math is hard, but you don't have to do it alone!
Please send comments and corrections to Anil Ada (aada@cs.cmu.edu).
Foreword

These notes are based on the lectures given by Anil Ada and Ariel Procaccia for the Fall 2017 edition of the course 15-251 “Great Ideas in Theoretical Computer Science” at Carnegie Mellon University. They are also closely related to the previous editions of the course, and in particular, lectures prepared by Ryan O’Donnell.

WARNING: The purpose of these notes is to complement the lectures. These notes do not contain full explanations of all the material covered during lectures. In particular, the intuition and motivation behind many concepts and proofs are explained during the lectures and not in these notes.

There are various versions of the notes that omit certain parts of the notes. Go to the course webpage to access all the available versions.

In the main version of the notes (i.e. the main document), each chapter has a preamble containing the chapter structure and the learning goals. The preamble may also contain some links to concrete applications of the topics being covered. At the end of each chapter, you will find a short quiz for you to complete before coming to recitation, as well as hints to selected exercise problems.

Note that some of the exercise solutions are given in full detail, whereas for others, we give all the main ideas, but not all the details. We hope the distinction will be clear.
Acknowledgements

The course 15-251 was created by Steven Rudich many years ago, and we thank him for creating this awesome course. Here is the webpage of an early version of the course:
http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15251-s04/Site/.
Since then, the course has evolved. The webpage of the current version is here:
http://www.cs.cmu.edu/~15251/.

Thanks to the previous instructors of 15-251, who have contributed a lot to the development of the course: Victor Adamchik, Luis von Ahn, Anupam Gupta, Venkatesan Guruswami, Bernhard Haeupler, John Lafferty, Ryan O’Donnell, Ariel Procaccia, Daniel Sleator and Klaus Sutner.

Thanks to Eric Bae, Seth Cobb, Teddy Ding, Ellen Kim, Aditya Krishnan, Xinran Liu, Matthew Salim, Ticha Sethapakdi, Vanessa Siriwalothakul, Natasha Vasthare, Jenny Wang, Ling Xu, Ming Yang, Stephanie You, Xingjian Yu and Nancy Zhang for sending valuable comments and corrections on an earlier draft of the notes. And thanks to Rosie Sun for sending valuable comments and corrections on the current draft.
## Contents

1 Strings and Encodings 1
   1.1 Alphabets and Strings 3
   1.2 Languages 5
   1.3 Encodings 7
   1.4 Computational Problems and Decision Problems 9

2 Deterministic Finite Automata 13
   2.1 Basic Definitions 15
   2.2 Irregular Languages 18
   2.3 Closure Properties of Regular Languages 19

3 Turing Machines 25
   3.1 Basic Definitions 27
   3.2 Decidable Languages 32

4 Countable and Uncountable Sets 37
   4.1 Basic Definitions 39
   4.2 Countable Sets 40
   4.3 Uncountable Sets 42

5 Undecidable Languages 47
   5.1 Existence of Undecidable Languages 49
   5.2 Examples of Undecidable Languages 49
   5.3 Undecidability Proofs by Reductions 52

6 Time Complexity 57
   6.1 Big-O, Big-Omega and Theta 59
   6.2 Worst-Case Running Time of Algorithms 60
   6.3 Complexity of Algorithms with Integer Inputs 62

7 The Science of Cutting Cake 67
   7.1 The Problem and the Model 69
   7.2 Cake Cutting Algorithms in the Robertson-Webb Model 70

8 Introduction to Graph Theory 77
   8.1 Basic Definitions 79
   8.2 Graph Algorithms 83
      8.2.1 Graph searching algorithms 83
      8.2.2 Minimum spanning tree 84
      8.2.3 Topological sorting 86
Chapter 1

Strings and Encodings
PREAMBLE

Chapter structure:

- Section 1.1 (Alphabets and Strings)
  - Definition 1.1 (Alphabet, symbol/character)
  - Definition 1.5 (String/word, empty string)
  - Definition 1.9 (Length of a string)
  - Definition 1.11 (Star operation on alphabets)
  - Definition 1.16 (Reversal of a string)
  - Definition 1.20 (Concatenation of strings)
  - Definition 1.24 (Powers of a string)
  - Definition 1.27 (Substring)

- Section 1.2 (Languages)
  - Definition 1.29 (Language)
  - Definition 1.37 (Reversal of a language)
  - Definition 1.39 (Concatenation of languages)
  - Definition 1.41 (Powers of a language)
  - Definition 1.45 (Star operation on a language)

- Section 1.3 (Encodings)
  - Definition 1.51 (Encoding of a set)

- Section 1.4 (Computational Problems and Decision Problems)
  - Definition 1.61 (Computational problem)
  - Definition 1.64 (Decision problem)

Chapter goals:

In the beginning, our goal is to build up, completely formally/mathematically, the important notions related to computation and algorithms. Our starting point is this chapter, which deals with how to formally represent data and how to formally define the concept of a computational problem.

In theoretical computer science, every kind of data is represented/encoded using finite-length strings. In this chapter, we introduce you to the formal definitions related to strings and encodings of objects with strings. We also present the definitions of “computational problem” and “decision problem”.

All the definitions in this chapter are at the foundation of the formal study of computation.
1.1 Alphabets and Strings

Definition 1.1 (Alphabet, symbol/character).
An alphabet is a non-empty, finite set, and is usually denoted by $\Sigma$. The elements of $\Sigma$ are called symbols or characters.

Example 1.2 (Unary alphabet).
A unary alphabet consists of one symbol. A common choice for that symbol is 1. So an example of a unary alphabet is $\Sigma = \{1\}$.

Example 1.3 (Binary alphabet).
A binary alphabet consists of two symbols. Often we represent those symbols using 0 and 1. So an example of a binary alphabet is $\Sigma = \{0, 1\}$. Another example of a binary alphabet is $\Sigma = \{a, b\}$ where a and b are the symbols.

Example 1.4 (Ternary alphabet).
A ternary alphabet consists of three symbols. So $\Sigma = \{0, 1, 2\}$ and $\Sigma = \{a, b, c\}$ are examples of ternary alphabets.

Definition 1.5 (String/word, empty string).
Given an alphabet $\Sigma$, a string (or word) over $\Sigma$ is a (possibly infinite) sequence of symbols, written as $a_1a_2a_3\ldots$, where each $a_i \in \Sigma$. The string with no symbols is called the empty string and is denoted by $\epsilon$.

Example 1.6 (Strings over the unary alphabet).
For $\Sigma = \{1\}$, the following is a list of 6 strings over $\Sigma$:

$$\epsilon, 1, 11, 111, 1111, 11111.$$  

Furthermore, the infinite sequence $111111\ldots$ is also a string over $\Sigma$.

Example 1.7 (Strings over the binary alphabet).
For $\Sigma = \{0, 1\}$, the following is a list of 8 strings over $\Sigma$:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000.$$  

The infinite strings $000000\ldots, 111111\ldots$ and $010101\ldots$ are also examples of strings over $\Sigma$.

Note 1.8 (Strings and quotation marks).
In our notation of a string, we do not use quotation marks. For instance, we use the notation $1010$ rather than “1010”, even though the latter notation using the quotation marks is the standard one in many programming languages. Occasionally, however, we may use quotation marks to distinguish a string like “1010” from another type of object with the representation 1010 (e.g. the binary number 1010).

Definition 1.9 (Length of a string).
The length of a string $w$, denoted $|w|$, is the number of symbols in $w$. If $w$ has an infinite number of symbols, then the length is undefined.

Example 1.10 (Lengths of 01001 and $\epsilon$).
Let $\Sigma = \{0, 1\}$. The length of the word 01001, denoted by $|01001|$, is equal to 5. The length of $\epsilon$ is 0.
Definition 1.11 (Star operation on alphabets). Let $\Sigma$ be an alphabet. We denote by $\Sigma^*$ the set of all strings over $\Sigma$ consisting of finitely many symbols. Equivalently, using set notation,

$$\Sigma^* = \{a_1a_2 \ldots a_n : n \in \mathbb{N}, \text{ and } a_i \in \Sigma \text{ for all } i\}.$$

Example 1.12 ($\{a\}^*$). For $\Sigma = \{a\}$, $\Sigma^*$ denotes the set of all finite-length words consisting of $a$’s. So

$$\{a\}^* = \{\epsilon, a, aa, aaaa, aaaaa, \ldots\}.$$

Example 1.13 ($\{0,1\}^*$). For $\Sigma = \{0,1\}$, $\Sigma^*$ denotes the set of all finite-length words consisting of 0’s and 1’s. So

$$\{0,1\}^* = \{\epsilon, 0, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}.$$

Note 1.14 (Finite vs infinite strings). We often use the words “string” and “word” to refer to a finite-length string/word. When we want to talk about infinite-length strings, we explicitly use the word “infinite”.

Note 1.15 (Size of $\Sigma^*$). By Definition 1.1 (Alphabet, symbol/character), an alphabet $\Sigma$ cannot be the empty set. This implies that $\Sigma^*$ is an infinite set since there are infinitely many strings of finite length over a non-empty $\Sigma$.

Definition 1.16 (Reversal of a string). For a string $w = a_1a_2 \ldots a_n$, the reversal of $w$, denoted $w^R$, is the string $w^R = a_na_{n-1} \ldots a_1$.

Example 1.17 (Reversal of 01001). The reversal of 01001 is 10010.

Example 1.18 (Reversal of 1). The reversal of 1 is 1.

Example 1.19 (Reversal of $\epsilon$). The reversal of $\epsilon$ is $\epsilon$.

Definition 1.20 (Concatenation of strings). If $u$ and $v$ are two strings in $\Sigma^*$, the concatenation of $u$ and $v$, denoted by $uv$ or $u \cdot v$, is the string obtained by joining together $u$ and $v$.

Example 1.21 (Concatenation of 101 and 001). If $u = 101$ and $v = 001$, then $uv = 101001$.

Example 1.22 (Concatenation of 101 and $\epsilon$). If $u = 101$ and $v = \epsilon$, then $uv = 101$.

Example 1.23 (Concatenation of $\epsilon$ and $\epsilon$). If $u = \epsilon$ and $v = \epsilon$, then $uv = \epsilon$. 

4
Definition 1.24 (Powers of a string).
For a word \( u \in \Sigma^* \) and \( n \in \mathbb{N} \), the \( n \)'th power of \( u \), denoted by \( u^n \), is the word obtained by concatenating \( u \) with itself \( n \) times.

Example 1.25 (Third power of 101).
If \( u = 101 \) then \( u^3 = 101101101 \).

Example 1.26 (Zeroth power of a string).
For any string \( u \), \( u^0 = \epsilon \).

Definition 1.27 (Substring).
We say that a string \( u \) is a substring of string \( w \) if \( w = xuy \) for some strings \( x \) and \( y \).

Example 1.28 (101 as a substring).
The string 101 is a substring of 11011 and also a substring of 0101. On the other hand, it is not a substring of 1001.

1.2 Languages

Definition 1.29 (Language).
Any (possibly infinite) subset \( L \subseteq \Sigma^* \) is called a language over the alphabet \( \Sigma \).

Example 1.30 (Language of even length strings).
Let \( \Sigma \) be an alphabet. Then \( L = \{ w \in \Sigma^*: |w| \text{ is even} \} \) is a language.

Example 1.31 (A language with one word).
Let \( \Sigma = \{0, 1\} \). Then \( L = \{101\} \) is a language.

Example 1.32 (\( \Sigma^* \) as a language).
Let \( \Sigma \) be an alphabet. Then \( L = \Sigma^* \) is a language.

Example 1.33 (Empty set as a language).
Let \( \Sigma \) be an alphabet. Then \( L = \emptyset \) is a language.

Note 1.34 (Size of a language).
Since a language is a set, the size of a language refers to the size of that set. A language can have finite or infinite size. This is not in conflict with the fact that every language consists of finite-length strings.

Note 1.35 (\( \emptyset \) vs \( \{\epsilon\} \)).
The language \( \{\epsilon\} \) is not the same language as \( \emptyset \). The former has size 1 whereas the latter has size 0.

Exercise 1.36 (Structural induction on words).
Let language \( L \subseteq \{0, 1\}^* \) be recursively defined as follows:

- \( \epsilon \in L \);
- if \( x, y \in L \), then \( 0xy0 \in L \).
Show, using (structural) induction, that for any word $w \in L$, the number of 0’s in $w$ is exactly twice the number of 1’s in $w$.

**Definition 1.37** (Reversal of a language). Given a language $L \subseteq \Sigma^*$, we define its reversal, denoted $L^R$, as the language

$$L^R = \{w^R \in \Sigma^* : w \in L\}.$$ 

**Example 1.38** (Reversal of $\{\epsilon, 1, 1010\}$). The reversal of the language $\{\epsilon, 1, 1010\}$ is $\{\epsilon, 1, 0101\}$.

**Definition 1.39** (Concatenation of languages). Given two languages $L_1, L_2 \subseteq \Sigma^*$, we define their concatenation, denoted $L_1L_2$ or $L_1 \cdot L_2$, as the language

$$L_1L_2 = \{uv \in \Sigma^* : u \in L_1, v \in L_2\}.$$ 

**Example 1.40** (Concatenation of $\{\epsilon, 1\}$ and $\{0, 01\}$). The concatenation of languages $\{\epsilon, 1\}$ and $\{0, 01\}$ is the language $\{0, 01, 10, 101\}$.

**Definition 1.41** (Powers of a language). Given a language $L \subseteq \Sigma^*$ and $n \in \mathbb{N}$, the $n$’th power of $L$, denoted $L^n$, is the language obtained by concatenating $L$ with itself $n$ times, that is,

$$L^n = L \cdot L \cdot L \cdots L,$$

Equivalently,

$$L^n = \{u_1u_2\cdots u_n \in \Sigma^* : u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$ 

**Example 1.42** ($\{1\}^3$). The 3rd power of $\{1\}$ is the language $\{111\}$.

**Example 1.43** ($\{\epsilon, 1\}^3$). The 3rd power of $\{\epsilon, 1\}$ is the language $\{\epsilon, 1, 11, 111\}$.

**Example 1.44** ($L^0$). The 0th power of any language $L$ is the language $\{\epsilon\}$.

**Definition 1.45** (Star operation on a language). Given a language $L \subseteq \Sigma^*$, we define the star of $L$, denoted $L^*$, as the language

$$L^* = \bigcup_{n \in \mathbb{N}} L^n.$$ 

Equivalently,

$$L^* = \{u_1u_2\cdots u_n \in \Sigma^* : n \in \mathbb{N}, u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.$$ 

**Example 1.46** ($\Sigma^*$). Given an alphabet $\Sigma$, consider the language $L = \Sigma \subseteq \Sigma^*$. Then $L^*$ is equal to $\Sigma^*$.

---

[1] We can omit parentheses as the order in which the concatenation $\cdot$ is applied does not matter.
Example 1.47 ($\{00\}^*$).
If $L = \{00\}$, then $L^*$ is the language consisting of all words containing an even number of 0’s and no other symbol.

Example 1.48 ($(\{00\}^*)^*$).
Let $L$ be the language consisting of all words containing an even number of 0’s and no other symbol. Then $L^* = L$.

Exercise 1.49 (Can you distribute star over intersection?).
Prove or disprove: If $L_1, L_2 \subseteq \{a, b\}^*$ are languages, then $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$.

Exercise 1.50 (Can you interchange star and reversal?).
Is it true that for any language $L$, $(L^*)^R = (L^R)^*$? Prove your answer.

1.3 Encodings

Definition 1.51 (Encoding of a set).
Let $A$ be a set (which is possibly countably infinite\(^3\)), and let $\Sigma$ be a alphabet. An encoding of the elements of $A$, using $\Sigma$, is an injective function $Enc : A \rightarrow \Sigma^*$. We denote the encoding of $a \in A$ by $\langle a \rangle$.\(^3\)

If $w \in \Sigma^*$ is such that there is some $a \in A$ with $w = \langle a \rangle$, then we say $w$ is a valid encoding of an element in $A$.

A set that can be encoded is called encodable.\(^4\)

Example 1.52 (Decimal encoding of naturals).
When we (humans) communicate numbers among ourselves, we usually use the base-10 representation, which corresponds to an encoding of $\mathbb{N}$ using the alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$. For example, we encode the number four as 4 and the number twelve as 12.

Example 1.53 (Binary encoding of naturals).
As you know, every number has a base-2 representation (which is also known as the binary representation). This representation corresponds to an encoding of $\mathbb{N}$ using the alphabet $\Sigma = \{0, 1\}$. For example, four is encoded as 100 and twelve is encoded as 1100.

Example 1.54 (Binary encoding of integers).
An integer is a natural number together with a sign, which is either negative or positive. Let $Enc : \mathbb{N} \rightarrow \{0, 1\}^*$ be any binary encoding of $\mathbb{N}$. Then we can extend this encoding to an encoding of $\mathbb{Z}$, by defining $Enc' : \mathbb{Z} \rightarrow \{0, 1\}^*$ as follows:

$$
Enc'(x) = \begin{cases} 
0Enc(x) & \text{if } x \geq 0, \\
1Enc(x) & \text{if } x < 0.
\end{cases}
$$

Effectively, this encoding of integers takes the encoding of natural numbers and precedes it with a bit indicating the integer’s sign.

\(^3\)We assume you know what a countable set is, however, we will review this concept in a future lecture.

\(^3\)Note that this angle-bracket notation does not specify the underlying encoding function as the particular choice of encoding function is often unimportant.

\(^4\)Not every set is encodable. Can you figure out exactly which sets are encodable?
Example 1.55 (Unary encoding of naturals).
It is possible (and straightforward) to encode the natural numbers using the alphabet $\Sigma = \{1\}$ as follows. Let $\text{Enc}(n) = 1^n$ for all $n \in \mathbb{N}$.

Example 1.56 (Ternary encoding of pairs of naturals).
Suppose we want to encode the set $A = \mathbb{N} \times \mathbb{N}$ using the alphabet $\Sigma = \{0, 1, 2\}$. One way to accomplish this is to make use of a binary encoding $\text{Enc'} : \mathbb{N} \to \{0, 1\}^*$ of the natural numbers. With $\text{Enc'}$ in hand, we can define $\text{Enc} : \mathbb{N} \times \mathbb{N} \to \{0, 1, 2\}^*$ as follows. For $(x, y) \in \mathbb{N} \times \mathbb{N}$, $\text{Enc}(x, y) = \text{Enc'}(x)2\text{Enc'}(y)$. Here the symbol 2 acts as a separator between the two numbers. To make the separator symbol advertise itself as such, we usually pick a symbol like # rather than 2. So the ternary alphabet is often chosen to be $\Sigma = \{0, 1, #\}$.

Example 1.57 (Binary encoding of pairs of naturals).
Having a ternary alphabet to encode pairs of naturals was convenient since we could use the third symbol as a separator. It is also relatively straightforward to take that ternary encoding and turn it into a binary encoding, as follows. Encode every element of the ternary alphabet in binary using two bits. For instance, if the ternary alphabet is $\Sigma = \{0, 1, #\}$, then we could encode 0 as 00, 1 as 01 and # as 11. This mapping allows us to convert any encoded string over the ternary alphabet into a binary encoding. For example, a string like #0#1 would have the binary representation 11001101.

Example 1.58 (Ternary encoding of graphs).
Let $A$ be the set of all undirected graphs. Every graph $G = (V, E)$ can be represented by its $|V|$ by $|V|$ adjacency matrix. In this matrix, every row corresponds to a vertex of the graph, and similarly, every column corresponds to a vertex of the graph. The $(i, j)$’th entry contains a 1 if $\{i, j\}$ is an edge, and contains a 0 otherwise. Below is an example.

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 \\
3 & 0 & 1 & 0 \\
4 & 1 & 0 & 1 \\
\end{array}
\]

Such a graph can be encoded using a ternary alphabet as follows. Take the adjacency matrix of the graph, view each row as a binary string, and concatenate all the rows by putting a separator symbol between them. The encoding of the above example would be

\[
\langle G \rangle = 0101#1010#0101#1010.
\]

Example 1.59 (Encoding of Python functions).
Let $A$ be the set of all functions in the programming language Python. Whenever we type up a Python function in a code editor, we are creating a string representation/encoding of the function, where the alphabet is all the Unicode symbols. For example, consider a Python function named absValue, which we can write as

```python
def absValue(N):
    if (N < 0): return -N
    else: return N
```

\[^5\text{We will define graphs formally in a future chapter, however, we assume you are already familiar with the concept.}\]
\[^6\text{https://en.wikipedia.org/wiki/Unicode}\]
By writing out the function, we have already encoded it. More specifically, \( \langle \text{absValue} \rangle \) is the string

```
def absValue(N):
    if (N < 0):
        return -N
    else:
        return N
```

Exercise 1.60 (Unary encoding of integers).
Describe an encoding of \( \mathbb{Z} \) using the alphabet \( \Sigma = \{1\} \).

### 1.4 Computational Problems and Decision Problems

**Definition 1.61** (Computational problem).
Let \( \Sigma \) be an alphabet. Any function \( f : \Sigma^* \rightarrow \Sigma^* \) is called a computational problem over the alphabet \( \Sigma \).

**Example 1.62** (Addition as a computational problem).
Consider the function \( g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \) defined as \( g(x, y) = x + y \). This is a function that expresses the addition problem in naturals. We can view \( g \) as a computational problem over an alphabet \( \Sigma \) once we fix an encoding of the domain \( \mathbb{N} \times \mathbb{N} \) using \( \Sigma \) and an encoding of the codomain \( \mathbb{N} \) using \( \Sigma \). For convenience, we take \( \Sigma = \{0, 1, \#\} \). Let \( \text{Enc} \) be the encoding of \( \mathbb{N} \times \mathbb{N} \) as described in Example 1.56 (Ternary encoding of pairs of naturals). Let \( \text{Enc}' \) be the encoding of \( \mathbb{N} \) as described in Example 1.53 (Binary encoding of naturals). Note that \( \text{Enc}' \) leaves the symbol \( \# \) unused in the encoding. We now define the computational problem \( f \) corresponding to \( g \). If \( w \in \Sigma^* \) is a word that corresponds to a valid encoding of a pair of numbers \((x, y)\) (i.e., \( \text{Enc}(x, y) = w \)), then define \( f(w) \) to be \( \text{Enc}'(x + y) \). If \( w \in \Sigma^* \) is not a word that corresponds to a valid encoding of a pair of numbers (i.e., \( w \) is not in the image of \( \text{Enc} \)), then define \( f(w) \) to be \( \# \). In the codomain, the \( \# \) symbol serves as an “error” indicator.

