  $q$ , if  $a \equiv b \pmod{p}$  and  $a \equiv b \pmod{q}$ , then  $a \equiv b \pmod{n}$ . By

definition of modular arithmetic, we have

$$\begin{aligned} a &= kp + b \quad (\text{since } a \equiv b \pmod{p}) \\ a &= mq + b \quad (\text{since } a \equiv b \pmod{q}) \end{aligned}$$

where  $k$  and  $m$  are integers. Then:

$$\begin{aligned} kp + b &= mq + b \\ kp &= mq \end{aligned}$$

Since  $p$  divides  $kp$ ,  $p$  also divides  $mq$ ; however, since  $p$  is a distinct prime from  $q$ , this implies that  $p$  divides  $m$  (by Euclid's lemma<sup>93</sup>, which states that if a prime  $p$  divides  $ab$  where  $a, b \in \mathbb{Z}$ , then  $p$  must divide at least one of  $a$  or  $b$ ). Thus,  $m = rp$  for some integer  $r$ , and we have

$$\begin{aligned} a &= mq + b \\ &= rpq + b \\ &\equiv b \pmod{pq} \end{aligned}$$

Now consider  $a^{1+k\phi(n)}$ . If  $a = 0$ , we trivially have

$$0^{1+k\phi(n)} \equiv 0 \pmod{n}$$

If  $a \neq 0$ , we have:

$$\begin{aligned} a^{1+k\phi(n)} &= a^{1+k(p-1)(q-1)} \\ &= a \cdot (a^{p-1})^{k(q-1)} \\ &\equiv a \cdot 1^{k(q-1)} \pmod{p} \quad (\text{by Fermat's little theorem}) \\ &\equiv a \pmod{p} \end{aligned}$$

Similarly:

$$\begin{aligned} a^{1+k\phi(n)} &= a^{1+k(p-1)(q-1)} \\ &= a \cdot (a^{q-1})^{k(p-1)} \\ &\equiv a \cdot 1^{k(p-1)} \pmod{q} \quad (\text{by Fermat's little theorem}) \\ &\equiv a \pmod{q} \end{aligned}$$

Applying our earlier result, we conclude that  $a^{1+k\phi(n)} \equiv a \pmod{n}$ . □

---

<sup>93</sup> [https://en.wikipedia.org/wiki/Euclid%27s\\_lemma](https://en.wikipedia.org/wiki/Euclid%27s_lemma)

As an example, let  $n = 6$ . We have  $\phi(n) = 2$ . Then:

$$\begin{aligned} 0^3 &= 0 & \equiv 0 \pmod{6} \\ 1^3 &= 1 & \equiv 1 \pmod{6} \\ 2^3 &= 8 = 2 + 1 \cdot 6 & \equiv 2 \pmod{6} \\ 3^3 &= 27 = 3 + 4 \cdot 6 & \equiv 3 \pmod{6} \\ 4^3 &= 64 = 4 + 10 \cdot 6 & \equiv 4 \pmod{6} \\ 5^3 &= 125 = 5 + 20 \cdot 6 & \equiv 5 \pmod{6} \end{aligned}$$

### Proof of Fermat's Little Theorem

Fermat's little theorem has many proofs; we take a look at a proof that relies solely on the existence of modular

inverses (Theorem 272).

Let  $p$  be prime, and let  $a \in \mathbb{Z}_p^+$  be an arbitrary nonzero element of  $\mathbb{Z}_p$ . Consider the set of elements

$$S = \{1a, 2a, \dots, (p-1)a\}$$

These elements are all nonzero and distinct when taken modulo  $p$ :

- Suppose  $ka \equiv 0 \pmod{p}$ . Since  $a \in \mathbb{Z}_p^+$ , it has an inverse  $a^{-1}$  modulo  $p$ , so we can multiply both sides by  $a^{-1}$  to obtain  $k \equiv 0 \pmod{p}$ . Since  $k \not\equiv 0 \pmod{p}$  for all elements in  $\{1, 2, \dots, p-1\}$ ,  $ka \not\equiv 0 \pmod{p}$  for all elements  $ka \in S$ .
- Suppose  $ka \equiv ma \pmod{p}$ . Multiplying both sides by  $a^{-1}$ , we obtain  $k \equiv m \pmod{p}$ . Since all pairs of elements in  $\{1, 2, \dots, p-1\}$  are distinct modulo  $p$ , all pairs of elements in  $S = \{1a, 2a, \dots, (p-1)a\}$  are also distinct modulo  $p$ .

Since there are  $p-1$  elements in  $S$ , and they are all nonzero and distinct modulo  $p$ , the elements of  $S$  when taken modulo  $p$  are exactly those in  $\mathbb{Z}_p^+$ . Thus, the products of the elements in each set are equivalent modulo  $p$ :

$$\begin{aligned} 1a \times 2a \times \dots \times (p-1)a &\equiv 1 \times 2 \times \dots \times (p-1) \pmod{p} \\ (1 \times 2 \times \dots \times (p-1)) \cdot a^{p-1} &\equiv 1 \times 2 \times \dots \times (p-1) \pmod{p} \end{aligned}$$

Since  $p$  is prime, all elements in  $\mathbb{Z}_p^+$  have an inverse modulo  $p$ , so we multiply both sides above by  $(1^{-1} \times 2^{-1} \times \dots \times (p-1)^{-1})$  to obtain

$$a^{p-1} \equiv 1 \pmod{p}$$

Now that we have established the underlying mathematical facts, we take a look at the RSA encryption protocol.

**Protocol 291 (RSA Encryption)** Suppose Bob wishes to send a message  $m$  to Alice, but the two can only communicate over an insecure channel.

- Alice chooses two very large, distinct primes  $p$  and  $q$ . We assume without loss of generality that  $m$ , when interpreted as an integer (using the bitstring representation of  $m$ ), is smaller than  $n = pq$ ; otherwise,  $m$  can be divided into smaller pieces that are then sent separately using this protocol.
- Alice chooses an element

$$e \in \mathbb{Z}_n^+ \text{ such that } \gcd(e, \phi(n)) = 1$$

as her public key, with the requirement that it be coprime with  $\phi(n) = (p-1)(q-1)$ . She computes the inverse of  $e$  modulo  $\phi(n)$

$$d \equiv e^{-1} \pmod{\phi(n)}$$

as her private key.

- Alice sends  $n$  and  $e$  to Bob.
- Bob computes

$$c \equiv m^e \pmod{n}$$

as the ciphertext and sends it to Alice.

- Alice computes

$$m' \equiv c^d \pmod{n}$$

with the result that  $m' = m$ .

This protocol is efficient. Alice can find large primes using an efficient primality test such as the *Fermat test* (page 271) – in fact, she need only do this once, reusing the same parameters  $n, e, d$  for future communication. Computing the inverse of  $e$  can be done efficiently using the extended Euclidean algorithm. Finally, modular exponentiation *can also be done efficiently* (page 245).

Does the protocol work? We have

$$m' \equiv c^d \equiv m^{ed} \pmod{n}$$

Since  $d \equiv e^{-1} \pmod{\phi(n)}$ , we have

$$\begin{aligned} ed &\equiv 1 \pmod{\phi(n)} \\ &= 1 + k\phi(n) \end{aligned}$$

by definition of modular arithmetic, where  $k \in \mathbb{N}$ . By [Theorem 289](#),

$$\begin{aligned} m' &\equiv m^{ed} \pmod{n} \\ &\equiv m^{1+k\phi(n)} \pmod{n} \\ &\equiv m \pmod{n} \end{aligned}$$

Thus, Alice does indeed recover the intended message  $m$ .

Finally, is the protocol secure? The public information that Eve can observe consists of  $n, e$ , and  $c \equiv m^e \pmod{n}$ . Eve **does not** know the private parameters  $p, q, d$ , which Alice has kept to herself. Can Eve recover  $m$ ? The *RSA assumption* states that she cannot do so efficiently.

**Claim 292 (RSA Assumption)** *There is no algorithm that efficiently computes  $m$  given:*

- a product of two distinct primes  $n$ ,
- an element  $e \in \mathbb{Z}_n^+$  such that  $\gcd(e, \phi(n)) = 1$ , and
- $m^e \pmod{n}$ .

Rather than trying to compute  $m$  directly, Eve could attempt to compute  $\phi(n)$ , which would allow her to compute  $d \equiv e^{-1} \pmod{\phi(n)}$  and thus decrypt the ciphertext  $c \equiv m^e \pmod{n}$  using the same process as Alice. However, recovering  $\phi(n) = (p-1)(q-1)$  is as hard as factoring  $n = pq$ , and the *factorization hardness assumption* states that she cannot do this efficiently.

**Claim 293 (Factorization Hardness Assumption)** *There is no efficient algorithm to compute the prime factorization  $p_1, p_2, \dots, p_k$  of an integer  $n$ , where*

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdots \cdot p_k^{a_k}$$

for  $a_i \in \mathbb{Z}^+$ .

Like the Diffie-Hellman and discrete logarithm assumptions, the RSA and factorization hardness assumptions have not been proven, but they appear to hold in practice. And like the former two assumptions, the latter two do not hold on *quantum computers* (page 261); we will discuss this momentarily.

**Exercise 294** Suppose that  $n = pq$  is a product of two distinct primes  $p$  and  $q$ . Demonstrate that obtaining  $\phi(n)$  is as hard as obtaining  $p$  and  $q$  by showing that given  $\phi(n)$ , the prime factors  $p$  and  $q$  can be computed efficiently.

**Exercise 295** The *ElGamal encryption scheme* relies on the hardness of the discrete logarithm problem to perform encryption, like Diffie-Hellman does for key exchange. The following describes how Bob can send an encrypted message to Alice using ElGamal. Assume that a large prime  $p$  and generator  $g$  of  $\mathbb{Z}_p^*$  have already been established.

- Alice chooses a private key  $a \in \mathbb{Z}_p^*$ , computes  $A \equiv g^a \pmod{p}$  as her public key, and sends  $A$  to Bob.
- Bob chooses a private key  $b \in \mathbb{Z}_p^*$ , computes  $B \equiv g^b \pmod{p}$  as his public key, and sends  $B$  to Alice.
- Alice and Bob both compute the shared key  $k \equiv g^{ab} \pmod{p}$ .
- Bob encrypts the message  $m$  as

$$c \equiv m \cdot k \equiv m \cdot g^{ab} \pmod{p}$$

and sends the ciphertext  $c$  to Alice.

- Show how Alice can recover  $m$  from the information she knows  $(p, g, a, A, B, k, c)$ .
- Demonstrate that if Eve has an efficient algorithm for recovering  $m$  from what she can observe  $(p, g, A, B, c)$ , she also has an efficient method for breaking Diffie-Hellman key exchange.

## 25.1 RSA Signatures

Thus far, we have focused on the privacy and confidentiality provided by encryption protocols. We briefly consider authentication and integrity, in the form of a *signature* scheme using RSA. The goal here is not to keep a secret – rather, given a message, we want to verify the identity of the author as well as the integrity of its contents. The way we do so is to essentially run the RSA encryption protocol “backwards” – rather than having Bob run the encryption process on a plaintext message and Alice the decryption on a ciphertext, we will have Alice apply the decryption function to a message she wants to sign, and Bob will apply the encryption function to the resulting signed message.

**Protocol 296 (RSA Signature)** Suppose Alice wishes to send a message  $m$  to Bob and allow Bob to verify that he receives the intended message. As with the *RSA encryption protocol* (page 258), Alice computes  $(e, n)$  as her public key and sends them to Bob, and she computes  $d \equiv e^{-1} \pmod{\phi(n)}$  as her private key. Then:

- Alice computes

$$s \equiv m^d \pmod{n}$$

and sends  $m$  and  $s$  to Bob.

- Bob computes

$$m' \equiv s^e \pmod{n}$$

and verifies that the result is equal to  $m$ .

The correctness of this scheme follows from the correctness of RSA encryption; we have:

$$\begin{aligned} m' &\equiv s^e \pmod{n} \\ &\equiv m^{ed} \pmod{n} \\ &\equiv m^{1+k\phi(n)} \pmod{n} \\ &\equiv m \pmod{n} \end{aligned}$$

Thus, if  $m' = m$ , Bob can be assured that Alice sent the message – only she knows  $d$ , so only she can efficiently compute  $m^d \pmod{n}$ . In essence, the pair  $m, m^d \pmod{n}$  acts as a *certificate* (page 140) that the sender knows the secret key  $d$ .

In practice, this scheme needs a few more details to avoid the possibility of *spoofing*, or having someone else send a message purporting to be from Alice. We leave these details as an exercise.

**Exercise 297** The RSA signature scheme as described above is subject to spoofing – it is possible for a third party to produce a pair  $m, s$  such that  $s \equiv m^d \pmod{n}$ .

- Describe one way in which someone other than Alice can produce a matching pair  $m, m^d \pmod{n}$ .
- Propose a modification to the scheme that addresses this issue.

**Hint:** Think about adding some form of padding to the message to assist in verifying that Alice was the author.

## 25.2 Quantum Computers and Cryptography

The abstraction of *standard Turing machines* (page 61) does not seem to quite capture the operation of *quantum computers*, but there are other models such as *quantum Turing machines*<sup>94</sup> and *quantum circuits*<sup>95</sup> that do so. The *standard Church-Turing thesis* (page 84) still applies – anything that can be computed on a quantum computer can be computed on a Turing machine. However, a quantum computer is a probabilistic model, so the *extended Church-Turing thesis* (page 139) does not apply – we do not know whether quantum computers can solve problems more efficiently than so-called *classical computers*. There are problems that are known to have efficient probabilistic algorithms on quantum computers but do not have known, efficient probabilistic algorithms on classical computers. Discrete logarithm and integer factorization, the problems that are core to Diffie-Hellman and RSA, are two such problems; they are both efficiently solvable on quantum computers by Shor's algorithm<sup>96</sup>.

In practice, scalable quantum computers pose significant implementation challenges, and they are a long way from posing a risk to the security of Diffie-Hellman and RSA. As of this writing, the largest number known to have been factored by Shor's algorithm on a quantum computer is 35, which was accomplished in 2019. This follows previous records of 21 in 2012 and 15 in 2001. Compare this to factorization on classical computers, where the 829-bit RSA-250 number<sup>97</sup> was factored in 2020. Typical keys currently used in implementations of RSA have at least 2048 bits, and both quantum and classical computers are far from attacking numbers of this size. Regardless, *post-quantum cryptography*<sup>98</sup> is an active area of research, so that if quantum computers do eventually become powerful enough to attack Diffie-Hellman or RSA, other cryptosystems can be used that are more resilient against quantum attacks.

In complexity-theoretic terms, the decision versions of discrete logarithm and integer factorization are in the complexity class BQP, which is the quantum analogue of BPP; in other words, BQP is the class of languages that have efficient *two-sided-error randomized algorithms* (page 283) on quantum computers. We know that

$$\text{BPP} \subseteq \text{BQP}$$

but we do not know whether this containment is strict. And like the relationship between BPP and NP, most complexity theorists do not believe that BQP contains all of NP. Neither the decision versions of discrete logarithm nor of integer factorization are believed to be NP-complete – they are believed (but not proven) to be *NP-Intermediate*, i.e. in the class NPI of languages that are in NP but neither in P nor NP-complete. It is known that if  $P \neq NP$ , then the class NPI is not empty.

On the other hand, if  $P = NP$ , then computational security is not possible – such security relies on the existence of hard problems that are efficiently verifiable, and no such problem exists if  $P = NP$ .

<sup>94</sup> [https://en.wikipedia.org/wiki/Quantum\\_Turing\\_machine](https://en.wikipedia.org/wiki/Quantum_Turing_machine)

<sup>95</sup> [https://en.wikipedia.org/wiki/Quantum\\_circuit](https://en.wikipedia.org/wiki/Quantum_circuit)

<sup>96</sup> [https://en.wikipedia.org/wiki/Shor%27s\\_algorithm](https://en.wikipedia.org/wiki/Shor%27s_algorithm)

<sup>97</sup> [https://en.wikipedia.org/wiki/RSA\\_numbers#RSA-250](https://en.wikipedia.org/wiki/RSA_numbers#RSA-250)

<sup>98</sup> [https://en.wikipedia.org/wiki/Post-quantum\\_cryptography](https://en.wikipedia.org/wiki/Post-quantum_cryptography)

## **Part VI**

# **Supplemental Material**

## SUPPLEMENTAL: ALGORITHMS

### 26.1 Non-master-theorem Recurrences

Recurrences that do not match the pattern required by the *master theorem* (page 14) can often be manipulated to do so using *substitutions*. We take a look at two examples here.

**Example 298** Consider the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

While we solved this recurrence above using the master theorem with log factors, we can also do so through substitution. We perform the substitution  $S(n) = T(n)/\log n$  to get the following:

$$\begin{aligned} S(n) \log n &= 2S\left(\frac{n}{2}\right) \log \frac{n}{2} + O(n \log n) \\ &= 2S\left(\frac{n}{2}\right)(\log n - \log 2) + O(n \log n) \\ &= 2S\left(\frac{n}{2}\right)(\log n - 1) + O(n \log n) \end{aligned}$$

To get this, we substituted  $T(n) = S(n) \log n$  and  $T(n/2) = S(n/2) \log(n/2)$  in the first step. Since  $\log n - 1 \leq \log n$ , we can turn the above into the inequality:

$$S(n) \log n \leq 2S\left(\frac{n}{2}\right) \log n + O(n \log n)$$

We can then divide out the  $\log n$  from each term to get:

$$S(n) \leq 2S\left(\frac{n}{2}\right) + O(n)$$

We now have something that is in the right form for applying the master theorem. Even though it is an inequality rather than an equality, because we are computing an upper bound, we can still apply the master theorem. We have  $k/b^d = 2/2^1 = 1$ , so we get:

$$S(n) = O(n \log n)$$

We can then undo the substitution  $S(n) = T(n)/\log n$  to get:

$$\begin{aligned} T(n) &= S(n) \log n = O(n \log n) \log n \\ &= O(n \log^2 n) \end{aligned}$$

**Exercise 299** In Example 298, we assumed that if  $f(n) = O(\log n)$ , then  $f(n)/\log n = O(1)$ . Prove from the definition of big-O that this holds.

---

CHAPTER  
**TWENTYSEVEN**

---

## SUPPLEMENTAL: COMPUTABILITY

### 27.1 Applying Rice's Theorem

*Rice's theorem* (page 130) gives us a third tool for proving the undecidability of a language, in addition to the direct proof we saw for  $L_{\text{ACC}}$  and Turing reduction from a known undecidable language. Given a language  $L$ , we need to do the following to apply Rice's theorem:

1. Define a semantic property  $\mathbb{P}$ , i.e. a set of languages.
2. Show that  $L_{\mathbb{P}} = L$ . In other words, we show that the language

$$L_{\mathbb{P}} = \{\langle M \rangle : M \text{ is a program and } L(M) \in \mathbb{P}\}$$

consists of the same set of elements as  $L$ .

3. Show that  $\mathbb{P}$  is nontrivial. This requires demonstrating that there is some recognizable language  $A \in \mathbb{P}$  and another recognizable language  $B \notin \mathbb{P}$ .

We can then conclude by Rice's theorem that  $L$  is undecidable.

**Example 300** Define  $L_{\Sigma^*}$  as follows:

$$L_{\Sigma^*} = \{\langle M \rangle : M \text{ is a program that accepts all inputs}\}$$

We use Rice's theorem to show that  $L_{\Sigma^*}$  is undecidable. Let  $\mathbb{P}$  be the semantic property:

$$\mathbb{P} = \{\Sigma^*\}$$

The property contains just a single language. Observe that  $M$  accepts all inputs exactly when  $L(M) = \Sigma^*$ . Thus, we can express  $L_{\Sigma^*}$  as:

$$L_{\Sigma^*} = \{\langle M \rangle : M \text{ is a program and } L(M) = \Sigma^*\}$$

This is the exact definition of  $L_{\mathbb{P}}$ , so we have  $L_{\Sigma^*} = L_{\mathbb{P}}$ .

We proceed to show that there is a recognizable language in  $\mathbb{P}$  and another not in  $\mathbb{P}$ .  $\Sigma^*$  is a recognizable language in  $\mathbb{P}$  – we can recognize  $\Sigma^*$  with a program that accepts all inputs.  $\emptyset$  is a recognizable language not in  $\mathbb{P}$  – we can recognize  $\emptyset$  with a program that rejects all inputs. Thus,  $\mathbb{P}$  is nontrivial.

Since  $\mathbb{P}$  is nontrivial, by Rice's theorem,  $L_{\mathbb{P}}$  is undecidable. Since  $L_{\mathbb{P}} = L_{\Sigma^*}$ ,  $L_{\Sigma^*}$  is undecidable.

**Example 301** Define  $L_{A376}$  as follows:

$$L_{A376} = \{\langle M \rangle : M \text{ is a program that accepts all strings of length less than 376}\}$$

We use Rice's theorem to show that  $L_{A376}$  is undecidable. Define  $S_{376}$  to be the set of all strings whose length is less than 376:

$$S_{376} = \{x \in \Sigma^* : |x| < 376\}$$

Then let  $\mathbb{P}$  be the semantic property:

$$\mathbb{P} = \{L \subseteq \Sigma^* : S_{376} \subseteq L\}$$

The property contains all languages that themselves contain all of  $S_{376}$ . If  $M$  accepts all inputs of length less than 376, then  $S_{376} \subseteq L(M)$ . We thus have:

$$\begin{aligned} L_{A376} &= \{\langle M \rangle : M \text{ is a program that accepts all strings of length less than 376}\} \\ &= \{\langle M \rangle : M \text{ is a program and } S_{376} \subseteq L(M)\} \\ &= \{\langle M \rangle : M \text{ is a program and } L(M) \in \mathbb{P}\} \\ &= L_{\mathbb{P}} \end{aligned}$$

We proceed to show that there is a recognizable language in  $\mathbb{P}$  and another not in  $\mathbb{P}$ .  $\Sigma^*$  is a recognizable language in  $\mathbb{P}$  – a program that recognizes  $\Sigma^*$  accepts all inputs, including all those of length less than 376.  $\emptyset$  is a recognizable language not in  $\mathbb{P}$  – a program that recognizes  $\emptyset$  does not accept any inputs, including those of length less than 376. Thus,  $\mathbb{P}$  is nontrivial.

