Computational Complexity Lecture Notes

Michael Levet

February 22, 2021

Contents

1 Combinatorial Circuits

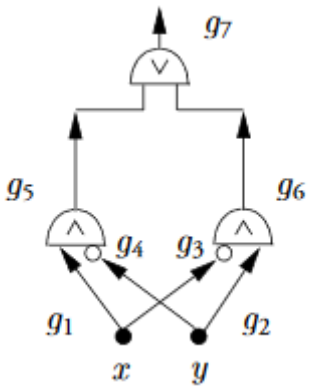
These notes follow closely [Sav97][Chapter 2]. Both [Lev18, Ros12] also served as key references.

1.1 Introduction to Circuits

Definition 1. A *logic circuit* is a directed acyclic graph whose vertices are labeled with the names of Boolean functions or variables (inputs). The vertices labeled with the names of Boolean functions are referred to as *logic gates*.

Remark 2. The above definition does not clearly highlight the difference between a circuit and a logic gate. In practice, logic gates often compute simple functions, such as AND, OR, and NOT. The goal then is to design circuits using these elementary building blocks to compute more complicated Boolean functions.

Example 3. Below is an example of a Boolean circuit. The AND and OR gates are clearly labeled. The NOT gates are denoted by the white circles.



Our goal in this section is to establish that Boolean circuits are a universal model of computation. That is, a function is Turing computable if and only if it is computable by an equivalent family of Boolean circuits. To this end, we seek to construct a way to convert a Turing Machine to an equivalent family of circuits, as well as to construct an equivalent Turing Machine from a given family of circuits. We begin by trying to understand the circuit model of computation, and then proceed to examine the Turing Machine model. Our first step in this translation is to convert Boolean circuits to a procedural algorithmic construct, known as a straight-line program.

Definition 4. A *straight-line program* is a sequence of steps, each of which is of one of the following forms:

- Input Step: $(s\,\texttt{READ}\,x)$, where x denotes an input variable,
- Output Step: $(s\,\texttt{OUTPUT}\,i)$,
- Computation Step: $(s\,\texttt{OP}\,i,\ldots,k)$.

Here, OP is the given operation being performed. Now s is the index or line number of the given step, and $i,\ldots,k$ reference values computed at previous steps. So necessarily, $s \geq i,\ldots,k$.

Example 5. Consider again the circuit from Example 3. We first examine a functional description of the given circuit.

$$
\begin{aligned}
g_1 &:= x \\
g_2 &:= y \\
g_3 &:= \overline{x} \\
g_4 &:= \overline{y} \\
g_5 &:= g_1 \wedge g_4 \\
g_6 &:= g_3 \wedge g_2 \\
g_7 &:= g_5 \vee g_6.
\end{aligned}
$$

In particular, the circuit first reads in the inputs and computes the relevant negations. The two AND gates are executed. Finally, the last OR gate is executed. We note that a circuit would actually execute g_1 and g_2 at the same time. Similarly, g_3 and g_4 would be executed in parallel, as would g_5 and g_6. Note that a straight-line program, as well as an sequential model of computation like a RAM or Turing Machine, would not execute these statements in parallel. When equating two models of computation, the goal is to show that they compute the same family of functions. Two different models need not (and usually, do not) compute a given function in the same manner. We may also ask as to the cost, in time or space, in converting between two models. This is a question we will address later.

Next, we construct the equivalent straight-line program for this circuit.

$$
\begin{aligned}
&(1\ \texttt{READ}\ x)\\
&(2\ \texttt{READ}\ y)\\
&(3\ \texttt{NOT}\ 1)\\
&(4\ \texttt{NOT}\ 2)\\
&(5\ \texttt{AND}\ 1, 4)\\
&(6\ \texttt{AND}\ 3, 2)\\
&(7\ \texttt{OR}\ 5, 6)\\
&(8\ \texttt{OUTPUT}\ 7)
\end{aligned}
$$

Remark 6. Informally, it may seem apparent at this point that every Boolean circuit computes a function of the form $f : \{0,1\}^n \to \{0,1\}^m$. We will formalize this shortly. Perhaps less apparent is that for every Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, there exists a Boolean circuit that computes f. This second direction motivates several questions, including both how to construct such circuits and measures of circuit complexity.

We proceed first by showing that every Boolean circuit computes a function of the form $f : \{0,1\}^n \to \{0,1\}^m$.

Definition 7. Let $\mathcal{C}$ be a Boolean circuit. Let g_s be the function computed by the sth step of the straight-line program $\mathcal{P}$ corresponding to $\mathcal{C}$. If the sth step is $(s\ \texttt{READ}\ x)$, then $g_s = x$. If it is the computation step $(s\ \texttt{OP}\ i, \ldots, k)$, then $g_s = \texttt{OP}(g_i, \ldots, g_k)$, where $g_i, \ldots, g_k$ are the functions computed at steps $i, \ldots, k \leq s$.

If $\mathcal{P}$ has n inputs and m outputs, then $\mathcal{P}$ computes a function $f : \{0,1\}^n \to \{0,1\}^m$. If $s_1, \ldots, s_m$ are the output steps, then $f = (g_{s_1}, \ldots, g_{s_m})$.

Remark 8. In light of this definition, we have that the function computed by the Boolean circuit $\mathcal{C}$ is precisely the function computed by the straight-line program $\mathcal{C}$.

We now turn to showing that every Boolean function can be computed by a family of Boolean circuits. Any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ can be expressed using a truth table. There are 2^n such rows in the truth table. One obvious approach is to build a circuit for each row, and then use AND gates on the outputs for the row circuits to generate the specified outputs for the function. Informally, such a circuit seems expensive, as we would require $\Theta(2^n)$ logic gates. We introduce some measures of circuit complexity to precisely measure the succinctness of a circuit.

1.2 Measures of Circuit Complexity

In order to measure the complexity of a circuit, we must first specify the allowed logic gates. This brings us to the notion of a basis.

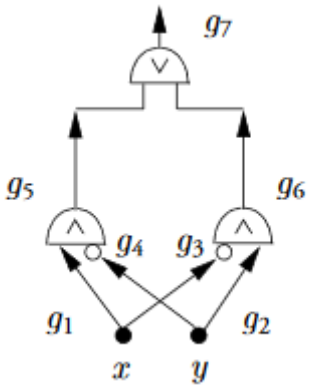
Definition 9. The *basis* Ω of a circuit is the set of operations that it can use. Bases for Boolean circuits may only use Boolean functions. The *standard basis* is $\Omega_0 := \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. We say that a basis Ω is *complete* if every Boolean function can be realized using precisely the operations in Ω.

Once we have a basis, we may then begin to discuss measures of circuit complexity.

Definition 10. Let $\mathcal{C}$ be a circuit. The *depth* of $\mathcal{C}$ is the number of gates on the longest path through the circuit. The *circuit size* of $\mathcal{C}$ is the number of gates it contains. Let $f : \{0,1\}^n \to \{0,1\}^m$ be a Boolean funciton, and let Ω be a basis. The *circuit depth* of f with respect to the basis Ω, dentoed $D_\Omega(f)$, is the minimum depth taken over all circuits that compute f. Similarly, the *circuit size* of f with respect to the basis Ω, denoted $C_\Omega(f)$, is the minimum size taken over all circuits that compute f.

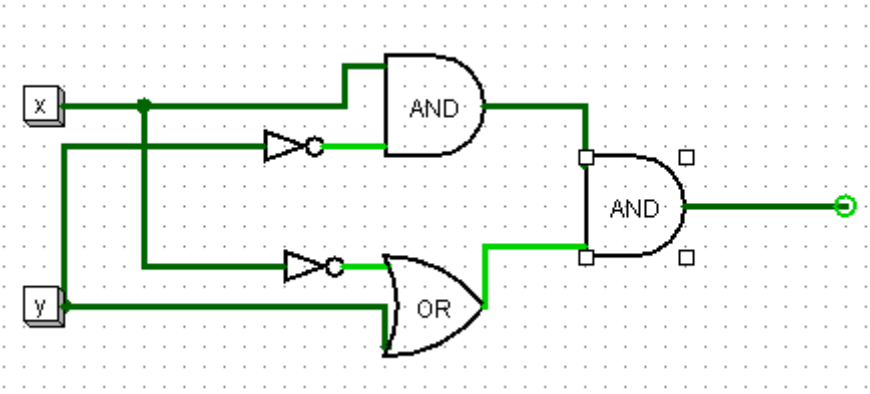
Remark 11. We note that in general, a circuit realizing $C_\Omega(f)$ may not also realize $D_\Omega(f)$.

Example 12. Again consider the circuit $\mathcal{C}$ from Example 3. Here, the depth of $\mathcal{C}$ is 3, realized by a sequence of a NOT gate, followed by an AND gate, and then an OR gate. The size of $\mathcal{C}$ is 5, as there are two NOT gates, two AND gates, and a single OR gate.



1.2.1 Exercises

(Recommended) Problem 1. Consider the following circuit.



Do the following.

(a) Give a functional description of this circuit. (See Example 2 in the notes.)

(b) Give an equivalent straight-line program corresponding to this circuit.

(Recommended) Problem 2. Recall that the standard basis is $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. Let $\mathcal{F}$ be the set of Boolean functions with codomain $\{0, 1\}$ that are realizable over Ω_0.[1]

(a) Show that $f \in \mathcal{F}$ if and only if f is realizable over the basis $\Omega = \{\mathsf{NAND}\}$.

(b) Let $\mathcal{C}$ be a circuit realizing $C_{\Omega_0}(f)$, and let $\mathcal{C}'$ be the circuit realizing f over the basis $\Omega = \{\mathsf{NAND}\}$ corresponding to your transformation in the previous part. Relate the circuit size of $\mathcal{C}'$ to $C_{\Omega_0}(f)$.

(c) Show that $f \in \mathcal{F}$ if and only if f is realizable over the basis $\Omega = \{\mathsf{XOR}, \mathsf{NOT}\}$.

(d) Let $\mathcal{C}$ be a circuit realizing $C_{\Omega_0}(f)$, and let $\mathcal{C}'$ be the circuit realizing f over the basis $\Omega = \{\mathsf{XOR}, \mathsf{NOT}\}$ corresponding to your transformation in the previous part. Relate the circuit size of $\mathcal{C}'$ to $C_{\Omega_0}(f)$.

(Recommended) Problem 3. A *Half-Adder* is a circuit that adds to single-digit binary numbers x and y. The Half-Adder outputs the sum $x+y \pmod 2$, as well as the carry bit. Note that the carry bit is a 1 precisely if a carry is generated by the addition .

(a) Implement the Half-Adder using only the AND, OR, and NOT gates.

[1] This is, in fact, all such Boolean functions. However, we have not proven this yet.

(b) Implement the Half-Adder, this time using the XOR gate. You may still use the AND, OR, and NOT gates. Comment on the difference in the number of gates required between your implementations in part (a) vs. part (b).

(Recommended) Problem 4. In this problem, we show the importance of having the NOT function in the standard basis. The *monotone basis* $\Omega_{\text{mon}} = \{\text{AND}, \text{OR}\}$, and we refer to a Boolean function using only the AND and OR operations as a *monotone Boolean function*. Do the following.

(a) Prove that if $f : \{0,1\}^n \to \{0,1\}$ is a monotone Boolean function, then we may write:

$$f(x_1, x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n) \vee (x_1 \wedge f(1, x_2, \ldots, x_n)).$$

(b) Denote $\preceq$ to be the ordering on $\{0,1\}^n$ where $x \preceq y$ if $x_i \leq y_i$ for all $i \in [n]$. Prove by induction on n that if $x, y \in \{0,1\}^n$ satisfy $x \preceq y$; then for any monotone Boolean function $f : \{0,1\}^n \to \{0,1\}$, we have that $f(x) \leq f(y)$.

(c) Deduce that NOT is not a monotone Boolean function. Conclude that Ω_{mon} is not a complete basis.

1.3 Boolean Normal Forms

Our goal is to show that every Boolean function can be realized using a Boolean circuit. One strategy to answer this question is as follows. Does there exist a basis Ω, such that every Boolean function can be placed into some standard/normal form with respect to Ω? A positive answer to this question would immediately allow us to construct a circuit for a given Boolean function. We introduce several normal forms, including the Disjunctive and Conjunctive Normal Forms, the Product of Sums Expansion, the Sum of Products Expansion, and the Ring-Sum Expansion.

1.3.1 POSE and SOPE Normal Forms

In this section, we introduce the Product of Sums Expansion (POSE) and Sum of Products Expansion (SOPE). We will also discuss the Conjunctive Normal Form (CNF), which is a special case of a POSE; as well as the Disjunctive Normal Form (DNF), which is a special type of SOPE. We begin with the definitions of POSE.

Definition 13 (Clause)**.** A *clause* is a Boolean function $\varphi : \{0,1\}^n \to \{0,1\}$ where φ consists of input variables or their negations, all added together (where addition is the OR operation).

Definition 14. Let $\varphi : \{0,1\}^n \to \{0,1\}$ be a Boolean function. A *Product of Sums Expansion* (or *POSE*) of φ is the conjunction of the clauses $C_1, \ldots, C_k$, such that :

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k.$$

It is helpful to think of a POSE as an AND or ORs.

Example 15. The following functions are all in POSE form.

- $(A \vee \neg B \vee C) \wedge (\neg D \vee E \vee F)$
- x
- $x \vee y$
- $(x \vee y) \wedge z$

In contrast, the following functions are **not** in POSE form.

- $\neg(x \vee y)$ is not in CNF, as the OR is nested within the NOT. Note that $\neg(x \vee y)$ could be writtin in POSE form, as follows: $\neg x \vee \neg y$.
- $(x \wedge y) \vee z$

We next introduce Sum of Products Expansion.

Definition 16. A *term* is a Boolean function $\varphi : \{0,1\}^n \to \{0,1\}$ where φ consists of input variables or their negations, all multiplied together (where multiplication is the AND operation).

Definition 17 (Sum of Products Expansion)**.** Let $\varphi : \{0,1\}^n \to \{0,1\}$ be a Boolean function. A *Sum of Products Expansion* (or *SOPE*) of φ is the disjunction of the terms $D_1, \ldots, D_k$ such that:

$$\varphi = D_1 \vee D_2 \ldots \vee D_k.$$

It is helpful to think of a SOPE as an OR or ANDs.

We now consider some examples of Boolean expressions in SOPE form.

Example 18. The following formulas are in SOPE form.

- $(x \wedge y \wedge \neg z) \vee (\neg a \wedge b \wedge c)$
- $(x \wedge y) \vee z$

- $x \wedge y$
- x

The following formulas are not in SOPE form.

- $\neg(x \vee y)$ is not in SOPE, as the OR is nested inside the NOT. Note that $\neg x \wedge \neg y$ is also in SOPE form.
- $x \vee (y \wedge (c \vee d))$ is not in SOPE, as the OR is nested inside the AND.

We next discuss how to construct the CNF and DNF representations of a Boolean function. We begin by examining the truth table of the Boolean function. The rows that evaluate to 1 provide the basis for the DNF, keeping the values for each variable in their respective columns. In order to compute the CNF, we examine the rows that evaluate to 0 and take the negations of each value.

Example 19. Consider the function $\varphi : \{0,1\}^3 \to \{0,1\}$, given by the following truth table.

x	y	z	φ
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

We now compute the CNF and DNF.

- **CNF:** To compute the CNF, we examine the rows that evaluate to 0. We include rows $1, 4, 6$, and 7. For each row, we invert the literals. So if $x = 0$ in the given row, we record x. If instead $x = 1$, we record $\overline{x}$. These rows contribute the following clauses.
  - **Row 1:** $(x \vee y \vee z)$
  - **Row 4:** $(x \vee \overline{y} \vee \overline{z})$
  - **Row 6:** $(\overline{x} \vee y \vee \overline{z})$
  - **Row 7:** $(\overline{x} \vee \overline{y} \vee z)$

  So the CNF formulation of φ is:

$$\varphi = (x \vee y \vee z) \wedge (x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z).$$

- **DNF:** To compute the DNF, we examine the rows that evaluate to 1. We include rows $2, 3, 5$, and 8. These rows contribute the following terms.
  - **Row 2:** $(\overline{x} \wedge \overline{y} \wedge z)$
  - **Row 3:** $(\overline{x} \wedge y \wedge \overline{z})$
  - **Row 5:** $(x \wedge \overline{y} \wedge \overline{z})$
  - **Row 8:** $(x \wedge y \wedge z)$

  So the DNF formulation of φ is:

$$\varphi = (\overline{x} \wedge \overline{y} \wedge z) \vee (\overline{x} \wedge y \wedge \overline{z}) \vee (x \wedge \overline{y} \wedge \overline{z}) \vee (x \wedge y \wedge z).$$

Remark 20. We note that for the constant function $\varphi = 1$, $\mathrm{CNF}(\varphi) = 1$ (i.e., the empty product). Similarly, for the constant function $\varphi = 0$, $\mathrm{DNF}(\varphi) = 0$ (i.e., the empty sum).

Remark 21. As any Boolean function can be expressed in either CNF or DNF, it follows that any Boolean function can be computed using a Boolean circuit over the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. We record this observation with the following theorem.

Theorem 22. Every Boolean function can be realized using a Boolean circuit over the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$.

Remark 23. We note that in practice, the CNF and DNF representations of a Boolean function may not be the most concise POSE and SOPE representations.

1.3.2 Ring-Sum Expansion

In this section, we introduce the Ring-Sum Expansion of a Boolean function f, which provides a way to express f over the basis $\{\mathsf{XOR}, \mathsf{AND}\}$. We begin with the definition of the Ring-Sum Expansion.

Definition 24. The *Ring-Sum Expansion* of a Boolean function f is the XOR ($\oplus$) of a constant and products of unnegated variables of f.

Remark 25. The set $\{0, 1\}$ together with the operations of addition ($\oplus$) and multiplication ($\wedge$) constitute the field $\mathbb{F}_2$. As a field is a ring, this motivates the terminology *Ring-Sum Expansion.*

The Ring-Sum Expansion can be constructed starting from the DNF representation of a Boolean function. We note that for $x \in \{0, 1\}$, $\overline{x} = 1 \oplus x$. We first replace any OR ($\vee$) operator with XOR ($\oplus$). We next replace each negated variable $\overline{x}$ in the DNF with $1 \oplus x$, and then apply the distribution law. If a term appears twice, we may cancel it out using commutativity of $\oplus$ and the fact that $x \oplus x = 0$.

Example 26. Consider the Boolean function $\varphi(x_1, x_2, x_3) = (\overline{x_1} \vee x_2) \wedge x_3$. We note that the DNF representation of φ is:

$$\varphi = \overline{x_1}x_2x_3 \ \vee \ \overline{x_1x_2}x_3 \ \vee \ x_1x_2x_3.$$

For succintcness, we note that a product corresponds to the $\wedge$ operator. So for instance, $\overline{x_1}x_2x_3 := \overline{x_1} \wedge x_2 \wedge x_3$. We now construct the Ring-Sum Expansion from the DNF representation of φ:

$$\begin{aligned}
\varphi &= [(1 \oplus x_1)x_2x_3] \oplus [(1 \oplus x_1)(1 \oplus x_2)x_3] \oplus x_1x_2x_3 \\
&= [x_2x_3 \oplus x_1x_2x_3] \oplus [x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_2x_3] \oplus x_1x_2x_3 \\
&= x_3 \oplus x_1x_3 \oplus x_1x_2x_3.
\end{aligned}$$

1.3.3 Exercises

(Recommended) Problem 5. Suppose φ is given by the following truth table.

x	y	z	φ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(a) Write φ in CNF.

(b) Write φ in DNF.

(Recommended) Problem 6. Consider the following Boolean functions in DNF. For each function ϕ, determine if there is a sequence of inputs such that ϕ evaluates to 1 on those inputs.

(a) $x \wedge \neg x$

(b) $(x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4) \vee (\neg x_1 \wedge \neg x_2 \wedge x_1 \wedge x_3)$

(c) $(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$

(Recommended) Problem 7. Suppose $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean function in DNF. Describe a polynomial-time algorithm to check whether φ has a satisfying instance. That is, establish that $\mathsf{DNF\text{-}SAT} \in \mathsf{P}$.

(Recommended) Problem 8. Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. Denote $\mathrm{CNF}(f)$ and $\mathrm{DNF}(f)$ to be the CNF and DNF representations of f, respectively.

(a) Show that $\mathsf{CNF}(f)$ is unique. That is, suppose that φ, φ' are CNF realizations of f. Show that a given clause C belongs to φ if and only if C belongs to φ.

(b) Show that $\mathrm{CNF}(f) = \overline{\mathrm{DNF}(f)}$.

