math is hard, but you don't have to do it alone!

October 4, 2017
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 ........................................... 2
   1.2 Languages .................................................................. 2
   1.3 Encodings .................................................................. 3
   1.4 Computational Problems and Decision Problems .............. 4

2 Deterministic Finite Automata ............................................ 5
   2.1 Basic Definitions ..................................................... 6
   2.2 Irregular Languages .................................................. 7
   2.3 Closure Properties of Regular Languages .................... 8

3 Turing Machines .............................................................. 11
   3.1 Basic Definitions ..................................................... 12
   3.2 Decidable Languages ................................................ 13

4 Countable and Uncountable Sets ........................................ 17
   4.1 Basic Definitions ..................................................... 18
   4.2 Countable Sets ........................................................ 19
   4.3 Uncountable Sets ..................................................... 20

5 Undecidable Languages ...................................................... 23
   5.1 Existence of Undecidable Languages ......................... 24
   5.2 Examples of Undecidable Languages ......................... 24
   5.3 Undecidability Proofs by Reductions ......................... 26

6 Time Complexity ............................................................... 29
   6.1 Big-O, Big-Omega and Theta ..................................... 30
   6.2 Worst-Case Running Time of Algorithms ................. 30
   6.3 Complexity of Algorithms with Integer Inputs ............. 31

7 The Science of Cutting Cake ............................................... 33
   7.1 The Problem and the Model ..................................... 34
   7.2 Cake Cutting Algorithms in the Robertson-Webb Model .... 35

8 Introduction to Graph Theory ............................................. 39
   8.1 Basic Definitions ..................................................... 40
   8.2 Graph Algorithms ................................................... 42
      8.2.1 Graph searching algorithms .............................. 42
      8.2.2 Minimum spanning tree .................................... 43
      8.2.3 Topological sorting ......................................... 44
Chapter 1

Strings and Encodings
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.

Definition 1.2 (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 $\epsilon$.

Definition 1.3 (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.

Definition 1.4 (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,

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

Definition 1.5 (Reversal of a string).
For a string $w = a_1a_2\ldots a_n$, the reversal of $w$, denoted $w^R$, is the string $w^R = a_na_{n-1}\ldots a_1$.

Definition 1.6 (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 $v$.

Definition 1.7 (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.

Definition 1.8 (Substring).
We say that a string $u$ is a substring of string $w$ if $w = xuy$ for some strings $x$ and $y$.

1.2 Languages

Definition 1.9 (Language).
Any (possibly infinite) subset $L \subseteq \Sigma^*$ is called a language over the alphabet $\Sigma$. 
Definition 1.10 (Reversal of a language).
Given a language \(L \subseteq \Sigma^*\), we define its reversal, denoted \(L^R\), as the language
\[
L^R = \{w^R \in \Sigma^* : w \in L\}.
\]

Definition 1.11 (Concatenation of languages).
Given two languages \(L_1, L_2 \subseteq \Sigma^*\), we define their concatenation, denoted \(L_1 L_2\) or \(L_1 \cdot L_2\), as the language
\[
L_1 L_2 = \{uv \in \Sigma^* : u \in L_1, v \in L_2\}.
\]

Definition 1.12 (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,\(^1\)
\[
L^n = L \cdot L \cdot L \cdots L, \quad \text{\(n\) times}.
\]
Equivalently,
\[
L^n = \{u_1 u_2 \cdots u_n \in \Sigma^* : u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.
\]

Definition 1.13 (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,
\[
L^* = \{u_1 u_2 \cdots u_n \in \Sigma^* : n \in \mathbb{N}, u_i \in L \text{ for all } i \in \{1, 2, \ldots, n\}\}.
\]

1.3 Encodings

Definition 1.14 (Encoding of a set).
Let \(A\) be a set (which is possibly countably infinite\(^2\)), and let \(\Sigma\) be a alphabet. An encoding of the elements of \(A\), using \(\Sigma\), is an injective function \(\text{Enc} : A \rightarrow \Sigma^*\). 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.\(^4\)

\(^1\)We can omit parentheses as the order in which the concatenation \(\cdot\) is applied does not matter.
\(^2\)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?
1.4 Computational Problems and Decision Problems

**Definition 1.15 (Computational problem).**
Let $\Sigma$ be an alphabet. Any function $f : \Sigma^* \rightarrow \Sigma^*$ is called a *computational problem* over the alphabet $\Sigma$.