Since  $\mathbb{P}$  is nontrivial, by Rice's theorem,  $L_{\mathbb{P}}$  is undecidable. Since  $L_{\mathbb{P}} = L_{A376}$ ,  $L_{A376}$  is undecidable.

**Example 302** Define  $L_{DEC}$  as follows:

$$L_{DEC} = \{\langle M \rangle : M \text{ is a program and } L(M) \text{ is decidable}\}$$

We use Rice's theorem to show that  $L_{DEC}$  is undecidable. Let  $\mathbb{P}$  be the semantic property:

$$\mathbb{P} = \{L \subseteq \Sigma^* : L \text{ is decidable}\}$$

The property contains all decidable languages, and  $L_{DEC} = L_{\mathbb{P}}$ .

We proceed to show that there is a recognizable language in  $\mathbb{P}$  and another not in  $\mathbb{P}$ .  $\Sigma^*$  is a recognizable language in  $\mathbb{P}$  – we can both decide and recognize  $\Sigma^*$  with a program that accepts all inputs.  $L_{ACC}$  is a recognizable language not in  $\mathbb{P}$  – we know that  $L_{ACC}$  is undecidable, and it is recognized by the universal Turing machine  $U$ . Thus,  $\mathbb{P}$  is nontrivial.

Since  $\mathbb{P}$  is nontrivial, by Rice's theorem,  $L_{\mathbb{P}}$  is undecidable. Since  $L_{\mathbb{P}} = L_{DEC}$ ,  $L_{DEC}$  is undecidable.

Rice's theorem is not applicable to all undecidable languages. Some examples of languages where Rice's theorem cannot be applied are:

- The language

$$L_{REJ} = \{(\langle M \rangle, x) : M \text{ is a program and } M \text{ rejects } x\}$$

does not consist of codes of programs on their own, so we cannot define a property  $\mathbb{P}$  such that  $L_{\mathbb{P}} = L_{REJ}$ . We must show undecidability either directly or through a Turing reduction.

- The language

$$L_{SmallTM} = \{\langle M \rangle : M \text{ has fewer than 100 states}\}$$

is concerned with the structure of  $M$  rather than its language  $L(M)$ . This is a *syntactic property* rather than a semantic property, and it is decidable by examining the code of  $M$ .

- The languages

$$L_{R376} = \{\langle M \rangle : M \text{ is a program that rejects all strings of length less than 376}\}$$

$$L_{L376} = \{\langle M \rangle : M \text{ is a program that loops on all strings of length less than 376}\}$$

do not have a direct match with a semantic property. In both cases,  $L(M) \cap S_{376} = \emptyset$ , but the two languages  $L_{R376}$  and  $L_{L376}$  are clearly distinct and non-overlapping. In fact, a program  $M_1$  that rejects all strings is in  $L_{R376}$  while a program  $M_2$  that loops on all strings is in  $L_{L376}$ , but both programs have the same language  $L(M_1) = L(M_2) = \emptyset$ . Thus, both  $L_{R376}$  and  $L_{L376}$  contain some programs and exclude others with the same language, so it is impossible to define a property  $\mathbb{P}$  such that  $L_{\mathbb{P}} = L_{R376}$  or  $L_{\mathbb{P}} = L_{L376}$ .

However, both  $L_{R376}$  and  $L_{L376}$  are actually undecidable. While we cannot apply Rice's theorem, we can show undecidability via a Turing reduction.

Thus, Rice's theorem allows us to take a shortcut for languages of the right format, but we still need the previous tools we saw for languages that do not follow the structure required by the theorem.

## 27.2 Computable Functions and Kolmogorov Complexity

Previously, we only considered decision problems, for which the answer is either yes or no. We formalized such a problem as a language  $L$  over an alphabet  $\Sigma$ , where  $L$  is the set of all yes instances. Then solving a decision problem means being able to determine whether  $x \in L$  for all inputs  $x \in \Sigma^*$ .

We can define a function  $f_L : \Sigma^* \rightarrow \{0, 1\}$  corresponding to the language  $L$ :

$$f_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

A program that decides  $L$  also *computes* the function  $f_L$ , meaning that it determines the value  $f_L(x)$  for any input  $x$ . For an arbitrary function  $f : \Sigma^* \rightarrow \Sigma^*$ , we say that  $f$  is *computable* if there exists a program that outputs  $f(x)$  given the input  $x$ .

Computability is a generalization of decidability for arbitrary functions. Decidability only applies to functions whose codomain is  $\{0, 1\}$ , while computability applies to any function whose codomain is comprised of finite-length strings.

In the Turing-machine model, we represent a result in  $\{0, 1\}$  with specific reject and accept states. For a function whose codomain is  $\Sigma^*$ , we cannot define a new state for each element in  $\Sigma^*$  – there are infinitely many elements in  $\Sigma^*$ , but the set of states  $Q$  for a Turing machine must be finite. Instead, we consider a Turing machine to output the string  $f(x)$  if it reaches a final state with  $f(x)$  written on its tape.

For a concrete programming model, we rely on whatever convention is used in that model for output, such as returning a value from a function or writing something to a standard output stream. In pseudocode, we typically just state “output  $w$ ” or “return  $w$ ” as an abstraction of outputting the value  $w$ .

### Equivalence of Functional and Decision Models

We can actually turn a functional problem into a decision one by recasting it as a series of yes/no questions. There are many ways to do so, and the following is one concrete way.

Let  $f : \Sigma^* \rightarrow \Sigma^*$  be a computable function. We can define a language corresponding to  $f$  as follows:

$$L_f = \{\langle x, i, \sigma \rangle \in \Sigma^* \times \mathbb{N} \times \Sigma : |f(x)| \geq i \text{ and the } i\text{th symbol of } f(x) \text{ is } \sigma\}.$$

In other words,  $\langle x, i, \sigma \rangle \in L_f$  when  $f(x)$  has at least  $i$  symbols, and the  $i$ th symbol is  $\sigma$ .

Given a decider  $D$  for  $L_f$ , we can construct a program  $C$  that computes  $f$  as follows:

```

function  $C(x)$ 
  for  $i = 1, 2, \dots$  do
    for all  $\sigma \in \Sigma$  do
      if  $D(\langle x, i, \sigma \rangle)$  accepts then
        print  $\sigma$ 
    if no symbol was written in the last loop then halt

```

This machine queries  $D$  for each possible symbol at position  $i$  of  $f(x)$ , printing the correct symbols one by one. It halts when it reaches an  $i$  for which there is no symbol.

To complete our demonstration of equivalence, we show how to construct  $D$  given a program  $C$  that computes  $f$ :

```

function  $D(x, i, \sigma)$ 
  if  $|C(x)| \geq i$  and the  $i$ th symbol of  $C(x)$  is  $\sigma$  then accept
  reject

```

This equivalence shows that we do not lose any power by restricting ourselves to decision problems, as we have done until this point.

Given any alphabet  $\Sigma$ , there are uncountably many functions  $f : \Sigma^* \rightarrow \Sigma^*$ . Since a Turing machine  $M$  can compute at most one function and the set of Turing machines is countable, there are uncountably many uncomputable functions over any alphabet  $\Sigma$ .

**Exercise 303** Use diagonalization to show that the set of functions  $f : \Sigma^* \rightarrow \Sigma^*$  is uncountable when  $\Sigma$  is the unary alphabet  $\Sigma = \{1\}$ .

As a specific example of an uncomputable function, consider the problem of data compression. Given a string  $w$ , we would like to encode its data as a shorter string  $w_c$  that still allows us to recover the original string  $w$ . This is known as *lossless compression*, and many algorithms for lossless compression are in use including the GIF format for images, ZIP and GZIP for arbitrary files, and so on. (There are *lossy* formats as well such as JPEG and MP3, where some information is lost as part of the compression process.) It can be demonstrated by a counting argument that any lossless compression algorithm has *uncompressible* strings for all lengths  $n \geq 1$ , where the result of compressing the string is no shorter than the original string.

**Exercise 304** Let  $C$  be an arbitrary compression algorithm for binary strings. Define  $C(w)$  to be the result of compressing the string  $w$ , where  $w \in \{0, 1\}^*$  and  $C(w) \in \{0, 1\}^*$ .

- Prove that for all  $n \geq 1$ , there are at most  $2^n - 1$  binary strings  $w$  such that the corresponding compressed string  $C(w)$  has length strictly less than  $n$ .
- Conclude that for all  $n \geq 1$ , there exists some uncompressible string  $w$  of length  $n$  such that  $C(w)$  has length at least  $n$ .

We specifically consider “compression by program,” where we compress a string  $w$  by writing a program that, given an empty input, outputs the string  $w$ . Assume we are working in a concrete programming language  $U$ . We define the *Kolmogorov complexity* of  $w$  in language  $U$  as the length of the shortest program in  $U$  that outputs  $w$ , given an empty string. We denote this by  $K_U(w)$ , and we have:

$$K_U(w) = \min\{|\langle M \rangle| : M \text{ is a program in language } U \text{ that outputs } w, \text{ given an empty input}\}$$

Observe that  $K_U(w) = O(|w|)$  for any  $w \in \{0, 1\}^*$  – we can just hardcode the string  $w$  in the program itself. For instance, the following is a C++ program that outputs  $w = 0101100101$ :

```
#include <iostream>

int main() {
    std::cout << "0101100101";
}
```

To output a different value  $w'$ , we need only swap out the hardcoded string  $w$  for  $w'$ . The total length of the program that has a string hardcoded is a constant number of symbols, plus the length of the desired output string itself. Thus, there exists a program of length  $O(|w|)$  that outputs  $w$ , so the shortest program that does so has length no more than  $O(|w|)$ , and  $K_{C++}(w) = O(|w|)$ . The same is true for any other programming language  $U$ .

For some strings  $w$ , we can define a much shorter program that outputs  $w$ . Consider  $w = 0^m$  for some large number  $m$ , meaning that  $w$  consists of  $m$  zeros. We can define a short program to output  $w$  as follows:

```
#include <iostream>

int main() {
    for (int i = 0; i < m; ++i)
        std::cout << "0";
}
```

As a concrete example, let  $m = 1000$ . The program that hardcodes  $w = 0^m$  would have a length on the order of 1000. But the program that follows the looping pattern above is:

```
#include <iostream>

int main() {
    for (int i = 0; i < 1000; ++i)
        std::cout << "0";
}
```

Here, we hardcode  $m$  rather than  $0^m$ . The former has length  $O(\log m)$  (for  $m = 1000$ , four digits in decimal or ten in binary), as opposed to the latter that has length  $m$ . Thus, we achieve a significant compression for  $w = 0^m$ , and we have  $K_{C++}(0^m) = O(\log m)$ .

We now demonstrate that the function  $K_U(w)$  is uncomputable for any programming language  $U$ . Assume for the purposes of contradiction that  $K_U(w)$  is computable. For a given length  $n$ , we define the program  $Q_n$  as follows:

```
function  $Q_n$ 
    for all  $x \in \{0, 1\}^n$  do
        if  $K_U(x) \geq n$  then return  $x$ 
```

As mentioned previously, any compression algorithm has uncompressible strings for every length  $n \geq 1$ . Thus,  $Q_n$  will find a string  $w_n$  such that  $K_U(w_n) \geq n$ , output it, and halt.

Since  $Q_n$  outputs  $w_n$  given an empty input, by definition, we have  $K_U(w_n) \leq |Q_n|$ ; that is, the Kolmogorov complexity of  $w_n$  in language  $U$  is no more than the length of  $Q_n$ , since  $Q_n$  itself is a program that outputs  $w_n$ . How long is  $Q_n$ ? The only part of  $Q_n$  that depends on  $n$  is the hardcoded value  $n$  in the loop that iterates over  $\{0, 1\}^n$ . As we saw before, it takes only  $O(\log n)$  symbols to encode a value  $n$ . Thus, we have  $|Q_n| = O(\log n)$ .

We have arrived at a contradiction: we have demonstrated that  $K_U(w_n) \geq n$  and also that  $K_U(w_n) \leq |Q_n| = O(\log n)$ . These two conditions cannot be simultaneously satisfied. Since we have reached a contradiction, our original assumption that  $K_U(w)$  is computable must be false, and  $K_U(w)$  is an uncomputable function.

---

CHAPTER  
TWENTYEIGHT

---

## SUPPLEMENTAL: RANDOMNESS

### 28.1 Primality Testing

A key component of many *cryptography* (page 242) algorithms is finding large prime numbers, with hundreds or even thousands of digits. A typical approach is to randomly choose a large number and then apply a *primality test* to determine whether it is prime – the *prime number theorem*<sup>99</sup> states that approximately  $1/m$  of the  $m$ -bit numbers are prime, so we only expect to check  $m$  numbers before finding one that is prime<sup>100</sup>. As long as the primality test itself is efficient with respect to  $m$ , the expected time to find a prime number with  $m$  bits is polynomial in  $m$ .

Formally, we wish to decide the following language:

$$\text{PRIMES} = \{m \in \mathbb{N} : m \text{ is prime}\} .$$

A positive integer  $m \geq 2$  is prime if it has no positive divisors other than 1 and itself. (For technical reasons, the number 1 is not considered prime, although its only positive divisors are 1 and itself, which are the same.) Equivalently, it has no divisor between 2 and  $m - 1$ , inclusive.

Is the language PRIMES in P? Let us consider the following simple algorithm to decide the language:

**Input:** a positive integer  $m \geq 2$   
**Output:** whether  $m$  is prime  
**function** IsPRIME( $m$ )  
    **for**  $i = 2$  to  $m - 1$  **do**  
        **if**  $m$  is divisible by  $i$  **then reject**  
    **accept**

This algorithm does  $O(m)$  trial divisions. Is this efficient? The size of the input  $m$  is the number of bits required to represent  $m$ , which is  $O(\log m)$ . Thus, the number of operations is  $O(m) = O(2^{\log m})$ , which is exponential in the size of  $m$ .

We can improve the algorithm by observing that if  $m$  has a factor within the interval  $[2, m - 1]$ , it must have a factor between 2 and  $\lfloor \sqrt{m} \rfloor$  (inclusive) – because the product of two or more larger numbers is larger than  $m$ . Thus, we need only iterate up to  $\lfloor \sqrt{m} \rfloor$ :

**function** IsPRIME( $m$ )  
    **for**  $i = 2$  to  $\lfloor \sqrt{m} \rfloor$  **do**  
        **if**  $m$  is divisible by  $i$  **then reject**  
    **accept**

<sup>99</sup> [https://en.wikipedia.org/wiki/Prime\\_number\\_theorem](https://en.wikipedia.org/wiki/Prime_number_theorem)

<sup>100</sup> This follows from the fact that the expected value of a geometric distribution<sup>Page 270, 101</sup> with parameter  $p$  is  $1/p$ .

<sup>101</sup> [https://en.wikipedia.org/wiki/Geometric\\_distribution](https://en.wikipedia.org/wiki/Geometric_distribution)

This algorithm does  $O(\sqrt{m})$  trial divisions. However, this still is not polynomial; we have:

$$O(\sqrt{m}) = O(m^{1/2}) = O(2^{(\log m)/2}).$$

To be efficient, a primality-testing algorithm must have runtime  $O(\log^k m)$  for some constant  $k$ . Neither of the algorithms above meet this threshold.

In fact, there is a known efficient algorithm for primality testing, the [AKS primality test](#)<sup>102</sup>. Thus, it is indeed the case that PRIMES  $\in \text{P}$ . However, this algorithm is somewhat complicated, and its running time is high enough to preclude it from being used in practice. Instead, we consider a randomized primality test that is efficient and works well in practice for most inputs. The algorithm we construct relies on the *extended Fermat's little theorem*.

**Theorem 305 (Extended Fermat's Little Theorem)** *Let  $n \in \mathbb{N}$  be a natural number such that  $n \geq 2$ . Let  $a$  be a witness in the range  $1 \leq a \leq n - 1$ . Then:*

- *If  $n$  is prime, then  $a^{n-1} \equiv 1 \pmod{n}$  for any witness  $a$ .*
- *If  $n$  is composite and  $n$  is not a Carmichael number, then  $a^{n-1} \equiv 1 \pmod{n}$  for at most half the witnesses  $1 \leq a \leq n - 1$ .*

We postpone discussion of Carmichael numbers for the moment. Instead, we take a look at some small cases of composite numbers to see that the extended Fermat's little theorem holds. We first consider  $n = 6$ . We have:

$$\begin{aligned} a = 1 : \quad 1^5 &\equiv 1 \pmod{6} \\ a = 2 : \quad 2^5 = 32 = 2 + 5 \cdot 6 &\equiv 2 \pmod{6} \\ a = 3 : \quad 3^5 = 243 = 3 + 40 \cdot 6 &\equiv 3 \pmod{6} \\ a = 4 : \quad 4^5 = 1024 = 4 + 170 \cdot 6 &\equiv 4 \pmod{6} \\ a = 5 : \quad 5^5 = 3125 = 5 + 520 \cdot 6 &\equiv 5 \pmod{6} \end{aligned}$$

We see that  $a^{n-1} \equiv 1 \pmod{n}$  for only the single witness  $a = 1$ . Similarly, we consider  $n = 9$ :

$$\begin{aligned} a = 1 : \quad 1^8 &\equiv 1 \pmod{9} \\ a = 2 : \quad 2^8 = 256 = 4 + 28 \cdot 9 &\equiv 4 \pmod{9} \\ a = 3 : \quad 3^8 = 6561 = 0 + 729 \cdot 9 &\equiv 0 \pmod{9} \\ a = 4 : \quad 4^8 = 65536 = 7 + 7281 \cdot 9 &\equiv 7 \pmod{9} \\ a = 5 : \quad 5^8 = 390625 = 7 + 43402 \cdot 9 &\equiv 7 \pmod{9} \\ a = 6 : \quad 6^8 = 1679616 = 0 + 186624 \cdot 9 &\equiv 0 \pmod{9} \\ a = 7 : \quad 7^8 = 5764801 = 4 + 640533 \cdot 9 &\equiv 4 \pmod{9} \\ a = 8 : \quad 8^8 = 16777216 = 1 + 1864135 \cdot 9 &\equiv 1 \pmod{9} \end{aligned}$$

Here, there are two witnesses  $a$  where  $a^{n-1} \equiv 1 \pmod{n}$ , out of eight total. Finally, we consider  $n = 7$ :

$$\begin{aligned} a = 1 : \quad 1^6 &\equiv 1 \pmod{7} \\ a = 2 : \quad 2^6 = 64 = 1 + 9 \cdot 7 &\equiv 1 \pmod{7} \\ a = 3 : \quad 3^6 = 729 = 1 + 104 \cdot 7 &\equiv 1 \pmod{7} \\ a = 4 : \quad 4^6 = 4096 = 1 + 585 \cdot 7 &\equiv 1 \pmod{7} \\ a = 5 : \quad 5^6 = 15625 = 1 + 2232 \cdot 7 &\equiv 1 \pmod{7} \\ a = 6 : \quad 6^6 = 46656 = 1 + 6665 \cdot 7 &\equiv 1 \pmod{7} \end{aligned}$$

Since 7 is prime, we have  $a^{n-1} \equiv 1 \pmod{n}$  for all witnesses  $a$ .

The extended Fermat's little theorem leads directly to a simple, efficient randomized algorithm for primality testing. This *Fermat primality test* is as follows:

<sup>102</sup> [https://en.wikipedia.org/wiki/AKS\\_primality\\_test](https://en.wikipedia.org/wiki/AKS_primality_test)

```
function FERMATTEST( $m$ )
    choose uniformly random  $a$  such that  $1 \leq a \leq m - 1$ 
    if  $a^{m-1} \equiv 1 \pmod{m}$  then accept
    reject
```

The modular exponentiation in this algorithm can be done with  $O(\log m)$  multiplications using a *divide and conquer* (page 13) strategy, and each multiplication can be done efficiently. Thus, this algorithm is polynomial with respect to the size of  $m$ .

As for correctness, we have:

- If  $m$  is prime, then the algorithm always accepts  $m$ . In other words:

$$\Pr[\text{the Fermat test accepts } m] = 1$$

- If  $m$  is composite and not a Carmichael number, then the algorithm rejects  $m$  with probability at least  $\frac{1}{2}$ . In other words:

$$\Pr[\text{the Fermat test rejects } m] \geq \frac{1}{2}$$

Thus, if the algorithm accepts  $m$ , we can be fairly confident that  $m$  is prime. And as we will see shortly, we can repeat the algorithm to obtain higher confidence that we get the right answer.

We now return to the problem of Carmichael numbers. A *Carmichael number* is a composite number  $n$  such that  $a^{n-1} \equiv 1 \pmod{n}$  for all witnesses  $a$  that are *relatively prime* to  $n$ , i.e.  $\gcd(a, n) = 1$ . This implies that for a Carmichael number, the Fermat test reports with high probability that the number is prime, despite it being composite. We call a number that passes the Fermat test with high probability a *pseudoprime*, and the Fermat test is technically a randomized algorithm for deciding the following language:

$$\text{PSEUDOPRIMES} = \left\{ m \in \mathbb{N} : \begin{array}{l} a^{m-1} \pmod{m} \equiv 1 \text{ for at least half} \\ \text{the witnesses } 1 \leq a \leq m-1 \end{array} \right\}$$

Carmichael numbers are much rarer than prime numbers, so for many applications, the Fermat test is sufficient to determine with high confidence that a randomly chosen number is prime. On the other hand, if the number is chosen by an adversary, then the Fermat test is unsuitable, and a more complex randomized algorithm such as the Miller-Rabin primality test must be used instead.

