CS 3800 - Lecture Notes

Lucas Sta. Maria
stamaria.l@northeastern.edu

October 2, 2023

Contents

Preface

These are my compiled lecture notes for *CS 3800 - Theory of Computation* at Northeastern University for the Fall 2023 semester. The core content of these notes are taken during lectures, and then supported with additional definitions and examples from the textbook. These notes are written to be more a reference than pedagogic material.

The textbook for this course is *Introduction to the Theory of Computation*, 3rd edition, by Michael Sipser. It's a fairly nice textbook, and if you're enrolled in the course I do recommend reading it. I took this class with the professor Daniel Wichs.

The latest version of these notes are accessible through `https://files.priime.dev/toc.pdf` directly. You can additionally view notes I've taken for other classes at `https://priime.dev/notes`.

If you're interested in my workflow, I use Emacs with the AucTex and CDLaTeX packages to quickly typeset the document. It is efficient enough to keep up during lectures.

There are bound to be mistakes and inaccuracies – if you identify them, feel free to share corrections to my email.

– Lucas

1 LECTURE 1

Definition 1.1. *Theory of computation* is about understanding the fundamental capabilities and limitations of computers.

Definition 1.2. Informal: A *computation* is a clearly specified procedure consisting of simple operations.

Definition 1.3. Informal: *Automata theory* is about modeling computation without memory.

- The *finite automaton* model is used in text processing
- The *context-free grammar* model is used in programming languages and artificial intelligence.

Definition 1.4. Informal: A *finite automata* is a machine with a fixed finite number of states that does not have memory. It is simple and elegant, modeling simple physical devices.

Definition 1.5. Informal: A *turing machine* is similar to a finite automata but with an unbounded length of memory tape. Turing machines can compute everything that any known computer can, using roughly the same number of steps.

Definition 1.6. Informal: *Computability* is about problems that can and cannot be computed.

- Can we decide whether programs terminate or run forever?
- Can we determine the truthiness of mathematical statements?

Definition 1.7. Informal: *Complexity theory* is about what can be computed efficiently.

- Are there problems where verifying a solution is efficient, but finding a solution is not?
- What makes some problems computationally hard and others easy?

Aside 1.8. What options do we have when we encounter a computationally hard problem?

- By understanding which aspect of the problem is at the root of the difficulty, you may be able to alter it so that the problem is more easily solvable.
- You may be able to settle for less than a perfect solution of the problem (*approximation*).
- Some problems are hard only in the worst case situation, but easy most of the time. In other words, a solution can run occasionally slow, but on average at a reasonable speed. You may use this solution if you can tolerate the occasional slowness.
- You may consider alternative types of computation, such as randomized computation, that can speed up certain tasks.

Aside 1.9. In complexity theory, the objective is to classify problems as easy ones and hard ones; whereas in computability theory, the classification of problems is by those that are solvable and those that are not.

Definition 1.10. Let Σ (or also Γ) be a nonempty finite set, called the *alphabet*. The elements of Σ are called *symbols*.

The strings over Σ are tuples written as $w = w_1 w_2 ... w_n$ where each $w_j \in \Sigma$.

Definition 1.11. A *string over an alphabet* is a finite sequence of symbols from that alphabet, usually written next to one another and not separated by commas.

Example 1.12. Consider the alphabet $\Sigma_1 = \{0, 1\}$.

Let $w_1 = 01001$. Then, w_1 is a string over Σ_1.

Let $w_2 = 01201$. w_2 is not a string over Σ_1, since 2 is not in the alphabet Σ_1.

Definition 1.13. The *length* of a string w is denoted $|w|$. There is a special string of length 0, called the empty string and denoted by ϵ.

Definition 1.14. The set of strings over Σ as

$$\Sigma^* = \cup_i \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \cdots$$

where $\Sigma^0 = \{\epsilon\}$

Definition 1.15. A string v is a *substring* of w if its symbols appear consecutively in w.

The empty string ϵ is a substring of every string. Any string v is a substring of itself.

Aside 1.16. This class will focus on computing functions

$$f : \Sigma^* \Rightarrow \{\text{accept, reject}\}$$

We want inputs to be strings since we can encode any input as a string.