**Definition 1.16 (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 $\{\text{Reject, Accept}\}$. 
Chapter 2

Deterministic Finite Automata
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).

**Definition 2.2** (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.

**Definition 2.3** (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 $w$.

**Definition 2.4** (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 $\delta : Q \times \Sigma^* \to Q$. 
**Definition 2.5 (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 $M$).\(^\dagger\)

---

\(^\dagger\)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.

---

**Definition 2.6 (Regular language).**

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

---

### 2.2 Irregular Languages

**Theorem 2.7 (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,\(^2\) 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 $w = 1^i$. Then since $M$ recognizes $L$, we know $0^i w = 0^j 1^i$ must end up in an accepting state. On the other hand, since $i \neq j$, $0^i w = 0^j 1^i$ is not in the language, and therefore cannot end up in an accepting state. This is the desired contradiction. \(\square\)

---

\(^2\)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.
\[ a^{2^i} \] end up in the same state in \( M \). Therefore \( a^{2^i}w \) and \( a^{2^i}w \), 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 \( a^{2^i}w \) ends up in a rejecting state (i.e., they end up in different states).

Consider the string \( w = a^{2^i} \). Then \( a^{2^i}w = a^{2^i}a^{2^i} = a^{2i+1} \), and therefore must end up in an accepting state. On the other hand, \( a^{2^i}w = a^{2^i}a^{2^i} = a^{2i+2^i} \). We claim that this word must end up in a rejecting state because \( 2^i + 2^i \) 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

\[
2^i < 2^i + 2^i < 2^i + 2^j = 2^{j+1},
\]

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

### 2.3 Closure Properties of Regular Languages

**Theorem 2.9** (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 \( \delta(q_0, w) \in F, (\delta(q_0, w), \delta'(q'_0, w)) \in F'' \) (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. \( \square \)
Corollary 2.10 (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 2.9 (Regular languages are closed under union)) and closed under complementation (Exercise (Are regular languages closed under complementation?)). The result then follows since \( A \cap B = \overline{A} \cup \overline{B} \).

Theorem 2.11 (Regular languages are closed under concatenation).
If \( L_1, L_2 \subseteq \Sigma^* \) are regular languages, then the language \( L_1 L_2 \) is also regular.

Proof. Given regular languages \( L_1 \) and \( L_2 \), we want to show that \( L_1 L_2 \) 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. To show \( L_1 L_2 \) is regular, we’ll construct a DFA \( M'' = (Q'', \Sigma, \delta'', q_0'', F'') \) that recognizes \( L_1 L_2 \). 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 \( w \in L_1 L_2 \) 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 \( w = w_1 w_2 \ldots w_n \in \Sigma^* \), a thread with respect to \( w \) is a sequence of states

\[
r_0, r_1, r_2, \ldots, r_i, s_{i+1}, s_{i+2}, \ldots, s_n,
\]

where \( r_0, r_1, \ldots, r_i \) is an accepting computation path of \( M \) with respect to \( w_1 w_2 \ldots w_i \) and \( q_0, s_{i+1}, s_{i+2}, \ldots, s_n \) is a computation path (not necessarily accepting) of \( M' \) with respect to \( w_{i+1} w_{i+2} \ldots w_n \). A thread like this corresponds to simulating \( M \) on \( w_1 w_2 \ldots w_i \) (at which point we require that an accepting state of \( M \) is reached), and then simulating \( M' \) on \( w_{i+1} w_{i+2} \ldots w_n \).