**IMPORTANT 1.63** (Computational problem as mapping instances to solutions).
A computational problem is often derived from a function \( g : I \rightarrow S \), where \( I \) is a set of objects called instances and \( S \) is a set of objects called solutions. The derivation is done through encodings \( \text{Enc} : I \rightarrow \Sigma^* \) and \( \text{Enc}' : S \rightarrow \Sigma^* \). With these encodings, we can create the computational problem \( f : \Sigma^* \rightarrow \Sigma^* \). In particular, if \( w = \langle x \rangle \) for some \( x \in I \), then we define \( f(w) \) to be \( \text{Enc}'(g(x)) \).

![Diagram](image)

One thing we have to be careful about is defining \( f(w) \) for a word \( w \in \Sigma^* \) that does not correspond to an encoding of an object in \( I \) (such a word does not correspond to an instance of the computational problem). To handle this, we can identify one of the strings in \( \Sigma^* \) as an error string and define \( f(w) \) to be that string.
Definition 1.64 (Decision problem).
Let $\Sigma$ be an alphabet. Any function $f : \Sigma^* \rightarrow \{0, 1\}$ is called a decision problem over the alphabet $\Sigma$. The codomain of the function is not important as long as it has two elements. Other common choices for the codomain are $\{\text{No, Yes}\}$, $\{\text{False, True}\}$ and $\{\text{Reject, Accept}\}$.

Example 1.65 (Primality testing as a decision problem).
Consider the function $g : \mathbb{N} \rightarrow \{\text{False, True}\}$ such that $g(x) = \text{True}$ if and only if $x$ is a prime number. We can view $g$ as a decision problem over an alphabet $\Sigma$ once we fix an encoding of the domain $\mathbb{N}$ using $\Sigma$. Take $\Sigma = \{0, 1\}$. Let $\text{Enc}$ be the encoding of $\mathbb{N}$ as described in Example 1.53 (Binary encoding of naturals). We now define the decision problem $f$ corresponding to $g$. If $w \in \Sigma^*$ is a word that corresponds to an encoding of a prime number, then define $f(w)$ to be True. Otherwise, define $f(w)$ to be False. (Note that in the case of $f(w) = \text{False}$, either $w$ is the encoding of a composite number, or $w$ is not a valid encoding of a natural number.

Note 1.66 (Decision problem as mapping instances to 0 or 1s).
As with a computational problem, a decision problem is often derived from a function $g : I \rightarrow \{0, 1\}$, where $I$ is a set of instances. The derivation is done through an encoding $\text{Enc} : I \rightarrow \Sigma^*$, which allows us to define the decision problem $f : \Sigma^* \rightarrow \{0, 1\}$. Any word $w \in \Sigma^*$ that does not correspond to an encoding of an instance is mapped to 0 by $f$.

IMPORTANT 1.67 (Correspondence between decision problems and languages).
There is a one-to-one correspondence between decision problems and languages. Let $f : \Sigma^* \rightarrow \{0, 1\}$ be some decision problem. Now define $L \subseteq \Sigma^*$ to be the set of all words in $\Sigma^*$ that $f$ maps to 1. This $L$ is the language corresponding to the decision problem $f$. Similarly, if you take any language $L \subseteq \Sigma^*$, we can define the corresponding decision problem $f : \Sigma^* \rightarrow \{0, 1\}$ as $f(w) = 1$ if and only if $w \in L$. We consider the set of languages and the set of decision problems to be the same set of objects.
Quiz

1. Let $L$ be the set of all strings of length at most 3 over the alphabet $\{a, b, c\}$. What is $|L|$?

2. Let $\Sigma$ be an alphabet. For which languages $L \subseteq \Sigma^*$ is it true that $L \cdot L$ is infinite?

3. Let $\Sigma$ be an alphabet. For which languages $L \subseteq \Sigma^*$ is it true that $L^*$ is infinite?

4. True or false: The set of real numbers is encodable.

5. Consider the following problem. The input is an array $A$ of $n$ integers together with a target integer $t$. The output is a subset $S \subseteq \{0, 1, \ldots, n - 1\}$ such that $\sum_{i \in S} A[i] = t$. If no such subset exists, the output is None. Formulate this as a computational problem.
Hints to Selected Exercises

Exercise 1.49 (Can you distribute star over intersection?):
Disprove the statement by providing a counterexample.

Exercise 1.50 (Can you interchange star and reversal?):
Show \((L^*)^R = (L^R)^*\). To do this, you need to argue both \((L^*)^R \subseteq (L^R)^*\) and \((L^R)^* \subseteq (L^*)^R\).
Chapter 2

Deterministic Finite Automata
PREAMBLE

Chapter structure:

- Section 8.1 (Basic Definitions)
  - Definition 2.1 (Deterministic Finite Automaton (DFA))
  - Definition 2.3 (Computation path for a DFA)
  - Definition 2.5 (A DFA accepting a string)
  - Definition 2.7 (Extended transition function)
  - Definition 2.9 (Language recognized/accepted by a DFA)
  - Definition 2.14 (Regular language)
- Section 2.2 (Irregular Languages)
  - Theorem 2.17 (\(0^n1^n\) is not regular)
  - Theorem 2.18 (A unary non-regular language)
- Section 2.3 (Closure Properties of Regular Languages)
  - Theorem 2.23 (Regular languages are closed under union)
  - Corollary 2.25 (Regular languages are closed under intersection)
  - Theorem 2.30 (Regular languages are closed under concatenation)

Chapter goals:

The goal of this chapter is to introduce you to a simple (and restricted) model of computation known as deterministic finite automata. This model is interesting to study in its own right, and has very nice applications, however, our main motivation to study this model is to use it as a stepping stone towards formally defining the notion of an algorithm in its full generality. Treating deterministic finite automata as a warm-up, we would like you to get comfortable with how one formally defines a model of computation, and then proves interesting theorems related to the model. Along the way, you will start getting comfortable with using a bit more sophisticated mathematical notation than you might be used to. You will see how mathematical notation helps us express ideas and concepts accurately, succinctly and clearly.

Applications:

- [https://cstheory.stackexchange.com/questions/8539/how-practical-is-automata-theory](https://cstheory.stackexchange.com/questions/8539/how-practical-is-automata-theory)
- [http://cap.virginia.edu](http://cap.virginia.edu)
2.1 Basic Definitions

Definition 2.1 (Deterministic Finite Automaton (DFA)).
A deterministic finite automaton (DFA) \( M \) is a 5-tuple
\[
M = (Q, \Sigma, \delta, q_0, F),
\]
where

- \( Q \) is a non-empty finite set (which we refer to as the set of states);
- \( \Sigma \) is a non-empty finite set (which we refer to as the alphabet of the DFA);
- \( \delta \) is a function of the form \( \delta : Q \times \Sigma \to Q \) (which we refer to as the transition function);
- \( q_0 \in Q \) is an element of \( Q \) (which we refer to as the start state);
- \( F \subseteq Q \) is a subset of \( Q \) (which we refer to as the set of accepting states).

Example 2.2 (A 4-state DFA).
Below is an example of how we draw a DFA:

- \( \Sigma = \{0, 1\} \), \( Q = \{q_0, q_1, q_2, q_3\} \), \( F = \{q_1, q_2\} \). The labeled arrows between the states encode the transition function \( \delta \), which can also be represented with a table as below (row \( q_i \in Q \) and column \( b \in \Sigma \) contains \( \delta(q_i, b) \)).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( q_0 )</td>
<td>( q_1 )</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( q_2 )</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( q_3 )</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( q_0 )</td>
<td>( q_2 )</td>
</tr>
</tbody>
</table>

Definition 2.3 (Computation path for a DFA).
Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA and let \( w = w_1 w_2 \cdots w_n \) be a string over an alphabet \( \Sigma \) (so \( w_i \in \Sigma \) for each \( i \in \{1, 2, \ldots, n\} \)). Then the computation path of \( M \) with respect to \( w \) is a sequence of states
\[
r_0, r_1, r_2, \ldots, r_n,
\]
where each \( r_i \in Q \), and such that
• $r_0 = q_0$;
• $\delta(r_{i-1},w_i) = r_i$ for each $i \in \{1, 2, \ldots, n\}$.

We say that the computation path is accepting if $r_n \in F$, and rejecting otherwise.

**Example 2.4** (An example of a computation path).
Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA in **Example 2.2** (A 4-state DFA) and let $w = 110110$. Then the computation path of $M$ with respect to $w$ is

$q_0, q_1, q_2, q_3, q_2, q_2, q_3$.

Since $q_3$ is not in $F$, this is a rejecting computation path.

**Definition 2.5** (A DFA accepting a string).
We say that DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a word $w \in \Sigma^*$ if the computation path of $M$ with respect to $w$ is an accepting computation path. Otherwise, we say that $M$ rejects the string $w$.

**Example 2.6** (An example of a DFA accepting a string).
Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA in **Example 2.2** (A 4-state DFA) and let $w = 01101$. Then the computation path of $M$ with respect to $w$ is

$q_0, q_0, q_1, q_2, q_3, q_2$.

This is an accepting computation path because the sequence ends with $q_2$, which is in $F$. Therefore $M$ accepts $w$.

**Definition 2.7** (Extended transition function).
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The transition function $\delta : Q \times \Sigma \to Q$ can be extended to $\delta^* : Q \times \Sigma^* \to Q$, where $\delta^*(q, w)$ is defined as the state we end up in if we start at $q$ and read the string $w$. In fact, often the star in the notation is dropped and $\delta$ is overloaded to represent both a function $\delta : Q \times \Sigma \to Q$ and a function $\delta : Q \times \Sigma^* \to Q$.

**Note 2.8** (Alternative definition of a DFA accepting a string).
Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Using the notation above, we can say that a word $w$ is accepted by the DFA $M$ if $\delta(q_0, w) \in F$.

**Definition 2.9** (Language recognized/accepted by a DFA).
For a deterministic finite automaton $M$, we let $L(M)$ denote the set of all strings that $M$ accepts, i.e. $L(M) = \{w \in \Sigma^* : M$ accepts $w\}$. We refer to $L(M)$ as the language recognized by $M$ (or as the language accepted by $M$, or as the language decided by $M$).\(^1\)

**Example 2.10** (Even number of 1’s).
The following DFA recognizes the language consisting of all binary strings that contain an even number of 1’s.

---

\(^1\)Here the word “accept” is overloaded since we also use it in the context of a DFA accepting a string. However, this usually does not create any ambiguity. Note that the letter $L$ is also overloaded since we often use it to denote a language $L \subseteq \Sigma^*$. In this definition, you see that it can also denote a function that maps a DFA to a language. Again, this overloading should not create any ambiguity.
Example 2.11 (Ends with 00).
The following DFA recognizes the language consisting of all binary strings that end with 00.

Exercise 2.12 (Draw DFAs).
For each language below (over the alphabet $\Sigma = \{0, 1\}$), draw a DFA recognizing it.

(a) $\{110, 101\}$
(b) $\{0, 1\}^* \setminus \{110, 101\}$
(c) $\{x \in \{0, 1\}^* : x$ starts and ends with the same bit}$
(d) $\{\epsilon, 110, 110110, 110110110, \ldots\}$
(e) $\{x \in \{0, 1\}^* : x$ contains 110 as a substring}$

Exercise 2.13 (Finite languages are regular).
Let $L$ be a finite language, i.e., it contains a finite number of words. Show that there is a DFA recognizing $L$.

Definition 2.14 (Regular language).
A language $L \subseteq \Sigma^*$ is called regular if there is a deterministic finite automaton $M$ such that $L = L(M)$.

Example 2.15 (Some examples of regular languages).
All the languages in Exercise 2.12 (Draw DFAs) are regular languages.

Exercise 2.16 (Equal number of 01’s and 10’s).
Is the language

$$\{w \in \{0, 1\}^* : w$ contains an equal number of occurrences of 01 and 10 as substrings.$\}$$

regular?
2.2 Irregular Languages

Theorem 2.17 ($0^n1^n$ is not regular).

Let $\Sigma = \{0, 1\}$. The language $L = \{0^n1^n : n \in \mathbb{N}\}$ is not regular.

Proof. Our goal is to show that $L = \{0^n1^n : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let’s assume that $L$ is regular.

Since $L$ is regular, by definition, there is some deterministic finite automaton $M$ that recognizes $L$. Let $k$ denote the number of states of $M$. For $n \in \mathbb{N}$, let $r_n$ denote the state that $M$ reaches after reading $0^n$ (i.e., $r_n = \delta(q_0, 0^n)$). By the pigeonhole principle, we know that there must be a repeat among $r_0, r_1, \ldots, r_k$ (a sequence of $k + 1$ states). In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i \neq j$ such that $r_i = r_j$. This means that the string $0^i$ and the string $0^j$ end up in the same state in $M$. Therefore $0^i w$ and $0^j w$, for any string $w \in \{0, 1\}^*$, end up in the same state in $M$. We’ll now reach a contradiction, and conclude the proof, by considering a particular $w$ such that $0^i w$ and $0^j w$ end up in different states.

Consider the string $w = 1^t$. Then since $M$ recognizes $L$, we know $0^i w = 0^j 1^t$ must end up in an accepting state. On the other hand, since $i \neq j$, $0^i w = 0^j 1^t$ is not in the language, and therefore cannot end up in an accepting state. This is the desired contradiction.

Theorem 2.18 (A unary non-regular language).

Let $\Sigma = \{a\}$. The language $L = \{a^{2^n} : n \in \mathbb{N}\}$ is not regular.

Proof. Our goal is to show that $L = \{a^{2^n} : n \in \mathbb{N}\}$ is not regular. The proof is by contradiction. So let’s assume that $L$ is regular.

Since $L$ is regular, by definition, there is some deterministic finite automaton $M$ that recognizes $L$. Let $k$ denote the number of states of $M$. For $n \in \mathbb{N}$, let $r_n$ denote the state that $M$ reaches after reading $a^{2^n}$ (i.e., $r_n = \delta(q_0, a^{2^n})$). By the pigeonhole principle, we know that there must be a repeat among $r_0, r_1, \ldots, r_k$ (a sequence of $k + 1$ states). In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i < j$ such that $r_i = r_j$. This means that the string $a^{2^i}$ and the string $a^{2^j}$ end up in the same state in $M$. Therefore $a^{2^i} w$ and $a^{2^j} w$, for any string $w \in \{a\}^*$, end up in the same state in $M$. We’ll now reach a contradiction, and conclude the proof, by considering a particular $w$ such that $a^{2^i} w$ ends up in an accepting state but $a^{2^j} w$ ends up in a rejecting state (i.e. they end up in different states).

Consider the string $w = a^{2^i}$. Then $a^{2^i} w = a^{2^i} a^{2^i} = a^{2^{i+1}}$, and therefore must end up in an accepting state. On the other hand, $a^{2^j} w = a^{2^j} a^{2^j} = a^{2^{j+2^i}}$. We claim that this word must end up in a rejecting state because $2^j + 2^i$ cannot be written as a power of 2 (i.e., cannot be written as $2^t$ for some $t \in \mathbb{N}$). To see this, note that since $i < j$, we have

$$2^j < 2^j + 2^i < 2^j + 2^i = 2^{j+1},$$

which implies that if $2^j + 2^i = 2^t$, then $j < t < j+1$. So $2^j + 2^i$ cannot be written as $2^t$ for $t \in \mathbb{N}$, and therefore $a^{2^{j+2^i}}$ leads to a reject state in $M$ as claimed.

---

The pigeonhole principle states that if $n$ items are put inside $m$ containers, and $n > m$, then there must be at least one container with more than one item. The name pigeonhole principle comes from thinking of the items as pigeons, and the containers as holes. The pigeonhole principle is often abbreviated as PHP.
Exercise 2.19 \((a^n b^n c^n)\) is not regular.  
Let \(\Sigma = \{a, b, c\}\). Prove that \(L = \{a^n b^n c^n : n \in \mathbb{N}\}\) is not regular.

Exercise 2.20 \((c^{2n+1} a^n b^{2n})\) is not regular.  
Let \(\Sigma = \{a, b, c\}\). Prove that \(L = \{c^{2n+1} a^n b^{2n} : n \in \mathbb{N}\}\) is not regular.

### 2.3 Closure Properties of Regular Languages

**Exercise 2.21** (Are regular languages closed under complementation?).  
Is it true that if \(L\) is regular, then its complement \(\Sigma^* \setminus L\) is also regular? In other words, are regular languages closed under the complementation operation?

**Exercise 2.22** (Are regular languages closed under subsets?).  
Is it true that if \(L \subseteq \Sigma^*\) is a regular language, then any \(L' \subseteq L\) is also a regular language?

**Theorem 2.23** (Regular languages are closed under union).  
Let \(\Sigma\) be some finite alphabet. If \(L_1 \subseteq \Sigma^*\) and \(L_2 \subseteq \Sigma^*\) are regular languages, then the language \(L_1 \cup L_2\) is also regular.

**Proof.** Given regular languages \(L_1\) and \(L_2\), we want to show that \(L_1 \cup L_2\) is regular. Since \(L_1\) and \(L_2\) are regular languages, by definition, there are DFAs \(M = (Q, \Sigma, \delta, q_0, F)\) and \(M' = (Q', \Sigma, \delta', q_0', F')\) that recognize \(L_1\) and \(L_2\) respectively (i.e., \(L(M) = L_1\) and \(L(M') = L_2\)). To show \(L_1 \cup L_2\) is regular, we'll construct a DFA \(M'' = (Q'', \Sigma, \delta'', q''_0, F'')\) that recognizes \(L_1 \cup L_2\). The definition of \(M''\) will make use of \(M\) and \(M'\). In particular:

- the set of states is \(Q'' = Q \times Q' = \{(q, q') : q \in Q, q' \in Q'\}\);

- the transition function \(\delta''\) is defined such that for \((q, q') \in Q''\) and \(a \in \Sigma\),  

\[
\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a));
\]

(Note that for \(w \in \Sigma^*\), \(\delta''((q, q'), w) = (\delta(q, w), \delta'(q', w))\).

- the initial state is \(q''_0 = (q_0, q'_0)\);

- the set of accepting states is \(F'' = \{(q, q') : q \in F \text{ or } q' \in F'\}\).

This completes the definition of \(M''\). It remains to show that \(M''\) indeed recognizes the language \(L_1 \cup L_2\), i.e., \(L(M'') = L_1 \cup L_2\). We will first argue that \(L_1 \cup L_2 \subseteq L(M'')\) and then argue that \(L(M'') \subseteq L_1 \cup L_2\). Both inclusions will follow easily from the definition of \(M''\) and the definition of a DFA accepting a string.

\(L_1 \cup L_2 \subseteq L(M'')\): Suppose \(w \in L_1 \cup L_2\), which means \(w\) either belongs to \(L_1\) or it belongs to \(L_2\). Our goal is to show that \(w \in L(M'')\). Without loss of generality, assume \(w\) belongs to \(L_1\), or in other words, \(M\) accepts \(w\) (the argument is essentially identical when \(w\) belongs to \(L_2\)). So we know that \(\delta(q_0, w) \in F\). By the definition of \(\delta''\), \(\delta''((q_0, q'_0), w) = (\delta(q_0, w), \delta'(q'_0, w))\). And since \(\delta(q_0, w) \in F\), \(\delta'(q'_0, w) \in F'\) (by the definition of \(F''\)). So \(w\) is accepted by \(M''\) as desired.

\(L(M'') \subseteq L_1 \cup L_2\): Suppose that \(w \in L(M'')\). Our goal is to show that \(w \in L_1\) or \(w \in L_2\). Since \(w\) is accepted by \(M''\), we know that \(\delta''((q_0, q'_0), w) = (\delta(q_0, w), \delta'(q'_0, w)) \in F''\). By the definition of \(F''\), this means that either \(\delta(q_0, w) \in F\) or \(\delta'(q'_0, w) \in F'\), i.e., \(w\) is accepted by \(M\) or \(M'\). This implies that either \(w \in L(M) = L_1\) or \(w \in L(M') = L_2\), as desired. 

\(\square\)
Note 2.24 (On proof write-up).
Observe that the proof of Theorem 2.23 (Regular languages are closed under union) contains very little information about how one comes up with such a proof or what is the “right” intuitive interpretation of the construction. Many proofs in the literature are actually written in this manner, which can be frustrating for the reader. We have explained the intuition and the cognitive process that goes into discovering the above proof during class. Therefore we chose not to include any of these details in the above write-up. However, we do encourage you to include a “proof idea” component in your write-ups when you believe that the intuition is not very transparent.

Corollary 2.25 (Regular languages are closed under intersection).
Let $\Sigma$ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cap L_2$ is also regular.

Proof. We want to show that regular languages are closed under the intersection operation. We know that regular languages are closed under union (Theorem 2.23 (Regular languages are closed under union)) and closed under complementation (Exercise 2.21 (Are regular languages closed under complementation?)). The result then follows since $A \cap B = \overline{A \cup \overline{B}}$. □

Exercise 2.26 (Direct proof that regular languages are closed under difference).
Give a direct proof (without using the fact that regular languages are closed under complementation, union and intersection) that if $L_1$ and $L_2$ are regular languages, then $L_1 \setminus L_2$ is also regular.

Exercise 2.27 (Finite vs infinite union).
(a) Suppose $L_1, \ldots, L_k$ are all regular languages. Is it true that their union $\bigcup_{i=0}^k L_i$ must be a regular language?
(b) Suppose $L_0, L_1, L_2, \ldots$ is an infinite sequence of regular languages. Is it true that their union $\bigcup_{i \geq 0} L_i$ must be a regular language?

Exercise 2.28 (Union of irregular languages).
Suppose $L_1$ and $L_2$ are not regular languages. Is it always true that $L_1 \cup L_2$ is not a regular language?