We want outputs to be boolean because

- Simplicity: already captures many interesting functions
- Powerful enough that most results will easily generalize

Definition 1.17. A *language* L over the alphabet Σ is a subset $L \subseteq \Sigma^*$. A language may be finite or infinite.

An equivalence exists between languages $L \subseteq \Sigma^*$ and functions $f : \Sigma^* \Rightarrow \{0, 1\}$:

- Given f, we can define $L = \{w : f(w) = \text{ accept }\}$.
- Given L, we can define $f(w) = \text{ accept }$ iff $w \in L$.

Computing $f(w)$ is the same as *deciding* if $w \in L$.

Definition 1.18. *Deciding* a language L means giving a "program" (e.g. automaton, Turing Machine, ...) that *accepts* inputs $w \in L$ and rejects $w \notin L$.

Aside 1.19. Our Goal

Study different models of computation and ask, what languages can we decide/compute in this model?

Aside 1.20. Observation

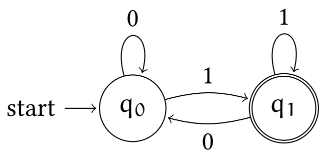
- Languages can be finite or infinite.
- In each of our models of computation, a "program" has a finite description.
- A program that decides a language, gives us a finite description of a language which might be infinite.

2 LECTURE 2

Definition 2.1. *Deterministic finite automata* are a very simple and limited model of computation, consisting of *states* and *transitions*. Transitions are labeled with symbols of the alphabet Σ. For each state, there is exactly one outgoing transition labeled with each alphabet symbol.

Definition 2.2. A *state diagram* is a graphical representation of a finite automaton, explicitly showing the automatons states and transitions.

Example 2.3. Simple example of a state diagram representing a finite automaton that accepts input if terminating in 1. q_1 is labeled as an *accepting* state, while q_0's lack of inner ring indicates it is a *rejecting* state. The output is either accept or reject, depending on the state after a string's last symbol has been read.

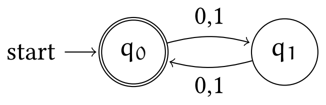


Definition 2.4. A finite automaton has a unique *start state*. Some of the states are *accept states*. We can label the states $q_0, q_1, \ldots$.

Definition 2.5. We can *execute* the automaton on any string $w \in \Sigma^*$.

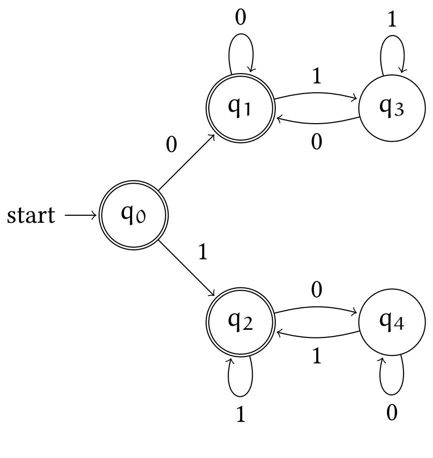
- Begin in start state.
- For each symbol of w, follow the corresponding transition.
- If we end up in an *accept state* then *accept*, else *reject*.

Example 2.6. Simple example that accepts input if the input string is of even length.



The input 0110 is accepted, but the input 01100 is not.

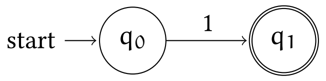
Example 2.7. An automaton that only accepts input if the input is ϵ or the start of the input is the same as the end of the input.



Definition 2.8. We define the language $L(M) = \{w \mid M \text{ accepts } w\}$. We say that M *recognizes* the language A if $A = L(M)$.

Definition 2.9. A *transition function* $\delta : Q \times \Sigma \Rightarrow Q$ defines the rules for moving between states.

Example 2.10. Consider the following state diagram, where the current state is q_0.



If an automaton is at state q_0 and reads the symbol 1, then it transitions to state q_1. This is represented by the transition function $\delta(q_0, 1) = q_1$.