For each way of writing \( w \) as \( uv \) where \( u \in L_1 \), there is a corresponding thread for it. Note that \( w \in L_1 L_2 \) if and only if there is a thread in which \( s_n \in F' \). 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 \( n = 10 \)):

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

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 \( Q'' = Q \times P(Q') \).

\[3\text{This means } r_0 = q_0, r_i \in F, \text{ and when the symbol } w_i \text{ is read, } M \text{ transitions from state } r_{j-1} \text{ to state } r_j. \text{ See Definition 2.2 (Computation path for a DFA).}\]

\[4\text{Recall that for any set } Q, \text{ the set of all subsets of } Q \text{ is called the power set of } Q, \text{ and is denoted by } P(Q).\]
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 \( M'' \):

- The set of states is \( Q'' = Q \times \mathcal{P}(Q') = \{(q,S) : q \in Q, S \subseteq Q'\} \).
  (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 \( a \in \Sigma \),

\[
\delta''((q,S),a) = \begin{cases} 
(\delta(q,a), \delta''_P(S,a)) & \text{if } \delta(q,a) \notin F, \\
(\delta(q,a), \delta''_P(S,a) \cup \{q_0\}) & \text{if } \delta(q,a) \in F.
\end{cases}
\]

(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

\[
q_0'' = \begin{cases}
(q_0, \emptyset) & \text{if } q_0 \notin F, \\
(q_0, \{q_0'\}) & \text{if } q_0 \in F.
\end{cases}
\]

(Initially, if \( q_0 \notin F \), 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 \( L_1 L_2 \), i.e. \( L(M'') = L_1 L_2 \), 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 \( w \in L_1 L_2 \) if and only if there is a thread with respect to \( w \) that ends in an accepting state of \( M' \).
Chapter 3

Turing Machines
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 $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ (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).

**Definition 3.2 (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.

1. 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 $\cdots \sqcup \sqcup \sqcup u v \sqcup \sqcup \sqcup \cdots$, 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.\(^2\)

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) $\alpha_0 = q_0x$, where $q_0$ is $M$’s initial state;

(ii) $\alpha_t \vdash_M \alpha_{t+1}$ for all $t = 0, 1, 2, \ldots, T - 1$;

\(^1\)Supernerd note: we will always assume $Q$ and $\Gamma$ are disjoint sets.

\(^2\)There 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 $u$. 

(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.

**Definition 3.3** (Decider Turing machine).
A Turing machine is called a **decider** if it halts on all inputs.

**Definition 3.4** (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 3.5** (Decidable language).
A language \( L \) is called **decidable** (or **computable**) if \( L = L(M) \) for some decider Turing machine \( M \).

**Definition 3.6** (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:

\[
\begin{align*}
M &: \text{Turing machine.} \\
x &: \text{string in } \Sigma^*. \\
U(\langle M, x \rangle): \\
1 &: \text{Simulate } M \text{ on input } x \text{ (i.e. run } M(x)). \\
2 &: \text{If it accepts, accept.} \\
3 &: \text{If it rejects, reject.}
\end{align*}
\]

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.

### 3.2 Decidable Languages

**Definition 3.7** (Languages related to encodings of DFAs).
Fix some alphabet \( \Sigma \). We define the following languages:

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

**Theorem 3.8** (\( \text{ACCEPTS}_{\text{DFA}} \) and \( \text{SELF-ACCEPTS}_{\text{DFA}} \) 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 ACCEPTS_{DFA}.

For SELF-ACCEPTS_{DFA}, we just need to slightly modify the above machine:

D: DFA.
M((D)):
1. Simulate D on input ⟨D⟩ (i.e. run D(⟨D⟩)).
2. If it accepts, accept.
3. If it rejects, reject.

Again, it is clear that this correctly decides SELF-ACCEPTS_{DFA}.

\[\square\]

Theorem 3.9 (EMPTY_{DFA} is decidable).
The language EMPTY_{DFA} is decidable.

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

A decider for EMPTY_{DFA} takes as input ⟨D⟩ for some DFA D = (Q, Σ, δ, q₀, F), and needs to determine if L(D) = ∅. In other words, it needs to determine if there is any string that D accepts. If we view the DFA as a directed graph, then notice that the DFA accepts some string if and only if there is a directed path from q₀ to some state in F. Therefore, the following decider decides EMPTY_{DFA} correctly.

D: DFA.
M(⟨D⟩):
1. Build a directed graph from ⟨D⟩.
2. Run a graph search algorithm starting from the starting state of D.
3. If a node corresponding to an accepting state is reached, reject.
4. Else, accept.

\[\square\]

Theorem 3.10 (EQ_{DFA} is decidable).
The language EQ_{DFA} is decidable.