(Recommended) Problem 9. Let $f_{\oplus}^{(n)} : \{0,1\}^n \to \{0,1\}^n$ be the parity function. That is,

$$f_{\oplus}^{n}(x_1, \ldots, x_n) = \begin{cases} 0 & : \sum_{i=1}^{n} x_i \equiv 0 \pmod 2, \\ 1 & : \text{otherwise.} \end{cases}$$

Our goal is to show that the SOPE representing f has exponentially many terms.

(a) Prove by induction on n that $f_{\oplus}^{(n)}$ has 2^{n-1} terms in the DNF representation.

(b) Our goal now is to show that the DNF representation of $f_{\oplus}^{(n)}$ cannot be simplified. Show that each variable uniquely specifies a term in the DNF representation of $f_{\oplus}^{(n)}$.

(c) Argue by contradiction that no term in the SOPE of $f_{\oplus}^{(n)}$ has fewer than n variables.

(d) Conclude that the DNF representation of $f_{\oplus}^{(n)}$ cannot be simplified. Deduce that the SOPE has exponentially many terms.

Remark 27. A similar approach in Problem 9 may be used to show that the POSE expansion of $f_{\oplus}^{(n)}$ has exponentially many terms.

(Recommended) Problem 10. Let:

$$f_{\vee}^{(n)}(x_1, \ldots, x_n) := \bigvee_{i=1}^{n} x_i.$$

Do the following.

(a) Find an equivalent expression for $x \vee y$ using the basis $\{\mathsf{AND}, \mathsf{XOR}\}$.

(b) Prove by induction that the Ring-Sum Expansion of $f_{\vee}^{(n)}$ contains every product term, except for the constant 1.

1.4 Parallel Prefix Circuits

The circuit model allows for more natural analysis and discussion of parallel computation, compared to sequential models such as the RAM or Turing Machine models. To this end, we ask about functions which can be effectively parallelized. Certain algebraic operations that satisfy the associative law constitute one such class of functions. The underlying set, coupled with the operation, is referred to as a *semi-group*. Precisely, we have the following definition.

Definition 28. A *semi-group* $(S, \odot)$ consists of a set S together with an associative, binary operator $\odot : S \times S \to S$.

Example 29. We recall several examples of semigroups.

(a) The set $S = \{0, 1\}$ together with the AND operator constitutes a semi-group.

(b) The set $S = \{0, 1\}$ together with the OR operator constitutes a semi-group.

(c) The natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ forms a semi-group under addition.

(d) Let $\{0, 1\}^*$ denote the set of all *finite* binary strings. The set $\{0, 1\}^*$ forms a semi-group under the operation of string concatenation.

Using the associativity of a semi-group $(S, \odot)$, we seek to compute a parallel prefix function, which returns the sequence running products for $x_1 \odot x_2 \odot \cdots \odot x_n$.

Definition 30. Let $(S, \odot)$ be a semi-group. A *prefix function* $\mathcal{P}_{\odot}^{(n)} : S^n \to S^n$ maps the input $(x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_n)$; where for $1 \leq i \leq n$, $y_i = x_1 \odot x_2 \odot \cdots \odot x_i$.

Suppose we have a semi-group $(S, \odot)$, as well as a logic gate for $\odot$ with fan-in 2. Then in order to compute:

$$x_1 \odot x_2 \odot \cdots \odot x_n,$$

we may use a binary tree where the leaves are the inputs and the root node is the output. Such a circuit has depth $\lceil \log_2(n) \rceil$. We require $\Theta(n^2)$ gates to return the sequence of running products. We leave the precise details in designing and analyzing such a circuit as an exercise for the reader.

We now ask whether we can use asymptotically fewer gates. Using a dynamic programming technique, we can construct an equivalent circuit with depth $2\lceil \log_2(n) \rceil$ that uses $\Theta(n)$ gates. While the depth of the circuit doubles, we note that the depth our new circuit is still $\Theta(\log(n))$. We now turn to constructing such a circuit. We begin with the definition of a parallel prefix circuit.
Our goal is to use dynamic programming to compute y_j. The circuit construction is recursive in nature. We introduce key observations and the high-level strategy before proceeding into designing the circuit. Define $x[r, r] = x_r$; and for $r \leq s$, let $x[r, s] := x_r \odot x_{r+1} \odot \cdots \odot x_s$. So we have that $y_j := x[1, j]$. Now as $\odot$ is associative, we have that $x[r, s] = x[r, t] \odot x[t + 1, s]$, for $r \leq t < s$. To make this clear, we have by definition that:

$$\begin{aligned} x[r, s] &= x_r \odot x_{r+1} \odot \cdots x_s, \\ x[r, t] &= x_r \odot x_{r+1} \odot \cdots x_t, \\ x[t + 1, s] &= x_{t+1} \odot x_{t+2} \odot \cdots x_s. \end{aligned}$$

So:

$$\begin{aligned} x[r, s] &= \Big(x_r \odot x_{r+1} \odot \cdots x_t \Big) \odot \Big(x_{t+1} \odot x_{t+2} \odot \cdots x_s \Big) \\ &= x[r, t] \odot x[t + 1, s]. \end{aligned}$$

We now turn to designing the circuit itself. Note that we assume the number of inputs n to be a power of 2; that is, $n = 2^k$. If the number of inputs is not a power of 2, we may add additional inputs until we have 2^k inputs for some k. When examining the outputs of the prefix function, we need only consider y_n; we may

ignore y_i for $i > n$.

The construction of our circuit to compute $\mathcal{P}_{\odot}^{(n)}$ is recursive in nature. Suppose we have inputs $x_1, \ldots, x_n$. We obtain the following $(n/2)$-tuple:

$$(x[1,2], x[3,4], \ldots, x[2^k - 1, 2^k]).$$

Now observe that $x[i, i+1] = x[i,i] \odot x[i+1, i+1]$. So suppose we compute $\mathcal{P}_{\odot}^{(n/2)}(x[1,2], \ldots, x[2^k - 1, 2^k])$. Recall that the output of $\mathcal{P}_{\odot}^{(n/2)}(x[1,2], \ldots, x[2^k - 1, 2^k]) = (z_1, \ldots, z_{n/2})$, where:

$$\begin{aligned} z_i &= x[1,2] \odot x[3,4] \odot \cdots \odot x[2i-1, 2i] \\ &= x[1, 2i]. \end{aligned}$$

That is,

$$\mathcal{P}_{\odot}^{(n/2)}(x[1,2], \ldots, x[2^k - 1, 2^k]) = (x[1,2], x[1,4], \ldots, x[1, 2^k]).$$

Note that $\mathcal{P}_{\odot}^{(n)}$ must also compute $x[1,1], x[1,3], x[1,5], \ldots, x[1, 2^k - 1]$. Thankfully, these are easy to compute once we have computed $x[1,2], x[1,4], \ldots, x[1,2^k]$. Namely, observe that $x[1, 2i+1] = x[1,2i] \odot x[2i+1]$. Note that as $x[1,1] = x_1$, we do not require an operation to compute $x[1,1]$. Thus, only $2^{k-1} - 1$ operations are required to compute $x[1,1], x[1,3], x[1,5], \ldots, x[1, 2^k - 1]$.

Before analyzing the circuit's complexity, we consider an example.

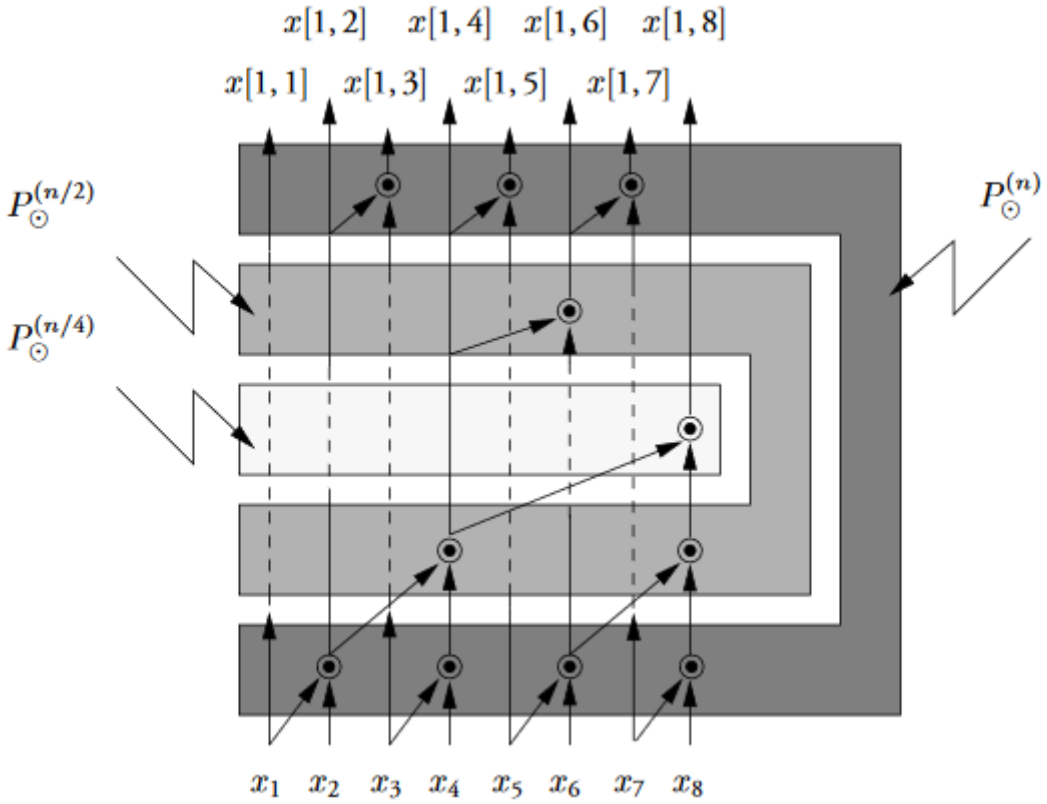
Example 31. Suppose we have $n = 2^3$ inputs, $x_1, \ldots, x_8$. Our goal is to compute $\mathcal{P}_{\odot}^{(8)}(x_1, \ldots, x_8)$. The circuit proceeds as follows.

- We first compute $x[1,2], x[3,4], x[5,6], x[7,8]$. In order to more easily follow the work in the recursive calls, we label $a_1 = x[1,2], a_2 = x[3,4], a_3 = x[5,6], a_4 = a[7,8]$.
- The $\mathcal{P}_{\odot}^{(8)}(x_1, \ldots, x_8)$ circuit now recursively computes $\mathcal{P}_{\odot}^{(4)}(a_1, a_2, a_3, a_4)$. Our recursive call starts by computing:

  $$\begin{aligned} &a_1 \odot a_2 \text{ (which we recall is } x[1,2] \odot x[3,4] = x[1,4]) \\ &a_3 \odot a_4 \text{ (which we recall is } x[5,6] \odot x[7,8] = x[5,8]) \end{aligned}$$

  In order to more easily follow the recursive calls, we label $b_1 = x[1,4]$ and $b_2 = x[5,8]$.
- The $\mathcal{P}_{\odot}^{(4)}(a_1, a_2, a_3, a_4)$ circuit now recursively computes $\mathcal{P}_{\odot}^{(2)}(b_1, b_2) = (b_1, b_1 \odot b_2) = (a[1,2], a[1,4]) = (x[1,4], x[1,8])$.
- We now return control to the $\mathcal{P}_{\odot}^{(4)}(a_1, a_2, a_3, a_4)$ circuit. We have computed $a[1,2]$ and $a[1,4]$. We note that $a[1,1] = 1$, and so we only require one additional operation to compute $a[1,3] = a[1,2] \odot a_3$. So the $\mathcal{P}_{\odot}^{(4)}(a_1, a_2, a_3, a_4)$ circuit has completed its execution and returns control to $\mathcal{P}_{\odot}^{(8)}(x_1, \ldots, x_8)$. Note that we have computed $a[1,1] = x[1,2]$, $a[1,2] = x[1,4]$, $a[1,3] = x[1,6]$, and $a[1,4] = x[1,8]$.
- Now that control has been returned to $\mathcal{P}_{\odot}^{(8)}(x_1, \ldots, x_8)$, it remains to compute $x[1,1], x[1,3], x[1,5]$, and $x[1,7]$. We again note that $x[1,1] = 1$, and so does not require any operations to compute. We need one operation to compute $x[1,3] = x[1,2] \odot x_3$, one operation to compute $x[1,5] = x[1,4] \odot x_5$, and one operation to compute $x[1,7] = x[1,6] \odot x_7$.
- $\mathcal{P}_{\odot}^{(8)}(x_1, \ldots, x_8)$ now outputs $(x[1,1], x[1,2], x[1,3], \ldots, x[1,8])$.

Pictorally, we include the diagram for this circuit from [Sav97][Chapter 2, Figure 2.13].



We now turn to analyzing the circuit construction.

- **Circuit Size:** We note that $\mathcal{P}_{\odot}^{(n)}$ uses an initial 2^{k-1} $\odot$ gates to compute $x[1,2], x[3,4], \ldots, x[2^k-1, 2^k]$. At the last layer, the circuit uses an additional $2^{k-1}-1$ gates to compute $x[1,3], \ldots, x[1, 2^k-1]$. No gate is needed to compute $x[1,1] = x_1$. This yields $2^k - 1$ total gates, in addition to the number of gates that the circuit for $\mathcal{P}_{\odot}^{(n/2)}$ uses. Therefore, the size of the circuit is given by the recurrence:

$$C(k) = \begin{cases} C(k-1) + 2^k - 1 & : k > 0, \\ 0 & : k = 0. \end{cases}$$

  We stress that the input variable k for the recurrence $C(k)$ is the same k as in the exponent of $n = 2^k$. Solving this recurrence using the unrolling method yields the closed form expression: $C(k) = 2^{k+1} - k - 2$. Noting that $n = 2^k$ (and so, $k = \log_2(n)$), we have that:

$$\begin{aligned} 2^{k+1} - k - 2 &= 2 \cdot 2^k - k - 2 \\ &= 2n - \log_2(n) - 2. \end{aligned}$$

  It follows that if Ω is a basis containing $\odot$, then $C_{\Omega}(\mathcal{P}_{\odot}^{(n)}) \leq 2n - \log_2(n) - 2$. We make no claims that the prefix circuit construction above is optimal. As a result, we only obtain an upper bound on $C_{\Omega}(\mathcal{P}_{\odot}^{(n)})$.

- **Circuit Depth:** We note that $\mathcal{P}_{\odot}^{(n)}$ has three layers:
  - The initial layer that computes $x[1,2], x[3,4], \ldots, x[2^k-1, 2^k]$,
  - The layer computing $\mathcal{P}_{\odot}^{(n/2)}(x[1,2], x[3,4], \ldots, x[2^k-1, 2^k])$, and
  - The layer computing $x[1,1], x[1,3], \ldots, x[1, 2^k-1]$.

  There are two layers, each of depth 1, not associated with the recursive call to $\mathcal{P}_{\odot}^{(n/2)}$. So the circuit depth satisfies the following recurrence:

$$D(k) = \begin{cases} D(k-1) + 2 & : k > 0, \\ 0 & : k = 0. \end{cases}$$

Again, we stress that the input variable k for the recurrence $C(k)$ is the same k as in the exponent of $n = 2^k$. Solving this recurrence using the unrolling method yields the closed form expression: $D(k) = 2k$. Noting that $k = \log_2(n)$, we have that $2k = 2\log_2(n)$. So the depth of our circuit is $2\log_2(n)$. It follows that if Ω is a basis containing $\odot$, then $D_\Omega(\mathcal{P}_\odot^{(n)}) \leq 2\log_2(n)$. We make no claims that the prefix circuit construction above is optimal. As a result, we only obtain an upper bound on $D_\Omega(\mathcal{P}_\odot^{(n)})$.

1.4.1 Exercises

(Recommended) Problem 11. Suppose we have the basis Ω with the associative, binary operator $\odot$. Suppose that the $\odot$ gates have fan-in 2 (that is, the $\odot$ gates accept two inputs). Do the following.

(a) Design a circuit of depth $\lceil \log_2(n) \rceil$ to compute $x_1 \odot x_2 \cdots \odot x_n$.

(b) Comment on why the associativity of $\odot$ is a necessary assumption for your circuit's correctness in part (a).

(c) Suppose that $n = 2^k$ for some $k \in \mathbb{N}$. Determine the size of your circuit.

(d) Suppose now that n is not a power of 2. Determine a suitable function $f(n)$ such that the size of your circuit is $\Theta(f(n))$. [**Hint:** We note that there exists $k \in \mathbb{N}$ such that $2^k < n < 2^{k+1}$.]

(e) Now suppose instead that the $\odot$ gate has unbounded fan-in. Design a circuit to compute $x_1 \odot x_2 \cdots \odot x_n$. What are the size and depth of your new circuit, and how do they compare to the size and depth of the circuit you designed in part (a)?

Definition 32. Let $k \in \mathbb{N}$. We say that a language $L \in \mathsf{NC}^k$ if there exists a uniform family of circuits $(C_n)_{n\in\mathbb{N}}$ over the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ such that the following conditions hold:

(a) A string $\omega \in \{0,1\}^*$ of length n is in L if and only if $C_n(\omega) = 1$ (that is, C_n on input ω evaluates to 1).

(b) Each circuit has fan-in 2 for the AND and OR gates, and fan-in 1 for the NOT gates.

(c) The circuit C_n has depth $O(\log^k(n))$ and size $O(n^m)$. The implicit constants and the exponent m both depend on L.

Definition 33. Let $k \in \mathbb{N}$. We say that a language $L \in \mathsf{AC}^k$ if there exists a uniform family of circuits $(C_n)_{n\in\mathbb{N}}$ over the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ such that the following conditions hold:

(a) A string $\omega \in \{0,1\}^*$ of length n is in L if and only if $C_n(\omega) = 1$ (that is, C_n on input ω evaluates to 1).

(b) Each circuit has fan-in 1 for the NOT gates.

(c) The circuit C_n has depth $O(\log^k(n))$ and size $O(n^m)$. The implicit constants and the exponent m both depend on L.

Remark 34. Note that the only difference between an NC circuit and an AC circuit is that the NC circuits have bounded fan-in for the AND and OR gates, while the AC circuits allow for unbounded fan-in for the AND and OR gates.

Remark 35. We also note that the uniformity condition means that there is a Turing Machine M such that on input 1^n, M can generate the circuit C_n. This condition is not important at present; however, it will become important later when we relate Turing Machines to circuits.

(Recommended) Problem 12. Fix $k \in \mathbb{N}$. Do the following.

(a) Show that $\mathsf{NC}^k \subseteq \mathsf{AC}^k$.

(b) Show that $\mathsf{AC}^k \subseteq \mathsf{NC}^{k+1}$.

Remark 36. Define:

$$\mathsf{NC} := \bigcup_{k \in \mathbb{N}} \mathsf{NC}^k.$$

In light of Problem 12, we have that:

$$\mathsf{NC} = \bigcup_{k \in \mathbb{N}} \mathsf{AC}^k.$$

Remark 37. It is known that $\mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{NC}^1$. For $k \geq 1$, none of the containments $\mathsf{NC}^k \subseteq \mathsf{AC}^k \subseteq \mathsf{NC}^{k+1}$ are known to be strict.

1.5 Parallel Prefix Addition

In this section, we examine a parallel prefix circuit for binary addition. Let $u \in \mathbb{N}$. We write the binary expansion of u as:

$$u = \sum_{i=0}^{n-1} u_i 2^i,$$

where each $u_i \in \{0, 1\}$ and $n := \lceil \log_2(u) \rceil$. Now let $v \in \mathbb{N}$, and consider the binary expansion of v:

$$v = \sum_{i=0}^{m-1} v_i 2^i.$$

For the rest of this section, we assume without loss of generality that $m = n$. Otherwise, we consider $\max\{m, n\}$. Our goal now is to obtain the binary representation of $u + v$, we we denote as:

$$u + v = \sum_{i=0}^{n} s_i 2^i.$$

Our primary goal is to determine the values of the s_i terms. We note that when we evaluate $u_i + v_i$, a carry bit is generated for our evaluationg of $u_{i+1} + v_{i+1}$. If $u_i = 1$ and $v_i = 1$, then a carry bit of 1 is generated. We denote this carry bit as c_{i+1}, as it will be factored into the calculation of s_{j+1}. If $i \geq 1$, then we also need to consider the carry bit generated at the $i - 1$ stage, which is c_i. If $c_i = 1$ and at least one of $u_i = 1$ or $v_i = 1$, then a carry bit of 1 is generated at stage i. That is, $c_{i+1} = 1$. Otherwise, $c_{i+1} = 0$. Effectively, our circuit will have to track both the carry bits and the s_i bits.

In order to compute these values, we introduce intermediary variables called *generate* and *propogate* bits.

- The ith generate bit is given by $g_i := u_i \wedge v_i$. That is, $g_i = 1$ if and only if adding u_i and v_i generates a 1 for the carry bit c_{i+1}.
- The ith propogate bit is given by $p_i := u_i \oplus v_i$. Observe that p_i and g_i cannot both be 1. So if $p_i = 1$, we have that either $u_i = 1$ or $v_i = 1$, but u_i and v_i are not both 1. It follows that the carry bit c_i from the previous stage determines whether $c_{i+1} = 1$.