Definition 2.11. A *deterministic finite automaton* consists of a tuple $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$ where:

- Q is a finite set called the *states.*
- Σ is a finite set called the *alphabet.*
- $\delta : Q \times \Sigma \Rightarrow Q$ is the *transition function.*
- $q_{\text{start}} \in Q$ is the *start state.*
- $F \subseteq Q$ is the set of *accept states (final states).*

Definition 2.12. Let $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$ be a finite automaton and let $w \in \Sigma^*$ be a string consisting of symbols $w = w_1, \dots, w_n$ with $w_i \in \Sigma$. We define the *extended transition function* $\delta^* : Q \times \Sigma^* \Rightarrow Q$ as:

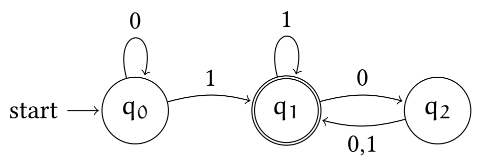
- $\delta^*(q, w) = q$ if $n = 0$ (i.e., $w = \epsilon$)
- $\delta^*(q, w) = \delta(\delta^*(q, w_1 \dots w_n, w_n))$ otherwise

Definition 2.13. A finite automaton $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$ *accepts* $w \in \Sigma^*$ if $\delta^*(q_{\text{start}}, w) \in F$ and otherwise M *rejects* w.

Definition 2.14. If A is the set of all strings that a finite automaton M accepts, then A is the *language of machine* M and is denoted by $L(M) = A$. We say that M *recognizes* A or M *accepts* A.

Aside 2.15. Since the verb "accept" is also used to describe individual strings that finite automata accept, it is preferable to use the verb "recognize" when describing the languages that finite automata accept.

Example 2.16. Consider the following finite automaton:



We can formally describe this finite automaton as

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. δ is described as

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

4. q_0 is the start state
5. $F = \{q_1\}$

Definition 2.17. The *language* of an automaton M is $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

Definition 2.18. A language L is *regular* if there exists some finite automaton M that recognizes L.

Finite automata yield a nice and simple algorithm to decide membership in regular languages:

- Fixed finite amount of memory.
- $O(n)$ time, single pass over the input (streaming).

Aside 2.19. Our Goals

- Can we characterize regular languages?
- Are there languages that aren't regular?

Example 2.20. A non-regular language would be $L = \{0^n 1^n \mid n \leq 0 \in \mathbb{Z}\}$, since counting requires memory.

Example 2.21. Composing Regular Languages

Suppose A and B are *regular* languages. Can we compose them to form other *regular* languages?

- The *union* of regular languages $A \cup B$
- The *concatenation* of regular languages $A \circ B$
- The *star* of a regular language A^*
- The *complement* of a regular language $\bar{A}$
- The *intersection* of regular languages $A \cap B$

Definition 2.22. Regular languages are *closed* under the *union, concatenation, star, complement,* and *intersection* operations. If you apply these operations to regular languages, you get a regular language.

Theorem 2.23. If A is a regular language, then so is $\bar{A}$.

Proof. Let $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$ be an automaton that recognizes A. Let $M' = Q, \Sigma, \delta, q_{\text{start}}, F' = \bar{F}$.

$$\begin{aligned} M \text{ accepts } w &\iff \delta^*(q_{\text{start}}, w) \in F \\ &\iff \delta^*(q_{\text{start}}, w) \notin F' \\ &\iff M' \text{ does not accept } w \end{aligned}$$

Therefore M' recognizes $\bar{A}$, and $\bar{A}$ is regular. □

3 LECTURE 3

Example 3.1. Can we show that regular languages are closed under concatenation? Consider a simple example.

Let $A = \{w \mid w \text{ contains a } 0\}$, and let $B = \{w \mid w \text{ contains exactly two 1s}\}$. Let $a = 01010 \in A$ and $b = 11 \in B$. Then, $a \circ b = 0101011$.

Yes, since 0101011 is in $A \circ B$.

Aside 3.2. Our Plan

We define a more flexible notion of automata called *non-deterministic finite automata* (NFA).