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

Our argument is going to use the fact that EMPTY_{DFA} is decidable (Theorem 3.9 (EMPTY_{DFA} is decidable)). In particular, the decider we present for EQ_{DFA} will use the decider for EMPTY_{DFA} as a subroutine. Let M denote a decider TM for EMPTY_{DFA}.

\[\square\]

---

\[^{3}\text{Even though we have not formally defined the notion of a graph yet, we do assume you are familiar with the concept from a prerequisite course and that you have seen some simple graph search algorithms like Breadth-First Search or Depth-First Search.}\]
A decider for $\text{EQ}_{\text{DFA}}$ 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:

$$(L(D_1) \cap L(D_2)) \cup (L(D_1) \cap L(D_2)).$$

Note that $L(D_1) = L(D_2)$ if and only if the symmetric difference is empty. Our decider for $\text{EQ}_{\text{DFA}}$ will construct a DFA $D$ such that $L(D) = (L(D_1) \cap L(D_2)) \cup (L(D_1) \cap L(D_2))$, 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 $D_1'$ and $D_2'$ that accept $L(D_1)$ and $L(D_2)$ respectively (see Exercise (Are regular languages closed under complementation?));
- construct a DFA that accepts $L(D_1) \cap L(D_2)$ by using the (constructive) proof that regular languages are closed under the intersection operation;
- 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 $(L(D_1) \cap L(D_2)) \cup (L(D_1) \cap L(D_2))$ by using the constructive proof that regular languages are closed under the union operation.

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

1. Construct DFA $D$ as described above.
2. Run $M(\langle D \rangle)$.
3. If it accepts, accept.
4. If it rejects, reject.

By our discussion above, the decider works correctly. □

---

4The 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 $(A \cap \overline{B}) \cup (\overline{A} \cap B)$.

5The constructive proof gives us a way to construct the DFA accepting $L(D_1) \cap L(D_2)$ given $D_1$ and $D_2$. 

15
Chapter 4

Countable and Uncountable Sets
4.1 Basic Definitions

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

- A function $f : A \to B$ 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 $f : A \to B$ 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 $f : A \to B$ 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) $A \leftrightarrow B$ if and only if $B \twoheadrightarrow A$;

(b) if $A \hookrightarrow B$ and $B \leftrightarrow C$, then $A \leftrightarrow C$;

(c) $A \leftrightarrow B$ if and only if $A \hookrightarrow B$ and $B \hookrightarrow A$.

Definition 4.3 (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 $B \twoheadrightarrow A$.\(^1\)

- We write $|A| < |B|$ if it is not the case that $|A| \geq |B|$.\(^2\)

Definition 4.4 (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. $|A| > |\mathbb{N}|$.

Theorem 4.5 (Characterization of countably infinite sets).
A set $A$ is countably infinite if and only if $|A| = |\mathbb{N}|$.

---

\(^1\) Even though not explicitly stated, $|B| \geq |A|$ has the same meaning as $|A| \leq |B|$.

\(^2\) Similar to above, $|B| > |A|$ has the same meaning as $|A| < |B|$.
4.2 Countable Sets

**Proposition 4.6** ($\mathbb{Z} \times \mathbb{Z}$ 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.\footnote{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.}

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. \hfill $\square$

**Proposition 4.7** ($\mathbb{Q}$ is countable).
The set of rational numbers $\mathbb{Q}$ is countable.

*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. $|\mathbb{Q}| \leq |\mathbb{N}|$. \hfill $\square$

**Proposition 4.8** ($\Sigma^*$ 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 $\Sigma^0$ 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.

Proposition 4.9 (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 4.8 (\(\Sigma^*\) is countable)), i.e., $|\Sigma^*| \leq |\mathbb{N}|$, the result follows. \qed

Proposition 4.10 (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

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, /, x\}.$$  

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

$$2x^3 - 1/34x^2 + 99/100x + 22/7$$

represents the polynomial

$$2x^3 - 1/34x^2 + 99/100x + 22/7.$$  

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. \qed

### 4.3 Uncountable Sets

Theorem 4.11 (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 $\mathcal{P}(A)$.  

20
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. \( \square \)

**Corollary 4.12** \(( \mathcal{P}(\mathbb{N}) \text{ is uncountable})\).

The set \( \mathcal{P}(\mathbb{N}) \) is uncountable.