With the propogate and generate bits defined, we observe the following.

- First, $s_i = p_i \oplus c_i$. We note that $p_i = u_i \oplus v_i$, so $s_i = u_i \oplus v_i \oplus c_i$. That is, we add u_i and v_i, and then add in the incoming carry bit.
- Second, we observe that the carry bit generated satisfies the following:

  $$c_{i+1} = (p_i \wedge c_i) \vee g_i.$$

  If $g_i = 1$, then both $u_i = 1$ and $v_i = 1$. So adding u_i and v_i generates a carry bit. Otherwise, $c_{i+1} = 1$ precisely if $u_i \oplus v_i = 1$ and $c_i = 1$.

We now turn to defining our parallel-prefix addition circuit. Rather than tracking the carry and s_j bits at each intermediary step, we instead track the propogate and generate bits at each stage. Precisely, we examine ordered pairs (p_i, g_i), where again p_i is the ith propogate bit and g_i is the ith generate bit. Our goal is to design a circuit that outputs whether a carry bit was generated at each stage. Note that the propogate and generate bits p_i and g_i depend only on the u_i and v_i, the ith positions of the binary numbers we are adding. So we may easily compute p_i and g_i in parallel.

Suppose we are considering consecutive indices (p_i, g_i) and (p_{i+1}, g_{i+1}). Note that $p_i = g_i$ precisely if $p_i = 0$ and $g_i = 0$. Similarly, $p_{i+1} = g_{i+1}$ precisely if $p_i = 0$ and $g_i = 0$. We may track whether a propogate was generated at stages i and $i + 1$ by considering $p_i \wedge p_{i+1}$. Now to check whether the carry bit c_{i+2} takes on the value 0 or 1, when restricting attention to positions i and $i + 1$, we check that $g_{i+1} = 1$ (which means that $u_i \wedge v_i = 1$) or $p_{i+1} \wedge g_i = 1$. Note that if $g_i = 1$, then $c_{i+1} = 1$. So $p_{i+1} \wedge g_i = 1$ implies that $p_{i+1} \wedge c_{i+1} = 1$.

So the $(i+1)$st carry bit propogates.

In order to define a parallel-prefix circuit, we first need a suitable associative operator. Define $\diamond : \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}^2$ by:

$$(a,b) \diamond (c,d) = (a \wedge c, (b \wedge c) \vee d).$$

Again, we write $a \wedge c$ as ac, to improve readability. We check that $\diamond$ is associative. Observe that:

$$\begin{aligned}((a,b) \diamond (c,d)) \diamond (e,f) &= (a,b) \diamond ((c,d) \diamond (e,f)) \\ &= (ace, bce \vee de \vee f).\end{aligned}$$

We now define our lookup table. Denote $\pi[j,j] = (p_j, g_j)$. For $j < k$, let $\pi[j,k] = \pi[j,k-1] \diamond \pi[k,k]$. By induction, we have the following:

- The first component of $\pi[j,k]$ is 1 if and only if a carry propogates through each stage $j, j+1, \ldots, k$.
- The second component of $\pi[j,k]$ is 1 if and only if a carry is generated at some stage r, where $j \leq r \leq k$; and that carry propogates from stage r through stage k.

We apply the parallel-prefix circuit from Section 1.4 with the associative operator $\diamond$. The circuit takes as input $(\pi[0,0], \pi[1,1], \ldots, \pi[n-1,n-1])$ and returns the sequence $(\pi[0,0], \pi[0,1], \ldots, \pi[0,n-1])$. We note that there is no carry bit of 1 when adding the ones place bits u_0 and v_0. So $c_0 = 0$. As the first component of $\pi[0,i]$ tracks whether a carry bit value of 1 propogated through each stage $0, \ldots, i$, we have that the first component of $\pi[0,i]$ is 0.

The second component of $\pi[0,i]$ tracks whether a a carry bit is generated on or before stage i. By unrolling the expression for c_{j+1}:

$$\begin{aligned}c_{j+1} &= p_j c_j \vee g_j \\ &= p_j(p_{j-1}c_{j-1} \vee g_{j-1}) = p_j p_{j-1} c_{j-1} \vee g_{j-1},\end{aligned}$$

we see that the expression in the second component of $\pi[0,i]$ is precisely c_{j+1}. This unrolling also highlights the necessity of recording whether a carry propogates through each stage in the first component. Now the bit s_i may be recovered from taking the Exclusive Or of p_i and the second component of $\pi[0,i-1]$.

1.5.1 Exercises

(Recommended) Problem 13. Recall the associative operator $\diamond : \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}$, given by:

$$(a,b) \diamond (c,d) = (ac, bc \vee d).$$

Denote $\pi[j,j] = (p_j, g_j)$. For $j < k$, let $\pi[j,k] = \pi[j,k-1] \diamond \pi[k,k]$.

(a) Show by induction on $\ell := k - j$ that the first component of $\pi[j,k]$ is 1 precisely if a carry bit of 1 propogates through the full adder stages indexed $j, \ldots, k$.

(b) Show by induction on $\ell := k - j$ that the second component of $\pi[j,k]$ is 1 precisely if a carry bit of 1 is generated at some stage r, where $j \leq r \leq k$, and that carry propogates from stage r through stage k.

1.6 Circuit Lower Bounds

In this section, we show that most Boolean function $f : \{0,1\}^n \to \{0,1\}$ require size exponential in n. The key idea is that there are more Boolean functions than there are small circuits. We note that there are 2^{2^n} such Boolean functions. We modify the proof from Kopparty, Kim, and Lund [KKL13], which is considerably simpler than that found in Savage [Sav97][Theorem 2.12.1].

We begin by analyzing the circuit size.

Theorem 38. Let $\epsilon \in (0,1)$. The fraction of Boolean functions $f : \{0,1\}^n \to \{0,1\}$ that have size complexity satisfying:

$$C_{\Omega_0}(f) > \frac{2^n}{2n}(1-\epsilon)$$

is at least $1 - 2^{-\epsilon \cdot 2^n}$ when $n \geq 6$.

Proof. Let $G \geq 0$, and denote $M(G)$ to be the number of Boolean functions that are computable using a circuit of at most G gates. Each gate accepts at most 2 inputs, and the order in which the inputs are selected does not matter. Now an input to a given gate may come from either another gate or a wire corresponding to a variable. There are $n+G-1$ such selections. So there are at most $\binom{n+G-1}{2} \leq \binom{n+G}{2}$ ways to select the inputs for a given gate. Now there are three gates in the standard basis Ω_0. So there are at most $3\binom{n+G}{2}$ ways to choose gates.

Now take:

$$G = \frac{2^n}{2n}(1-\epsilon).$$

We show the number of circuits of size at most G is asymptotically smaller than the total number of Boolean functions on n variables. We note that there are 2^{2^n} Boolean functions on n variables. Now we have that:

$$\begin{aligned}
M(G) &\leq 3^G \binom{n+G}{2}^G && (1)\\
&< 3^G (n+G)^{2G} && (2)\\
&< 3^G (2G)^{2G} && (3)\\
&< 3^{2G} (2G)^{2G} && (4)\\
&= 6^{2G} \cdot G^{2G} && (5)\\
&= 6^{2 \cdot 2^n (1-\epsilon)/(2n)} \cdot \left(\frac{2^n(1-\epsilon)}{n}\right)^{2 \cdot 2^n (1-\epsilon)/(2n)} && (6)\\
&= 6^{2^n(1-\epsilon)/n} \cdot \left(\frac{2^n(1-\epsilon)}{n}\right)^{2^n(1-\epsilon)/n} && (7)\\
&= \left(\frac{6(1-\epsilon)}{n}\right)^{2^n(1-\epsilon)/n} \cdot (2^n)^{2^n(1-\epsilon)/n} && (8)\\
&= \left(\frac{6(1-\epsilon)}{n}\right)^{2^n(1-\epsilon)/n} \cdot 2^{2^n(1-\epsilon)} && (9)\\
&\leq 2^{2^n(1-\epsilon)}. && (10)
\end{aligned}$$

We note that the upper bound at line (10) holds whenever $n \geq 6$. As $M(G) < 2^{2^n(1-\epsilon)}$, it follows that the fraction of Boolean functions requiring more than G gates is:

$$1 - 2^{-\epsilon \cdot 2^n},$$

as desired. □

1.7 Chapter Exercises

(Recommended) Problem 14. It is often desirable to take a circuit over the standard basis $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ and convert it to an equivalent circuit without the negations. This motivates *dual-rail logic* circuits, which allow us to effectively push the negations down to the inputs. In dual-rail logic, the variable $|x\rangle$ is represented by the pair $(x, \overline{x})$. In particular, observe that $|0\rangle = (0, 1)$ and $|1\rangle = (1, 0)$. We now turn to defining the dual-rail logical operations.

- The DRL-AND operation $\wedge$ is defined by $|x\rangle \wedge |y\rangle = |x \wedge y\rangle$. Note that if $|x\rangle = (x, \overline{x})$ and $|y\rangle = (y, \overline{y})$, then we may realize $|x \wedge y\rangle$ over a classical circuit, using the standard basis $\Omega_0 := \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ as follows:
$$|x\rangle \wedge |y\rangle = (x \wedge y, \overline{x \wedge y}) = (x \wedge y, \overline{x} \vee \overline{y}).$$
- The DRL-OR operation $\vee$ is defined by $|x\rangle \vee |y\rangle = |x \vee y\rangle$.
- The DRL-NOT operation can be realized by physically twisting the wires on $(x, \overline{x})$, and so we do not include DRL-NOT in our basis.

Do the following.

(a) Show how to realize DRL-OR over a classical circuit, using the standard basis $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$.

(b) Deduce that every Boolean function $\varphi : \{0,1\}^n \to \{0,1\}^m$ can be realized by a dual-rail logic circuit in which the standard NOT gates are only used on input variables (to obtain the pair $(x, \overline{x})$).

(c) Let $\mathsf{DRL} = \{\mathsf{DRL\text{-}AND}, \mathsf{DRL\text{-}OR}\}$ be the standard dual-rail logic basis. Let $\varphi : \{0,1\}^n \to \{0,1\}$. Show that: $C_{\mathsf{DRL}}(\varphi) \leq 2C_{\Omega_0}(\varphi)$.

(d) Show that $D_{\mathsf{DRL}}(\varphi) \leq D_{\Omega_0}(\varphi) + 1$. For circuits over Ω_0, do not count the NOT gates as contributing to the depth.[2]

(Recommended) Problem 15. For this problem, we restrict attention to the basis $\Omega = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}, \mathsf{XOR}\}$, where the AND, OR, and XOR gates all have fan-in 2. We are interested in the Membership problem, which takes as input $x \in \{0,1\}^n$ and $y \in \{0,1\}$; the goal is to decide whether there exists an $i \in [n]$ such that $x_i = y$.

(a) Design a circuit that computes the function $f^{(n)}_{\mathsf{equality}} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, where $f_{\mathsf{equality}}(x, y) = 1$ if and only if $x = y$. [**Hint:** Start with the case when $n = 1$.]

(b) The *membership* function $f^{(n)}_{\text{membership}} : \{0,1\}^n \times \{0,1\} \to \{0,1\}$ is given by $f^{(n)}_{\text{membership}}(x_1, \ldots, x_n; y) = 1$ if and only if there exists an $i \in [n]$ such that $x_i = y$. Using part (a), design a circuit to compute $f^{(n)}_{\text{membership}}$.

(c) Analyze the size and depth of your circuit in part (b). In particular, conclude that the Membership problem is in NC^1. [**Note:** NC^1 circuits are defined over the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$. Why does allowing for XOR gates not change the result?]

(d) Strengthen the above result to show that the Membership problem is in AC^0. That is, if we allow for unbounded fan-in, then we only require a constant depth circuit. [**Note:** The same note from part (c) about the XOR gate applies to AC circuits as well.]

Definition 39. A function $\varphi(x_1, \ldots, x_n)$ is said to be *symmetric* if for every permutation $\pi \in \mathrm{Sym}(n)$, we have that:
$$\varphi(x_1, \ldots, x_n) = \varphi(x_{\pi(1)}, \ldots, x_{\pi(n)}).$$

The building block for our symmetric functions are the *elementary symmetric functions* $e^{(n)}_t : \{0,1\}^n \to \{0,1\}$, where $0 \leq t \leq n$, given by:
$$e^{(n)}_t(x_1, \ldots, x_n) = \begin{cases} 1 & : \text{if } \sum_{i=1}^n x_i = t, \\ 0 & : \text{otherwise.} \end{cases}$$

[2] This is a standard assumption in Circuit Complexity.

The next problems serve as practice for using elementary symmetric functions to design circuits that compute more complicated symmetric functions.

(Recommended) Problem 16. Let $n \in \mathbb{Z}^+$, and let $1 \leq t \leq n$. The *threshold function* is given by:

$$\tau_t^{(n)}(x_1, \ldots, x_n) = \begin{cases} 1 & : \text{if } \sum_{i=1}^n x_i \geq t, \\ 0 & : \text{otherwise.} \end{cases}$$

Describe how to build an AC circuit using the standard basis and elementary symmetric functions.

(Recommended) Problem 17. Let $n \in \mathbb{Z}^+$. The *binary sort* function $f_{\text{sort}}^{(n)} : \{0,1\}^n \to \{0,1\}$ sorts an n-tuple into descending order. Here, for $x \in \{0,1\}^n$, we have:

$$f_{\text{sort}}^{(n)}(x) = (\tau_1^{(n)}(x), \tau_2^{(n)}(x), \ldots, \tau_n^{(n)}(x)).$$

Show that beyond the cost of implementing the elementary symmetric functions, we can realize $f_{\text{sort}}^{(n)}$ using an additional depth of $O(\log(n))$ and an additional $O(n)$ gates from the standard basis $\Omega_0 = \{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$.

(Recommended) Problem 18. Let $m, n \in \mathbb{Z}^+$, and let $0 \leq c \leq m-1$. Define the *modulus function* to be:

$$f_{c,\ \text{mod } m}^{(n)}(x_1, \ldots, x_n) = \begin{cases} 1 & : \text{if } \sum_{i=1}^n x_i \equiv c \pmod{m}, \\ 0 & : \text{otherwise.} \end{cases}$$

Describe how to build a circuit using the standard basis and elementary symmetric functions. Pay close attention to the fact that n is *fixed*.

(Recommended) Problem 19. Recall from Problem 12 that $\mathsf{NC}^0 \subseteq \mathsf{AC}^0$. In this problem, we show that the containment is strict.

(a) Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. Show that if f is computed by an NC^0 circuit of depth d, then the circuit depends on at most 2^d inputs.

(b) Show that the following function is not in NC^0.

$$f_\wedge^{(n)}(x_1, \ldots, x_n) = \bigwedge_{i=1}^n x_i.$$

(c) Deduce that $\mathsf{NC}^0 \subsetneq \mathsf{AC}^0$.

(Recommended) Problem 20. In this problem, we seek to better understand the space between NC^1 and NC^2. Let L be the set of languages decidable by deterministic Turing Machines[3] where the input is read-only and only $O(\log(n))$ additional space is available for read and write access. Let NL denote the set of languages decidable by non-deterministic Turing Machines where the input is read-only and only $O(\log(n))$ additional space is available for read and write access. Equivocally, NL is the set of languages L; where if $x \in L$, we can verify this in space $O(\log(n))$. Observe that:

$$\mathsf{L} \subseteq \mathsf{NL}.$$

Our goal will be to establish upper bounds for NL. To do this, we consider the Connectivity problem, which takes as input a directed graph $G(V, E)$ and two vertices $u, v \in V(G)$; we seek to decide whether there is a $u \to v$ path in G. The Connectivity problem is NL-complete under logspace reductions.

(a) We think of a directed graph $G(V, E)$ as a binary relation $E \subseteq V(G) \times V(G)$. The *transitive closure* of E (which we may also refer to as the transitive closure of G) is the smallest transitive relation that contains E. Let $u, v \in V(G)$. Show that there is a $u \to v$ path if and only if (u, v) is in the transitive closure of E.

[3]You may think of Turing Machines as algorithms. We are not concerned with the technicalities of Turing Machines.

(b) For the remainder of this problem, we will consider Boolean matrix multiplication, where addition is given by the OR operator and multiplication is given by the AND operator. Let A be the adjacency matrix of the graph G. Let $M := (A \vee I)$, where I is the identity matrix. Prove by induction that there is a directed path of length at most k from $u \to v$ if and only if $M^k_{uv} = 1$. It follows immediately that we can recover the transitive closure of G from M^n, where n is the number of vertices.

(c) Let A, B be $n \times n$ Boolean matrices. Show that computing AB is in NC^1. [**Hint:** When multiplying non-Boolean matrices X and Y, we have that $XY_{ij} = \langle R_i(X), C_j(Y)^T \rangle$, where $R_i(X)$ is the ith row vector of X and $C_j(Y)$ is the jth column vector of Y.]

(d) Let M be as defined in part (b). Show that we can compute M^n in NC^2. Deduce that $\mathsf{NL} \subseteq \mathsf{NC}^2$.

(e) Strengthen the bound in part (d) to show that $\mathsf{NL} \subseteq \mathsf{AC}^1$.

(f) The complexity class SAC^k is the set of languages decidable by AC^k circuits, where the AND gets have fan-in 2. Compare this to general AC^k circuits, the AND gates have unbounded fan-in. Strengthen part (e) to show that $\mathsf{NL} \subseteq \mathsf{SAC}^1$.

Remark 40. We have the following relations:

$$\mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{SAC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2.$$

We have not proven that $\mathsf{AC}^0 \neq \mathsf{NC}^1$, nor have we shown that $\mathsf{NC}^1 \subseteq \mathsf{L}$.

2 Computability

Our goal in this section is to introduce the notion of Turing Machines, as well as some basic notions from Computability Theory. Computational Complexity began by trying to extend notions of Computability to solvable (decidable) problems. Rather than providing our algoritheorems with unlimited resources, we sought to ask which questions were decidable within given resource constraints. The most natural resource contraints to consider were time and space. While Computability Theory is very well understood, many of the analogues in Computational Complexity are not. We begin with an introduction to Turing Machines and undecidability.

These notes follow closely [Sav97][Chapter 5] and [Lev20].

2.1 Turing Machines

The Turing Machine is an abstract model of computation which has a finite set of states, as well as an infinite work tape. We assume the work tape has a left-most starting cell and is only infinite in the rightwards direction. Each cell contains a letter or is left blank. The Turing Machine has a tape head, which starts over the left-most cell. The tape head can only examine one cell at a time. Now computation steps are precisely state transitions. The Turing Machine transition function takes as input the current state and the current letter under its tape head. The transition function then does three things:

- Transitions the Turing Machine to a new state,
- Writes a new symbol to the current cell (possibly the blank symbol β, which corresponds to erasing the current letter), and
- Moves the tape head one cell, either to the left or to the right.

Intuitively, it may be helpful to think of the Turing Machine as managing an infinite, doubly-linked list data structure, where each node stores a writeable letter.

Turing Machines are used to solve decision problems. Given a language L, we ask whether it is possible to design a Turing Machine M such that for every string x, M correctly decides whether $x \in L$. To this end, the Turing Machine has explicit accept and reject states, which take effect immediately upon being reached. We will see later that not every language L can be decided in this manner. Given a Turing Machine M, we denote the set of strings that M accepts as $L(M)$.

2.1.1 Deterministic Turing Machine

We now define the standard deterministic Turing Machine as follows.

Definition 41 (Deterministic Turing Machine)**.** A Deterministic Turing Machine is a 7-tuple

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

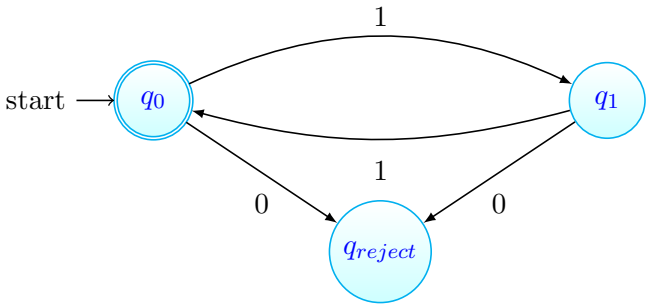
where Q, Σ, Γ are all finite sets and:

- Q is the set of states.
- Σ is the input alphabet, not containing the blank symbol β.
- Γ is the tape alphabet, where $\beta \in \Gamma$ and $\Sigma \subset \Gamma$. In particular, it may be helpful for the Turing Machine to have additional letters to use internally.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L, R}\}$ is the transition function, which takes a state and tape character and returns the new state, the tape character to write to the current cell, then a direction for the tape head to move one cell to the left or right (denoted by L or R respectively).
- $q_0 \in Q$, the initial state.
- $q_{\text{accept}} \in Q$, the accept state.