Exercise 2.29 (Regularity of suffixes and prefixes).
Suppose $L \subseteq \Sigma^*$ is a regular language. Show that the following languages are also regular:

$$\text{SUFFIXES}(L) = \{ x \in \Sigma^* : yx \in L \text{ for some } y \in \Sigma^* \},$$
$$\text{PREFIXES}(L) = \{ y \in \Sigma^* :yx \in L \text{ for some } x \in \Sigma^* \}.$$  

Theorem 2.30 (Regular languages are closed under concatenation).
If $L_1, L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1L_2$ is also regular.
Proof. Given regular languages \( L_1 \) and \( L_2 \), we want to show that \( L_1L_2 \) is regular. Since \( L_1 \) and \( L_2 \) are regular languages, by definition, there are DFAs \( M = (Q, \Sigma, \delta, q_0, F) \) and \( M' = (Q', \Sigma, \delta', q'_0, F') \) that recognize \( L_1 \) and \( L_2 \) respectively. To show \( L_1L_2 \) is regular, we'll construct a DFA \( M'' = (Q'', \Sigma, \delta'', q''_0, F'') \) that recognizes \( L_1L_2 \). The definition of \( M'' \) will make use of \( M \) and \( M' \).

Before we formally define \( M'' \), we will introduce a few key concepts and explain the intuition behind the construction.

We know that \( w \in L_1L_2 \) if and only if there is a way to write \( w \) as \( uv \) where \( u \in L_1 \) and \( v \in L_2 \). With this in mind, we first introduce the notion of a thread. Given a word \( w = w_1w_2 \ldots w_n \in \Sigma^* \), a thread with respect to \( w \) is a sequence of states

\[
q_0, q_1, q_2, \ldots, q_i, q_{i+1}, q_{i+2}, \ldots, q_n,
\]

where \( q_0, q_1, \ldots, q_i \) is an accepting computation path of \( M \) with respect to \( w_1w_2 \ldots w_i \) and \( q'_0, q_{i+1}, q_{i+2}, \ldots, q_n \) is a computation path (not necessarily accepting) of \( M' \) with respect to \( w_i+1w_i+2 \ldots w_n \). A thread like this corresponds to simulating \( M \) on \( w_1w_2 \ldots w_i \) (at which point we require that an accepting state of \( M \) is reached), and then simulating \( M' \) on \( w_i+1w_i+2 \ldots w_n \). For each way of writing \( w \) as \( uv \) where \( u \in L_1 \), there is a corresponding thread for it. Note that \( w \in L_1L_2 \) if and only if there is a thread in which \( q_n \in F' \). Our goal is to construct the DFA \( M'' \) such that it keeps track of all possible threads, and if one of the threads ends with a state in \( F' \), then \( M'' \) accepts.

At first, it might seem like one cannot keep track of all possible threads using only constant number of states. However this is not the case. Let’s identify a thread with its sequence of \( s_j \)’s (i.e. the sequence of states from \( Q' \) corresponding to the simulation of \( M' \)). Consider two threads (for the sake of example, let’s take \( n = 10 \)):

\[
\begin{align*}
s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10} \\
s'_5, s'_6, s'_7, s'_8, s'_9, s'_{10}
\end{align*}
\]

If, say, \( s_i = s'_i = q' \in Q' \) for some \( i \), then \( s_j = s'_j \) for all \( j > i \) (in particular, \( s_{10} = s'_{10} \)). At the end, all we care about is whether \( s_{10} \) or \( s'_{10} \) is an accepting state of \( M' \). So at index \( i \), we do not need to remember that there are two copies of \( q' \); it suffices to keep track of one copy. In general, at any index \( i \), when we look at all the possible threads, we want to keep track of the unique states that appear at that index, and not worry about duplicates. Since we do not need to keep track of duplicated states, what we need to remember is a subset of \( Q' \) (recall that a set cannot have duplicated elements).

The construction of \( M'' \) we present below keeps track of all the threads using constant number of states. Indeed, the set of states is \( Q'' = Q \times \mathcal{P}(Q') \), where the first component keeps track of which state we are at in \( M \), and the second component keeps track of all the unique states of \( M' \) that we can be at if we are following one of the possible threads.

Before we present the formal definition of \( M'' \), we introduce one more definition. Recall that the transition function of \( M' \) is \( \delta' : Q' \times \Sigma \to Q' \). Using \( \delta' \) we define a new function \( \delta_p : \mathcal{P}(Q') \times \Sigma \to \mathcal{P}(Q') \) as follows. For \( S \subseteq Q' \) and \( a \in \Sigma \), \( \delta_p(S, a) \) is defined to be the set of all possible states that we can end up at if we start in a state in \( S \) and read the symbol \( a \). In other words,

\[
\delta_p(S, a) = \{ \delta'(q', a) : q' \in S \}.
\]

It is appropriate to view \( \delta_p \) as an extension/generalization of \( \delta' \).

Here is the formal definition of \( M'' \):

\footnote{This means \( r_0 = q_0, r_j \in F \), and when the symbol \( w_j \) is read, \( M \) transitions from state \( r_{j-1} \) to state \( r_j \), See Definition 2.3 (Computation path for a DFA)).}

\footnote{Recall that for any set \( Q \), the set of all subsets of \( Q \) is called the power set of \( Q \), and is denoted by \( \mathcal{P}(Q) \).}
• The set of states is \( Q'' = Q \times P(Q') = \{ (q, S) : q \in Q, S \subseteq Q' \} \).
  (The first coordinate keeps track of which state we are at in the first machine \( M \), and the second coordinate keeps track of the set of states we can be at in the second machine \( M' \) if we follow one of the possible threads.)

• The transition function \( \delta'' \) is defined such that for \( (q, S) \in Q'' \) and \( a \in \Sigma \),
  \[
  \delta''((q, S), a) = \begin{cases} 
  (\delta(q, a), \delta'_P(S, a)) & \text{if } \delta(q, a) \not\in F, \\
  (\delta(q, a), \delta'_P(S, a) \cup \{q'_0\}) & \text{if } \delta(q, a) \in F.
  \end{cases}
  \]
  (The first coordinate is updated according to the transition rule of the first machine. The second coordinate is updated according to the transition rule of the second machine. Since for the second machine, we are keeping track of all possible states we could be at, the extended transition function \( \delta'_P \) gives us all possible states we can go to when reading a character \( a \). Note that if after applying \( \delta \) to the first coordinate, we get a state that is an accepting state of the first machine, a new thread must be created and kept track of. This is accomplished by adding \( q'_0 \) to the second coordinate.)

• The initial state is
  \[
  q''_0 = \begin{cases} 
  (q_0, \emptyset) & \text{if } q_0 \not\in F, \\
  (q_0, \{q'_0\}) & \text{if } q_0 \in F.
  \end{cases}
  \]
  (Initially, if \( q_0 \not\in F \), then there are no threads to keep track of, so the second coordinate is the empty set. On the other hand, if \( q_0 \in F \), then there is already a thread that we need to keep track of – the one corresponding to running the whole input word \( w \) on the second machine – so we add \( q'_0 \) to the second coordinate to keep track of this thread.)

• The set of accepting states is \( F'' = \{ (q, S) : q \in Q, S \subseteq Q', S \cap F' \neq \emptyset \} \).
  (In other words, \( M'' \) accepts if and only if there is a state in the second coordinate that is an accepting state of the second machine \( M' \). So \( M'' \) accepts if and only if one of the possible threads ends in an accepting state of \( M' \).)

This completes the definition of \( M'' \).

To see that \( M'' \) indeed recognizes the language \( L_1L_2 \), i.e. \( L(M'') = L_1L_2 \), note that by construction, \( M'' \) with input \( w \), does indeed keep track of all the possible threads. And it accepts \( w \) if and only if one of those threads ends in an accepting state of \( M' \). The result follows since \( w \in L_1L_2 \) if and only if there is a thread with respect to \( w \) that ends in an accepting state of \( M' \). \( \square \)
Quiz

1. Fix some alphabet $\Sigma$. How many DFAs are there with exactly one state?

2. Let $L \subseteq \{a\}^*$ be a language consisting of all strings of $a$’s of odd length except length 15251. Is $L$ regular?

3. Let $L$ be the set of all strings in $\{0, 1\}^*$ that contain at least 15251 0’s and at most 15251 1’s. Is $L$ regular?

4. True or false: Let $L_1 \oplus L_2$ denote the set of all words in either $L_1$ or $L_2$, but not both. If $L_1$ and $L_2$ are regular, then so is $L_1 \oplus L_2$.

5. True or false: For languages $L$ and $L'$, if $L \subseteq L'$ and $L$ is non-regular, then $L'$ is non-regular.

6. True or false: If $L \subseteq \Sigma^*$ is non-regular, then $L = \Sigma^* \backslash L$ is non-regular.
Hints to Selected Exercises

Exercise 2.13 (Finite languages are regular):
First think about whether languages of size 1 are regular? Are languages of size 2 regular? (The first part of Exercise (Draw DFAs) might help.) Can you generalize your idea to any finite size language?

Exercise 2.16 (Equal number of 01’s and 10’s):
Yes, it is.

Exercise 2.21 (Are regular languages closed under complementation?):
The answer is yes. How can you construct a DFA recognizing $\Sigma^* \setminus L$ given that you have a DFA recognizing $L$?

Exercise 2.22 (Are regular languages closed under subsets?):
The answer is no. Find a counter-example.

Exercise 2.27 (Finite vs infinite union):
The answer for the first part is yes, and the second part is no.

Exercise 2.28 (Union of irregular languages):
The answer is no. Find a counter-example.
Chapter 3

Turing Machines
PREAMBLE

Chapter structure:

- Section 8.1 (Basic Definitions)
  - Definition 3.1 (Turing machine)
  - Definition 3.6 (A TM accepting or rejecting a string)
  - Definition 3.9 (Decider Turing machine)
  - Definition 3.10 (Language accepted and decided by a TM)
  - Definition 3.11 (Decidable language)
  - Definition 3.18 (Universal Turing machine)
- Section 3.2 (Decidable Languages)
  - Definition 3.22 (Languages related to encodings of DFAs)
  - Theorem 3.23 (ACCEPTS_{DFA} and SELF-ACCEPTS_{DFA} are decidable)
  - Theorem 3.24 (EMPTY_{DFA} is decidable)
  - Theorem 3.25 (EQ_{DFA} is decidable)

Chapter goals:

In this chapter, our main goal is to introduce the definition of a Turing machine, which is the standard mathematical model for any kind of computational device. As such, this definition is very foundational. As we discuss in lecture, the physical Church-Turing thesis asserts that any kind of physical device or phenomenon, when viewed as a computational process mapping input data to output data, can be simulated by some Turing machine. Thus, rigorously studying Turing machines does not just give us insights about what our laptops can or cannot do, but also tells us what the universe can and cannot do computationally.

This chapter kicks things off with examples of decidable languages (i.e. decision problems that we can compute). Next chapter, we will start exploring the limitations of computation. Some of the examples we cover in this chapter will serve as a warm up to other examples we will discuss in the next chapter in the context of uncomputability.
3.1 Basic Definitions

**Definition 3.1 (Turing machine).**
A Turing machine (TM) $M$ is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where

- $Q$ is a non-empty finite set (which we refer to as the *set of states*);
- $\Sigma$ is a non-empty finite set that does not contain the *blank symbol* $\sqcup$ (which we refer to as the *input alphabet*);
- $\Gamma$ is a finite set such that $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$ (which we refer to as the *tape alphabet*);
- $\delta$ is a function of the form $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ (which we refer to as the *transition function*);
- $q_0 \in Q$ is an element of $Q$ (which we refer to as the *initial state* or *starting state*);
- $q_{\text{acc}} \in Q$ is an element of $Q$ (which we refer to as the *accepting state*);
- $q_{\text{rej}} \in Q$ is an element of $Q$ such that $q_{\text{rej}} \neq q_{\text{acc}}$ (which we refer to as the *rejecting state*).

**Example 3.2 (A 5-state TM).**
Below is an example of how we draw a TM:

![Turing Machine Diagram](image)

In this example, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \sqcup\}$, $Q = \{q_0, q_a, q_{\text{rej}}, q_{\text{acc}}, q_b\}$. The labeled arrows between the states encode the transition function $\delta$. As an example, the arrow from state $q_0$ to $q_a$ represents $\delta(q_0, a) = (q_a, \sqcup, R)$. The above picture is called the *state diagram* of the Turing machine.

**Note 3.3 (Equivalence of Turing machines).**
We’ll consider two Turing machines to be equivalent/same if they are the same machine up to renaming the elements of the sets $Q, \Sigma$ and $\Gamma$. 

27
Note 3.4 (No transition out of accepting and rejecting states).
In the transition function $\delta$ of a TM, we don’t really care about how we define the output of $\delta$ when the input state is $q_{\text{acc}}$ or $q_{\text{rej}}$ because once the computation reaches one of these states, it stops. We explain this below in Definition 3.6 (A TM accepting or rejecting a string).

IMPORTANT 3.5 (A Turing machine uses a tape).
A Turing Machine is always accompanied by a tape that is used as memory. The tape is just a sequence of cells that can hold any symbol from the tape alphabet. The tape can be defined so that it is infinite in two directions (so we could imagine indexing the cells using the integers $\mathbb{Z}$), or it could be infinite in one direction, to the right (so we could imagine indexing the cells using the natural numbers $\mathbb{N}$). Initially, an input $w_1 \ldots w_n \in \Sigma^*$ is put on the tape so that symbol $w_i$ is placed on the cell with index $i - 1$. In these notes, we assume our tape is infinite in two directions.

Definition 3.6 (A TM accepting or rejecting a string).
Let $M$ be a Turing machine where $Q$ is the set of states, $\sqcup$ is the blank symbol, and $\Gamma$ is the tape alphabet. To understand how $M$’s computation proceeds we generally need to keep track of three things: (i) the state $M$ is in; (ii) the contents of the tape; (iii) where the tape head is. These three things are collectively known as the “configuration” of the TM. More formally: a configuration for $M$ is defined to be a string $uqv \in (\Gamma \cup Q)^*$, where $u, v \in \Gamma^*$ and $q \in Q$. This represents that the tape has contents $\cdots \sqcup \sqcup \sqcup u v \sqcup \sqcup \cdots$, the head is pointing at the leftmost symbol of $v$, and the state is $q$. We say the configuration is accepting if $q$ is $M$’s accept state and that it’s rejecting if $q$ is $M$’s reject state.

Suppose that $M$ reaches a certain configuration $\alpha$ (which is not accepting or rejecting). Knowing just this configuration and $M$’s transition function $\delta$, one can determine the configuration $\beta$ that $M$ will reach at the next step of the computation. (As an exercise, make this statement precise.) We write $\alpha \vdash_M \beta$ and say that “$\alpha$ yields $\beta$ (in $M$)”. If it’s obvious what $M$ we’re talking about, we drop the subscript $M$ and just write $\alpha \vdash \beta$.

Given an input $x \in \Sigma^*$ we say that $M(x)$ halts if there exists a sequence of configurations (called the computation trace) $\alpha_0, \alpha_1, \ldots, \alpha_T$ such that:

(i) $\alpha_0 = q_0 x$, where $q_0$ is $M$’s initial state;
(ii) $\alpha_t \vdash_M \alpha_{t+1}$ for all $t = 0, 1, 2, \ldots, T - 1$;
(iii) $\alpha_T$ is either an accepting configuration (in which case we say $M(x)$ accepts) or a rejecting configuration (in which case we say $M(x)$ rejects).

Otherwise, we say $M(x)$ loops.

IMPORTANT 3.7 (Turing machines can loop forever).
Given any DFA and any input string, the DFA always halts and makes a decision to either reject or accept the string. The same is not true for Turing machines. It is possible that a Turing machine does not make a decision when given an input string, and instead, loops forever. So given a TM $M$ and an input string $x$, there are 3 options when we run $M$ on $x$:

---

1Supernerd note: we will always assume $Q$ and $\Gamma$ are disjoint sets.
2There are some technicalities: The string $u$ cannot start with $\sqcup$ and the string $v$ cannot end with $\sqcup$. This is so that the configuration is always unique. Also, if $v = \epsilon$ it means the head is pointing at the $\sqcup$ immediately to the right of $u$. 

28
• $M$ accepts $x$;
• $M$ rejects $x$;
• $M$ loops forever.

This is an important distinction between DFAs and TMs.

**Exercise 3.8** (Practice with configurations).

(a) Suppose $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ is a Turing machine. We want you to formally define $\alpha \vdash_M \beta$. More precisely, suppose $\alpha = uqv$, where $q \in Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}$. Precisely describe $\beta$.

(b) Let $M$ denote the Turing machine shown below, which has input alphabet $\Sigma = \{0\}$ and tape alphabet $\Gamma = \{0, x, \_\}$. (Note on notation: A transition label usually has two symbols, one corresponding to the symbol being read, and the other corresponding to the symbol being written. If a transition label has one symbol, the interpretation is that the symbol being read and written is exactly the same.)

We want you to prove that $M$ accepts the input 0000 using the definition on the previous page. More precisely, we want you to write out the computation trace

\[ \alpha_0 \vdash_M \alpha_1 \vdash_M \cdots \vdash_M \alpha_T \]

for $M(0000)$. You do not have to justify it; just make sure to get $T$ and $\alpha_0, \ldots, \alpha_T$ correct!

**Definition 3.9** (Decider Turing machine).
A Turing machine is called a decider if it halts on all inputs.
**Definition 3.10** (Language accepted and decided by a TM).
Let $M$ be a Turing machine (not necessarily a decider). We denote by $L(M)$ the set of all strings that $M$ accepts, and we call $L(M)$ the language accepted by $M$. When $M$ is a decider, we say that $M$ decides the language $L(M)$.

**Definition 3.11** (Decidable language).
A language $L$ is called decidable (or computable) if $L = L(M)$ for some decider Turing machine $M$.

**Exercise 3.12** (A simple decidable language).
Give a description of the language decided by the TM shown in the example corresponding to **Definition 3.1** (Turing machine).

**Exercise 3.13** (Drawing TM state diagrams).
For each language below, draw the state diagram of a TM that decides the language. You can use any finite tape alphabet $\Gamma$ containing the elements of $\Sigma$ and the symbol $\bot$.

(a) $L = \{0^n1^n : n \in \mathbb{N}\}$, where $\Sigma = \{0, 1\}$.

(b) $L = \{0^n : n$ is a nonnegative integer power of 2$\}$, where $\Sigma = \{0\}$.

**IMPORTANT 3.14** (The Church-Turing Thesis).
The Church-Turing Thesis (CTT)$^3$ states that any computation that can be conducted in this universe (constrained by the laws of physics of course), can be carried out by a TM. There are a couple of important things to highlight. First, CTT says nothing about the efficiency of the simulation.$^4$ Second, CTT is not a mathematical statement, but a physical claim about the universe we live in (similar to claiming that the speed of light is constant). The implications of CTT is far-reaching. For example, CTT claims that any computation that can be carried out by a human can be carried out by a TM. Other implications are discussed in lecture.

**Note 3.15** (Low-level, medium-level, high-level descriptions of TMs).
A low-level description of a TM is given by specifying the 7-tuple in its definition. This information is often presented using a picture of its state diagram. A medium-level description includes an English description of the movement and behavior of the tape head, as well as how the contents of the tape is changing, as the computation is being carried out. A high-level description is pseudocode or an algorithm written in English. Usually, an algorithm is written in a way so that a human could read it, understand it, and carry out its steps. By CTT, there is a TM that can carry out the same computation. Unless explicitly stated otherwise, you can present a TM using a high-level description.

---

$^3$The statement we are using here is often called the Physical Church-Turing Thesis and is more general than the original Church-Turing Thesis. In the original Church-Turing Thesis, computation is considered to correspond to a human following step-by-step instructions.

$^4$As an example, quantum computers can be simulated by TMs, but in certain cases, we believe that the simulation can be exponentially slower.
Note 3.16 (Encodings of machines).
In Chapter 1 we saw that we can use the notation $\langle \cdot \rangle$ to denote an encoding of objects belonging to any countable set. For example, if $D$ is a DFA, we can write $\langle D \rangle$ to denote the encoding of $D$ as a string. If $M$ is a TM, we can write $\langle M \rangle$ to denote the encoding of $M$. There are many ways one can encode DFAs and TMs. We will not be describing a specific encoding scheme as this detail will not be important for us.\footnote{As an example, if $P$ is some Python program, we can take $\langle P \rangle$ to be the string that represents the source code of the program. A DFA or a TM can also be viewed as a piece of code (as discussed in lecture). So we could define an encoded DFA or TM to be the string that represents that code.}

Recall that when we want to encode a tuple of objects, we use the comma sign. For example, if $M_1$ and $M_2$ are two Turing machines, we write $\langle M_1, M_2 \rangle$ to denote the encoding of the tuple $(M_1, M_2)$. As another example, if $M$ is a TM and $x \in \Sigma^*$, we can write $\langle M, x \rangle$ to denote the encoding of the tuple $(M, x)$.

IMPORTANT 3.17 (Code is data).
The fact that we can encode different types of objects with strings has the corollary that a Turing machine, or any piece of code, can be viewed as a string, and therefore as data. This means code can take as input other code (in fact, code can take itself as the input). This point of view has several important implications, one of which is the fact that we can come up with a Turing machine, which given as input the description of any Turing machine, can simulate it. This simulator Turing machine is called a universal Turing machine.

Definition 3.18 (Universal Turing machine).
Let $\Sigma$ be some finite alphabet. A universal Turing machine $U$ is a Turing machine that takes $\langle M, x \rangle$ as input, where $M$ is a TM and $x$ is a word in $\Sigma^*$, and has the following high-level description:

$M$: Turing machine. $x$: string in $\Sigma^*$.

$U(\langle M, x \rangle)$:
1. Simulate $M$ on input $x$ (i.e. run $M(x)$).
2. If it accepts, accept.
3. If it rejects, reject.

Note that if $M(x)$ loops forever, then $U$ loops forever as well. To make sure $M$ always halts, we can add a third input, an integer $k$, and have the universal machine simulate the input TM for at most $k$ steps.

IMPORTANT 3.19 (Checking the input type).
When we give a high-level description of a TM, we often assume that the input given is of the correct form/type. For example, with the Universal TM above, we assumed that the input was the encoding $\langle M, x \rangle$ where $M$ is a TM and $x$ is an input string for $M$. But technically, the input to the universal TM could be any finite-length string. What do we do if the input string does not correspond to a valid encoding of an expected type of input object?