### 28.1.1 The Miller-Rabin Test

The Miller-Rabin test is designed around the fact that for prime  $m$ , the only square roots of 1 modulo  $m$  are -1 and 1. More specifically, if  $x$  is a square root of 1 modulo  $m$ , we have

$$x^2 \equiv 1 \pmod{m}$$

By *definition of modular arithmetic* (page 243), this means that

$$x^2 - 1 = km$$

for some integer  $k$  (i.e.  $m$  evenly divides the difference of  $x^2$  and 1). We can factor  $x^2 - 1$  to obtain:

$$(x+1)(x-1) = km$$

When  $m$  is composite, so that  $m = pq$  for integers  $p, q > 1$ , then it is possible for  $p$  to divide  $x+1$  and  $q$  to divide  $x-1$ , in which case  $pq = m$  divides their product  $(x+1)(x-1)$ . However, when  $m$  is prime, the only way for the

equation to hold is if  $m$  divides either  $x + 1$  or  $x - 1$ ; otherwise, the prime factorization of  $(x + 1)(x - 1)$  does not contain  $m$ , and by the fundamental theorem of arithmetic<sup>103</sup>, it cannot be equal to a number whose prime factorization contains  $m$ <sup>104</sup>. Thus, we have either  $x + 1 = am$  for some integer  $a$ , or  $x - 1 = bm$  for some  $b \in \mathbb{Z}$ . The former gives us  $x \equiv -1 \pmod{m}$ , while the latter results in  $x \equiv 1 \pmod{m}$ . This, if  $m$  is prime and  $x^2 \equiv 1 \pmod{m}$ , then either  $x \equiv 1 \pmod{m}$  or  $x \equiv -1 \pmod{m}$ .

The Miller-Rabin test starts with the Fermat test: choose a witness  $1 \leq a \leq m - 1$  and check whether  $a^{m-1} \equiv 1 \pmod{m}$ . If this does not hold, then  $m$  fails the Fermat test and therefore is not prime. If it does indeed hold, then the test checks the square root  $a^{\frac{1}{2}(m-1)}$  of  $a^{m-1}$  to see if it is 1 or -1. If it is 1, the test checks the square root  $a^{\frac{1}{4}(m-1)}$  of  $a^{\frac{1}{2}(m-1)}$  to see if it is 1 or -1, and so on. The termination conditions are as follows:

- The test finds a square root of 1 that is not -1 or 1. By the reasoning above,  $m$  must be composite.
- The test finds a square root of 1 that is -1. The reasoning above only holds for square roots of 1, so the test cannot continue by computing the square root of -1. In this case,  $m$  may be prime or composite.
- The test reaches some  $r$  for which  $1/2^r \cdot (m - 1)$  is no longer an integer. Then it cannot compute  $a$  to this power modulo  $m$ . In this case,  $m$  may be prime or composite.

To compute these square roots, we first extract powers of 2 from the number  $m - 1$  (which is even for odd  $m$ , the cases of interest):

```
function EXTRACTPOWERSOF2(x)
  if x is odd then return (x, 0)
  (d, s') = EXTRACTPOWERSOF2(x/2) return (d, s' + 1)
```

So, EXTRACTPOWERSOF2( $m - 1$ ) computes  $d$  and  $s$  such that

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

where  $\gcd(d, 2) = 1$ . Then we have  $a^{2^{s-1}d}$  is the square root of  $a^{2^s d}$ , since

$$(a^{2^{s-1}d})^2 = a^{2 \cdot 2^{s-1}d} = a^{2^s d}$$

The full Miller-Rabin algorithm is as follows:

```
function MILLERRABIN(m)
  choose uniformly random  $1 \leq a \leq m - 1$ 
  compute  $s, d$  such that  $m - 1 = 2^s d$  and  $d$  is odd
  compute  $a^d, a^{2d}, a^{4d}, \dots, a^{2^{s-1}d}, a^{2^s d}$ 
  if  $a^{2^s d} \not\equiv 1 \pmod{m}$  then reject
  for  $t = s - 1$  down to 0 do
    if  $a^{2^t d} \equiv -1 \pmod{m}$  then accept
    else if  $a^{2^t d} \not\equiv 1 \pmod{m}$  then reject
  accept
```

This algorithm is efficient:  $a^d$  can be computed using  $O(\log d) = O(\log m)$  multiplications, and it takes another  $O(\log m)$  multiplications to compute  $a^{2d}, a^{4d}, \dots, a^{2^s d}$  since  $s = O(\log m)$ . Each multiplication is efficient as it is done modulo  $m$ , so the entire algorithm takes polynomial time in the size of  $m$ .

As for correctness, we have:

- If  $m$  is prime, then the algorithm accepts  $m$  with probability 1.

<sup>103</sup> [https://en.wikipedia.org/wiki/Fundamental\\_theorem\\_of\\_arithmetic](https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic)

<sup>104</sup> Euclid's lemma<sup>Page 273, 105</sup> more directly states that if  $m$  is prime and  $m$  divides a product  $ab$ , then  $m$  must divide either  $a$  or  $b$ .

<sup>105</sup> [https://en.wikipedia.org/wiki/Euclid%27s\\_lemma](https://en.wikipedia.org/wiki/Euclid%27s_lemma)

- If  $m$  is composite, then the algorithm rejects  $m$  with probability at least  $\frac{3}{4}$ .

The latter is due to the fact that when  $m$  is composite,  $m$  passes the Miller-Rabin test for at most  $\frac{1}{4}$  of the witnesses  $1 \leq a \leq m - 1$ . Unlike the Fermat test, there are no exceptions to this, making the Miller-Rabin test a better choice in many applications.

**Exercise 306** *Polynomial identity testing* is the problem of determining whether two polynomials  $p(x)$  and  $q(x)$  are the same, or equivalently, whether  $p(x) - q(x)$  is the zero polynomial.

- Let  $d$  be an upper bound on the degree of  $p(x)$  and  $q(x)$ . Show that for a randomly chosen integer  $a \in [1, 4d]$ , if  $p(x) \neq q(x)$ , then  $\Pr[p(a) \neq q(a)] \geq \frac{3}{4}$ .

**Hint:** A nonzero polynomial  $r(x)$  of degree at most  $d$  can have at most  $d$  roots  $s_i$  such that  $r(s_i) = 0$ .

- Devise an efficient, randomized algorithm  $A$  to determine whether  $p(x)$  and  $q(x)$  are the same, with the behavior that:

- if  $p(x) = q(x)$ , then

$$\Pr[A \text{ accepts } p(x), q(x)] = 1$$

- if  $p(x) \neq q(x)$ , then

$$\Pr[A \text{ rejects } p(x), q(x)] \geq \frac{3}{4}$$

Assume that a polynomial can be efficiently evaluated on a single input.

## 28.2 Multiplicative Chernoff Bounds

Like Markov's inequality, *Chernoff bounds* are a form of concentration bounds that allow us to reason about the deviation of a random variable from its expectations. There are multiple variants of Chernoff bounds; we restrict ourselves to the following “multiplicative” versions.

Let  $X = X_1 + \dots + X_n$  be the sum of independent indicator random variables, where the indicator  $X_i$  has the expectation

$$\mathbb{E}[X_i] = \Pr[X_i = 1] = p_i$$

Let  $\mu$  be the expected value of  $X$ :

$$\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = \sum_i p_i$$

Here, we've applied linearity of expectation to relate the expectation of  $X$  to that of the indicators  $X_i$ . The Chernoff bounds then are as follows.

**Theorem 307 (Multiplicative Chernoff Bound – Upper Tail)** *Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent indicator random variables with  $\mathbb{E}[X_i] = p_i$ , and  $\mu = \sum_i p_i$ . Suppose we wish to bound the probability of  $X$  exceeding its expectation  $\mu$  by at least a factor of  $1 + \delta$ , where  $\delta > 0$ . Then*

$$\Pr[X \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu$$

**Theorem 308 (Multiplicative Chernoff Bound – Lower Tail)** Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent indicator random variables with  $\mathbb{E}[X_i] = p_i$ , and  $\mu = \sum_i p_i$ . Suppose we wish to bound the probability of  $X$  being below its expectation  $\mu$  by at least a factor of  $1 - \delta$ , where  $0 < \delta < 1$ . Then

$$\Pr[X \leq (1 - \delta)\mu] \leq \left(\frac{e^{-\delta}}{(1 - \delta)^{1-\delta}}\right)^\mu$$

These inequalities can be unwieldy to work with, so we often use the following simpler, looser bounds.

**Theorem 309 (Multiplicative Chernoff Bound – Simplified Upper Tail)** Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent indicator random variables with  $\mathbb{E}[X_i] = p_i$ , and  $\mu = \sum_i p_i$ . Suppose we wish to bound the probability of  $X$  exceeding its expectation  $\mu$  by at least a factor of  $1 + \delta$ , where  $\delta > 0$ . Then

$$\Pr[X \geq (1 + \delta)\mu] \leq e^{-\frac{\delta^2 \mu}{2+\delta}}$$

**Theorem 310 (Multiplicative Chernoff Bound – Simplified Lower Tail)** Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent indicator random variables with  $\mathbb{E}[X_i] = p_i$ , and  $\mu = \sum_i p_i$ . Suppose we wish to bound the probability of  $X$  being below its expectation  $\mu$  by at least a factor of  $1 - \delta$ , where  $0 < \delta < 1$ . Then

$$\Pr[X \leq (1 - \delta)\mu] \leq e^{-\frac{\delta^2 \mu}{2}}$$

Before we proceed to make use of these bounds, we first prove that the unsimplified upper-tail and lower-tail bounds hold<sup>106</sup>.

**Proof 311 (Proof of Upper-tail Chernoff Bound)** To demonstrate the upper-tail Chernoff bound, we make use of the *Chernoff bounding technique* – rather than reasoning about the random variable  $X$  directly, we instead reason about  $e^{tX}$ , since small deviations in  $X$  turn into large deviations in  $e^{tX}$ . We have that  $X \geq (1 + \delta)\mu$  exactly when  $e^{tX} \geq e^{t(1+\delta)\mu}$  for any  $t \geq 0$ ; we obtain this by raising  $e^t \geq 1$  to the power of both sides of the former inequality. Then

$$\begin{aligned} \Pr[X \geq (1 + \delta)\mu] &= \Pr[e^{tX} \geq e^{t(1+\delta)\mu}] \\ &\leq \frac{\mathbb{E}[e^{tX}]}{e^{t(1+\delta)\mu}} \end{aligned}$$

where the latter step follows from Markov's inequality. Continuing, we have

$$\begin{aligned} \frac{\mathbb{E}[e^{tX}]}{e^{t(1+\delta)\mu}} &= e^{-t(1+\delta)\mu} \cdot \mathbb{E}[e^{tX}] \\ &= e^{-t(1+\delta)\mu} \cdot \mathbb{E}[e^{t(X_1 + \dots + X_n)}] \\ &= e^{-t(1+\delta)\mu} \cdot \mathbb{E}\left[\prod_i e^{tX_i}\right] \end{aligned}$$

We now make use of the fact that the  $X_i$  (and therefore the  $e^{tX_i}$ ) are independent. For **independent** random variables  $Y$  and  $Z$ , we have  $\mathbb{E}[YZ] = \mathbb{E}[Y] \cdot \mathbb{E}[Z]$  (see the [appendix](#) (page 231) for a proof). Thus,

$$e^{-t(1+\delta)\mu} \cdot \mathbb{E}\left[\prod_i e^{tX_i}\right] = e^{-t(1+\delta)\mu} \prod_i \mathbb{E}[e^{tX_i}]$$

The  $X_i$  are indicators, with  $\Pr[X_i = 1] = p_i$  and  $\Pr[X_i = 0] = 1 - p_i$ . When  $X_i = 1$ ,  $e^{tX_i} = e^t$ , and when

<sup>106</sup> Refer to the [appendix](#) (page 300) for proofs of the simplified bounds.

$X_i = 0$ ,  $e^{tX_i} = e^0 = 1$ . Thus, the distribution of  $e^{tX_i}$  is

$$\begin{aligned}\Pr[e^{tX_i} = e^t] &= p_i \\ \Pr[e^{tX_i} = 1] &= 1 - p_i\end{aligned}$$

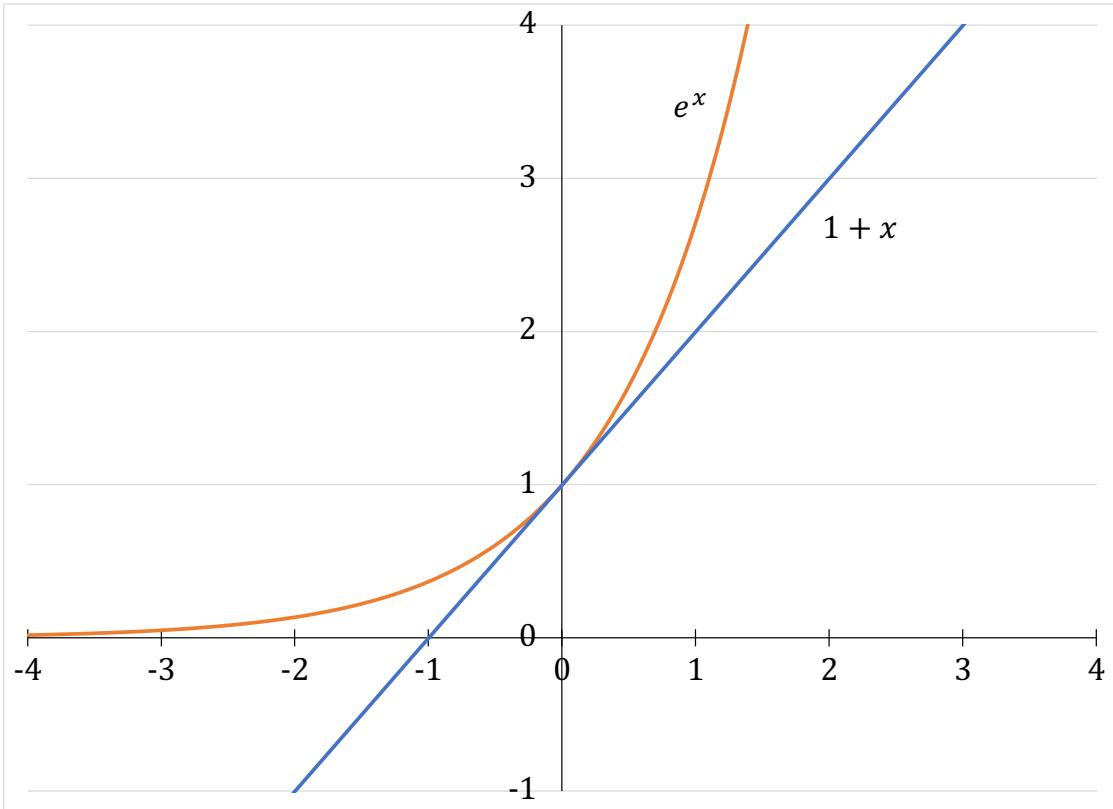
We can then compute the expectation of  $e^{tX_i}$ :

$$\begin{aligned}\mathbb{E}[e^{tX_i}] &= e^t \cdot \Pr[e^{tX_i} = e^t] + 1 \cdot \Pr[e^{tX_i} = 1] \\ &= e^t p_i + 1 - p_i \\ &= 1 + p_i(e^t - 1)\end{aligned}$$

Plugging this into the upper bound that resulted from Markov's inequality, we get

$$e^{-t(1+\delta)\mu} \prod_i \mathbb{E}[e^{tX_i}] = e^{-t(1+\delta)\mu} \prod_i 1 + p_i(e^t - 1)$$

We can further simplify this expression by observing that  $1 + x \leq e^x$  for all  $x$ :



Using  $x = p_i(e^t - 1)$ , we have

$$1 + p_i(e^t - 1) \leq e^{p_i(e^t - 1)}$$

Applying this to our previously computed upper bound, we get

$$\begin{aligned}
 e^{-t(1+\delta)\mu} \prod_i 1 + p_i(e^t - 1) &\leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t - 1)} \\
 &= e^{-t(1+\delta)\mu} e^{(e^t - 1) \sum_i p_i} \\
 &= e^{-t(1+\delta)\mu} e^{(e^t - 1)\mu} \\
 &= e^{\mu(e^t - 1 - t(1+\delta))}
 \end{aligned}$$

We can now choose  $t$  to minimize the exponent, which in turn minimizes the value of the exponential itself. Let  $f(t) = e^t - 1 - t(1+\delta)$ . Then we compute the derivative with respect to  $t$  and set that to zero to find an extremum:

$$\begin{aligned}
 f(t) &= e^t - 1 - t(1 + \delta) \\
 f'(t) &= e^t - (1 + \delta) = 0 \\
 e^t &= 1 + \delta \\
 t &= \ln(1 + \delta)
 \end{aligned}$$

Computing the second derivative  $f''(\ln(1+\delta)) = e^{\ln(1+\delta)} = 1+\delta$ , we see that it is positive since  $\delta > 0$ , therefore  $t = \ln(1 + \delta)$  is a minimum. Substituting it into our earlier expression, we obtain

$$\begin{aligned}
 e^{\mu(e^t - 1 - t(1+\delta))} &\leq e^{\mu(e^{\ln(1+\delta)} - 1 - (1+\delta)\ln(1+\delta))} \\
 &= e^{\mu((1+\delta) - 1 - (1+\delta)\ln(1+\delta))} \\
 &= e^{\mu(\delta - (1+\delta)\ln(1+\delta))} \\
 &= (e^{\delta - (1+\delta)\ln(1+\delta)})^\mu \\
 &= \left(\frac{e^\delta}{e^{(1+\delta)\ln(1+\delta)}}\right)^\mu \\
 &= \left(\frac{e^\delta}{(e^{\ln(1+\delta)})^{1+\delta}}\right)^\mu \\
 &= \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu
 \end{aligned}$$

**Proof 312 (Proof of Lower-tail Chernoff Bound)** The proof of the lower-tail Chernoff bound follows similar reasoning as that of the upper-tail bound. We have:

$$\begin{aligned}
 \Pr[X \leq (1 - \delta)\mu] &= \Pr[-X \geq -(1 - \delta)\mu] \\
 &= \Pr[e^{-tX} \geq e^{-t(1-\delta)\mu}] \\
 &\leq \frac{\mathbb{E}[e^{-tX}]}{e^{-t(1-\delta)\mu}}
 \end{aligned}$$

Here, we can apply Markov's inequality to the random variable  $e^{-tX}$  since it is nonnegative, unlike  $-X$ . Contin-

uing, we have

$$\begin{aligned}
 \frac{\mathbb{E}[e^{-tX}]}{e^{-t(1-\delta)\mu}} &= e^{t(1-\delta)\mu} \cdot \mathbb{E}[e^{-tX}] \\
 &= e^{t(1-\delta)\mu} \cdot \mathbb{E}[e^{-t(X_1+\dots+X_n)}] \\
 &= e^{t(1-\delta)\mu} \cdot \mathbb{E}\left[\prod_i e^{-tX_i}\right] \\
 &= e^{t(1-\delta)\mu} \prod_i \mathbb{E}[e^{-tX_i}]
 \end{aligned}$$

In the last step, we make use of the fact that the  $X_i$  and therefore the  $e^{-tX_i}$  are independent. The distribution of  $e^{-tX_i}$  is

$$\begin{aligned}
 \Pr[e^{-tX_i} = e^{-t}] &= p_i \\
 \Pr[e^{-tX_i} = 1] &= 1 - p_i
 \end{aligned}$$

Then the expectation of  $e^{-tX_i}$  is:

$$\begin{aligned}
 \mathbb{E}[e^{-tX_i}] &= e^{-t} \cdot \Pr[e^{-tX_i} = e^{-t}] + 1 \cdot \Pr[e^{-tX_i} = 1] \\
 &= e^{-t} p_i + 1 - p_i \\
 &= 1 + p_i(e^{-t} - 1)
 \end{aligned}$$

Plugging this into the bound we've computed so far, we get

$$e^{t(1-\delta)\mu} \prod_i \mathbb{E}[e^{-tX_i}] = e^{t(1-\delta)\mu} \prod_i 1 + p_i(e^{-t} - 1)$$

As with the upper-tail, we use the fact that  $1 + x \leq e^x$  for all  $x$  to simplify this, with  $x = p_i(e^{-t} - 1)$ :

$$\begin{aligned}
 e^{t(1-\delta)\mu} \prod_i 1 + p_i(e^{-t} - 1) &\leq e^{t(1-\delta)\mu} \prod_i e^{p_i(e^{-t}-1)} \\
 &= e^{t(1-\delta)\mu} e^{(e^{-t}-1) \sum_i p_i} \\
 &= e^{t(1-\delta)\mu} e^{(e^{-t}-1)\mu} \\
 &= e^{\mu(e^{-t}-1+t(1-\delta))}
 \end{aligned}$$

We choose  $t$  to minimize the exponent. Let  $f(t) = e^{-t} - 1 + t(1 - \delta)$ . Then we compute the derivative with respect to  $t$  and set that to zero to find an extremum:

$$\begin{aligned}
 f(t) &= e^{-t} - 1 + t(1 - \delta) \\
 f'(t) &= -e^{-t} + (1 - \delta) = 0 \\
 e^{-t} &= 1 - \delta \\
 t &= -\ln(1 - \delta)
 \end{aligned}$$

Computing the second derivative  $f''(-\ln(1 - \delta)) = e^{(-\ln(1 - \delta))} = 1 - \delta$ , we see that it is positive since

$0 < \delta < 1$ , therefore  $t = \ln(1 - \delta)$  is a minimum. Substituting it into our earlier expression, we obtain

$$\begin{aligned}
e^{\mu(e^{-t} - 1 + t(1-\delta))} &\leq e^{\mu(e^{\ln(1-\delta)} - 1 - (1-\delta)\ln(1-\delta))} \\
&= e^{\mu((1-\delta) - 1 - (1-\delta)\ln(1-\delta))} \\
&= e^{\mu(-\delta - (1-\delta)\ln(1-\delta))} \\
&= (e^{-\delta - (1-\delta)\ln(1-\delta)})^\mu \\
&= \left(\frac{e^{-\delta}}{e^{(1-\delta)\ln(1-\delta)}}\right)^\mu \\
&= \left(\frac{e^{-\delta}}{(e^{\ln(1-\delta)})^{1-\delta}}\right)^\mu \\
&= \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^\mu
\end{aligned}$$

In lieu of applying Chernoff bounds to the algorithm for estimating  $\pi$ , we consider a different example of flipping a coin. If we flip a biased coin with probability  $p$  of heads, we expect to see  $np$  heads. Let  $H$  be the total number of heads, and let  $H_i$  be an indicator variable corresponding to whether the  $i$ th flip is heads. We have  $\Pr[H_i = 1] = p$ , and  $\mathbb{E}[H] = np$  by linearity of expectation.