- multiple options for the "next state", guess which to take

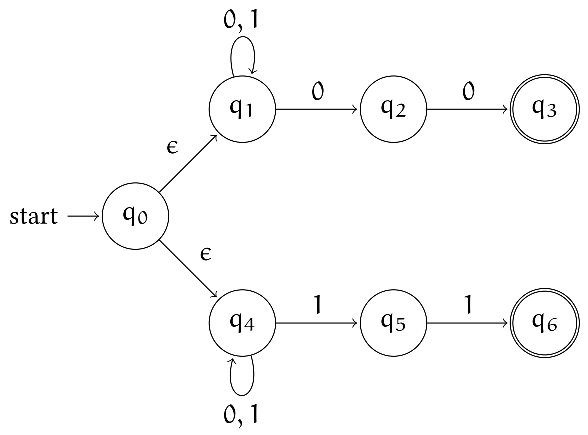
To distinguish, we will call the standard automata *deterministic finite automata* (DFA).

Can always convert NFA to DFA.

Definition 3.3. A *non-deterministic finite automata* introduces transitions that can be labeled with ϵ, along with states that can have an arbitrary number of outgoing transitions with each symbol.

The NFA *accepts* a string w if there is *some way* to execute the computation that ends in the accept state.

Example 3.4. Consider the following NFA:



Let $w = 01100$. There exists an execution that ends in an accept state: $q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{0} q_1 \xrightarrow{1} q_1 \xrightarrow{1} q_1 \xrightarrow{0} q_2 \xrightarrow{0} q_3$. Thus, the NFA accepts w.

The $L(M)$ for the above NFA M is $L(M) = \{w \mid w \text{ ends with 00 or 11}\}$.

Definition 3.5. A *non-deterministic finite automata* consists of a tuple $M = (Q, \Sigma, \delta, q_{\text{start}}, F)$ where:

- Q is a finite set called the *states.*
- Σ is a finite set called the *alphabet.*
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \Rightarrow \mathcal{P}(Q)$ is the *transition function.*
- $q_{\text{start}} \in Q$ is the *start state.*
- $F \subseteq Q$ is the set of *accept states (final states).*

Definition 3.6. The ϵ-closure of a state q is defined as $E(q) = \{q' : q' \text{ reachable from } q \text{ via epsilon-transitions}\}$.

The ϵ-closure of a set $P \subseteq Q$ is defined as $E(P) = \cup_{p \in P} E(p)$.

Definition 3.7. The *extended transition function* $d^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ is defined as follows:

- $\delta^*(q, w) = E(q)$ if $|w| = 0$
- $\delta^*(q, w) = E\left(\cup_{p \in \delta^*(q, w_1, \ldots, w_{n-1})} \delta(p, w_n)\right)$ otherwise

Definition 3.8. An NFA M *accepts* a string $w \in \Sigma^*$ if $\delta^*(q_{\text{start}}, w) \cap F \neq \varnothing$. Else we say that M *rejects* w.

Definition 3.9. The *language* of an NFA M is $L(M) = \{w \in \Sigma^* | M \text{ accepts } w\}$.

Theorem 3.10. For any NFA N there exists a DFA D such that $L(D) = L(N)$.

Corollary 3.10.1. A language L is regular if and only if there exists an NFA that recognizes L.

Theorem 3.11. If A and B are regular languages then so is $A \cup B$.

Proof. Let M_A and M_b be DFAs that recognizes the languages A and B respectively. We shall construct an NFA N that recognizes $A \cup B$.

Add a new start state to N. Add ϵ-transitions from the new start state to the states of M_A and M_B.

Since $A \cup B$ is recognized by an NFA, then $A \cup B$ is regular by 3.10.1. □

Theorem 3.12. If A and B are regular languages then so is $A \circ B$.

Proof. Let M_A and M_b be DFAs that recognizes the languages A and B respectively. We shall construct an NFA N that recognizes $A \circ B$.

We add ϵ-transitions from the accepting states of M_A to the start state of M_B. Let the start state of N be the start state of M_A, and the accept states of N be the accept states of M_B.

Since $A \circ B$ is recognized by an NFA, then $A \circ B$ is regular by 3.10.1. □

Theorem 3.13. If A is a regular language then so is A^*.

Proof. Let M_A be a DFA that recognizes the language A. We shall construct an NFA N that recognizes A^*.

Add a new start state which is also an accepting state. Add an ϵ-transition from the new start state to the old start state. Add ϵ-transitions from the accept states to the old start state. □

Theorem 3.14. If A and B are regular languages then so is $A \cap B$.