- $q_{\text{reject}} \in Q$, the reject state where $q_{\text{reject}} \neq q_{\text{accept}}$.

We now consider some example of a Turing Machine.

Example 42. Let $\Sigma = \{0, 1\}$ and let $L = \{1^{2k} : k \in \mathbb{N}\} = (11)^*$. For a reader who is familiar with regular languages, observe that L is regular as L can be expressed using the regular expression $(11)^*$. A simplified computational model, known as a finite state automaton (FSM), can easily be constructed to accept L. Such a FSM diagram is provided below.



Now let's construct a Turing Machine to accept $(11)^*$. The construction of the Turing Machine, is in fact, almost identical to that of the finite state automaton. The Turing Machine will start with the input string on the far-left of the tape, with the tape head at the start of the string. The Turing Machine has $Q = \{q_0, q_1, q_{reject}, q_{accept}\}$, $\Sigma = \{0, 1\}$, and $\Gamma = \{0, 1, \beta\}$. Let the Turing Machine start at q_0 and read in the character under the tape head. If it is not a 1 or the empty string, enter q_{reject} and halt. Otherwise, if the string is empty, enter q_{accept} and halt. On the input of a 1, transition to q_1 and move the tape head one cell to the right. While in q_1, read in the character on the tape head. If it is a 1, transition to q_0 and move the tape head one cell to the right. Otherwise, enter q_{reject} and halt. The Turing Machine always halts, and accepts the string if and only if it halts in state q_{accept}.

Observe the similarities between the Turing Machine and finite state automaton. The intuition should follow that any language accepted by a finite state automaton (ie., any regular language) can also be accepted by a Turing Machine. Formally, the Turing Machine simulates the finite state automaton by omitting the ability to write to the tape or move the tape head to the left. We now consider a second example of a Turing Machine accepting a more complicated language (namely, a context-free language).

Example 43. Let $\Sigma = \{0, 1\}$ and let $L = \{0^n 1^n : n \in \mathbb{N}\}$. For a reader familiar with automata theory, we note that L is context-free, but not regular. We provide a Turing Machine to accept this language. The Turing Machine has a tape alphabet of $\Gamma = \{0, 1, \hat{0}, \hat{1}\}$ and set of states $Q = \{q_0, q_{\text{find-1}}, q_{\text{find-0}}, q_{\text{validate}}, q_{\text{accept}}, q_{\text{reject}}\}$. Conceptually, rather than using a stack as a pushdown automaton would, the Turing Machine will use its tape head. Intuitively, the Turing Machine starts with a 0 and marks it, then moves the tape head to the right one cell at a time looking for a corresponding 1 to mark. Once it finds and marks the 1, the Turing Machine then moves the tape head to the left one cell at a time searching for the next unmarked 0 to mark. It then repeats this procedure, looking for another unmarked 1 to mark. If it finds an unpaired 0 or 1, it rejects the string. This procedure repeats until either the string is rejected, or we mark all pairs of 0's and 1's. In the latter case, the Turing Machine accepts the string.

So initially the Turing Machine starts at q_0 with the input string on the far-left of the tape, with the tape head above the first character. If the string is empty, the Turing Machine enters q_{accept} and halts. If the first character is a 1, the Turing Machine enters q_{reject} and halts. If the first character is 0, the Turing Machine replaces it with $\hat{0}$. It then moves the tape head to the right one cell and transitions to state $q_{\text{find-1}}$.

At state $q_{\text{find-1}}$, the Turing Machine moves the tape head to the right and stays at $q_{\text{find-1}}$ for each $0 or \hat{0}$ character it reads in and writes back the character it parsed. If at $q_{\text{find-1}}$ and the Turing Machine reads 1, then it writes $\hat{1}$ to the tape, moves the tape head to the left, and transitions to $q_{\text{find-0}}$. If no 1 is found, the Turing Machine enters q_{reject} and halts.

At state $q_{\text{find-0}}$, the Turing Machine moves the tape head to the left and stays at $q_{\text{find-0}}$ until it reads in 0. If the Turing Machine reads in 0 at state $q_{\text{find-0}}$, it replaces the 0 with $\hat{0}$. It then moves the tape head to the

right one cell and transitions to state $q_{\text{find-1}}$. If no 0 is found once we have reached the far-left cell, the Turing Machine transitions to state q_{validate}.

At state q_{validate}, the Turing Machine transitions to the right one cell at a time while staying at q_{validate}. If it encounters any 1, it enters q_{reject}. Otherwise, the Turing Machine enters q_{accept} once reading in β.

Remark 44. Now that we provided formal specifications for a couple Turing Machines, we provide a more abstract representation from here on out. We are more interested in studying the power of Turing Machines rather than the individual state transitions, so high level procedures suffice for our purposes. This high level procedure from the above example provides sufficient detail to simulate the Turing Machine. So for our purposes, this level of detail is sufficient:

"*Intuitively, the Turing Machine starts with a* 0 *and marks it, then moves the tape head to the right one cell at a time looking for a corresponding* 1 *to mark. Once it finds and marks the* 1*, the Turing Machine then moves the tape head to the left one cell at a time searching for the next unmarked* 0 *to mark. It then repeats this procedure, looking for another unmarked* 1 *to mark. If it finds an unpaired* 0 *or* 1*, it rejects the string. This procedure repeats until either the string is rejected, or we mark all pairs of* 0*'s and* 1*'s. In the latter case, the Turing Machine accepts the string.*"

Definition 45 (Recursively Enumerable Language)**.** A language L is said to be *recursively enumerable* if there exists a deterministic Turing Machine M such that $L(M) = L$. Note that if $\omega \notin L$, the machine M need not halt on ω.

Definition 46 (Decidable Language)**.** A language L is said to be *decidable* if L is there exists some Turing Machine M such that $L(M) = L$ and M halts on all inputs. We say that M *decides* L.

Remark 47. Every decidable language is clearly recursively enumerable. The converse is not true, and this will be shown later with the undecidabilitiy of the Halting problem.

2.1.2 Multitape Turing Machine

The Turing Machine is quite a robust model, in the sense that the standard deterministic model accepts and decides precisely the same languages as the multitape and non-deterministic variants. It should be noted that one model may actually be more efficient than another. In regards to language acceptance and computability, we ignore issues of efficiency and complexity. However, the same techniques we use to show that these models are equally powerful can be leveraged to show that two models of computation are equivalent both with regards to power and some measure of efficiency. That is, to show that the two models solve the same set of problems using a comparable amount of resources (e.g., polynomial time). This is particularly important in complexity theory, but we also leverage these techniques when showing Turing Machines equivalent to other models such as (but not limited to) the RAM model and the λ-calculus.

We begin by introducing the Multitape Turing Machine.

Definition 48 (Multitape Turing Machine)**.** A *k-tape Turing Machine* is an extension of the standard deterministic Turing Machine in which there are k tapes with infinite memory and a fixed beginning. The input initially appears on the first tape, starting at the far-left cell. The transition function is the addition difference, allowing the k-tape Turing Machine to simultaneously read from and write to each of the k-tapes, as well as move some or all of the tape cells. Formally, the transition function is given below:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\text{L, R, S}\}^k.$$

The expression:

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, S, \ldots, R)$$

indicates that the TM is on state q_i, reading a_m from tape m for each $m \in [k]$. Then for each $m \in [k]$, the TM writes b_m to the cell in tape m highlighted by its tape head. The mth component in $\{\text{L, R, S}\}^k$ indicates that the mth tape head should move left, right, or remain stationary respectively.

Our first goal is to show that the standard deterministic Turing Machine is equally as powerful as the k-tape Turing Machine, for any $k \in \mathbb{N}$. We need to show that the languages accepted (decided) by deterministic Turing Machines are exactly those languages accepted (decided) by the multitape variant. The initial approach of

a set containment argument is correct. The details are not as intuitively obvious. Formally, we show that for any multitape Turing Machine, there exists an deterministic Turing Machine; and for any deterministic Turing Machine, there exists an equivalent multitape Turing Machine. In other words, we show how one model simulates the other and vice-versa. This implies that the languages accepted (decided) by one model are precisely the languages accepted (decided) by the other model.

Theorem 49. A language is recursively enumerable (decidable) if and only if some multitape Turing Machine accepts (decides) it.

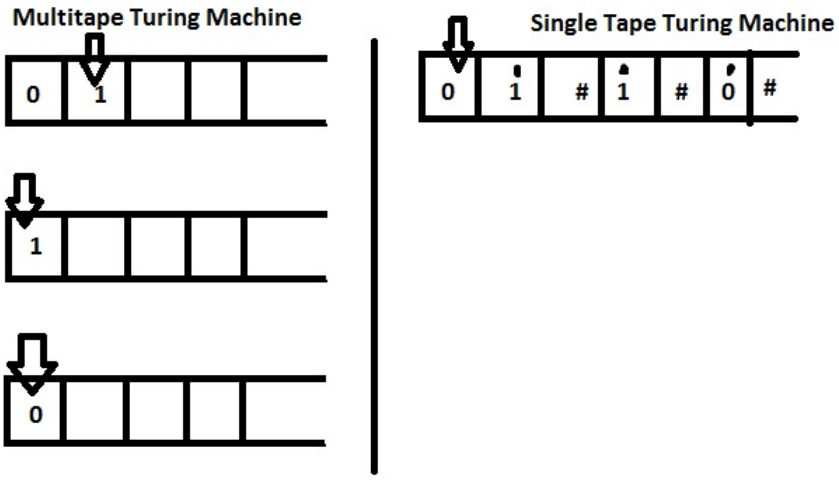
Proof. We begin by showing that the multitape Turing Machine model is at least as power as the standard deterministic Turing Machine model. Clearly, a standard deterministic Turing Machine is a 1-tape Turing Machine. So every language accepted (decided) by a standard deterministic Turing Machine is also accepted (decided) by some multitape Turing Machine.

Conversely, let M be a multitape Turing Machine with k tapes. We construct a standard determnistic Turing Machine M' to simulate M, which shows that $L(M') = L(M)$. As M has k-tapes, it is necessary to represent the strings on each of the k tapes on a single tape. It is also necessary to represent the placement of each of the k tape heads of M on the one tape of M'. This is done by using a special marker. For each symbol $c \in \Gamma(M)$, we include c and $\hat{c}$ in Γ', where $\hat{c}$ indicates a tape head on M is on the character c. We then have a special delimiter symbol $\#$, which separates the strings on each of the k tapes. So $|\Gamma(M')| = 2|\Gamma(M)| + 1$. M' simulates M in the following manner.

- M' scans the tape from the first delimiter to the last delimiter to determine which symbols are marked as under the tape heads on M.
- M' then evaluates the transition function of M, then makes a second pass along the tape to update the symbols on the tape.
- If at any point, the tape head of M' falls on a delimiter symbol $\#$, M would have reached the end of that specific tape. So M' shifts the string, cell by cell, starting at the current delimiter inclusive. A blank symbol is then overwritten on the delimiter.

Thus, M' simulates M. So any language accepted (decided) by a multitape Turing Machine is accepted (decided) by a single tape Turing Machine. □

Below is an illustration of a multitape Turing Machine an an equivalent single tape Turing Machine.



Remark 50. If the k-tape Turing Machine takes T steps, then each tape uses at most $T + 1$ cells. So the equivalent one-tape deterministic Turing Machine constructed in the proof of Theorem 4.1 takes $(k \cdot (T+1))^2 = \mathcal{O}(T^2)$ steps.

2.1.3 Non-deterministic Turing Machines

We now introduce the non-deterministic Turing Machine. The non-deterministic Turing Machine has a single, infinite tape with an end at the far-left. Its sole difference with the deterministic Turing Machine is the transition function. For a given state, letter pair $(q, a) \in Q \times \Gamma$, the transition function evaluated at this pair $\delta(q, a)$ specifies a unique next state. For a non-deterministic Turing Machine, there may be multiple permissible states to visit next. We make this precise in the definition by defining the transition function to return a subset of elements from $Q \times \Gamma \times \{L, R\}$. For a computation, the non-deterministic Turing Machine makes a selection of which of the possible transitions to consider from the permissible choices. With this in mind, we formalize the non-deterministic Turing Machine.

Definition 51 (Non-deterministic Turing Machine)**.** A Non-Deterministic Turing Machine is a 7-tuple

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

where Q, Σ, Γ are all finite sets and:

- Q is the set of states.
- Σ is the input alphabet, not containing the blank symbol β.
- Γ is the tape alphabet, where $\beta \in \Gamma$ and $\Sigma \subset \Gamma$. In particular, it may be helpful for the Turing Machine to have additional letters to use internally.
- $\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$ is the transition function, which takes a state and tape character and returns the new state, the tape character to write to the current cell, then a direction for the tape head to move one cell to the left or right (denoted by L or R respectively).
- $q_0 \in Q$, the initial state.
- $q_{\text{accept}} \in Q$, the accept state.
- $q_{\text{reject}} \in Q$, the reject state where $q_{\text{reject}} \neq q_{\text{accept}}$.

We now show that the deterministic and non-deterministic variants are equally powerful. The proof for this is by simulation. Before introducing the proof, let's conceptualize this. Earlier in this section, a graph theory intuition was introduced for understanding the definition of what it means for a string to be accepted by a non-deterministic Turing Machine. That definition of string acceptance dealt with the existence of a choice string such that the non-deterministic Turing Machine would reach the accept state q_{accept} from the starting state q_0. The graph theory analog was that there existed a path for the input string from q_0 to q_{accept}.

So the way a deterministic Turing Machine simulates a non-deterministic Turing Machine is through, essentially, a breadth-first search. More formally, what actually happens is that a deterministic multitape Turing Machine is used to simulate a non-deterministic Turing Machine. It does this by generating choice strings in lexicographic order and simulating the non-deterministic Turing Machine on each choice string until the string is accepted or all the possibilities are exhausted.

Given the non-deterministic Turing Machine has a finite number of transitions, there are a finite number of choice input strings to generate. Thus, a multitape deterministic Turing Machine will always be able to determine if an input string is accepted by the non-deterministic Turing Machine. It was already proven that a multitape Turing Machine can be simulated by a standard deterministic Turing Machine, so it follows that any language accepted by a non-deterministic Turing Machine can also be accepted by a deterministic Turing Machine.

Theorem 52. A language is recursively enumerable (decidable) if and only if it is accepted (decided) by some non-deterministic Turing Machine.

Proof. A deterministic Turing Machine is clearly non-deterministic. So it suffices to show that every non-deterministic Turing Machine has an equivalent deterministic Turing Machine. From Theorem 49, it suffices to construct a multitape Turing Machine equivalent for every non-deterministic Turing Machine. The proof is

by simulation. Let M be a non-deterministic Turing Machine.

We construct a three-tape Turing Machine M' to simulate all possibilities. The first tape contains the input string and is used as a read-only tape. The second tape is used to simulate M, and the third tape is the enumeration tape in which we enumerate the branches of the non-deterministic Turing Machine. Let:

$$b = \max_{q \in Q, a \in \Gamma} |\delta_M(q, a)|.$$

The tape alphabet of M' is $\Gamma(M) \cup [b]$. On the third tape of M', we enumerate strings over $[b]^n$ in lexical order, where n is the length of the input string. At state i in the computation, we utilize the transition indexed by the number on the ith cell on the third tape.

Formally, M' works as follows:

1. M' is started with the input ω on the first tape.
2. We then copy the input string to the second tape and generate 0^ω.
3. Simulate M on ω using the choice string on ω. If at any point, the transition specified by the third tape is undefined (which may occur if too few choices are available), we terminate the simulation of M and generate the next string in lexical order on the third tape. We then repeat step (3).
4. M' accepts (rejects) ω if and only some simulation of M on ω accepts (rejects) ω.

By construction, $L(M') = L(M)$, yielding the desired result. □

2.1.4 Exercises

(Recommended) Problem 21. Let $L = \{a^n b^{2n} : n \in \mathbb{Z}^+\}$. Provide a high level description for a Turing Machine that accepts L. Your description may suppress individual state transitions, but it should be detailed enough to allow you (or one of your classmates) to construct the exact transition function. In particular, your description should specify the movement of the tape head and when the Turing Machine writes to the tape.

2.2 Undecidability

In this section, we examine the limits of computation as a means to solve problems. This is important for several reasons. First, problems that cannot be solved need to be simplified to a formulation that is more amenable to computational approaches. Second, the techniques used in proving languages to be undecidable, including reductions and diagonalization, appear repeatedly in complexity theory. Lastly, undecidability is an interesting topic in its own right.

The canonical result in computability theory is the undecidability of the halting problem. Intuitively, no algoritheorem exists to decide if a Turing Machine halts on an arbitrary input string. While the result seems abstract and unimportant, the results are actually far reaching. Software engineers seek better ways to determine the correctness of their programs. The undecidability of the halting problem provides an impossibility result for software engineers; no such techniques exist to validate arbitrary computer programs. We formalize this with the following language:

$$A_{\text{TM}} = \{\langle M, w\rangle : M \text{ is a Turing Machine that accepts } w\}.$$

It turns out that A_{TM} is undecidable. We actually start by showing that the following language undecidable:

$$L_{\text{diag}} = \{\omega_i : \omega_i \text{ is the ith string in } \Sigma^*, \text{ which is accepted by the ith Turing Machine } M_i\}.$$

L_{diag} is designed to leverage a diagonalization argument. We note that Turing Machines are representable as finite strings (just like computer programs), and that the set of finite length strings over an alphabet is countable. So we can enumerate Turing Machines using $\mathbb{N}$. Similarly, we also enumerate input strings from Σ^* using $\mathbb{N}$. Before proving L_{diag} undecidable, we need to show that decidable languages are precisely those that are recursively enumerable and whose complements are recursively enumerable. We unpack this idea before proving the theorem.

Definition 53. Let REC be the set of decidable languages, and let RE be the set of recursively enumerable languages. Denote coRE to be the set of languages L, where $\overline{L} \in \mathsf{RE}$.

Theorem 54. A language L is decidable if and only if L and $\overline{L}$ are recursively enumerable. That is, $\mathsf{REC} = \mathsf{RE} \cap \mathsf{coRE}$.

Proof. Suppose first L is decidable, and let M be a decider for L. As M decides L, M also accepts L. So L is recursively enumerable. Now define $\overline{M}$ to be a Turing Machine that, on input ω, simulates M on ω. $\overline{M}$ accepts (rejects) ω if and only if M rejects (accepts) ω. As M is a decider, $\overline{M}$ decides $\overline{L}$. So $\overline{L}$ is also recursively enumerable.

Conversely, suppose L and $\overline{L}$ are recursively enumerable. Let B and $\overline{B}$ be Turing Machines that accept L and $\overline{L}$ respectively. We construct a Turing Machine K to decide L. K works as follows. On input ω, K simulates B and $\overline{B}$ in parallel on ω. As L and $\overline{L}$ are recursively enumerable, at least one of B or $\overline{B}$ will halt and accept ω. If B accepts ω, then so does K. Otherwise, $\overline{B}$ accepts ω and K rejects ω. So K decides L. □

Remark 55. While $\mathsf{RE} \cap \mathsf{coRE}$ is very well understood, the analogues in Computational Complexity remain long-standing open problems. As an example, it is unknown whether $\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$.

In order to show that L_{diag} is undecidable, we have by Theorem 54 that it suffices to show $\overline{L_{\text{diag}}}$ is not recursively enumerable. This is the meat of the proof for the undecidability of the halting problem. It turns out that L_{diag} is recursively enumerable, which is easy to see.

Theorem 56. L_{diag} is recursively enumerable.

Proof. We construct an acceptor D for L_{diag} which works as follows. On input ω_i, D simulates M_i on ω_i and accepts ω_i if and only if M_i accepts ω_i. So $L(D) = L_{\text{diag}}$, and L_{diag} is recursively enumerable. □

We now show that $\overline{L_{\text{diag}}}$ is not recursively enumerable.

Theorem 57. $\overline{L_{\text{diag}}}$ is not recursively enumerable.

Proof. Suppose to the contrary that $\overline{L_{\text{diag}}}$ is recursively enumerable. Let $k \in \mathbb{N}$ such that the Turing Machine M_k accepts $\overline{L_{\text{diag}}}$. Suppose $\omega_k \in \overline{L_{\text{diag}}}$. Then M_k accepts ω_k, as $L(M_k) = \overline{L_{\text{diag}}}$. However, $\omega_k \in \overline{L_{\text{diag}}}$ implies that M_k does not accept ω_k, a contradiction.

Suppose instead $\omega_k \notin \overline{L_{\text{diag}}}$. Then $\omega_k \notin L(M_k) = \overline{L_{\text{diag}}}$. Since M_k does not accept ω_k, it follows by definition of $\overline{L_{\text{diag}}}$ that $\omega_k \in \overline{L_{\text{diag}}}$, a contradiction. So $\omega_k \in \overline{L_{\text{diag}}}$ if and only if $\omega_k \notin \overline{L_{\text{diag}}}$. So $\overline{L_{\text{diag}}}$ is not recursively enumerable. □

Corollary 58. L_{diag} is undecidable.