Suppose the coin is fair. What is the probability of getting at least six heads out of ten flips? This is a fractional deviation of  $\delta = 6/5 - 1 = 0.2$ , and applying the upper-tail Chernoff bound gives us:

$$\begin{aligned}
\Pr[H \geq (1 + 0.2) \cdot 5] &\leq \left(\frac{e^{0.2}}{1.2^{1.2}}\right)^5 \\
&\approx 0.9814^5 \\
&\approx 0.91
\end{aligned}$$

What is the probability of getting at least 60 heads out of 100 flips, which is the same fractional deviation  $\delta = 0.2$  from the expectation? Applying the upper tail again, we get

$$\begin{aligned}
\Pr[H \geq (1 + 0.2) \cdot 50] &\leq \left(\frac{e^{0.2}}{1.2^{1.2}}\right)^{50} \\
&\approx 0.9814^{50} \\
&\approx 0.39
\end{aligned}$$

Thus, the probability of deviating by a factor of  $1 + \delta$  decreases significantly as the number of samples increases. For this particular example, we can compute the actual probabilities quite tediously from exact formulas for a [binomial distribution](#)<sup>107</sup>, which yields 0.37 for getting six heads out of ten flips and 0.03 for 60 heads out of 100 flips. However, this approach becomes more and more expensive as  $n$  increases, and the Chernoff bound produces a reasonable result with much less work.

<sup>107</sup> [https://en.wikipedia.org/wiki/Binomial\\_distribution](https://en.wikipedia.org/wiki/Binomial_distribution)

### 28.2.1 Polling Analysis with Chernoff Bounds

Recall that in *polling* (page 236), a confidence level  $1 - \gamma$  and margin of error  $\varepsilon$  requires

$$\Pr\left[\left|\frac{X}{n} - p\right| \leq \varepsilon\right] \geq 1 - \gamma$$

or equivalently

$$\Pr\left[\left|\frac{X}{n} - p\right| > \varepsilon\right] < \gamma$$

Formally, we define indicator variables  $X_i$  as

$$X_i = \begin{cases} 1 & \text{if person } i \text{ supports the candidate} \\ 0 & \text{otherwise} \end{cases}$$

for each person  $i$  in the set that we poll. Then  $X = X_1 + \dots + X_n$  is the sum of independent indicator variables, with

$$\mu = \mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = np$$

For an arbitrary margin of error  $\varepsilon$ , we want to bound the probability

$$\begin{aligned} \Pr\left[\left|\frac{X}{n} - p\right| > \varepsilon\right] &= \Pr\left[\frac{X}{n} - p > \varepsilon\right] + \Pr\left[\frac{X}{n} - p < -\varepsilon\right] \\ &= \Pr\left[\frac{X}{n} > \varepsilon + p\right] + \Pr\left[\frac{X}{n} < -\varepsilon + p\right] \\ &= \Pr[X > \varepsilon n + pn] + \Pr[X < -\varepsilon n + pn] \\ &= \Pr[X > \varepsilon n + \mu] + \Pr[X < -\varepsilon n + \mu] \\ &= \Pr[X > (1 + \varepsilon n/\mu)\mu] + \Pr[X < (1 - \varepsilon n/\mu)\mu] \\ &= \Pr[X \geq (1 + \varepsilon n/\mu)\mu] + \Pr[X \leq (1 - \varepsilon n/\mu)\mu] \\ &\quad - \Pr[X = (1 + \varepsilon n/\mu)\mu] - \Pr[X = (1 - \varepsilon n/\mu)\mu] \\ &\leq \Pr[X \geq (1 + \varepsilon n/\mu)\mu] + \Pr[X \leq (1 - \varepsilon n/\mu)\mu] \end{aligned}$$

In the second-to-last step, we use the fact that  $X = x$  and  $X > x$  are disjoint events, since a random variable maps each sample point to a single value, and that  $(X \geq x) = (X = x) \cup (X > x)$ .

We now have events that are in the right form to apply a Chernoff bound, with  $\delta = \varepsilon n/\mu$ . We first apply the simplified upper-tail bound to the first term, obtaining

$$\begin{aligned} \Pr[X \geq (1 + \varepsilon n/\mu)\mu] &\leq e^{-\frac{\delta^2 \mu}{2+\delta}} \quad \text{where } \delta = \varepsilon n/\mu \\ &= e^{-\frac{(\varepsilon n)^2 \mu}{\mu^2(2+\varepsilon n/\mu)}} \\ &= e^{-\frac{(\varepsilon n)^2}{\mu(2+\varepsilon n/\mu)}} \\ &= e^{-\frac{(\varepsilon n)^2}{2\mu+\varepsilon n}} \\ &= e^{-\frac{(\varepsilon n)^2}{2np+\varepsilon n}} \\ &= e^{-\frac{\varepsilon^2 n}{2p+\varepsilon}} \end{aligned}$$

We want this to be less than some value  $\beta$ :

$$\begin{aligned} e^{-\frac{\varepsilon^2 n}{2p+\varepsilon}} &< \beta \\ \frac{1}{\beta} &< e^{\frac{\varepsilon^2 n}{2p+\varepsilon}} \\ \ln\left(\frac{1}{\beta}\right) &< \frac{\varepsilon^2 n}{2p+\varepsilon} \\ n &> \frac{2p+\varepsilon}{\varepsilon^2} \ln\left(\frac{1}{\beta}\right) \end{aligned}$$

Unfortunately, this expression includes  $p$ , the quantity we're trying to estimate. However, we know that  $p \leq 1$ , so we can set

$$n > \frac{2+\varepsilon}{\varepsilon^2} \ln\left(\frac{1}{\beta}\right)$$

to ensure that  $\Pr[X \geq (1 + \varepsilon n/\mu)\mu] < \beta$ .

We can also apply the simplified lower-tail bound to the second term in our full expression above:

$$\begin{aligned} \Pr[X \leq (1 - \varepsilon n/\mu)\mu] &\leq e^{-\frac{\delta^2 \mu}{2}} \quad \text{where } \delta = \varepsilon n/\mu \\ &= e^{-\frac{(\varepsilon n)^2 \mu}{2\mu^2}} \\ &= e^{-\frac{(\varepsilon n)^2}{2\mu}} \\ &= e^{-\frac{(\varepsilon n)^2}{2np}} \\ &= e^{-\frac{\varepsilon^2 n}{2p}} \end{aligned}$$

As before, we want this to be less some value  $\beta$ :

$$\begin{aligned} e^{-\frac{\varepsilon^2 n}{2p}} &< \beta \\ \frac{1}{\beta} &< e^{\frac{\varepsilon^2 n}{2p}} \\ \ln\left(\frac{1}{\beta}\right) &< \frac{\varepsilon^2 n}{2p} \\ n &> \frac{2p}{\varepsilon^2} \ln\left(\frac{1}{\beta}\right) \end{aligned}$$

We again observe that  $p \leq 1$ , so we can set

$$n > \frac{2}{\varepsilon^2} \ln\left(\frac{1}{\beta}\right)$$

to ensure that  $\Pr[X \leq (1 - \varepsilon n/\mu)\mu] < \beta$ .

To bound both terms simultaneously, we can choose  $\beta = \gamma/2$ , so that

$$\Pr[X \geq (1 + \varepsilon n/\mu)\mu] + \Pr[X \leq (1 - \varepsilon n/\mu)\mu] < 2\beta = \gamma$$

To achieve this, we require

$$\begin{aligned} n &> \max\left(\frac{2+\varepsilon}{\varepsilon^2} \ln\left(\frac{2}{\gamma}\right), \frac{2}{\varepsilon^2} \ln\left(\frac{2}{\gamma}\right)\right) \\ &= \frac{2+\varepsilon}{\varepsilon^2} \ln\left(\frac{2}{\gamma}\right) \end{aligned}$$

For example, for a 95% confidence level and a margin of error of  $\pm 2\%$ , we have  $\varepsilon = 0.02$  and  $\gamma = 0.05$ . Plugging those values into the result above, we need no more than

$$\frac{2 + \varepsilon}{\varepsilon^2} \ln\left(\frac{2}{\gamma}\right) = \frac{2.02}{0.02^2} \ln 40 \approx 18623$$

samples to achieve the desired confidence level and margin of error. Observe that this does not depend on the total population size!

## 28.3 Probabilistic Complexity Classes

The *Fermat primality test* (page 271) is an example of a *one-sided-error randomized algorithm* – an input that is pseudoprime is always accepted, while a non-pseudoprime is sometimes accepted and sometimes rejected. We can define complexity classes corresponding to decision problems that have efficient one-sided-error randomized algorithms.

**Definition 313 (RP)** RP is the class of languages that have efficient one-sided-error randomized algorithms with no *false positives*. A language  $L$  is in RP if there is an efficient randomized algorithm  $A$  such that:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq c$
- if  $x \notin L$ ,  $\Pr[A \text{ rejects } x] = 1$

Here,  $c$  must be a constant greater than 0. Often,  $c$  is chosen to be  $\frac{1}{2}$ .

**Definition 314 (coRP)** coRP is the class of languages that have efficient one-sided-error randomized algorithms with no *false negatives*. A language  $L$  is in coRP if there is an efficient randomized algorithm  $A$  such that:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] = 1$
- if  $x \notin L$ ,  $\Pr[A \text{ rejects } x] \geq c$

Here,  $c$  must be a constant greater than 0. Often,  $c$  is chosen to be  $\frac{1}{2}$ .

RP stands for *randomized polynomial* time. If a language  $L$  is in RP, then its complement language  $\bar{L}$  is in coRP – an algorithm for  $L$  with no false positives can be converted into an algorithm for  $\bar{L}$  with no false negatives. The Fermat test produces no false negatives, so PSEUDOPRIMES  $\in$  coRP. Thus, the language

$$\text{PSEUDOPRIMES} = \left\{ m \in \mathbb{N} : \begin{array}{l} a^{m-1} \bmod m \equiv 1 \text{ for at least half} \\ \text{the witnesses } 1 \leq a \leq m-1 \end{array} \right\}$$

is in RP.

The constant  $c$  in the definition of RP and coRP is arbitrary. With any  $c > 0$ , we can *amplify* the probability of success by repeatedly running the algorithm. For instance, if we have a randomized algorithm  $A$  with no false positives for a language  $L$ , we have:

$$\begin{aligned} x \in L &\implies \Pr[A \text{ accepts } x] \geq c \\ x \notin L &\implies \Pr[A \text{ accepts } x] = 0 \end{aligned}$$

We can construct a new algorithm  $B$  as follows:

```
function B(x)
    run A(x) twice (with fresh randomness each time)
    if it accepts at least once then accept
        reject
```

$B$  just runs  $A$  twice on the input, accepting if at least one run of  $A$  accepts;  $B$  only rejects if both runs of  $A$  reject. Thus, the probability that  $B$  rejects  $x$  is:

$$\begin{aligned}\Pr[B \text{ rejects } x] &= \Pr[A \text{ rejects } x \text{ in run 1}, A \text{ rejects } x \text{ in run 2}] \\ &= \Pr[A \text{ rejects } x \text{ in run 1}] \cdot \Pr[A \text{ rejects } x \text{ in run 2}] \\ &= \Pr[A \text{ rejects } x]^2\end{aligned}$$

Here, we have used the fact that the two runs of  $A$  are independent, so the probability of  $A$  rejecting twice is the product of the probabilities it rejects each time. This gives us:

$$\begin{aligned}x \in L \implies \Pr[B \text{ rejects } x] &\leq (1 - c)^2 \\ x \notin L \implies \Pr[B \text{ rejects } x] &= 1\end{aligned}$$

or equivalently:

$$\begin{aligned}x \in L \implies \Pr[B \text{ accepts } x] &\geq 1 - (1 - c)^2 \\ x \notin L \implies \Pr[B \text{ accepts } x] &= 0\end{aligned}$$

Repeating this reasoning, if we modify  $B$  to run  $A$   $k$  times, we get:

$$\begin{aligned}x \in L \implies \Pr[B \text{ accepts } x] &\geq 1 - (1 - c)^k \\ x \notin L \implies \Pr[B \text{ accepts } x] &= 0\end{aligned}$$

By applying a form of Bernoulli's inequality<sup>108</sup>,  $(1 + x)^r \leq e^{rx}$ , we get:

$$x \in L \implies \Pr[B \text{ accepts } x] \geq 1 - e^{-ck}$$

If we want a particular lower bound  $1 - \varepsilon$  for acceptance of true positives, we need

$$\begin{aligned}e^{-ck} &\leq \varepsilon \\ -ck &\leq \ln \varepsilon \\ k &\geq \frac{-\ln \varepsilon}{c} = \frac{\ln(1/\varepsilon)}{c}\end{aligned}$$

This is a constant for any constants  $c$  and  $\varepsilon$ . Thus, we can amplify the probability of accepting a true positive from  $c$  to  $1 - \varepsilon$  for any  $\varepsilon$  with a constant  $\ln(1/\varepsilon)/c$  number of repetitions. The same reasoning can be applied to amplify one-sided-error algorithms with no false negatives.

One-sided-error randomized algorithms are a special case of *two-sided-error randomized algorithms*. Such an algorithm for a language  $L$  can produce the wrong result for an input  $x$  whether or not  $x \in L$ . However, the probability of getting the wrong result is bounded by a constant.

**Definition 315 (BPP)** BPP is the class of languages that have efficient two-sided-error randomized algorithms. A language  $L$  is in BPP if there is an efficient randomized algorithm  $A$  such that:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq c$
- if  $x \notin L$ ,  $\Pr[A \text{ rejects } x] \geq c$

Here,  $c$  must be a constant greater than  $\frac{1}{2}$ , so that the algorithm produces the correct answer the majority of the time. Often,  $c$  is chosen to be  $\frac{2}{3}$  or  $\frac{3}{4}$ .

BPP stands for *bounded-error probabilistic polynomial* time. Given the symmetric definition of a two-sided error algorithm, it is clear that the class BPP is closed under complement – if a language  $L \in \text{BPP}$ , then  $\bar{L} \in \text{BPP}$  as well.

Languages in RP and in coRP both trivially satisfy the conditions for BPP; for instance, we have the following for RP:

<sup>108</sup> [https://en.wikipedia.org/wiki/Bernoulli%27s\\_inequality](https://en.wikipedia.org/wiki/Bernoulli%27s_inequality)

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq c$
- if  $x \notin L$ ,  $\Pr[A \text{ rejects } x] = 1 \geq c$

Thus,  $\text{RP} \cup \text{coRP} \subseteq \text{BPP}$ . By similar reasoning, we can relate P to these classes as follows:

$$\begin{aligned} \text{P} &\subseteq \text{RP} \subseteq \text{BPP} \\ \text{P} &\subseteq \text{coRP} \subseteq \text{BPP} \end{aligned}$$

As with one-sided-error algorithms, the probability of success for two-sided-error randomized algorithms can be amplified arbitrarily, as *we will see shortly* (page 287).

One-sided-error and two-sided-error randomized algorithms are known as *Monte Carlo* algorithms. Such algorithms have a bounded runtime, but they may produce the wrong answer. Contrast this with *Las Vegas* algorithms, which always produce the correct answer, but where the runtime bound only holds *in expectation*. We can define a complexity class for languages that have Las Vegas algorithms with expected runtime that is polynomial in the size of the input.

**Definition 316 (ZPP)** ZPP is the class of languages that have efficient Las Vegas algorithms. A language  $L$  is in ZPP if there is a randomized algorithm  $A$  such that:

- If  $x \in L$ ,  $A$  always accepts  $x$ .
- If  $x \notin L$ ,  $A$  always rejects  $x$ .
- The expected runtime of  $A$  on input  $x$  is  $O(|x|^k)$  for some constant  $k$ .

There is a relationship between Monte Carlo and Las Vegas algorithms: if a language  $L$  has both a one-sided-error algorithm with no false positives and a one-sided-error algorithm with no false negatives, then it has a Las Vegas algorithm, and vice versa. In terms of complexity classes, we have  $L$  is in both RP and coRP if and only if  $L$  is in ZPP. This implies that

$$\text{RP} \cap \text{coRP} = \text{ZPP}$$

We demonstrate this as follows. Suppose  $L$  is in  $\text{RP} \cap \text{coRP}$ . Then it has an efficient, one-sided-error algorithm  $A$  with no false positives such that:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq \frac{1}{2}$
- if  $x \notin L$ ,  $\Pr[A \text{ accepts } x] = 0$

Here, we have selected  $c = \frac{1}{2}$  for concreteness.  $L$  also has an efficient, one-sided algorithm  $B$  with no false negatives such that:

- if  $x \in L$ ,  $\Pr[B \text{ rejects } x] = 0$
- if  $x \notin L$ ,  $\Pr[B \text{ rejects } x] \geq \frac{1}{2}$

We can construct a new algorithm  $C$  as follows:

```
function C(x)
    while true do
        if A(x) accepts then accept
        if B(x) rejects then reject
```

Since  $A$  only accepts  $x \in L$  and  $C$  only accepts when  $A$  does,  $C$  only accepts  $x \in L$ . Similarly,  $B$  only rejects  $x \notin L$ , so  $C$  only rejects  $x \notin L$ . Thus,  $C$  always produces the correct answer for a given input.

To show that  $L \in \text{ZPP}$ , we must also demonstrate that the expected runtime of  $C$  is polynomial. For each iteration  $i$  of  $C$ , we have:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x \text{ in iteration } i] \geq \frac{1}{2}$

- if  $x \notin L$ ,  $\Pr[B \text{ rejects } x \text{ in iteration } i] \geq \frac{1}{2}$

Thus, if  $C$  gets to iteration  $i$ , the probability that  $C$  terminates in that iteration is at least  $\frac{1}{2}$ . We can model the number of iterations as a random variable  $X$ , and  $X$  has the probability distribution:

$$\Pr[X > k] \leq \left(\frac{1}{2}\right)^k$$

This is similar to a *geometric distribution*, where

$$\Pr[Y > k] = (1 - p)^k$$

for some  $p \in (0, 1]$ . The expected value of such a distribution is  $\mathbb{E}[Y] = 1/p$ . This gives us:

$$\mathbb{E}[X] \leq 2$$

Thus, the expected number of iterations of  $C$  is no more than two, and since  $A$  and  $B$  are both efficient, calling them twice is also efficient.

We have shown that  $\text{RP} \cap \text{coRP} \subseteq \text{ZPP}$ . We will proceed to show that  $\text{ZPP} \subseteq \text{RP}$ , meaning that if a language has an efficient Las Vegas algorithm, it also has an efficient one-sided-error algorithm with no false positives. Assume that  $C$  is a Las Vegas algorithm for  $L \in \text{ZPP}$ , with expected runtime of no more than  $|x|^k$  steps for an input  $x$  and some constant  $k$ . We construct a new algorithm  $A$  as follows:

```
function  $A(x)$  simulate  $C(x)$  for up to  $2|x|^k$  steps (or until it halts)
    if it accepts then accept
    reject
```

$A$  is clearly efficient in the size of the input  $x$ . It also has no false positives, since it only accepts if  $C$  accepts within the first  $2|x|^k$  steps, and  $C$  always produces the correct answer. All that is left is to show that if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq \frac{1}{2}$  (again, we arbitrarily choose  $c = \frac{1}{2}$ ).

Let  $S$  be the number of steps before  $C$  accepts  $x$ . By assumption, we have

$$\mathbb{E}[S] \leq |x|^k$$

$A$  runs  $C$  for  $2|x|^k$  steps, so we need to bound the probability that  $C$  takes more than  $2|x|^k$  steps on  $x$ . We have:

$$\begin{aligned} \Pr[S > 2|x|^k] &\leq \Pr[S \geq 2|x|^k] \\ &\leq \frac{\mathbb{E}[S]}{2|x|^k} \\ &\leq \frac{|x|^k}{2|x|^k} \\ &= \frac{1}{2} \end{aligned}$$

Here, we applied Markov's inequality in the second step. Then:

$$\begin{aligned} \Pr[A \text{ accepts } x] &= \Pr[S \leq 2|x|^k] \\ &= 1 - \Pr[S > 2|x|^k] \\ &\geq \frac{1}{2} \end{aligned}$$

Thus,  $A$  is indeed a one-sided-error algorithm with no false positives, so  $L \in \text{RP}$ . A similar construction can be used to demonstrate that  $L \in \text{coRP}$ , allowing us to conclude that  $\text{ZPP} \subseteq \text{RP} \cap \text{coRP}$ . Combined with our previous proof that  $\text{RP} \cap \text{coRP} \subseteq \text{ZPP}$ , we have  $\text{RP} \cap \text{coRP} = \text{ZPP}$ .

One final observation we will make is that if a language  $L$  is in RP, then it is also in NP. Since  $L \in RP$ , there is an efficient, one-sided-error randomized algorithm  $A$  to decide  $L$ .  $A$  is allowed to make random choices, and we can model each choice as a coin flip, i.e. being either 0 or 1 according to some probability distribution. We can represent the combination of these choices in a particular run of  $A$  as a binary string  $c$ . This enables us to write an efficient verifier  $V$  for  $L$  as follows:

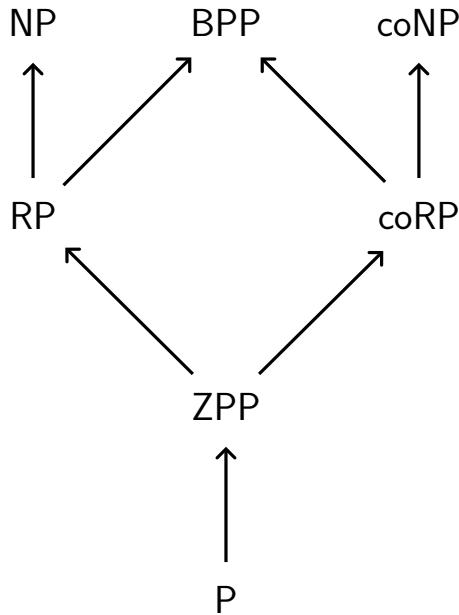
```
function  $V(x, c = c_1c_2 \cdots c_m)$  simulate  $A(x)$ , using  $c_i$  for the  $i$ th random bit used by  $A$ 
    if  $A$  accepts then accept
    reject
```

If  $x \in L$ , then  $\Pr[A \text{ accepts } x] \geq \frac{1}{2}$ , so at least half the possible sequences of random choices lead to  $A$  accepting  $x$ . On the other hand, if  $x \notin L$ , then  $\Pr[A \text{ rejects } x] = 1$ , so all sequences of random choices lead to  $A$  rejecting  $x$ . Thus,  $V$  accepts at least half of all possible certificates when  $x \in L$ , and  $V$  rejects all certificates when  $x \notin L$ , so  $V$  is a verifier for  $L$ . Since  $A$  is efficient,  $V$  is also efficient.