Proof. Follows by De Morgan's laws: $A \cap B = \overline{\bar{A} \cup \bar{B}}$. □

Aside 3.15. Intutition for $\mathsf{NFA} \Rightarrow \mathsf{DFA}$.

- NFA can have many valid computation sequences. After reading w, can *reach* a subset of the states Q.
- States of DFA correspond to subsets of NFA states.
  - NFA states: $Q_{\text{NFA}} = \{Q_1, \ldots, Q_n\}$
  - DFA states: $Q_{\text{DFA}} = \mathcal{P}(Q_{\text{NFA}})$

Want: After reading w: DFA in state $S \iff$ NFA can reach states $q_i \in S$.

We can set the start state of the DFA to be $E(q_0)$ where q_0 is the start state of the NFA. We can set the accept states of the DFA to be $F_{\text{DFA}} = \{S \mid S \cap F_{\text{NFA}} \neq \varnothing\}$. We make the transitions $\delta_{\text{DFA}}(S, x) = E\left(\bigcup_{q \in S} \delta_{\text{NFA}}(q, x)\right)$. Some states in the DFA are unreachable – we can delete them.

Summary: $N = (Q, \Sigma, \delta_{\text{NFA}}, q_{\text{start}}, F)$ is transformed to $D = Q', \Sigma, \delta_{\text{DFA}}, q'_{\text{start}}, F'$.

4 LECTURE 4

Lemma 4.1. For all $w \in \Sigma*: \delta^*_{\text{NFA}}(q_{\text{start}}, w) = \delta^*_{\text{DFA}}(q'_{\text{start}}, w)$.

Proof. Proof by induction on length of $w = w_1, \ldots, w_n$.

- Base case $n = 0$:

$$\begin{aligned}\delta^*_{\text{NFA}}(q_{\text{start}}, \epsilon) &= E(q_{\text{start}}) \\ &= q'_{\text{start}} \\ &= \delta^*_{\text{start}}\left(q'_{\text{start}}, \epsilon\right)\end{aligned}$$

- Induction:

$$\begin{aligned}\delta^*_{\text{NFA}}(q_{\text{start}}, w) &= E\left(\cup_{p \in \delta^*_{\text{NFA}}(q_{\text{start}}, w_1, \ldots, w_{-1})} \delta_{\text{NFA}}(p, w_n)\right) \\ &= E\left(\cup_{p \in \delta^*_{\text{DFA}}(q'_{\text{start}}, w_1, \ldots, w_{-1})} \delta_{\text{NFA}}(p, w_n)\right) \\ &= \delta_{\text{DFA}}\left(\delta^*_{\text{DFA}}(q'_{\text{start}}, w_1, \ldots, w_{n-1})\right) \\ &= \delta^*_{\text{DFA}}(q'_{\text{start}}, w)\end{aligned}$$

□

Theorem 4.2. For all $w \in \Sigma^*$, N accepts $w \iff$ D accepts w.

Proof.

$$\begin{aligned}\text{N accepts } w &\iff \delta^*_{\text{NFA}}(q_{\text{start}}, w) \cap F \neq \emptyset \\ &\iff \delta^*_{\text{DFA}}(q'_{\text{start}}, w) \cap F \neq \emptyset \\ &\iff \delta^*_{\text{DFA}}(q'_{\text{start}}, w) \in F' \\ &\iff \text{D accepts } w\end{aligned}$$

□

Theorem 4.3. For any NFA N, there exists a DFA D such that $L(D) = L(N)$.

Proof. Let $N = (Q, \Sigma, \delta_{\text{NFA}}, q_{\text{start}}, F)$ be an NFA. Let $D = (Q', \Sigma, \delta_{\text{DFA}}, q'_{\text{start}}, F')$ be a DFA defined as:

- $Q' = \mathcal{P}(Q)$
- $q'_{\text{start}} = E\left(q_{\text{start}}\right)$
- $F' = \{S \mid S \cap F \neq \emptyset\}$
- $\delta_{\text{DFA}}(S, x) = E\left(\cup_{q \in S} \delta_{\text{NFA}}(q, x)\right)$

By above lemma and theorem, so it follows that $L(D) = L(N)$. □

Aside 4.4. Ways to specify a regular language:

- Build a DFA or NFA
- Use the regular operations to build a "complex" language from "simple" ones.