Proof. This follows immediately from Theorem 54, as L_{diag} is recursively enumerable, while $\overline{L_{\text{diag}}}$ is not. □

2.3 Reducibility

The goal of a reduction is to transform one problem A into another problem B. If we know how to solve this second problem B, then this yields a solution for A. Essentially, we transform A into B, solve it in B, then apply this solution in A. Reductions thus allow us to order problems based on how hard they are. In particular, if we know that A is undecidable, a reduction immediately implies that B is undecidable. Otherwise, a Turing Machine to decide B could be used to decide A. Reductions are also a standard tool in complexity theory, where we transform problems with some bound on resources (such as time or space bounds). In computability theory, reductions need only be computable. We formalize the notion of a reduction with the following two definitions.

Definition 59 (Computable Function)**.** A function $f : \Sigma^* \to \Sigma^*$ is a *computable function* if there exists some Turing Machine M such that on input ω, M halts with just $f(\omega)$ written on the tape.

Definition 60 (Many-to-One Reduction)**.** Let A, B be languages. A *many-to-one reduction* from A to B is a computable function $f : \Sigma^* \to \Sigma^*$ such that $\omega \in A$ if and only if $f(\omega) \in B$. We say that A is reducible to B, denoted $A \leq_m B$, if there exists a many-to-one reduction from A to B.

We deal with reductions in a similar high-level manner as Turing Machines, providing sufficient detail to indicate how the original problem instances are transformed into instances of the target problem. In order for reductions to be useful in computability theory, we need an initial undecidable problem. This is the language L_{diag} from the previous section. With the idea of a reduction in mind, we proceed to show that A_{TM} is undecidable.

Theorem 61. A_{TM} is undecidable.

Proof. It suffices to show $L_{\text{diag}} \leq_m A_{\text{TM}}$. The function $f : \Sigma^* \to \Sigma^*$ maps $\omega_i \in L_{\text{diag}}$ to $\langle M_i, \omega_i \rangle \in A_{\text{TM}}$. Any string not in L_{diag} is mapped to ϵ under f. As Turing Machines are enumerable, a Turing Machine can clearly write $\langle M_i, \omega_i \rangle$ to the tape when started with ω_i. So f is computable. Furthermore, observe that $\omega_i \in L_{\text{diag}}$ if and only if $\langle M_i, \omega_i \rangle \in A_{\text{TM}}$. So f is a reduction from L_{diag} to A_{TM} and we conclude that A_{TM} is undecidable. □

With A_{TM} in tow, we prove the undecidability of the halting problem, which is given by:

$$H_{\text{TM}} = \{\langle M, w \rangle : M \text{ is a Turing Machine that halts on the string } w\}.$$

Theorem 62. H_{TM} is undecidable.

Proof. It suffices to show that $A_{\text{TM}} \leq_m H_{\text{TM}}$. Each element of A_{TM} is clearly an element of H_{TM}. So we map each element of A_{TM} to itself in H_{TM}, and all other strings to ϵ. This map is clearly a reduction, so H_{TM} is undecidable. □

The reductions to show A_{TM} and H_{TM} undecidable have been rather trivial. We will examine some additional undecidable problems. In particular, the reduction will be from A_{TM}. The idea moving forward is to pick a desirable solution and return it if and only if a Turing Machine M halts on a string ω. A decider for the target problem would thus give us a decider for A_{TM}, which is undecidable. We illustrate the concept below.

Theorem 63. Let $E_{\text{TM}} = \{\langle M \rangle : M \text{ is a Turing Machine s.t. } L(M) = \emptyset\}$. E_{TM} is undecidable.

Proof. It suffices to show that $A_{\text{TM}} \leq_m E_{\text{TM}}$. For each instance of $\langle M, \omega \rangle \in A_{\text{TM}}$, we construct an instance of E_{TM} M' as follows. On input $x \neq \omega$, M' rejects x. Otherwise, M' simulates M on ω. If M accepts (rejects) ω, then M' rejects (accepts) ω. So $\langle M, \omega \rangle \in A_{\text{TM}}$ implies that $M' \in E_{\text{TM}}$. So E_{TM} is undecidable. □

We use the same idea to show that it is undecidable if a Turing Machine accepts the empty string. Observe above that our desirable solution for E_{TM} was $\emptyset$. Then M' accepted the desired solution if and only if the instance Turing Machine M accepted ω. We *conditioned* acceptance of the target instance based on the original problem. In this next problem, the target solution is ϵ, the empty string.

Theorem 64. Let $L_{\text{ES}} = \{\langle M \rangle : M \text{ is a Turing Machine that accepts } \epsilon\}$. L_{ES} is undecidable.

Proof. We show that $A_{\text{TM}} \leq_m E_{\text{TM}}$. Let $\langle M, \omega \rangle \in A_{\text{TM}}$. We construct an instance of L_{ES}, M', as follows. On input $x \neq \epsilon$, M' rejects x. Otherwise, M' simulates M on ω. M' accepts ϵ if and only if M accepts ω. So $\langle M, \omega \rangle \in A_{\text{TM}}$ if and only if $M' \in L_{\text{ES}}$. This function is clearly computable, so L_{ES} is undecidable. □

Recall that any regular language is decidable. We may similarly ask if a given language is regular. It turns out that this new problem is undecidable.

Theorem 65. Let $L_{\text{Reg}} = \{L : L \text{ is regular }\}$. L_{Reg} is undecidable.

Proof. We reduce A_{TM} to L_{Reg}. Let $\langle M, \omega \rangle \in A_{\text{TM}}$. We construct a Turing Machine M' such that $L(M')$ is regular if and only if M accepts ω. M' works as follows. On input x, M' accepts x if it is of the form $0^n 1^n$ for some $n \in \mathbb{N}$. Otherwise, M' simulates M on ω, and accepts x if and only if M accepts ω. So $L(M') = \Sigma^*$ if and only if M accepts ω, and $L(M') = \{0^n 1^n : n \in \mathbb{N}\}$ otherwise which is not regular. Thus, $L(M') \in L_{\text{Reg}}$ if and only if $\langle M, \omega \rangle \in A_{\text{TM}}$. So L_{Reg} is undecidable. □

The common theme in each of these undecidability results is that not every language satisfies the given property. This leads us to one of the major results in computability theory: Rice's Theorem. Intuitively, Rice's Theorem states that any non-trivial property is undecidable. A property is said to be trivial if it applies to either every language or no language. We formalize it as follows.

Theorem 66 (Rice)**.** Let C be a non-empty, proper subset of RE. Then C is undecidable.

Proof. We reduce A_{TM} to C. Without loss of generality, suppose $\emptyset \in C$. As C is a proper subset of RE, $\overline{C}$ is non-empty. Let $L \in \overline{C}$. Let $\langle M, \omega \rangle \in A_{\text{TM}}$. We consturct a Turing Machine M' as follows. On input x, M' rejects x if $x \notin L$. Otherwise, M' simulates M on ω. M' rejects x if and only if M accepts ω. So $L(M') = \emptyset \in C$ if and only if $\langle M, \omega \rangle \in A_{\text{TM}}$. Otherwise, $L(M') = L$. Thus, C is undecidable. □

Remark 67. Observe that the proof of Rice's Theorem is a template for the previous undecidability proofs in this section. Rice's Theorem generalizes all of our undecidability results and provides an easy test to determine if a language is undecidable. In short, to show a property undecidable, it suffices to exhibit a language satisfying said property and a language that does not satisfy said property.

2.3.1 Exercises

(Recommended) Problem 22. Consider the language:

$$L = \{\langle M, \omega \rangle : M \text{ halts in at most 2020 steps when run on } \omega\}.$$

Do the following.

(a) Show that L is recursively enumerable.

(b) Give a many-to-one reduction from Halt_{TM} to L. Conclude that L is undecidable.

(Recommended) Problem 23. *Fermat's Last Theorem* states that there are no positive integer solutions to the equation $x^n + y^n = z^n$ for any integer $n > 2$. This theorem was conjectured by Fermat in 1637 and finally proven in 1994 by Andrew Wiles. Consider the following Turing Machine M, which searches for counter-examples to Fermat's Last Theorem:

"M ignores any input (so it can be assumed to run when started on a blank tape), and proceeds to enumerate 4-tuples of the form (x, y, z, n) where x, y, z, n are positive integers and $n > 2$. After enumerating each 4-tuple,

M computes x^n, y^n, z^n, then checks that $x^n + y^n = z^n$ and $n > 2$. M halts if $x^n + y^n = z^n$ and $n > 2$. Otherwise, M proceeds to the next 4-tuple."

Explain how an algorithm to decide the Halting Problem could be used to decide if Fermat's Last Theorem were true. [**Note:** You do not need to know anything about Fermat's Last Theorem to answer this question.]

(Recommended) Problem 24. Let $\mathcal{C}$ be the set of recursively enumerable languages that are co-finite. That is, $L \in \mathcal{C}$ precisely if $\overline{L}$ is finite. Using Rice's Theorem, show that $\mathcal{C}$ is undecidable.

2.4 Oracle Turing Machines

In this section, we introduce Oracle Turing Machines. The key idea is as follows. Suppose we are trying to solve a computational problem L. Given access to an algorithm A to solve a different (possibly harder) problem L', can we use A to solve L? If so, how many queries to A are required? Note that we make no assumptions that L' is decidable, or even recursively enumerable. In this sense, we can make precise bottlenecks or barriers in solving computational problems. In particular, measuring the number of queries to a given oracle, known as the *query complexity*, is a very active area in Computational Complexity (particularly in Quantum Complexity). It is quite possible that one has come across examples of this already in an Algorithms or Theory of Computation course, in reducing a search problem to the corresponding decision problem. For example, one may have seen already how to use a SAT oracle to find explicit satisfying instances for Boolean formulas.

We begin with the definition of an Oracle.

Definition 68. Let Σ be an alphabet. An *oracle* is a set $\mathcal{O} \subset \Sigma^*$.

We next introduce the Oracle Turing Machine model. Informally, an Oracle Turing Machine M is a Turing Machine with the ability to query an oracle. We think of M as separate from the underlying oracle. That is, the transition function is associated to M and is independent of the oracle that is used. While the transition function is fixed, whether M accepts the input string ω depends significantly on the underlying oracle. Suppose we have the Oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. On input ω, M poses an Oracle query to ask whether the string x is in the Oracle. If $x \in \mathcal{O}_1$ and $x \notin \mathcal{O}_2$, then M may behave differently based on which oracle is used. In particular, equipping M with $\mathcal{O}_1$ (which we denote $M^{\mathcal{O}_1}$) might result in $M^{\mathcal{O}_1}(\omega) = 1$, while $M^{\mathcal{O}_2}(\omega) = 0$.

Definition 69. An *Oracle Turing Machine* M is a Turing Machine, which is parameterized by an Oracle $\mathcal{O}$. M with three additional states: $q_{\text{query}}, q_{\text{yes}}$, and q_{no}, as well as an additional tape to which M has both read and write access. The Turing Machine may query the Oracle by entering the state q_{query} and writing a string x to the additional tape. If x is in the underlying Oracle, then M transitions to q_{yes}. Otherwise, M transitions to q_{no}. Querying the Oracle takes a single computation step.

With the notion of an Oracle Turing Machine in tow, we can now begin talking about complexity classes relative to Oracles. We refer to such classes as *relativized*. Note as well that some care needs to be taken, in that we can only define relativized complexity classes that have characterizations in terms of Turing Machines. We have not discussed how to define relativized complexity classes where the underlying model of computation is a circuit, for instance.

Definition 70. Let $\mathcal{C}$ be a complexity class characterized by Turing Machines, and let $\mathcal{O}$ be an Oracle. A language $L \in \mathcal{C}^{\mathcal{O}}$ if there exists a $\mathcal{C}$-Turing Machine M with oracle $\mathcal{O}$, denoted $M^{\mathcal{O}}$, such that $M^{\mathcal{O}}$ accepts L.

Example 71. As an example, the language $L \in \mathsf{P}^{\mathsf{SAT}}$ if there exists a polynomial-time deterministic, Oracle Turing Machines M^{SAT} that decides L.

We may want to talk about relativizing a complexity class $\mathcal{A}$ with respect to a second complexity class $\mathcal{B}$, as opposed to just an individual problem. This brings us to the following definition.

Definition 72. Let $\mathcal{A}$ and $\mathcal{B}$ be complexity classes defined in terms of Turing Machines. Define:

$$\mathcal{A}^{\mathcal{B}} := \bigcup_{L \in \mathcal{B}} \mathcal{A}^L.$$

Remark 73. We note that if $\mathcal{B}$ has a complete problem L under $\mathcal{A}$-computable reductions, then $\mathcal{A}^{\mathcal{B}} = \mathcal{A}^L$. We organize this statement with the next theorem, the proof of which is left as an exercise.

Theorem 74. Let $\mathcal{A}$ and $\mathcal{B}$ be complexity classes defined in terms of Turing Machines. Suppose that $\mathcal{B}$ has a complete problem L, under $\mathcal{A}$-computable many-to-one reductions. Then $\mathcal{A}^L = \mathcal{A}^{\mathcal{B}}$.

Example 75. Take $\mathcal{A} = \mathsf{P}$, $\mathcal{B} = \mathsf{NP}$, and $L = \mathsf{SAT}$. We note that NP-complete problems are P-computable. So by Theorem 74 $\mathsf{P}^{\mathsf{NP}} = \mathsf{P}^{\mathsf{SAT}}$.

Example 76. Take $\mathcal{A} = \mathsf{L}$, $\mathcal{B} = \mathsf{NP}$, and $L = \mathsf{SAT}$. We note that NP-complete problems are P-computable. While L-computable functions are computable in P, the converse is not known to be true. So Theorem 74 is not known to apply.

Example 77. Take $\mathcal{A} = \mathsf{P}$ and $\mathcal{B} = \mathsf{BPP}$.[4] It remains open as to whether BPP has complete problems. So under current knowledge, Theorem 74 is not applicable in this setting.

Now that we have some handle on the notion of an Oracle, we conclude with the notion of a Turing reduction. Informally, we say that a language L_1 is *Turing reducible* to L_2 if given an algorithm to solve L_2, we can solve L_1. The idea of providing an algorithm is formalized with the notion of Oracle Turing Machines.

Definition 78 (Turing Reduction)**.** Let L_1 and L_2 be languages. We say that L_1 is *Turing reducible* to L_2, dentoed $L_1 \leq_T L_2$, if there exists an Oracle Turing Machine M^{L_2} that decides L_1.

Remark 79. Recall that the notion of a Turing reduction is how we intuitively explain why the many-to-one Karp reductions indeed establish hardness. Suppose that $L_1 \leq_m L_2$, and let $\varphi : \Sigma^* \to \Sigma^*$ be a many-to-one Karp reduction. Suppose that we can solve L_2. Let $\omega \in \Sigma^*$. In order to decide if $\omega \in L_1$, we compute $\varphi(\omega)$ and use the algorithm for L_2 to test whether $\varphi(\omega) \in L_2$. As φ is a reduction, we conclude that $\omega \in L_1$ if and only if the algorithm for L_2 returns that $\varphi(\omega) \in L_2$. So in fact, the fact that L_1 is many-to-one Karp reducible to L_2 implies that L_1 is Turing reducible to L_2.

2.4.1 Exercises

(Recommended) Problem 25. Do the following.

(a) Show that $\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{NP}}$.

(b) Show that $\overline{\mathsf{SAT}} \in \mathsf{P}^{\mathsf{NP}}$. [**Hint:** Use the fact that $\mathsf{P}^{\mathsf{NP}} = \mathsf{P}^{\mathsf{SAT}}$.]

(c) Deduce that $\mathsf{coNP} \subseteq \mathsf{P}^{\mathsf{NP}}$.

(Recommended) Problem 26. Let $\mathcal{A}$ and $\mathcal{B}$ be complexity classes defined in terms of Turing Machines. Suppose that $\mathcal{B}$ has a complete problem L, under $\mathcal{A}$-computable reductions. Prove that $\mathcal{A}^L = \mathcal{A}^{\mathcal{B}}$.

Definition 80. Let $\mathcal{A}$ and $\mathcal{B}$ be complexity classes defined in terms of Turing Machines. We say that $\mathcal{A}$ is *low* for $\mathcal{B}$ if $\mathcal{B} = \mathcal{B}^{\mathcal{A}}$. That is, access to an $\mathcal{A}$-oracle does not increase the power of $\mathcal{B}$-Turing Machines.

(Recommended) Problem 27. Recall that REC is the set of decidable languages, and RE is the set of recursively enumerable languages. Do the following.

(a) Show that REC is low for itself. That is, show that $\mathsf{REC} = \mathsf{REC}^{\mathsf{REC}}$.

(b) Show that REC is low for RE. That is, show that $\mathsf{RE} = \mathsf{RE}^{\mathsf{REC}}$.

(Recommended) Problem 28. Let L_1 and L_2 be languages. Suppose that L_1 is many-to-one (Karp) reducible to L_2; that is, $L_1 \leq_m L_2$. Show that L_1 is Turing reducible to L_2; that is, $L_1 \leq_T L_2$.

[4] A language L is in BPP if there exists a randomized Turing Machine M such that for all $x \in L$, $M(x) = 1$ with probability at least $2/3$; and for all $x \notin L$, $M(x) = 1$ with probability at most $1/3$. Here, BPP stands for *bounded-error probabilistic polynomial time*.

2.5 Arithmetic Hierarchy

In this section, we introduce the Arithmetic Hierarchy. A key idea behind Oracle Turing Machines is as follows: given the ability to solve a specific problem, what additional problems can we solve? It is natural to ask as to the problems we could solve, under the assumption that the Halting problem was solvable. Similarly, are there problems that cannot be solved, even in the presence of an Oracle to the Halting problem? These questions motivate the Arithmetic Hierarchy.

We begin by denoting $\Sigma_1^0 = \mathsf{RE}$ and $\Pi_1^0 = \mathsf{co}\Sigma_1^0 = \mathsf{coRE}$. Denote $\Delta_1^0 := \Sigma_1^0 \cap \Pi_1^0$. So $\Delta_1^0 = \mathsf{REC}$. We now define the Arithmetic Hierarchy.

Definition 81. For $i > 1$, denote $\Sigma_i^0 = \mathsf{RE}^{\Sigma_{i-1}^0}$ and $\Pi_i^0 = \mathsf{co}\Sigma_i^0$. Denote $\Delta_i^0 = \Sigma_i^0 \cap \Pi_i^0$.

Remark 82. We note that Π_i^0 also has a characterization in terms of Oracles. Namely, $\Pi_i^0 = \mathsf{coRE}^{\Pi_{i-1}^0}$. We will not formally prove this equivalence.

Definition 83 (Arithmetic Hierarchy)**.** The *Arithmetic Hierarchy*, denoted AH is:

$$\mathsf{AH} = \bigcup_{n \in \mathbb{Z}^+} \Sigma_i^0.$$

We now ask whether all the Σ_i^0, Π_i^0, and Δ_i^0 classes are distinct. It turns out that $\Sigma_i^0 \neq \Pi_i^0$ (see Problem 29). We next introduce the following theorem, which establishes clear containment relations amongst the Σ_i^0 and Π_i^0 classes. Hence, we have a better handle on the hierarchy.

Theorem 84. For each $i \geq 1$, we have that $\Sigma_i^0 \subseteq \Delta_{i+1}^0$ and $\Pi_i^0 \subseteq \Delta_{i+1}^0$.

We leave the proof as an exercise. It follows from Theorem 84 and Problem 30 that for all $i \geq 1$, $\Sigma_i^0 \subsetneq \Sigma_{i+1}^0$ and $\Pi_i^0 \subsetneq \Pi_{i+1}^0$. So AH does not collapse to any level i.

2.5.1 Exercises

(Recommended) Problem 29. Let $\mathcal{O}$ be an arbitrary oracle. Do the following.

(a) Show that $\mathsf{REC}^{\mathcal{O}}$ is closed under complements.

(b) Show that $\mathsf{REC}^{\mathcal{O}} = \mathsf{RE}^{\mathcal{O}} \cap \mathsf{coRE}^{\mathcal{O}}$.

(c) Exhibit a language $L \in \mathsf{RE}^{\mathcal{O}}$ such that $\overline{L} \notin \mathsf{RE}^{\mathcal{O}}$. Note that as $L \in \mathsf{RE}^{\mathcal{O}}$, $\overline{L} \in \mathsf{coRE}^{\mathcal{O}}$. [**Hint:** Mimic the proof techniques in the Undecidability section of the Computability notes.]