Even though this is not explicitly written, we will implicitly assume that the first thing our machine does is check whether the input is a valid encoding of an object with the expected type. If it is not, the machine rejects. If it is, then it will carry on with the specified instructions.

The important thing to keep in mind is that in our descriptions of Turing machines, this step of checking whether the input string has the correct form (i.e. that it is a valid encoding) will never be explicitly written, and we don’t expect you to explicitly write it either. That being said, be aware that this check is implicitly there.
3.2 Decidable Languages

**Exercise 3.20** (Decidability is closed under intersection and union).
Let $L$ and $K$ be decidable languages. Show that $L \cap K$ and $L \cup K$ are also decidable by presenting high-level descriptions of TMs deciding them.

**Exercise 3.21** (Decidable language based on pi).
Let $L \subseteq \{3\}^*$ be defined as follows: $x \in L$ if and only if $x$ appears somewhere in the decimal expansion of $\pi$. For example, the strings $\epsilon$, 3, and 33 are all definitely in $L$, because

$$\pi = 3.1415926535897932384626433 \ldots$$

Prove that $L$ is decidable. No knowledge in number theory is required to solve this question.

**Definition 3.22** (Languages related to encodings of DFAs).
Fix some alphabet $\Sigma$. We define the following languages:

- $\text{ACCEPTS}_{\text{DFA}} = \{(D, x) : D \text{ is a DFA that accepts the string } x\}$,
- $\text{SELF-ACCEPTS}_{\text{DFA}} = \{(D) : D \text{ is a DFA that accepts the string } \langle D \rangle\}$,
- $\text{EMPTY}_{\text{DFA}} = \{(D) : D \text{ is a DFA with } L(D) = \emptyset\}$,
- $\text{EQ}_{\text{DFA}} = \{(D_1, D_2) : D_1 \text{ and } D_2 \text{ are DFAs with } L(D_1) = L(D_2)\}$.

**Theorem 3.23** (ACCEPTS_{DFA} and SELF-ACCEPTS_{DFA} are decidable).
The languages $\text{ACCEPTS}_{\text{DFA}}$ and $\text{SELF-ACCEPTS}_{\text{DFA}}$ are decidable.

*Proof.* Our goal is to show that $\text{ACCEPTS}_{\text{DFA}}$ and $\text{SELF-ACCEPTS}_{\text{DFA}}$ are decidable languages. To show that these languages are decidable, we will give high-level descriptions of TMs deciding them.

For $\text{ACCEPTS}_{\text{DFA}}$, the decider is essentially the same as a universal TM:

- $D$: DFA, $x$: string.
- $M((D, x))$:
  1. Simulate $D$ on input $x$ (i.e. run $D(x)$).
  2. If it accepts, accept.
  3. If it rejects, reject.

It is clear that this correctly decides $\text{ACCEPTS}_{\text{DFA}}$.

For $\text{SELF-ACCEPTS}_{\text{DFA}}$, we just need to slightly modify the above machine:

- $D$: DFA.
- $M((\langle D \rangle))$:
  1. Simulate $D$ on input $\langle D \rangle$ (i.e. run $D(\langle D \rangle)$).
  2. If it accepts, accept.
  3. If it rejects, reject.

Again, it is clear that this correctly decides $\text{SELF-ACCEPTS}_{\text{DFA}}$. 

\[\]
Theorem 3.24 (EMPTY\textsubscript{DFA} is decidable).
The language \text{EMPTY}\textsubscript{DFA} is decidable.

Proof. Our goal is to show \text{EMPTY}\textsubscript{DFA} is decidable and we will do so by constructing a decider for \text{EMPTY}\textsubscript{DFA}.

A decider for \text{EMPTY}\textsubscript{DFA} takes as input \langle D \rangle for some DFA $D = (Q, \Sigma, \delta, q_0, F)$, and needs to determine if $L(D) = \emptyset$. In other words, it needs to determine if there is any string that $D$ accepts. If we view the DFA as a directed graph,$^6$ then notice that the DFA accepts some string if and only if there is a directed path from $q_0$ to some state in $F$. Therefore, the following decider decides \text{EMPTY}\textsubscript{DFA} correctly.

$\begin{aligned}
D &: \text{DFA}. \\
M(\langle D \rangle): &\quad 1. \text{Build a directed graph from } \langle D \rangle. \\
&\quad 2. \text{Run a graph search algorithm starting from the starting state of } D. \\
&\quad 3. \text{If a node corresponding to an accepting state is reached, reject.} \\
&\quad 4. \text{Else, accept.}
\end{aligned}$

\hfill \square

Theorem 3.25 (EQ\textsubscript{DFA} is decidable).
The language \text{EQ}\textsubscript{DFA} is decidable.

Proof. Our goal is to show that \text{EQ}\textsubscript{DFA} is decidable. We will do so by constructing a decider for \text{EQ}\textsubscript{DFA}.

Our argument is going to use the fact that \text{EMPTY}\textsubscript{DFA} is decidable (Theorem 3.24 (EMPTY\textsubscript{DFA} is decidable)). In particular, the decider we present for \text{EQ}\textsubscript{DFA} will use the decider for \text{EMPTY}\textsubscript{DFA} as a subroutine. Let $M$ denote a decider TM for \text{EMPTY}\textsubscript{DFA}.

A decider for \text{EQ}\textsubscript{DFA} takes as input \langle $D_1$, $D_2$ \rangle, where $D_1$ and $D_2$ are DFAs. It needs to determine if $L(D_1) = L(D_2)$ (i.e. accept if $L(D_1) = L(D_2)$ and reject otherwise). We can determine if $L(D_1) = L(D_2)$ by looking at their symmetric difference$^7$

$$(L(D_1) \cap L(D_2)) \cup (\overline{L(D_1)} \cap L(D_2)).$$

Note that $L(D_1) = L(D_2)$ if and only if the symmetric difference is empty. Our decider for \text{EQ}\textsubscript{DFA} will construct a DFA $D$ such that $L(D) = (L(D_1) \cap L(D_2)) \cup (\overline{L(D_1)} \cap L(D_2))$, and then run $M(\langle D \rangle)$ to determine if $L(D) = \emptyset$. This then tells us if $L(D_1) = L(D_2)$.

To give a bit more detail, observe that given $D_1$ and $D_2$, we can

- construct DFAs $\overline{D_1}$ and $\overline{D_2}$ that accept $L(\overline{D_1})$ and $L(\overline{D_2})$ respectively (see Exercise 2.21 (Are regular languages closed under complementation?));
- construct a DFA that accepts $L(D_1) \cap \overline{L(D_2)}$ by using the (constructive) proof that regular languages are closed under the intersection operation$^8$.

$^6$Even though we have not formally defined the notion of a graph yet, we do assume you are familiar with the concept from a prerequisite course and that you have seen some simple graph search algorithms like Breadth-First Search or Depth-First Search.

$^7$The symmetric difference of sets $A$ and $B$ is the set of all elements that belong to either $A$ or $B$, but not both. In set notation, it corresponds to $(A \cap \overline{B}) \cup (\overline{A} \cap B)$.

$^8$The constructive proof gives us a way to construct the DFA accepting $L(D_1) \cap \overline{L(D_2)}$ given $D_1$ and $D_2$. 

33
• construct a DFA that accepts $L(D_1) \cap L(D_2)$ by using the proof that regular languages are closed under the intersection operation;

• construct a DFA, call it $D$, that accepts $(L(D_1) \cap \overline{L(D_2)}) \cup (\overline{L(D_1)} \cap L(D_2))$ by using the constructive proof that regular languages are closed under the union operation.

The decider for $\text{EQ}_{\text{DFA}}$ is as follows.

\begin{center}
\begin{tabular}{l}
$D_1$: DFA. $D_2$: DFA. \\
$M'(\langle D_1, D_2 \rangle)$: \\
1. Construct DFA $D$ as described above. \\
2. Run $M(\langle D \rangle)$. \\
3. If it accepts, accept. \\
4. If it rejects, reject.
\end{tabular}
\end{center}

By our discussion above, the decider works correctly. ☐

**IMPORTANT 3.26 (Decidability through reductions).**
Suppose $L$ and $K$ are two languages and $K$ is decidable. We say that solving $L$ reduces to solving $K$ if given a decider $M_K$ for $K$, we can construct a decider for $L$ that uses $M_K$ as a subroutine, thereby establishing $L$ is also decidable. For example, the proof of Theorem 3.25 (EQ$_{\text{DFA}}$ is decidable) shows that solving $\text{EQ}_{\text{DFA}}$ reduces to solving $\text{EMPTY}_{\text{DFA}}$. Reduction is a powerful tool to expand the landscape of decidable languages.

**Exercise 3.27 (Practice with decidability through reductions).**

(a) Let $L = \{ \langle D_1, D_2 \rangle : D_1$ and $D_2$ are DFAs with $L(D_1) \subset L(D_2)\}$. Show that $L$ is decidable.

(b) Let $K = \{ \langle D \rangle : D$ is a DFA that accepts $w^R$ whenever it accepts $w \}$, where $w^R$ denotes the reversal of $w$. Show that $K$ is decidable. For this question, you can use the fact given a DFA $D$, there is an algorithm to construct a DFA $D'$ such that $L(D') = L(D)^R = \{w^R : w \in L(D)\}$.

---

Note on notation: for sets $A$ and $B$, we write $A \subset B$ if $A \subseteq B$ and $A \neq B$. 

34
Quiz

1. True or false: A TM can have an infinite number of states.

2. True or false: It is possible that in the definition of a TM, $\Sigma = \Gamma$, where $\Sigma$ is the input alphabet, and $\Gamma$ is the tape alphabet.

3. True or false: On every input, any TM either accepts or rejects.

4. True or false: Consider a TM such that the starting state $q_0$ is also the accepting state $q_{\text{accept}}$. It is possible that this TM does not halt on some inputs.

5. Is the following statement true, false, or hard to determine with the knowledge we have so far? $\emptyset$ is decidable.

6. Is the following statement true, false, or hard to determine with the knowledge we have so far? $\Sigma^*$ is decidable.

7. True or false: $L \subseteq \Sigma^*$ is undecidable if and only if $\Sigma^* \setminus L$ is undecidable.

8. Is the following statement true, false, or hard to determine with the knowledge we have so far? The language $\{ \langle M \rangle : M$ is a TM with $L(M) = \emptyset \}$ is decidable.
Hints to Selected Exercises

Exercise 3.21 (Decidable language based on pi):
Case on the different possibilities that $L$ could be. For example, one option is that $L = \{3\}^*$, but there are other options too. Show that in all cases $L$ is decidable.

Exercise 3.27 (Practice with decidability through reductions):
Part (a): You may want to use a decider for EMPTY$_{DFA}$ and a decider for EQ$_{DFA}$.
Part (b): You may want to use a decider for EQ$_{DFA}$ together with the given fact in the description of the problem (i.e. given any DFA $D$, there is an algorithm to construct DFA $D'$ such that $L(D') = L(D)^R$).
Chapter 4

Countable and Uncountable Sets
PREAMBLE

Chapter structure:

- Section 8.1 (Basic Definitions)
  - Definition 4.1 (Injection, surjection, and bijection)
  - Theorem 4.2 (Relationships between different types of functions)
  - Definition 4.4 (Comparison of cardinality of sets)
  - Definition 4.6 (Countable and uncountable sets)
  - Theorem 4.7 (Characterization of countably infinite sets)

- Section 4.2 (Countable Sets)
  - Proposition 4.10 ($\mathbb{Z} \times \mathbb{Z}$ is countable)
  - Proposition 4.11 ($\mathbb{Q}$ is countable)
  - Proposition 4.12 ($\Sigma^*$ is countable)
  - Proposition 4.14 (The set of Turing machines is countable)
  - Proposition 4.15 (The set of polynomials with rational coefficients is countable)

- Section 4.3 (Uncountable Sets)
  - Theorem 4.17 (Cantor’s Theorem)
  - Corollary 4.18 ($\mathcal{P}(\mathbb{N})$ is uncountable)
  - Corollary 4.19 (The set of languages is uncountable)
  - Definition 4.20 ($\Sigma^\omega$)
  - Theorem 4.21 ($\{0, 1\}^\omega$ is uncountable)

Chapter goals:

In this chapter, we would like to remind you the concepts of countable and uncountable sets, as well as the general techniques involved in countability and uncountability proofs. Even though it may seem like we are diverging from the main discussion on Turing machines and decidability, we’ll see in the next chapter that the concepts in this chapter are intimately related to the concepts of decidability and undecidability. Countable and uncountable sets, together with the diagonalization proof technique for showing a set is uncountable, have major applications in proving the limits of computation.
4.1 Basic Definitions

Definition 4.1 (Injection, surjection, and bijection).
Let $A$ and $B$ be two (possibly infinite) sets.

- A function $f : A \rightarrow B$ is called **injective** if for any $a, a' \in A$ such that $a \neq a'$, we have $f(a) \neq f(a')$. We write $A \hookrightarrow B$ if there exists an injective function from $A$ to $B$.
- A function $f : A \rightarrow B$ is called **surjective** if for all $b \in B$, there exists an $a \in A$ such that $f(a) = b$. We write $A \twoheadrightarrow B$ if there exists a surjective function from $A$ to $B$.
- A function $f : A \rightarrow B$ is called **bijective** (or one-to-one correspondence) if it is both injective and surjective. We write $A \leftrightarrow B$ if there exists a bijective function from $A$ to $B$.

Theorem 4.2 (Relationships between different types of functions).
Let $A, B$, and $C$ be three (possibly infinite) sets. Then,

(a) $A \hookrightarrow B$ if and only if $B \twoheadrightarrow A$;
(b) if $A \hookrightarrow B$ and $B \twoheadrightarrow C$, then $A \hookrightarrow C$;
(c) $A \leftrightarrow B$ if and only if $A \hookrightarrow B$ and $B \twoheadrightarrow A$.

Exercise 4.3 (Exercise with injections and surjections).
Prove parts (a) and (b) of the above theorem.

Definition 4.4 (Comparison of cardinality of sets).
Let $A$ and $B$ be two (possibly infinite) sets.

- We write $|A| = |B|$ if $A \leftrightarrow B$.
- We write $|A| \leq |B|$ if $A \hookrightarrow B$, or equivalently, if $B \twoheadrightarrow A$.
- We write $|A| < |B|$ if it is not the case that $|A| \geq |B|$.

Note 4.5 (Sanity checks for comparing cardinality of sets).
Theorem 4.2 (Relationships between different types of functions) justifies the use of the notation $=, \leq, \geq, <$ and $>$. The properties we would expect to hold for this type of notation indeed do hold. For example, $|A| \leq |B|$ and $|B| \leq |A|$ if and only if $|A| = |B|$. If $|A| \leq |B| \leq |C|$, then $|A| \leq |C|$. If $|A| \leq |B| < |C|$, then $|A| < |C|$, and so on.

Definition 4.6 (Countable and uncountable sets).
- A set $A$ is called **countable** if $|A| \leq |\mathbb{N}|$.
- A set $A$ is called **countably infinite** if it is countable and infinite.
- A set $A$ is called **uncountable** if it is not countable, i.e. $|A| > |\mathbb{N}|$.

1Even though not explicitly stated, $|B| \geq |A|$ has the same meaning as $|A| \leq |B|$.
2Similar to above, $|B| > |A|$ has the same meaning as $|A| < |B|$.
Theorem 4.7 (Characterization of countably infinite sets).
A set \( A \) is countably infinite if and only if \( |A| = |\mathbb{N}| \).

Exercise 4.8 (Proof of the characterization of countably infinite sets).
Prove the above theorem.

Note 4.9 (Only two options for countable sets).
The above theorem implies that if \( A \) is countable, there are two options: either \( A \) is finite, or \( |A| = |\mathbb{N}| \).

4.2 Countable Sets

Proposition 4.10 (\( \mathbb{Z} \times \mathbb{Z} \) is countable).
The set \( \mathbb{Z} \times \mathbb{Z} \) is countable.

Proof. We want to show that \( \mathbb{Z} \times \mathbb{Z} \) is countable. We will do so by listing all the elements of \( \mathbb{Z} \times \mathbb{Z} \) such that every element eventually appears in the list. This implies that there is a surjective function \( f \) from \( \mathbb{N} \) to \( \mathbb{Z} \times \mathbb{Z} \): \( f(i) \) is defined to be the \( i \)'th element in the list. Since there is a surjection from \( \mathbb{N} \) to \( \mathbb{Z} \times \mathbb{Z} \), \( |\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}| \), and \( \mathbb{Z} \times \mathbb{Z} \) is countable.\(^3\)

We now describe how to list the elements of \( \mathbb{Z} \times \mathbb{Z} \). Consider the plot of \( \mathbb{Z} \times \mathbb{Z} \) on a 2-dimensional grid. Starting at \((0, 0)\) we list the elements of \( \mathbb{Z} \times \mathbb{Z} \) using a spiral shape, as shown below.

(The picture shows only a small part of the spiral.) Since we have a way to list all the elements such that every element eventually appears in the list, we are done.  

Proposition 4.11 (\( \mathbb{Q} \) is countable).
The set of rational numbers \( \mathbb{Q} \) is countable.

Note that it is not a requirement that we give an explicit formula for \( f(i) \). In fact, sometimes in such proofs, an explicit formula may not exist. This does not make the proof any less rigorous. Also note that this proof highlights the fact that the notion of countable is equivalent to the notion of listable, which can be informally defined as the ability to list the elements of the set so that every element eventually appears in the list.

\(^3\)Note that it is not a requirement that we give an explicit formula for \( f(i) \). In fact, sometimes in such proofs, an explicit formula may not exist. This does not make the proof any less rigorous. Also note that this proof highlights the fact that the notion of countable is equivalent to the notion of listable, which can be informally defined as the ability to list the elements of the set so that every element eventually appears in the list.
Proof. We want to show $\mathbb{Q}$ is countable. We will make use of the previous proposition to establish this. In particular, every element of $\mathbb{Q}$ can be written as a fraction $a/b$ where $a, b \in \mathbb{Z}$. In other words, there is a surjection from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Q}$ that maps $(a, b)$ to $a/b$ (if $b = 0$, map $(a, b)$ to say 0). This shows that $|\mathbb{Q}| \leq |\mathbb{Z} \times \mathbb{Z}|$. Since $\mathbb{Z} \times \mathbb{Z}$ is countable, i.e. $|\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}|$, $\mathbb{Q}$ is also countable, i.e. $|\mathbb{Q}| \leq |\mathbb{N}|$.

**Proposition 4.12** ($\Sigma^*$ is countable).

Let $\Sigma$ be a finite set. Then $\Sigma^*$ is countable.

**Proof.** Recall that $\Sigma^*$ denotes the set of all words/strings over the alphabet $\Sigma$ with finitely many symbols. We want to show $\Sigma^*$ is countable. We will do so by presenting a way to list all the elements of $\Sigma^*$ such that eventually all the elements appear in the list.

For each $n = 0, 1, 2, \ldots$, let $\Sigma^n$ denote the set of words in $\Sigma^*$ that have length exactly $n$. Note that $\Sigma^n$ is a finite set for each $n$, and $\Sigma^*$ is a union of these sets: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$. This gives us a way to list the elements of $\Sigma^*$ so that any element of $\Sigma^*$ eventually appears in the list. First list the elements of $\Sigma^0$, then list the elements of $\Sigma^1$, then list the elements of $\Sigma^2$, and so on. This way of listing the elements gives us a surjective function $f$ from $\mathbb{N}$ to $\Sigma^*$: $f(i)$ is defined to be the $i$'th element in the list. Since there is a surjection from $\mathbb{N}$ to $\Sigma^*$, $|\Sigma^*| \leq |\mathbb{N}|$, and $\Sigma^*$ is countable.

**IMPORTANT 4.13** (The CS method of showing countability).

One of the most powerful techniques for showing that a set $A$ is countable is to show that $A$ is encodable (i.e., there is an injective function $\text{Enc} : A \rightarrow \Sigma^*$ for some finite alphabet $\Sigma$). This is because if $A$ is encodable, then $|A| \leq |\Sigma^*| \leq |\mathbb{N}|$. So if the set $A$ is such that you can “write down” each element of $A$ using a finite number of symbols, then $A$ is countable. We call this method the “CS method” of showing countability.

**Proposition 4.14** (The set of Turing machines is countable).

The set of all Turing machines $\{M : M$ is a TM$\}$ is countable.

**Proof.** Let $T = \{M : M$ is a TM$\}$. We want to show that $T$ is countable. We will do so by using the CS method of showing a set is countable.

Given any Turing machine, there is a way to encode it with a finite length string because each component of the 7-tuple has a finite description. In particular, the mapping $M \mapsto \langle M \rangle$, where $\langle M \rangle \in \Sigma^*$, for some finite alphabet $\Sigma$, is an injective map (two distinct Turing machines cannot have the same encoding). Therefore $|T| \leq |\Sigma^*|$. And since $\Sigma^*$ is countable (Proposition 4.12 ($\Sigma^*$ is countable)), i.e., $|\Sigma^*| \leq |\mathbb{N}|$, the result follows.

**Proposition 4.15** (The set of polynomials with rational coefficients is countable).

The set of all polynomials in one variable with rational coefficients is countable.

**Proof.** Let $\mathbb{Q}[x]$ denote the set of all polynomials in one variable with rational coefficients. We want to show that $\mathbb{Q}[x]$ is countable and we will do so using the CS method. Let $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, /, x\}$. 

41
Then observe that every element of $\mathbb{Q}[x]$ can be written as a string over this alphabet. For example,

$$2x^3 - \frac{1}{34}x^2 + \frac{99}{100}x + \frac{22}{7}$$

represents the polynomial

$$2x^3 - \frac{1}{34}x^2 + \frac{99}{100}x + \frac{22}{7}.$$ 

This implies that there is a surjective map from $\Sigma^*$ to $\mathbb{Q}[x]$. And therefore $|\mathbb{Q}[x]| \leq |\Sigma^*|$. Since $\Sigma^*$ is countable, i.e. $|\Sigma^*| \leq |\mathbb{N}|$, $\mathbb{Q}[x]$ is also countable. 

**Exercise 4.16** (Practice with countability proofs).
Show that the following sets are countable.

(a) $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

(b) The set of all functions $f : A \to \mathbb{N}$, where $A$ is a finite set.