Example 4.5. We can compose the following atomic expressions with operations to define regular expressions.

- Atomic expressions: $\emptyset, \epsilon, \Sigma$
- Operations: $*, \circ, \cup$

Definition 4.6. R is a regular expression over an alphabet Σ if:

- It's an atomic expression: $R = \emptyset$ or $R = \epsilon$ or $R = a$ for some $a \in \Sigma$

- It's composition: If R_1, R_2 are regular expressions then so is $R = R_1 \cup R_2$ or $R = (R_1 \circ R_2)$ or $R = R_1^*$ regular expressions of complexity $max(n_1, n_2) + 1$.

Definition 4.7. The language of a regular expression R, denoted $L(R)$, is defined inductively as:

- Atomic expression: $L(\emptyset) = \emptyset$, $L(\epsilon) = \epsilon$, $L(a) = a$ for $a \in \Sigma$
- Composed expression: If R_1 and R_2 are regular languages,

$$\begin{aligned} L(R_1 \cup R_2) &= L(R_1) \cup L(R_2) \\ L(R_1 \circ R_2) &= L(R_1) \circ L(R_2) \\ L(R_1^*) &= L(R_1)^* \end{aligned}$$

Aside 4.8. Notation is complex, we can simplify it.

- Omit parenthesis when unnecessary.
  - Precedence: $*$ precedes $\circ$ precedes U
- Instead of $R_1 \circ R_2$, write $R_1 R_2$.
- Write Σ instead of $\cup_{a \in \Sigma} a$.
- Write R^+ to denote RR^*.
- Write R^n to denote n concatenations of R.

Theorem 4.9. A language A is regular *if and only if* there is a regular expression R such that $A = L(R)$.

Aside 4.10. Ways of specifying A is regular:

- DFAs
- NFAs
- Regular expressions

$$\begin{aligned} A \text{ is regular } &\iff \exists \text{ DFA } D : A = L(D) \\ &\iff \exists \text{ NFA } N : A = L(N) \\ &\iff \exists \text{ RE } \quad R : A = L(R) \end{aligned}$$

5 LECTURE 5

Definition 5.1. A *generalized nondeterministic finite automaton* (*GNFA*) are NFAs wherein the transition arrows may have any regular expressions as labels. It also has additional rules for convenience (a "special form"):

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state.
- The start state and accept state are different states.
- Except for the start and accept state, one arrow goes from every state to every other state and also from each state to itself.

Theorem 5.2. A DFA can be converted into a GNFA in the special form.

Aside 5.3. Not all languages are regular, for example $L = \{0^n 1^n : n \geq 0\}$. We cannot construct a DFA for L, since it would need to remember how many zeroes it has seen.

Theorem 5.4. $L = \{0^n 1^n : n \geq 0\}$ is not regular.

Proof. Assume by contradiction there is a DFA $M = (Q, \Sigma, \delta, F)$ that recognizes L. Let $|Q| = p$ (the number of states).

Let $r_k \in Q$ be the state M reaches after reading 0^k. Then for some $0 \leq i < j \leq p$ we have $r_i = r_j$.

M must *accept* $0^i 1^i \in L$, thus M *accepts* $0^j 1^i \notin L$. □

Definition 5.5. Pumping Lemma (Slides Definition)

If L is a regular language, then:

There exists some integer $p \geq 0$ for all strings $w \in L$ of length $|w| \geq p$ such that there exists strings x, y, z where $w = xyz$, for $|y| > 0$ and $|xy| \leq p$ where for all integers $i \geq 0 : xy^i z \in L$.

p is the *pumping length.*

Definition 5.6. Pumping Lemma (Textbook Definition)

If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Aside 5.7. The Pumping Lemma states that all regular languages have a special property. If we can show that a language does not have this property, then we are guaranteed that the language is not regular. The property states that all strings can be "pumped" if they are at least as long as a certain value, called the *pumping length.*

Aside 5.8. To use the pumping lemma to prove that a language L is not regular, first assume that L is regular to obtain a contradiction. Then using the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in L cannot be pumped. Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x, y and z, and, for each division, finding a value i where $xy^i z \notin L$. The existence of s contradicts the pumping lemma if L were regular, hence L cannot be regular.