(d) In light of part (c), conclude that there exist problems that do not belong to $\mathsf{REC}^{\mathcal{O}}$. That is, for every oracle $\mathcal{O}$, there exist undecidable problems relative to $\mathcal{O}$.

Remark 85. Observe that the proofs in the non-relativized setting (i.e., when dealing with Turing Machines that do not have oracles) went through immediately and were not sensitive to the presence of oracles. This is a phenomenon in Computability Theory that does not happen in Computational Complexity. The P vs. NP problem, for instance, is sensitive to the presence of oracles. As a result, Computability techniques are insufficient to resolve the P vs. NP problem. This barrier, known as the *Relativization Barrier*, is a celebrated result from 1975 due to Baker, Gill, and Solovay.

(Recommended) Problem 30. Fix $i \geq 1$. Show the following.

(a) $\Sigma_i^0 \subseteq \Delta_{i+1}^0$.

(b) $\Pi_i^0 \subseteq \Delta_{i+1}^0$.

3 Structural Complexity

Computational Complexity takes a different perspective than the study of Algorithms. We seek to classify problems into classes according to whether they can be solved within specified resource bounds. Structural Complexity Theory seeks to examine the relationships between these complexity classes. We assume familiarity with the classes P and NP, as well as NP-completeness.

3.1 Ladner's Theorem

One key approach to resolving the P vs. NP problem is to find a problem $L \in \mathsf{NP}$ such that $L \notin \mathsf{P}$ and L is not NP-complete. We refer to such languages as NP-intermediate. Showing that there exists an NP-intermediate language would show that $\mathsf{P} \neq \mathsf{NP}$. We now consider the converse. There are three possibilities to resolve the P vs. NP problem. First, suppose that $\mathsf{P} = \mathsf{NP}$. Then there are no NP-intermediate languages. Suppose instead that $\mathsf{P} \neq \mathsf{NP}$. There are two cases:

(a) Every language $L \in \mathsf{NP}$ either belongs to P or is NP-complete.

(b) There exists an NP-intermediate language L.

Ladner's Theorem establishes that if $\mathsf{P} \neq \mathsf{NP}$, then there is an NP-intermediate language L [Lad75]. In light of Ladner's result, there has been a great deal of effort to find NP-intermediate problems. There have been numerous candidates since 1975. Some of these candidates have been shown to be in P. For others, their complexity status remains open. We provide some examples.

(a) The Integer Factorization and Discrete Logarithm problems are conjectured to be NP-intermediate. To date, the best known algorithms for these problems rely on heavy-handed machinery from Algebraic Number Theory, such as Number Field Sieves.

(b) Isomorphism problem such as Cayley Group Isomorphism and Graph Isomorphism are conjectured to be NP-intermediate. The trivial bound for Cayley Group Isomorphism is $n^{\log_p(n)+O(1)}$, where p is the smallest prime dividing n. This is obtained via a generator-enumeration technique, which Miller attributes to Tarjan [Mil78]. The best known algorithm for Cayley Group Isomorphism is $n^{(1/4)\log_p(n)+O(1)}$ due to Rosenbaum [Ros13] (see [LGR16, Sec. 2.2]). Even the impressive body of work on practical algorithms for this problem, led by Eick, Holt, Leedham-Green and O'Brien (e. g., [BEO02, ELGO02, BE99, CH03]) still results in an $n^{\Theta(\log n)}$-time algorithm in the general case [Wil19].

For Graph Isomorphism, the best known algorithmic upper bound is $n^{\Theta(\log^2(n))}$ due to Babai [Bab15]. Cayley Group Isomorphism is many-to-one polynomial-time reducible to Graph Isomorphism, and so the Cayley Group Isomorphism problem remains a key bottleneck to determining whether Graph Isomorphism belongs to P.

(c) Primality testing was a long-standing candidate to be NP-intermediate. In 2002, Agrawal, Kayal, and Saxena showed that Primality $\in$ P [AKS02]. It is noteworthy that Kayal and Saxena were both undergraduate students working with Agrawal. The techniques in their paper are largely accessible after a semester of Abstract Algebra at the undergraduate level. We also note that the implicit constant associated with the AKS procedure is quite large, which makes the procedure impractical. In practice, primality testing is still handled using probabilistic algorithms.

(d) Linear Programming was long-conjectured to be NP-intermediate. The Simplex algorithm was the long-standing algorithmic tool in this area. Despite the fact that the Simplex algorithm performed well in practice, it still had a worst-case exponential runtime. In 1979, Leonid Khachiyan introduced the Ellipsoid algorithm, which was the first polynomial-time algorihtm for Linear Programming [Kha80]. Thus, we have that Linear Programming $\in$ P. It is worth noting that the implicit constant associated with the Ellipsoid algorithm is quite large. So in practice, the Simplex algorithm is still widely used today.

We now turn our attention to proving Ladner's Theorem. The weak version of Ladner's Theorem provides only one intermediate language, if $\mathsf{P} \neq \mathsf{NP}$.

Theorem 86 (Ladner (weak), 1975)**.** Suppose that $\mathsf{P} \neq \mathsf{NP}$. Then there exists a language $L \in \mathsf{NP}$ such that $L \notin \mathsf{P}$ and L is not NP-complete.

Ladner proved something much stronger. Namely, if $\mathsf{P} \neq \mathsf{NP}$, then there is a strict infinite hierarchy of NP-intermediate languages. It is widely believed that $\mathsf{P} \neq \mathsf{NP}$. However, finding even one NP-intermediate language remains an open problem.

Theorem 87 (Ladner (strong), 1975)**.** Suppose that $\mathsf{P} \neq \mathsf{NP}$. Then there exists a sequence of languages $(L_i)_{i \in \mathbb{N}}$ satisfying the following.

(a) $L_{i+1} \subsetneq L_i$ for all $i \in \mathbb{N}$.

(b) $L_i \notin \mathsf{P}$ for all $i \in \mathbb{N}$.

(c) L_i is not NP-complete for all $i \in \mathbb{N}$.

(d) $L_i \not\leq_p L_{i+1}$ for all $i \in \mathbb{N}$.

In order to prove Theorems 86 and 87, we prove a more general theorem first.

Theorem 88. Suppose that $L \notin \mathsf{P}$ is computable. Then there exists a language $K \notin \mathsf{P}$ such that $K \leq_p L$ and $L \not\leq_p K$.

The key technique in proving Theorem 88 is to diagonalize against both polynomial-time Turing Machines to ensure that $K \notin \mathsf{P}$ and polynomial-time reductions from L to K to ensure that $L \not\leq_p K$. We alternate between diagonalizing against polynomial-time Turing Machines at one stage, and then against polynomial-time reductions in the next stage.

Proof of Theorem 88. We construct the language K via a diagonalization argument. Namely, define:

$$K := \{x \in L : f(|x|) \text{ is even}\},$$

where $f(x)$ is a function we will construct later. Let $(M_i)_{i \in \mathbb{Z}^+}$ be an enumeration of polynomial-time deterministic Turing Machines, which enumerates P. Let $(F_i)_{i \in \mathbb{Z}^+}$ be an enumeration of polynomial-time Turing Machines without restriction to their output lengths. We note that $(F_i)_{i \in \mathbb{Z}^+}$ includes polynomial-time many-to-one reductions from L to K.

Let M_L be a decider for L. We now define f recursively as follows. First, define $f(0) = f(1) = 2$. We associate f with the Turing Machine M_f that computes f. On input 1^n (with $n > 1$), M_f proceeds in two stages, each lasting exactly n steps.[5] At the first stage, M_f computes $f(0), f(1), \ldots$, until it runs out of time. Suppose that the last vaue M_f computed at the first stage was $f(x) = k$. We will either have that $f(n) = k$ or $f(n) = k+1$, depending on the second stage.

At the second stage, we have one of two cases.

- **Case 1:** Suppose that $k = 2i$. Here, we diagonalize against the ith language $L(M_i)$ in P as follows. The goal is to find a string z such that $z \in (L(M_i) \triangle K)$. We enumerate such strings z in lexicographic order, and then compute $M_i(z), M_L(z)$[6], and $f(|z|)$ for all such strings. Note that by definition of K, we must compute $f(|z|)$ to ensure that $f(|z|)$ is even. If such a string is found in the allotted time (n steps), then M_f outputs $k+1$ (so M_f can proceed to diagonalize against polynomial-time reductions on the next iteration). Otherwise, M_f outputs k (as we have not successfully diagonalized against M_i yet).
- **Case 2:** Suppose that $k = 2i - 1$. Here, we diagonalize against the ith polynomial-time computable function F_i. In this case, M_f searches for a string z such that F_i is not a valid Karp reduction on z. That is, either:
  - $z \in L$, but $F_i(z) \notin K$; or
  - $z \notin L$, but $F_i(z) \in K$.

[5] In order to prove Theorem 88, it is not necessary to limit the number of steps M_f can take to be polynomial in n. However, as $L \notin \mathsf{P}$, M_L does not run in polynomial-time. So in order to apply Theorem 88 to prove Theorems 86 and 87, we need to build in the clocking condition into the proof here.

[6] By limiting M_f to n steps at the second stage, we ensure the (partial) computation of M_L does not run for so long that it prohibits K from belonging to NP.

We accomplish this by computing $F_i(z), M_L(z), M_L(F_i(z))$, and $f(|F_i(z)|)$. Here, we use clocking to ensure that M_L is not taking too long. If such a string z is found in the allotted time, then the output of M_f is $k+1$. Otherwise, M_f outputs k.

We now show that K satisfies the following conditions.

(a) $K \leq_p L$,

(b) $K \notin \mathsf{P}$, and

(c) $L \not\leq_p K$.

Proposition 89. $K \leq_p L$.

Proof. The inclusion map $\iota : K \to L$ sending $\iota(\omega) = \omega$ is a polynomial-time computable reduction. □

Proposition 90. $K \notin \mathsf{P}$.

Proof. Suppose to the contrary that $K \in \mathsf{P}$. Then there exists a polynomial time deterministic Turing Machine M_i such that $L(M_i) = K$. By Case 1 of the second stage in the construction on M_f, no string z was found satisfying $z \in K$ and $z \notin L(M_i)$ (or vice-versa). So $f(n)$ is even for all but finitely many n. So K and L coincide for all but finitely many strings. As $L \setminus K$ is finite, $L \setminus K$ can be decided in polynomial time. Together with the fact that $K \in \mathsf{P}$, we have that $L = K \cup (L \setminus K) \in \mathsf{P}$, contradicting the assumption that $L \notin \mathsf{P}$. □

Proposition 91. $L \not\leq_p K$.

Proof. Exercise. □

□

3.1.1 Exercises

(Recommended) Problem 31. Prove Proposition 91. [**Hint:** Argue by contradiction. Suppose that we have a polynomial-time computable reduction F_i that takes L to K. Argue that $f(n)$ will be even for only finitely many n.]

(Recommended) Problem 32. Using Theorem 88, prove Theorem 86.

(Recommended) Problem 33. Using Theorem 88, prove Theorem 87.

3.2 Introduction to Space Complexity

In this section, we introduce notions of space complexity. Our goal is to develop familiarity with the space complexity classes $\mathsf{PSPACE}, \mathsf{L}$, and NL, to the extent necessary to develop the theory of structural complexity. These complexity classes have a rich theory in and of themselves. We defer to [Sip96, AB09] for more comprehensive treatments.

In order to discuss space complexity, we consider a class of multitape Turing Machines, which we refer to as *space-bounded Turing Machines*. Precisely, we consider Turing Machines that use three tapes. These tapes are as follows:

(a) The input tape, for which we may only read input.

(b) The work tape, for which we have both read and write access.

(c) The output tape, for which we have both read and write access.

Ony non-blank cells on the work and output count towards the amount of space used. We now define the DSPACE and NSPACE complexity classes.

Definition 92. Let $f : \mathbb{N} \to \mathbb{N}$. We say that the language $L \in \mathsf{DSPACE}(f(n))$ if there exists a deterministic space-bounded Turing Machine M such that M decides L and M uses at most $O(f(n))$ space. Similarly, we say that $L \in \mathsf{NSPACE}(f(n))$ if there exists a non-deterministic space-bounded Turing Machine M such that M decides L and M uses at most $O(f(n))$ space.

Clearly, $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$. We next relate our space and time complexity classes.

Lemma 93. We have that $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n))$.

Proof. We note that a deterministic Turing Machine that takes k steps writes to at most k cells. The result follows. □

We now show that $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$.

Theorem 94. We have that $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))})$.

Proof. The idea is to count Turing Machine configurations. Each Turing Machine state transition costs a single computation step and takes us from one configuration to another. Furthermore, each configuration is visited at most once; otherwise the Turing Machine enters an infinite loop[7]. So counting the number of Turing Machine configurations provides an upper bound on the runtime.

Let $L \in \mathsf{NSPACE}(f(n))$, and suppose that we have a non-deterministic space-bounded Turing Machine M that accepts L. Without loss of generality, suppose that M uses exactly $f(n)$ tape cells. Each configuration consists of the following:

(a) A state from $Q(M)$.

(b) The positions of the input, work, and output tape heads. There are n possible positions for the input tape head, at most $f(n)$ possible positions for the work tape head, and at most $f(n)$ possibe positions for the output tape head. The selections of the positions for each tape head are independent. So by the rule of product, there are at most $n \cdot (f(n))^2$ possible selections for the tape head positions.

(c) There are at most $f(n)$ non-blank cells between the work and output tapes. So there are at most $|\Gamma|^{f(n)}$ possible ways to fill the non-blank cells.

[7]Implicitly, we are describing a directed acyclic graph structure, which is commonly referred to as a *configuration graph*. We are not leveraging the graph structure in this proof, so we omit those details. However, the configuration graph is a useful tool in space complexity, such as in establishing a natural NL-complete problem.

Note that selecting the state, positions for the tape heads, and ways to fill the non-blank cells are independent selections. So by the rule of product, we have at most:

$$|Q(M)| \cdot n \cdot (f(n))^2 \cdot |\Gamma|^{f(n)} \leq |\Gamma|^{O(f(n))}$$

possible configurations. Now note that $k = 2^{\log_2(k)}$. So if $k := |\Gamma| > 2$, we have that:

$$\begin{aligned} |\Gamma|^{O(f(n))} &= (2^{\log_2(k)})^{O(f(n))} \\ &= 2^{\log_2(k) \cdot O(f(n))} \\ &= 2^{O(f(n))}. \end{aligned}$$

The result follows. □

We conclude by introducing some standard complexity classes and complete problems.

3.2.1 PSPACE

Definition 95. The complexity class PSPACE is defined to be:

$$\mathsf{PSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{DSPACE}(n^k).$$

PSPACE also has complete problems under polynomial-time reductions. We introduce one such canonical problem here- TQBF, though we won't prove that $\mathsf{TQBF} \in \mathsf{PSPACE}$.

Definition 96. Let $\varphi(x_1, \ldots, x_n)$ be a Boolean formula. We consider *fully quantified Boolean formulas*, which are of the form:

$$\Phi := Q_1 x_1,\, Q_2 x_2,\, \ldots, Q_n x_n\, \varphi(x_1, \ldots, x_n).$$

Here, each quantifier Q_i is either an existential quantifier ($\exists$) or a universal quantifier ($\forall$). We note that a fully quantified Boolean formula is either always true or always false.

Definition 97 (TQBF)**.** The True Quantified Boolean Formulas (TQBF) problem is defined as follows:

- Instance: A fully quantified Boolean formula Φ.
- Decision: Is Φ true?

Example 98. Let:

$$\Phi := \forall x\, \exists y\, [(x \vee y) \wedge (\overline{x} \vee \overline{y})].$$

Observe that Φ is true: take $y = \overline{y}$. On the other hand,

$$\Psi := \exists x\, \forall y\, [(x \vee y) \wedge (\overline{x} \vee \overline{y})]$$

is false. If $x = y$, then either $(x \vee y)$ is true or $(\overline{x} \vee \overline{y})$ is true (but not both).

Theorem 99. We have that TQBF is PSPACE-complete.

Remark 100. We note that TQBF remains PSPACE-complete if we insist that the Boolean formula is in conjunctive normal form, where each clause has exactly 3 literals (that is, the Boolean formula is in $3 - \mathsf{CNF}$). We will use this fact later when discussing Interactive Proofs.

3.2.2 L and NL

Definition 101. The complexity class $\mathsf{L} := \mathsf{DSPACE}(\log(n))$. Similarly, $\mathsf{NL} := \mathsf{NSPACE}(\log(n))$.

We note that NL has complete problems, under logspace reductions. We denote the relation that L_1 is logspace many-to-one reducible to L_2 as $L_1 \leq_\ell L_2$. Informally, complete problems capture the essence of a given complexity class. For this reason, it is natural to insist that reductions are computable within said complexity class. Hence, we consider complete problems for NL under logspace reductions.

Remark 102. We note that every non-trivial problem in L is L-complete under logspace reductions. For this reason, complete problems are often considered under more restrictive notions of reducibility, such as AC^0-reducibility. We won't pursue this direction any further.

The main result we consider in this section is in establishing an NL-complete problem. Namely, the Connectivity problem.

Definition 103. The Connectivity problem is defined as follows.

- Instance: A graph $G(V, E)$, and vertices u and v.
- Decision: Does there exist a $u \to v$ path in G?

Theorem 104. The Connectivity problem is NL-complete.

Our proof is adopted from [Kat11].

Proposition 105. We have that Connectivity $\in$ NL.

Proof. Let $G(V, E)$ be our graph on n vertices, and let $u, v \in V(G)$ be the specified start and end vertices. We exhibit a non-deterministic algorithm that always rejects if no such $u \to v$ path exists and sometimes accepts if there exists a $u \to v$ path. The idea is to non-deterministically guess the vertices in the path one at a time. The algorithm proceeds as follows.

1. If $u = v$, we terminate.
2. Set $v_{\text{current}} := u$.
3. For $i = 0$ to n:
   (a) Guess a vertex w.
   (b) If w is not neighbor of v_{current}, then reject.
   (c) If $w = t$, accept.
   (d) Otherwise, set $v_{\text{current}} = w$.
4. If we have not decided after the loop terminates, then we reject, as any path has length at most n.

We note that the vertices are specified by binary strings of length $\lceil \log_2(n) \rceil$. At any given iteration, we store v_{current} and our guess w. So only $2\lceil \log_2(n) \rceil \in O(\log(n))$ bits of memory are being stored. Thus, Connectivity $\in$ NL. □

We now show that Connectivity is NL-hard. The idea is to take an NL-TM and map it to its corresponding configuration graph. We then ask whether there is a path from the initial configuration to a specified accepting configuration.

Proposition 106. We have that Connectivity is NL-hard.

Proof. Let $L \in \mathsf{NL}$, and let M be an NL-TM that accepts L. Fix a string $\omega \in \Sigma^*$ of length n. We note that as M, when run on ω, uses space $c \cdot \log(n)$, then M has $2^{c \cdot \log(n)} = n^c$ possible configurations. We construct a configuration graph $G(V, E)$, where the configurations are the vertices. Two configurations C_1 and C_2 are adjacent if there is a state transition of M that takes us from C_1 to C_2. Let C_0 be the initial configuration. By construction, we have that $\omega \in L$ if and only if there exists an accepting configuration C^* that is reachable from C_0.

We note that we don't construct the configuration graph outright, as this would require at most $O(n^{2c})$ cells in memory to store the graph. Instead, we note that at any step, we only compute the configurations corresponding to the initial and final states, as well as the given configuration we are considering at any iteration when we are guessing the path. As there are n^c configurations, only $c \cdot \lceil \log_2(n) \rceil$ bits are required to store a given configuration. We are storing $3c \cdot \lceil \log_2(n) \rceil \in O(\log(n))$ bits in memory at a given stage. So our reduction is logspace computable, as desired. □

3.2.3 Exercises

(Recommended) Problem 34. Theorem 94 yields some immediate corollaries.

(a) Show that a logspace computation can be simulated in polynomial time. That is, using Theorem 94 show that: $\mathsf{NL} \subseteq \mathsf{P}$. [**Note:** As a result, we obtain that logspace-computable many-to-one reductions are stronger than polynomial-time computable many-to-one reductions. That is, $L_1 \leq_\ell L_2 \implies L_1 \leq_p L_2$.]

(b) Define:

$$\mathsf{EXPTIME} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}(2^{O(n^k)}).$$

Show that $\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$.

(Recommended) Problem 35. A *finite state automaton* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is our finite set of states,
- Σ is our alphabet,
- $\delta : Q \times \Sigma \to Q$ is our transition function,
- q_0 is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

A finite state automaton parses the input string $\omega = (\omega_1, \ldots, \omega_n)$ one character at a time in order, starting from ω_1. No characters are revisited, and the finite state automaton does not have a work tape.The finite state automaton writes either 0 or 1 to the output tape, depending on whether it accepts the given string. Do the following.

(a) Design a finite state automaton to accept the language $L \subseteq \{0,1\}^*$, where each string in L has an even number of 1's.