### 4.3 Uncountable Sets

**Theorem 4.17** (Cantor’s Theorem).
For any set $A$, $|\mathcal{P}(A)| > |A|$.

**Proof.** We want to show that for any (possibly infinite) set $A$, we have $|\mathcal{P}(A)| > |A|$. The proof that we present here is called the *diagonalization argument*. The proof is by contradiction. So assume that there is some set $A$ such that $|\mathcal{P}(A)| \leq |A|$. By definition, this means that there is a surjective function from $A$ to $\mathcal{P}(A)$. Let $f : A \to \mathcal{P}(A)$ be such a surjection. So for any $S \in \mathcal{P}(A)$, there exists an $s \in A$ such that $f(s) = S$. Now consider the set

$$S = \{a \in A : a \notin f(a)\}.$$ 

Since $S$ is a subset of $A$, $S \in \mathcal{P}(A)$. So there is an $s \in A$ such that $f(s) = S$. But then if $s \notin S$, by the definition of $S$, $s$ is in $f(s) = S$, which is a contradiction. If $s \in S$, then by the definition of $S$, $s$ is not in $f(s) = S$, which is also a contradiction. So either way, we get a contradiction, as desired.

**Corollary 4.18** ($\mathcal{P}(\mathbb{N})$ is uncountable).
The set $\mathcal{P}(\mathbb{N})$ is uncountable.

**Corollary 4.19** (The set of languages is uncountable).
Let $\Sigma$ be a finite set with $|\Sigma| > 0$. Then $\mathcal{P}(\Sigma^*)$ is uncountable.

**Proof.** We want to show that $\mathcal{P}(\Sigma^*)$ is uncountable, where $\Sigma$ is a non-empty finite set. For such a $\Sigma$, note that $\Sigma^*$ is a countably infinite set (Proposition 4.12 ($\Sigma^*$ is countable)). So by Theorem 4.7 (Characterization of countably infinite sets), we know $|\Sigma^*| = |\mathbb{N}|$. Theorem 4.17 (Cantor’s Theorem) implies that $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$. So we have $|\mathbb{N}| = |\Sigma^*| < |\mathcal{P}(\Sigma^*)|$, which shows, by the definition of uncountable sets, that $\mathcal{P}(\Sigma^*)$ is uncountable.
Definition 4.20 ($\Sigma^\infty$).
Let $\Sigma$ be some finite alphabet. We denote by $\Sigma^\infty$ the set of all infinite length words over the alphabet $\Sigma$. Note that $\Sigma^* \cap \Sigma^\infty = \emptyset$.

Theorem 4.21 ($\{0, 1\}^\infty$ is uncountable).
The set $\{0, 1\}^\infty$ is uncountable.

Proof. Our goal is to show that $\{0, 1\}^\infty$ is uncountable. One can prove this simply by observing that $\{0, 1\}^\infty \leftrightarrow \mathcal{P}(\mathbb{N})$, and using Corollary 4.18 ($\mathcal{P}(\mathbb{N})$ is uncountable). Here, we will give a direct proof using a diagonalization argument. The proof is by contradiction, so assume that $\{0, 1\}^\infty$ is countable. By definition, this means that $|\{0, 1\}^\infty| \leq |\mathbb{N}|$, i.e. there is a surjective map $f$ from $\mathbb{N}$ to $\{0, 1\}^\infty$. Consider the table in which the $i$'th row corresponds to $f(i)$.
Below is an example.

<table>
<thead>
<tr>
<th>$f(1)$</th>
<th>0 0 0 0 0 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(2)$</td>
<td>1 1 1 1 1 1 ...</td>
</tr>
<tr>
<td>$f(3)$</td>
<td>0 1 0 1 0 0 ...</td>
</tr>
<tr>
<td>$f(4)$</td>
<td>1 0 1 0 1 0 ...</td>
</tr>
<tr>
<td>$f(5)$</td>
<td>0 0 1 1 0 0 ...</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
</tbody>
</table>

(The elements in the diagonal are highlighted.) Using $f$, we construct an element $a$ of $\{0, 1\}^\infty$ as follows. If the $i$'th symbol of $f(i)$ is 1, then the $i$'th symbol of $a$ is defined to be 0. And if the $i$'th symbol of $f(i)$ is 0, then the $i$'th symbol of $a$ is defined to be 1. Notice that the $i$'th symbol of $f(i)$, for $i = 1, 2, 3, \ldots$ corresponds to the diagonal elements in the above table. So we are creating this element $a$ of $\{0, 1\}^\infty$ by taking the diagonal elements, and flipping their value.

Now notice that the way $a$ is constructed implies that it cannot appear as a row in this table. This is because $a$ differs from $f(1)$ in the first symbol, it differs from $f(2)$ in the second symbol, it differs from $f(3)$ in the third symbol, and so on. So it differs from every row of the table and hence cannot appear as a row in the table. This leads to the desired contradiction because $f$ is a surjective function, which means every element of $\{0, 1\}^\infty$, including $a$, must appear in the table.

Exercise 4.22 (Uncountable sets are closed under supersets).
Prove that if $A$ is uncountable and $A \subseteq B$, then $B$ is also uncountable.

IMPORTANT 4.23 (Uncountability through $\{0, 1\}^\infty$).
One of the most powerful techniques for showing that a set $A$ is uncountable is to show that $|A| \geq |\{0, 1\}^\infty|$, i.e. there is a surjection from $A$ to $\{0, 1\}^\infty$, or equivalently, there is an injection from $\{0, 1\}^\infty$ to $A$. One strategy for establishing this is to identify a subset of $A$ that is in one-to-one correspondence with $\{0, 1\}^\infty$. 43
Exercise 4.24 (Practice with uncountability proofs).
Show that the following sets are uncountable.

(a) The set of all bijective functions from $\mathbb{N}$ to $\mathbb{N}$.

(b) $\{x_1 x_2 x_3 \ldots \in \{1, 2\}^\infty : \text{for all } n \geq 1, \sum_{i=1}^n x_i \not\equiv 0 \mod 4\}$
Quiz

1. True or false: \( \{0, 1\}^* \cap \{0, 1\}^\infty = \emptyset \).
2. True or false: \( |\{0, 1, 2\}^*| = |\mathbb{Q} \times \mathbb{Q}| \).
3. True or false: \( |\mathcal{P}(\{0, 1\}^\infty)| = |\mathcal{P}(\mathcal{P}(\{0, 1\}^\infty))| \).
4. True or false: The set of all non-regular languages is countable.
5. True or false: There is a surjection from \( \{0, 1\}^\infty \) to \( \{0, 1, 2, 3\}^\infty \).
Hints to Selected Exercises

Exercise 4.8 (Proof of the characterization of countably infinite sets):
One of the directions should be relatively straightforward. For the other direction, given $A$ which is countable and infinite, try to find a way to order the elements of $A$. Then the bijection with $\mathbb{N}$ can be: $n$ maps to the $n$’th element of $A$ in the defined order.

Exercise 4.16 (Practice with countability proofs):
Use the CS method for both parts.

Exercise 4.24 (Practice with uncountability proofs):
In both cases, try to identify a subset of the set that is in one-to-one correspondence with $\{0, 1\}^\infty$. 
Chapter 5

Undecidable Languages
Chapter structure:

- Section 5.1 (Existence of Undecidable Languages)
  - Theorem 5.1 (Almost all languages are undecidable)

- Section 5.2 (Examples of Undecidable Languages)
  - Definition 5.3 (Halting problem)
  - Theorem 5.4 (Turing’s Theorem)
  - Definition 5.6 (Languages related to encodings of TMs)
  - Theorem 5.7 (ACCEPTS is undecidable)
  - Theorem 5.8 (EMPTY is undecidable)
  - Theorem 5.9 (EQ is undecidable)

- Section 5.3 (Undecidability Proofs by Reductions)
  - Theorem 5.14 (HALTS ≤ EMPTY)
  - Theorem 5.15 (EMPTY ≤ HALTS)

Chapter goals:

In this chapter, we formally prove that almost all languages are undecidable using the countability and uncountability concepts from the previous chapter. We also present (with proofs) several explicit examples of undecidable languages. By the Church-Turing Thesis, these results highlight the inherent limitations of computation.

An important tool in showing that a language is undecidable is the concept of a reduction. We present this technique in this chapter. Reductions play an extremely important role in computer science. In fact, we will revisit them in a future chapter (in the context of the famous P vs NP problem.)

Our goal in this chapter is for you to get comfortable with undecidability proofs and the concept of reductions, as they are at the core of the study of computation.
5.1 Existence of Undecidable Languages

**Theorem 5.1** (Almost all languages are undecidable).

Fix some alphabet $\Sigma$. There are languages $L \subseteq \Sigma^*$ that are not decidable.

**Proof.** To prove the result, we simply observe that the set of all languages is uncountable whereas the set of decidable languages is countable. First, consider the set of all languages. Since a language $L$ is defined to be a subset of $\Sigma^*$, the set of all languages is $P(\Sigma^*)$. By Corollary 4.19 (The set of languages is uncountable), we know that this set is uncountable. Now consider the set of all decidable languages, which we’ll denote by $D$. Let $T$ be the set of all TMs. By Proposition 4.14 (The set of Turing machines is countable), we know that $T$ is countable. Furthermore, the mapping $M \mapsto L(M)$ can be viewed as a surjection from $T$ to $D$ (if $M$ is not a decider, just map it to $\emptyset$). So $|D| \leq |T|$. Since $T$ is countable, this shows $D$ is countable and completes the proof. \[ \square \]

**Note 5.2** (Constructive vs non-constructive proofs).

The argument above is called non-constructive because it does not present an explicit undecidable language. A constructive argument would prove the undecidability of an explicit language. We present such an argument below (Theorem 5.4 (Turing’s Theorem)).

5.2 Examples of Undecidable Languages

**Definition 5.3** (Halting problem).

The halting problem is defined as the decision problem corresponding to the language $\text{HALTS} = \{ \langle M, x \rangle : M \text{ is a TM which halts on input } x \}$.

**Theorem 5.4** (Turing’s Theorem).

The language $\text{HALTS}$ is undecidable.

**Proof.** Our goal is to show that $\text{HALTS}$ is undecidable. The proof is by contradiction, so assume that $\text{HALTS}$ is decidable. By definition, this means that there is a decider TM, call it $M_{\text{HALTS}}$, that decides $\text{HALTS}$. We construct a new TM, which we’ll call $M_{\text{TURING}}$, that uses $M_{\text{HALTS}}$ as a subroutine. The description of $M_{\text{TURING}}$ is as follows:

\[
M: \text{TM.} \\
M_{\text{TURING}}(\langle M \rangle): \\
1. \text{Run } M_{\text{HALTS}}(\langle M, M \rangle). \\
2. \text{If it accepts, go into an infinite loop.} \\
3. \text{If it rejects, accept.}
\]

We get the desired contradiction once we consider what happens when we feed $M_{\text{TURING}}$ as input to itself, i.e. when we run $M_{\text{TURING}}(\langle M_{\text{TURING}} \rangle)$.

If $M_{\text{HALTS}}(\langle M_{\text{TURING}}, M_{\text{TURING}} \rangle)$ accepts, then $M_{\text{TURING}}(\langle M_{\text{TURING}} \rangle)$ is supposed to halt by the definition of $M_{\text{HALTS}}$. However, from the description of $M_{\text{TURING}}$ above, we see that it goes into an infinite loop. This is a contradiction. The other option is that $M_{\text{HALTS}}(\langle M_{\text{TURING}}, M_{\text{TURING}} \rangle)$ rejects. Then $M_{\text{TURING}}(\langle M_{\text{TURING}} \rangle)$ is supposed to lead to an infinite loop. But from the description of $M_{\text{TURING}}$ above, we see that it accepts, and therefore halts. This is a contradiction as well. \[ \square \]
Note 5.5 (Diagonalization argument for undecidability).
The above proof is called a diagonalization argument as it is very similar to the proof of Cantor’s theorem (Theorem 4.17 (Cantor’s Theorem)). As in the proof of Theorem 4.21 (\(\{0, 1\}^\infty\) is uncountable), we can present the above proof using a table and flipping its diagonal elements to get the desired contradiction. We do so below.

Reproof: The proof is by contradiction, so assume that HALTS is decidable. By definition, this means that there is a decider TM, call it \(M_{HALTS}\), that decides HALTS.

The set of all Turing machines is countable (Proposition 4.14 (The set of Turing machines is countable)). Let \(M_1, M_2, \ldots\) be a listing of all Turing machines in some arbitrary order. We now consider a table in which row \(i\) corresponds to \(M_i\) and column \(i\) corresponds to \(\langle M_i \rangle\). At entry corresponding to row \(i\) and column \(j\), we indicate whether \(M_i(\langle M_j \rangle)\) halts or loops forever. If it loops forever, we put a \(\infty\) symbol, and if it halts, we put \(H\).

We now create a new row in this table by taking the diagonal elements of the table and flipping their value (an \(\infty\) is flipped to an \(H\), and an \(H\) is flipped to an \(\infty\)). Notice that \(M_{TURING}\) constructed in the previous proof corresponds exactly to this new row we have created. We are able to construct \(M_{TURING}\) (and therefore the row it corresponds to) because we have a decider for HALTS. The contradiction is reached because on the one hand, \(M_{TURING}\) should appear as a row in the table since all the Turing machines are listed. On the other hand, the row of \(M_{TURING}\) differs from every row in the table (by construction, it differs from row \(i\) in the \(i\)'th column), and therefore cannot be in the table.

Definition 5.6 (Languages related to encodings of TMs).
We define the following languages:

\[
\begin{align*}
\text{ACCEPTS} &= \{\langle M, x \rangle : M \text{ is a TM that accepts the input } x\}, \\
\text{EMPTY} &= \{\langle M \rangle : M \text{ is a TM with } L(M) = \emptyset\}, \\
\text{EQ} &= \{\langle M_1, M_2 \rangle : M_1 \text{ and } M_2 \text{ are TMs with } L(M_1) = L(M_2)\}.
\end{align*}
\]

Theorem 5.7 (ACCEPTS is undecidable).
The language ACCEPTS is undecidable.

Proof. We want to show that ACCEPTS is undecidable. The proof is by contradiction, so assume ACCEPTS is decidable and let \(M_{ACCEPTS}\) be a decider for it. We will use this decider to come up with a decider for HALTS. Since HALTS is undecidable (Theorem 5.4 (Turing’s Theorem)), this argument will allow us to reach a contradiction.

Here is our decider for HALTS:
We now argue that this machine indeed decides HALTS. To do this, we’ll show that no matter what input is given to our machine, it always gives the correct answer.

First let’s assume we get any input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \in \text{HALTS} \). In this case our machine is supposed to accept. Since \( M(x) \) halts, we know that \( M(x) \) either ends up in the accepting state, or it ends up in the rejecting state. If it ends up in the accepting state, then \( M(\text{ACCEPTS}(\langle M, x \rangle)) \) accepts (on line 1 of our machine’s description), and so our program accepts and gives the correct answer on line 2. If on the other hand, \( M(x) \) ends up in the rejecting state, then \( M'(x) \) ends up in the accepting state. Therefore \( M(\text{ACCEPTS}(\langle M', x \rangle)) \) accepts (on line 4 of our machine’s description), and so our program accepts and gives the correct answer on line 5.

Now let’s assume we get any input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \notin \text{HALTS} \). In this case our machine is supposed to reject. Since \( M(x) \) does not halt, it never reaches the accepting or the rejecting state. By the construction of \( M' \), this also implies that \( M'(x) \) never reaches the accepting or the rejecting state. Therefore first \( M(\text{ACCEPTS}(\langle M, x \rangle)) \) (on line 1 of our machine’s description) will reject. And then \( M(\text{ACCEPTS}(\langle M', x \rangle)) \) (on line 4 of our machine’s description) will reject. Thus our program will reject as well, and give the correct answer on line 6.

We have shown that no matter what the input is, our machine gives the correct answer and decides HALTS. This is the desired contradiction and we conclude that \( \text{ACCEPTS} \) is undecidable.

\[ \square \]

**Theorem 5.8 (EMPTY is undecidable).**

The language \( \text{EMPTY} \) is undecidable.

**Proof.** We want to show that \( \text{EMPTY} \) is undecidable. The proof is by contradiction, so suppose \( \text{EMPTY} \) is decidable, and let \( M_{\text{EMPTY}} \) be a decider for it. Using this decider, we will construct a decider for \( \text{ACCEPTS} \). However, we know that \( \text{ACCEPTS} \) is undecidable (Theorem 5.7 (\( \text{ACCEPTS} \) is undecidable)), so this argument will allow us to reach a contradiction.

We construct a TM that decides \( \text{ACCEPTS} \) as follows.

\[
\begin{align*}
&M: \text{TM.} \ x: \text{string.} \\
&M_{\text{HALTS}}((M, x)):
1 & \text{Run } M_{\text{ACCEPTS}}((M, x)). \\
2 & \text{If it accepts, accept.} \\
3 & \text{Construct string } \langle M' \rangle \text{ by flipping the accept and reject states of } \langle M \rangle. \\
4 & \text{Run } M_{\text{ACCEPTS}}((M', x)). \\
5 & \text{If it accepts, accept.} \\
6 & \text{If it rejects, reject.}
\end{align*}
\]

We now argue that this machine indeed decides \( \text{HALTS} \). To do this, we’ll show that no matter what input is given to our machine, it always gives the correct answer.

First let’s assume we get any input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \in \text{HALTS} \). In this case our machine is supposed to accept. Since \( M(x) \) halts, we know that \( M(x) \) either ends up in the accepting state, or it ends up in the rejecting state. If it ends up in the accepting state, then \( M_{\text{ACCEPTS}}((M, x)) \) accepts (on line 1 of our machine’s description), and so our program accepts and gives the correct answer on line 2. If on the other hand, \( M(x) \) ends up in the rejecting state, then \( M'(x) \) ends up in the accepting state. Therefore \( M_{\text{ACCEPTS}}((M', x)) \) accepts (on line 4 of our machine’s description), and so our program accepts and gives the correct answer on line 5.

Now let’s assume we get any input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \notin \text{HALTS} \). In this case our machine is supposed to reject. Since \( M(x) \) does not halt, it never reaches the accepting or the rejecting state. By the construction of \( M' \), this also implies that \( M'(x) \) never reaches the accepting or the rejecting state. Therefore \( M_{\text{ACCEPTS}}((M, x)) \) (on line 1 of our machine’s description) will reject. And then \( M_{\text{ACCEPTS}}((M', x)) \) (on line 4 of our machine’s description) will reject. Thus our program will reject as well, and give the correct answer on line 6.

We have shown that no matter what the input is, our machine gives the correct answer and decides \( \text{HALTS} \). This is the desired contradiction and we conclude that \( \text{ACCEPTS} \) is undecidable.
We now argue that this machine indeed decides ACCEPTS. To do this, we’ll show that no matter what input is given to our machine, it always gives the correct answer.

First let’s assume we get an input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \in \text{ACCEPTS} \), i.e. \( x \in L(M) \). Then observe that \( L(M') = \Sigma^* \), because for any input \( y \), \( M'(y) \) will accept. When we run \( M_{\text{EMPTY}}((M')) \) on line 6, it rejects, and so our machine accepts and gives the correct answer.

Now assume that we get an input \( \langle M, x \rangle \) such that \( \langle M, x \rangle \notin \text{ACCEPTS} \), i.e. \( x \notin L(M) \). Then either \( M(x) \) rejects, or loops forever. If it rejects, then \( M'(y) \) rejects for any \( y \). If it loops forever, then \( M'(y) \) gets stuck on line 3 for any \( y \). In both cases, \( L(M') = \emptyset \). When we run \( M_{\text{EMPTY}}((M')) \) on line 6, it accepts, and so our machine rejects and gives the correct answer.

Our machine always gives the correct answer, so we are done. 

**Exercise 5.10 (Practice with undecidability proofs).**
Show that the following languages are undecidable.

(a) \( \text{EMPTY-HALTS} = \{ \langle M \rangle : M \text{ is a TM and } M(\epsilon) \text{ halts} \} \).

(b) \( \text{FINITE} = \{ \langle M \rangle : M \text{ is a TM that accepts finitely many strings} \} \).

### 5.3 Undecidability Proofs by Reductions

**Important 5.11 (Undecidability proofs by reduction).**
In the last section, we have used the same proof technique over and over again. It will be convenient to abstract away this technique and give it a name. Fix some alphabet \( \Sigma \). Let \( A \) and \( B \) be two languages. We say that \( A \) reduces to \( B \), written \( A \leq B \), if we are able to do the following: assume \( B \) is decidable (for
the sake of argument), and then show that \( A \) is decidable by using the decider for \( B \) as a black-box subroutine. Here the languages \( A \) and \( B \) may or may not be decidable to begin with. But observe that if \( A \leq B \) and \( B \) is decidable, then \( A \) is also decidable. Equivalently, taking the contrapositive, if \( A \leq B \) and \( A \) is undecidable, then \( B \) is also undecidable. So when \( A \leq B \), we think of \( B \) as being at least as hard as \( A \) with respect to decidability (which justifies using the less-than-or-equal-to sign).

Note 5.12 (Turing reductions).
In the literature, the above idea is formalized using the notion of a *Turing reduction* (with the corresponding symbol \( \leq_T \)). In order to define it formally, we need to define Turing machines that have access to an *oracle*. This level of detail will not be important for us, so we choose to omit the formal definition in our notes.

Note 5.13 (Already established reductions).
The proofs of Theorem 5.7 (*ACCEPTS* is undecidable), Theorem 5.8 (*EMPTY* is undecidable), and Theorem 5.9 (*EQ* is undecidable) correspond to \( \text{HALTS} \leq \text{ACCEPTS} \), \( \text{ACCEPTS} \leq \text{EMPTY} \) and \( \text{EMPTY} \leq \text{EQ} \) respectively.

Theorem 5.14 (\( \text{HALTS} \leq \text{EMPTY} \)).
\( \text{HALTS} \leq \text{EMPTY} \).