We summarize the known relationships between complexity classes as follows:

$$\begin{aligned} P &\subseteq ZPP \subseteq RP \subseteq NP \\ P &\subseteq ZPP \subseteq RP \subseteq BPP \\ P &\subseteq ZPP \subseteq coRP \subseteq coNP \\ P &\subseteq ZPP \subseteq coRP \subseteq BPP \end{aligned}$$

These relationships are represented pictorially, with edges signifying containment, as follows:



Here,  $coNP$  is the class of languages whose complements are in  $NP$ :

$$coNP = \{L : \bar{L} \in NP\}$$

We do not know if any of the containments above are strict, and we do not know the relationships between  $NP$ ,  $coNP$ , and  $BPP$ . It is commonly **believed** that  $P$  and  $BPP$  are equal and thus  $BPP$  is contained in  $NP \cap coNP$ , that neither  $P$  nor  $BPP$  contain all of  $NP$  or all of  $coNP$ , and that  $NP$  and  $coNP$  are not equal. However, none of these conjectures

has been proven<sup>109</sup>.

## 28.4 Amplification for Two-Sided-Error Algorithms

Previously, we saw how to *amplify the probability of success for one-sided-error randomized algorithms* (page 282). We now consider amplification for *two-sided-error algorithms* (page 283). Recall that such an algorithm  $A$  for a language  $L$  has the following behavior:

- if  $x \in L$ ,  $\Pr[A \text{ accepts } x] \geq c$
- if  $x \notin L$ ,  $\Pr[A \text{ rejects } x] \geq c$

Here,  $c$  must be a constant that is strictly greater than  $\frac{1}{2}$ .

Unlike in the one-sided case with no false positives, we cannot just run the algorithm multiple times and observe if it accepts at least once. A two-sided-error algorithm can accept both inputs in and not in the language, and it can reject both such inputs as well. However, we observe that because  $c > \frac{1}{2}$ , we expect to get the right answer the majority of the time when we run a two-sided-error algorithm on an input. More formally, suppose we run the algorithm  $n$  times. Let  $Y_i$  be an indicator random variable that is 1 if the algorithm accepts in the  $i$ th run. Then:

- if  $x \in L$ ,  $\mathbb{E}[Y_i] = \Pr[Y_i = 1] \geq c$
- if  $x \notin L$ ,  $\mathbb{E}[Y_i] = \Pr[Y_i = 1] \leq 1 - c$

Let  $Y = Y_1 + \dots + Y_n$  be the total number of accepts out of  $n$  trials. By linearity of expectation, we have:

- if  $x \in L$ ,  $\mathbb{E}[Y] \geq cn > \frac{n}{2}$  (since  $c > \frac{1}{2}$ )
- if  $x \notin L$ ,  $\mathbb{E}[Y] \leq (1 - c)n < \frac{n}{2}$  (since  $1 - c < \frac{1}{2}$ )

This motivates an amplification algorithm  $B$  that runs the original algorithm  $A$  multiple times and takes the majority vote:

```
function  $B(x)$ 
    run  $A(x)$   $n$  times (with fresh randomness each time)
    if it accepts at least  $n/2$  times then accept
        reject
```

Suppose we wish to obtain a bound that is within  $\gamma$  of 1:

- if  $x \in L$ ,  $\Pr[B \text{ accepts } x] \geq 1 - \gamma$
- if  $x \notin L$ ,  $\Pr[B \text{ rejects } x] \geq 1 - \gamma$

What value for  $n$  should we use in  $B$ ?

We first consider the case where  $x \in L$ . The indicators  $Y_i$  are independent, allowing us to use Hoeffding's inequality on their sum  $Y$ .  $B$  accepts  $x$  when  $Y \geq \frac{n}{2}$ , or equivalently,  $\frac{Y}{n} \geq \frac{1}{2}$ . We want

$$\Pr\left[\frac{Y}{n} \geq \frac{1}{2}\right] \geq 1 - \gamma$$

<sup>109</sup> It is known, however, that if  $P = NP$ , then  $P = BPP$ . This is because  $BPP$  is contained within the polynomial hierarchy<sup>110</sup>, and one of the consequences of  $P = NP$  is the “collapse” of the hierarchy. The details are beyond the scope of this text.

<sup>110</sup> [https://en.wikipedia.org/wiki/Polynomial\\_hierarchy](https://en.wikipedia.org/wiki/Polynomial_hierarchy)

or equivalently,

$$\begin{aligned}\Pr\left[\frac{Y}{n} < \frac{1}{2}\right] &\leq \Pr\left[\frac{Y}{n} \leq \frac{1}{2}\right] \\ &= \Pr\left[\frac{Y}{n} \leq c - (c - \frac{1}{2})\right] \\ &= \Pr\left[\frac{Y}{n} \leq c - \varepsilon\right] \\ &\leq \gamma\end{aligned}$$

where  $\varepsilon = c - \frac{1}{2}$ . To apply Hoeffding's inequality, we need something of the form

$$\Pr\left[\frac{Y}{n} \leq p - \varepsilon\right]$$

where  $p = \mathbb{E}\left[\frac{Y}{n}\right]$ . Unfortunately, we do not know the exact value of  $p$ ; all we know is that  $p = \mathbb{E}\left[\frac{Y}{n}\right] \geq c$ . However, we know that because  $p \geq c$ ,

$$\Pr\left[\frac{Y}{n} \leq c - \varepsilon\right] \leq \Pr\left[\frac{Y}{n} \leq p - \varepsilon\right]$$

In general, the event  $X \leq a$  includes at most as many sample points as  $X \leq b$  when  $a \leq b$ ; the latter includes all the outcomes in  $X \leq a$ , as well as those in  $a < X \leq b$ . We thus need only compute an upper bound on  $\Pr\left[\frac{Y}{n} \leq p - \varepsilon\right]$ , and that same upper bound will apply to  $\Pr\left[\frac{Y}{n} \leq c - \varepsilon\right]$ . Taking  $\varepsilon = c - \frac{1}{2}$  and applying the lower-tail Hoeffding's inequality, we get:

$$\begin{aligned}\Pr\left[\frac{Y}{n} \leq p - \varepsilon\right] &= \Pr\left[\frac{Y}{n} \leq p - (c - \frac{1}{2})\right] \\ &\leq e^{-2(c-1/2)^2 n}\end{aligned}$$

We want this to be bound by  $\gamma$ :

$$\begin{aligned}e^{-2(c-1/2)^2 n} &\leq \gamma \\ e^{2(c-1/2)^2 n} &\geq \frac{1}{\gamma} \\ 2(c-1/2)^2 n &\geq \ln(\frac{1}{\gamma}) \\ n &\geq \frac{1}{2(c-1/2)^2} \ln(\frac{1}{\gamma})\end{aligned}$$

As a concrete example, suppose  $c = \frac{3}{4}$ , meaning that  $A$  accepts  $x \in L$  with probability at least  $\frac{3}{4}$ . Suppose we want  $B$  to accept  $x \in L$  at least 99% of the time, giving us  $\gamma = 0.01$ . Then:

$$\begin{aligned}n &\geq \frac{1}{2(3/4 - 1/2)^2} \ln(\frac{1}{0.01}) \\ &= \frac{1}{2 \cdot 0.25^2} \ln 100 \\ &\approx 36.8\end{aligned}$$

Thus, it is sufficient for  $B$  to run  $n \geq 37$  trials of  $A$  on  $x$ .

We now consider  $x \notin L$ . We want  $B$  to reject  $x$  with probability at least  $1 - \gamma$ , or equivalently,  $B$  to accept  $x$  with probability at most  $\gamma$ :

$$\Pr\left[\frac{Y}{n} \geq \frac{1}{2}\right] \leq \gamma$$

Similar to before, we have

$$\begin{aligned}\Pr\left[\frac{Y}{n} \geq \frac{1}{2}\right] &= \Pr\left[\frac{Y}{n} \geq (1 - c) + (c - \frac{1}{2})\right] \\ &= \Pr\left[\frac{Y}{n} \geq (1 - c) + \varepsilon\right]\end{aligned}$$

with  $\varepsilon = c - \frac{1}{2}$ . Since  $p = \mathbb{E}\left[\frac{Y}{n}\right] \leq (1 - c)$ , we have

$$\Pr\left[\frac{Y}{n} \geq (1 - c) + \varepsilon\right] \leq \Pr\left[\frac{Y}{n} \geq p + \varepsilon\right]$$

This follows from similar reasoning as earlier: an event  $X \geq a$  contains no more sample points than  $X \geq b$  when  $a \geq b$ . With  $\varepsilon = c - \frac{1}{2}$ , the upper-tail Hoeffding's inequality gives us:

$$\begin{aligned}\Pr\left[\frac{Y}{n} \geq p + \varepsilon\right] &= \Pr\left[\frac{Y}{n} \geq p + (c - \frac{1}{2})\right] \\ &\leq e^{-2(c-1/2)^2 n}\end{aligned}$$

We want this to be no more than  $\gamma$ , which leads to the same solution as before:

$$n \geq \frac{1}{2(c - 1/2)^2} \ln\left(\frac{1}{\gamma}\right)$$

Using the same concrete example as before, if we want  $B$  to reject  $x \notin L$  at least 99% of the time when  $A$  rejects  $x \notin L$  with probability at least  $\frac{3}{4}$ , it suffices for  $B$  to run  $n \geq 37$  trials of  $A$  on  $x$ .

# **Part VII**

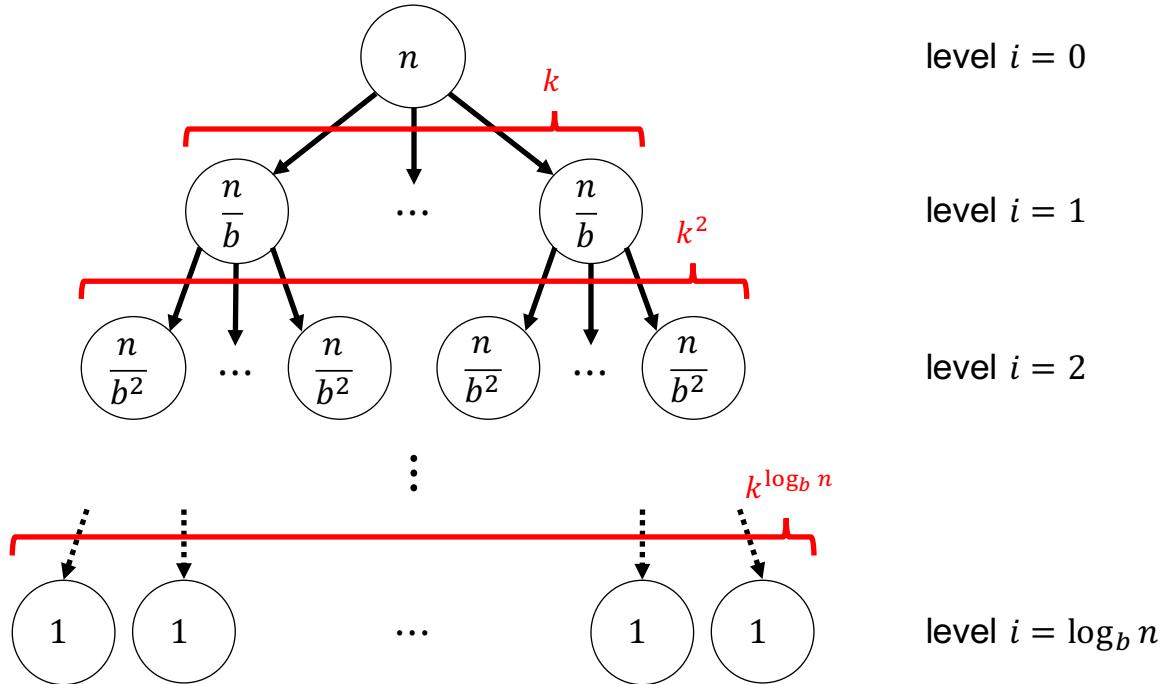
# **Appendix**

## 29.1 Proof of the Master Theorem

We first prove the standard *master theorem* (page 14) without log factors.

**Proof 317** We prove the master theorem for the case where  $n$  is a power of  $b$ ; when this is not the case, we can pad the input size to the smallest power of  $b$  that is larger than  $n$ , which increases the input size by at most the constant factor  $b$  (can you see why?).

First, we determine how many subproblems we have at each level of the recursion. Initially, we have a single problem of size  $n$ . We subdivide it into  $k$  subproblems, each of size  $n/b$ . Each of those gets subdivided into  $k$  subproblems of size  $n/b^2$ , for a total of  $k^2$  problems of size  $n/b^2$ . Those  $k^2$  problems in turn are subdivided into a total of  $k^3$  problems of size  $n/b^3$ .



In general, at the  $i$ th level of recursion (with  $i = 0$  the initial problem), there are  $k^i$  subproblems of size  $n/b^i$ . We

assume that the base case is when the problem size is 1, which is when

$$\begin{aligned} 1 &= \frac{n}{b^i} \\ b^i &= n \\ i &= \log_b n \end{aligned}$$

Thus, there are  $1 + \log_b n$  total levels in the recursion.

We now consider how much work is done in each subproblem, which corresponds to the  $O(n^d)$  term of the recurrence relation. For a subproblem of size  $n/b^i$ , this is

$$\begin{aligned} O\left(\left(\frac{n}{b^i}\right)^d\right) &= O\left(\frac{n^d}{b^{id}}\right) \\ &= b^{-id} \cdot O(n^d) \end{aligned}$$

With  $k^i$  subproblems at level  $i$ , the total work  $T_i$  at level  $i$  is

$$\begin{aligned} T_i &= \frac{k^i}{b^{id}} \cdot O(n^d) \\ &= \left(\frac{k}{b^d}\right)^i \cdot O(n^d) \end{aligned}$$

Summing over all the levels, we get a total work

$$\begin{aligned} T &= \sum_{i=0}^{\log_b n} T_i \\ &= \sum_{i=0}^{\log_b n} \left(\left(\frac{k}{b^d}\right)^i \cdot O(n^d)\right) \\ &= O(n^d) \cdot \sum_{i=0}^{\log_b n} \left(\frac{k}{b^d}\right)^i \\ &= O(n^d) \cdot \sum_{i=0}^{\log_b n} r^i \\ &= O(n^d \cdot \sum_{i=0}^{\log_b n} r^i) \\ &= O(n^d \cdot (1 + r + r^2 + \dots + r^{\log_b n})) \end{aligned}$$

where  $r = k/b^d$ .

Since we are working with asymptotics, we only care about the dominating term in the sum

$$\sum_{i=0}^{\log_b n} r^i = 1 + r + r^2 + \dots + r^{\log_b n}$$

There are three cases:

- $r < 1$ : The terms have decreasing value, so that the initial term 1 is the largest and dominates<sup>111</sup>. Thus, we have

$$\begin{aligned} T &= O(n^d \cdot (1 + r + r^2 + \dots + r^{\log_b n})) \\ &= O(n^d) \end{aligned}$$

- $r = 1$ : The terms all have the same value 1. Then

$$\begin{aligned} T &= O(n^d \cdot \sum_{i=0}^{\log_b n} r^i) \\ &= O(n^d \cdot \sum_{i=0}^{\log_b n} 1) \\ &= O(n^d \cdot (1 + \log_b n)) \\ &= O(n^d + n^d \log_b n) \\ &= O(n^d \log_b n) \end{aligned}$$

where we discard the lower-order term in the last step. Since logarithms of different bases differ only by a multiplicative constant factor (see the [logarithmic identity for changing the base](#)<sup>114</sup>), which does not affect asymptotics, we can elide the base:

$$\begin{aligned} T &= O(n^d \log_b n) \\ &= O(n^d \log n) \end{aligned}$$

- $r > 1$ : The terms have increasing value, so that the final term  $r^{\log_b n}$  is the largest. Then we have

$$\begin{aligned} T &= O(n^d \cdot (1 + r + r^2 + \dots + r^{\log_b n})) \\ &= O(n^d r^{\log_b n}) \\ &= O(n^d (\frac{k}{b^d})^{\log_b n}) \\ &= O(n^d \cdot k^{\log_b n} \cdot b^{-d \log_b n}) \\ &= O(n^d \cdot k^{\log_b n} \cdot (b^{\log_b n})^{-d}) \end{aligned}$$

To simplify this further, we observe the following:

- $b^{\log_b n} = n$ , which we can see by taking  $\log_b$  of both sides.
- $k^{\log_b n} = n^{\log_b k}$ . We show this as follows:

$$k^{\log_b n} = (b^{\log_b k})^{\log_b n}$$

Here, we applied the previous observation to substitute  $k = b^{\log_b k}$ . We proceed to get

$$\begin{aligned} k^{\log_b n} &= (b^{\log_b k})^{\log_b n} \\ &= (b^{\log_b n})^{\log_b k} \\ &= n^{\log_b k} \end{aligned}$$

applying the prior observation once again.

Applying both observations, we get

$$\begin{aligned} T &= O(n^d \cdot k^{\log_b n} \cdot (b^{\log_b n})^{-d}) \\ &= O(n^d \cdot n^{\log_b k} \cdot n^{-d}) \\ &= O(n^{\log_b k}) \end{aligned}$$

Here, we cannot discard the base of the logarithm, as it appears in the exponent and so affects the value by an exponential rather than a constant factor.

Thus, we have demonstrated all three cases of the master theorem.  $\square$

<sup>111</sup> We're being a bit sloppy here – the largest term in a sum doesn't necessarily dominate, as can be seen with a divergent series such as the harmonic series:<sup>7</sup>  $\sum_{n=1}^{\infty} 1/n$ . However, what we have here is a geometric series:<sup>8</sup>  $\sum_{i=0}^n r^i$ , which has the closed-form solution  $\sum_{i=0}^n r^i = (1 - r^{n+1})/(1 - r)$  for  $r \neq 1$ . When  $0 \leq r < 1$  (and therefore  $r^{n+1} < 1$ ), this is bounded from above by the constant  $1/(1 - r)$ , so the sum is  $O(1)$ .

<sup>114</sup> [https://en.wikipedia.org/wiki/Logarithm#Change\\_of\\_base](https://en.wikipedia.org/wiki/Logarithm#Change_of_base)

We now proceed to prove the version with log factors.

**Proof 318** We need to modify the reasoning in Proof 317 to account for the amount of work done in each subproblem. For a subproblem of size  $n/b^i$ , we now have work

$$O\left(\left(\frac{n}{b^i}\right)^d \log^w \frac{n}{b^i}\right)$$

We consider the cases of  $k/b^d \leq 1$  and  $k/b^d > 1$  separately.

- $k/b^d \leq 1$ : We compute an upper bound on the work as follows:

$$\begin{aligned} O\left(\left(\frac{n}{b^i}\right)^d \log^w \frac{n}{b^i}\right) &= O\left(\frac{n^d}{b^{id}} \cdot (\log^w n - \log^w b^i)\right) \\ &= b^{-id} \cdot O(n^d \cdot (\log^w n - \log^w b^i)) \\ &= b^{-id} \cdot O(n^d \log^w n) \end{aligned}$$

since  $n^d \cdot (\log^w n - \log^w b^i) \leq n^d \log^w n$  for  $b > 1$ , and we only need an upper bound for the asymptotics. Then the work  $T_i$  at level  $i$  is

$$\begin{aligned} T_i &= \frac{k^i}{b^{id}} \cdot O(n^d \log^w n) \\ &= \left(\frac{k}{b^d}\right)^i \cdot O(n^d \log^w n) \end{aligned}$$

Summing over all the levels, we get a total work

$$\begin{aligned} T &= \sum_{i=0}^{\log_b n} T_i \\ &= O(n^d \log^w n) \cdot \sum_{i=0}^{\log_b n} \left(\frac{k}{b^d}\right)^i \\ &= O(n^d \log^w n) \cdot \sum_{i=0}^{\log_b n} r^i \\ &= O(n^d \log^w n \cdot \sum_{i=0}^{\log_b n} r^i) \end{aligned}$$

where  $r = k/b^d$ . When  $r < 1$ , the initial term of the summation dominates as before, so we have

$$\begin{aligned} T &= O(n^d \log^w n \cdot (1 + r + r^2 + \dots + r^{\log_b n})) \\ &= O(n^d \log^w n) \end{aligned}$$

When  $r = 1$ , the terms all have equal size, also as before. Then

$$\begin{aligned} T &= O(n^d \log^w n \cdot (1 + r + r^2 + \dots + r^{\log_b n})) \\ &= O(n^d \log^w n \cdot (1 + \log_b n)) \\ &= O(n^d \log^w n \log_b n) \\ &= O(n^d \log^w n \log n) \\ &= O(n^d \log^{w+1} n) \end{aligned}$$

As before, we drop the lower order term, as well as the base in  $\log_b n$  since it only differs by a constant factor from  $\log n$ .