**Corollary 4.13** \((\text{The set of languages is uncountable})\).

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 4.8 (\( \Sigma^* \text{ is countable})\)). So by Theorem 4.5 (Characterization of countably infinite sets), we know \(|\Sigma^*| = |\mathbb{N}|\). Theorem 4.11 (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. \( \square \)

**Definition 4.14** \((\Sigma^\infty)\).

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 4.15** \((\{0,1\}^\infty \text{ is uncountable})\).

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 4.12 (\( \mathcal{P}(\mathbb{N}) \text{ is uncountable})\). 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></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(1) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
</tr>
<tr>
<td>( f(2) )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>\ldots</td>
</tr>
<tr>
<td>( f(3) )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
</tr>
<tr>
<td>( f(4) )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>\ldots</td>
</tr>
<tr>
<td>( f(5) )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>\ldots</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td></td>
<td></td>
<td></td>
<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. \qed
Chapter 5

Undecidable Languages
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 $\mathcal{P}(\Sigma^*)$. By Corollary 4.13 (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 4.9 (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. \[\square\]

5.2 Examples of Undecidable Languages

**Definition 5.2** (Halting problem).

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

**Theorem 5.3** (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:

<table>
<thead>
<tr>
<th>$M$: TM.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{\text{TURING}}(\langle M \rangle)$:</td>
</tr>
<tr>
<td>1 Run $M_{\text{HALTS}}(\langle M, M \rangle)$.</td>
</tr>
<tr>
<td>2 If it accepts, go into an infinite loop.</td>
</tr>
<tr>
<td>3 If it rejects, accept.</td>
</tr>
</tbody>
</table>

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. \[\square\]

**Definition 5.4** (Languages related to encodings of TMs).

We define the following languages:

$\text{ACCEPTS} = \{ \langle M, x \rangle : M$ is a TM that accepts the input $x \}$,
Theorem 5.5 (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 \( M_{\text{ACCEPTS}} \) be a decider for it. We will use this decider to come up with a decider for HALTS. Since HALTS is undecidable (Theorem 5.3 (Turing’s Theorem)), this argument will allow us to reach a contradiction.

Here is our decider for HALTS:

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

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 ACCEPTS is undecidable.

\[\square\]

Theorem 5.6 (EMPTY is undecidable).
The language EMPTY is undecidable.

Proof. We want to show that EMPTY is undecidable. The proof is by contradiction, so suppose EMPTY is decidable, and let \( M_{\text{EMPTY}} \) be a decider for it. Using this decider, we will construct a decider for ACCEPTS. However, we know that ACCEPTS is undecidable (Theorem 5.5 (ACCEPTS is undecidable)), so this argument will allow us to reach a contradiction.

We construct a TM that decides ACCEPTS as follows.
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 \( M_{\text{EMPTY}}(\langle M' \rangle) \) 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 \( M_{\text{EMPTY}}(\langle M' \rangle) \) 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.\( \square \)

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

Proof. The proof is by contradiction, so assume EQ is decidable, and let \( M_{\text{EQ}} \) be a decider for it. Using this decider, we will construct a decider for EMPTY. However, EMPTY is undecidable (Theorem 5.6 (EMPTY is undecidable)), so this argument allows us to reach the desired contradiction.

We construct a TM that decides EMPTY as follows.

\[
\begin{align*}
M & \colon \text{TM.} \\
M_{\text{EMPTY}} & \colon \\
1 & \text{Construct the string } \langle M' \rangle \text{ where } M' \text{ is a TM that rejects every input.} \\
2 & \text{Run } M_{\text{EQ}}(\langle M, M' \rangle). \\
3 & \text{If it accepts, accept.} \\
4 & \text{If it rejects, reject.}
\end{align*}
\]

It is not difficult to see that this machine indeed decides EMPTY. Notice that \( L(M') = \emptyset \). So when we run \( M_{\text{EQ}}(\langle M, M' \rangle) \) on line 2, we are deciding whether \( L(M) = L(M') \), i.e. whether \( L(M) = \emptyset \).\( \square \)

5.3 Undecidability Proofs by Reductions

Theorem 5.8 (HALTS \( \leq \) EMPTY).
HALTS \( \leq \) EMPTY.
Proof. (This can be considered as an alternative proof of Theorem 5.6 (EMPTY is undecidable).) We want to show that deciding HALTS reduces to deciding EMPTY. For this, we assume EMPTY is decidable. Let \( M_{\text{EMPTY}} \) be a decider for EMPTY. We need to construct a TM that decides HALTS. We do so now.