*Proof.* (This can be considered as an alternative proof of Theorem 5.8 (*EMPTY* is undecidable).) We want to show that deciding \( \text{HALTS} \) reduces to deciding \( \text{EMPTY} \). For this, we assume \( \text{EMPTY} \) is decidable. Let \( M_{\text{EMPTY}} \) be a decider for \( \text{EMPTY} \). We need to construct a TM that decides \( \text{HALTS} \). We do so now.

\[
M: \text{TM. } x: \text{string.}
M_{\text{HALTS}}((M,x)):
1. \text{Construct the following string, which we call } (M').
2. "M'(y):
3. \text{Run } M(x).
4. \text{Ignore the output and accept."
5. \text{Run } M_{\text{EMPTY}}((M')).
6. \text{If it accepts, reject.}
7. \text{If it rejects, accept.}
\]

We now argue that this machine indeed decides \( \text{HALTS} \). First consider an input \((M,x)\) such that \((M,x) \in \text{HALTS}\). Then \( L(M') = \Sigma^* \) since in this case \( M' \) accepts every string. So when we run \( M_{\text{EMPTY}}((M')) \) on line 8, it rejects, and our machine accepts and gives the correct answer.

Now consider an input \((M,x)\) such that \((M,x) \notin \text{HALTS}\). Then notice that whatever input is given to \( M' \), it gets stuck in an infinite loop when it runs \( M(x) \). Therefore \( L(M') = \emptyset \). So when we run \( M_{\text{EMPTY}}((M')) \) on line 8, it accepts, and our machine rejects and gives the correct answer. □
Theorem 5.15 (EMPTY \leq HALTS).

EMPTY \leq HALTS.

Proof. We want to show that deciding EMPTY reduces to deciding HALTS. For this, we assume HALTS is decidable. Let \(M_{HALTS}\) be a decider for HALTS. Using it, we need to construct a decider for EMPTY. We do so now.

\[
\begin{align*}
M & \colon TM. \\
M_{EMPTY}(\langle M \rangle) & : \\
& 1. \text{Construct the following string, which we call } \langle M' \rangle. \\
& 2. \text{”} M'(x): \\
& \quad \text{For } t = 1, 2, 3, \ldots: \\
& \quad \quad \text{For each } y \text{ with } |y| \leq t: \\
& \quad \quad \quad \text{Simulate } M(y) \text{ for at most } t \text{ steps.} \\
& \quad \quad \quad \text{If it accepts, accept."} \\
& 3. \text{Run } M_{HALTS}(\langle M', \epsilon \rangle). \\
& 4. \text{If it accepts, reject.} \\
& 5. \text{If it rejects, accept.}
\end{align*}
\]

We now argue that this machine indeed decides EMPTY. First consider an input \(\langle M \rangle\) such that \(\langle M \rangle \in \text{EMPTY}\). Observe that the only way \(M'\) halts is if \(M(y)\) accepts for some string \(y\). This cannot happen since \(L(M) = \emptyset\). So \(M'(x)\), for any \(x\), does not halt (note that \(M'\) ignores its input). This means that when we run \(M_{HALTS}(\langle M', \epsilon \rangle)\), it rejects, and so our decider above accepts, as desired.

Now consider an input \(\langle M \rangle\) such that \(\langle M \rangle \notin \text{EMPTY}\). This means that there is some word \(y\) such that \(M(y)\) accepts. Note that \(M'\), by construction, does an exhaustive search, so if such a \(y\) exists, then \(M'\) will eventually find it, and accept. So \(M'(x)\) halts for any \(x\). When we run \(M_{HALTS}(\langle M', \epsilon \rangle)\), it accepts, and our machine rejects and gives the correct answer. \(\Box\)

Exercise 5.16 (Practice with reduction definition).
Let \(A, B \subseteq \{0, 1\}^*\) be languages. Prove or disprove the following claims.

(a) If \(A \leq B\) then \(B \leq A\).

(b) If \(A \leq B\) and \(B\) is regular, then \(A\) is regular.

Exercise 5.17 (Practice with reduction proofs).
Show the following.

(a) ACCEPTS \leq HALTS.

(b) HALTS \leq EQ.
Quiz

1. True or false: For languages $K$ and $L$, if $K \leq L$, then $L$ is undecidable.
2. True or false: The set of undecidable languages is countable.
3. True or false: If a language $L$ is undecidable, then $L$ is infinite.
4. True or false: $\Sigma^* \leq \emptyset$.
5. True or false: $\text{HALTS} \leq \Sigma^*$. 
Hints to Selected Exercises

Exercise 5.10 (Practice with undecidability proofs):
As usual, the proofs will be by contradiction. In both cases, show how to decide HALTS given a
decider for the language in question.

Exercise 5.16 (Practice with reduction definition):
Both parts are false.
Chapter 6

Time Complexity
Chapter goals:

So far, we have formally defined what a computational/decision problem is, what an algorithm is, and saw that most (decision) problems are undecidable. We also saw some explicit and interesting examples of undecidable problems. Nevertheless, it turns out that many problems that we care about are actually decidable. So the next natural thing to study is the computational complexity of problems. If a problem is decidable, but the most efficient algorithm solving it takes vigintillion computational steps even for reasonably sized inputs, then practically speaking, that problem is still undecidable.

Our goal in this chapter is to introduce the right language to express and analyze the running time of algorithms, which in return helps us determine which problems are practically decidable, and which problems are (or seem to be) practically undecidable.

Most of the ideas in this chapter will probably be familiar to you at some level as most introductory computer science courses do talk about (asymptotic) running time of algorithms. Nevertheless, it is important to specify some of the details that we will be using as we slowly start approaching the famous P vs NP question, which is a question about the computational complexity of problems.
6.1 Big-O, Big-Omega and Theta

**Definition 6.1 (Big-O).**
For \( f : \mathbb{R}^+ \to \mathbb{R}^+ \) and \( g : \mathbb{R}^+ \to \mathbb{R}^+ \), we write \( f(n) = O(g(n)) \) if there exist constants \( C > 0 \) and \( n_0 > 0 \) such that for all \( n \geq n_0 \),
\[
f(n) \leq Cg(n).
\]
In this case, we say that \( f(n) \) is big-O of \( g(n) \).

**Exercise 6.2 (Practice with big-O).**
Show that \( 3n^2 + 10n + 30 \) is \( O(n^2) \).

**Definition 6.3 (Big-Omega).**
For \( f : \mathbb{R}^+ \to \mathbb{R}^+ \) and \( g : \mathbb{R}^+ \to \mathbb{R}^+ \), we write \( f(n) = \Omega(g(n)) \) if there exist constants \( c > 0 \) and \( n_0 > 0 \) such that for all \( n \geq n_0 \),
\[
f(n) \geq cg(n).
\]
In this case, we say that \( f(n) \) is big-Omega of \( g(n) \).

**Exercise 6.4 (Practice with big-Omega).**
Show that \( n! \) is \( \Omega(n^n) \).

**Definition 6.5 (Theta).**
For \( f : \mathbb{R}^+ \to \mathbb{R}^+ \) and \( g : \mathbb{R}^+ \to \mathbb{R}^+ \), we write \( f(n) = \Theta(g(n)) \) if
\[
f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n)).
\]
This is equivalent to saying that there exists constants \( c, C, n_0 > 0 \) such that for all \( n \geq n_0 \),
\[
 cn \leq f(n) \leq Cn.
\]
In this case, we say that \( f(n) \) is Theta of \( g(n) \).

**Proposition 6.6 (Logarithms in different bases).**
For any constant \( b > 1 \),
\[
 \log_b n = \Theta(\log n).
\]

**Proof.** It is well known that \( \log_b n = \frac{\log_n n}{\log_n b} \). In particular \( \log_b n = \frac{\log_{10} n}{\log_{10} b} \). Then taking \( c = C = \frac{1}{\log_{10} 2} \) and \( n_0 = 1 \), we see that \( c \log_2 n \leq \log_b n \leq C \log_2 n \) for all \( n \geq n_0 \). Therefore \( \log_b n = \Theta(\log_2 n) \).

**Note 6.7 (Does the base of a logarithm matter?).**
Since the base of a logarithm only changes the value of the log function by a constant factor, it is usually not relevant in big-O, big-Omega or Theta notation. So most of the time, when you see a log function present inside \( O(\cdot) \), \( \Omega(\cdot) \), or \( \Theta(\cdot) \), the base will be ignored. E.g., instead of writing \( \ln n = \Theta(\log_2 n) \), we actually write \( \ln n = \Theta(\log_2 n) \). That being said, if the log appears in the exponent, the base matters. For example, \( n^{\log_2 5} \) is asymptotically different from \( n^{\log_3 5} \).

**Exercise 6.8 (Practice with Theta).**
Show that \( \log_2(n!) = \Theta(n \log n) \).

\(^1\)The reason we don’t call it big-Theta is that there is no separate notion of little-theta, whereas little-o \( o(\cdot) \) and little-omega \( \omega(\cdot) \) have meanings separate from big-O and big-Omega. We don’t cover little-o and little-omega in this course.
6.2 Worst-Case Running Time of Algorithms

**Definition 6.9** (Worst-case running time of an algorithm).
Suppose we are using some computational model in which what constitutes a step in an algorithm is understood. Suppose also that for any input $x$, we have an explicit definition of its length. The worst-case running time of an algorithm $A$ is a function $T_A : \mathbb{N} \rightarrow \mathbb{N}$ defined by

$$T_A(n) = \max_{\text{instances/inputs } x \text{ of length } n} \text{number of steps } A \text{ takes on input } x.$$

We drop the subscript $A$ and just write $T(n)$ when $A$ is clear from the context.

**IMPORTANT 6.10** (Input length).
We use $n$ to denote the input length. Unless specified otherwise, $n$ is defined to be the number of bits in a reasonable binary encoding of the input. It is also common to define $n$ in other ways. For example, if the input is an array or a list, $n$ can denote the number of elements.

**IMPORTANT 6.11** (Our model when measuring running time).
In the Turing machine model, a step in the computation corresponds to one application of the transition function of the machine. However, when measuring running time, often we will not be considering the Turing machine model.

If we don’t specify a particular computational model, by default, our model will be closely related to the Random Access Machine (RAM) model. Compared to TMs, this model aligns better with the architecture of the computers we use today. We will not define this model formally, but instead point out two properties of importance.

First, given a string or an array, accessing any index counts as 1 step.

Second, arithmetic operations count as 1 step as long as the numbers involved are “small”. We say that a number $y$ is small if it can be upper bounded by a polynomial in $n$, the input length. That is, $y$ is small if there is some constant $k$ such that $y$ is $O(n^k)$. As an example, suppose we have an algorithm $A$ that contains a line like $x = y + z$, where $y$ and $z$ are variables that hold integer values. Then we can count this line as a single step if $y$ and $z$ are both small. Note that whether a number is small or not is determined by the length of the input to the algorithm $A$.

We say that a number is large, if it is not small, i.e., if it cannot be upper bounded by a polynomial in $n$. In cases where we are doing arithmetic operations involving large numbers, we have to consider the algorithms used for the arithmetic operations and figure out their running time. For example, in the line $x = y + z$, if $y$ or $z$ is a large number, we need to specify what algorithm is being used to do the addition and what its running time is. A large number should be treated as a string of digits/characters. Arithmetic operations on large numbers should be treated as string manipulation operations and their running time should be figured out accordingly.

**Note 6.12** (Asymptotic complexity).
The expression of the running time of an algorithm using big-O, big-Omega or Theta notation is referred to as asymptotic complexity estimate of the algorithm.
Definition 6.13 (Names for common growth rates).

- **Constant time**: \( T(n) = O(1) \).
- **Logarithmic time**: \( T(n) = O(\log n) \).
- **Linear time**: \( T(n) = O(n) \).
- **Quadratic time**: \( T(n) = O(n^2) \).
- **Polynomial time**: \( T(n) = O(n^k) \) for some constant \( k > 0 \).
- **Exponential time**: \( T(n) = O(2^n) \) for some constant \( k > 0 \).

Exercise 6.14 (Composing polynomial time algorithms).
Suppose that we have an algorithm \( A \) that runs another algorithm \( A' \) once as a subroutine. We know that the running time of \( A \) is \( O(n^k) \), \( k \geq 1 \), and the work done by \( A \) is \( O(n^i) \), \( i \geq 1 \), if we ignore the subroutine \( A' \) (i.e., we don’t count the steps taken by \( A' \)). What kind of upper bound can we give for the total running-time of \( A \) (which includes the work done by \( A' \))?

Note 6.15 (Intrinsic complexity).
The intrinsic complexity of a computational problem refers to the asymptotic time complexity of the most efficient algorithm that computes the problem.\(^2\)

Proposition 6.16 (Intrinsic complexity of \( \{0^k1^k : k \in \mathbb{N}\} \)).
The intrinsic complexity of \( L = \{0^k1^k : k \in \mathbb{N}\} \) is \( \Theta(n) \).

Proof. We want to show that the intrinsic complexity of \( L = \{0^k1^k : k \in \mathbb{N}\} \) is \( \Theta(n) \). The proof has two parts. First, we need to argue that the intrinsic complexity is \( O(n) \). Then, we need to argue that the intrinsic complexity is \( \Omega(n) \).

To show that \( L \) has intrinsic complexity \( O(n) \), all we need to do is present an algorithm that decides \( L \) in time \( O(n) \). We leave this as an exercise to the reader.

To show that \( L \) has intrinsic complexity \( \Omega(n) \), we show that no matter what algorithm is used to decide \( L \), the number of steps it takes must be at least \( n \). We prove this by contradiction, so assume that there is some algorithm \( A \) that decides \( L \) using \( n-1 \) steps or less. Consider the input \( x = 0^k1^k \) (where \( n = 2k \)). Since \( A \) uses at most \( n-1 \) steps, there is at least one index \( j \) with the property that \( A \) does not access \( x[j] \). Let \( x' \) be the input that is the same as \( x \), except the \( j \)'th coordinate is reversed. Since \( A \) does not access the \( j \)'th coordinate, it has no way of distinguishing between \( x \) and \( x' \). In other words, \( A \) behaves exactly the same when the input is \( x \) or \( x' \). But this contradicts the assumption that \( A \) correctly decides \( L \) because \( A \) should accept \( x \) and reject \( x' \). \( \square \)

Exercise 6.17 (TM complexity of \( \{0^k1^k : k \in \mathbb{N}\} \)).
In the TM model, a step corresponds to one application of the transition function. Show that \( L = \{0^k1^k : k \in \mathbb{N}\} \) can be decided by a TM in time \( O(n \log n) \).
Is this statement directly implied by Proposition 6.16 (Intrinsic complexity of \( \{0^k1^k : k \in \mathbb{N}\} \))?\(^2\)

Exercise 6.18 (Is polynomial time decidability closed under concatenation?).
Assume the languages \( L_1 \) and \( L_2 \) are decidable in polynomial time. Prove or give a counter-example: \( L_1L_2 \) is decidable in polynomial time.

---

\(^2\)For certain computational problems, the intrinsic complexity may not be well-defined. In some cases, there can be a sequence of algorithms that solve a certain computational problem, where each algorithm in the sequence is asymptotically more efficient than the one before.
### 6.3 Complexity of Algorithms with Integer Inputs

**IMPORTANT 6.19** (Integer inputs are large numbers).
Given a computational problem with an integer input $x$, notice that $x$ is a large number (if $x$ is $n$ bits long, its value can be about $2^n$, so it cannot be upper bounded by a polynomial in $n$). Therefore arithmetic operations involving $x$ cannot be treated as 1-step operations. Computational problems with integer input(s) are the most common examples in which we have to deal with large numbers, and in these situations, one should be particularly careful about analyzing running time.

**Definition 6.20** (Integer addition and integer multiplication problems).
In the integer addition problem, we are given two $n$-bit numbers $x$ and $y$, and the output is their sum $x + y$. In the integer multiplication problem, we are given two $n$-bit numbers $x$ and $y$, and the output is their product $xy$.

**Note 6.21** (Algorithms for integer addition).
Consider the following algorithm for the integer addition problem (we’ll assume the inputs are natural numbers for simplicity).

```plaintext
x: natural number. y: natural number.
Addition((x, y)):
1. For $i = 1$ to $x$:
   2. $y = y + 1$.
3. Return $y$.
```

This algorithm has a loop that repeats $x$ many times. Since $x$ is an $n$-bit number, the worst-case complexity of this algorithm is $\Omega(2^n)$.

In comparison, the following well-known algorithm for integer addition has time complexity $O(n)$.

```plaintext
x: natural number. y: natural number.
Addition((x, y)):
1. carry = 0.
2. For $i = 0$ to $n - 1$:
   3. columnSum = $x[i] + y[i] + carry$.
   4. $z[i] = \text{columnSum} \mod 2$.
   5. carry = (columnSum - $z[i]) / 2$.
6. $z[n] = \text{carry}$.
7. Return $z$.
```

Note that the arithmetic operations inside the loop are all $O(1)$ time since the numbers involved are all bounded (i.e., their values do not depend on $n$). Since the loop repeats $n$ times, the overall complexity is $O(n)$.

It is easy to see that the intrinsic complexity of integer addition is $\Omega(n)$ since it takes at least $n$ steps to write down the output, which is either $n$ or $n + 1$ bits long. Therefore we can conclude that the intrinsic complexity of integer addition is $\Theta(n)$. The same is true for integer subtraction.
Exercise 6.22 (Running time of the factoring problem).
Consider the following problem: Given as input a positive integer $N$, output a non-trivial factor\(^3\) of $N$ if one exists, and output False otherwise. Give a lower bound using the $\Omega(\cdot)$ notation for the running-time of the following algorithm solving the problem:

\[
\begin{align*}
N: & \text{ natural number.} \\
\text{Non-Trivial-Factor}((N)): & \\
1 \text{ For } i = 2 \text{ to } N - 1: & \\
2 \quad \text{If } N \% i == 0: & \text{ Return } i. \\
3 \quad \text{Return False.}
\end{align*}
\]

Note 6.23 (Grade-school algorithms for multiplication and division).
The grade-school algorithms for the integer multiplication and division problems have time complexity $O(n^2)$. You may use these facts in your arguments without proof.

Note 6.24 (The best-known multiplication algorithm).
The best known multiplication algorithm has running time that is extremely close to $O(n \log n)$. So there are much smarter ways to do multiplication than the grade-school algorithm.

Exercise 6.25 (251st root).
Consider the following computational problem. Given as input a number $A \in \mathbb{N}$, output $\lfloor A^{1/251} \rfloor$. Determine whether this problem can be computed in worst-case polynomial-time, i.e. $O(n^k)$ time for some constant $k$, where $n$ denotes the number of bits in the binary representation of the input $A$. If you think the problem can be solved in polynomial time, give an algorithm in pseudocode, explain briefly why it gives the correct answer, and argue carefully why the running time is polynomial. If you think the problem cannot be solved in polynomial time, then provide a proof.

\[^3\text{A non-trivial factor is a factor that is not equal to 1 or the number itself.}\]
Quiz

1. True or false: \( n^{\log_2 5} = \Theta(n^{\log_3 5}) \).

2. True or false: \( n^{\log_2 n} = \Omega(n^{1.5251}) \).

3. True or false: \( f(n) = O(g(n)) \) if and only if \( g(n) = \Omega(f(n)) \).

4. True or false: Let \( \Sigma = \{0, 1\} \) and let \( L = \{0^n : n \in \mathbb{N}^+\} \). There is a Turing Machine \( A \) deciding \( L \) whose running time \( T_A \) satisfies “\( T_A(n) \) is \( O(n) \)”.

5. True or false: Continuing previous question, every Turing Machine \( B \) that decides \( L \) has running time \( T_B \) satisfying “\( T_B(n) \) is \( \Omega(n) \)”.

6. What is the running time of the following algorithm in terms of \( n \), the input length, using the big-O notation?

```python
def isPrime(N):
    if (N < 2):
        return False
    if (N == 2):
        return True
    if (N mod 2 == 0):
        return False
    maxFactor = ceiling(N**0.5)
    for factor in range(3,maxFactor+1,2):
        if (N mod factor == 0):
            return False
    return True
```
Hints to Selected Exercises

Exercise 6.17 (TM complexity of \(\{0^k1^k : k \in \mathbb{N}\}\)):
Think about \(\log n\) iterations with each iteration being \(O(n)\) steps.

Exercise 6.18 (Is polynomial time decidability closed under concatenation?)
The statement is true.

Exercise 6.22 (Running time of the factoring problem):
It is not a polynomial-time algorithm.

Exercise 6.25 (251st root):
Binary search.
Chapter 7

The Science of Cutting Cake
PREAMBLE

Chapter structure:

- Section 7.1 (The Problem and the Model)
  - Definition 7.1 (Cake cutting problem)
  - Proposition 7.2 (An observation about the $V_i$'s)
  - Proposition 7.3 (Envy-freeness implies proportionality)
  - Definition 7.4 (The Robertson-Webb model)
- Section 7.2 (Cake Cutting Algorithms in the Robertson-Webb Model)
  - Proposition 7.5 (Cut and Choose algorithm for 2 players)
  - Theorem 7.6 (Dubins-Spanier Algorithm)
  - Theorem 7.8 (Even-Paz Algorithm)
  - Theorem 7.10 (Edmonds-Pruhs Theorem)

Chapter goals:

In this chapter, we turn our attention to a different computational model related to an important social concern about how to fairly allocate divisible resources under some constraints. There are a couple of goals of this chapter. First, it provides a completely new model of computation with its own rules on what counts a computational step and what the input length is. As such, we hope that this will expand your horizon on what we can view as a computational process, and how we can measure its complexity. Second, this chapter presents one of the many real-world applications of theoretical computer science. Finding fair ways of dividing limited resources is very important, and studying this problem mathematically rigorously provides provable solutions.

Applications:

- [http://procaccia.info/papers/cakesurvey.cacm.pdf](http://procaccia.info/papers/cakesurvey.cacm.pdf)
- [http://www.spliddit.org](http://www.spliddit.org)
7.1 The Problem and the Model

Definition 7.1 (Cake cutting problem). We refer to the interval \([0, 1] \subset \mathbb{R}\) as the cake, and the set \(N = \{1, 2, \ldots, n\}\) as the set of players. A piece of cake is any set \(X \subseteq [0, 1]\) which is a finite union of disjoint intervals. Let \(\mathcal{X}\) denote the set of all possible pieces of cake. Each player \(i \in N\) has a valuation function \(V_i : \mathcal{X} \to \mathbb{R}\) that satisfies the following 4 properties.