- $k/b^d > 1$ : We start by observing that since  $\log^w n$  is subpolynomial in  $n$ , it grows slower than any polynomial in  $n$ . We have

$$\frac{k}{b^d} > 1$$

or equivalently

$$\begin{aligned} \log_b k &> d \\ 0 &< \log_b k - d \end{aligned}$$

We choose a small exponent  $\epsilon$  such that

$$0 < \epsilon < \log_b k - d$$

For instance, we can choose  $\epsilon = (\log_b k - d)/2$ . Then

$$\begin{aligned} \log_b k &> d + \epsilon \\ k &> b^{d+\epsilon} \\ \frac{k}{b^{d+\epsilon}} &> 1 \end{aligned}$$

We also have

$$\log^w n \leq Cn^\epsilon$$

for a sufficiently large constant  $C$  and  $n \geq 1$ . For a subproblem of size  $n/b^i$ , we now have work

$$\begin{aligned} O\left(\left(\frac{n}{b^i}\right)^d \log^w \frac{n}{b^i}\right) &= O\left(\left(\frac{n}{b^i}\right)^d \cdot C\left(\frac{n}{b^i}\right)^\epsilon\right) \\ &= O(C\left(\frac{n}{b^i}\right)^{d+\epsilon}) \end{aligned}$$

Then our work  $T_i$  at level  $i$  is

$$T_i = k^i \cdot O(C\left(\frac{n}{b^i}\right)^{d+\epsilon})$$

The total work  $T$  is

$$\begin{aligned}
 T &= \sum_{i=0}^{\log_b n} (k^i \cdot O(C(\frac{n}{b^i})^{d+\epsilon})) \\
 &= O\left(\sum_{i=0}^{\log_b n} (k^i \cdot C(\frac{n}{b^i})^{d+\epsilon})\right) \\
 &= O(Cn^{d+\epsilon} \cdot \sum_{i=0}^{\log_b n} (k^i \cdot (b^{-i})^{d+\epsilon})) \\
 &= O(Cn^{d+\epsilon} \cdot \sum_{i=0}^{\log_b n} (k^i \cdot (b^{d+\epsilon})^{-i})) \\
 &= O(Cn^{d+\epsilon} \cdot \sum_{i=0}^{\log_b n} (\frac{k}{b^{d+\epsilon}})^i)
 \end{aligned}$$

Since  $k/b^{d+\epsilon} > 1$ , the last term in the summation dominates as in [Proof 317](#), so we get

$$\begin{aligned}
 T &= O(Cn^{d+\epsilon} \cdot \sum_{i=0}^{\log_b n} (\frac{k}{b^{d+\epsilon}})^i) \\
 &= O(Cn^{d+\epsilon} \cdot (\frac{k}{b^{d+\epsilon}})^{\log_b n}) \\
 &= O(Cn^{d+\epsilon} \cdot k^{\log_b n} \cdot (b^{-(d+\epsilon)})^{\log_b n}) \\
 &= O(Cn^{d+\epsilon} \cdot k^{\log_b n} \cdot (b^{\log_b n})^{-(d+\epsilon)}) \\
 &= O(Cn^{d+\epsilon} \cdot k^{\log_b n} \cdot n^{-(d+\epsilon)}) \\
 &= O(Ck^{\log_b n}) \\
 &= O(Cn^{\log_b k})
 \end{aligned}$$

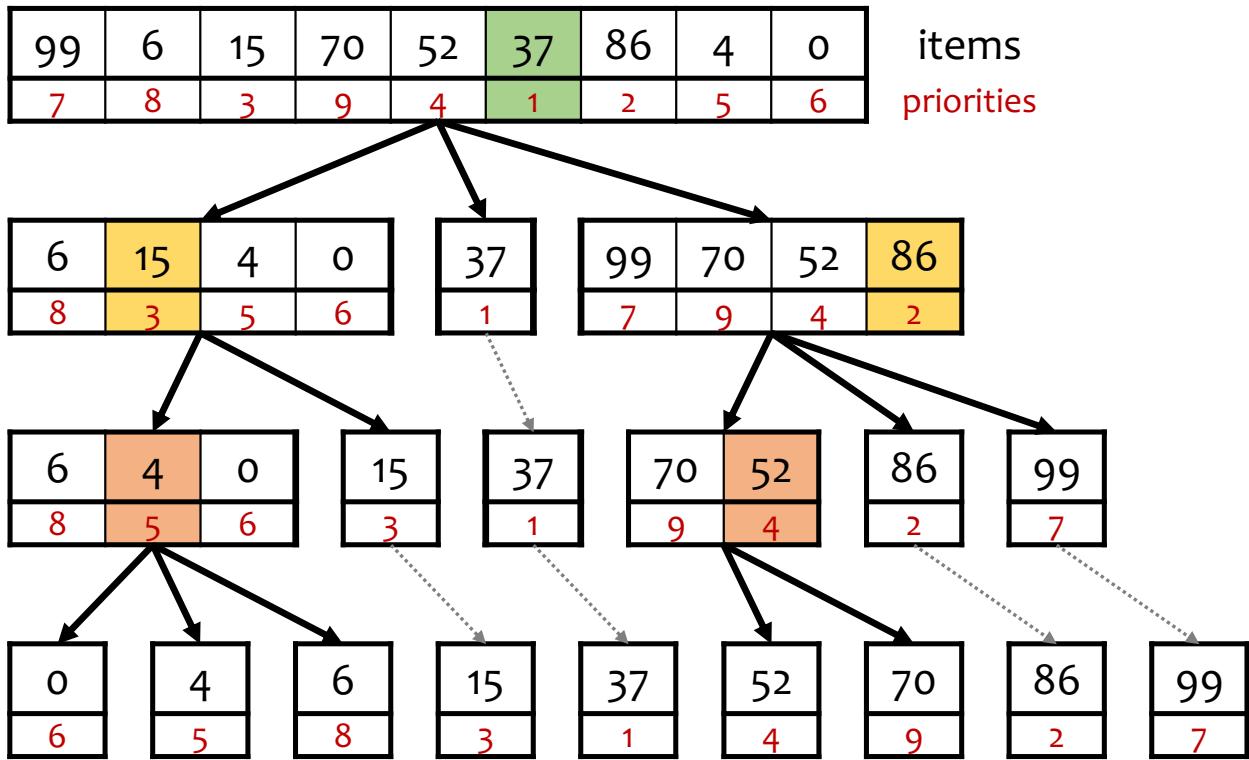
In the last step, we applied the observation that  $k^{\log_b n} = n^{\log_b k}$ . Folding the constant  $C$  into the big-O notation, we end up with

$$\begin{aligned}
 T &= O(Ck^{\log_b n}) \\
 &= O(n^{\log_b k})
 \end{aligned}$$

as required.

## 29.2 Alternative Analysis of Quick Sort

We can conduct an alternative analysis of the [quick-sort algorithm](#) (page 219) by modifying the algorithm so that all the randomness is fixed at the beginning. Given  $n$  elements, we start by choosing a permutation of the numbers  $\{1, 2, \dots, n\}$  uniformly at random to be the *priorities* of the elements. Then when choosing a pivot for a subset of the elements, we pick the element that has the lowest priority over that subset. The following illustrates an execution of this modified algorithm:



In the first step, the element 37 has the lowest priority, so it is chosen as the pivot. At the second level, the element 15 has the minimal priority in the first recursive call, while the element 86 has the lowest priority in the second call. At the third level of the recursion, the element 4 has lowest priority over its subset, and similarly the element 52 for the subset {70, 52}.

The full definition of the modified algorithm is as follows:

**Algorithm 319 (Modified Quick Sort)**

```

function MODIFIEDQUICKSORT( $A[1, \dots, n]$ )
   $P$  = a uniformly random permutation of  $\{1, \dots, n\}$ 
  return PRIORITIZEDSORT( $A, P$ )
function PRIORITIZEDSORT( $A[1, \dots, n], P[1, \dots, n]$ )
  if  $n = 1$  then return  $A$ 
   $p$  = index of the smallest element in  $P$ 
   $(L, PL, R, PR) = \text{PARTITION}(A, P, p)$ 
  return PRIORITIZEDSORT( $L, PL$ ) +  $A[p]$  + PRIORITIZEDSORT( $R, PR$ )
function PARTITION( $A[1, \dots, n], P[1, \dots, n], p$ )
  initialize empty arrays  $L, PL, R, PR$ 
  for  $i = 1$  to  $n$  do
    if  $i \neq p$  and  $A[i] < A[p]$  then
       $L = L + A[i]$ 
       $PL = PL + P[i]$ 
    else if  $i \neq p$  then
       $R = R + A[i]$ 
       $PR = PR + P[i]$ 
  return  $(L, PL, R, PR)$ 

```

This modified algorithm matches the behavior of the original randomized quick sort in two important ways:

1. In both algorithms, when a pivot is chosen among a subset consisting of  $m$  elements, each element is chosen with a uniform probability  $1/m$ . This is explicitly the case in the original version. In the modified version, the priorities assigned to each of the  $m$  elements are unique, and for any set of  $m$  priorities, each element receives the lowest priority in exactly  $1/m$  of the permutations of those priorities.
2. The two algorithms compare elements in exactly the same situation – when partitioning the array, each element is compared to the pivot. (The modified version also compares priorities to find the minimal, but we can just ignore that in our analysis. Even if we did consider them, they just increase the number of comparisons by a constant factor of two.)

Thus, the modified algorithm models the execution of the original algorithm in both the evolution of the recursion as well as the element comparisons. We proceed to analyze the modified algorithm, and the result will equally apply to the original.

For simplicity, we assume for the purposes of analysis that the initial array does not contain any duplicate elements. We have defined our two versions of quick sort to be *stable*, meaning that they preserve the relative ordering of duplicate elements. Thus, even when duplicates are present, the algorithm distinguishes between them, resulting in the same behavior as if there were no duplicates.

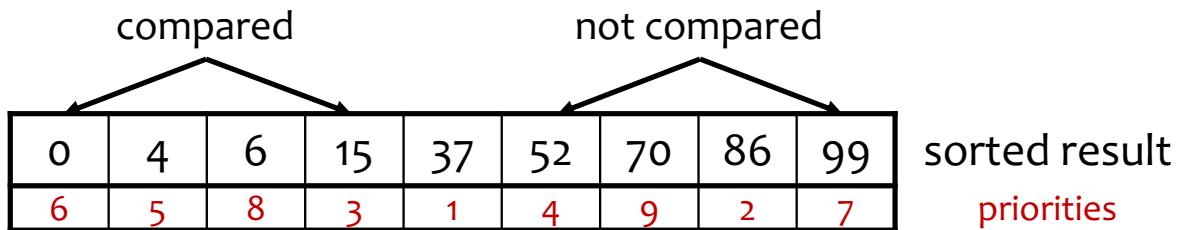
**Claim 320** Let  $A$  be an array of  $n$  elements, and let  $S$  be the result of sorting  $A$ . Let  $\text{priority}(S[i])$  be the priority that was assigned to the element  $S[i]$  in the modified quick-sort algorithm. Then  $S[i]$  and  $S[j]$  were compared by the algorithm if and only if  $i \neq j$ , and one of the following holds:

$$\text{priority}(S[i]) = \min_{i \leq k \leq j} \{\text{priority}(S[k])\}, \text{ or}$$

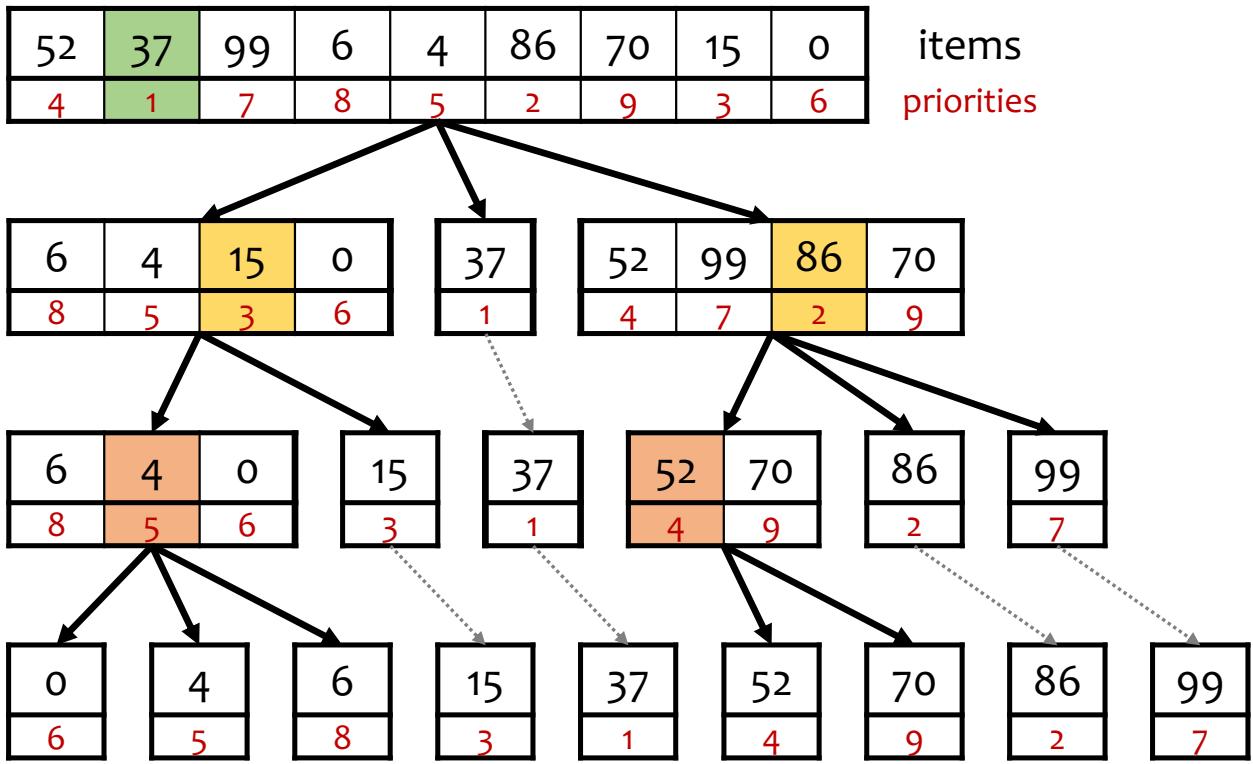
$$\text{priority}(S[j]) = \min_{i \leq k \leq j} \{\text{priority}(S[k])\}$$

In other words,  $S[i]$  and  $S[j]$  are compared exactly when the lowest priority among the elements  $S[i], S[i + 1], \dots, S[j - 1], S[j]$  is that of either  $S[i]$  or  $S[j]$ .

The claim above holds regardless of the original order of the elements in  $A$ . For example, take a look at the result from our illustration of the modified algorithm above:



As stated above, two elements are compared only in the partitioning step, and one of the two must be the pivot. Referring to the execution above, we can see that 0 and 15 were compared in the second level of the recursion, whereas 52 and 99 were never compared to each other. Now consider a different initial ordering of the elements, but with the same priorities assigned to each element:



The execution of the algorithm is essentially the same! It still picks 37 as the first pivot, since it has the lowest priority, and forms the same partitions; the only possible difference is the ordering of a partition. Subsequent steps again pick the same pivots, since they are finding the minimal priority over the same subset of elements and associated priorities. Thus, given just the sorted set of elements and their priorities, we can determine which elements were compared as stated in [Claim 320](#), regardless of how they were initially ordered. We proceed to prove [Claim 320](#):

**Proof 321** First, we show that if the lowest priority among the elements  $S[i], S[i+1], \dots, S[j-1], S[j]$  is that of either  $S[i]$  or  $S[j]$ , then  $S[i]$  and  $S[j]$  are compared. Without loss of generality, suppose  $S[i]$  is the element with the lowest priority in this subset. This implies that no element in  $S[i+1], \dots, S[j]$  will be picked as a pivot prior to  $S[i]$  being picked. Thus, all pivots chosen by the algorithm before  $S[i]$  must be either less than  $S[i]$  or greater than  $S[j]$ . For each such pivot, the elements  $S[i], S[i+1], \dots, S[j]$  are all placed in the same partition – they are in sorted order, and if the pivot is less than  $S[i]$ , all of these elements are larger than the pivot, while if the pivot is greater than  $S[j]$ , then all these elements are smaller than the pivot. Then when the algorithm chooses  $S[i]$  as the pivot,  $S[j]$  is in the same subset of the array as  $S[i]$  and thus will be compared to  $S[i]$  in the partitioning step. The same reasoning holds for when  $S[j]$  has the lowest priority – it will eventually be chosen as the pivot, at which point  $S[i]$  will be in the same subset and will be compared to  $S[j]$ .

Next, we show that if the lowest priority among the elements  $S[i], S[i+1], \dots, S[j-1], S[j]$  is neither that of  $S[i]$  nor of  $S[j]$ , then  $S[i]$  and  $S[j]$  are never compared. As long as the algorithm chooses pivots outside of the set  $S[i], S[i+1], \dots, S[j]$ , the two elements  $S[i]$  and  $S[j]$  remain in the same subset of the array without having been compared to each other – they remain in the same subset since the pivots so far have been either less than both  $S[i]$  and  $S[j]$  or greater than both of them, and they have not been compared yet since neither has been chosen as a pivot. When the algorithm first chooses a pivot from among the elements  $S[i], S[i+1], \dots, S[j-1], S[j]$ , the pivot it chooses must be one of  $S[i+1], \dots, S[j-1]$ , since one of those elements has lower priority than both  $S[i]$  and  $S[j]$ . In the partitioning step for that pivot,  $S[i]$  and  $S[j]$  are placed into separate partitions, since  $S[i]$  is smaller than the pivot while  $S[j]$  is larger than the pivot. Since  $S[i]$  and  $S[j]$  end up in separate partitions, they cannot be compared in subsequent steps of the algorithm.  $\square$

We can now proceed to compute the expected number of comparisons. Let  $X_{ij}$  be an indicator random variable such that  $X_{ij} = 1$  if  $S[i]$  and  $S[j]$  were compared at some point in the algorithm,  $X_{ij} = 0$  otherwise. From [Claim 320](#), we

can conclude that

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}$$

This is because each of the  $j - i + 1$  elements in  $S[i], S[i+1], \dots, S[j-1], S[j]$  has equal chance of having the lowest priority among those elements, and  $S[i]$  and  $S[j]$  are compared when one of those two elements has lowest priority among this set.

We also observe that a pair of elements is compared at most once – they are compared once in the partitioning step if they are in the same subarray and one of the two is chosen as a pivot, and after the partitioning step is over, the pivot is never compared to anything else. Thus, the total number of comparisons  $X$  is just the number of pairs that are compared, out of the  $n(n - 1)/2$  distinct pairs:

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

The rest of the analysis is as *before* (page 219), where we concluded that  $\mathbb{E}[X] \leq 2n \ln n = O(n \log n)$ . Since the modified quick-sort algorithm models the behavior of the original randomized algorithm, this result also applies to the latter.

## 29.3 Proof of the Simplified Multiplicative Chernoff Bounds

We previously showed (page 274) that the unsimplified Chernoff bounds

$$\begin{aligned}\Pr[X \geq (1 + \delta)\mu] &\leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu \quad \text{for } \delta > 0 \\ \Pr[X \leq (1 - \delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1 - \delta)^{1-\delta}}\right)^\mu \quad \text{for } 0 < \delta < 1\end{aligned}$$

hold. We now demonstrate that the simplified bounds

$$\begin{aligned}\Pr[X \geq (1 + \delta)\mu] &\leq e^{-\frac{\delta^2 \mu}{2+\delta}} \quad \text{for } \delta > 0 \\ \Pr[X \leq (1 - \delta)\mu] &\leq e^{-\frac{\delta^2 \mu}{2}} \quad \text{for } 0 < \delta < 1\end{aligned}$$

follow from the unsimplified bounds.

**Proof 322** We first consider the upper-tail Chernoff bound. For  $\delta > 0$ , we have

$$\begin{aligned}\Pr[X \geq (1 + \delta)\mu] &\leq \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu \\ &= \left(\frac{e^\delta}{(e^{\ln(1+\delta)})^{1+\delta}}\right)^\mu \\ &= \left(\frac{e^\delta}{e^{(1+\delta)\ln(1+\delta)}}\right)^\mu \\ &= (e^{\delta - (1+\delta)\ln(1+\delta)})^\mu\end{aligned}$$

From a list of logarithmic identities<sup>115</sup>, we obtain the inequality

$$\ln(1 + x) \geq \frac{2x}{2 + x}$$

for  $x > 0$ . This gives us

$$\begin{aligned}-\ln(1+x) &\leq -\frac{2x}{2+x} \\ e^{-\ln(1+x)} &\leq e^{-\frac{2x}{2+x}}\end{aligned}$$

Applying this to our Chernoff-bound expression with  $x = \delta$ , we get

$$\begin{aligned}\Pr[X \geq (1+\delta)\mu] &\leq (e^{\delta-(1+\delta)\ln(1+\delta)})^\mu \\ &\leq (e^{\delta-(1+\delta)\frac{2\delta}{2+\delta}})^\mu \\ &= (e^{\frac{\delta(2+\delta)-2\delta(1+\delta)}{2+\delta}})^\mu \\ &= (e^{\frac{2\delta+\delta^2-2\delta-2\delta^2}{2+\delta}})^\mu \\ &= (e^{\frac{-\delta^2}{2+\delta}})^\mu \\ &= e^{-\frac{\delta^2\mu}{2+\delta}}\end{aligned}$$

resulting in the simplified upper-tail bound.