(b) The set of languages accepted by finite state automata are precisely the regular languages, denoted REG. Show that $\mathsf{REG} = \mathsf{DSPACE}(O(1))$. [**Hint:** To show that $\mathsf{DSPACE}(O(1)) \subseteq \mathsf{REG}$, show that the configuration graph for a $\mathsf{DSPACE}(O(1))$-TM can be viewed as a deterministic finite state automaton. You may assume, without loss of generality, the TMs may have multiple accept states.]

(c) A *non-deterministic finite state automaton* is defined similarly as a deterministic finite state automaton $(Q, \Sigma, \delta, q_0, F)$, except where the transition function is of the form:

$$\delta : Q \times \Sigma \to 2^Q.$$

Kleene's Theorem provides that L is accepted by some non-deterministic finite state automaton if and only if $L \in \mathsf{REG}$.[8] Show that $\mathsf{REG} = \mathsf{NSPACE}(O(1))$.

(d) Conclude that $\mathsf{DSPACE}(O(1)) = \mathsf{NSPACE}(O(1))$.

[8] More precisely, Kleene's Theorem states that $L \in \mathsf{REG}$ if and only if L is accepted by some deterministic finite state automaton. The need to consider non-deterministic variants comes out in the proof, where it is subsequently shown that non-deterministic finite state automata can be converted to equivalent deterministic finite state automata. We defer to [Lev20] for a more thorough treatment of regular languages.

(Recommended) Problem 36. Define:

$$\mathsf{NPSPACE} := \bigcup_{k \in \mathbb{N}} \mathsf{NSPACE}(n^k).$$

Show that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. [**Hint:** Suppose that an $\mathsf{NPSPACE}$ machine uses space $O(n^k)$. Then there are $2^{O(n^k)}$ possible configurations. How much space does it cost to store a single configuration? What about to traverse the configuration graph?]

(Recommended) Problem 37. While we have shown that $\mathsf{P} \subseteq \mathsf{PSPACE}$ (see Lemma 93), we have not shown that $\mathsf{NP} \subseteq \mathsf{PSPACE}$. To do so, show that $\mathsf{SAT} \in \mathsf{PSPACE}$. [**Hint:** Try a brute force approach to solve SAT.]

(Recommended) Problem 38. Show that $\mathsf{TQBF} \in \mathsf{PSPACE}$. [**Hint:** Apply the same technique as in Problem 37.]

3.3 Baker-Gill-Solovay Theorem

Computational Complexity began as a generalization of Computability Theory. The key idea is to ask what problems can and cannot be solved under given resource constraints. Many of the core techniques in Structural Complexity Theory, such as diagonalization and simulation, arise from Computability Theory. As a result, it is very natural to ask whether Computabiltiy techniques are sufficient to resolve the P vs. NP problem. In 1975, Baker, Gill, and Solovay showed that Computability techniques are insufficient to resolve the P vs. NP problem. The key observation is that Computability techniques are not sensitive to the presence of oracles. However, there are oracles A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, but $\mathsf{P}^B \neq \mathsf{NP}^B$.

Theorem 107 (Baker-Gill-Solovay, 1975.)**.** There exist oracles A and B such that $\mathsf{P}^A = \mathsf{NP}^A$, but $\mathsf{P}^B \neq \mathsf{NP}^B$.

We begin with the oracle A. It suffices to take A to be a PSPACE-complete problem.

Proposition 108. Let A be PSPACE-complete. Then $\mathsf{P}^A = \mathsf{NP}^A$.

We note that $\mathsf{P}^A \subseteq \mathsf{NP}^A$ for the same reasons that $\mathsf{P} \subseteq \mathsf{NP}$. The proof that $\mathsf{P} \subseteq \mathsf{NP}$ is not sensitive to the presence of oracles. To show that $\mathsf{NP}^A \subseteq \mathsf{P}^A$, it suffices to show that the following chain of inclusions holds:

$$\mathsf{NP}^A \subseteq \mathsf{PSPACE} \subseteq \mathsf{P}^A.$$

We leave it as an exercise to establish this chain of inclusions. See Problem 39.

We now turn towards constructing an oracle B, for which $\mathsf{P}^B \neq \mathsf{NP}^B$.

Proposition 109. There exists an oracle B such that $\mathsf{P}^B \neq \mathsf{NP}^B$.

Proof. We construct an oracle B inductively; such that for each $i \in \mathbb{N}$, B contains at most one word of length i. Precisely, we construct a sequence of oracles $(B_i)_{i\in\mathbb{N}}$ such that:

$$B := \bigcup_{i\in\mathbb{N}} B_i.$$

Our strategy is then to show that the following language:

$$L := \{0^i : B \text{ has a string of length } i\}$$

is in $\mathsf{NP}^B \setminus \mathsf{P}^B$. In the process of constructing B, we also maintain a list of forbidden words. Denote $\mathcal{F}_i$ to be the set of forbidden words encountered after constructing B_i.

We proceed via diagonalization, constructing oracles $(B_i)_{i\in\mathbb{N}}$. Let $(M_\ell)_{\ell\in\mathbb{N}}$ be an enumeration of Oracle Turing Machines. When we consider M_ℓ, we will equip M_ℓ with the oracle B_ℓ (that is, we consider $M_\ell^{B_\ell}$).[9] Define $B_0 := \emptyset$ and $\mathcal{F}_0 := \emptyset$. Fix $i \geq 0$, and suppose that we have constructed B_i, where B_i has at most one word of length j for each $0 \leq j < i$, and no words of length at least i. Similarly, suppose that we have constructed the set $\mathcal{F}_i$ of forbidden words. We simulate $M_i^{B_i}$ on 0^i for $i^{\log i}$ steps. If $M_i^{B_i}$ queries a word y of length at least i, we assume that $y \notin B$ and add y to the set of forbidden words. Let:

$$\mathcal{F}_{i+1} := \mathcal{F}_i \cup \{y \in \Sigma^* : |y| \geq i \text{ and } M_i^{B_i} \text{ queried } y\}.$$

We now turn our attention to constructing B_{i+1}. We have the following cases:

- **Case 1:** Suppose that in the first $i^{\log i}$ steps that $M_i^{B_i}$ rejects 0^i. If there is a word ω of length i that does not belong to $\mathcal{F}_i$, then set $B_{i+1} := B_i \cup \{\omega\}$. Such a word ω may be selected deterministically, such as by selecting the first non-forbidden word in lexicographic order of Σ^i. If no such word ω exists, then $B_{i+1} := B_i$.
- **Case 2:** Suppose that $M_i^{B_i}$ does not reject 0^i in the first $i^{\log i}$ steps. Then no word of length i is placed in B.

[9] Recall that an M_ℓ is defined by its transition function. The oracle is a parameter, and not part of the definition of M_ℓ.

We also do not include a word of length i in B_{i+1} if all words of length i are forbidden. Note that for sufficiently large i, not all words of length i will be forbidden. To see this, observe that the maximum number of forbidden words of length at most i is:

$$\sum_{j=1}^{i} j^{\log j} \leq i(i^{\log i}) = i^{1+\log i}.$$

Now not all words of length i are forbidden if $|\Sigma|^i \geq 2^i > i^{1+\log i}$. Note that $2^i > i^{1+\log i}$ holds precisely if $i > (\log i)(1+\log i)$. This last inequality holds for $i \geq 32$. In particular, it follows that B is an infinite set. We again note that:

$$B := \bigcup_{i \in \mathbb{N}} B_i.$$

and

$$L := \{0^i : B \text{ has a word of length } i\}.$$

We claim that $L \in \mathsf{NP}^B \setminus \mathsf{P}^B$.

Lemma 110. We have that $L \in \mathsf{NP}^B$.

Proof. Exercise. □

Lemma 111. We have that $L \notin \mathsf{P}^B$.

Proof. Suppose to the contrary that $L \in \mathsf{P}^B$. So there exists an Oracle Turing Machine M_k such that $L = L(M_k^B)$. Let $p(n)$ be the runtime complexity function for M_k^B. As B is infinite, so is L. Thus, M_k accepts arbitrarily long strings. So we may- without loss of generality- assume that $k \geq 32$ and $2^k \geq p(k)$. We now analyze whether $0^k \in L$. We have the following cases.

- **Case 1:** Suppose that M_k^B accepts 0^k. So $0^k \in L$, which implies that B has a string ω of length k. By construction of B, it follows that $M_k^{B_k}$ rejects 0^k. However, as $B_k \subseteq B$ and B has no words of length at least k that are queried by $M_k^{B_k}$ on 0^k, it follows that $M_k^{B_k}$ and M_k^B behave identically on 0^k. So M_k^B rejects 0^k, a contradiction.
- **Case 2:** Suppose that M_k^B rejects 0^k. We leave it as an exercise to derive a contradiction by showing that this implies that $0^k \in L$. See Problem 41.

□

□

3.3.1 Exercises

(Recommended) Problem 39. Let A be PSPACE-complete. Do the following.

(a) Show that $\mathsf{NP}^A \subseteq \mathsf{PSPACE}$.

(b) Show that $\mathsf{PSPACE} \subseteq \mathsf{P}^A$.

(c) Conclude that $\mathsf{P}^A = \mathsf{NP}^A$.

(Recommended) Problem 40. Prove Lemma 110.

(Recommended) Problem 41. Complete the proof of Lemma 111. Suppose that M_k^B rejects 0^k. Derive a contradiction by showing that this implies that $0^k \in L$.

3.4 Polynomial-Time Hierarchy: Introduction

We recall the definitions of NP and coNP.

Definition 112 (NP)**.** We say that a language $L \in \mathsf{NP}$ if there exists a polynomial $p(\cdot)$ depending only on L and a verifier V such that if $x \in L$, then there exists a string y of length at most $p(|x|)$, such that $V(x, y) = 1$ and V runs in time $O(p(|x|))$. Here, y is the *certificate*.

Remark 113. We may write the definition of NP in quantifier notation:

$$x \in L \iff \exists y \text{ s.t. } |y| \leq p(|x|), V(x, y) = 1. \tag{11}$$

This expression is often abbreviated as follows, though (11) is the formalism and intended meaning.

$$x \in L \iff \exists y, V(x, y) = 1.$$

Similarly, coNP is defined as follows.

Definition 114 (coNP)**.** We say that a language $L \in \mathsf{coNP}$ if there exists a polynomial $p(\cdot)$ depending only on L and a verifier V such that if $x \in L$, then for every y of length at most $p(|x|)$, that $V(x, y) = 0$ and V runs in time $O(p(|x|))$.

Remark 115. We may write the definition of coNP in quantifier notation:

$$x \in L \iff \forall y \text{ s.t. } |y| \leq p(|x|), V(x, y) = 0. \tag{12}$$

This expression is often abbreviated as follows, though (12) is the formalism and intended meaning.

$$x \in L \iff \forall y, V(x, y) = 0.$$

Our goal now is to generalize NP and coNP. We begin with some motivation. Recall the Independent Set decision problem, which takes as input a graph $G(V, E)$ and integer k, and asks if G has an independent set of size k. Recall that Independent Set is NP-Complete. In particular, Independent Set $\in$ NP.

Now consider the Maximum Independent Set problem, which again takes as input a graph $G(V, E)$ and integer k. Here, we ask whether the largest size independent set in G has k vertices. Here, we need to verify a couple conditions:

- G has an independent set of k vertices. This is precisely the condition that Independent Set $\in$ NP.
- G does not have an independent set of $k + 1$ vertices. We note that verifying this second condition is a coNP problem.

So effectively, $(G, k) \in$ Maximum Independent Set $\iff$ there exists a small certificate of one type and no small certificate of another type. This motivates the definition of a new complexity class, which we will call Σ_2^p.

Definition 116. We say that a language $L \in \Sigma_2^p$ (pronounced Sigma-2) if there exists a polynomial $p(\cdot)$ depending only on L and a verifier V such that if $\omega \in L$, then there exists a string x of length at most $p(|\omega|)$, such that for all strings y of length at most $p(|\omega|)$, $V(\omega, x, y) = 1$ and V runs in time $O(p(|\omega|))$.

We may again express the definition of Σ_2^p in quantifier notation.

$$\omega \in L \iff \exists x \text{ s.t. } |x| \leq p(|\omega|), \forall y \text{ s.t. } |y| \leq p(|\omega|), M(\omega, x, y) = 1.$$

As we saw with NP and coNP, the quantified expression for Σ_2^p is commonly abbreviated as follows.

$$\omega \in L \iff \exists x, \forall y, M(\omega, x, y) = 1.$$

Example 117. We note that Maximum Independent Set $\in \Sigma_2^p$. Here, x is the certificate for the independent set of size k, and y is a vertex set of size $k+1$. In other words, M checks that x is an independent set of size k and that y is a $(k+1)$-size vertex set is not an independent set. Note that M itself does not consider all such $(k+1)$-size vertex sets at once. Rather, the quantifier states that for any given $(k+1)$-size vertex set y, that M will check that y is not an independent set.

3.4.1 Σ_i^p

We now turn to generalizing Σ_2^p. Here, the subscript 2 indicates that we use two quantifiers. We define Σ_i^p to use i quantifiers, starting with $\exists$, and then alternating between $\exists$ and $\forall$. This is formalized as follows.

Definition 118. We say that the language $L \in \Sigma_i^p$ if there exists a polynomial $p(\cdot)$ depending only on L and a verifier V such that:

$$\omega \in L \iff \exists x_1, \forall x_2, \exists x_3, \forall x_4, \ldots, Q_i x_i, V(\omega, x_1, \ldots, x_i) = 1,$$

$|x_j| \leq p(|\omega|)$ for all $1 \leq j \leq i$, and V runs in time $O(p(|\omega|))$. Note that Q_i indicates a quantifier. In particular, Q_i is an existential quantifier if i is odd and a universal quantifier if i is even.

So for Σ_3^p, the abbreviated quantified expression is:

$$\omega \in L \iff \exists x_1, \forall x_2, \exists x_3, V(\omega, x_1, x_2, x_3) = 1.$$

Similarly, for Σ_4^p, the abbreviated quantified expression is:

$$\omega \in L \iff \exists x_1, \forall x_2, \exists x_3, \forall x_4, V(\omega, x_1, x_2, x_3, x_4) = 1.$$

Remark 119. It is also worth noting that $\mathsf{NP} = \Sigma_1^p$.

Theorem 120. Fix $i \geq 1$. We have that $\Sigma_i^p \subseteq \Sigma_{i+1}^p$.

Proof. Let $L \in \Sigma_i^p$. Let $M(\omega; x_1, \ldots, x_i)$ be the Σ_i^p Turing Machine that accepts L. We construct a Σ_{i+1}^p Turing Machine $N(\omega; x_1, \ldots, x_i, x_{i+1})$ that accepts L as follows. On input $(\omega; x_1, \ldots, x_i, x_{i+1})$, N runs M on $(\omega; x_1, \ldots, x_i)$. N accepts ω if and only if $M(\omega; x_1, \ldots, x_i) = 1$. As M runs in polynomial time, so does N. The result follows. □

3.4.2 Π_i^p

We now turn our attention to generalizing coNP. Note that the quantified expression for coNP is obtained by negating the quantifiers for NP. We define the complexity class $\Pi_i^p := \mathsf{co}\Sigma_i^p$. That is, we negate the quantified expression for Σ_i^p to obtain a similar definition regarding alternating quantifiers. However, we begin with a universal quantifier rather than an existential quantifier. This definition is formalized as follows.

Definition 121. We say that the language $L \in \Pi_i^p$ if there exists a polynomial $p(\cdot)$ depending only on L and a verifier V such that if $\omega \in L$

$$\omega \in L \iff \forall x_1, \exists x_2, \forall x_3, \exists x_4, \ldots, Q_i x_i, V(\omega, x_1, \ldots, x_i) = 0,$$

$|x_j| \leq p(|\omega|)$ for all $1 \leq j \leq i$, and V runs in time $O(p(|\omega|))$. Note that Q_i indicates a quantifier. In particular, Q_i is an existential quantifier if i is even and a universal quantifier if i is odd.

So for Π_3^p, the abbreviated quantified expression is:

$$\omega \in L \iff \forall x_1, \exists x_2, \forall x_3, V(\omega, x_1, x_2, x_3) = 0.$$

Similarly, for Π_4^p, the abbreviated quantified expression is:

$$\omega \in L \iff \forall x_1, \exists x_2, \forall x_3, \exists x_4, V(\omega, x_1, x_2, x_3) = 0.$$

We now establish the following relations amongst the classes of the polynomial time heirarchy.

Theorem 122. Show that $\Pi_i^p \subseteq \Pi_{i+1}^p$.

Proof. Exercise □

Theorem 123. Show that $\Sigma_i^p \subseteq \Pi_{i+1}^p$.

Proof. Exercise. □

Theorem 124. Show that $\Pi_i^p \subseteq \Sigma_{i+1}^p$.

Proof. Exercise. □

As a final note, we define the Polynomial-Time Heirarchy formally:

Definition 125. The *Polynomial-Time Heirarchy*, denoted PH, is:

$$\mathsf{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i^p.$$

Remark 126. It is conjectured that $\Sigma_i^p \neq \Pi_i^p$ for each $i \geq 1$. In particular, this conjecture implies that Σ_i^p and Π_i^p are both strictly contained in Σ_{i+1}^p and Π_{i+1}^p.

3.4.3 Exercises

(Recommended) Problem 42. Prove Theorem 122. Show that $\Pi_i^p \subseteq \Pi_{i+1}^p$.

(Recommended) Problem 43. Prove Theorem 123. Show that $\Sigma_i^p \subseteq \Pi_{i+1}^p$.

(Recommended) Problem 44. Prove Theorem 124. Show that $\Pi_i^p \subseteq \Sigma_{i+1}^p$.

3.5 Structure of Polynomial-Time Hierarchy

It is conjectured that $\Sigma_i^p \neq \Pi_i^p$ for each $i \geq 1$. In particular, this conjecture implies that Σ_i^p and Π_i^p are both strictly contained in Σ_{i+1}^p and Π_{i+1}^p. In this section, we examine conditions under which PH collapses, as well as consequences thereof.

Theorem 127. Suppose that for some $i \geq 1$, we have that $\Sigma_i^p = \Pi_i^p$. Then $\mathsf{PH} = \Sigma_i^p$.

In order to prove Theorem 127, we first show that if $\Sigma_i^p = \Pi_i^p$, then $\Sigma_i^p = \Sigma_{i+1}^p = \Pi_{i+1}^p$. Theorem 127 then follows by induction.

Proposition 128. Suppose that for some $i \geq 1$, we have that $\Sigma_i^p = \Pi_i^p$. Then $\Sigma_i^p = \Sigma_{i+1}^p$.

Proof. By Theorem 120, we have that $\Sigma_i^p \subseteq \Sigma_{i+1}^p$. We show that $\Sigma_{i+1}^p \subseteq \Sigma_i^p$. Let $L \in \Sigma_{i+1}^p$. So there exists a Σ_{i+1}^p Turing Machine $M(\omega; x_1, \ldots, x_{i+1})$ that accepts L. Suppose that $\omega \in L$. Then the following relation holds:

$$\exists x_1, \forall x_2, \ldots, Q x_{i+1}\, M(\omega; x_1, \ldots, x_{i+1}) = 1.$$

Define:

$$L' = \{\langle \omega, x_1 \rangle : \forall x_2, \ldots, Q x_{i+1}\, M(\omega; x_1, \ldots, x_{i+1}) = 1\}.$$

So M is a Π_i^p Turing Machine that accepts L'. Note that $L \in \Sigma_{i+1}^p$ if and only if $L' \in \Pi_i^p$. As $\Pi_i^p = \Sigma_i^p$, there exists a Σ_i^p Turing Machine $N(\langle \omega, x_1; x_2, \ldots, x_{i+1})$ that accepts L'. We define a Σ_i^p Turing Machine $N(\omega; \langle x_1, x_2 \rangle, \ldots, x_{i+1})$ that accepts L as follows. On input $(\omega; \langle x_1, x_2 \rangle, x_3, \ldots, x_{i+1})$, N' simulates $N(\langle \omega, x_1 \rangle; x_2, \ldots, x_{i+1})$. Now N' accepts ω if and only if N accepts ω. So $L \in \Sigma_i^p$, as desired. □

It remains to show that if $\Sigma_i^p = \Pi_i^p$, then $\Sigma_i^p = \Pi_{i+1}^p$. The key idea here is to use the fact that $\Sigma_i^p = \Pi_i^p = \Sigma_{i+1}^p$, by the previous proposition, as well as the fact that $\Pi_{i+1}^p = \mathsf{co}\Sigma_{i+1}^p$. We leave this as an exercise.

Proposition 129. Suppose that for some $i \geq 1$, we have that $\Sigma_i^p = \Pi_i^p$. Then $\Sigma_i^p = \Pi_{i+1}^p$.