- **Normalized**: \(V_i([0, 1]) = 1\).
- **Non-negative**: For any \(X \in \mathcal{X}\), \(V_i(X) \geq 0\).
- **Additive**: For \(X, Y \in \mathcal{X}\) with \(X \cap Y = \emptyset\), \(V_i(X \cup Y) = V_i(X) + V_i(Y)\).
- **Divisible**: For every interval \(I \subseteq [0, 1]\) and \(0 \leq \lambda \leq 1\), there exists a subinterval \(I' \subseteq I\) such that \(V_i(I') = \lambda V_i(I)\).

The goal is to find an allocation \(A_1, A_2, \ldots, A_n\), where for each \(i\), \(A_i\) is a piece of cake allocated to player \(i\). The allocation is assumed to be a partition of the cake \([0, 1]\), i.e., the \(A_i\)'s are disjoint and their union is \([0, 1]\). There are 2 properties desired about the allocation:

- **Proportionality**: For all \(i \in N\), \(V_i(A_i) \geq 1/n\).
- **Envy-Freeness**: For all \(i, j \in N\), \(V_i(A_i) \geq V_i(A_j)\).

Proposition 7.2 (An observation about the \(V_i\)'s).
Let \(A_1, \ldots, A_n\) be an allocation in the cake cutting problem. Then for each player \(i\), we have \(\sum_{j \in N} V_i(A_j) = 1\).

**Proof.** Given any player \(i\), our goal is to show that \(\sum_{j \in N} V_i(A_j) = 1\). This will follow from the additivity and normality properties of the valuation functions.

First, recall that the \(A_i\)'s form a partition of \([0, 1]\). So
\[
A_1 \cup A_2 \cup \cdots \cup A_n = [0, 1],
\]
and the \(A_i\)'s are pairwise disjoint. Now take an arbitrary player \(i\). By the normality property, we know \(V_i([0, 1]) = 1\). Combining this with the additivity property, we have
\[
1 = V_i([0, 1]) = V_i(A_1 \cup A_2 \cup \cdots \cup A_n) = V_i(A_1) + V_i(A_2) + \cdots + V_i(A_n),
\]
i.e., \(\sum_{j \in N} V_i(A_j) = 1\).

Proposition 7.3 (Envy-freeness implies proportionality).
If an allocation is envy-free, then it is proportional.

**Proof.** Let’s assume we have an allocation \(A_1, \ldots, A_n\) that is envy-free. We want to show that it must also be proportional. Take an arbitrary player \(i\). By the previous proposition, we have \(\sum_{j \in N} V_i(A_j) = 1\). Therefore, there must be \(k \in N\) such that \(V_i(A_k) \geq 1/n\) (otherwise the sum could not be 1). The envy-freeness property implies that \(V_i(A_i) \geq V_i(A_k)\), and so \(V_i(A_i) \geq 1/n\). This establishes that the allocation must be proportional.
Definition 7.4 (The Robertson-Webb model).
We use the Robertson-Webb model to express cake cutting algorithms and measure their running times. In this model, the input size is considered to be the number of players \( n \). There is a referee who is allowed to make two types of queries to the players:

- \( \text{Eval}_i(x, y) \), which returns \( V_i([x, y]) \),
- \( \text{Cut}_i(x, \alpha) \), which returns \( y \) such that \( V_i([x, y]) = \alpha \). (If no such \( y \) exists, it returns “None”.)

The referee follows an algorithm/strategy and chooses the queries that she wants to make. What the referee chooses as a query depends only on the results of the queries she has made before. At the end, she decides on an allocation \( A_1, A_2, \ldots, A_n \), and the allocation depends only on the outcomes of the queries. The time complexity of the algorithm, \( T(n) \), is the number of queries she makes for \( n \) players and the worst possible \( V_i \)’s. So

\[
T(n) = \max_{(V_1, \ldots, V_n)} \text{number of queries when the valuations are } (V_1, \ldots, V_n).
\]

7.2 Cake Cutting Algorithms in the Robertson-Webb Model

Proposition 7.5 (Cut and Choose algorithm for 2 players).
When \( n = 2 \), there is always an allocation that is proportional and envy-free.

Proof. Given \( n = 2 \) players, we will describe a way to allocate the cake so that it is proportional and envy-free. We first describe how to find the allocation. We then argue why that allocation is envy-free and proportional.

We can describe how the allocation is found in the following way. The first player marks a point \( y \) in the cake so that \( V_1([0, y]) = V_1([y, 1]) = 1/2 \) (this can be done because of the divisibility property). Then player 2 chooses the piece (among \([0, y]\) and \([y, 1]\)) that he values more. The remaining piece is what player 1 gets. In the Robertson-Webb model, this algorithm corresponds to the following. The referee first queries \( \text{Cut}_1(0, 1/2) \). Say this returns the value \( y \). Then the referee queries \( \text{Eval}_2(0, y) \) and \( \text{Eval}_2(y, 1) \). Whichever gives the larger value, referee assigns that piece to player 2. The remaining piece is assigned to player 1.

The allocation is envy-free: From player 1’s perspective, both players get a piece of the cake of value 1/2. Therefore \( V_1(A_1) \geq V_1(A_2) \) is satisfied. From player 2’s perspective, since he gets to choose the piece of larger value to him, \( V_2(A_2) \geq V_2(A_1) \) is satisfied. (Also note that we must have \( V_2(A_2) > 1/2 \).)

The allocation is proportional: It is not hard to see that the algorithm is proportional since each player gets a piece of value at least 1/2. \( \square \)

Theorem 7.6 (Dubins-Spanier Algorithm).
There is an algorithm of time complexity \( \Theta(n^2) \) that produces an allocation for the cake cutting problem that satisfies the proportionality property.

\footnote{In fact, just querying \( \text{Eval}_2(0, y) \) is enough.}

\footnote{It is common to describe a cake cutting algorithm in terms of what players do to agree on an allocation. In the Robertson-Webb model we have described, this would correspond to a referee applying Eval and Cut queries to determine the allocation. The two points of views are equivalent as long as the actions of the players can be described using Eval and Cut queries.}
Proof. Our goal is to describe a cake cutting algorithm with \( \Theta(n^2) \) complexity that produces a proportional allocation. We first describe the algorithm. We then argue that it indeed produces a proportional allocation. Finally, we show that its complexity is \( \Theta(n^2) \).

The algorithm is as follows. The referee first makes \( n \) queries: Cut \(_i\)(0, 1/n) for all \( i \). She computes the minimum among these values, which we’ll denote by \( y \). Let’s assume \( j \) is the player that corresponds to the minimum value. Then the referee assigns \( A_j = [0, y] \). So player \( j \) gets a piece that she values at 1/n. After this, we remove player \( j \), and repeat the process on the remaining cake. So in the next stage, the referee makes \( n-1 \) queries, Cut \(_i\)(\( y \), 1/n) for \( i \neq j \), figures out the player corresponding to the minimum value, and assigns her the corresponding piece of the cake, which she values at 1/n. This repeats until there is one player left. The last player gets the piece that is left.\(^3\)

We have to show that the algorithm’s time complexity is \( \Theta(n^2) \) and that it produces a proportional allocation. First we show that the allocation is proportional. Notice that if the queries that the referee makes never return “None”, then at each iteration, until one player is left, the player \( j \) who is removed is assigned \( A_j \) such that \( V_j(A_j) = 1/n \). So it suffices to argue that:

(i) the queries never return “None”,

(ii) the last player, call it \( \ell \), gets \( A_\ell \) such that \( V_\ell(A_\ell) \geq 1/n \).

To show (i), assume we have just completed iteration \( k \), where \( k \in \{1, 2, \ldots, n-1\} \). Let \( j \) be an arbitrary player who has not been removed yet. The important observation is that all the pieces that have been removed so far have value at most 1/n to player \( j \) (take a moment to verify this). So the cake remaining after iteration \( k \) has value at least \( 1 - (k/n) \geq 1/n \) for player \( j \). This argument holds for any \( k \in \{1, 2, \ldots, n-1\} \) and any player \( j \) that remains after iteration \( k \). So the queries never return “None”. Part (ii) actually follows from the same argument. The cake remaining after iteration \( n-1 \) has value at least \( 1 - (n-1)/n = 1/n \) for the last player. This completes the proof that the allocation is proportional.

Now we show that the time complexity is \( \Theta(n^2) \). To do this, we’ll first argue that the number of queries is \( O(n^2) \), and then argue that it is \( \Omega(n^2) \). Note that the algorithm has \( n \) iterations, and at iteration \( i \), it makes \( n + 1 - i \) queries. There is one exception, which is the last iteration when only one player is left. In that case, we don’t make any queries. So the total number of queries is

\[
n + (n - 1) + (n - 2) + \cdots + 2.
\]

We can upper bound this as follows:

\[
n + (n - 1) + (n - 2) + \cdots + 2 \leq \underbrace{n + n + \cdots + n}_{n \text{ times}} = n^2.
\]

This implies that the number of queries is \( O(n^2) \). We can also lower bound the number of queries by lower bounding the first \( n/2 \) terms in the sum by \( n/2 \):

\[
n + (n - 1) + (n - 2) + \cdots + 2 \geq \underbrace{n/2 + n/2 + \cdots + n/2}_{n/2 \text{ times}} = \frac{n^2}{4}.
\]

This implies that the number of queries is \( \Omega(n^2) \). Hence, the number of queries is \( \Theta(n^2) \).

\(^3\)Note that it is perfectly fine to describe an algorithm in a paragraph as long as you explain clearly what the algorithm does. A pseudocode is not required.
Exercise 7.7 (Practice with cutting cake).
Design a cake cutting algorithm for a set of players \( N = \{1, \ldots, n\} \) that finds an allocation \( A \) with the property that there exists a permutation/bijection \( \pi_A : N \rightarrow N \) such that for all \( i \in N \), \( V_i(A_i) \geq \frac{1}{2^k} \). In words, there is an order on the players such that the first player has value at least 1/2 for her piece, the second player has value at least 1/4, and so on. The complexity of your algorithm in the Robertson-Webb model should be \( O(n^2) \).

Theorem 7.8 (Even-Paz Algorithm).
Assume \( n \) is a power of 2, i.e., \( n = 2^t \) for some \( t \in \mathbb{N} \). There is an algorithm of time complexity \( \Theta(n \log n) \) that produces an allocation for the cake cutting problem that satisfies the proportionality property.

Proof. Our goal is to present a cake cutting algorithm with \( \Theta(n \log n) \) complexity that produces a proportional allocation. The assumption that \( n \) is a power of 2 is there for simplicity in describing and analyzing the algorithm. Below, we first present the algorithm. Next we show that its complexity is \( \Theta(n \log n) \). And finally, we show that it produces a proportional allocation.

Our algorithm will be recursive, so we give some flexibility for the input by allowing it to consist of an interval \( [x, y] \subseteq [0, 1] \) and a subset of players \( S \subseteq \{1, 2, \ldots, n\} \). Our algorithm’s name is EP, and we would initially call it with input in which \( [x, y] = [0, 1] \) and \( S = \{1, 2, \ldots, n\} \). Below is the description of EP. A verbal explanation of what the algorithm does follows its description.

\[
\begin{align*}
[x, y]: & \text{ interval in } [0, 1]. \quad k: \text{ integer in } \{0, 1, 2, \ldots, n\}. \\
S: & \text{ subset of } \{1, 2, \ldots, n\} \text{ with } |S| = k. \\
EP(\langle [x, y], k, S \rangle): & \\
1 & \text{ If } k = 1 \text{ and } S = \{i\} \text{ for some } i, \text{ then let } A_i = [x, y]. \\
2 & \text{ Else:} \\
3 & \text{ For } i \in S, \text{ let } z_i = \text{Cut}_i(x, \text{Eval}(x, y)/2). \\
4 & \text{ Sort the } z_i \text{ so that } z_{i_1} \leq z_{i_2} \leq \cdots \leq z_{i_k}. \text{ Let } z^* = z_{i_{k/2}}. \\
5 & \text{ Run } EP(\langle [x, z^*], k/2, \{i_1, \ldots, i_{k/2}\} \rangle). \\
6 & \text{ Run } EP(\langle [z^*, y], k/2, \{i_{k/2+1}, \ldots, i_k\} \rangle).
\end{align*}
\]

The base case of the algorithm is when there is only one player. In this case we give the whole piece \([x, y]\) to that player. Otherwise, each player \( i \) makes a mark \( z_i \) such that \( V_i([x, z_i]) = \frac{1}{2} V_i([x, y]) \). Let \( z^* \) denote the \( n/2 \) mark from the left. We first recurse on \([x, z^*]\) and the left \( n/2 \) players, and then we recurse on \([z^*, y]\) and the right \( n/2 \) players.

We have to show that the algorithm’s time complexity is \( \Theta(n \log n) \) and that it produces a proportional allocation. First we show that the time complexity \( T(n) \) is \( \Theta(n \log n) \). Observe that the recursive relation that \( T(n) \) satisfies is

\[
T(1) = 0, \quad T(n) = 2n + 2T(n/2) \quad \text{for } n > 1.
\]

The base case corresponds to line 1 of the algorithm, and in this case, we don’t make any queries. In \( T(n) = 2n + 2T(n/2) \), the \( 2n \) comes from line 3 where we make 2 queries for each player. The \( 2T(n/2) \) comes from the two recursive calls on lines 5 and 6. To solve the recursion, i.e., to figure out the formula for \( T(n) \), we draw the associated recursion tree.
The root (top) of the tree corresponds to the input $S = \{1, 2, \ldots, n\}$ and is therefore labeled with an $n$. This branches off into two nodes, one corresponding to each recursive call. These nodes are labeled with $n/2$ since they correspond to recursive calls in which $|S| = n/2$. Those nodes further branch off into two nodes, and so on, until at the very bottom, we end up with nodes corresponding to inputs $S$ with $|S| = 1$. The number of queries made for each node of the tree is provided with a label on top of the node. For example, at the root (top), we make $2n$ queries before we do our recursive calls. This is why we put a $2n$ on top of that node. Similarly, every other node can be labeled.

We can divide the nodes of the tree into levels according to how far a node is from the root. So the root corresponds to level 0, the nodes it branches off to correspond to level 1, and so on. Observe that level $j$ has exactly $2^j$ nodes. The nodes that are at level $j$ make $2n/2^j$ queries. Therefore, the total number of queries made for level $j$ is $2n$. The only exception is the last level. The nodes at the last level correspond to the base case and don’t make any queries. In total, there are exactly $1 + \log_2 n$ levels (since we are counting the root as well). Thus, the total number of queries, and hence the time complexity, is exactly $2n \log_2 n$, which is $\Theta(n \log n)$.

We now prove that the allocation obtained by the algorithm is proportional. Observe that when we make the recursive call on $[x^*, z]$ and the left $n/2$ players, all these players value $[x^*, z]$ at least at $1/2$. Similarly, when we make the recursive call on $[z^*, y]$ and the right $n/2$ players, all these players value $[z^*, y]$ at least at $1/2$. This property is preserved at each level of the recursion in the following way. At level $\ell$ of the recursion, the players are divided into groups of size $n/2^\ell$. If each player values the corresponding interval at least at $1/2^\ell$, then at level $\ell + 1$, the players will value the interval that they are “assigned to” at least at $1/2^{\ell+1}$. In particular, when $\ell = \log_2 n$, each group is a singleton, and each player gets assigned a piece of cake that she values at least at $1/2^{\log_2 n} = 1/n$. This shows that the allocation is proportional.

Exercise 7.9 (Finding an envy-free allocation).

We say that a valuation function $V$ is \textit{piecewise constant} if there are points $x_1, x_2, \ldots, x_k \in [0, 1]$ such that $0 = x_1 < x_2 < \cdots < x_k = 1$ and for each $i \in \{1, 2, \ldots, k-1\}$, $V([x_i, x_{i+1}])$ is uniformly distributed over $[x_i, x_{i+1}]$. Suppose we have $n$ players such that each player has a piecewise constant valuation function. Show that in this case, an envy-free allocation always exists.

\footnote{Uniformly distributed means that if we were to take any subinterval $I$ of $[x_i, x_{i+1}]$ whose density/size is $\alpha$ fraction of the density/size of $[x_i, x_{i+1}]$, then $V(I) = \alpha \cdot V([x_i, x_{i+1}])$.}
Theorem 7.10 (Edmonds-Pruhs Theorem).
Any algorithm that produces an allocation satisfying the proportionality property must have time complexity $\Omega(n \log n)$. 
Quiz

1. True or false: When there are two players, an envy-free allocation can be found using a single query in the Robertson-Webb model.

2. True or false: In the allocation output by the Even-Paz algorithm, there always exists a player who is not envious.

3. True or false: In the allocation output by the Dubins-Spanier algorithm, there always exists a player who is not envious.

An allocation is called equitable if $V_i(A_i) = V_k(A_k)$ for any two players $i$ and $k$.

4. True or false: Any equitable allocation is proportional.

5. True or false: Any envy-free allocation is equitable.
Hints to Selected Exercises

Exercise 7.7 (Practice with cutting cake):
Modify Dubins-Spanier algorithm.

Exercise 7.9 (Finding an envy-free allocation):
For each player, make a mark for the points $x_1, \ldots, x_k$ describing their piecewise constant valuation function. How should you distribute the subinterval between any two adjacent marks?
Chapter 8

Introduction to Graph Theory
PREAMBLE

Chapter structure:

- Section 8.1 (Basic Definitions)
  - Definition 8.1 (Undirected graph)
  - Definition 8.6 (Neighborhood of a vertex)
  - Definition 8.8 ($d$-regular graphs)
  - Theorem 8.9 (Handshake Theorem)
  - Definition 8.11 (Paths and cycles)
  - Definition 8.12 (Connected graph, connected component)
  - Theorem 8.13 (Min number of edges to connect a graph)
  - Definition 8.14 (Tree, leaf, internal node)
  - Definition 8.19 (Directed graph)
  - Definition 8.21 (Neighborhood, out-degree, in-degree, sink, source)

- Section 8.2 (Graph Algorithms)
  - Definition 8.23 (Arbitrary-first search (AFS) algorithm)
  - Definition 8.25 (Breadth-first search (BFS) algorithm)
  - Definition 8.27 (Depth-first search (DFS) algorithm)
  - Definition 8.31 (Minimum spanning tree (MST) problem)
  - Theorem 8.33 (MST cut property)
  - Theorem 8.34 (Jarník-Prim algorithm for MST)
  - Definition 8.38 (Topological order of a directed graph)
  - Definition 8.41 (Topological sorting problem)
  - Lemma 8.42 (Acyclic directed graph has a sink)
  - Theorem 8.45 (Topological sort via DFS)

Chapter goals:

In the study of computational complexity of languages and computational problems, graphs play a very fundamental role. This is because an enormous number of computational problems that arise in computer science can be abstracted away as problems on graphs, which model pairwise relations between objects. This is great for various reasons. For one, this kind of abstraction removes unnecessary distractions about the problem and allows us to focus on its essence. Second, there is a huge literature on graph theory, so we can use this arsenal to better understand the computational complexity of graph problems. Applications of graphs are too many and diverse to list here, but we’ll name a few to give you an idea: communication networks, finding shortest routes in various settings, finding matchings between two sets of objects, social network analysis, kidney exchange protocols, linguistics, topology of atoms, and compiler optimization.

Our goal in this chapter is to introduce you to graph theory by providing the basic definitions and some well-known graph algorithms.
8.1 Basic Definitions

Definition 8.1 (Undirected graph).
An undirected graph \( G \) is a pair \((V, E)\), where

- \( V \) is a finite non-empty set called the set of vertices (or nodes),
- \( E \) is a set called the set of edges, and every element of \( E \) is of the form \( \{u, v\} \) for distinct \( u, v \in V \).

Example 8.2 (A graph with 6 vertices and 4 edges).
Let \( G = (V, E) \) where
\[
V = \{v_1, v_2, v_3, v_4, v_5, v_6\}
\]
and
\[
E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_4, v_5\}\}.
\]
We usually draw graphs in a way such that a vertex corresponds to a dot and an edge corresponds to a line connecting two dots. For example, the graph we have defined can be drawn as follows:

Note 8.3 (\( n \) and \( m \)).
Given a graph \( G = (V, E) \), we usually use \( n \) to denote the number of vertices \( |V| \) and \( m \) to denote the number of edges \( |E| \).

IMPORTANT 8.4 (Representations of graphs).
There are two common ways to represent a graph. Let \( v_1, v_2, \ldots, v_n \) be some arbitrary ordering of the vertices. In the adjacency matrix representation, a graph is represented by an \( n \times n \) matrix \( A \) such that
\[
A[i, j] = \begin{cases} 
1 & \text{if } \{v_i, v_j\} \in E, \\
0 & \text{otherwise}.
\end{cases}
\]
The adjacency matrix representation is not always the best representation of a graph. In particular, it is wasteful if the graph has very few edges. For such graphs, it can be preferable to use the adjacency list representation. In the adjacency list representation, you are given an array of size \( n \) and the \( i \)'th entry of the array contains a pointer to a linked list of vertex \( i \)'s neighbors.

Exercise 8.5 (Max number of edges in a graph).
In an \( n \)-vertex graph, what is the maximum possible value for the number of edges in terms of \( n \)?

\(^1\)Often the word “undirected” is omitted.
Definition 8.6 (Neighborhood of a vertex).
Let \( G = (V, E) \) be a graph, and \( e = \{u, v\} \in E \) be an edge in the graph. In this case, we say that \( u \) and \( v \) are neighbors or adjacent. We also say that \( u \) and \( v \) are incident to \( e \). For \( v \in V \), we define the neighborhood of \( v \), denoted \( N(v) \), as the set of all neighbors of \( v \), i.e. \( N(v) = \{u : \{v, u\} \in E\} \). The size of the neighborhood, \( |N(v)| \), is called the degree of \( v \), and is denoted by \( \deg(v) \).

Example 8.7 (Example of neighborhood and degree).
Consider Example 8.2 (A graph with 6 vertices and 4 edges). We have \( N(v_1) = \{v_2, v_3\} \), \( \deg(v_1) = \deg(v_2) = \deg(v_3) = 2 \), \( \deg(v_4) = \deg(v_5) = 1 \), and \( \deg(v_6) = 0 \).