For the lower-tail Chernoff bound, we have for  $0 < \delta < 1$ :

$$\begin{aligned}\Pr[X \leq (1-\delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^\mu \\ &= \left(\frac{e^\delta}{(e^{\ln(1-\delta)})^{1-\delta}}\right)^\mu \\ &= \left(\frac{e^{-\delta}}{e^{(1-\delta)\ln(1-\delta)}}\right)^\mu \\ &= (e^{-\delta-(1-\delta)\ln(1-\delta)})^\mu\end{aligned}$$

The Taylor series for the natural logarithm<sup>116</sup> gives us:

$$\begin{aligned}\ln(1-x) &= -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \dots \\ &\geq -x - \frac{x^2}{2} \quad \text{for } 0 < x < 1 \\ -\ln(1-x) &\leq x + \frac{x^2}{2}\end{aligned}$$

The second step follows since  $\frac{x^3}{3} + \frac{x^4}{4} + \dots$  is positive for  $x > 0$ . Applying the result to our Chernoff-bound expression with  $x = \delta$ , we obtain

$$\begin{aligned}\Pr[X \leq (1-\delta)\mu] &\leq (e^{-\delta-(1-\delta)\ln(1-\delta)})^\mu \\ &\leq (e^{-\delta+(1-\delta)(\delta+\frac{\delta^2}{2})})^\mu \\ &= (e^{-\delta+\delta-\delta^2+\frac{\delta^2}{2}-\frac{\delta^3}{2}})^\mu \\ &= (e^{-\frac{\delta^2}{2}-\frac{\delta^3}{2}})^\mu\end{aligned}$$

Since  $\delta > 0$ , we have  $\delta^3/2 > 0$ , and

$$-\frac{\delta^2}{2} - \frac{\delta^3}{2} \leq -\frac{\delta^2}{2}$$

Thus,

$$\begin{aligned}\Pr[X \leq (1 - \delta)\mu] &\leq (e^{-\frac{\delta^2}{2} - \frac{\delta^3}{2}})^\mu \\ &\leq (e^{-\frac{\delta^2}{2}})^\mu \\ &= e^{-\frac{\delta^2 \mu}{2}}\end{aligned}$$

completing our proof of the simplified lower-tail bound.  $\square$

<sup>115</sup> [https://en.wikipedia.org/wiki/List\\_of\\_logarithmic\\_identities#Inequalities](https://en.wikipedia.org/wiki/List_of_logarithmic_identities#Inequalities)

<sup>116</sup> [https://en.wikipedia.org/wiki/Mercator\\_series](https://en.wikipedia.org/wiki/Mercator_series)

## 29.4 Proof of the Upper-Tail Hoeffding's Inequality

The upper-tail Hoeffding's inequality we use (Theorem 259) states that for a sequence of independent indicator random variables  $X_1, \dots, X_n$ ,

$$\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] \leq e^{-2\varepsilon^2 n}$$

where  $X = X_1 + \dots + X_n$ ,  $\mathbb{E}[X_i] = p_i$ , and  $\mathbb{E}\left[\frac{1}{n}X\right] = \frac{1}{n} \sum_i p_i = p$ . We proceed to prove a more general variant of this bound:

**Theorem 323** *Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent random variables such that  $X_i \in [0, 1]$ , and the  $X_i$  have expectation  $\mathbb{E}[X_i] = p_i$ , respectively. Let*

$$p = \mathbb{E}\left[\frac{1}{n}X\right] = \frac{1}{n} \sum_i p_i$$

*Let  $\varepsilon > 0$  be a deviation from the expectation. Then*

$$\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] \leq \exp(-2\varepsilon^2 n)$$

*(Note that  $\exp(x)$  is alternate notation for  $e^x$ .)*

Here, we do not require that the  $X_i$  are indicators, just that they are in the range  $[0, 1]$ .

We proceed to prove Theorem 323. We have

$$\begin{aligned}\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] &= \Pr[X \geq n(p + \varepsilon)] \\ &= \Pr[e^{sX} \geq \exp(sn(p + \varepsilon))]\end{aligned}$$

The latter step holds for all  $s \geq 0$ . The process of working with the random variable  $e^{sX}$  rather than  $X$  directly is the *Chernoff bounding technique*; the idea is that small deviations in  $X$  turn into large deviations in  $e^{sX}$ , so that Markov's inequality (Theorem 227) produces a more useful result. We will choose an appropriate value of  $s$  later. For any value

of  $s$ ,  $e^{sX}$  is nonnegative, so we can apply Markov's inequality to obtain:

$$\begin{aligned}\Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] &\leq \mathbb{E}[e^{sX}] / \exp(sn(p + \varepsilon)) \\ &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[e^{sX}] \\ &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[\exp(s(X_1 + X_2 + \dots + X_n))] \\ &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[e^{sX_1} e^{sX_2} \dots e^{sX_n}] \\ &= \exp(-sn(p + \varepsilon)) \prod_i \mathbb{E}[e^{sX_i}]\end{aligned}$$

In the last step, we applied the fact that the  $X_i$  are independent to conclude that the random variables  $e^{sX_i}$  are also independent, and therefore by Lemma 250, the expectation of their product is the product of their expectations.

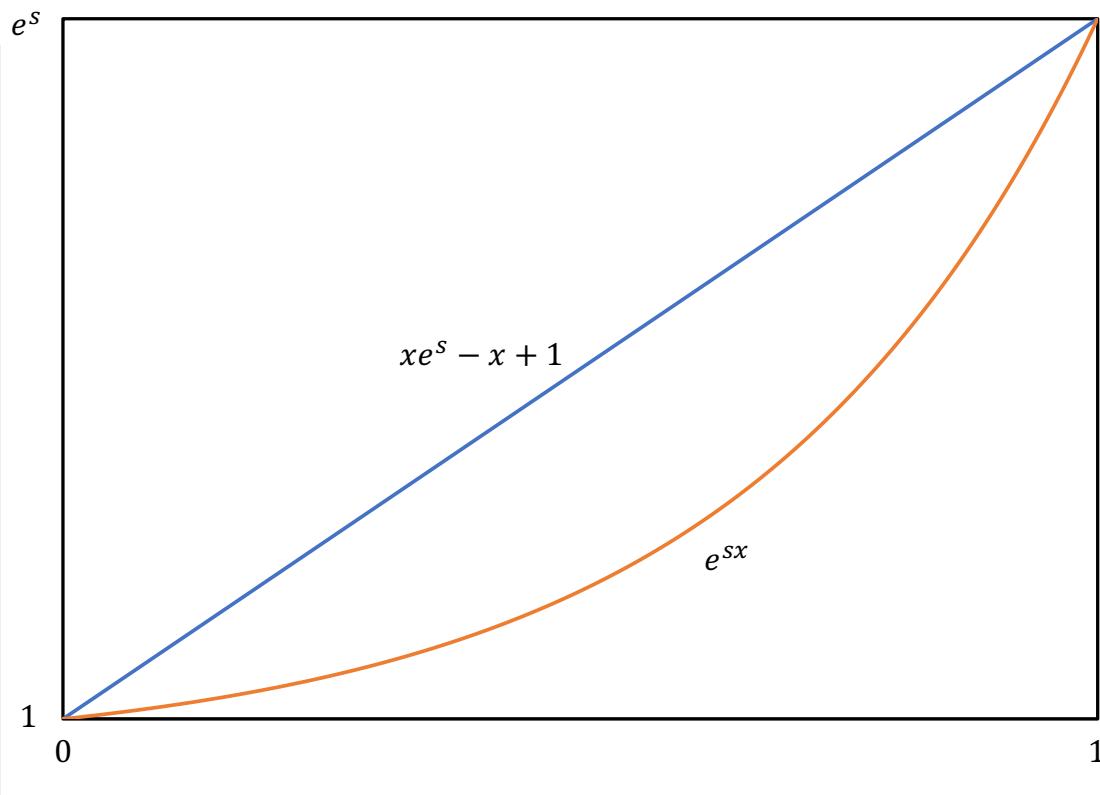
We now need to establish a bound on  $\mathbb{E}[e^{sX_i}]$ .

**Lemma 324** *Let  $X$  be a random variable such that  $X \in [0, 1]$  and  $\mathbb{E}[X] = p$ . Then*

$$\mathbb{E}[e^{sX}] \leq \exp(sp + s^2/8)$$

for all  $s \in \mathbb{R}$ .

**Proof 325** We first observe that because  $e^{sx}$  is a convex function<sup>117</sup>, we can bound it from above on the interval  $[0, 1]$  by the line that passes through the endpoints 1 and  $e^s$ :



This line has slope  $e^s - 1$  and y-intercept 1, giving us  $xe^s - x + 1$ . Thus,

$$e^{sx} \leq xe^s - x + 1$$

in the interval  $0 \leq x \leq 1$ . Then

$$\begin{aligned}\mathbb{E}[e^{sX}] &\leq \mathbb{E}[Xe^s - X + 1] \\ &= (e^s - 1)\mathbb{E}[X] + 1 \\ &= pe^s - p + 1\end{aligned}$$

where the second step follows from linearity of expectation. We now have an upper bound on  $\mathbb{E}[e^{sX}]$ , but it is not a convenient one – we would like it to be of the form  $e^\alpha$ , so that we can better use it in proving [Theorem 323](#), for which we've already obtained a bound that contains an exponential. Using the fact that  $z = e^{\ln z}$ , we get

$$\begin{aligned}\mathbb{E}[e^{sX}] &\leq pe^s - p + 1 \\ &= \exp(\ln(pe^s - p + 1))\end{aligned}$$

To further bound this expression, let

$$\phi(s) = \ln(pe^s - p + 1)$$

Then finding an upper bound for  $\phi(s)$  will in turn allow us to bound  $e^{\phi(s)}$ . By the Lagrange form of Taylor's theorem<sup>[118](#)</sup>, we get

$$\phi(s) = \phi(0) + s\phi'(0) + \frac{1}{2}s^2\phi''(v)$$

for some  $v \in [0, s]$ . To differentiate  $\phi(s)$ , we let  $f(s) = \ln s$  and  $g(s) = pe^s - p + 1$ , so that  $\phi(s) = (f \circ g)(s)$ . Then by the chain rule<sup>[119](#)</sup>, we have<sup>[120](#)</sup>

$$\begin{aligned}\phi'(s) &= (f \circ g)'(s) \\ &= f'(g(s)) \cdot g'(s) \\ &= \frac{1}{g(s)} \cdot g'(s) \quad (\text{since } f'(s) = \frac{d}{ds} \ln s = \frac{1}{s}) \\ &= \frac{1}{pe^s - p + 1} \cdot pe^s \\ &= \frac{pe^s}{pe^s - p + 1}\end{aligned}$$

To compute  $\phi''(s)$ , we now let  $\hat{f}(s) = pe^s$ , so that  $\phi'(s) = \hat{f}(s)/g(s)$ . Then by the quotient rule<sup>[121](#)</sup>, we get

$$\begin{aligned}\phi''(s) &= \frac{\hat{f}'(s)g(s) - \hat{f}(s)g'(s)}{g^2(s)} \\ &= \frac{pe^s(pe^s - p + 1) - pe^s pe^s}{(pe^s - p + 1)^2} \\ &= \frac{pe^s}{pe^s - p + 1} \cdot \frac{(pe^s - p + 1) - pe^s}{pe^s - p + 1} \\ &= \frac{pe^s}{pe^s - p + 1} \left(1 - \frac{pe^s}{pe^s - p + 1}\right) \\ &= t(1 - t)\end{aligned}$$

where  $t = \frac{pe^s}{pe^s - p + 1}$ . Observe that  $t(1 - t)$  is a parabola with a maximum at  $t = \frac{1}{2}$ :

$$\begin{aligned} h(t) &= t(1 - t) = t - t^2 \\ h'(t) &= 1 - 2t \\ h''(t) &= -2 \end{aligned}$$

Setting  $h'(t) = 1 - 2t = 0$ , we get that  $t = \frac{1}{2}$  is an extremum, and since the second derivative  $h''(\frac{1}{2})$  is negative, it is a maximum. Thus, we have

$$\begin{aligned} \phi''(s) &= t(1 - t) \\ &\leq \frac{1}{2}(1 - \frac{1}{2}) \\ &= \frac{1}{4} \end{aligned}$$

We can now plug everything into our result from applying Taylor's theorem:

$$\begin{aligned} \phi(s) &= \phi(0) + s\phi'(0) + \frac{1}{2}s^2\phi''(v) \\ &= \ln(p - p + 1) + s\frac{p}{p - p + 1} + \frac{1}{2}s^2\phi''(v) \\ &= sp + \frac{1}{2}s^2\phi''(v) \\ &\leq sp + \frac{1}{2}s^2 \cdot \frac{1}{4} \\ &= sp + \frac{s^2}{8} \end{aligned}$$

Thus,

$$\begin{aligned} \mathbb{E}[e^{sX}] &\leq e^{\phi(s)} \\ &\leq \exp(sp + s^2/8) \end{aligned}$$

as claimed. □

<sup>117</sup> [https://en.wikipedia.org/wiki/Convex\\_function](https://en.wikipedia.org/wiki/Convex_function)

<sup>118</sup> [https://en.wikipedia.org/wiki/Taylor%27s\\_theorem](https://en.wikipedia.org/wiki/Taylor%27s_theorem)

<sup>119</sup> [https://en.wikipedia.org/wiki/Chain\\_rule](https://en.wikipedia.org/wiki/Chain_rule)

<sup>120</sup> The function  $g(s) = pe^s - p + 1$  does not have any real roots for  $p \in [0, 1]$ , so it is safe to divide by it.

<sup>121</sup> [https://en.wikipedia.org/wiki/Quotient\\_rule](https://en.wikipedia.org/wiki/Quotient_rule)

Continuing our proof of Theorem 323, we have

$$\Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] \leq \exp(-sn(p + \varepsilon)) \prod_i \mathbb{E}[e^{sX_i}]$$

Applying Lemma 324, we get

$$\begin{aligned} \Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] &\leq \exp(-sn(p + \varepsilon)) \prod_i \mathbb{E}[e^{sX_i}] \\ &\leq \exp(-sn(p + \varepsilon)) \prod_i \exp(sp_i + s^2/8) \\ &= \exp(-sn(p + \varepsilon)) \cdot \exp\left(\sum_i (sp_i + s^2/8)\right) \\ &= \exp(-sn(p + \varepsilon)) \cdot \exp(snp + s^2n/8) \\ &= \exp(-sn\varepsilon + s^2n/8) \end{aligned}$$

We now choose  $s$  to minimize the exponent  $r(s) = -sn\varepsilon + s^2n/8$ . We have:

$$\begin{aligned} r(s) &= -sn\varepsilon + \frac{n}{8}s^2 \\ r'(s) &= -n\varepsilon + \frac{n}{8} \cdot 2s = -n\varepsilon + \frac{n}{4}s \\ r''(s) &= \frac{n}{4} \end{aligned}$$

We have another parabola, and since the second derivative is positive, we obtain a minimum for  $r(s)$  at

$$\begin{aligned} r'(s) &= -n\varepsilon + \frac{n}{4}s = 0 \\ s &= 4\varepsilon \end{aligned}$$

Then

$$\begin{aligned} r(4\varepsilon) &= -4n\varepsilon^2 + \frac{n}{8} \cdot 16\varepsilon^2 \\ &= -4n\varepsilon^2 + 2n\varepsilon^2 \\ &= -2n\varepsilon^2 \end{aligned}$$

Thus,

$$\begin{aligned} \Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] &\leq \Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] \\ &\leq \exp(-2n\varepsilon^2) \end{aligned}$$

completing our proof of Theorem 323.

## 29.5 General Case of Hoeffding's Inequality

We can further generalize Theorem 323 to the case where the individual random variables  $X_i$  are in the range  $[a_i, b_i]$ . We start by generalizing Lemma 324 to obtain Hoeffding's lemma<sup>122</sup>.

**Lemma 326 (Hoeffding's Lemma)** *Let  $X$  be a random variable such that  $X \in [a, b]$ , where  $a < b$ , and  $\mathbb{E}[X] = p$ . Then*

$$\mathbb{E}[e^{sX}] \leq \exp(sp + s^2(a - b)^2/8)$$

for all  $s \in \mathbb{R}$ .

**Proof 327** Let  $X' = \frac{X-a}{b-a}$ . Then  $X'$  is a random variable such that  $X' \in [0, 1]$ , and

$$\mathbb{E}[X'] = \mathbb{E}\left[\frac{X-a}{b-a}\right] = \frac{p-a}{b-a}$$

by linearity of expectation. Let  $p' = \frac{p-a}{b-a}$ . Applying Lemma 324 to  $X'$ , we get

$$\mathbb{E}[e^{s'X'}] \leq \exp(s'p' + s'^2/8)$$

<sup>122</sup> [https://en.wikipedia.org/wiki/Hoeffding%27s\\_lemma](https://en.wikipedia.org/wiki/Hoeffding%27s_lemma)

Now consider  $e^{sX}$ . We have  $X = a + (b - a)X'$ , so

$$\begin{aligned}\mathbb{E}[e^{sX}] &= \mathbb{E}[\exp(sa + s(b - a)X')] \\ &= e^{sa} \mathbb{E}[\exp(s(b - a)X')]\end{aligned}$$

Let  $s' = s(b - a)$ . From the result above, we have

$$\begin{aligned}e^{sa} \mathbb{E}[\exp(s(b - a)X')] &= e^{sa} \mathbb{E}[e^{s'X'}] \\ &\leq e^{sa} \exp(s'p' + s'^2/8) \\ &= \exp(sa + s'p' + s'^2/8)\end{aligned}$$

Substituting  $p' = \frac{p-a}{b-a}$  and  $s' = s(b - a)$ , we obtain the following for the exponent:

$$\begin{aligned}sa + s'p' + \frac{1}{8}s'^2 &= sa + s(b - a)\frac{p - a}{b - a} + \frac{1}{8}s^2(b - a)^2 \\ &= sa + s(p - a) + \frac{1}{8}s^2(b - a)^2 \\ &= sp + \frac{1}{8}s^2(b - a)^2\end{aligned}$$

Thus,

$$\begin{aligned}\mathbb{E}[e^{s'X'}] &\leq \exp(sa + s'p' + s'^2/8) \\ &= \exp(sp + s^2(b - a)^2/8)\end{aligned}$$

as claimed. □

We can now derive the general case of Hoeffding's inequality<sup>123</sup>.

**Theorem 328 (Hoeffding's Inequality – General Case)** Let  $X = X_1 + \dots + X_n$ , where the  $X_i$  are independent random variables such that  $X_i \in [a_i, b_i]$  for  $a_i < b_i$  and  $\mathbb{E}[X_i] = p_i$ . Let

$$p = \mathbb{E}\left[\frac{1}{n}X\right] = \frac{1}{n} \sum_i p_i$$

Let  $\varepsilon > 0$  be a deviation from the expectation. Then

$$\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] \leq \exp\left(-\frac{2n^2\varepsilon^2}{\sum_i (b_i - a_i)^2}\right)$$

<sup>123</sup> [https://en.wikipedia.org/wiki/Hoeffding%27s\\_inequality](https://en.wikipedia.org/wiki/Hoeffding%27s_inequality)

**Proof 329** We proceed as in the proof of Theorem 323 to obtain

$$\begin{aligned}
 \Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] &= \Pr[X \geq n(p + \varepsilon)] \\
 &= \Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] \\
 &\leq \mathbb{E}[e^{sX}] / \exp(sn(p + \varepsilon)) \quad (\text{Markov's inequality}) \\
 &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[e^{sX}] \\
 &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[\exp(s(X_1 + X_2 + \dots + X_n))] \\
 &= \exp(-sn(p + \varepsilon)) \cdot \mathbb{E}[e^{sX_1} e^{sX_2} \dots e^{sX_n}] \\
 &= \exp(-sn(p + \varepsilon)) \prod_i \mathbb{E}[e^{sX_i}] \quad (\text{independence of the } X_i)
 \end{aligned}$$

Applying Hoeffding's lemma (Lemma 326), we obtain

$$\begin{aligned}
 \exp(-sn(p + \varepsilon)) \prod_i \mathbb{E}[e^{sX_i}] &\leq \exp(-sn(p + \varepsilon)) \prod_i \exp(sp_i + s^2(b_i - a_i)^2/8) \\
 &= \exp(-sn(p + \varepsilon)) \cdot \exp\left(\sum_i (sp_i + s^2(b_i - a_i)^2/8)\right) \\
 &= \exp(-sn(p + \varepsilon)) \cdot \exp\left(snp + \frac{s^2}{8} \sum_i (b_i - a_i)^2\right) \\
 &= \exp\left(-sn\varepsilon + \frac{s^2}{8} \sum_i (b_i - a_i)^2\right)
 \end{aligned}$$

We choose  $s$  to minimize the exponent  $r(s)$ :

$$\begin{aligned}
 r(s) &= -sn\varepsilon + \frac{s^2}{8} \sum_i (b_i - a_i)^2 \\
 r'(s) &= -n\varepsilon + \frac{s}{4} \sum_i (b_i - a_i)^2 \\
 r''(s) &= \frac{1}{4} \sum_i (b_i - a_i)^2
 \end{aligned}$$

Since  $a_i < b_i$ , we have  $b_i - a_i > 0$  for all  $i$ , so that  $r''(s)$  is positive. Thus, we obtain a minimum for  $r(s)$  when  $r'(s) = 0$ :

$$\begin{aligned}
 r'(s) &= -n\varepsilon + \frac{s}{4} \sum_i (b_i - a_i)^2 = 0 \\
 s &= \frac{4n\varepsilon}{\sum_i (b_i - a_i)^2}
 \end{aligned}$$

Then

$$\begin{aligned}
 r\left(\frac{4n\varepsilon}{\sum_i (b_i - a_i)^2}\right) &= -n\varepsilon \frac{4n\varepsilon}{\sum_i (b_i - a_i)^2} + \frac{16n^2\varepsilon^2}{(\sum_i (b_i - a_i)^2)^2} \cdot \frac{1}{8} \sum_i (b_i - a_i)^2 \\
 &= -\frac{4n^2\varepsilon^2}{\sum_i (b_i - a_i)^2} + \frac{2n^2\varepsilon^2}{\sum_i (b_i - a_i)^2} \\
 &= -\frac{2n^2\varepsilon^2}{\sum_i (b_i - a_i)^2}
 \end{aligned}$$

Thus,

$$\begin{aligned}\Pr\left[\frac{1}{n}X \geq p + \varepsilon\right] &\leq \Pr[e^{sX} \geq \exp(sn(p + \varepsilon))] \\ &\leq \exp\left(-\frac{2n^2\varepsilon^2}{\sum_i(b_i - a_i)^2}\right)\end{aligned}$$

completing our proof of the general case of Hoeffding's inequality. □

## **Part VIII**

### **About**

---

**CHAPTER  
THIRTY**

---

**ABOUT**

This text was originally written for EECS 376, the Foundations of Computer Science course at the University of Michigan, by [Amir Kamil](#)<sup>124</sup> in Fall 2020. This is version 0.5 of the text.