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

We now argue that this machine indeed decides 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 \( ⟨M, x⟩ \not\in \text{HALTS} \). 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.

\begin{theorem}(EMPTY \leq HALTS).\end{theorem}
\( EMPTY \leq HALTS. \)

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

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

We now argue that this machine indeed decides EMPTY. First consider an input \( ⟨M⟩ \) such that \( ⟨M⟩ \in \text{EMPTY} \). 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 \( M_{\text{HALTS}}(⟨M’, \epsilon⟩) \), it rejects, and so our decider above accepts, as desired.

Now consider an input \( ⟨M⟩ \) such that \( ⟨M⟩ \not\in \text{EMPTY} \). 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 \( M_{\text{HALTS}}(⟨M’, \epsilon⟩) \), it accepts, and our machine rejects and gives the correct answer. \qed
Chapter 6

Time Complexity
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 \( g(n) \).

**Definition 6.2 (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 \( g(n) \).

**Definition 6.3 (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 \( n \geq n_0 \),
\[
cg(n) \leq f(n) \leq Cg(n).
\]
In this case, we say that \( f(n) \) is Theta of \( g(n) \).

**Proposition 6.4 (Logarithms in different bases).**
For any constant \( b > 1 \),
\[
\log_b n = \Theta(\log n).
\]

*Proof.* It is well known that \( \log_b n = \frac{\log_2 n}{\log_2 b} \). In particular \( \log_b n = \frac{\log_2 n}{\log_2 2} = \log_2 n \). Then taking \( c = C = \frac{1}{\log_2 b} \) 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 \( \log_b n = \Theta(\log_2 n) \). \(\square\)

6.2 Worst-Case Running Time of Algorithms

**Definition 6.5 (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 \( T_A : \mathbb{N} \to \mathbb{N} \) defined by
\[
T_A(n) = \max_{\text{number of steps } A \text{ takes on input } x} \text{instances/inputs } x \text{ of length } n.
\]
We drop the subscript \( A \) and just write \( T(n) \) when \( A \) is clear from the context.

---

1The 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.
**Definition 6.6** (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:** \( T(n) = O(2^{n^k}) \) for some constant \( k > 0 \).

**Proposition 6.7** (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 \( x' \). \( \square \)

### 6.3 Complexity of Algorithms with Integer Inputs

**Definition 6.8** (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 \( xy \).
Chapter 7

The Science of Cutting Cake
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., \(\sum_{j \in N} V_i(A_j) = 1\). \(\Box\)

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. \(\Box\)
**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 \( \text{Eval}_2(y, 1) \).\(^1\) Whichever gives the larger value, referee assigns that piece to player 2. The remaining piece is assigned to player 1.\(^2\)

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.

**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.

\(^1\)In fact, just querying \( \text{Eval}_2(0, y) \) is enough.

\(^2\)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.
Theorem 7.7 (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.

$$
[x, y]: \text{interval in } [0, 1]. \quad k: \text{integer in } \{0, 1, 2, \ldots, n\}.
$$

$$
S: \text{subset of } \{1, 2, \ldots, n\} \text{ with } |S| = k.
$$

EP($$([x, y], k, S))$$):

1. If $k = 1$ and $S = \{i\}$ for some $i$, then let $A_i = [x, y]$.
2. Else:
   3. For $i \in S$, let $z_i = \text{Cut}_i(x, \text{Eval}_i(x, y)/2)$.
   4. Sort the $z_i$ so that $z_{i_1} \leq z_{i_2} \leq \cdots \leq z_{i_k}$. Let $z^* = z_{i_{k/2}}$.
   5. Run EP($$([x, z^*], k/2, \{i_1, \ldots, i_{k/2}\})$$).
   6. Run EP($$([z^*, y], k/2, \{i_{k/2+1}, \ldots, i_k\})$$).

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. \( \square \)

Theorem 7.8 (Edmonds-Pruhs Theorem).