We similarly obtain that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{PH} = \mathsf{P}$. The key observation is that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{NP} = \mathsf{coNP}$. Recall that $\Sigma_1^p = \mathsf{NP}$ and $\Pi_1^p = \mathsf{coNP}$. Theorem 127 now applies immediately. We leave it as an exercise to fill in the details.

Theorem 130. Suppose that $\mathsf{P} = \mathsf{NP}$. Then $\mathsf{PH} = \mathsf{P}$.

We next discuss the relationship between PH and PSPACE. We first observe that if PH has a complete problem, then PH collapses to some level i.

Theorem 131. Suppose that PH has a complete problem under polynomial-time reductions. Then $\mathsf{PH} = \Sigma_i^p$ for some $i \geq 1$.

As PSPACE has complete problems under polynomial-time reductions, we note that if $\mathsf{PH} = \mathsf{PSPACE}$, then PH collapses.

Remark 132. While PH is unlikely to have complete problems, both Σ_i^p and Π_i^p have complete problems. Establishing these complete problems is quite involved, similar to proving the Cook-Levin Theorem. We omit the proofs for this reason.

Definition 133 ($\Sigma_i^p\mathsf{SAT}$)**.**

- Instance: Let φ be a Boolean formula.
- Decision: Does the following relation hold?

$$\exists x_1\, \forall x_2, \ldots, Q x_i\, \varphi(x_1, \ldots, x_i) = 1,$$

  where $x_1, \ldots, x_k \in \{0, 1\}$.

Definition 134 ($\Pi_i^p\mathsf{SAT}$)**.**

- Instance: Let φ be a Boolean formula.
- Decision: Does the following relation hold?

$$\forall x_1\, \exists x_2, \ldots, Q x_i\, \varphi(x_1, \ldots, x_i) = 1,$$

  where $x_1, \ldots, x_k \in \{0, 1\}$.

Theorem 135. For each $i \geq 1$, $\Sigma_i^p\mathsf{SAT}$ is Σ_i^p-complete, and $\Pi_i^p\mathsf{SAT}$ is Π_i^p-complete.

3.5.1 Exercises

(Recommended) Problem 45. Prove Proposition 129. Suppose that for some $i \geq 1$, we have that $\Sigma_i^p = \Pi_i^p$. Then $\Sigma_i^p = \Pi_{i+1}^p$.

(Recommended) Problem 46. The goal of this problem is to prove Theorem 130. That is, if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{PH} = \mathsf{P}$.

(a) Show that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{NP} = \mathsf{coNP}$.

(b) Apply Theorem 127 to show that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{P} = \mathsf{PH}$.

(Recommended) Problem 47. The goal of this problem is to show that it is unlikely that $\mathsf{PH} = \mathsf{PSPACE}$.

(a) Prove Theorem 131. Suppose that PH has a complete problem under polynomial-time reductions. Then $\mathsf{PH} = \Sigma_i^p$ for some $i \geq 1$.

(b) Show that if PH does not collapse, then $\mathsf{PH} \neq \mathsf{PSPACE}$.

(Recommended) Problem 48. Show that $\mathsf{PH} \subseteq \mathsf{PSPACE}$ in two ways.

(a) First, give a simulation argument. Show how to design a PSPACE algorithm for $\Sigma_i^p\mathsf{SAT}$ (or if you prefer, for $\Pi_i^p\mathsf{SAT}$).

(b) Second, reduce to TQBF. [**Hint:** Observe that $\Sigma_i^p\mathsf{SAT}$ and $\Pi_i^p\mathsf{SAT}$ instances are also instances of TQBF.]

3.6 Polynomial-Time Hierarchy and Oracles

We may alternatively define the complexity classes in PH in terms of oracles. This is more in the spirit of the Arithmetic Hierarchy. Namely, denote $\Sigma_0^p = \Pi_0^p = \mathsf{P}$. For $i \geq 1$, we have that $\Sigma_i^p = \mathsf{NP}^{\Sigma_{i-1}^p}$ and $\Pi_i^p = \mathsf{coNP}^{\Pi_{i-1}^p}$. We prove the former relation here. The latter falls out as a corollary, which we leave as an exercise.

Theorem 136. For each $i \geq 1$, we have that $\Sigma_i^p = \mathsf{NP}^{\Sigma_{i-1}^p}$.

Proof. The proof is by induction on i. When $i = 1$, we have that:

$$\begin{aligned}\Sigma_1^p &= \mathsf{NP}^{\Sigma_0^p} \\ &= \mathsf{NP}^{\mathsf{P}} \\ &= \mathsf{NP}.\end{aligned}$$

Fix $i \geq 1$, and suppose that $\Sigma_i^p = \mathsf{NP}^{\Sigma_{i-1}^p}$. We now show that $\Sigma_{i+1}^p = \mathsf{NP}^{\Sigma_i^p}$.

Lemma 137. We have that $\Sigma_{i+1}^p \subseteq \mathsf{NP}^{\Sigma_i^p}$.

Proof. Exercise. □

Lemma 138. We have that $\mathsf{NP}^{\Sigma_i^p} \subseteq \Sigma_{i+1}^p$.

Proof. Let $L \in \mathsf{NP}^{\Sigma_i^p}$, and let $M^{\Sigma_i^p}$ be the corresponding Oracle TM. Let $\omega \in L$. Suppose that y is a corresponding certificate, and that $M^{\Sigma_i^p}$ makes m queries: $(q_1, a_1), \ldots, (q_m, a_m)$, where the q_i are the queries and the a_i are the answers. In particular, the following are equivalent:

(a) $\omega \in L$.

(b) There exists a certificate y such that $M^{\Sigma_i^p}(\omega, y) = 1$.

(c) There exist queries $(q_1, a_1), \ldots, (q_m, a_m)$, where both m and the (q_j, a_j) pairs depend on y, such that:

$$\bigwedge_{j=1}^{m} M_j(q_j) = a_j,$$

for Σ_i^p Turing Machines $M_1, \ldots, M_m$.

We note that if $M_j(q_j) = 1$, then $q_j \in L(M_j)$. In particular, as M_j is a Σ_i^p Turing Machine, we have by the inductive hypothesis that:

$$\exists y_{j,1}, \forall y_{j,2}, \ldots, Q y_{j,i}\, R_j(q_j, y_{j,1}, \ldots, y_{j,i}) = 1, \tag{13}$$

where R_j is the Σ_i^p relation corresponding to $L(M_j)$. Now if $M_j(q_j) = 0$, then $q_j \notin L(M_j)$. Now $\overline{L(M_j)} \in \Pi_i^p$. It follows that:

$$\forall y'_{j,2}, \exists y'_{j,3}, \ldots, Q y'_{j,i+1}\, R_j(q_j, y'_{j,2}, \ldots, y'_{j,k+1}) = 0. \tag{14}$$

Combining (13) and (14), we have that $M_j(q_j) = a_j$ if and only if:

$$\begin{aligned}&\exists y_{j,1}, \forall (y_{j,2}, y'_{j,2}), \ldots, Q_i(y_{j,i}, y'_{j,i}), Q_{k+1}(y'_{j,i+1}) \\ &[a_j = 1 \wedge R_j(q_j, y_{j,1}, \ldots, y_{j,i}) = 1] \vee [a_j = 0 \wedge R_j(q_j, y'_{j,2}, \ldots, y'_{j,i+1}) = 0].\end{aligned}$$

Our strategy now is to collect the $y_{j,k}$ and $y'_{j,k}$ strings into a Σ_{i+1}^p formula. We have that $M(\omega) = 1$ if and only if:

$$\exists q_1, \ldots, q_m, y, y_{1,1}, \ldots, y_{m,1}, \tag{15}$$
$$\forall y_{1,2}, \ldots, y_{m,2}, y'_{1,2}, \ldots, y'_{m,2}, \tag{16}$$
$$\exists y_{1,3}, \ldots, y_{m,3}, y'_{1,3}, \ldots, y'_{m,3}, \tag{17}$$
$$\vdots \tag{18}$$
$$Q_i\,(y_{1,i}, \ldots, y_{m,i}, y'_{1,i}, \ldots, y'_{m,i}), \tag{19}$$
$$Q_{i+1}\,(y'_{1,i+1}, \ldots, y'_{m,i+1}) \tag{20}$$
$$\bigwedge_{j=1}^{m} [a_j = 1 \wedge R_j(q_j, y_{j,1}, \ldots, y_{j,i}) = 1] \vee [a_j = 0 \wedge R_j(q_j, y'_{j,2}, \ldots, y'_{j,i+1}) = 0]. \tag{21}$$

In particular, (21) is our Σ_{i+1}^p formula for L. □

□

3.6.1 Exercises

(Recommended) Problem 49. Here, we prove Lemma 137. Let $i \geq 1$, and suppose that $\Sigma_i^p = \mathsf{NP}^{\Sigma_{i-1}^p}$. We show that $\Sigma_{i+1}^p \subseteq \mathsf{NP}^{\Sigma_i^p}$. Let $L \in \Sigma_{i+1}^p$, and let M be the Σ_i^p Turing Machine that accepts L. Suppose that $\omega \in L$. We have that:

$$\exists x_1, \forall x_2, \ldots, Q x_{i+1}\, M(\omega; x_1, \ldots, x_{i+1}) = 1,$$

where the x_i are all of length at most poly($|\omega|$). Define:

$$L' = \{\langle \omega, x_1 \rangle : \forall x_2, \ldots, Q x_{i+1}\, M(\omega; x_1, \ldots, x_{i+1}) = 1\}.$$

Do the following.

(a) Show that $\langle \omega, x_1 \rangle \not\in L'$ if and only if

$$\exists x_2, \ldots, \overline{Q} x_{i+1}\, M(\omega; x_1, \ldots, x_{i+1}) = 0.$$

(b) Show that $\overline{L'} \in \Sigma_i^p$.

(c) Design an $\mathsf{NP}^{\Sigma_i^p}$ Turing Machine to accept L.

(d) Conclude that $\Sigma_{i+1}^p \subseteq \mathsf{NP}^{\Sigma_i^p}$.

(Recommended) Problem 50. We showed that for $i \geq 0$, $\Sigma_{i+1}^p = \mathsf{NP}^{\Sigma_i^p}$. By definition, $\Pi_{i+1}^p = \mathsf{co}\Sigma_{i+1}^p$. So $\Pi_{i+1}^p = \mathsf{coNP}^{\Sigma_i^p}$. Show that: $\Pi_{i+1}^p = \mathsf{coNP}^{\Pi_i^p}$.

3.7 Time Hierarchy Theorem

It is often easy to show that complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$ satisfy $\mathcal{C}_1 \subseteq \mathcal{C}_2$. Determining whether $\mathcal{C}_1 = \mathcal{C}_2$ is significantly harder. For instance, it is quite easy to show that $\mathsf{P} \subseteq \mathsf{NP}$. However, determining whether $\mathsf{P} = \mathsf{NP}$ is the central open problem in Theoretical Computer Science. In this section, we explore tools which allow us to separate certain complexity classes. We begin with the Time Hierarchy Theorem.

Definition 139. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if $f(n) \geq n$ for all $n \in \mathbb{N}$. Similarly, we say that f is *space-constructible* if $f(n) \geq \log(n)$ for all $n \in \mathbb{N}$.

Remark 140. When dealing with time-complexity, we often assume that our runtime functions are time-constructible to ensure that the Turing Machines can read the entire inputs. We similarly assume that our space-complexity functions are space-constructible.

Before introducing the Time Hierarchy Theorem, we introduce the following fact.

Theorem 141. Let M be a k-tape Turing Machine. If M takes T steps to process a string ω, then a two-tape Turing Machine can simulate M in $O(T \log(T))$ steps.

Theorem 142 (Time Hierarchy Theorem)**.** Suppose that $f, g : \mathbb{N} \to \mathbb{N}$ be time-constructible functions such that $f(n)\log(f(n)) \in o(g(n))$. Then $\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n))$.

Proof. Clearly, $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n))$. We use diagonalization to construct a language $L \in \mathsf{DTIME}(g(n))$ such that $L \notin \mathsf{DTIME}(f(n))$. We first recall that Turing Machines can be encoded as strings. Let $(w_n)_{n\in\mathbb{N}}$ be an enumeration of deterministic Turing Machine encodings, where each M_n is given by its corresponding encoding $\omega_n := \langle M_n \rangle$. We may also encode our Turing Machines in such a way as to allow for redundancy. Namely, for each Turing Machine M and each $k \in \mathbb{N}$, we may have the encoding $\langle M \rangle 1^k$. Using such an encoding scheme, any given Turing Machine will appear infinitely many times in our enumeration.

On input ω_n, we simulate M_n on input ω_n. We note that M_n may have more than two tapes. By Theorem 141, we may simulate M_n using only two tapes with a multiplicative overhead of $O(\log(n))$. We also note that there is a cost of converting the symbols used by M_n into a fixed, standard alphabet. So each step of M_n can be simulated in time $c \cdot T(n)\log(T(n))$, where $T(n)$ is the runtime complexity function for M_n. In particular, if $T(n) = f(n)$, then M_n can be simulated in time $O(f(n)\log(f(n)))$.

Now define:

$$L = \{\omega_n : M_n(\omega_n) \text{ halts after } g(|\omega_n|) \text{ steps and } M_n \text{ rejects } \omega_n\}.$$

We claim that $L \in \mathsf{DTIME}(g(n))$ and $L \notin \mathsf{DTIME}(f(n))$.

Lemma 143. $L \in \mathsf{DTIME}(g(n))$.

Proof. On a given string ω, we may check whether ω is a valid Turing Machine encoding; and if so, simulate the corresponding Turing Machine M_ω for at most $g(|\omega|)$ steps. We accept ω if and only if M_ω halts and rejects ω. Otherwise, we reject ω. □

Lemma 144. $L \notin \mathsf{DTIME}(f(n))$.

Proof. Suppose to the contrary that $L \in \mathsf{DTIME}(f(n))$. Then there exists a deterministic Turing Machine M such that $L(M) = L$ and M runs in time $O(f(n))$. Let ω be an encoding of M such that $f(|\omega|)\log(f(|\omega|)) < g(|\omega|)$. We analyze whether $\omega \in M$ to obtain our contradiction. We leave the details as an exercise. □

□

3.7.1 Exercises

(Recommended) Problem 51. Complete the proof of Lemma 144. Let ω be an encoding of M such that $f(|\omega|)\log(f(|\omega|) < g(|\omega|)$ (such an encoding exists as $f(n)\log(f(n)) \in o(g(n))$), the $\mathsf{DTIME}(f(n))$ Turing Machine that accepts L. We ask whether $\omega \in L$.

(a) Suppose that $\omega \in L$. As M accepts L, we have that M accepts ω in $f(|\omega|)$ steps. Using the definition of L, deduce that $\omega \notin L$ to obtain a contradiction.

(b) Suppose instead that $\omega \notin L$. Using a similar argument as part (a), deduce that $\omega \in L$ to obtain a contradiction.

(c) Conclude that $L \notin \mathsf{DTIME}(f(n))$.

(Recommended) Problem 52. Define:

$$\mathsf{EXPTIME} := \bigcup_{c \in \mathbb{N}} \mathsf{DTIME}(2^{n^c}).$$

Do the following.

(a) Show that $\mathsf{P} \subseteq \mathsf{DTIME}(n^{\log(n)})$.

(b) Fix $c \geq 1$. Show that $\mathsf{DTIME}(n^{\log(n)}) \subsetneq \mathsf{DTIME}(2^{n^c})$.

(c) Deduce that $\mathsf{P} \neq \mathsf{EXPTIME}$.

Remark 145. We note that $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$. In light of Problem 52, we note that one of these containments is strict, though we don't know which one. It is believed that all of these containments are strict.

(Recommended) Problem 53. Let $\epsilon > 0$. Show that $\mathsf{DTIME}(n^k) \subsetneq \mathsf{DTIME}(n^{k+\epsilon})$. That is, there are problems in P that require arbitrarily large exponents to solve.

(Recommended) Problem 54. In this problem, we prove the Space Hierarchy Theorem, which states that if $f, g : \mathbb{N} \to \mathbb{N}$ are space-constructible functions satisfying $f(n) \in o(g(n))$, then $\mathsf{DSPACE}(f(n)) \subsetneq \mathsf{DSPACE}(g(n))$. As in the proof of the Time Hierarchy Theorem, let $(w_n)_{n \in \mathbb{N}}$ be an enumeration of deterministic Turing Machine encodings, where each M_n is given by its corresponding encoding $\omega_n := \langle M_n \rangle$. Note that Turing Machine encodings are not unique. It is in fact the case that any given Turing Machine will appear infinitely many times in our enumeration. Define:

$$L = \{\omega_n : M_n(\omega_n) \text{ halts and rejects } \omega_n \text{ using space } g(|\omega_n|) \text{ and taking at most } 2^{g(n)} \text{ steps.}\}.$$

Do the following.

(a) Why do we clock the execution of $M_n(\omega_n)$ to $2^{g(n)}$ steps?

(b) Show that $L \in \mathsf{DSPACE}(g(n))$.

(c) We now show that $L \notin \mathsf{DSPACE}(f(n))$. Suppose to the contrary that there exists a $\mathsf{DSPACE}(f(n))$ Turing Machine M that decides L. Let ω be an encoding of M such that $f(|\omega|) < g(|\omega|)$ (such an encoding exists as $f(n) \in o(g(n))$).

(i) Suppose that $\omega \in L$. As M accepts L, we have that M accepts ω using at most $g(|\omega|)$ space and taking at most $2^{g(|\omega|)}$ steps. Using the definition of L, deduce that $\omega \notin L$ to obtain a contradiction.

(ii) Suppose that $\omega \notin L$. Using a similar argument as in part (a), deduce that $\omega \in L$ to obtain a contradiction.

(iii) Conclude that $L \notin \mathsf{DSPACE}(f(n))$.

(Recommended) Problem 55. Using the Space Hierarchy Theorem (Problem 54), show that $\mathsf{L} \subsetneq \mathsf{DSPACE}(n)$. Conclude that $\mathsf{L} \subsetneq \mathsf{PSPACE}$.

Remark 146. We have that:

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \ldots \subseteq \mathsf{NC} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}.$$

In light of Problem 55, we have that one of these containments is strict. It is widely believed that all of these containments are strict.

References

[AB09] Sanjeev Arora and Boaz Barak, *Computational complexity: A modern approach*, 1st ed., Cambridge University Press, USA, 2009.

[AKS02] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, *Primes is in p*, Ann. of Math **2** (2002), 781–793.

[Bab15] László Babai, *Graph isomorphism in quasipolynomial time*, CoRR **abs/1512.03547** (2015).

[BE99] Hans Ulrich Besche and Bettina Eick, *Construction of finite groups*, J. Symb. Comput. **27** (1999), no. 4, 387–404.

[BEO02] Hans Ulrich Besche, Bettina Eick, and E.A. O'Brien, *A millennium project: Constructing small groups*, Intern. J. Alg. and Comput **12** (2002), 623–644.

[CH03] John J. Cannon and Derek F. Holt, *Automorphism group computation and isomorphism testing in finite groups*, J. Symb. Comput. **35** (2003), 241–267.

[ELGO02] Bettina Eick, C. R. Leedham-Green, and E. A. O'Brien, *Constructing automorphism groups of p-groups*, Comm. Algebra **30** (2002), no. 5, 2271–2295. MR 1904637

[Kat11] Jonathan Katz, *Notes on complexity theory- lecture 7*, Sep 2011.

[Kha80] L. Khachiyan, *Polynomial algorithms in linear programming*, Ussr Computational Mathematics and Mathematical Physics **20** (1980), 53–72.

[KKL13] Swastick Koppartiy, John Kim, and Ben Lund, *Lecture 1: Course overview, circuits, and formulas*, 2013.

[Lad75] Richard E. Ladner, *On the structure of polynomial time reducibility*, J. ACM **22** (1975), no. 1, 155–171.

[Lev18] Michael Levet, *Fundamentals of computer science lecture notes*, 2018.

[Lev20] Michael Levet, *Theory of computation- lecture notes.*

[LGR16] François Le Gall and David J. Rosenbaum, *On the group and color isomorphism problems*, arXiv:1609.08253, 2016.

[Mil78] Gary L. Miller, *On the nlog n isomorphism technique (a preliminary report)*, Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '78, Association for Computing Machinery, 1978, p. 51–58.

[Ros12] Kenneth Rosen, *Discrete mathematics and its applications*, 7 ed., 2012.

[Ros13] David J. Rosenbaum, *Bidirectional collision detection and faster deterministic isomorphism testing*, ArXiv **abs/1304.3935** (2013).

[Sav97] John E. Savage, *Models of computation: Exploring the power of computing*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[Sip96] Michael Sipser, *Introduction to the theory of computation*, 1st ed., International Thomson Publishing, 1996.

[Wil19] James B. Wilson, *The threshold for subgroup profiles to agree is logarithmic*, Theory of Computing **15** (2019), no. 19, 1–25.