- *Contributions:* [Chris Peikert](#)<sup>125</sup>, [Thatchaphol Saranurak](#)<sup>126</sup>

This text is licensed under the [Creative Commons Attribution-ShareAlike 4.0 International license](#)<sup>127</sup>.

Please report bugs and other issues [here](#)<sup>128</sup>.

---

<sup>124</sup> <https://amirkamil.com>

<sup>125</sup> <https://web.eecs.umich.edu/~cpeikert/>

<sup>126</sup> <https://sites.google.com/site/thsaranurak/>

<sup>127</sup> <https://creativecommons.org/licenses/by-sa/4.0/>

<sup>128</sup> <https://github.com/eecs376/issues/issues>

# INDEX

## Symbols

"no larger than"  
  injective, 92  
0-1 knapsack problem, *see* Knapsack problem  
0-1 random variable, *see* Indicator random variable  
3CNF, 169, 175  
  exact, 214  
3SAT, 169

## A

Abstract data type, 222  
Abstraction, 3  
Accept, 50, 63, 85  
Accept state, 53  
Acceptance problem for TMs, 102  
Adversary, 214  
AKS primality test, 271  
Algorithms as games, 214  
All-pairs shortest path, 3  
  Floyd-Warshall algorithm, 40  
All-pairs shortest paths, 38  
Alpha approximation, *see* Approximation  
Alphabet, 48  
  input alphabet, 62  
  tape alphabet, 62  
Alternation, 124  
  dovetailing, 128  
Amplification, 228, 282, 287  
  one-sided, 282  
  two-sided, 287  
Approximation, 192, 201  
  approximation ratio, 192  
  benchmark, 195  
  in expectation, 214  
   $\alpha$ -approximation, 192  
   $\alpha$ -approximation algorithm, 192  
Asymmetric encryption, 256  
Authentication, 242, 260  
Automata, 50  
  Chomsky hierarchy, 51  
  finite automata, 50, 51, 53  
  linear-bounded automata, 51

pushdown automata, 51  
state machine, 50  
Turing machine, 51, 61  
Automaton, *see* Automata  
Averaging argument, 210

B

Baby-step giant-step, 255  
Backtracking, *see* Dynamic programming, 30  
Barber paradox, 100  
Benchmark, *see* Approximation, 195  
Big data, 236  
Binary search tree, 222  
Binomial distribution, 234, 279  
Black box, 106  
Boolean formula, 149, 201  
  3CNF, 169, 175  
  3SAT (*language*), 169  
  assignment, 149  
  clause, 169, 175  
  CNF, 169, 175  
  conjunctive normal form, 169, 175  
  conversion to 3CNF, 170  
  equivalence to circuit, 160  
  exact 3CNF, 214  
  literal, 149  
  MAX-E3SAT, 214  
  operator, 149  
  SAT (*language*), 150  
  satisfiability, 149, 150  
  satisfying assignment, 149, 175  
  variable, 149  
BPP (*complexity class*), 261, 283  
BQP (*complexity class*), 261

## C

Cardinality, 92, 93  
  countable, 93  
  countably infinite, 93  
  diagonalization, 97  
  of the integers  $\mathbb{Z}$ , 93  
  of the pairs of naturals  $\mathbb{N} \times \mathbb{N}$ , 94

- of the positive rationals  $\mathbb{Q}^+$ , 94
- of the rationals  $\mathbb{Q}$ , 95
- of the reals  $\mathbb{R}$ , 95
- of the set of finite binary strings  $\{0,1\}^*$ , 97
- of the set of languages, 97
- of the set of machines, 97
- uncountable, 95
- Carmichael number, 271, 272
- Cell, *see* Turing machine, 61
- Certificate, 143, 261
- Chain rule, 304
- Chebyshev's inequality, 229, 232
- Chernoff bounding technique, 275, 302
- Chernoff bounds, 229, 274, 280
  - Chernoff-Hoeffding bounds, 234
  - multiplicative, 229, 274
  - proof, 275
  - proof of simplified, 300
- Chernoff-Hoeffding bounds, *see* Hoeffding's inequality
- Chomsky hierarchy, 51
- Church-Turing thesis, 84
  - extended, 139, 261
- Ciphertext, 242
- Circuit satisfiability, 160
- Classical computer, 261
- Clause, 169, 175
- Clique, 174, 181
  - CLIQUE (*language*), 188
  - search algorithm, 189
- CLIQUE (*language*), 174, 181
- Closed, 89
  - decidable languages, 89
  - recognizable languages, 124
  - two-sided error (*BPP*), 283
- Closest-pair problem, 19
  - 1D algorithm, 20
  - 2D algorithm, 24
  - $\delta$ -strip, 22
- CNF, *see* Conjunctive normal form
- Co-recognizable, 126
- Code, 100
- Combined-greedy algorithm, *see* Knapsack problem, 200
- Complement, 47
- Complete graph, 44, 141, 181
- Completeness, 134
- Complexity
  - space, 4, 137
  - time, 4, 137
- Complexity class, 137
  - BPP, 261, 283
  - BQP, 261
- coNP, 286
- coRP, 282
- DTIME( $t(n)$ ), 137
- NP, 145
- NPI, 261
- P, 139
- R, 137
- RE, 137
- RP, 282
- ZPP, 284
- Compression, 268
  - by program, 268
  - lossless, 268
  - lossy, 268
- Computable function, 267
- Computational security, 242, 261
- Computational step, 62
- Concentration bounds, 229, 234, 274
  - Chernoff bounds, 229, 274
  - Hoeffding's inequality, 234
- Conditional probability, 216
- Conditional security, 242
- Confidence level, 236
- Confidentiality, 242, 260
- Configuration, 116, 152
- Congruence, 243
- Conjunctive normal form, *see* Boolean formula, 169, 175
  - conversion to 3CNF, 170
- Connected, 41
  - minimally connected, 41
- coNP, 286
- Context-free language, 51
- Context-sensitive language, 51
- Convex function, 303
- Cook-Levin theorem, 151, 152, 165
  - 3SAT version, 172
- Coprime, 4, 246, 256, 258
- coRP (*complexity class*), 282
- Countable, 93
- Countably infinite, 93
- Cover
  - set, 182
  - vertex, 177
- Crossing edge
  - maximum cut, 196
  - vertex cover, 181
- Cryptography, 242
  - asymmetric, 256
  - ciphertext, 242
  - Diffie-Hellman, 253, 254, 256, 260
  - Diffie-Hellman assumption, 254, 259
  - discrete logarithm assumption, 255, 259
  - ElGamal, 260

- factorization hardness assumption, 259  
 Kerckhoff's principle, 242, 254  
 key, 242  
 key exchange, 253, 256  
 one-time pad, 248  
 padding, 249, 261  
 plaintext, 242  
 post-quantum, 261  
 private key, 253, 256, 258, 260  
 public key, 253, 256, 258, 260  
 public-key cryptosystem, 256  
 RSA, 256  
 RSA assumption, 259  
 RSA encryption, 258  
 RSA signature, 260  
 spoofing, 261  
 statistical attack, 250  
 symmetric, 253, 256  
*Cut*, *see* Maximum cut, 196  
*Cycle*, 41
- D**
- De Morgan's laws, 149, 161  
 Decidable, 85  
     efficiently decidable, 139  
 Decide, 50, 85  
 Decider, 85  
 Decision problem, 4, 47  
 Degree, 194  
 Delta strip, *see* Closest-pair problem  
 Derandomized, 216  
 Deterministic, 203  
 Deterministic finite automata, *see* Finite automata  
 DFA, *see* Finite automata  
 Diagonalization, 97  
 Diffie-Hellman assumption, 254, 259  
 Diffie-Hellman key exchange, 253, 254, 256, 260  
 Discrete logarithm, 255, 260, 261  
     baby-step giant-step, 255  
     Shor's algorithm, 261  
 Discrete logarithm assumption, 255, 259  
 Divide and conquer, 13  
 Double-cover algorithm, *see* Vertex cover, 194  
 Dovetailing, 128  
 DTIME, 137  
 Dynamic programming, 25  
     backtracking, 30  
     bottom-up table, 26  
     top-down memoized, 26  
     top-down recursive, 25
- E**
- Efficiently decidable, 139
- Efficiently verifiable, 143  
 ElGamal encryption, 260  
 Euclid's algorithm, 5  
     extended, 246  
     potential function, 10  
     time complexity, 11  
 Euclid's division lemma, 243  
 Euclid's lemma, 257, 273  
 Euler's totient function, 254  
 Event, 204  
 Exact 3CNF, 214  
 Exact problem, 188  
 Expectation, 209  
     conditional, 216  
     linearity of expectation, 212  
 Expected value, 209  
 Extended Church-Turing thesis, 139, 261  
 Extended Euclidean algorithm, 246  
 Extended Fermat's little theorem, 271
- F**
- Factorization hardness assumption, 259  
 False negative, 282  
 False positive, 282  
 Fan-out, 160  
 Fast modular exponentiation, 245  
     bottom-up, 246  
     top-down, 245  
 Fermat primality test, 259, 271  
 Fermat's little theorem, 256  
     extended, 271  
     proof, 257  
 Final state, *see* Turing machine, 62  
 Finite acceptor, *see* Finite automata  
 Finite automata, 50, 51, 53  
     formal definition, 57  
     language of, 58  
 Finite-state automata, *see* Finite automata  
 Finite-state machine, *see* Finite automata  
 Flipping game, 7  
 Floyd-Warshall algorithm, 3, 40  
 Frequency table, 250  
 Fully connected, 141  
 Function, 92  
     partial, 92  
     total, 92  
 Functional problem, 188, 267  
     equivalence with decision, 267  
 Fundamental theorem of arithmetic, 272
- G**
- Gadget, 178  
 GCD, *see* Greatest common divisor  
 Generator, 253

Geometric distribution, 270, 285  
 Greatest common divisor, 4  
   Euclid's algorithm, 5  
 Greedy algorithm, 41

**H**

Halt, 62  
 Halting problem, 104  
 Hamiltonian cycle, 184  
   HAMCYCLE (*language*), 185  
 Hamiltonian path, 184  
   HAMPATH (*language*), 185  
 Hard-code, 109  
 Head, *see* Turing machine, 61  
 Heuristics, 201  
 Hoeffding's inequality, 229, 234, 237, 239, 287  
   general case, 306, 307  
   Hoeffding's lemma, 306  
   proof, 302  
 Hoeffding's lemma, 306

**I**

i.i.d., *see* Independent identically distributed  
 In place, 219  
 Incomplete, 135  
 Independent identically distributed, 233  
 Independent random variables, 207  
 Independent set, 181  
   INDEPENDENT-SET (*language*), 181  
 Indicator random variable, 207  
 Induced subgraph, 174  
 Information-theoretic security, 242  
 Initial state, *see* Turing machine, 62  
 Injective, 92  
 Integer factorization, 259, 261  
   Shor's algorithm, 261  
 Integer multiplication, 16  
   Karatsuba algorithm, 18  
 Integrity, 242, 260  
 Intermediate vertex, 39  
 Interpreter, 102  
 Inverse, 246

**J**

Job scheduling, 238  
 Joint probability, 205

**K**

Karatsuba algorithm, 18  
 Karp reduction, 162  
 Kerckhoff's principle, 242, 254  
 Key, 242  
   private, 253, 256, 258, 260

public, 253, 256, 258, 260  
 Key exchange, 253, 256  
 Kleene star, 49  
 Knapsack problem, 199, 201  
   combined-greedy algorithm, 200  
   fractional knapsack problem, 201  
   relatively greedy algorithm, 199  
   single-greedy algorithm, 200  
 Kolmogorov complexity, 268  
 Kruskal's algorithm, *see* Minimum spanning tree, 42  
   proof, 44

**L**

L (*subscripted language*)  
 $\{\varepsilon\}$ , 130  
 3, 113  
 A376, 265  
 ACC, 102  
 DEC, 266  
 EQ, 111  
 EVEN, 113  
 FOO, 110  
 HALT, 104  
 HATES-EVENS, 129  
 L376, 266  
 NO-FOO, 126  
 NO-FOO-BAR, 127  
 QTILE, 115  
 R376, 266  
 REJ, 129, 266  
 SmallTM, 266  
 TILE, 115  
 WritesOne, 133  
 $\Sigma^*$ , 265  
 $\varepsilon$ -ACC, 113  
 $\varepsilon$ -HALT, 108  
 $\emptyset$ , 111  
 Language, 4, 47, 49  
   context-free, 51  
   context-sensitive, 51  
   of a machine, 58, 85  
   recursively-enumerable, 51  
   regular, 51  
 Las Vegas algorithm, 284  
 Law of large numbers, 229  
 LCS, *see* Longest common subsequence  
 Library, 106  
 Limited budget, 142, 174, 178, 188, 196  
 Linear-bounded automata, 51  
 Linearity of expectation, 212  
 LIS, *see* Longest increasing subsequence  
 Literal, *see* Boolean formula, 149  
 Load balancing, 238  
 Local-search algorithm, *see* Maximum cut, 197

Logic gate, 160  
 LONG-PATH (*language*), 186  
 Longest common subsequence, 33  
 Longest increasing subsequence, 31  
 Loop, 63, 85  
 Lossless compression, 268  
 Lossy compression, 268

**M**

M. C. Escher, 113  
 Mapping reduction  
     polynomial-time, 162  
 Margin of error, 236  
 Markov's inequality, 211, 275, 302  
     reverse Markov's inequality, 211  
 Master theorem, 14  
     non-master-theorem recurrences, 263  
     proof, 291  
     substitution, 263  
     with log factors, 15  
 Max-E3SAT, 214  
     derandomized algorithm, 216  
 Maximally acyclic, 41  
 Maximization problem, 188, 192  
 Maximum clique, *see* Clique  
 Maximum cut, 196, 201  
     local-search algorithm, 197  
     MAXCUT (*language*), 196  
     randomized algorithm, 218  
 MAZE, 145  
 Memoization, *see* Dynamic programming, 26  
 Merge sort, 13, 219  
 Method of conditional probabilities, 216  
 Miller-Rabin primality test, 272  
 Minimally connected, 41  
 Minimization problem, 188, 192  
 Minimum spanning tree, 41  
     definition, 41  
     Kruskal's algorithm, 42  
         proof of Kruskal's algorithm, 44  
 Minimum vertex cover, *see* Vertex cover  
 Modular arithmetic, 243  
     congruence, 243  
     modulus, 243  
     reduction, 243  
 Modular exponentiation, 245, 254, 259, 272  
 Modular inverse, 246  
 Modulus, 243  
 Monte Carlo algorithm, 228, 284  
 Monte Carlo method, 228  
 MST, *see* Minimum spanning tree  
 Multiplication, *see* Integer multiplication

**N**

NP (*complexity class*), 145  
     relationship with P, 147  
 NP-Complete, 165, 261  
 NP-Hard, 164  
 NP-Intermediate, 261  
 NPI (*complexity class*), 261

**O**

One-sided-error randomized algorithm, 282  
     amplification, 282  
 One-time pad, 248  
     reuse, 250  
 One-to-one, 92  
 Optimal substructure, *see* Dynamic programming, 25  
 Optimization problem, 188, 192  
 Oracle, 106  
 Overlapping subproblems, *see* Dynamic programming, 25

**P**

P (*complexity class*), 139  
     relationship with NP, 147  
 Padding, 249, 261  
 Pair-wise disjoint edges, 195  
 PALINDROME, 137  
 Partial function, 92  
 Partition, 196  
 Penrose tiling, 114  
 Pi (*estimating its value*), 228  
 Pivot, 219  
 Plaintext, 242  
 Planar graph, 201  
 Polling, 236  
     confidence level, 236  
     margin of error, 236  
     sampling theorem, 238, 240  
     with Chernoff bounds, 280  
     with Hoeffding's inequality, 237  
 Poly-time reduction, *see* Polynomial-time reduction  
 Polynomial composition, 138  
 Polynomial hierarchy, 287  
 Polynomial identity testing, 274  
 Polynomial-time mapping reduction, 162  
 Polynomial-time reduction, 161  
 Post-quantum cryptography, 261  
 Potential function, 9  
     for Euclid's algorithm, 10  
 Potential method, 9  
 Predicate, 47  
 Primality testing, 259, 270  
     AKS primality test, 271

Carmichael number, 271, 272  
 Fermat primality test, 259, 271  
 Miller-Rabin primality test, 272  
 PRIMES (*language*), 270  
 pseudoprime, 272  
 PSEUDOPRIMES (*language*), 272  
 Prime factorization, 259, 272  
 Prime number theorem, 270  
 Principle of optimality, 25  
 Privacy, 242, 260  
 Private key, 253, 256, 258, 260  
 Probabilistic, 203  
 Probability
 

- Chebyshev's inequality, 229, 232
- Chernoff bounds, 229, 274
- conditional, 216
- conditional expectation, 216
- event, 204
- expectation, 209
- expected value, 209
- Hoeffding's inequality, 234
- independent identically distributed, 233
- independent random variables, 207
- indicator random variable, 207
- joint probability, 205
- law of large numbers, 229
- linearity of expectation, 212
- outcome, 204
- probability distribution, 208
- probability space, 204
- random variable, 207
- sample space, 204
- sampling theorem, 238, 240
- Standard deviation, 230
- union bound, 206, 238
- Variance, 229

 Probability space, 204  
 Program analysis, 133  
 Property, 131  
 Pseudocode, 3, 267  
 Pseudopolynomial time, 199  
 Pseudoprime, 272  
 Public key, 253, 256, 258, 260  
 Public-key cryptosystem, 256  
 Pumping lemma, 60  
 Pushdown automata, 51

## Q

Quadrant, 115, 228  
 Quantum computer, 261  
 Quantum Turing machine, 261  
 Quick sort, 219, 296
 

- analysis, 220, 296

 Quotient rule, 304

## R

R (*complexity class*), 137  
 Random variable, 207  
 Randomized, 203  
 Randomness, 203  
 RE (*complexity class*), 137  
 Recognizable, 123  
 Recognize, 123  
 Recursively-enumerable language, *see* RE (*complexity class*), *see* Recognizable, 51  
 Reduction, 106
 

- modular arithmetic, 243
- polynomial-time, 161
- Turing, 106

 Regular language, 51  
 Reject, 50, 63, 85  
 Relatively greedy algorithm, *see* Knapsack problem, 199  
 Relatively prime, 272  
 Reverse Markov's inequality, 211  
 Rice's theorem, 132
 

- program analysis, 133

 Rock-paper-scissors, 203  
 RP (*complexity class*), 282  
 RSA, 256
 

- ecryption, 258

 RSA assumption, 259  
 RSA signature, 260

## S

Sample space, 204  
 Sampling theorem, 238, 240  
 SAT, *see* Boolean formula, 150, 188  
 Satisfiability, *see* Boolean formula, 149, 150
 

- circuit, 160

 Search problem, 188  
 Self-hosting, 101  
 Semantic property, 131
 

- trivial, 131

 Set cover, 182
 

- SET-COVER (*language*), 182

 Shor's algorithm, 261  
 Shortest path, *see* All-pairs shortest path, 3  
 Shortest paths, 38  
 Signature, 260  
 Simple path, 39  
 Simulate, 102  
 Single-cover algorithm, *see* Vertex cover, 193  
 Single-greedy algorithm, *see* Knapsack problem, 200  
 Skip list, 223  
 Sorting algorithm
 

- in place, 219
- merge sort, 13, 219

- quick sort, 219, 296  
 stable, 298  
**S**oundness, 134  
**S**pace complexity, 4, 137  
**S**panning tree, *see* Minimum spanning tree  
**S**poofing, 261  
**S**tandard deviation, 230  
**S**tate, *see* Finite automata, 50, 53, *see also* Turing machine, 61  
 final, 62  
 initial, 62  
**S**tate diagram, 64  
**S**tate machine, 50  
**S**tatistical attack, 250  
**S**tring, 48  
**S**ubroutine, 102  
**S**ubsequence, 33  
**S**ubstitution, 263  
**S**ymmetric encryption, 253, 256
- T**ableau, *see* Cook-Levin theorem, 152  
**T**ape, *see* Turing machine, 61  
**T**aylor's theorem, 304  
**T**iling, 113  
 non-periodic, 114  
 Penrose tiling, 114  
 Wang tile, 115  
**T**ime complexity, 4, 137  
**T**otal function, 92  
**T**ransition, 50, 53  
**T**ransition function, 57, *see* Turing machine, 62  
**T**raveling salesperson problem, 3, 141, 201  
 definition, 141  
 limited-budget version, 142  
 TSP (*language*), 146, 186  
**T**ree, 41  
 definitions, 41  
 minimum spanning tree, 41  
**T**rivial semantic property, 131  
**T**SP, *see* Traveling salesperson problem  
**T**uring completeness, 91  
**T**uring machine, 4, 51, 61  
 equivalence of models, 89  
 formal definition, 61  
 language of, 85  
 quantum, 261  
 transition function, 62  
 two-tape model, 89  
 universal, 102  
**T**uring reduction, 106  
**T**wo-sided-error randomized algorithm, 261, 283  
 amplification, 287  
**T**ype, 47
- Type system, 135
- U**
- Uncompressible, 268  
**U**ncomputable, 268  
**U**nconditional security, 242  
**U**ncountable, 95  
**U**ndecidable, 99  
**U**nion bound, 206, 238  
**U**niversal gate set, 160  
**U**niversal Turing machine, 102  
 language of, 102  
**U**nrecognizable, 99, 123  
**U**nsound, 135
- V**
- Variance, 229  
**V**erifiable, 140  
 efficiently verifiable, 143  
 verifier, 143  
**V**erifier, 143  
**V**ertex cover, 177, 188  
 approximation, 193  
 double-cover algorithm, 194  
 greedy algorithm, 194  
 search algorithm, 190  
 single-cover algorithm, 193  
 VERTEX-COVER (*language*), 178
- W**
- Wang tile, 115  
**W**eighted Task Selection, 28  
**W**indow, *see* Cook-Levin theorem, 156  
**W**TS, *see* Weighted Task Selection
- Z**
- ZPP (*complexity class*), 284