Any algorithm that produces an allocation satisfying the proportionality property must have time complexity \( \Omega(n \log n) \).
Chapter 8

Introduction to Graph Theory
8.1 Basic Definitions

**Definition 8.1** (Undirected graph).
An undirected graph\(^1\) \(G\) 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 \(u, v \in V\).

**Definition 8.2** (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 \(deg(v)\).

**Definition 8.3** (d-regular graphs).
A graph \(G = (V, E)\) is called \(d\)-regular if every vertex \(v \in V\) satisfies \(deg(v) = d\).

**Theorem 8.4** (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 put 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 \(\sum_{v \in V} deg(v) = 2m\). \(\square\)

**Definition 8.5** (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 \(v_0, v_1, \ldots, v_{k-1}, v_0\).

\(^1\)Often the word “undirected” is omitted.
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 8.6** (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 8.7** (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.

\[m = n - 1 \implies G\] is acyclic: If \(m = n - 1\), 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.

\[G\] is acyclic \(\implies m = n - 1\): To prove this direction (using the contrapositive), assume \(m > n - 1\). We know that (i) can happen at most \(n - 1\) times. So in at least one of the steps, (ii) must happen. This implies \(G\) contains a cycle.
Definition 8.8 (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.

Definition 8.9 (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 \( u = v \).

Definition 8.10 (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 \( \text{deg}_\text{out}(u) \), is \( |N(u)| \). The in-degree of \( u \), denoted \( \text{deg}_\text{in}(u) \), 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.

8.2 Graph Algorithms

8.2.1 Graph searching algorithms

Definition 8.11 (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.

\[
G = (V, E): \text{graph. } s: \text{vertex in } V. \\
\text{AFS}((G, s)): \\
1 \text{ Put } s \text{ into bag.} \\
2 \text{ While bag is non-empty:} \\
3 \quad \text{Pick an arbitrary vertex } v \text{ from bag.} \\
4 \quad \text{If } v \text{ is unmarked:} \\
5 \quad \quad \text{Mark } v. \\
6 \quad \quad \text{For each neighbor } w \text{ of } v: \\
7 \quad \quad \quad \text{Put } w \text{ into bag.}
\]

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 \((\bot, s)\).
Definition 8.12 (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.

Definition 8.13 (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.

8.2.2 Minimum spanning tree

Definition 8.14 (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 $c : E \to \mathbb{R}^+$. 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.\footnote{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.} For convenience, we’ll assume that the edges have unique edge costs, i.e. $e \neq e' \implies c(e) \neq c(e')$.

Theorem 8.15 (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 $c(e) < c(e')$).
Theorem 8.16 (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 $c : E \to \mathbb{R}^+$:

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 $V'$.
4. Output $E'$.

By Theorem 8.15 (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.

8.2.3 Topological sorting

Definition 8.17 (Topological order of a directed graph).
A topological order of an $n$-vertex directed graph $G = (V, A)$ is a bijection $f : V \to \{1, 2, \ldots, n\}$ such that if $(u, v) \in A$, then $f(u) < f(v)$.

Definition 8.18 (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 8.19 (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.

\[\begin{array}{c}
\text{Proof.} \\
\text{By contrapositive: If a directed graph has no sink vertices, then it means that every vertex has an outgoing edge. Start with any vertex, and follow an outgoing edge to arrive at a new vertex. Repeat this process. At some point, you have to visit a vertex that you have visited before. This forms a cycle.}
\end{array}\]

\[\begin{array}{c}
\text{Theorem 8.20 (Topological sort via DFS).} \\
\text{There is an } O(n + m) \text{-time algorithm that solves the topological sorting problem.}
\end{array}\]

Proof. The algorithm is a slight variation of DFS.

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

\[\begin{array}{l}
G = (V, A): \text{directed graph.} \\
v: v \in V. \\
\text{DFS'}(\langle G, v \rangle):
\begin{align*}
1 & \quad \text{Mark } v \text{ as “visited”;} \\
2 & \quad \text{For each neighbor } u \text{ of } v: \\
3 & \quad \text{If } u \text{ is not marked visited:} \\
4 & \quad \text{Run DFS}(\langle G, u \rangle). \\
5 & \quad f(v) = p. \\
6 & \quad p = p - 1.
\end{align*}
\end{array}\]

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 \(f(u) < f(v)\).