Theorem 5.9. $L = \{0^n 1^n : n \geq 0\}$ is not regular.

Proof. Proof using the pumping lemma.

- Adversary picks integer $p \geq 0$
- You pick $w = 0^p 1^p$

- Adversary picks x, y, z. Since $|xy| \leq p$, y contains just 0s.
- You pick $i = 2$, where $xy^2z = xyyz = 0^{p+|y|}1^p \notin L$. You win

$\square$

Theorem 5.10. $L = \{ww : w \in \{0,1\}^*\}$ is not regular.

Proof. Proof using the pumping lemma.

- Adversary picks integer $p \geq 0$
- You pick $w = 0^p10^p1$
- Adversary picks x, y, z. Since $|xy| \leq p$, y contains just 0s
- You pick $i = 2$, since $xy^2z = 0^{p+|y|}10^p1 \notin L$

$\square$

6 LECTURE 6

Aside 6.1. The big questions:

- Can we rigorously define a general notion of an algorithm?
- Are there problems that cannot be solved by *any* algorithm?

Aside 6.2. In 1928, David Hilbert asked for an "algorithm" that takes in as input a mathematical statement and decides whether the statement is true or false. There was shown to be no algorithm:

- Kurt Gödel relied on *recursive functions*
- Alonzo Church developed λ-*calculus*
- Alan Turing developed the *Turing Machine*

All these definitions are equivalent.

Aside 6.3. Our Plan:

- Define Turing Machines
- Show Turing Machines are powerful enough to implement any "reasonable algorithm".
- Show that some problems are "undecidable".

Definition 6.4. (Informal) A *Turing Machine* initially has an infinite memory tape, containing the input and followed by "blanks", with the tape head starting at the left-most position. In each step, the machine can overwrite the symbol under the tape head and move the tape left/right. At any point in time, the machine can halt the computation and *accept* or *reject*.

Aside 6.5. The Turing Machine is like a DFA. Transitions read or write under the tape head, and can move the tape head left or right. There are special accept and reject states. If the computation enters such a state, it immediately halts.

Definition 6.6. A *Turing Machine* consists of a tuple $M = \{Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}\}$ where:

- Q is a finite set called the *states*.
- Σ is a *input alphabet*.
- Γ is a *tape alphabet* such that $\Sigma \subseteq \Gamma$ and Γ contains a special blank symbol which is not in Σ.
- $q_{\text{start}} \in Q$ is the *start state*.
- $q_{\text{accept}} \in Q$ is the *accept state*.
- $q_{\text{reject}} \in Q$ is the *reject state*.
- $\delta : Q' \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*, where $Q' = Q - q_{\text{accept}}, q_{\text{reject}}$.

Definition 6.7. Let $M = Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}$ be a Turing Machine.

A *configuration* of M is a tuple $C = (u, q, v)$ such that $u, v \in \Gamma^*$ and $q \in Q$.

A configuration C *yields* C' if M goes from C to C' in 1 step.

A *start configuration* of M on input w is $q_{\text{start}}w$.

An *accepting* configuration is one where the state is q_{accept}.

Definition 6.8. Let $M = Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}}$ be a Turing Machine.

M *accepts* w if there is a sequence of configurations $C_1, C_2, \cdots, C_n$ such that:

- C_1 is the start configuration of M on input w
- C_i yields C_{i+1} for $i = 1, \ldots, n-1$
- C_n is an *accepting* state configuration

Definition 6.9. A Turing Machine M on input w can either *accept*, *reject*, or *loop*. Accepting or rejecting is known as *halting*.

Definition 6.10. For a Turing Machine M, we define $L(M) = \{w \mid M \text{ accepts } w\}$. Say that M *recognizes* the language $L(M)$.

Definition 6.11. We say that M *decides* L if

- M accepts $w \in L$ and M rejects $w \notin L$.
- Equivalently: M recognizes L and M always halts.

Definition 6.12. A language L is *recognizable* if there is some turing machine that recognizes L.

Definition 6.13. A language L is *decidable* if there is some turing machine that decides L.

Definition 6.14. A *tape-head level description* is a way to describe how the tape-head should walk across the tape and what it should write.