Definition 8.8 (\( d \)-regular graphs).
A graph \( G = (V, E) \) is called \( d \)-regular if every vertex \( v \in V \) satisfies \( \deg(v) = d \).

Theorem 8.9 (Handshake Theorem).
Let \( G = (V, E) \) be a graph. Then
\[
\sum_{v \in V} \deg(v) = 2m.
\]

Proof. Our goal is to show that the sum of the degrees of all the vertices is equal to twice the number of edges. We will use a double counting argument to establish the equality. This means we will identify a set of objects and count it size in two different ways. One way of counting it will give us \( \sum_{v \in V} \deg(v) \), and the second way of counting it will give us \( 2m \). This then immediately implies that \( \sum_{v \in V} \deg(v) = 2m \).

We now proceed with the double counting argument. For each vertex \( v \in V \), put a “token” on all the edges it is incident to. We want to count the total number of tokens. Every vertex \( v \) is incident to \( \deg(v) \) edges, so the total number of tokens is \( \sum_{v \in V} \deg(v) \). On the other hand, each edge \( \{u, v\} \) in the graph will get two tokens, one from vertex \( u \) and one from vertex \( v \). So the total number of tokens put is \( 2m \). Therefore it must be that \( \sum_{v \in V} \deg(v) = 2m \). \( \square \)

Exercise 8.10 (Application of Handshake Theorem).
Is it possible to have a party with 251 people in which everyone knows exactly 5 other people in the party?

Definition 8.11 (Paths and cycles).
Let \( G = (V, E) \) be a graph. A path of length \( k \) in \( G \) is a sequence of distinct vertices
\[
v_0, v_1, \ldots, v_k
\]
such that \( \{v_{i-1}, v_i\} \in E \) for all \( i \in \{1, 2, \ldots, k\} \). In this case, we say that the path is from vertex \( v_0 \) to vertex \( v_k \).

A cycle of length \( k \) (also known as a \( k \)-cycle) in \( G \) is a sequence of vertices
\[
v_0, v_1, \ldots, v_{k-1}, v_0
\]
such that \( v_0, v_1, \ldots, v_{k-1} \) is a path, and \( \{v_0, v_{k-1}\} \in E \). In other words, a cycle is just a “closed” path. The starting vertex in the cycle is not important. So for example,
\[
v_1, v_2, \ldots, v_{k-1}, v_0, v_1
\]
would be considered the same cycle. Also, if we list the vertices in reverse order, we consider it to be the same cycle. For example,

\[ v_0, v_{k-1}, v_{k-2} \ldots, v_1, v_0 \]

represents the same cycle as before.

A graph that contains no cycles is called acyclic.

**Definition 8.12** (Connected graph, connected component).
Let \( G = (V, E) \) be a graph. We say that two vertices in \( G \) are connected if there is a path between those two vertices. We say that \( G \) is connected if every pair of vertices in \( G \) is connected.

A subset \( S \subseteq V \) is called a connected component of \( G \) if \( G \) restricted to \( S \), i.e. the graph \( G' = (S, E' = \{ \{u, v\} \in E : u, v \in S\}) \), is a connected graph, and \( S \) is disconnected from the rest of the graph (i.e. \( \{u, v\} \notin E \) when \( u \in S \) and \( v \notin S \)).

Note that a connected graph is a graph with only one connected component.

**Theorem 8.13** (Min number of edges to connect a graph).
Let \( G = (V, E) \) be a connected graph with \( n \) vertices and \( m \) edges. Then \( m \geq n - 1 \). Furthermore, \( m = n - 1 \) if and only if \( G \) is acyclic.

**Proof.** We first prove that a connected graph with \( n \) vertices and \( m \) edges satisfies \( m \geq n - 1 \). Take \( G \) and remove all its edges. This graph consists of isolated vertices and therefore contains \( n \) connected components. Let’s now imagine a process in which we put back the edges of \( G \) one by one. The order in which we do this does not matter. At the end of this process, we must end up with just one connected component since \( G \) is connected. When we put back an edge, there are two options. Either

(i) we connect two different connected components by putting an edge between two vertices that are not already connected, or

(ii) we put an edge between two vertices that are already connected, and therefore create a cycle.

Observe that if (i) happens, then the number of connected components goes down by 1. If (ii) happens, the number of connected components remains the same. So every time we put back an edge, the number of connected components in the graph can go down by at most 1. Since we start with \( n \) connected components and end with 1 connected component, (i) must happen at least \( n - 1 \) times, and hence \( m \geq n - 1 \). This proves the first part of the theorem. We now prove \( m = n - 1 \iff G \) is acyclic.

\[ m = n - 1 \implies G \text{ is acyclic: If } m = n - 1, \text{ then (i) must have happened at each step since otherwise, we could not have ended up with one connected component. Note that (i) cannot create a cycle, so in this case, our original graph must be acyclic.} \]

\[ G \text{ is acyclic } \implies m = n - 1: \text{ To prove this direction (using the contrapositive), assume } m > n - 1. \text{ We know that (i) can happen at most } n - 1 \text{ times. So in at least one of the steps, (ii) must happen. This implies } G \text{ contains a cycle.} \]

**Definition 8.14** (Tree, leaf, internal node).
A graph satisfying two of the following three properties is called a tree:

(i) connected,

(ii) \( m = n - 1 \),
(iii) acyclic.

A vertex of degree 1 in a tree is called a leaf. And a vertex of degree more than 1 is called an internal node.

Exercise 8.15 (Equivalent definitions of a tree).
Show that if a graph has two of the properties listed in Definition 8.14 (Tree, leaf, internal node), then it automatically has the third as well.

Exercise 8.16 (A tree has at least 2 leaves).
Let $T$ be a tree with at least 2 vertices. Show that $T$ must have at least 2 leaves.

Exercise 8.17 (Max degree is at most number of leaves).
Let $T$ be a tree with $L$ leaves. Let $\Delta$ be the largest degree of any vertex in $T$. Prove that $\Delta \leq L$.

Note 8.18 (Root, parent, child, sibling, etc.).
Given a tree, we can pick an arbitrary node to be the root of the tree. In a rooted tree, we use “family tree” terminology: parent, child, sibling, ancestor, descendant, lowest common ancestor, etc. (We assume you are already familiar with these terms.)

Definition 8.19 (Directed graph).
A directed graph $G$ is a pair $(V,A)$, where
- $V$ is a finite set called the set of vertices (or nodes),
- $A$ is a finite set called the set of directed edges (or arcs), and every element of $A$ is a tuple $(u,v)$ for $u,v \in V$. If $(u,v) \in A$, we say that there is a directed edge from $u$ to $v$. Note that $(u,v) \neq (v,u)$ unless $u = v$.

Note 8.20 (Drawing directed graphs).
Below is an example of how we draw a directed graph:

\[
\begin{array}{c}
\text{\includegraphics{example.png}}
\end{array}
\]

Definition 8.21 (Neighborhood, out-degree, in-degree, sink, source).
Let $G = (V,A)$ be a directed graph. For $u \in V$, we define the neighborhood of $u$, $N(u)$, as the set $\{v \in V : (u,v) \in A\}$. The out-degree of $u$, denoted $\deg_{\text{out}}(u)$, is $|N(u)|$. The in-degree of $u$, denoted $\deg_{\text{in}}(u)$, is the size of the set $\{v \in V : (v,u) \in A\}$. A vertex with out-degree 0 is called a sink. A vertex with in-degree 0 is called a source.

Note 8.22 (Paths and cycles in directed graphs).
The notions of paths and cycles naturally extend to directed graphs. For example, we say that there is a path from $u$ to $v$ if there is a sequence of distinct vertices $u = v_0, v_1, \ldots, v_k = v$ such that $(v_{i-1}, v_i) \in A$ for all $i \in \{1, 2, \ldots, k\}$. 82
8.2 Graph Algorithms

8.2.1 Graph searching algorithms

Definition 8.23 (Arbitrary-first search (AFS) algorithm).
The arbitrary-first search algorithm, denoted AFS, is the following generic algorithm for searching a given graph. Below, “bag” refers to an arbitrary data structure that allows us to add and retrieve objects.

\[ G = (V, E); \text{ graph. } s: \text{ vertex in } V. \]
\[ \text{AFS}(⟨G, s⟩): \]

1. Put \( s \) into bag.
2. While bag is non-empty:
   3. Pick an arbitrary vertex \( v \) from bag.
   4. If \( v \) is unmarked:
      5. Mark \( v \).
      6. parent\( (v) = p. \)
      7. For each neighbor \( w \) of \( v \):
         8. Put \( (v, w) \) into bag.

Note that when a vertex \( w \) is added to the bag, it gets there because it is the neighbor of a vertex \( v \) that has been just marked by the algorithm. In this case, we’ll say that \( v \) is the parent of \( w \) (and \( w \) is the child of \( v \)). Explicitly keeping track of this parent-child relationship is convenient, so we modify the above algorithm to keep track of this information. Below, a tuple of vertices \((v, w)\) has the meaning that vertex \( v \) is the parent of \( w \). The initial vertex \( s \) has no parent, so we denote this situation by \((⊥, s)\).

\[ G = (V, E); \text{ graph. } s: \text{ vertex in } V. \]
\[ \text{AFS2}(⟨G⟩): \]

1. For \( v \) not marked as visited:
2. Run AFS(⟨G, v⟩).

Note 8.24 (Traversing all the vertices in the graph).
Note that AFS\((G, s)\) visits all the vertices in the connected component that \( s \) is a part of. If we want to traverse all the vertices in the graph, and the graph has multiple connected components, then we can do:

\[ G = (V, E); \text{ graph. } \]
\[ \text{AFS2}(⟨G⟩): \]

1. For \( v \) not marked as visited:
2. Run AFS(⟨G, v⟩).
Definition 8.25 (Breadth-first search (BFS) algorithm).
The breadth-first search algorithm, denoted BFS, is AFS where the bag is chosen to be a queue data structure.

Note 8.26 (Running time of BFS).
The running time of BFS\((G, s)\) is \(O(m)\), where \(m\) is the number of edges of the input graph. If we do a BFS for each connected component, the total running time is \(O(m + n)\), where \(n\) is the number of vertices.\(^2\) (We are assuming the graph is given as an adjacency list.)

Definition 8.27 (Depth-first search (DFS) algorithm).
The depth-first search algorithm, denoted DFS, is AFS where the bag is chosen to be a stack data structure.

Note 8.28 (Recursive DFS).
There is a natural recursive representation of the DFS algorithm, as follows.

<table>
<thead>
<tr>
<th>(\text{DFS}(\langle G, s \rangle))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mark (s).</td>
</tr>
<tr>
<td>2 For each neighbor (v) of (s):</td>
</tr>
<tr>
<td>3 If (v) is unmarked:</td>
</tr>
<tr>
<td>4 Run DFS((\langle G, v \rangle)).</td>
</tr>
</tbody>
</table>

Note 8.29 (Running time of DFS).
The running time of DFS\((G, s)\) is \(O(m)\), where \(m\) is the number of edges of the input graph. If we do a DFS for each connected component, the total running time is \(O(m + n)\), where \(n\) is the number of vertices. (We are assuming the graph is given as an adjacency list.)

Note 8.30 (Search algorithms on directed graphs).
The search algorithms presented above can be applied to directed graphs as well.

8.2.2 Minimum spanning tree

Definition 8.31 (Minimum spanning tree (MST) problem).
In the minimum spanning tree problem, the input is a connected undirected graph \(G = (V, E)\) together with a cost function \(c : E \rightarrow \mathbb{R}^+\). The output is a subset of the edges of minimum total cost such that, in the graph restricted to these edges, all the vertices of \(G\) are connected.\(^3\) For convenience, we’ll assume that the edges have unique edge costs, i.e. \(e \neq e' \implies c(e) \neq c(e')\).

\(^2\)Take a moment to reflect on why this is the case.
\(^3\)Obviously this subset of edges would not contain a cycle since if it did, we could remove any edge on the cycle, preserve the connectivity property, and obtain a cheaper set. Therefore, this set forms a tree.
Note 8.32 (Unique edges costs imply unique MST).
With unique edge costs, the minimum spanning tree is unique.

Theorem 8.33 (MST cut property).
Suppose we are given an instance of the MST problem. For any \( V' \subseteq V \), let \( e = \{u, w\} \) be the cheapest edge with the property that \( u \in V' \) and \( w \in V \setminus V' \). Then \( e \) must be in the minimum spanning tree.

Proof. Let \( T \) be the minimum spanning tree. The proof is by contradiction, so assume that \( e = \{u, w\} \) is not in \( T \). Since \( T \) spans the whole graph, there must be a path from \( u \) to \( w \) in \( T \). Let \( e' = \{u', w'\} \) be the first edge on this path such that \( u' \in V' \) and \( w' \in V \setminus V' \). Let \( T_{e-e'} = (T \setminus \{e'\}) \cup \{e\} \). If \( T_{e-e'} \) is a spanning tree, then we reach a contradiction because \( T_{e-e'} \) has lower cost than \( T \) (since \( c(e) < c(e') \)).

Let \( T_{e-e'} \) be a spanning tree: Clearly \( T_{e-e'} \) has \( n - 1 \) edges (since \( T \) has \( n - 1 \) edges). So if we can show that \( T_{e-e'} \) is connected, this would imply that \( T_{e-e'} \) is a tree and touches every vertex of the graph, i.e., \( T_{e-e'} \) is a spanning tree. Consider any two vertices \( s, t \in V \). There is a unique path from \( s \) to \( t \) in \( T \). If this path does not use the edge \( e' = \{u', w'\} \), then the same path exists in \( T_{e-e'} \), so \( s \) and \( t \) are connected in \( T_{e-e'} \). If the path does use \( e' = \{u', w'\} \), then instead of taking the edge \( \{u', w'\} \), we can take the following path: take the path from \( u' \) to \( u \), then take the edge \( e = \{u, w\} \), then take the path from \( w \) to \( w' \). So replacing \( \{u', w'\} \) with this path allows us to construct a sequence of vertices starting from \( s \) and ending at \( t \), such that each consecutive pair of vertices is an edge. Therefore \( s \) and \( t \) are connected. \( \square \)

Theorem 8.34 (Jarník-Prim algorithm for MST).
There is an algorithm that solves the MST problem in polynomial time.

Proof. We first present the algorithm which is due to Jarník and Prim. Given an undirected graph \( G = (V, E) \) and a cost function \( c : E \to \mathbb{R}^+ \):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( V' = {u} ) (for some arbitrary ( u \in V ))</td>
</tr>
<tr>
<td>2</td>
<td>( E' = \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>While ( V' \neq V ):</td>
</tr>
<tr>
<td>4</td>
<td>Let ( {u, v} ) be the minimum cost edge such that ( u \in V' ) but ( v \notin V' ).</td>
</tr>
<tr>
<td>5</td>
<td>Add ( {u, v} ) to ( E' ).</td>
</tr>
<tr>
<td>6</td>
<td>Add ( v ) to ( V' ).</td>
</tr>
</tbody>
</table>

\( G = (V, E) \); graph. \( c : E \to \mathbb{R}^+ \); edge costs. 
\( \text{MST}(\langle G, c \rangle) \):
1. \( V' = \{u\} \) (for some arbitrary \( u \in V \))
2. \( E' = \emptyset \)
3. While \( V' \neq V \):
4. Let \( \{u, v\} \) be the minimum cost edge such that \( u \in V' \) but \( v \notin V' \).
5. Add \( \{u, v\} \) to \( E' \).
6. Add \( v \) to \( V' \).
By Theorem 8.33 (MST cut property), the algorithm always adds an edge that must be in the MST. The number of iterations is $n - 1$, so all the edges of the MST are added to $E'$. Therefore the algorithm correctly outputs the unique MST.

The running time of the algorithm can be upper bounded by $O(nm)$ because there are $O(n)$ iterations, and the body of the loop can be done in $O(m)$ time.

Exercise 8.35 (MST with negative costs).
Suppose an instance of the Minimum Spanning Tree problem is allowed to have negative costs for the edges. Explain whether we can use the Jarník-Prim algorithm to compute the minimum spanning tree in this case.

Exercise 8.36 (Maximum spanning tree).
Consider the problem of computing the maximum spanning tree, i.e., a spanning tree that maximizes the sum of the edge costs. Explain whether the Jarník-Prim algorithm solves this problem if we modify it so that at each iteration, the algorithm chooses the edge between $V'$ and $V\setminus V'$ with the maximum cost.

Exercise 8.37 (Kruskal’s algorithm).
Consider the following algorithm for the MST problem (which is known as Kruskal’s algorithm). Start with MST being the empty set. Go through all the edges of the graph one by one from the cheapest to the most expensive. Add the edge to the MST if it does not create a cycle. Show that this algorithm correctly outputs the MST.

### 8.2.3 Topological sorting

**Definition 8.38** (Topological order of a directed graph).
A topological order of an $n$-vertex directed graph $G = (V,A)$ is a bijection $f : V \rightarrow \{1, 2, \ldots, n\}$ such that if $(u,v) \in A$, then $f(u) < f(v)$.

**Example 8.39** (Example of topological order).
On the left, we have a directed graph, and on the right, we represent the topological order of the graph.

Here, $f(e) = 1$, $f(d) = 2$, $f(a) = 3$, $f(b) = 4$, and $f(c) = 5$. 

86
**Exercise 8.40** (Cycle implies no topological order).
Show that if a directed graph has a cycle, then it does not have a topological order.

**Definition 8.41** (Topological sorting problem).
In the topological sorting problem, the input is a directed acyclic graph, and the output is a topological order of the graph.

**Lemma 8.42** (Acyclic directed graph has a sink).
If a directed graph is acyclic, then it has a sink vertex.

*Proof*. By contrapositive: If a directed graph has no sink vertices, then it means that every vertex has an outgoing edge. Start with any vertex, and follow an outgoing edge to arrive at a new vertex. Repeat this process. At some point, you have to visit a vertex that you have visited before. This forms a cycle. □

**Note 8.43** (Topological sort - naïve algorithm).
The following algorithm solves the topological sorting problem in polynomial time.

\[ G = (V, A): \text{directed acyclic graph.} \]

\[
\text{Top-Sort-Naïve}(⟨G⟩):
\begin{align*}
1 & \quad p = |V|.
2 & \quad \text{While } p \geq 1:
3 & \quad \quad \text{Find a sink vertex } v \text{ and remove it from } G.
4 & \quad \quad f(v) = p.
5 & \quad \quad p = p - 1.
6 & \quad \text{Output } f.
\end{align*}
\]

**Exercise 8.44** (Topological sort, correctness of naïve algorithm).
Show the algorithm above correctly solves the topological sorting problem, i.e., show that for \((u, v) \in A, f(u) < f(v)\). What is the running time of this algorithm?

**Theorem 8.45** (Topological sort via DFS).
There is a \(O(n + m)\)-time algorithm that solves the topological sorting problem.

*Proof*. The algorithm is a slight variation of DFS.

\[ G = (V, A): \text{directed acyclic graph.} \]

\[
\text{Top-Sort}(⟨G⟩):
\begin{align*}
1 & \quad p = |V|.
2 & \quad \text{For } v \text{ not marked as visited:}
3 & \quad \quad \text{Run DFS}(⟨G, v⟩).
\end{align*}
\]
\[
G = (V, A): \text{directed graph. } v: v \in V.
\]

DFS'(⟨G, v⟩):
1 Mark v as “visited”.
2 For each neighbor u of v:
3 If u is not marked visited:
4 Run DFS(⟨G, u⟩).
5 \( f(v) = p \).
6 \( p = p - 1 \).

4 Output \( f \).

The running time is the same as DFS. To show the correctness of the algorithm, all we need to show is that for \((u, v) \in A\), \( f(u) < f(v) \). There are two cases to consider.

- **Case 1**: \( u \) is visited before \( v \). In this case observe that DFS(⟨G, v⟩) will finish before DFS(⟨G, u⟩). Therefore \( f(v) \) will be assigned a value before \( f(u) \), and so \( f(u) < f(v) \).

- **Case 2**: \( v \) is visited before \( u \). Notice that we cannot visit \( u \) from DFS(⟨G, v⟩) because that would imply that there is a cycle. Therefore DFS(⟨G, u⟩) is called after DFS(⟨G, v⟩) is completed. As before, \( f(v) \) will be assigned a value before \( f(u) \), and so \( f(u) < f(v) \).

\(\square\)
Quiz

1. True or false: For a graph $G = (V, E)$, if for any $u, v \in V$ there exists a unique path from $u$ to $v$, then $G$ is a tree.

2. True or false: Depth-first-search algorithm runs in $O(n)$ time for a connected graph, where $n$ is the number of vertices of the input graph.

3. True or false: If a graph on $n$ vertices has $n - 1$ edges, then it must be acyclic.

4. True or false: If a graph on $n$ vertices has $n - 1$ edges, then it must be connected.

5. True or false: If a graph on $n$ vertices has $n - 1$ edges, then it must be a tree.

6. True or false: The degree sum of a graph is $\sum_{v \in V} \deg(v)$. Every tree on $n$ vertices has exactly the same degree sum.

7. True or false: In a directed graph a self-loop, i.e. an edge of the form $(u, u)$, is allowed by the definition.

8. True or false: Every directed graph has a topological order.

9. True or false: Suppose a graph has 2 edges with the same cost. Then there are at least 2 MSTs of the graph.

10. True or false: Let $G$ be a 5-regular graph (i.e. a graph in which every vertex has degree exactly 5). It is possible that $G$ has 15251 edges.
Hints to Selected Exercises

Exercise 8.15 (Equivalent definitions of a tree):
Make use of Theorem (Min number of edges to connect a graph) and its proof.

Exercise 8.16 (A tree has at least 2 leaves):
Use the Handshake Theorem.

Exercise 8.17 (Max degree is at most number of leaves):
There are at least 3 different solutions to this problem. One uses the Handshake Theorem. Another uses induction on the number of vertices.

Exercise 8.35 (MST with negative costs):
Yes, we can.

Exercise 8.36 (Maximum spanning tree):
Yes, it does. Consider multiplying the costs by $-1$.

Exercise 8.37 (Kruskal’s algorithm):
The correctness of the algorithm follows from the MST cut property. Show by induction that every time the algorithm decides to add an edge, it adds one that must be in the MST (by the MST cut property).