7 LECTURE 7

Example 7.1. Consider $L = \{w\#w : w \in 0, 1^*\}$. Let's give a tape-head level description for it.

- Check input is of form $0, 1^*\#0, 1^*$, otherwise reject.
- Walk tape-head from left to right:
  - If everything crossed out other than #, accept.
  - Find first non crossed out $b \in 0, 1$ on the left of # and the first non crossed out $b' \in 0, 1$ to the right of #
    - If one of b or b' doesn't exist, reject
    - If $b \neq b'$, $\texttt{reject}$
  - Return tape-head all the way to the left, and repeat

Aside 7.2. Can also consider Turing Machines that output more than just "accept/reject". Define the output of a turing machine as the contents of its tape when it enters an accept/reject state.

Definition 7.3. A turing machine M computes a function $f : \Sigma^* \to \Sigma^*$ if on every input $w \in \Sigma^*$ the turing machine halts and its tape contains $f(w)$. We say that f is *computable* if some turing machine M computes it.

Can check that basic functions such as addition, multiplication, division, etc. are computable.

Aside 7.4. We want to show that adding various features to turing machines does not make them any more powerful. We can build a compiler that takes a turing machine with additional features and converts the turing machine to a standard turing machine.

Aside 7.5. We can create a compiler from a multi-tape turing machine to a one tape turing machine.

- Store contents of all tapes sequentially on a single tape separated by #.
- Remember tape-head positions by storing an "underlined" version of tape symbols.
- Each step of multi-tape turing machine is simulated by scanning entire tape of a single-tape turing machine.
- Might need more space on one of the tapes – in that case shift everything over by 1 to the right.

Example 7.6. Show that there is a turing machine that computes binary addition for a and b.

- Use two extra tapes. Copy a to the tape t_1, b to t_2 in reverse order and clear the main tape.
- Return all tape heads to the left. Remember 1 bit for the carry.
- Add two bits under tape-heads 1 and 2 plus carry. Place result modulo 2 on the main tape, update the carry. Move all tape heads one right. Repeat until values under tape-head 1 and 2 are both blank.
- Reverse contents of main tape.

Example 7.7. Let's define a random access turing machine, that can read and write to arbitrary locations in memory without scanning a tape. Memory is modeled as an infinite array R. We shall show how to compile this to a standard Turing Machine.

- In addition to standard tape that contains the input the turing machine has location and value tapes.
- There is a special write instruction which sets `R[location] = value` using content of tapes.
- There is a read transition which sets the contents of the value tape to `R[location]`.

TODO: Write the compiler (I'm eating).

Definition 7.8. *Church-Turing Thesis*

Any "algorithm" can be implemented on a Turing Machine.

- This is not a formal statement and hence cannot prove or disprove it.
- Turing Machines capture our informal understanding of what an algorithm is.

Aside 7.9. From now on, to describe a turing machine, we can just describe an algorithm using high-level pseudocode. It should be sufficiently clear that a human can follow, and we should be confident enough in it that we can convert it to a turing machine.

Aside 7.10. Inputs to turing machines are strings, but we can encode anything as a string.

Definition 7.11. For some object O, let $\langle O \rangle$ denote the encoding of O as a string.

Definition 7.12. There is a turing machine $M_{\text{UNIV}}(\langle M \rangle, w)$ that can run any other turing machine. It takes in an input of any turing machine M and any string w. It then runs M on w.

- If M accepts w, then $M_{\text{UNIV}}(\langle M \rangle, w)$ accepts.
- If M rejects w, then $M_{\text{UNIV}}(\langle M \rangle, w)$ rejects.
- If M loops on w, then $M_{\text{UNIV}}(\langle M \rangle, w)$ will also loop.

Think of turing machines as algorithms and a universal turing machine as a *general purpose computer*.

Definition 7.13. A *non-deterministic turing machine* has multiple possible actions the turing machine can take at any point in time. Its transition function is $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. If in state q and read symbol a there are several possible options for the next state, write symbol, and direction.

- The turing machine accepts if there *exists* some way to run the computation that ends in an accept state.
- The turing machine rejects if all computations are rejecting.

Aside 7.14. Why turing machines?

- *Simplicity*: mathematical definitions should be simple.
- *Locality*: will be useful in a few proofs.